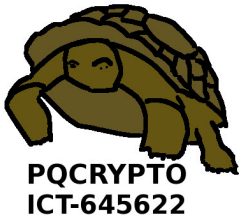


PQCRYPTO

Post-Quantum Cryptography for Long-Term Security

Project number: Horizon 2020 ICT-645622

Internet: Portfolio

Due date of deliverable: 1. March 2018
Actual submission date: 12. April 2018

Start date of project: 1. March 2015 Duration: 3 years

Coordinator:
Technische Universiteit Eindhoven
Email: coordinator@pqcrypto.eu.org
www.pqcrypto.eu.org

Revision 0.0

Project co-funded by the European Commission within Horizon 2020		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Internet: Portfolio

Wouter Castryck
(Including documentation written by many authors.)

12. April 2018
Revision 0.0

The work described in this report has in part been supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO.

Abstract

This portfolio compiles the specifications of the post-quantum cryptosystems which, according to the members of the PQCRYPTO consortium, are the most promising alternatives for all currently deployed digital signature, key establishment and public-key encryption schemes (corresponding to the three main cryptographic tasks of secure internet communication). Our document covers 20 of the 22 proposals to the NIST standardization project in which PQCRYPTO is involved.

Keywords: Post-quantum cryptography, internet, digital signatures, key establishment, public-key encryption

1 Introduction

In this portfolio we list the cryptographic systems that were judged, by the members of the PQCRYPTO consortium, to be the most promising candidates for performing the central tasks in secure internet communication (namely digital signatures, key establishment, and to a lesser extent public-key encryption) in a post-quantum context. Currently, pre-quantum systems performing these tasks are incorporated in ubiquitous internet protocols like TLS and SSH, which are invoked multiple billions of times each day. Replacing these systems by post-quantum versions is central to the successful deployment of post-quantum cryptography on the internet. For each system we provide many concrete parameter sets, give detailed security analyses, discuss pros and cons, and analyze the runtimes of a.o. our reference C implementations (which are part of our library `libpqcrypto`, discussed in D2.4).

All our proposals were submitted to the ongoing NIST standardization project on post-quantum cryptography,[1] whose submission deadline was 30 November 2017 and which will run for at most five years. Concretely, the current document covers 20 of the 22 PQCRYPTO submissions, out of the 69 admissible submissions that NIST received in total. More precisely, this portfolio is a compilation of the respective system specifications of these 20 proposals, of which we give a concise overview in the next section; we stress that this also includes work by our many collaborators world-wide. For the complete list of all 69 submissions, we refer to the website https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions, which also discusses the current status of each competitor. We note that PQCRYPTO plays an active role in the assessment of the other proposals, which already led to complete breaks of Edon-K, HK7, RVB, SRTPI, WalnutDSA, as well as to the discovery of a few other vulnerabilities. The PQCRYPTO submissions stand firm to date.

2 Overview

An overview of the 20 proposals is given below, along with the page number where the corresponding specifications can be found. The names of the PQCRYPTO contributors are printed in bold.

For the sake of comparison, we have included the targeted functionality (digital signatures, key encapsulation, or public-key encryption) and the area of the underlying hard mathematical problem (codes, lattices, multivariate systems of equations, or hash functions). In the latter case this can be ambiguous, so we stress that this is included for indicative purposes only and we refer to the specifications for more details. The same concern applies to the stated public (pk), secret (sk), ciphertext (ctxt) and signature (sig) sizes in kilobytes (kB): most submissions propose several parameter sets, and the table only samples one from these. Where possible we have selected a parameter set that targets NIST's security category 5, which means that breaking the said scheme should amount to an effort equivalent to breaking AES-256. But apart from security considerations (and how aggressively one plays this game), the space requirements are also affected by trade-offs between key and ciphertext/signature sizes, speed, failure rates, etc. Again we refer to the respective specifications for more details.

[1] https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/

Name	Functionality and sizes (at NIST security cat. 5)	Underlying mathematics	Contributors (PQCRYPTO members in **bold**)	pp.
BIG QUAKE	Key Encapsulation pk: 146.29 kB sk: 40.82 kB ctxt: 0.48 kB (for `BIG_QUAKE_5`)	Codes (QC Goppa)	Magali Bardet **Élise Barelli** Olivier Blazy **Rodolfo Canto-Torres** **Alain Couvreur** Philippe Gaborit Ayoub Otmani **Nicolas Sendrier** **Jean-Pierre Tillich**	6
BIKE (BIKE-1, BIKE-2, BIKE-3)	Key Encapsulation pk: 4.00 kB sk: 0.54 kB ctxt: 4.00 kB (for `BIKE-2 level 5`)	Codes (QC-MDPC)	Nicolas Aragon Paulo S. L. M. Barreto Slim Bettaieb Loïc Bidoux Olivier Blazy Philippe Gaborit Jean-Christophe Deneuville Philippe Gaborit **Shay Gueron** **Tim Güneysu** Carlos Aguilar Melchor Edoardo Persichetti **Nicolas Sendrier** **Jean-Pierre Tillich** Gilles Zémor	38
Classic McEliece	Key Encapsulation pk: 1022.78 kB sk: 13.58 kB ctxt: 0.22 kB (for `mceliece6960119`)	Codes (Goppa)	**Daniel J. Bernstein** Tung Chou **Tanja Lange** Ingo von Maurich Rafael Misoczki Ruben Niederhagen Edoardo Persichetti Christiane Peters **Peter Schwabe** **Nicolas Sendrier** Jakub Szefer Wen Wang	90
CRYSTALS- Dilithium	Digital Signatures pk: 1.72 kB sk: 3.77 kB sig: 3.29 kB (for `Dilithium-IV`, security cat. 3)	Lattices (MLWE/MSIS)	Léo Ducas Eike Kiltz Tancrède Lepoint Vadim Lyubashevsky **Peter Schwabe** Gregor Seiler Damien Stehlé	128
CRYSTALS- Kyber	Key Encapsulation pk: 1.41 kB sk: 3.10 kB ctxt: 1.49 kB (for `Kyber1024`)	Lattices (MLWE)	Roberto Avanzi **Joppe W. Bos** Léo Ducas Eike Kiltz Tancrède Lepoint Vadim Lyubashevsky	158

			John M. Schanck **Peter Schwabe** Gregor Seiler Damien Stehlé	
DAGS	Key Encapsulation pk: 11.34 kB sk: 2178.00 kB ctxt: 1.58 kB (for `DAGS_5`)	Codes (Dyadic GS)	**Gustavo Banegas** Paulo S. L. M. Barreto Brice Odilon Boidje Pierre-Louis Cayrel Gilbert Ndollane Dione Kris Gaj Cheikh Thiécoumba Gueye Richard Haeussler Jean Belo Klamti Ousmane Ndiaye Duc Tri Nguyen Edoardo Persichetti Jefferson E. Ricardini	190
FrodoKEM	Key Encapsulation pk: 15.27 kB sk: 30.54 kB ctxt: 15.40 kB (for `Frodo-976`, security cat. 3)	Lattices (LWE)	Michael Naehrig Erdem Alkim **Joppe W. Bos** Léo Ducas Karen Easterbrook Brian LaMacchia Patrick Longa Ilya Mironov Valeria Nikolaenko Christopher Peikert Ananth Raghunathan Douglas Stebila	213
Gui	Digital Signatures pk: 5789.20 kB sk: 155.90 kB sig: 0.65 kB (for `Gui-448`)	Multivariate systems of equations	Jintai Ding Ming-Shen Chen Albrecht Petzoldt Dieter Schmidt **Bo-Yin Yang**	259
KINDI	PK Encryption and Key Encapsulation pk: 1.94 kB sk: 2.25 kB ctxt: 2.63 kB (for `KINDI-256-5-2-2`)	Lattices (MLWE)	**Rachid El Bansarkhani**	291
LUOV	Digital Signatures pk: 98.60 kB sk: 0.03 kB sig: 0.51 kB (for `LUOV-8-117-404`)	Multivariate systems of equations	**Ward Beullens** **Bart Preneel** **Alan Szepieniec** **Frederik Vercauteren**	315
MQDSS	Digital Signatures pk: 0.09 kB sk: 0.05 kB sig: 66.2 kB (for `MQDSS-31-64`, security cat. 4)	Multivariate systems of equations	**Ming-Shing Chen** **Andreas Hülsing** **Joost Rijneveld** Simona Samardjiska **Peter Schwabe**	348
NewHope	Key Encapsulation	Lattices	Thomas Poppelmann	436

	pk: 1.78 kB sk: 3.59 kB ctxt: 2.16 kB (for `NewHope1024-CCA-KEM`)	(RLWE)	Erdem Alkim Roberto Avanzi **Joppe W. Bos** Léo Ducas Antonio de la Piedra **Peter Schwabe** Douglas Stebila	
NTRU-HRSS-KEM	Key Encapsulation pk: 1.11 kB sk: 1.38 kB ctxt: 1.25 kB (for security cat. 1)	Lattices (NTRU)	John M. Schanck **Andreas Hülsing** **Joost Rijneveld** **Peter Schwabe**	480
NTRU Prime	Key Encapsulation pk: 1.19 kB sk: 1.56 kB ctxt: 1.02 kB (for `sntrup4591761`)	Lattices (NTRU)	**Daniel J. Bernstein** **Chitchanok Chuengsatiansup** **Tanja Lange** **Christine van Vrenendaal**	506
Picnic	Digital Signatures pk: 0.06 kB sk: 0.03 kB sig: 204.56 kB (for `picnic-L5-UR`)	Hash functions (SHA family)	Greg Zaverucha Melissa Chase David Derler Steven Goldfeder Claudio Orlandi Sebastian Ramacher **Christian Rechberger** Daniel Slamanig	546
qTESLA	Digital Signatures pk: 8.03 kB sk: 8.06 kB sig: 6.03 kB (for `qTESLA-256`)	Lattices (RLWE)	Sedat Akleylek Erdem Alkim Paulo S. L. M. Barreto **Johannes Buchmann** Edward Eaton Gus Gutoski **Juliane Krämer** Patrick Longa **Harun Polat** Jefferson E. Ricardini Gustavo Zanon	568
Rainbow	Digital Signatures pk: 1683.30 kB sk: 1244.40 kB sig: 0.20 kB (for `Rainbow Vc`)	Multivariate systems of equations	Jintai Ding Ming-Shing Chen Albrecht Petzoldt Dieter Schmidt **Bo-Yin Yang**	599
Ramstake	Key Encapsulation pk: 92.42 kB sk: 184.81 kB ctxt: 93.86 kB (for `Ramstake RS 756839`)	Lattices (sparse integers)	**Alan Szepieniec**	634
SABER	PK Encryption and Key Encapsulation pk: 1.27 kB sk: 0.38 kB (`PKE`) 1.72 kB (`KEM`)	Lattices (MLWR)	**Jan-Pieter D'Anvers** **Angshuman Karmakar** **Sujoy Sinha Roy** **Frederik Vercauteren**	654

	ctxt: 1.44 kB (for `FireSaber-PKE` resp. `FireSaber-KEM`)			
SPHINCS+	Digital Signatures pk: 0.06 kB sk: 0.13 kB sig: 29.09 kB (for `SPHINCS`$^+$`-256s`)	Hash functions (SHA family)	**Daniel J. Bernstein** Christoph Dobraunig Maria Eichlseder Scott Fluhrer Stefan-Lukas Gazdag **Andreas Hülsing** Panos Kampanakis **Stefan Kölbl** **Tanja Lange** Martin M. Lauridsen Florian Mendel Ruben Niederhagen **Christian Rechberger** **Joost Rijneveld** **Peter Schwabe**	685

BIG QUAKE

BInary **G**oppa **QUA**si–cyclic **K**ey **E**ncapsulation

Magali Bardet, University of Rouen, France
Élise Barelli, Inria & École Polytechnique, France
Olivier Blazy, University of Rouen, France
Rodolfo Canto–Torres, Inria, France
Alain Couvreur, Inria & École Polytechnique, France
Philippe Gaborit, University of Limoges, France
Ayoub Otmani, University of Rouen, France
Nicolas Sendrier, Inria, France
Jean-Pierre Tillich, Inria, France

Principal submitter: Alain Couvreur.

Auxiliary submitters: Listed above.

Inventors/Developers: Same as the submitters. Relevant prior work is credited where appropriate.

Implementation Owners: Submitters.

Email Address (preferred): alain.couvreur@inria.fr

Postal Address and Telephone (if absolutely necessary):
Alain Couvreur, LIX, École Polytechnique 91128 Palaiseau Cédex, +33 1 74 85 42 66 .

Signature: x. See also printed version of "Statement by Each Submitter".

Abstract

The present proposal is a key encapsulation scheme based on a Niederreiter–like public key encryption scheme using binary quasi–cyclic Goppa codes.

Contents

1 Introduction

1.1 Motivation for this proposal

The original McEliece system [McE78] is the oldest public key cryptosystem which is still resistant to classical and quantum computers. It is based on binary Goppa codes. Up to now, all known attacks on the scheme have at least exponential complexity. A security proof is given in [Sen10] which relies on two assumptions (i) the hardness of decoding a generic linear code and (ii) distinguishing a Goppa code from a random linear code. It is well known to provide extremely fast encryption and fast decryption [BCS13], but has large public keys, about 200 kilobytes for 128 bits of security and slightly less than one megabyte for 256 bits of security [BLP08].

The aim of this proposal is to propose a key-encapsulation scheme based on binary Goppa codes by reducing the key size by a moderate factor ℓ in the range [3..19]. This is obtained by using binary quasi–cyclic Goppa codes of order ℓ instead of plain binary Goppa codes. The rationale behind this is that for the original McEliece cryptosystem key-recovery attacks have a much higher complexity than message recovery attacks. By focusing on quasi–cyclic Goppa codes, there is a security loss with respect to key recovery attacks but this loss is affordable due to the big gap between the complexity of key recovery attacks and message recovery attacks, and because the security loss with respect to message recovery attacks is negligible.

1.2 Quasi–cyclic codes in cryptography

This is not the first time that quasi–cyclic codes have been proposed in this context. The first proposal can be traced back to [Gab05] where quasi–cyclic subcodes of BCH codes are suggested. This proposal was broken in [OTD10], essentially because the number of possible keys was too low.

A second proposal based on quasi–cyclic alternant codes (a family of codes containing the Goppa code family) was made in [BCGO09]. Because of a too large order of quasi cyclicity, all the parameters of this proposal have been broken in [FOPT10]. Another proposal with quasi–dyadic and quasi–p–adic Goppa codes was given in [MB09, BLM11]. Some parameters have been broken in [FOPT10, FOP$^+$16b]. In both cases, the corresponding attacks are based on an algebraic modeling of the key recovery attack and using Groebner basis techniques to solve them. This can be done in this case because the quasi–cyclic/dyadic structure allows to drastically reduce the number of variables in the system when compared to the polynomial system associated to unstructured alternant or Goppa codes.

Later on [FOP$^+$16a] provided a further insight on these algebraic attacks by proving that in the case of quasi–cyclic alternant or Goppa codes of order ℓ it is possible to construct another alternant or Goppa code whose length is divided by ℓ without knowing the secret algebraic structure of the code. This code is called the *folded code* there. There is a strong relation between this code and the *invariant code* considered in [Bar17]. This explains why in the case of key-recovery attacks attacking quasi–cyclic alternant or Goppa codes we can reduce the problem to a key recovery of a much smaller code.

This sequence of proposals and subsequent attacks lead to the following observations. For a code based scheme using quasi–cyclic algebraic codes to be secure, the following requirements are fundamental:

1. The family of codes providing the keys should be large enough;
2. The security of the key must be studied in terms of the public key and all the smaller codes deriving from the public key (invariant code, folded code, see §4 for further details).
3. The cryptosystem should be resistant to attacks on the message that is generic decoding algorithms.

1.3 Type of proposal

We propose a public key encryption scheme (PKE) which is converted into a key encapsulation mechanism (KEM) using a generic transformation due to Hohheinz, Hövelmanns and Kiltz [HHK17] in order to get an INDCCA2 security. Our public key encryption scheme is a Niederreiter–like scheme. Compared to the original Niederreiter scheme, our proposal avoids the computation of a bijection between words of fixed length and constant weight words. This avoids cumbersome computations involving large integers and provides a light scheme more suitable for embedded system with restricted computing resources. The PKE is proved to be IND–CPA and the generic conversion described in [HHK17] leads to an IND–CCA2 KEM.

Acknowledgements

The submitters express their gratitude to Daniel Augot and Julien Lavauzelle for their helpful comments. Submitters are are supported by French ANR grant CBCrypt and by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO.

2 Goppa codes, QC Goppa codes

2.1 Context

In what follows, any finite field is an extension of the binary field $\mathbb{F}_2$.That is, any field is of the form $\mathbb{F}_{2^m}$ for some positive integer m.

2.2 Vectors, matrices

Vectors and matrices are respectively denoted in bold letters and bold capital letters such as $\boldsymbol{a}$ and $\boldsymbol{A}$. We always denote the entries of a vector $\boldsymbol{u} \in \mathbb{F}_q^n$ by $u_0, \ldots, u_{n-1}$.

2.3 Polynomials

Given a finite field $\mathbb{F}_{2^m}$ for some positive m, the ring of polynomials with coefficients in $\mathbb{F}_q$ is denoted by $\mathbb{F}_q[z]$, while the subspace of $\mathbb{F}_q[z]$ of polynomials of degree strictly less than t is denoted by $\mathbb{F}_q[z]_{<t}$. For every rational fraction $P \in \mathbb{F}_q(z)$, with no poles at the elements $u_0, \ldots, u_{n-1}$, $P(\boldsymbol{u})$ stands for $(P(u_0), \ldots, P(u_{n-1}))$. In particular for a vector $\boldsymbol{y} = (y_0, \ldots, y_{n-1})$ that has only nonzero entries, the vector $\boldsymbol{y}^{-1}$ denotes $(y_0^{-1}, \ldots, y_{n-1}^{-1})$.

2.4 Generalized Reed Solomon codes and alternant codes

Definition 1 (Generalized Reed-Solomon code)**.** Let $q = 2^m$ for some positive integer m and k, n be integers such that $1 \leqslant k < n \leqslant q$. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two n-tuples such that the entries of $\boldsymbol{x}$ are pairwise distinct elements of $\mathbb{F}_q$ and those of $\boldsymbol{y}$ are nonzero elements in $\mathbb{F}_q$. The *generalized Reed-Solomon* code (GRS in short) $\mathbf{GRS}_k(\boldsymbol{x}, \boldsymbol{y})$ of dimension k associated to $(\boldsymbol{x}, \boldsymbol{y})$ is defined as

$$\mathbf{GRS}_k(\boldsymbol{x}, \boldsymbol{y}) \stackrel{\text{def}}{=} \Big\{ \big(y_0 P(x_0), \ldots, y_{n-1} P(x_{n-1})\big) \;\Big|\; P \in \mathbb{F}_q[z]_{<k} \Big\}.$$

Reed-Solomon codes correspond to the case where $\boldsymbol{y} = (1\ 1\ \cdots\ 1)$ and are denoted as $\mathbf{RS}_k(\boldsymbol{x})$. The vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are called respectively the *support* and the *multiplier* of the code.

A GRS code of dimension k with support $\boldsymbol{x}$ and multiplier $\boldsymbol{y}$ has a generator matrix of the form:

$$\begin{pmatrix} y_0 & \cdots & y_{n-1} \\ x_0 y_0 & \cdots & x_{n-1} y_{n-1} \\ \vdots & & \vdots \\ x_0^{k-1} y_0 & \cdots & x_{n-1}^{k-1} y_{n-1} \end{pmatrix}.$$

This leads to the definition of alternant codes. See for instance [MS86, Chap. 12, §2].

Definition 2 (Binary alternant code)**.** Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$ be a support and a multiplier as defined in Definition 1. Let r be a positive integer, the binary alternant code $\mathscr{A}_r(\boldsymbol{x}, \boldsymbol{y})$ is defined as

$$\mathscr{A}_r(\boldsymbol{x}, \boldsymbol{y}) \stackrel{\text{def}}{=} \mathbf{GRS}_r(\boldsymbol{x}, \boldsymbol{y})^{\perp} \cap \mathbb{F}_2^n.$$

The integer r is referred to as the *degree* of the alternant code and m as its *extension degree*.

Another definition of alternant code, which will be useful in the proposal is given below.

Proposition 1. *Let $\boldsymbol{x}, \boldsymbol{y}, r$ be as in Definition 2. The binary alternant code $\mathscr{A}_r(\boldsymbol{x}, \boldsymbol{y})$ is the right kernel of the matrix:*

$$\boldsymbol{H} = \begin{pmatrix} y_0 & \cdots & y_{n-1} \\ x_0 y_0 & \cdots & x_{n-1} y_{n-1} \\ \vdots & & \vdots \\ x_0^{r-1} y_0 & \cdots & x_{n-1}^{r-1} y_{n-1} \end{pmatrix}.$$

Proposition 2 ([MS86, Chap. 12, § 2])**.** *Let $\boldsymbol{x}, \boldsymbol{y}, r$ be as in Definition 1.*

1. $\dim_{\mathbb{F}_2} \mathscr{A}_r(\boldsymbol{x}, \boldsymbol{y}) \geqslant n - mr$;
2. $d_{min}(\mathscr{A}_r(\boldsymbol{x}, \boldsymbol{y})) \geqslant r + 1$;

where $d_{min}(\cdot)$ denotes the minimum distance of a code.

The key feature of an alternant code is the following fact (see [MS86, Chap. 12, § 9]):

Fact 1. There exists a polynomial time algorithm decoding all errors of Hamming weight at most $\lfloor \frac{r}{2} \rfloor$ once the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are known.

2.5 Binary Goppa codes

Definition 3. Let $\boldsymbol{x} \in \mathbb{F}_q^n$ be a vector with pairwise distinct entries and $\Gamma \in \mathbb{F}_q[z]$ be a polynomial such that $\Gamma(x_i) \neq 0$ for all $i \in \{0, \ldots, n-1\}$. The binary Goppa code $\mathscr{G}(\boldsymbol{x}, \Gamma)$ associated to Γ and supported by $\boldsymbol{x}$ is defined as

$$\mathscr{G}(\boldsymbol{x}, \Gamma) \stackrel{\text{def}}{=} \mathscr{A}_{\deg \Gamma}(\boldsymbol{x}, \Gamma(\boldsymbol{x})^{-1}).$$

We call Γ the *Goppa polynomial* and m the *extension degree* of the Goppa code.

The interesting point about this subfamily of alternant codes is that under some conditions, Goppa codes can correct more errors than a general alternant code.

Theorem 1 ([SKHN76, Theorem 4])**.** *Let $\Gamma \in \mathbb{F}_q[z]$ be a square-free polynomial. Let $\boldsymbol{x} \in \mathbb{F}_q^n$ be a vector of pairwise distinct entries, then*

$$\mathscr{G}(\boldsymbol{x}, \Gamma) = \mathscr{G}\left(\boldsymbol{x}, \Gamma^2\right).$$

From Fact 1, if viewed as $\mathscr{A}_{2\deg\Gamma}(\boldsymbol{x}, \Gamma(\boldsymbol{x})^{-2})$ the Goppa code corrects up to $r = \deg\Gamma$ errors in polynomial-time instead of only $\lfloor \frac{\deg\Gamma}{2} \rfloor$ if viewed as $\mathscr{A}_{\deg\Gamma}(\boldsymbol{x}, \Gamma^{-1}(\boldsymbol{x})))$. On the other hand, these codes have dimension $\geqslant n - mr$ instead of $\geqslant n - 2mr$.

2.6 Quasi–cyclic codes

In what follows ℓ denotes a positive integer.

2.6.1 Definitions

Definition 4. Let ℓ be a positive integer and $\sigma : \mathbb{F}_2^\ell \to \mathbb{F}_2^\ell$ be the cyclic shift map:

$$\sigma : \left\{ \begin{array}{ccc} \mathbb{F}_2^\ell & \longrightarrow & \mathbb{F}_2^\ell \\ (x_0, x_1, \ldots, x_{\ell-1}) & \longmapsto & (x_{\ell-1}, x_0, x_1, \ldots, x_{\ell-2}) \end{array} \right.$$

Now, let n be an integer divisible by ℓ, we define the ℓ–th quasi–cyclic shift σ_ℓ as the map obtained by applying σ block-wise on blocks of length ℓ:

$$\sigma_\ell : \left\{ \begin{array}{ccc} \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2^n \\ \left(\boldsymbol{x}_0 \mid \ldots \mid \boldsymbol{x}_{\frac{n}{\ell}-1}\right) & \longmapsto & \left(\sigma(\boldsymbol{x}_0) \mid \ldots \mid \sigma\left(\boldsymbol{x}_{\frac{n}{\ell}-1}\right)\right) \end{array} \right. ,$$

where $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\frac{n}{\ell}-1}$ denote consecutive sub-blocks of ℓ bits.

This notion is illustrated by Figure 1.

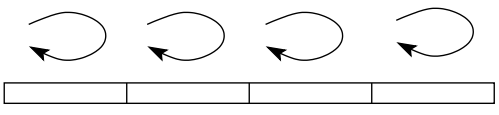


Figure 1: Illustration of the quasi–cyclic shift

Definition 5. A code $\mathscr{C} \subseteq \mathbb{F}_2^n$ is said to be *ℓ–quasi–cyclic* (ℓ–QC) if the code is stable by the quasi–cyclic shift map σ_ℓ. ℓ is also called the *order* of quasi–cyclicity of the code.

Example 1. The following matrix is a generator matrix for a 3–quasi–cyclic code.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

2.6.2 Polynomial representation

Given a positive integer n such that ℓ divides n, then any vector of $\mathbb{F}_2^n$ can be divided into $\frac{n}{\ell}$ blocks of length ℓ. To $(m_0, \ldots, m_{\ell-1})$, one associates naturally the polynomial $m(z) = m_0 + m_1 z + \ldots + m_{\ell-1} z^{\ell-1} \in \mathbb{F}_2[z]/(z^\ell - 1)$. If we let

$$\mathcal{R} \stackrel{\text{def}}{=} \mathbb{F}_2[z]/(z^\ell - 1),$$

then, we get a canonical isomorphism

$$\mathbb{F}_2^n \xrightarrow{\sim} \mathcal{R}^{\frac{n}{\ell}}$$

and, under this isomorphism, the quasi–cyclic shift corresponds to the scalar multiplication by z. Hence quasi–cyclic codes can be regarded as $\mathcal{R}$–sub–modules of $\mathcal{R}^{\frac{n}{\ell}}$.

Example 2. The code of Example 1 corresponds to the submodule of $\mathcal{R}^{\frac{n}{\ell}}$ spanned by

$$(1 \mid 1 + z).$$

2.6.3 Operations on quasi–cyclic codes

For the analysis of some attacks and hence for the security analysis of our codes, we need to introduce the notions of *invariant* code and *folded* code. These notions will be frequently used in Sections 3 and 4.

Notation 1. We denote by $\mathbf{Punct}_\ell$ the function

$$\mathbf{Punct}_\ell : \left\{ \begin{array}{ccc} \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2^{\frac{n}{\ell}} \\ (x_0, \ldots, x_{n-1}) & \longmapsto & (x_0, x_\ell, x_{2\ell}, \ldots, x_{n-\ell}) \end{array} \right. .$$

That is, the map that keeps only the first entry of each block.

Definition 6 (Folding map)**.** Let n be a positive integer such that ℓ divides n. The *folding map* on $\mathbb{F}_2^n$ is the map obtained by summing up the components of each block:

$$\varphi_\ell : \left\{ \begin{array}{ccc} \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2^{\frac{n}{\ell}} \\ (x_0, \ldots, x_{n-1}) & \longmapsto & \left(\sum_{i=0}^{\ell} x_i, \sum_{i=\ell}^{2\ell-1} x_i, \ldots, \sum_{i=n-\ell}^{n-1} x_i \right) \end{array} \right. .$$

Equivalently, $\varphi_\ell \stackrel{\text{def}}{=} \mathbf{Punct}_\ell \circ (\mathrm{Id} + \sigma_\ell + \cdots + {\sigma_\ell}^{\ell-1})$.

Definition 7 (Folded code)**.** Given an ℓ–quasi–cyclic code $\mathscr{C} \subseteq \mathbb{F}_2^n$, the *folded code* $\varphi_\ell(\mathscr{C}) \subseteq \mathbb{F}_2^{\frac{n}{\ell}}$ is the image of $\mathscr{C}$ by the folding map.

Definition 8 (Invariant code)**.** Given an ℓ–quasi–cyclic code $\mathscr{C} \subseteq \mathbb{F}_2^n$, the invariant code $\mathscr{C}^{\sigma_\ell} \subseteq \mathbb{F}_2^{\frac{n}{\ell}}$ is the code composed of words fixed by σ_ℓ whose redundant entries have been removed, i.e.

$$\mathscr{C}^{\sigma_\ell} \stackrel{\text{def}}{=} \mathbf{Punct}_\ell\left(\{\boldsymbol{c} \in \mathscr{C} \mid \sigma_\ell(\boldsymbol{c}) = \boldsymbol{c}\}\right).$$

Remark 1. In the previous definitions, the map $\mathbf{Punct}_\ell$ is always applied to *invariant* words, i.e. words such that $\sigma_\ell(\boldsymbol{x}) = \boldsymbol{x}$. Such words are constant on each block. Therefore, the use of $\mathbf{Punct}_\ell$ is only to remove repetitions. Actually one could have replaced $\mathbf{Punct}_\ell$ by any map that keeps one and only one arbitrary entry per block.

We recall a relation between folded and invariant code.

Proposition 3 ([Bar17])**.** *If ℓ is odd (which always holds in the present proposal), then*

$$\varphi_\ell(\mathscr{C}) = \mathscr{C}^{\sigma_\ell}.$$

Example 3. If we reconsider the code in Example 1, then, we are in the context of the above proposition, hence the invariant and the folding code are both equal to the code spanned by the vector $\begin{pmatrix}1 & 0\end{pmatrix}$ which corresponds to apply $\mathbf{Punct}_\ell$ on

$$\begin{pmatrix}1 & 1 & 1 & 0 & 0 & 0\end{pmatrix}.$$

2.7 Block–circulant matrices

Definition 9. Let ℓ be a positive integer. Let $\boldsymbol{M}$ be a matrix. The matrix is said to be ℓ–block–circulant if it splits into $\ell \times \ell$ circulant blocks, i.e. blocks of the form

$$\begin{pmatrix} a_0 & a_1 & \cdots & \cdots & a_{\ell-1} \\ a_{\ell-1} & a_0 & \cdots & \cdots & a_{\ell-2} \\ \vdots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ a_1 & a_2 & \cdots & a_{\ell-1} & a_0 \end{pmatrix}$$

2.8 Quasi–cyclic Goppa codes

There are several manners to construct ℓ–QC Goppa codes. See for instance reference [Ber00b, Ber00a].

In this proposal, we will consider ℓ–QC binary Goppa codes for some prime integer ℓ constructed as follows. The exact constraints on ℓ are given in §3.1 and justified in §4.

- Let ℓ be a prime dividing $2^m - 1$. Let ζ_ℓ be a primitive ℓ–th root of unity;
- Let n, t be positive integers divisible by ℓ and set $r \stackrel{\text{def}}{=} \frac{t}{\ell}$;
- The support $\boldsymbol{x} = (x_0, \ldots, x_{n-1})$ is a vector of elements of $\mathbb{F}_{2^m}$ whose entries are pairwise distinct. It splits into n/ℓ blocks of length ℓ of the form $(x_{i\ell}, x_{i\ell+1}, \ldots, x_{(i+1)\ell-1})$ such that for any $j \in \{1, \ldots, \ell-1\}$, $x_{i\ell+j} = \zeta_\ell^j x_{i\ell}$. That is, the support is a disjoint union of orbits under the action of the cyclic group generated by ζ_ℓ. From now on, such blocks are referred to as ζ–**orbits**.

- The Goppa polynomial $\Gamma(z)$ is chosen as $\Gamma(z) = g(z^\ell)$ for some monic polynomial $g \in \mathbb{F}_{2^m}[z]$ of degree $r = t/\ell$ such that $g(z^\ell)$ is irreducible.

Proposition 4. *The Goppa code* $\mathscr{G}(\boldsymbol{x}, \Gamma)$ *is* ℓ*–QC.*

2.9 QC–Goppa codes of interest in the present proposal

The ℓ–QC Goppa codes we will consider are those which satisfy the following condition:

Condition 1. The quasi–cyclic code admits a parity–check of the form

$$\boldsymbol{H} = \left(\, I_{n-k} \mid \boldsymbol{M} \, \right)$$

such that $\boldsymbol{M}$ is ℓ–block–circulant.

For a given quasi–cyclic Goppa code it is unclear that such a property is verified. But we observed that, after a possible blockwise permutation of the support (so that the resulting code is still a quasi–cyclic Goppa code of order ℓ) this property is in general satisfied by the codes presented in §2.8. In Appendix A we give an algorithm to find a blockwise permutation providing a quasi–cyclic code satisfying Condition 1. This may happen for instance if no disjoint union of $\frac{n-k}{\ell}$ blocks is an information set. Among the 5000 tests we ran on various parameters, this situation never happened.

Remark 2. Of course, for Condition 1 to be satisfied, the dimension of the code should be a multiple of ℓ. This necessary condition is satisfied if the designed dimension of the Goppa code $n - m \deg g(z^\ell)$ equals the actual dimension (indeed, $\deg(g(z^\ell)) = \ell \cdot \deg(g)$), which almost always holds.

Definition 10. From now on, Goppa codes satisfying Condition 1 are referred to as *Systematic quasi–cyclic Goppa codes.*

The choice of systematic quasi–cyclic codes instead of general quasi–cyclic codes is twofold. First, it makes the security reduction to follow (see §2.10) less technical. Second, such matrices permit to reduce optimally the public key size. Indeed, from such a matrix, $(I_{n-k} \mid \boldsymbol{M})$, it is sufficient to publish only the first row of each circulant block in $\boldsymbol{M}$. Hence, this leads to a public key size $k \times \frac{n-k}{\ell}$. See §5 for further details.

Remark 3. Actually, in our reference implementation, we store the first *column* of each column of blocks.

2.10 Difficult problems from coding theory

Definition 11 ((Search) ℓ-Quasi–Cyclic Syndrome Decoding (ℓ–QCSD) Problem)**.** For positive integers n, t, ℓ, a random parity check matrix $\mathbf{H}$ of a systematic ℓ–quasi–cyclic code $\mathscr{C}$ of dimension k and a uniformly random vector $\mathbf{s} \in \mathbb{F}^{n-k}$, the *Search ℓ-Quasi-Cyclic Syndrome Decoding Problem* ℓ–$\mathsf{QCSD}(n, k, w)$ asks to find $\mathbf{e} = (e_0, \ldots, e_{n-1}) \in \mathbb{F}_2^n$ of Hamming weight t, and $\mathbf{s}^\top = \mathbf{H} \cdot \boldsymbol{e}^\top$.

It would be somewhat more natural to choose the parity-check matrix $\mathbf{H}$ to be made up of independent uniformly random circulant submatrices, rather than with the special form required by Condition 1. We choose this distribution so as to make the security reduction to follow less technical. It is readily seen that, for fixed ℓ, when choosing quasi-cyclic codes

with this more general distribution, one obtains with non-negligible probability, a quasi-cyclic code that satisfies Condition 1. Therefore requiring quasi-cyclic codes to be systematic does not hurt the generality of the decoding problem for quasi-cyclic codes.

Assumption 2. Although there is no general complexity result for quasi-cyclic codes, decoding these codes is considered hard. There exist general attacks which use the cyclic structure of the code [Sen11] but these attacks have only a very limited impact on the practical complexity of the problem. The conclusion is that in practice, the best attacks are the same as those for non-circulant codes up to a small factor.

Definition 12 (Decisional Indistinguishability of Quasi–Cyclic Goppa Codes from Public Key Sampling (DIQCG problem))**.** Given a random ℓ–quasi–cyclic random code in systematic form and an ℓ–quasi–cyclic Goppa code, distinguish the two types of code.

Assumption 3. For parameters considered for our cryptosystem this problem is considered hard, see Section 4 for further details on the best attacks in this case.

3 Presentation of the scheme

3.1 Notation

In what follows and until the end of the present document, we fix the following notation.

- m denotes a positive integer which refers to the extension degree of a field $\mathbb{F}_{2^m}$. In our reference implementation $m = 12, 14, 16$ or 18.
- ℓ denotes a prime primitive integer which divides $2^m - 1$. By *primitive* we mean that ℓ is prime and 2 generates the cyclic group $\mathbb{Z}/\ell\mathbb{Z}^\times$ of nonzero elements in $\mathbb{Z}/\ell\mathbb{Z}$. The rationale behind this requirement will be explained in § 5.1.
- ζ_ℓ denotes a primitive ℓ–th root of the unity in $\mathbb{F}_{2^m}$.
- n denotes a positive integer $n < 2^m - 1$ which refers to the length of a code. It should be an integer multiple of ℓ.
- $\boldsymbol{x} = (x_0, \ldots, x_{n-1})$ denotes the support of the Goppa code. It has length n and splits into $\frac{n}{\ell}$ blocks of length ℓ such that each block is composed of elements in geometric progression. That is to say:

  $$(x_0, x_1, \ldots, x_{\ell-1}) = (x_0, \zeta_\ell x_0, \zeta_\ell^2 x_0, \ldots, \zeta_\ell^{\ell-1} x_0),$$

  and more generally,

  $$\forall a \in \left\{0, \ldots, \frac{n}{\ell} - 1\right\}, \quad (x_{a\ell}, x_{a\ell+1}, \ldots, x_{(a+1)\ell-1}) = (x_{a\ell}, \zeta_\ell x_{a\ell}, \zeta_\ell^2 x_{a\ell}, \ldots, \zeta_\ell^{\ell-1} x_{a\ell}).$$

- $g(z) \in \mathbb{F}_{2^m}[z]$ denotes a polynomial and the Goppa polynomial of our QC–Goppa codes will be $g(z^\ell)$.
- r denotes the degree of $g(z)$ and $t = r\ell$ denotes the degree of the Goppa polynomial $g(z^\ell)$. Notice that the design minimum distance of this Goppa code is $2t + 1$, therefore, t also denotes the error correcting capacity of the code.
- σ_ℓ denotes the ℓ-th quasi–cyclic shift (see Definition 4).

3.2 Key generation

Consider an ℓ–QC Goppa code of length n and dimension k with ℓ dividing n, k and satisfying Condition 1. Let $\boldsymbol{H}$ be a systematic parity–check matrix for this code:

$$\boldsymbol{H} = (\ I_{n-k} \mid \boldsymbol{M}) \tag{1}$$

where $\boldsymbol{M}$ is an ℓ–blocks–circulant matrix.

Definition 13. Given a matrix $\boldsymbol{H}$ as in (1), we define $\psi(\boldsymbol{H})$ as the matrix obtained from $\boldsymbol{M}$ by extracting only the first row of each block. That is, $\psi(\boldsymbol{H})$ is obtained by stacking rows of $\boldsymbol{M}$ with indexes $0, \ell, 2\ell, \ldots, (n-k)-\ell$.

Note that $\boldsymbol{H}$ is entirely determined by $\psi(\boldsymbol{H})$.

- **Public key** The matrix $\psi(\boldsymbol{H})$.
- **Secret key** The support $\boldsymbol{x}$ and the Goppa polynomial $\Gamma(z) = g(z^\ell)$.

More precisely, Algorithm 1 describes the full key generation algorithm. This algorithm calls Algorithm 3, described in Appendix A. Algorithm 3, performs block–Gaussian elimination on an input matrix $\boldsymbol{H}_0$ and, if succeeds, returns a pair $(\boldsymbol{H}, \tau)$ where $\boldsymbol{H}$ denotes a systematic block–circulant matrix and τ denotes the permutation on blocks applied on $\boldsymbol{H}_0$ in order to get the systematic form $\boldsymbol{H}$.

Algorithm 1: Full key generation algorithm

Input : Positive integers ℓ, $t = \ell r$, n, m such that $\ell | n$
Output: The public and secret key

1 **while** *TRUE* **do**
2 $g(z) \leftarrow$ Random monic polynomial in $\mathbb{F}_{2^m}[z]$ of degree r such that $g(z^\ell)$ is irreducible;
3 $u_0, \ldots, u_{\frac{n}{\ell}-1} \leftarrow$ random elements of $\mathbb{F}_{2^m}$ such that for any pair $i, j \in \{0, \ldots, \frac{n}{\ell} - 1\}$ with $i \neq j$ and any $s \in \{0, \ldots, \ell - 1\}$, we have $u_i \neq \zeta_\ell^s u_j$;
4 $\boldsymbol{x} \leftarrow (u_0, \zeta_\ell u_0, \zeta_\ell^2 u_0, \ldots, \zeta_\ell^{\ell-1} u_0, u_1, \zeta_\ell u_1, \ldots, \zeta_\ell^{\ell-1} u_{\frac{n}{\ell}-1})$;
5 $\boldsymbol{H}_0 \leftarrow$ parity–check matrix of $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$;
6 **if** *Algorithm 3 returns FALSE (See Appendix A, page 29)* **then**
7 **Go to line** 2 ;
8 **end**
9 **else**
10 $\boldsymbol{H}, \tau \leftarrow$ output of Algorithm 3;
11 $\boldsymbol{x} \leftarrow \tau(\boldsymbol{x})$;
12 **break** ;
13 **end**
14 **end**
15 **Public key** $\leftarrow (\psi(\boldsymbol{H}), t)$ (see Definition 13);
16 **Secret key** $\leftarrow (\boldsymbol{x}, g(z^\ell))$.

3.3 Description of the public key encryption scheme

3.3.1 Context

- Bob has published his public key $(\psi(\boldsymbol{H}), t)$. His secret key is denoted as the pair $(\boldsymbol{x}, g(z^\ell))$.
- One uses a hash function $\mathcal{H}$. In the reference implementation, we used SHA3;

Suppose Alice wants to send an encrypted message to Bob using Bob's public key. The plaintext is denoted by $\boldsymbol{m} \in \mathbb{F}_2^s$ where s is a parameter of the scheme.

3.3.2 Encryption

(1) e is drawn at random among the set of words of weight t in $\mathbb{F}_2^n$.

(2) Alice sends $\boldsymbol{c} \leftarrow (\boldsymbol{m} \oplus \mathcal{H}(\boldsymbol{e}),\ \boldsymbol{H} \cdot \boldsymbol{e}^\top)$ to Bob.

3.3.3 Decryption

(1) Bob received $(\boldsymbol{c}_1,\ \boldsymbol{c}_2)$.

(2) Using his secret key, Bob computes $\boldsymbol{e} \in \mathbb{F}_2^n$ as the word of weight $\leqslant t$ such that $\boldsymbol{c}_2 = \boldsymbol{H} \cdot \boldsymbol{e}^\top$.

(3) Bob computes $\boldsymbol{m} \leftarrow \boldsymbol{c}_1 \oplus \mathcal{H}(\boldsymbol{e})$.

3.4 Description of the KEM

3.4.1 Context

Alice and Bob want to share a common session secret key K. Moreover,

- Bob publishes his public key $(\psi(\boldsymbol{H}), t)$. His secret key is denoted as the pair $(\boldsymbol{x}, g(z^\ell))$.
- One uses a hash function $\mathcal{H}$. In the reference implementation, we used SHA3.
- To perform a KEM, we need to de–randomize the PKE. This requires the use of a function $\mathcal{F} : \{0,1\}^* \to \{\boldsymbol{x} \in \mathbb{F}_2^n \mid w_H(\boldsymbol{x}) = t\}$ taking an arbitrary binary string as input and returning a word of weight t. The construction of this function is detailed further in § 3.4.4.
- We also introduce a security parameter s which will be the number of bits of security. That is, $s = 128$ (resp. 192, resp. 256) for a 128 (resp. 192, resp. 256) bits security proposal, i.e. for NIST security Levels 1, resp. 3, resp. 5.

3.4.2 Key encapsulation mechanism

(1) Alice generates a random $\boldsymbol{m} \in \mathbb{F}_2^s$;

(2) $\boldsymbol{e} \leftarrow \mathcal{F}(\boldsymbol{m})$;

(3) Alice sends $\boldsymbol{c} \leftarrow (\boldsymbol{m} \oplus \mathcal{H}(\boldsymbol{e}), \boldsymbol{H} \cdot \boldsymbol{e}^T, \mathcal{H}(m))$ to Bob;

(4) The session key is defined as:

$$K \leftarrow \mathcal{H}(\boldsymbol{m}, \boldsymbol{c}).$$

3.4.3 Decapsulation

(1) Bob received $\boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3)$;

(2) Using his secret key, Bob can find $\boldsymbol{e}'$ of weight $\leqslant t$ such that $\boldsymbol{c}_2 = \boldsymbol{H} \cdot \boldsymbol{e}'^T$;

(3) Bob computes, $\boldsymbol{m}' \leftarrow \boldsymbol{c}_1 \oplus \mathcal{H}(\boldsymbol{e}')$;

(4) Bob computes $\boldsymbol{e}'' = \mathcal{F}(\boldsymbol{m}')$.

(5) **If** $\boldsymbol{e}'' \neq \boldsymbol{e}'$ or $\mathcal{H}(\boldsymbol{m}') \neq \boldsymbol{c}_3$ then **abort**.

(6) **Else**, Bob computes the session key:

$$K \leftarrow \mathcal{H}(\boldsymbol{m}', \boldsymbol{c}).$$

3.4.4 The function $\mathcal{F}$

The function $\mathcal{F}$ takes an arbitrary binary string as input and returns a word of length n and weight t. The algorithm is rather simple. Here again, the function depends on the choice of a hash function $\mathcal{H}$. In our reference implementation, we chose SHA3.

The construction of the constant weight word, is performed in constant time using an algorithm close to Knuth's algorithm which generates a uniformly random permutation of the set $\{0, \ldots, n-1\}$. Here, the randomness is replaced by calls of the hash function $\mathcal{H}$. The algorithm of evaluation of $\mathcal{F}$ is detailed in Algorithm 2.

Algorithm 2: Function $\mathcal{F}$: construction of a word of weight t

Input : A binary vector $\boldsymbol{m}$, integers n, t
Output: A word of weight t in $\mathbb{F}_2^n$

1 $\boldsymbol{u} \leftarrow (0, 1, 2, \ldots, n-2, n-1)$;
2 $\boldsymbol{b} \leftarrow \boldsymbol{m}$;
3 **for** i **from** 0 **to** $t-1$ **do**
4 $\quad j \leftarrow \mathcal{H}(\boldsymbol{b}) \mod (n-i-1)$;
5 $\quad$ **Swap** entries u_i and u_{i+j} in u;
6 $\quad \boldsymbol{b} \leftarrow \mathcal{H}(\boldsymbol{b})$;
7 **end**
8 $\boldsymbol{e} \leftarrow$ vector with 1's at positions $u_0, \ldots, u_{t-1}$ and 0's elsewhere;
9 **return** $\boldsymbol{e}$

Further details about line 4 Actually the step $j \leftarrow \mathcal{H}(\boldsymbol{b}) \mod (n-i-1)$ should be detailed. If the hash function $\mathcal{H}$ outputs 256 or 512 bit strings, converting this string to a big integer and then reducing modulo $(n-i-1)$ would be inefficient. Hence, the approach consists in

Step 1. truncating $\mathcal{H}(b)$ to a string of s bytes, where s is larger than the byte size of n. In our proposal, $n < 2^{14}$, hence taking $s = 3$ is reasonable and is the choice of our reference implementation.

Step 2. convert this s–bytes string to an integer A:

(a) If $A > 2^{8s} - (2^{8s} \mod n-i-1)$ then go to Step 1 (this should be done to assert a uniformity of the drawn integers in $\{0, \ldots, n-i-2\}$);

(b) else set $j = A \mod (n-i-1)$

Remark 4. Because of Sub-step (2a), we cannot make sure the evaluation of $\mathcal{F}$ is done in constant time, which could represent a weakness (in terms of side channel attacks). To address this issue, first notice that the probability that Sub-step 2a happens is low, and can be reduced significantly by increasing s. Second, one can get almost constant time by finishing the evaluation of $\mathcal{F}$ by performing a small number of fake evaluations of $\mathcal{H}$ to guarantee a constant number of calls of $\mathcal{H}$ with a high probability. This precaution is not implemented in our reference implementation.

3.5 Semantic security

3.5.1 IND-CPA security of the PKE

Theorem 4. *Under the* Decisional indistinguishability of QC Goppa from Public Key Sampling (DIQCG problem)*, and the ℓ–QCSD Problem, the encryption scheme presented above in indistinguishable against Chosen Plaintext Attack in the Random Oracle Model.*

Proof. We are going to proceed in a sequence of games. The simulator first starts from the real scheme. First we replace the public key matrix by a random element, and then we use the ROM to solve the ℓ–QCSD.

We denote the ciphertext of the PKE by $\boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$ and recall that $\boldsymbol{c}_1 = \boldsymbol{m} \oplus \mathcal{H}(e)$ and $\boldsymbol{c}_2 = \boldsymbol{H} \cdot \boldsymbol{e}^\top$.

We start from the normal game G_0: We generate the public key $\boldsymbol{H}$ honestly, and $\boldsymbol{e}$ and $\boldsymbol{c}_1$ also.

- In game G_1, we now replace $\boldsymbol{H}$ by a random block–circulant systematic matrix, the rest is identical to the previous game. From an adversary point of view, the only difference is the distribution on $\boldsymbol{H}$, which is either generated at random, or as a quasi–cyclic Goppa parity–check matrix. This is exactly the *DIQCG* problem, hence

$$\mathsf{Adv}_{\mathcal{A}}^{G_0} \leqslant \mathsf{Adv}_{\mathcal{A}}^{G_1} + \mathsf{Adv}_{\mathcal{A}}^{\mathrm{DIQCG}}$$

- In game G_2, we now proceed as earlier except we replace $\mathcal{H}(\boldsymbol{e})$ by random. It can be shown, that by monitoring the call to the ROM, the difference between this game and the previous one can be reduced to the ℓ–QCSD problem, so that:

$$\mathsf{Adv}_{\mathcal{A}}^{G_1} \leqslant 2^{-\lambda} + 1/q_G \cdot \mathsf{Adv}_{\mathcal{A}}^{\ell-\mathrm{QCSD}},$$

  where q_G denotes the number of calls to the random oracle.

- In a final game G_3 we replace $\boldsymbol{c}_1 = \boldsymbol{m} \oplus \mathsf{Rand}$ by just $\boldsymbol{c}_1 = \mathsf{Rand}$, which leads to the conclusion.

□

- $\mathsf{Setup}(1^\lambda)$: as before, except that s will be the length of the symmetric key being exchanged, typically $s = 128, 192$, or 256.
- $\mathsf{KeyGen}(\mathsf{param})$: exactly as before.
- $\mathsf{Encapsulate}(\mathbf{pk})$: generate[a] $\mathbf{m} \xleftarrow{\$} \mathbb{F}^s$ (this will serve as a seed to derive the shared key). Derive the randomness $\theta \leftarrow \mathcal{G}(\mathbf{m})$. Generate the ciphertext $c \leftarrow (\mathbf{u}, \mathbf{v}) = \mathcal{E}.\mathsf{Encrypt}(\mathbf{pk}, \mathbf{m}, \theta)$, and derive the symmetric key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$. Let $\mathbf{d} \leftarrow \mathcal{H}(\mathbf{m})$, and send $(\mathbf{c}, \mathbf{d})$.
- $\mathsf{Decapsulate}(\mathbf{sk}, \mathbf{c}, \mathbf{d})$: Decrypt $\mathbf{m}' \leftarrow \mathcal{E}.\mathsf{Decrypt}(\mathbf{sk}, \mathbf{c})$, compute $\theta' \leftarrow \mathcal{G}(\mathbf{m}')$, and (re-)encrypt $\mathbf{m}'$ to get $\mathbf{c}' \leftarrow \mathcal{E}.\mathsf{Encrypt}(\mathbf{pk}, \mathbf{m}', \theta')$. If $\mathbf{c} \neq \mathbf{c}'$ or $\mathbf{d} \neq \mathcal{H}(\mathbf{m}')$ then abort. Otherwise, derive the shared key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$.

[a] Symbol "$\xleftarrow{\$}$" means "uniformly random element of".

Figure 2: Description of our proposal KEM.

3.5.2 Conversion to an IND-CCA2 KEM/DEM

Let $\mathcal{E}$ be an instance of the public key encryption scheme defined in § 3.3. Let $\mathcal{G}$, $\mathcal{H}$, and $\mathcal{K}$ be hash functions, in our implementation, we chose SHA3. The KEM–DEM version of the system cryptosystem is defined as follows:

According to [HHK17], the KEM-DEM version of our PKE is IND–CCA2.

4 Known attacks and counter–measures

We split this section in two parts : key-recovery attacks and message recovery attacks.

4.1 Key recovery attacks

For classical Goppa codes $\mathscr{G}(\boldsymbol{x}, \Gamma)$, a naive brute force key recovery attack consists in enumerating all the possible irreducible polynomials of degree t. In [FOP+16b, FOP+16a] and then in [Bar17], it has been proved that the security of quasi–cyclic Goppa codes (and more generally quasi–cyclic *alternant codes*) reduces to that of the *invariant code* (see Definition 8). Moreover, we have the following result.

Theorem 5 ([FOP+16a, Bar17])**.** *Let $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$ be a QC–Goppa code. Then,*

$$\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)^{\sigma_\ell} = \mathscr{G}\left(\mathbf{Punct}_\ell(\boldsymbol{x}^\ell), g(z)\right)$$

where

$$\boldsymbol{x}^\ell \overset{def}{=} (x_0^\ell, x_1^\ell, \ldots, x_{n-1}^\ell)$$

and the map $\mathbf{Punct}_\ell$ is defined in Notation 1 page 8.

This result is crucial for the security analysis. Indeed, since the public key permits to construct the Goppa code $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$, anybody can compute the invariant code, which is a Goppa code too. Moreover, as soon as the structure of the Goppa code is recovered, lifting to the quasi–cyclic Goppa code is possible. See [FOP+16a].

4.1.1 Exhaustive search on Goppa Polynomials and supports

A brute force attack could consist in enumerating all the possible polynomials $g(z^\ell)$ of $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$. Then guess the support as a disjoint union of ζ_ℓ–orbits, i.e. a disjoint union of ordered sets of the form $(a, \zeta_\ell a, \zeta_\ell^2 a, \ldots, \zeta_\ell^{\ell-1} a)$ (see § 2.8). If we guessed the good support as a non-ordered set, it is possible to get the good permutation using Sendrier's Support Splitting Algorithm (SSA in short, [Sen00]). If it fails, then try with another support defined as a union of ζ–orbits until the good ordering of the support is obtained thanks to SSA.

Actually, this brute force approach can be done on the invariant code and then a lift operation permits to recover the public code. Hence we can proceed as follows.

- Perform brute force search among monic irreducible polynomials $g(z)$ of degree r;
- Guess the support $\mathbf{Punct}_\ell(\boldsymbol{x}^\ell) = \mathbf{Punct}_\ell(x_0^\ell, x_1^\ell, \ldots, x_{n-1}^\ell)$. Note that the elements of the support set are ℓ–th power. Hence there exists only $\frac{2^m-1}{\ell}$ such powers and we need to guess a good subset of length $\frac{n}{\ell}$ among them.
- Perform SSA to check whether the support set is the good one and, if it is, get the permutation and hence the ordered support;
- Deduce from this data the actual Goppa polynomial $g(z^\ell)$ and the good support by extracting ℓ–th roots, here again, the way to find the blockwise good ordering of the elements of the support can be done using either SSA or by solving a linear system.

Thus, let us estimate the maximum number of guesses we need to perform. We need to count the number of monic polynomials $g(z) \in \mathbb{F}_{2^m}[z]$ of degree r such that $g(z^\ell)$ is irreducible.

Set

$$s_r(2^m) \stackrel{\text{def}}{=} \#\{g(z) \in \mathbb{F}_{2^m}[z] \mid \deg(g) = r \text{ and } g(z^\ell) \text{ is irreducible}\}.$$

Remind that the number $m_r(2^m)$ of possible g's, i.e. of monic irreducible polynomials of degree r in $\mathbb{F}_{2^m}[z]$ is given by the well–known formula:

$$m_r(2^m) = \frac{1}{r} \sum_{d|r} \mu(d) 2^{\frac{mr}{d}}, \tag{2}$$

where $\mu(\cdot)$ denotes the Möbius function defined as

$$\mu(r) \stackrel{\text{def}}{=} \sum_{\substack{1 \leqslant k \leqslant r \\ \gcd(k,r)}} e^{2i\pi \frac{k}{r}}. \tag{3}$$

Remark 5. Asymptotically $m_r(2^m) \sim \frac{2^{mr}}{r}$.

Lemma 6.

$$s_r(2^m) \geqslant \left(1 - \frac{1}{\ell}\right) m_r(2^m).$$

The proof of Lemma 6 is given in Appendix C, where a more precise formula is given for $s_r(2^m)$. Consequently, the number of public keys is bounded below by the quantity

$$\#\text{public keys} \geqslant \left(1 - \frac{1}{\ell}\right) m_r(2^m). \tag{4}$$

Remark 6. Actually the number of public keys is much larger since we did not consider the fact that the support of the code need not be the whole $\mathbb{F}_{2^m} \setminus \{0\}$ and hence to estimate the actual number of keys, we need to estimate the number of all the pairs $(\boldsymbol{x}, g(z^\ell))$ where $\boldsymbol{x}$ denotes the support. Then, study the action of the affine group on this set and consider a system of representatives. The point is that for a full support (i.e. the set of elements of the support is $\mathbb{F}_{2^m} \setminus \{0\}$) two distinct Goppa polynomials give two distinct codes and hence there are at least as many keys as $s_r(2^m)$.

4.1.2 Distinguisher on the invariant code

In [FGO$^+$10], it is proved that high rate Goppa codes are distinguishable from random ones in polynomial time. To assert the security of the system, the public Goppa code **and** the invariant code should be indistinguishable from random ones. Hence, the parameters of the code should be chosen in order to be out of the reach of this distinguisher.

The following statement rephrases the results of [FGO$^+$10] in a simpler manner.

Proposition 7. *Consider an irreducible binary Goppa code of length n, extension degree m, associated to an irreducible polynomial of degree r. Then the Goppa code is distinguishable in polynomial time from a random code of the same length and dimension as soon as*

$$n > \max_{2 \leqslant s \leqslant r} \frac{ms}{2} \left(s(m - 2e - 1) + 2^e + 2 \right),$$

where $e = \lceil \log_2 s \rceil + 1$.

4.1.3 Algebraic cryptanalysis

The point of such an attack is to recover the structure of the Goppa code. Namely, the support $\boldsymbol{x}$ and the Goppa polynomial $g(z^\ell)$.

The algebraic modeling $A_{X,Y'}$ proposed in [FOP$^+$16b] consists in $\frac{k}{\ell}(t-1)$ equations in $\frac{n}{\ell}-2$ variables X and $\frac{n}{\ell} - \frac{k}{\ell}$ variables Y, that are bi-homogeneous in the X's and Y's variables (a polynomial f is bi-homogeneous of bi-degree (d_1, d_2) if $f(\alpha X, \beta Y) = \alpha^{d_1}\beta^{d_2} f(X, Y)\ \forall (\alpha, \beta) \in \mathbb{F}^2$). For each $1 \leqslant u \leqslant t-1$, there are $\frac{k}{\ell}$ equations of bi-degree $(u, 1)$ in the modeling. And as the Goppa codes considered are binary Goppa codes, even more equations may be added. The system $\mathtt{McE}_{X,Y'}$ in [FOP$^+$16b] contains $\frac{k}{\ell}$ equations of bi-degree $(u, 1)$ for $1 \leqslant u \leqslant t$ and $\frac{k}{\ell}$ equations of bi-degree $(u, 2)$ for $1 \leqslant u \leqslant 2t-1$.

Exhaustive search on the X_i's or Y_i's From the algebraic system we can extract a bilinear system in the X_i's and the Y_j's. A possible attack is to perform an exhaustive search for one of the X_i's or Y_j's, and solve a linear system for the others. The number of unknowns is at least $\frac{n}{\ell} - \frac{k}{\ell} - 2 = \frac{mt}{\ell} - 2$ (after specialization of 1 or 2 values for X and Y) and the cost of the search is asymptotically $2^{m(\frac{mt}{\ell}-2)}$. The bit complexity is $m(\frac{mt}{\ell} - 2)$ which is large enough.

Solving by Groebner basis algorithms A good indicator for the complexity of Groebner basis algorithms is the index of regularity of the ideal (denoted by d_{reg}), since in the homogeneous case it is a bound on the degree of the polynomials in the minimal Groebner basis of the system. For zero-dimensional ideal, the Hilbert series of the ideal is a polynomial,

and the index of regularity is the degree of this polynomial plus 1. This means that during the computation of a Groebner basis, we will have to compute polynomials that may possibly have as many monomials as the number of monomials of degree d_{reg}, which is $\binom{n+d_{reg}}{d_{reg}}$ for polynomials in n variables.

It has been shown in [BFS04, Bar04] that, if the system of generators of the ideal form a semi–regular sequence of s polynomials of degree $d_1, \ldots, d_s$ in n variables, then the Hilbert series is given by $\left[\frac{\prod_{i=1}^{s}(1-z^{d_i})}{(1-z)^n}\right]^+$ where $\left[\sum_{i\geqslant 0} a_i z^i\right]^+$ is the series $\sum_{i\geqslant 0} a_i z^i$ truncated at the least index i such that $a_i \leqslant 0$.

We show that for the parameters we propose, if the algebraic system from [FOP$^+$16b] where semi–regular (we know it is not), the index of regularity would be large, and that even if the index of regularity of the algebraic system would be small, the size of the polynomials in this degree is beyond the security level.

4.1.4 Algebraic attacks on the invariant code

The cost of such attacks is the most difficult to estimate for many reasons:

- The choice of the algebraic modeling, i.e. the polynomial system we have to solve by Groebner bases methods is not unique. We will suggest here some modeling which have been proposed in the literature but cannot assert that they are the only possible model lings;
- The choice of the monomial ordering has no influence on the theoretical complexity in the worst case, but may have a significant influence on practical complexities.
- Theoretical results on the complexity of Groebner bases suppose the polynomial system to be semi–regular which is not true for the algebraic systems to follow.
- Hence, we provide an analysis of the possible work factor but this approach requires a more thorough study.

4.2 Message recovery attacks

4.2.1 Generic decoding algorithms

Resistance to ISD and their variants. See Christiane Peters' software [Pet10]. We provide here an improve version of her software called **CaWoF** (for **Ca**lculate **Wo**rk **Fa**ctor) [CT16], which tests all the most efficient generic decoding algorithms. Namely:

- Prange [Pra62];
- Stern [Ste88];
- Dumer [Dum91];
- May, Meurer, Thomae [MMT11];
- Becker, Joux, May, Meurer [BJMM12].

4.2.2 About the influence of quasi–cyclicity

Decoding one out of many The ℓ–quasi–cyclicity of the code may be used to improve the efficiency of the decoding using Sendrier's Decoding One out Of Many (DOOM, [Sen11]). Such approach permits to improve the efficiency by a factor $\sqrt{\ell}$. Since in our proposal the largest proposed ℓ is 19, the use of DOOM may in the best case provide a less than 5 bits reduction of the work factor. Note that it is possible that this gain will be undermined by the practical complexity of a DOOM implementation. In § 5.3, we choose parameters so that the work factors provided by **CaWoF** (which does not take DOOM into account) are at least 1 bit above the limit for $\ell = 3$, 2 bits above the limit for $\ell = 5$ and 13, and 3 bits above the limit for $\ell = 19$. For any of our , we looked 3 bits security for 3–QC codes, 4 for 5, 11, 13–QC codes and 5 for 17–QC codes

An attack based on folding There is also another way to use the quasi-cyclicity for performing message recovery attacks. It consists in using the folding. Let φ_ℓ be the folding operation (see Definition 6). Assume that we want to decode $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ where $\boldsymbol{c}$ belongs to the quasi-cyclic Goppa code $\mathscr{C} \stackrel{\text{def}}{=} \mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$ of a certain length n and $\boldsymbol{e}$ is an error of weight t. We clearly have

$$\varphi_\ell(\boldsymbol{y}) = \varphi_\ell(\boldsymbol{c}) + \varphi_\ell(\boldsymbol{e})$$

where $\varphi_\ell(\boldsymbol{c})$ belongs to $\varphi_\ell(\mathscr{C})$ which is the Goppa code $\mathscr{G}\left(\mathbf{Punct}_\ell(\boldsymbol{x}^\ell), g(z)\right)$ by Theorem 5. Each coordinate of $\varphi_\ell(\boldsymbol{e})$ is a sum of ℓ coordinates of $\boldsymbol{e}$. By the piling up lemma (see for instance [Mat93])

$$\mathbb{E}\left\{|\varphi_\ell(\boldsymbol{e})|\right\} = \frac{n(1-(1-2p)^\ell)}{2\ell}, \tag{5}$$

where $p \stackrel{\text{def}}{=} \frac{t}{n}$. We have

$$\begin{aligned}\frac{n(1-(1-2p)^\ell)}{2\ell} &\approx \frac{n(2\ell p - 2\ell(\ell-1)p^2}{2\ell} \\ &\approx t - (\ell-1)\frac{t^2}{n}.\end{aligned}$$

Let

$$t' \stackrel{\text{def}}{=} \lfloor t - (\ell-1)\frac{t^2}{n} \rfloor.$$

In other words, our task is to decode $\varphi_\ell(\boldsymbol{y})$ for about t' errors in a Goppa code of length n/ℓ and degree $r = \deg g$. If t' happens to be below the Gilbert-Varshamov distance $d_{\text{GV}}\left(\frac{n}{\ell}, \frac{n}{\ell} - rm\right)$ corresponding to the length $\frac{n}{\ell}$ and dimension $\frac{n}{\ell} - rm$, then we expect that there is typically a single solution to the decoding problem and that it corresponds to the folding of $\boldsymbol{e}$. Recall that this distance is defined by:

Definition 14 (Gilbert-Varshamov distance)**.** Let $h(x) \stackrel{\text{def}}{=} -x\log_2(x) - (1-x)\log_2(1-x)$ be the binary entropy function and h^{-1} be its inverse ranging over $[0, \frac{1}{2}]$. The Gilbert Varshamov distance $d_{\text{GV}}(n,k)$ of a code of length n and dimension k is defined by

$$d_{\text{GV}}(n,k) \stackrel{\text{def}}{=} n \cdot h^{-1}\left(1 - \frac{k}{n}\right).$$

We can hope to find $\varphi_\ell(\boldsymbol{e})$ by decoding $\varphi_\ell(\mathscr{C})$ with generic decoding techniques. It turns out that we gain in the complexity of decoding when we have to decode the folded code instead of decoding the original code with generic decoding techniques. Once we have the folding $\varphi_\ell(\boldsymbol{e})$ of the error we can use this information to perform decoding of the original code $\mathscr{C}$ by puncturing all the positions in a block which corresponds to a position in the support of $\varphi_\ell(\boldsymbol{e})$. We erase at least t' errors belonging to the support of $\boldsymbol{e}$ in this way. There remains about $t - t' \approx (\ell - 1)\frac{t^2}{n}$ errors which can be recovered by generic decoding techniques. We will chose our parameters in order to avoid this case. We namely choose our parameters so that

$$d_{\mathrm{GV}}(n', k') < t'.$$

The best strategy for an attack in the latter case seems to be

1. hope that the folded error has a certain prescribed weight s;
2. compute all possible errors $\boldsymbol{e}'$ in $\mathbb{F}_2^{n'}$ of weight s that have the same syndrome as $\varphi_\ell(\boldsymbol{y})$;
3. Puncture for each such error the s blocks of $\mathscr{C}$ that belong to the support of $\boldsymbol{e}'$. Decode the punctured code for at most $t - s$ errors.

The attack is then optimized over the choices of s.

4.3 Exploiting Quantum Computations.

Recall first that the NIST proposes to evaluate the quantum security as follows:

1. A quantum computer can only perform quantum computations of limited depth. They introduce a parameter, MAXDEPTH, which can range from 2^{40} to 2^{96}. This accounts for the practical difficulty of building a full quantum computer.
2. The amount (or bits) of security is not measured in terms of absolute time but in the time required to perform a specific task.

Regarding the second point, the NIST presents 6 security categories which correspond to performing a specific task. For example Task 1, related to Category 1, consists of finding the 128 bit key of a block cipher that uses AES-128. The security is then (informally) defined as follows:

Definition 15. A cryptographic scheme is secure with respect to Category k iff any attack on the scheme requires computational resources comparable to or greater than those needed to solve Task k.

In the sequel we will estimate that our scheme reaches a certain security level according to the NIST metric and show that the attack takes more quantum resources than a quantum attack on AES. We will use for this the following proposition.

Proposition 8. *Let f be a Boolean function which is equal to 1 on a fraction α of inputs which can be implemented by a quantum circuit of depth D_f and whose gate complexity is C_f. Using Grover's algorithm for finding an input x of f for which $f(x) = 1$ can not take less quantum resources than a Grover's attack on AES-N as soon as*

$$\frac{D_f \cdot C_f}{\alpha} \geqslant 2^N D_{AES-N} \cdot C_{AES-N}$$

where D_{AES-N} and C_{AES-N} are respectively the depth and the complexity of the quantum circuit implementing AES-N.

This proposition is proved in Appendix B. The point is that (essentially) the best quantum attack on our scheme consists in using Grover's search on either the message attacks or the key recovery attacks where Grover's search can be exploited. The message attacks consist essentially in applying Grover's algorithm on the information sets computed in Prange's algorithm (this is Bernstein's algorithm [Ber10]). Theoretically there is a slightly better algorithm consisting in quantizing more sophisticated ISD algorithms [KT17], however the improvement is tiny and the overhead in terms of circuit complexity make Grover's algorithm used on top of the Prange algorithm preferable in our case.

5 Parameters

In this section we propose some parameters for various security levels. We start with informal discussions which explain in which range we choose our parameters.

5.1 Choice of the quasi–cyclicity order ℓ

The quasi–cyclicity order guarantees the reduction of the key size compared to non quasi–cyclic Goppa codes. The larger the ℓ the smaller the public key.

On the other hand, too large ℓ's may lead to algebraic attacks such as [FOPT10, FOP$^+$16b, FOP$^+$16a]. In addition we suggested in § 3, that ℓ should be prime and *primitive*, which means that 2 generates $(\mathbb{Z}/\ell\mathbb{Z})^\times$, or equivalently that the polynomial $1+z+\cdots+z^{\ell-1}$ is irreducible in $\mathbb{F}_2[z]$. The motivation for this property is to limit the possibilities for the attacker to construct intermediary codes which could help to build an attack. Therefore

- ℓ **should be prime** since if not, for any $e|\ell$, then the code $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$ is also e–quasi–cyclic and one can construct an intermediary invariant code $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)^{\sigma_\ell{}^e}$ which is nothing but $\mathscr{G}\left(\boldsymbol{x}^e, g(z^{\frac{\ell}{e}})\right)$. Thus, it is possible to construct an intermediary code from the single knowledge of a generator matrix of the public code and this intermediary code is a smaller Goppa code with a Goppa polynomial and support strongly related to the public code.

  Of course, we cannot avoid that an attacker can compute the invariant code, but we guess that having the possibility to build intermediary Goppa codes would be a help for the attacker, hence we reject this possibility by requiring ℓ to be prime.

- ℓ **should be primitive** Indeed, in [FOP$^+$16a], from a public Goppa code, the authors consider the *folded* code (see Definition 7), This folding is nothing but the image of the code by the map $\mathrm{id} + \sigma_\ell + \sigma_\ell{}^2 + \cdots + \sigma_\ell{}^{\ell-1}$. In the same manner, if the polynomial $1+z+\cdots+z^{\ell-1}$ is reducible over $\mathbb{F}_2$, then, for any divisor $P(z)$ of this polynomial, one can construct an intermediary quasi–cyclic subcode of the public code, which is the image of $\mathscr{G}\left(\boldsymbol{x}, g(z^\ell)\right)$ by the map $P(\sigma_\ell)$. This code is not a Goppa code in general but we guess that its structure could be helpful for an attacker. Therefore, we exclude this possibility by requiring ℓ to be primitive. Among the odd prime numbers below 20, the primitive ones are

$$\ell \in \{3, 5, 11, 13, 19\}$$

In particular we exclude 7 and 17.

Remark 7. At several places in the discussion above we suggest that having some data could "help an attacker". We emphasize that these arguments are only precautions, we actually do **not** know how to use such data for cryptanalysis. In particular, the choice of ℓ to be primitive is more a precaution than a necessary condition for the security.

5.2 Choice of the field extension m

To provide a binary Goppa code, we first need to choose a finite extension $\mathbb{F}_{2^m}$ of $\mathbb{F}_2$. Let us first discuss the choice of m.

Informal discussion on m

By *informal*, we mean that, for the moment, we do not clarify what we mean by *large* or *small*.

(i) A large m provides codes which are "far from" generalized Reed–Solomon codes. Hence, when m is large Goppa codes have less structure. Note that q–ary Goppa codes with $m = 2$ have been broken by a polynomial-time distinguishing and filtration attack in [COT17] and that rather efficient algebraic attacks for small m ($m = 2$ or 3) over non prime q–ary fields exist [FPdP14]. This encourages to avoid too low values of m. In addition, m should be large enough to have a large enough code length.

(ii) On the other hand m should not be too large since it has a negative influence on the rate of the code. That is to say, for a fixed error correcting capacity t an a fixed code length n, the dimension is $n - mt$, hence the rate is $1 - m\frac{t}{n}$.

(iii) Finally, to get ℓ–quasi–cyclic codes, ℓ should divide $2^m - 1$ (see § 2.8) and ℓ should not be too large to prevent algebraic attacks as [FOPT10, FOP$^+$16b, FOP$^+$16a]. Thus, $2^m - 1$ should have small factors.

In this proposal we suggest that a good tradeoff between (i) and (ii) would be $m \in \{12, \ldots, 18\}$ To seek for ℓ's, let us factorize the corresponding $2^m - 1$'s.

- $\mathbb{F}_{2^{12}}$: $2^{12} - 1 = 3^2 \cdot 5 \cdot 7 \cdot 13$.
- $\mathbb{F}_{2^{13}}$: $2^{13} - 1$ is prime.
- $\mathbb{F}_{2^{14}}$: $2^{14} - 1 = 3 \cdot 43 \cdot 127$.
- $\mathbb{F}_{2^{15}}$: $2^{15} - 1 = 7 \cdot 31 \cdot 151$.
- $\mathbb{F}_{2^{16}}$: $2^{16} - 1 = 3 \cdot 5 \cdot 17 \cdot 257$.
- $\mathbb{F}_{2^{17}}$: $2^{17} - 1$ is prime.
- $\mathbb{F}_{2^{18}}$: $2^{18} - 1 = 3^3 \cdot 7 \cdot 19 \cdot 73$.

This immediately excludes $m = 13$ and 17. To prevent algebraic attacks, we prefer avoiding ℓ's larger than 20 and, as explained above and since we look only for primitive ℓ's our proposal will focus on

- 3, 5 and 13–quasi–cyclic Goppa codes with $m = 12$ (only for security Level 1, i.e. AES128)
- 3–quasi–cyclic Goppa codes with $m = 14$ (for Levels 2 and 3, i.e. respectively AES192 and AES256)
- 5–quasi–cyclic Goppa codes with $m = 16$ (for Levels 2 and 3)
- 19–quasi–cyclic Goppa codes with $m = 18$. (for Levels 2 and 3)

5.3 Proposition of parameters

In the following tables we use notation

- m : extension degree of the field of definition of the support and Goppa polynomial over $\mathbb{F}_2$;
- n length of the quasi–cyclic code;
- k dimension of the quasi–cyclic code;
- ℓ denotes the order of quasi–cyclicity of the code;
- r denotes the degree of $g(z)$;
- t denotes error–correcting capacity, which is nothing but the degree of $g(z^\ell)$;
- w_{msg} work factor for message recovery errors. It is computed using **CaWoF** library;
- Keys is a lower bound for the number of possible Goppa polynomials (see (4));
- Max Dreg denotes the maximal degree of regularity that such a system could have in order that the size of the Macaulay matrix does not exceed 2^{128} bits under the assumption that Gaussian elimination's cost on $n \times n$ matrices is $\Omega(n^2)$.

5.3.1 Parameters for reaching NIST security level 1 (AES128)

m	n	k	ℓ	Size (bytes)	r	$t = r\ell$ ($\deg g(z^\ell)$)	w_{msg}	Keys	Max Dreg
12	3600	2664	3	103896	26	78	129	1027	8
12	3500	2480	5	63240	17	85	130	684	9
12	3510	2418	13	25389	7	91	132	263	11

5.3.2 Parameters for reaching NIST security level 3 (AES192)

m	n	k	ℓ	Size (bytes)	r	$t = r\ell$ ($\deg g(z^\ell)$)	w_{msg}	Keys	Max Dreg
14	6000	4236	3	311346	42	126	193	5751	11
16	7000	5080	5	243840	24	120	195	6798	12
18	7410	4674	19	84132	8	152	195	2696	16

5.3.3 Parameters for reaching NIST security level 5 (AES256)

m	n	k	ℓ	Size bytes	r	$t = r\ell$ ($\deg g(z^\ell)$)	w_{msg}	Keys	Max Dreg
14	9000	7110	3	559913	45	135	257	6039	14
16	9000	6120	5	440640	36	180	260	8129	15
18	10070	6650	19	149625	10	190	263	3412	20

6 Implementation

6.1 Reference implementation

We provide a reference implementation of the public key encryption scheme converted into a key encapsulation mechanism. That is to say, our implementation performs the encapsulation and decapsulation mechanism as described in § 3.4.2 and 3.4.3.

We remind that the hash function used in the reference implementation is SHA3.

6.2 Optimized implementation

Is the same as the reference implementation.

7 Performance Analysis

The platform used in the experiments was equipped with an Intel® Xeon™ E3-1240 v5 clocked at 3.50GHz with 32 GB of RAM and 8 MB of cache. The operating system is 64 bits Linux. The program was compiled with gcc using the `-O4` optimization option.

For the performance (and for the KAT in the next section) we selected three sets of parameters corresponding respectively to the security levels 1, 3, and 5.

- `BIG_QUAKE_1`, corresponding to $(m, n, \ell, t) = (12, 3510, 13, 91)$.
- `BIG_QUAKE_3`, corresponding to $(m, n, \ell, t) = (18, 7410, 19, 152)$.
- `BIG_QUAKE_5`, corresponding to $(m, n, \ell, t) = (18, 10070, 19, 190)$.

7.1 Running time in Milliseconds

	`BIG_QUAKE_1`	`BIG_QUAKE_3`	`BIG_QUAKE_5`
Key Generation	268	2 469	4 717
Encapsulation	1.23	3.00	4.46
Decapsulation	1.41	9.11	13.7

7.2 Space Requirements in Bytes

	`BIG_QUAKE_1`	`BIG_QUAKE_3`	`BIG_QUAKE_5`
Public Key	25 482	84 132	149 800
Secret Key	14 772	30 860	41 804
Ciphertext	201	406	492

8 Known Answer Tests – KAT

The KAT file are available in the submission package for `BIG_QUAKE_1`, `BIG_QUAKE_3`, and `BIG_QUAKE_5`:

- `KAT/PQCkemKAT_BIG_QUAKE_1.req`
- `KAT/PQCkemKAT_BIG_QUAKE_1.rsp`
- `KAT/PQCkemKAT_BIG_QUAKE_3.req`
- `KAT/PQCkemKAT_BIG_QUAKE_3.rsp`
- `KAT/PQCkemKAT_BIG_QUAKE_5.req`
- `KAT/PQCkemKAT_BIG_QUAKE_5.rsp`

For each KAT we generated 10 samples.

References

[Bar04] Magali Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie*. PhD thesis, Université Paris VI, December 2004. http://tel.archives-ouvertes.fr/tel-00449609/en/.

[Bar17] Élise Barelli. On the security of some compact keys for McEliece scheme. In *WCC Workshop on Coding and Cryptography*, September 2017.

[BCGO09] Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. Reducing key length of the McEliece cryptosystem. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 77–97, Gammarth, Tunisia, June 21-25 2009.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *LNCS*, pages 250–272. Springer, 2013.

[Ber00a] Thierry P. Berger. Goppa and related codes invariant under a prescribed permutation. *IEEE Trans. Inform. Theory*, 46(7):2628–2633, 2000.

[Ber00b] Thierry P. Berger. On the cyclicity of Goppa codes, parity-check subcodes of Goppa codes and extended Goppa codes. *Finite Fields Appl.*, 6(3):255–281, 2000.

[Ber10] Daniel J. Bernstein. Grover vs. McEliece. In Nicolas Sendrier, editor, *Post-Quantum Cryptography 2010*, volume 6061 of *LNCS*, pages 73–80. Springer, 2010.

[BFS04] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004.

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In *Advances in Cryptology - EUROCRYPT 2012*, LNCS. Springer, 2012.

[BLM11] Paulo Barreto, Richard Lindner, and Rafael Misoczki. Monoidic codes in cryptography. In *Post-Quantum Cryptography 2011*, volume 7071 of *LNCS*, pages 179–199. Springer, 2011.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. In *Post-Quantum Cryptography 2008*, volume 5299 of *LNCS*, pages 31–46, 2008.

[COT17] Alain Couvreur, Ayoub Otmani, and Jean-Pierre Tillich. Polynomial time attack on wild McEliece over quadratic extensions. *IEEE Trans. Inform. Theory*, 63(1):404–427, Jan 2017.

[CT16] Rodolfo Canto Torres. CaWoF, C library for computing asymptotic exponents of generic decoding work factors, 2016. https://gforge.inria.fr/projects/cawof/.

[Dum91] Ilya Dumer. On minimum distance decoding of linear codes. In *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pages 50–52, Moscow, 1991.

[FGO+10] Jean-Charles Faugère, Valérie Gauthier, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. A distinguisher for high rate McEliece cryptosystems. IACR Cryptology ePrint Archive, Report2010/331, 2010. http://eprint.iacr.org/.

[FOP+16a] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Frédéric de Portzamparc, and Jean-Pierre Tillich. Folding alternant and Goppa Codes with non-trivial automorphism groups. *IEEE Trans. Inform. Theory*, 62(1):184–198, 2016.

[FOP+16b] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Frédéric de Portzamparc, and Jean-Pierre Tillich. Structural cryptanalysis of McEliece schemes with compact keys. *Des. Codes Cryptogr.*, 79(1):87–112, 2016.

[FOPT10] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 279–298, 2010.

[FPdP14] Jean-Charles Faugère, Ludovic Perret, and Frédéric de Portzamparc. Algebraic attack against variants of McEliece with Goppa polynomial of a special form. In *Advances in Cryptology - ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 21–41, Kaoshiung, Taiwan, R.O.C., December 2014. Springer.

[Gab05] Philippe Gaborit. Shorter keys for code based cryptography. In *Proceedings of the 2005 International Workshop on Coding and Cryptography (WCC 2005)*, pages 81–91, Bergen, Norway, March 2005.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[KT17] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. preprint, arXiv:1703.00263 [cs.CR], February 2017.

[Mat93] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology - EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397, Lofthus, Norway, May 1993. Springer.

[MB09] Rafael Misoczki and Paulo Barreto. Compact McEliece keys from Goppa codes. In *Selected Areas in Cryptography*, Calgary, Canada, August 13-14 2009.

[McE78] Robert J. McEliece. *A Public-Key System Based on Algebraic Coding Theory*, pages 114–116. Jet Propulsion Lab, 1978. DSN Progress Report 44.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $O(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 107–124. Springer, 2011.

[MS86] Florence J. MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes*. North–Holland, Amsterdam, fifth edition, 1986.

[OTD10] Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. Cryptanalysis of two McEliece cryptosystems based on quasi-cyclic codes. *Special Issues of Mathematics in Computer Science*, 3(2):129–140, January 2010.

[Pet10] Christiane Peters. Information-set decoding for linear codes over $\mathbf{F}_q$. In *Post-Quantum Cryptography 2010*, volume 6061 of *LNCS*, pages 81–94. Springer, 2010.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[Sen00] Nicolas Sendrier. Finding the permutation between equivalent linear codes: The support splitting algorithm. *IEEE Trans. Inform. Theory*, 46(4):1193–1203, 2000.

[Sen10] Nicolas Sendrier. On the use of structured codes in code based cryptography. In L. Storme S. Nikova, B. Preneel, editor, *Coding Theory and Cryptography III*, pages 59–68. The Royal Flemish Academy of Belgium for Science and the Arts, 2010.

[Sen11] Nicolas Sendrier. Decoding one out of many. In *Post-Quantum Cryptography 2011*, volume 7071 of *LNCS*, pages 51–67, 2011.

[SKHN76] Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, and Toshihiko Namekawa. Further results on Goppa codes and their applications to constructing efficient binary codes. *IEEE Trans. Inform. Theory*, 22:518–526, 1976.

[Ste88] Jacques Stern. A method for finding codewords of small weight. In G. D. Cohen and J. Wolfmann, editors, *Coding Theory and Applications*, volume 388 of *LNCS*, pages 106–113. Springer, 1988.

[Zal99] Christof Zalka. Grover's quantum searching algorithm is optimal. *Phys. Rev. A*, 60:2746–2751, October 1999.

Appendix

A How to get systematic blockwise circulant parity check matrix?

Note that Condition 1 page 10 is not necessarily satisfied. However, it is in general possible to deduce from a general quasi–cyclic Goppa code another QC Goppa code satisfying this condition by applying a permutation preserving the quasi–cyclicity, i.e. a block–wise permutation. Hence, we introduce a second condition

Condition 2. The quasi–cyclic code has an information set which is a disjoint union of blocks.

Clearly, a quasi–cyclic code satisfying Condition 2 can provide after a blockwise permutation a quasi–cyclic code satisfying Condition 1.

Algorithm 3, permits the computation of such a block–wise permutation if it exists. It returns FALSE, if such a permutation is not found which happens for instance if Condition 2 is not satisfied. Applying this algorithm on quasi–cyclic Goppa codes as defined in § 2.8, then after 5000 experiments on quasi–cyclic Goppa codes of various parameters, the algorithm never returned FALSE.

As an input of the algorithm, we need a parity–check matrix $\boldsymbol{H}_0$ which is blockwise circulant. More precisely, the rows of $\boldsymbol{H}_0$ are of the form $c_0, \sigma_\ell(c_0), \ldots, {\sigma_\ell}^{\ell-1}(c_0)$, $c_1, \sigma_\ell(c_1), \ldots,$ ${\sigma_\ell}^{\ell-1}(c_1)$, $\ldots, c_s, \sigma_\ell(c_s), \ldots, {\sigma_\ell}^{\ell-1}(c_s)$. The matrix need not be full rank.

Algorithm 3: Checking Condition 1

Input : A block–wise circulant parity–check matrix $\boldsymbol{H}_0 \in \mathbb{F}_2^{(n-k)\times n}$ of an ℓ–QC Goppa code of length n and dimension k with $\ell | n, k$

Output: Returns FALSE if no block permutation is found. Else, returns TRUE together with

- a blockwise permutation τ to get a systematic code;
- a systematic blockwise circulant parity–check matrix $\boldsymbol{H}$ of the permuted code.

Note. *The Matrix $\boldsymbol{H}_0$ is split into $\ell \times \ell$ square blocks. They are denoted by B_{ij} for $0 \leqslant i < \frac{n-k}{\ell}$ and $0 \leqslant j < \frac{n}{\ell}$. Similarly, the blocks of ℓ rows are denoted by L_i and the blocks of columns by C_j*;
$\tau \leftarrow$ Id (Identity permutation on $\{0, \ldots, n-1\}$);
$\boldsymbol{H} \leftarrow \boldsymbol{H}_0$;
for *i from 0 to $\frac{n-k}{\ell} - 1$* **do**
  if *There exists t, $i \leqslant t < \frac{n-k}{\ell}$ such that B_{ti} is invertible* **then**
    $B \leftarrow B_{ti}$;
    Swap block–rows L_i and L_t;
    $L_i \leftarrow B^{-1} L_i$;
    Eliminate blocks below and above the (i,i)–th one by Gaussian elimination;
  end
  else if *There exists t, $i \leqslant t < \frac{n-k}{\ell}$ and j, $i < j < n$ such that B_{tj} is invertible* **then**
    Swap columns C_i and C_j in $\boldsymbol{H}$;
    $\tau \leftarrow \tau_{ij} \circ \tau$ (τ_{ij} denotes the transposition of i, j);
    Swap rows L_i and L_t in $\boldsymbol{H}$;
    $L_i \leftarrow B^{-1} L_i$;
    Eliminate blocks below and above the (i,i)–th one by Gaussian elimination in $\boldsymbol{H}$;
  end
  else
    return FALSE;
  end
end
return TRUE, $\boldsymbol{H}$, τ;

B Proof of Proposition 8

Let us first recall the proposition we want to prove

Proposition 8. *Let f be a Boolean function which is equal to 1 on a fraction α of inputs which can be implemented by a quantum circuit of depth D_f and whose gate complexity is C_f. Using Grover's algorithm for finding an input x of f for which $f(x) = 1$ can not take less quantum resources than a Grover's attack on AES-N as soon as*

$$\frac{D_f \cdot C_f}{\alpha} \geqslant 2^N D_{AES-N} \cdot C_{AES-N}$$

where D_{AES-N} and C_{AES-N} are respectively the depth and the complexity of the quantum circuit implementing AES-N.

Proof. Following Zalka[Zal99], the best way is to perform Grover's algorithm sequentially with the maximum allowed number of iterations in order not to go beyond MAXDEPTH. Grover's algorithm consists of iterations of the following procedure:

- Apply $U : |0\rangle|0\rangle \to \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}|x\rangle|f(x)\rangle$.
- Apply a phase flip on the second register to get $\sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}(-1)^{f(x)}|x\rangle|f(x)\rangle$.
- Apply $U^\dagger$.

If we perform I iterations of the above for $I \leqslant \frac{1}{\sqrt{\alpha}}$ then the winning probability is upper bounded by αI^2. In our setting, we can perform $I = \frac{\mathsf{MAXDEPTH}}{D_f}$ sequentially before measuring, and each iteration costs time C_f. At each iteration, we succeed with probability αI^2 and we need to repeat this procedure $\frac{1}{\alpha I^2}$ times to get a result with constant probability. From there, we conclude that the total complexity Q is:

$$Q = \frac{1}{\alpha I^2} \cdot I \cdot C_f = \frac{D_f \cdot C_f}{\alpha \mathsf{MAXDEPTH}}. \tag{6}$$

A similar reasoning performed on using Grover's search on AES-N leads to a quantum complexity

$$Q_{AES-N} = \frac{2^N D_{AES-N} \cdot C_{AES-N}}{\mathsf{MAXDEPTH}}. \tag{7}$$

The proposition follows by comparing (6) with (7). □

C Proof of Lemma 6

Remind that $m_r(2^m)$ denotes the number of irreducible polynomials of degree r in $\mathbb{F}_{2^m}[z]$ and $s_r(2^m)$ denotes the number of irreducible polynomials of degree r such that $g(z^\ell)$ is irreducible.

Clearly for $g(z^\ell)$ to be irreducible $g(z)$ should be irreducible too. Conversely, if $g(z)$ is irreducible, and $g(z^\ell)$ reducible, then the factorization of $g(z^\ell)$ is of the form

$$g(z^\ell) = \prod_{i=0}^{\ell-1} h(\zeta^i z) \tag{8}$$

for some irreducible polynomial h. Remind that ζ denotes a primitive ℓ–th root of unity in $\mathbb{F}_{2^m}$. Indeed, the finite subgroup of order ℓ of the affine group spanned by the map $z \mapsto \zeta z$ acts on polynomials as $f(z) \mapsto f(\zeta z)$. Under this action, $g(z^\ell)$ is fixed, hence the polynomials of its irreducible decomposition form an orbit under this action. Moreover, since ℓ is prime, the orbit has size ℓ.

Thus, the polynomials $g(z)$ such that $g(z^\ell)$ is reducible has the form (8). The number of such polynomials is bounded below by $m_r(2^m)/\ell$ which leads to

$$s_r(2^m) \geqslant \left(1 - \frac{1}{\ell}\right) m_r(2^m).$$

Remark 8. Actually one could prove that $s_r(2^m)$ is defined by the following recursive formula:

$$s_r(2^m) = \begin{cases} 2^m - 1 & \text{if} \quad r = 1 \\ m_r(2^m)(1 - \frac{1}{\ell}) & \text{if} \quad \ell \nmid r \\ m_r(2^m) - \frac{1}{\ell}(m_r(2^m) - s_{r/\ell}(2^r)) & \text{else.} \end{cases}$$

BIKE:
Bit Flipping Key Encapsulation

Nicolas Aragon, University of Limoges, France
Paulo S. L. M. Barreto, University of Washington Tacoma, USA
Slim Bettaieb, Worldline, France
Loïc Bidoux, Worldline, France
Olivier Blazy, University of Limoges, France
Jean-Christophe Deneuville, INSA-CVL Bourges and University of Limoges, France
Philippe Gaborit, University of Limoges, France
Shay Gueron, University of Haifa, and Amazon Web Services, Israel
Tim Güneysu, Ruhr-Universität Bochum, and DFKI, Germany,
Carlos Aguilar Melchor, University of Toulouse, France
Rafael Misoczki, Intel Corporation, USA
Edoardo Persichetti, Florida Atlantic University, USA
Nicolas Sendrier, INRIA, France
Jean-Pierre Tillich, INRIA, France
Gilles Zémor, IMB, University of Bordeaux, France

Submitters: The team listed above is the principal submitter. There are no auxiliary submitters.

Inventors/Developers: Same as the principal submitter. Relevant prior work is credited where appropriate.

Implementation Owners: Submitters, Amazon Web Services, Intel Corporation.

Email Address (preferred): rafael.misoczki@intel.com

Postal Address and Telephone (if absolutely necessary):
Rafael Misoczki, Intel Corporation, Jones Farm 2 Building, 2111 NE 25th Avenue, Hillsboro, OR 97124, +1 (503) 264 0392.

Signature: x. See also printed version of “Statement by Each Submitter".

Contents

1 Introduction

This document presents BIKE, a suite of algorithms for key encapsulation based on quasi-cyclic moderate density parity-check (QC-MDPC) codes that can be decoded using bit flipping decoding techniques. In particular, this document highlights the number of security, performance and simplicity advantages that make BIKE a compelling candidate for post-quantum key encapsulation standardization.

1.1 Notation and Preliminaries

Table 1 presents the used notation and is followed by preliminary concepts.

Notation	Description
$\mathbb{F}_2$:	Finite field of 2 elements.
$\mathcal{R}$:	The cyclic polynomial ring $\mathbb{F}_2[X]/\langle X^r - 1\rangle$.
$\lvert v\rvert$:	The Hamming weight of a binary polynomial v.
$u \xleftarrow{\$} U$:	Variable u is sampled uniformly at random from set U.
h_j:	The j-th column of a matrix H, as a row vector.
$\star$:	The component-wise product of vectors.

Table 1: Notation

Definition 1 (Linear codes). *A* binary (n, k)-linear code $\mathcal{C}$ *of length* n *dimension* k *and co-dimension* $r = (n - k)$ *is a* k*-dimensional vector subspace of* $\mathbb{F}_2^n$.

Definition 2 (Generator and Parity-Check Matrices). *A matrix* $G \in \mathbb{F}_2^{k\times n}$ *is called a* generator matrix *of a binary* (n, k)*-linear code* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{mG \mid m \in \mathbb{F}_2^k\}.$$

A matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ *is called a* parity-check matrix *of* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{c \in \mathbb{F}_2^n \mid Hc^T = 0\}.$$

A *codeword* $c \in \mathcal{C}$ of a vector $m \in \mathbb{F}_2^{(n-r)}$ is computed as $c = mG$. A *syndrome* $s \in \mathbb{F}_2^r$ of a vector $e \in \mathbb{F}_2^n$ is computed as $s^T = He^T$.

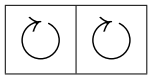
1.2 Quasi-Cyclic Codes

A binary circulant matrix is a square matrix where each row is the rotation one element to the right of the preceding row. It is completely defined by its first row. A block-circulant matrix is formed of circulant square blocks of identical size. The size of the circulant blocks is called the *order*. The *index* of a block-circulant matrix is the number of circulant blocks in a row.

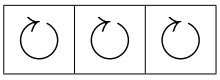
1.2.1 Definition

Definition 3 (Quasi-Cyclic Codes)**.** *A binary quasi-cyclic (QC) code of index n_0 and order r is a linear code which admits as generator matrix a block-circulant matrix of order r and index n_0. A (n_0, k_0)-QC code is a quasi-cyclic code of index n_0, length $n_0 r$ and dimension $k_0 r$.*

For instance:

$G =$ 

The rows of G span a $(2, 1)$-QC code

$G =$ 

The rows of G span a $(3, 1)$-QC code

1.2.2 Representation of QC Codes

Representation of Circulant Matrices. There exists a natural ring isomorphism, which we denote φ, between the binary $r \times r$ circulant matrices and the quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$. The circulant matrix A whose first row is $(a_0, \ldots, a_{r-1})$ is mapped to the polynomial $\varphi(A) = a_0 + a_1 X + \cdots + a_{r-1} X^{r-1}$. This will allow us to view all matrix operations as polynomial operations.

Transposition. For any $a = a_0 + a_1 X + a_2 X^2 + \cdots + a_{r-1} X^{r-1}$ in $\mathcal{R}$, we define $a^T = a_0 + a_{r-1} X + \cdots + a_1 X^{r-1}$. This will ensure $\varphi(A^T) = \varphi(A)^T$.

Vector/Matrix Product. We may extend the mapping φ to any binary vector of $\mathbb{F}_2^r$. For all $\mathbf{v} = (v_0, v_1, \ldots, v_{r-1})$, we set $\varphi(\mathbf{v}) = v_0 + v_1 X + \cdots + v_{r-1} X^{r-1}$. To stay consistent with the transposition, the image of the column vector $\mathbf{v}^T$ must be $\varphi(\mathbf{v}^T) = \varphi(\mathbf{v})^T = v_0 + v_{r-1} X + \cdots + v_1 X^{r-1}$. It is easily checked that $\varphi(\mathbf{v}A) = \varphi(\mathbf{v})\varphi(A)$ and $\varphi(A\mathbf{v}^T) = \varphi(A)\varphi(\mathbf{v})^T$.

Representation of QC Codes as Codes over a Polynomial Ring. The generator matrix of (n_0, k_0)-QC code can be represented as an $k_0 \times n_0$ matrix over $\mathcal{R}$. Similarly any parity check matrix can be viewed as an $(n_0 - k_0) \times n_0$ matrix over $\mathcal{R}$. Respectively

$$G = \begin{pmatrix} g_{0,0} & \cdots & g_{0,n_0-1} \\ \vdots & & \vdots \\ g_{k_0-1,0} & \cdots & g_{k_0-1,n_0-1} \end{pmatrix}, H = \begin{pmatrix} h_{0,0} & \cdots & h_{0,n_0-1} \\ \vdots & & \vdots \\ h_{n_0-k_0-1,0} & \cdots & h_{n_0-k_0-1,n_0-1} \end{pmatrix}$$

with all $g_{i,j}$ and $h_{i,j}$ in $\mathcal{R}$. In all respects, a binary (n_0, k_0)-QC code can be viewed as an $[n_0, k_0]$ code over the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$.

1.3 QC-MDPC Codes

A binary MDPC (Moderate Density Parity Check) code is a binary linear code which admits a somewhat sparse parity check matrix, with a typical density of order $O(1/\sqrt{n})$. The existence of such a matrix allows the use of iterative decoders similar to those used for LDPC (Low Density Parity Check) codes, widely deployed for error correction in telecommunication.

1.3.1 Definition

Definition 4 (QC-MDPC codes). *An (n_0, k_0, r, w)-QC-MDPC code is an (n_0, k_0) quasi-cyclic code of length $n = n_0 r$, dimension $k = k_0 r$, order r (and thus index n_0) admitting a parity-check matrix with constant row weight $w = O(\sqrt{n})$.*

Remark 1. *Asymptotically, a QC-MDPC code could efficiently correct up to $t = O(\sqrt{n} \log n)$ errors. This is a corollary of Theorem 1 given in paragraph "Asymptotic Analysis for MDPC Codes" that follows. In this work, the parity-check row weight w and the error weight t will be chosen so that $wt = O(n)$. This is precisely the regime where the probability of error is expected to decay exponentially in the codelength n (see Theorem 1).*

1.3.2 Decoding - The Bit Flipping Algorithm

The decoding of MPDC code can be achieved by various iterative decoders. Among those, the *bit flipping algorithm* is particularly interesting because of its simplicity. In Algorithm 1 as it is given here the instruction to determine the threshold τ is unspecified. We will always consider regular codes, that is all columns of h have the same weight d and we denote $T = \tau d$. There are several rules for computing the threshold T:

Algorithm 1 `Bit Flipping Algorithm`

Require: $H \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$
Ensure: $eH^T = s$
1: $e \leftarrow 0$
2: $s' \leftarrow s$
3: **while** $s' \neq 0$ **do**
4: $\quad \tau \leftarrow$ threshold $\in [0, 1]$, found according to some predefined rule
5: $\quad$ **for** $j = 0, \ldots, n-1$ **do**
6: $\quad\quad$ **if** $|h_j \star s'| \geq \tau\, |h_j|$ **then**
7: $\quad\quad\quad e_j \leftarrow e_j + 1 \mod 2$
8: $\quad s' \leftarrow s - eH^T$
9: **return** e

h_j denotes the j-th column of H, as a row vector, $'\star'$ denotes the componentwise product of vectors, and $|h_j \star s|$ is the number of unchecked parity equations involving j.

- the maximal value of $|h_j \star s|$ minus some δ (typically $\delta = 5$), as in [31],
- precomputed values depending on the iteration depth, as in [12],
- variable, depending on the weight of the syndrome s', as in [11].

The algorithm takes as input a parity check matrix H and a word s and, if it stops, returns an error pattern e whose syndrome is s. If H is sparse enough and there exists an error e of small enough weight such that $s = eH^T$, then, with high probability, the algorithm stops and returns e.

Asymptotic Analysis for MDPC Codes For a fixed code rate k/n, let us denote w the weight of the rows of H and t the number of errors we are able to decode. Both w and t are functions of n. For LDPC codes, w is a constant and t will be a constant proportion of n, that is $wt = \Omega(n)$. For MDPC codes, we have $w = \Omega(\sqrt{n})$ and the amount of correctable errors will turn out to be a little bit higher than $t = \Omega(\sqrt{n})$.

To understand this point, let us first notice that experimental evidence seems to indicate that the decoding error probability is dominated by the probability that the first round of the algorithm is unable to reduce significantly the number of initial errors. What we call here "round" of the decoding algorithm is an execution of for loop 5 in Algorithm 1. It also seems that at the first round of the decoding algorithm the individual bits of the syndrome bits s_i can be approximated by

independent random variables. This independence assumption can also be made for the vectors $h_j \star s = h_j \star s'$ at the first round. In other words, we make the following assumptions.

Assumption 1. *Let P_{err} be the probability that the bit flipping algorithm fails to decode. Let e^1 be the value of error-vector e after executing for loop 5 once in Algorithm 1 and let e^0 be the true error vector. Let $\Delta e = e^0 + e^1$ (addition is performed in $\mathbb{F}_2$) be the error vector that would remain if we applied the correction e^1 to the true error vector e^0.*

- *There exists a constant α in $(0, 1)$ such that*

$$P_{err} \leq \mathbb{P}(|\Delta e| \geq \alpha t).$$

- *The syndrome bits s_i are independent random variables.*
- *For $j = 0, \ldots, n-1$, the $h_j \star s$ are independent random variables.*

By making these assumptions we can prove that

Theorem 1. *Under assumption 1, the probability P_{err} that the bit flipping algorithm fails to decode with fixed threshold $\tau = \frac{1}{2}$ is upper-bounded by*

$$P_{err} \leq \frac{1}{\sqrt{\alpha \pi t}} e^{\frac{\alpha t w}{8} \ln\left(1-\varepsilon^2\right) + \frac{\alpha t}{8} \ln(n) + O(t)},$$

where $\varepsilon \stackrel{def}{=} e^{-\frac{2wt}{n}}$.

This theorem is proved in Section A of the appendix. This theorem shows that the probability of error decays exponentially in the codelength when $wt = O(n)$ and that the number of correctable errors is a little bit larger than $O(\sqrt{n})$ when $w = O(\sqrt{n})$: it can be as large as some constant $\beta\sqrt{n}\ln n$ as the upper-bound in this theorem is easily shown to converge to 0 for a small enough constant β.

Decoding with a Noisy Syndrome Noisy syndrome decoding is a variation of syndrome decoding in which, given H and s, we look for $e \in \mathbb{F}_2^n$ such that $s - eH^T$ and e are both of small weight. The bit flipping algorithm can be adapted to noisy syndromes. Two things must be modified. First the stopping condition: we do not require the quantity $s - eH^T$ to be null, only to have a small weight. Second, since we need to quantify the weight in this stopping condition, we need to specify a target weight u. For input (H, s, u) a pair e is returned such that $s = e' + eH^T$ for some e' of weight at most u. If $u = 0$ we have the usual bit flipping algorithm.

Algorithm 2 `Extended Bit Flipping Algorithm`

Require: $H \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, integer $u \geq 0$
Ensure: $\left|s - eH^T\right| \leq u$
1: $e \leftarrow 0$
2: $s' \leftarrow s$
3: **while** $|s'| > u$ **do**
4: $\quad \tau \leftarrow$ threshold $\in [0,1]$, found according to some predefined rule // *whatever that means*
5: $\quad$ **for** $j = 0, \ldots, n-1$ **do**
6: $\quad\quad$ **if** $|h_j \star s'| \geq \tau\,|h_j|$ **then**
7: $\quad\quad\quad e_j \leftarrow e_j + 1 \mod 2$
8: $\quad s' \leftarrow s - eH^T$
9: **return** e

Again if H is sparse enough and there exists a solution the algorithm will stop with high probability. Note that if the algorithm stops, it returns a solution within the prescribed weight, but this solution might to be unique. In the case of MDPC codes, the column weight and the error weight are both of order $\sqrt{r}$ and the solution is unique with high probability.

Noisy Syndrome *vs.* Normal Bit Flipping Interestingly, for MDPC codes, noisy syndromes affect only marginally the performance of the bit flipping algorithm. In fact, if e is the solution of $s = e' + eH^T$, then it is also the solution of $s = (e, 1)H'^T$ where H' is obtained by appending e' as $n+1$-th column. For MDPC codes, the error vector e' has a density which is similar to that of H and thus H' is sparse and its last column is not remarkably more or less sparse. Thus applying the bit flipping algorithm to (H', s) is going to produce e, except that we do not allow the last position to be tested in the loop and control is modified to stop the loop when the syndrome s' is equal to the last column of H'. Since we never test the last position we don't need to know the value of the last column of H' except for the stopping condition which can be replaced by a test on the weight. Thus we emulate (almost) the noisy syndrome bit flipping by running the bit flipping algorithm on a code of length $n+1$ instead of n, to correct $|e|+1$ errors instead of $|e|$.

QC-MDPC Decoding for Decryption Quasi-cyclicity does not change the decoding algorithm. The above algorithm will be used for $(2,1)$-QC MDPC codes. It allows us to define the procedure specified as follows. For any triple $(s, h_0, h_1) \in \mathcal{R}^3$ and any integer u

$\texttt{Decode}(s, h_0, h_1, u)$ returns $(e_0, e_1) \in \mathcal{R}^2$ with $|e_0 h_0 + e_1 h_1 + s| \leq u$.

The fourth argument u is an integer. If $u = 0$ the algorithm stops when $e_0 h_0 + e_1 h_1 = s$, that is the noiseless syndrome decoding, else it stops when $e_0 h_0 + e_1 h_1 = s + e$ from some e of weight at most u, that is the noisy syndrome decoding. In addition we will bound the running time (as a function of the block size r) and stop with a failure when this bound is exceeded.

1.4 Key Encapsulation Mechanisms

A key encapsulation mechanism (KEM) is composed by three algorithms: Gen which outputs a public encapsulation key pk and a private decapsulation key sk, Encaps which takes as input an encapsulation key pk and outputs a ciphertext c and a symmetric key K, and Decaps which takes as input a decapsulation key sk and a cryptogram c and outputs a symmetric key K or a decapsulation failure symbol $\perp$. For more details on KEM definitions, we refer the reader to [14].

2 Algorithm Specification (2.B.1)

BIKE relies purely on ephemeral keys, meaning that a new key pair is generated at each key exchange. In this way, the GJS attack [21], which depends on observing a large number of decoding failures for a same private key, is not applicable.

In the following we will present three variants of BIKE, which we will simply label BIKE-1, BIKE-2 and BIKE-3. All of the variants follow either the McEliece or the Niederreiter framework, but each one has some important differences, which we will discuss individually.

For a security level λ, let r be a prime such that $(X^r - 1)/(X - 1) \in \mathbb{F}_2[X]$ is irreducible, d_v be an odd integer and t be an integer such that decoding t errors with a uniformly chosen binary linear error-correcting code of length $n = 2r$ and dimension r, as well as recovering a base of column weight d_v given an arbitrary base of a code of the same length and dimension, both have a computational cost in $\Omega(\exp(\lambda))$. See Section 5 for a detailed discussion on parameters selection.

We denote by $\mathbf{K} : \{0,1\}^n \to \{0,1\}^{\ell_K}$ the hash function used by encapsulation and decapsulation, where ℓ_K is the desired symmetric key length (typically 256 bits).

2.1 BIKE-1

In this variant, we privilege a fast key generation by using a variation of McEliece. A preliminary version of this approach appears in [4].

First, in contrast to QC-MDPC McEliece [31] (and any QC McEliece variant), we do not compute the inversion of one of the private cyclic blocks and then multiply it by the whole private matrix to get systematic form. Instead, we hide the private code structure by simply multiplying its sparse private matrix by any random, dense cyclic block. The price to pay is the doubled size for the public key and the data since the public key will not feature an identity block anymore.

Secondly, we interpret McEliece encryption as having the message conveyed in the error vector, rather than the codeword. This technique is not new, following the lines of Micciancio's work in [30] and having already been used in a code-based scheme by Cayrel et al. in [9].

2.1.1 KeyGen

- Input: λ, the target quantum security level.
- Output: the sparse private key (h_0, h_1) and the dense public key (f_0, f_1).

0. Given λ, set the parameters r, w as described above.
1. Generate $h_0, h_1 \xleftarrow{\$} \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$.
2. Generate $g \xleftarrow{\$} \mathcal{R}$ of odd weight (so $|g| \approx r/2$).
3. Compute $(f_0, f_1) \leftarrow (gh_1, gh_0)$.

2.1.2 Encaps

- Input: the dense public key (f_0, f_1).
- Output: the encapsulated key K and the cryptogram c.

1. Sample $(e_0, e_1) \in \mathcal{R}^2$ such that $|e_0| + |e_1| = t$.
2. Generate $m \xleftarrow{\$} \mathcal{R}$.
3. Compute $c = (c_0, c_1) \leftarrow (mf_0 + e_0, mf_1 + e_1)$.
4. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$.

2.1.3 Decaps

- Input: the sparse private key (h_0, h_1) and the cryptogram c.
- Output: the decapsulated key K or a failure symbol $\perp$.

1. Compute the syndrome $s \leftarrow c_0 h_0 + c_1 h_1$.
2. Try to decode s (noiseless) to recover an error vector (e'_0, e'_1).
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, output $\perp$ and halt.
4. Compute $K \leftarrow \mathbf{K}(e'_0, e'_1)$.

2.2 BIKE-2

In this variant, we follow Niederreiter's framework with a systematic parity check matrix. The main advantage is that this only requires a single block of length r for all the objects involved in the scheme, and thus yields a very compact formulation. On the other hand, this means that it is necessary to perform a polynomial inversion. In this regard, it is worth mentioning that an inversion-based key generation can be significantly slower than encryption (e.g., up to 21x as reported in [28]). A possible solution is to use a batch key generation as described in Section 3.4.

2.2.1 KeyGen

- Input: λ, the target quantum security level.
- Output: the sparse private key (h_0, h_1) and the dense public key h.

0. Given λ, set the parameters r, w as described above.
1. Generate $h_0, h_1 \xleftarrow{\$} \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$.
2. Compute $h \leftarrow h_1 h_0^{-1}$.

2.2.2 Encaps

- Input: the dense public key h.
- Output: the encapsulated key K and the cryptogram c.

1. Sample $(e_0, e_1) \in \mathcal{R}^2$ such that $|e_0| + |e_1| = t$.
2. Compute $c \leftarrow e_0 + e_1 h$.
3. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$.

2.2.3 Decaps

- Input: the sparse private key (h_0, h_1) and the cryptogram c.
- Output: the decapsulated key K or a failure symbol $\perp$.

1. Compute the syndrome $s \leftarrow ch_0$.
2. Try to decode s (noiseless) to recover an error vector (e'_0, e'_1).
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, output $\perp$ and halt.
4. Compute $K \leftarrow \mathbf{K}(e'_0, e'_1)$.

2.3 BIKE-3

This variant follows the work of Ouroboros [15]. Looking at the algorithms description, the variant resembles BIKE-1, featuring fast, inversion-less key generation and two blocks for public key and data. The main difference is that the decapsulation invokes the decoding algorithm on a "noisy" syndrome. This also means that BIKE-3 is fundamentally distinct from BIKE-1 and BIKE-2, mainly in terms of security and security-related aspects like choice of parameters. We will discuss this in the appropriate section.

2.3.1 KeyGen

- Input: λ, the target quantum security level.
- Output: the sparse private key (h_0, h_1) and the dense public key (f_0, f_1).

0. Given λ, set the parameters r, w as described above.
1. Generate $h_0, h_1 \xleftarrow{\$} \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$.
2. Generate $g \xleftarrow{\$} \mathcal{R}$ of odd weight (so $|g| \approx r/2$).
3. Compute $(f_0, f_1) \leftarrow (h_1 + gh_0, g)$.

2.3.2 Encaps

- Input: the dense public key (f_0, f_1).
- Output: the encapsulated key K and the cryptogram c.

1. Sample $(e, e_0, e_1) \in \mathcal{R}^3$ with $|e| = t/2$ and $|e_0| + |e_1| = t$.
2. Compute $c = (c_0, c_1) \leftarrow (e + e_1 f_0, e_0 + e_1 f_1)$.
3. Compute $K \leftarrow \mathbf{K}(e_0, e_1)$.

2.3.3 Decaps

- Input: the sparse private key (h_0, h_1) and the cryptogram c.
- Output: the decapsulated key K or a failure symbol $\perp$.

1. Compute the syndrome $s \leftarrow c_0 + c_1 h_0$.
2. Try to decode s (with noise at most $t/2$) to recover error vector (e'_0, e'_1).
3. If $|(e'_0, e'_1)| \neq t$ or decoding fails, output $\perp$ and halt.
4. Compute $K \leftarrow \mathbf{K}(e'_0, e'_1)$.

For ease of comparison, we provide a summary of the three schemes in the Table below.

	BIKE-1	BIKE-2	BIKE-3
SK	(h_0, h_1) with $\|h_0\| = \|h_1\| = w/2$		
PK	$(f_0, f_1) \leftarrow (gh_1, gh_0)$	$(f_0, f_1) \leftarrow (1, h_1 h_0^{-1})$	$(f_0, f_1) \leftarrow (h_1 + gh_0, g)$
Enc	$(c_0, c_1) \leftarrow (mf_0 + e_0, mf_1 + e_1)$	$c \leftarrow e_0 + e_1 f_1$	$(c_0, c_1) \leftarrow (e + e_1 f_0, e_0 + e_1 f_1)$
	$K \leftarrow \mathbf{K}(e_0, e_1)$		
Dec	$s \leftarrow c_0 h_0 + c_1 h_1$; $u \leftarrow 0$	$s \leftarrow c h_0$; $u \leftarrow 0$	$s \leftarrow c_0 + c_1 h_0$; $u \leftarrow t/2$
	$(e'_0, e'_1) \leftarrow \texttt{Decode}(s, h_0, h_1, u)$		
	$K \leftarrow \mathbf{K}(e'_0, e'_1)$		

Table 2: Algorithm Comparison

We remark that e can be represented with only $\lceil \log_2 \binom{n}{t} \rceil$ bits and such a compact representation can be used if memory is the preferred metric of optimization (the hash function $\mathbf{K}$ would need to be changed as well to receive $\lceil \log_2 \binom{n}{t} \rceil$ bits instead of n).

2.4 Suggested Parameters

The parameters suggested in this section refer to the security levels indicated by NIST's call for papers, which relate to the hardness of a key search attack on a block cipher, like AES. More precisely, we indicate parameters for Levels 1, 3 and 5, corresponding to the security of AES-128, AES-192 and AES-256 respectively.

In addition, the block size r is chosen so that the MDPC decoder described in Section 2.5 has a failure rate not exceeding 10^{-7} (validated through exhaustive simulation). Table 3 summarizes these three parameter suggestions.

	BIKE-1 and BIKE-2				BIKE-3			
Security	n	r	w	t	n	r	w	t
Level 1	20,326	10,163	142	134	22,054	11,027	134	154
Level 3	39,706	19,853	206	199	43,366	21,683	198	226
Level 5	65,498	32,749	274	264	72,262	36,131	266	300

Table 3: Suggested Parameters.

2.5 Decoding

In all variants of BIKE, we will consider the decoding as a black box running in bounded time and which either returns a valid error pattern or fails. It takes as arguments a (sparse) parity check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, a syndrome $s \in \mathbb{F}_2^{n-k}$, and an integer $u \geq 0$. Any returned value e is such that the Hamming distance between eH^T and s is smaller than u.

For given BIKE parameters r, w, t and variant, the key features are going to be the decoding time and the DFR (Decoding Failure Rate). Let $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$, the decoder input (H, s, u) is such that:

- the matrix H is block-circulant of index 2, that is a $H = (h_0^T \quad h_1^T) \in \mathcal{R}^{1\times 2}$ such that $|h_0| = |h_1| = w/2$
- the integer u is either 0 (noiseless syndrome decoding, BIKE-1 and BIKE-2) or $t/2$ (noisy syndrome decoding, BIKE-3).
- the syndrome s is equal to $e' + e_0h_0 + e_1h_1$ for some triple $(e', e_0, e_1) \in \mathcal{R}^3$ such that $|e'| = u$ and $|e_0| + |e_1| = t$.

For each parameter set and each BIKE variant, the decoder input is entirely defined by h_0, h_1, e', e_0, e_1. The DFR is defined as the probability for the decoder to fail when the input (h_0, h_1, e', e_0, e_1) is distributed uniformly such that $|h_0| = |h_1| = w/2$, $|e'| = u$, and $|e_0| + |e_1| = t$.

2.5.1 One-Round Decoding

We will use the decoder defined in Algorithm 3. As it is defined, this algorithm returns a valid error pattern when it stops but it may not stop. In practice, A maximum running time is set, when this maximum is reached the algorithm stops with a failure. For given BIKE parameters r, w, and t, we have $n = 2r$ and $k = r$. In addition, we must (1) set values for S and δ and (2) provide a rule for computing the threshold (instruction 1).

Algorithm 3 `One-Round Bit Flipping Algorithm`

Require: $H \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, integer $u \geq 0$
Ensure: $\left|s - eH^T\right| \leq u$

$T \leftarrow \texttt{threshold}(|s|)$
for $j = 0, \ldots, n-1$ **do**
  $\ell \leftarrow \texttt{min}(\texttt{ctr}(H, s, j), T)$
  $J_\ell \leftarrow J_\ell \cup \{j\}$ *// all J_ℓ empty initially*
$e \leftarrow J_T$ (**)
$s' \leftarrow s - eH^T$
while $|s'| > S$ **do** (***)
  for $\ell = 0, \ldots, \delta$ **do** (***)
    $e' \leftarrow \texttt{check}(H, s', J_{T-\ell}, d/2)$
    $(e, s') \leftarrow (e + e', s' - e'H^T)$ *// update error and syndrome*
$e' \leftarrow \texttt{check}(H, s', e, d/2)$ (**)
$(e, s') \leftarrow (e + e', s' - e'H^T)$ *// update error and syndrome*
while $|s'| > u$ **do**
  $j \leftarrow \texttt{guess\_error\_pos}(H, s', d/2)$
  $(e_j, s') \leftarrow (e_j + 1, s' + h_j)$ (*)
return e

`check`(H, s, J, T)
  $e \leftarrow 0$
  for $j \in J$ **do**
    if `ctr`$(H, s, j) \geq T$ **then**
      $e_j \leftarrow 1$
  return e

`guess_error_pos`(H, s, T)
  loop *// until success*
    $i \xleftarrow{\$} s$ (**)
    for $j \in eq_i$ **do** (*),(**)
      if `ctr`$(H, s, j) \geq T$ **then**
        return j

`ctr`(H, s, j)
  return $|h_j \cap s|$ (*),(**)

`threshold`(S)
  return function of r, w, t, and S

(*) *h_j the j-th column of H (as a row vector), eq_i the i-th row of H*
(**) *we identify binary vectors with the set of their non zero positions*
(***) *the algorithm uses two parameters S and δ which depend of r, w, and t*

Threshold Selection Rule. This rule derives from [10]. We use the notation of the algorithm, $s = eH^T$ is the input syndrome and e the corresponding (unknown) error. We denote $d = w/2$ the column weight of H. Let

$$\pi_1 = \frac{|s| + X}{td} \text{ and } \pi_0 = \frac{w\,|s| - X}{(n-t)d} \text{ where } X = \sum_{\ell \text{ odd}} (\ell - 1)\frac{r\binom{w}{\ell}\binom{n-w}{t-\ell}}{\binom{n}{t}}.$$

The counter value $|h_j \cap d|$ follows a distribution very close to a binomial distribution[1] $B(d, \pi_1)$ and $B(d, \pi_0)$ respectively if $e_j = 1$ or $e_j = 0$. From that it follows that the best threshold is the smallest integer T such that

$$t\binom{d}{T}\pi_1^T(1-\pi_1)^{d-T} \geq (n-t)\binom{d}{T}\pi_0^T(1-\pi_0)^{d-T},$$

that is (note that $\pi_1 \geq \pi_0$)

$$T = \left\lceil \frac{\log \frac{n-t}{t} + d\log\frac{1-\pi_0}{1-\pi_1}}{\log\frac{\pi_1}{\pi_0} + \log\frac{1-\pi_0}{1-\pi_1}} \right\rceil. \tag{1}$$

This value depends only of $n = 2r$, $w = 2d$, $t = |e|$ the error weight, and $|s|$ the syndrome weight. Details can be found in [10]. For any set of parameters thresholds can be precomputed.

In practice for a given set of parameters the formula (1) is very accurately approximated, in the relevant range for the syndrome weight, by an affine function:

- for BIKE-1 and BIKE-2
  - security level 1: $T = \lceil 13.530 + 0.0069722\ |s| \rceil$,
  - security level 3: $T = \lceil 15.932 + 0.0052936\ |s| \rceil$,
  - security level 5: $T = \lceil 17.489 + 0.0043536\ |s| \rceil$,
- for BIKE-3
  - security level 1: $T = \lceil 13.209 + 0.0060515\ |s| \rceil$,
  - security level 3: $T = \lceil 15.561 + 0.0046692\ |s| \rceil$,
  - security level 5: $T = \lceil 17.061 + 0.0038459\ |s| \rceil$.

2.6 Auxiliary Functions

Possible realizations of the auxiliary functions required by BIKE are described next. Other techniques can be used as long as they meet the target security level.

[1] $B(n, p)$ the number of success out of n Bernouilli trials of probability p

2.6.1 Pseudorandom Random Generators

Three types of pseudorandom bits stream generation are considered: no constraints on the output weight (Alg. 4), odd weight (Alg. 5), and specific weight w (Alg. 6). The common building block for them is AES-CTR-PRF based on AES-256, in CTR mode (following NIST SP800-90A guidelines [3]). For typical BIKE parameters the number of calls to AES with a given key is way below the restrictions on using AES in CTR mode. We remark that such AES-CTR-PRF generator is very efficient on modern processors equipped with dedicated AES instructions (e.g., AES-NI).

Algorithm 4 GenPseudoRand(seed, len)

Require: seed (32 bytes)
Ensure: $\bar{z}$ (len pseudo-random bits z embedded in array of bytes).
1: $s = \text{AES-CTR-INIT}(\text{seed}, 0, 2^{32}-1)$
2: $z = \text{truncate}_{\text{len}}(\text{AES-CTR-PRF}(s, \text{len}))$
3: **return** $\bar{z}$

Algorithm 5 GenPseudoRandOddWeight(seed, len)

Require: seed (32 bytes), len
Ensure: $\bar{z}$ (len pseudorandom bits z of odd weight, in a byte array).
1: z = GenPseudoRand(seed, len)
2: **if** $weight(z)$ is even **then** z[0] = z[0] $\oplus$1
3: **return** $\bar{z}$

Algorithm 6 WAES-CTR-PRF(s, wt, len)

Require: s (AES-CTR-PRF state), wt (32 bits), len
Ensure: A list (wlist) of wt bit-positions in $[0, \ldots, \text{len}-1]$, updated s.
1: wlist= ϕ; valid_ctr = 0
2: **while** valid_ctr < wt **do**
3: (pos, s) = AES-CTR-PRF(s, 4)
4: **if** ((pos < len) AND (pos $\notin$ wlist)) **then**
5: wlist = wlist $\cup$ {pos}; valid_ctr = valid_ctr + 1
6: **return** wlist, s

2.6.2 Efficient Hashing

In this section, we describe a parallelized hash technique (see [17, 18, 20]) that can be used to accelerate the hashing process. We stress that a simple hash (e.g., SHA2 or SHA3 hash family) call can be used instead if (for interoperability reasons, for instance) a standard hash function is preferred. Let hash be a hash function with digest length of ld bytes that uses a compression function compress which consumes a block of hbs bytes. The ParallelizedHash, with s slices, and pre-padding length srem, is described in Alg. 7. In our accompanying implementations, we instantiated hash with SHA-384.

Algorithm 7 ParallelizedHash

Require: an array of la bytes array[la − 1 : 0], such that la ≥ s > 0
Ensure: digest (ld bytes)

```
1:  procedure ComputeSliceLen(la)
2:      tmp := floor(la/s) − slicerem
3:      α := floor(tmp/hbs)
4:      return α × hbs + slicerem
5:  procedure ParallelizedHash(array, la)
6:      ls := ComputeSliceLen(la)
7:      lrem := la - (ls × s)
8:      for i := 0 to (s -1) do
9:          slice[i] = array[(i + 1) × ls − 1 : i × ls]
10:         X[i] = hash(slice[i])
11:     Y = array[la − 1: ls × s]
12:     YX= Y ∥ X[s − 1] ∥ X[s − 2] . . . ∥ X[0]
13:     return hash(YX)
```

3 Performance Analysis (2.B.2)

In this section, we discuss the performance of BIKE with respect to both latency and communication bandwidth. The performance numbers presented in sections 3.1, 3.2 and 3.3 refer to our reference code implementation, while section 3.4 refers to optimizations and their corresponding latency gains.

The platform used in the experiments was equipped with an Intel® Core™ i5-6260U CPU running at 1.80GHz. This platform has 32 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and 4,096K L3 cache. Intel® Turbo Boost and Intel®

Hyper-Threading technologies were all disabled. For each benchmark, the process was executed 25 times to warm-up the caches, followed by 100 iterations that were clocked (using the RDTSC instruction) and averaged. To minimize the effect of background tasks running on the system, each such experiment was repeated 10 times, and averaged. Our code was compiled using gcc/g++ 5.4.0 (build 20160609) with OpenSSL library (v1.0.2g, 1 Mar 2016) and NTL library (v6.2.1-1).

Regarding memory requirements, we remark that BIKE private keys are composed by $(h_0, h_1) \in \mathcal{R}$ with $|h_0| = |h_1| = w/2$. Each element can either be represented by (r) bits or, in a more compact way, by the $w/2$ non-zero positions, yielding a $(\frac{w}{2} \cdot \lceil \log_2(r) \rceil)$-bits representation.

3.1 Performance of BIKE-1

3.1.1 Memory Cost

Table 4 summarizes the memory required for each quantity.

Quantity	Size	Level 1	Level 3	Level 5
Private key	$w \cdot \lceil \log_2(r) \rceil$	$2,130$	$2,296$	$4,384$
Public key	n	$20,326$	$43,786$	$65,498$
Ciphertext	n	$20,326$	$43,786$	$65,498$

Table 4: Private Key, Public Key and Ciphertext Size in Bits.

3.1.2 Communication Bandwidth

Table 5 shows the bandwidth cost per message.

Message Flow	Message	Size	Level 1	Level 3	Level 5
Init. $\rightarrow$ Resp.	(f_0, f_1)	n	$20,326$	$43,786$	$65,498$
Resp. $\rightarrow$ Init.	(c_0, c_1)	n	$20,326$	$43,786$	$65,498$

Table 5: Communication Bandwidth in Bits.

3.1.3 Latency

Operation	Level 1	Level 3	Level 5
Key Generation	730,025	1,709,921	2,986,647
Encapsulation	689,193	1,850,425	3,023,816
Decapsulation	2,901,203	7,666,855	17,483,906

Table 6: Latency Performance in Number of Cycles.

3.2 Performance of BIKE-2

3.2.1 Memory Cost

Table 7 summarizes the memory required for each quantity.

Quantity	Size	Level 1	Level 3	Level 5
Private key	$w \cdot \lceil \log_2(r) \rceil$	2,130	3,296	4,384
Public key	r	10,163	21,893	32,749
Ciphertext	r	10,163	21,893	32,749

Table 7: Private Key, Public Key and Ciphertext Size in Bits.

3.2.2 Communication Bandwidth

Table 8 shows the bandwidth cost per message.

Message Flow	Message	Size	Level 1	Level 3	Level 5
Init. $\rightarrow$ Resp.	f_1	r	10,163	21,893	32,749
Resp. $\rightarrow$ Init.	c	r	10,163	21,893	32,749

Table 8: Communication Bandwidth in Bits.

3.2.3 Latency

Operation	Level 1	Level 3	Level 5
Key Generation	6,383,408	22,205,901	58,806,046
Encapsulation	281,755	710,970	1,201,161
Decapsulation	2,674,115	7,114,241	16,385,956

Table 9: Latency Performance in Number of Cycles.

3.3 Performance of BIKE-3

3.3.1 Memory Cost

Table 10 summarizes the memory required for each quantity.

Quantity	Size	Level 1	Level 3	Level 5
Private key	$w \cdot \lceil \log_2(r) \rceil$	2,010	3,168	4,522
Public key	n	22,054	43,366	72,262
Ciphertext	n	22,054	43,366	72,262

Table 10: Private Key, Public Key and Ciphertext Size in Bits.

3.3.2 Communication Bandwidth

Table 11 shows the bandwidth cost per message.

Message Flow	Message	Size	Level 1	Level 3	Level 5
Init. $\rightarrow$ Resp.	(f_0, f_1)	n	22,054	43,366	72,262
Resp. $\rightarrow$ Init.	(c_0, c_1)	n	22,054	43,366	72,262

Table 11: Communication Bandwidth in Bits.

3.3.3 Latency

Operation	Level 1	Level 3	Level 5
Key Generation	433,258	1,100,372	2,300,332
Encapsulation	575,237	1,460,866	3,257,675
Decapsulation	3,437,956	7,732,167	18,047,493

Table 12: Latency Performance in Number of Cycles.

3.4 Optimizations and Performance Gains

Optimizations for BIKE and corresponding performance gains are discussed next.

3.4.1 BIKE-2 Batch Key Generation

BIKE-2 key generation needs to compute a (costly) polynomial inversion, as described in Section 2.2. To reduce the impact of this costly operation and still benefit from the lower communication bandwidth offered by BIKE-2, we propose a *batch* version of BIKE-2 key generation. The main benefit of this approach is that only one polynomial inversion is computed for every N key generations, assuming a predefined $N \in \mathbb{N}$, instead of one inversion per key generation.

This technique is based on Montgomery's trick [32] and assumes that multiplication is fairly less expensive than inversion. As a toy example, suppose that one needs to invert two polynomials $x, y \in \mathcal{R}$. Instead of computing the inverse of each one separately, it is possible to compute them with one inversion and three multiplications: set $tmp = x \cdot y$, $inv = tmp^{-1}$ and then recover $x^{-1} = y \cdot inv$ and $y^{-1} = x \cdot inv$. This can be easily generalized to $N > 2$ polynomials: in this case, $2N$ multiplications are needed and inverses need to be recovered one at a time and in order. Because of this, our implementation requires the maintenance of a global variable $0 \leq \texttt{keyindex} < N$ that must be accessible only to the legitimate party willing to generate BIKE-2 keys and increased after each key generation. Algorithm 8 describes this optimization. Most of the work is done in the first key generation ($\texttt{keyindex} = 0$). In this way, the amortized cost of BIKE-2 key generation is reduced significantly as illustrated in Table 13.

Algorithm 8 `BIKE-2 Batch Key Generation`

Require: `keyindex`, $N \in \mathbb{N}$, code parameters (n, k, w)
Ensure: $(h_{0,0}, \ldots, h_{0,N-1}, h_1) \in \mathcal{R}^{N+1}$, $|h_{0,i}|_{0 \le i < N} = |h_1| = w$

1: Sample $h_1 \xleftarrow{\$} \mathcal{R}$ such that $|h_1| = w$
2: **if** $\texttt{keyindex} = 0$ **then**
3: Sample $h_{0,i} \xleftarrow{\$} \mathcal{R}$ such that $|h_{0,i}| = w$ for $0 < i < N$
4: $\texttt{prod}_{0,0} = \texttt{h}_{0,0}$
5: $\texttt{prod}_{0,\texttt{i}} = \texttt{prod}_{0,\texttt{i}-1} \cdot \texttt{h}_{0,\texttt{i}}$, for $1 \le i < N$
6: $\texttt{prod}_{1,\texttt{N}-1} = \texttt{prod}_{0,\texttt{N}-1}^{-1}$
7: $\texttt{prod}_{1,\texttt{i}} = \texttt{prod}_{1,\texttt{i}+1} \cdot \texttt{h}_{0,\texttt{i}+1}$, for $N - 2 \ge i > 0$
8: $inv = \texttt{prod}_{1,1} \cdot \texttt{h}_{0,1}$
9: **else**
10: $inv = \texttt{prod}_{1,\texttt{keyindex}} \cdot \texttt{prod}_{0,\texttt{keyindex}-1}$
11: $h \leftarrow h_1 \cdot inv$
12: $\texttt{keyindex} \leftarrow \texttt{keyindex} + 1$
13: **return** $(h_{0,\texttt{keyindex}}, h_1, h)$

Operation	Reference	Batch	Gain (%)
Level 1	6,383,408	1,647,843	74.18%
Level 3	22,205,901	4,590,452	79.32%
Level 5	58,806,046	9,296,144	84.19%

Table 13: Reference Versus Batch Key Generation (in cycles, for $N = 100$).

We stress that an implementer interested in the benefits offered by BIKE-2 batch key generation will need to meet the additional security requirements of protecting from adversaries and securely updating the variables `keyindex`, $prod_0$ and $prod_1$. It is also important to stress that the keys generated through this batch process are not related to each other. Finally, we remark that the use (or not) of the batch optimization does not impact on the encapsulation and decapsulation processes described in Section 2.2.

3.5 Additional Implementation

To illustrate the potential performance that BIKE code may achieve when running on modern platforms, we report some results of an additional implementation.

These preliminary BIKE-1 and BIKE-2 results can be expected to be further improved.

The performance is reported in processor cycles (lower is better), reflecting the performance per a *single core*. The results were obtained with the same measurement methodology declared in Section 3. The results are reported in Tables 14, 15, and 16 for BIKE-1, and in Tables 17, 18, and 19 for BIKE-2.

The additional implementation code. The core functionality was written in x86 assembly, and wrapped by assisting C code. The implementations use the PCLMULQDQ, AES−NI and the AVX2 and AVX512 architecture extensions. The code was compiled with gcc (version 5.4.0) in 64-bit mode, using the "O3" Optimization level, and run on a Linux (Ubuntu 16.04.3 LTS) OS. Details on the implementation and optimized components are provided in [16], and the underlying primitives are available in [19].

The benchmarking platform. The experiments were carried out on a platform equipped with the latest 8^{th} Generation Intel® Core^{TM} processor ("Kaby Lake") - Intel® Xeon® Platinum 8124M CPU at 3.00 GHz Core® $i5-750$. The platform has 70 GB RAM, 32K L1d and L1i cache, 1,024K L2 cache, and 25,344K L3 cache. It was configured to disable the Intel® Turbo Boost Technology, and the Enhanced Intel Speedstep® Technology.

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	0.09	0.11	1.13	0.20	0.15	5.30
AVX512	0.09	0.11	1.02	0.19	0.13	4.86

Table 14: Performance (in millions of cycles) of BIKE-1 Level 1.

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	11.99	0.27	2.70	12.45	0.39	10.74
AVX512	11.99	0.25	2.14	12.34	0.34	8.93

Table 19: Performance (in millions of cycles) of BIKE-2 Level 5.

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	0.25	0.28	3.57	0.45	0.36	16.74
AVX512	0.25	0.27	2.99	0.45	0.33	15.26

Table 15: Performance (in millions of cycles) of BIKE-1 Level 3.

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	0.25	0.29	2.75	0.67	0.42	9.84
AVX512	0.25	0.27	2.24	0.69	0.36	8.27

Table 16: Performance (in millions of cycles) of BIKE-1 Level 5.

4 Known Answer Values – KAT (2.B.3)

4.1 KAT for BIKE-1

The KAT files of BIKE-1 are available in:

- req file: `KAT/PQCkemKAT_BIKE1-Level1_2542.req`
- rsp file: `KAT/PQCkemKAT_BIKE1-Level1_2542.rsp`
- req file: `KAT/PQCkemKAT_BIKE1-Level3_4964.req`
- rsp file: `KAT/PQCkemKAT_BIKE1-Level3_4964.rsp`
- req file: `KAT/PQCkemKAT_BIKE1-Level5_8188.req`
- rsp file: `KAT/PQCkemKAT_BIKE1-Level5_8188.rsp`

4.2 KAT for BIKE-2

The KAT files of BIKE-2 are available in:

- req file: `KAT/PQCkemKAT_BIKE2-Level1_2542.req`

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	4.38	0.09	1.12	4.46	0.12	5.55
AVX512	4.38	0.08	0.86	4.45	0.11	5.12

Table 17: Performance (in millions of cycles) of BIKE-2 Level 1.

	—			Constant time implementation		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
AVX2	7.77	0.17	2.88	8.04	0.27	17.36
AVX512	7.79	0.18	3.48	8.05	0.23	15.63

Table 18: Performance (in millions of cycles) of BIKE-2 Level 3.

- rsp file: `KAT/PQCkemKAT_BIKE2-Level1_2542.rsp`
- req file: `KAT/PQCkemKAT_BIKE2-Level3_4964.req`
- rsp file: `KAT/PQCkemKAT_BIKE2-Level3_4964.rsp`
- req file: `KAT/PQCkemKAT_BIKE2-Level5_8188.req`
- rsp file: `KAT/PQCkemKAT_BIKE2-Level5_8188.rsp`

4.3 KAT for BIKE-3

The KAT files of BIKE-3 are available in:

- req file: `KAT/PQCkemKAT_BIKE3-Level1_2758.req`
- rsp file: `KAT/PQCkemKAT_BIKE3-Level1_2758.rsp`
- req file: `KAT/PQCkemKAT_BIKE3-Level3_5422.req`
- rsp file: `KAT/PQCkemKAT_BIKE3-Level3_5422.rsp`
- req file: `KAT/PQCkemKAT_BIKE3-Level5_9034.req`
- rsp file: `KAT/PQCkemKAT_BIKE3-Level5_9034.rsp`

5 Known Attacks (2.B.5)

This section discusses the practical security aspects of our proposal.

5.1 Hard Problems and Security Reduction

In the generic (*i.e.* non quasi-cyclic) case, the two following problems were proven NP-complete in [6].

Problem 1 (Syndrome Decoding – SD).
Instance: $H \in \mathbb{F}_2^{(n-k)\times n}$, $s \in \mathbb{F}_2^{n-k}$, *an integer* $t > 0$.
Property: *There exists* $e \in \mathbb{F}_2^n$ *such that* $|e| \leq t$ *and* $eH^T = s$.

Problem 2 (Codeword Finding – CF).
Instance: $H \in \mathbb{F}_2^{(n-k)\times n}$, *an integer* $t > 0$.
Property: *There exists* $c \in \mathbb{F}_2^n$ *such that* $|c| = t$ *and* $cH^T = 0$.

In both problems the matrix H is the parity check matrix of a binary linear $[n, k]$ code. Problem 1 corresponds to the decoding of an error of weight t and Problem 2 to the existence of a codeword of weight t. Both are also conjectured to be hard on average. This is argued in [1], together with results which indicate that the above problems remain hard even when the weight is very small, i.e. $t = n^{\varepsilon}$, for any $\varepsilon > 0$. Note that all known solvers for one of the two problems also solve the other and have a cost exponential in t.

5.1.1 Hardness for QC codes.

Coding problems (SD and CF) in a QC-code are NP-complete, but the result does not hold for when the index is fixed. In particular, for $(2, 1)$-QC codes or $(3, 1)$-QC codes, which are of interest to us, we do not know whether or not SD and CF are NP-complete.

Nevertheless, they are believed to be hard on average (when r grows) and the best solvers in the quasi-cyclic case have the same cost as in the generic case up to a small factor which never exceeds the order r of quasi-cyclicity. The problems below are written in the QC setting, moreover we assume that the parity check matrix H is in systematic form, that is the first $(n_0 - k_0) \times (n_0 - k_0)$ block of H is the identity matrix. For instance, for $(2, 1)$-QC and $(3, 1)$-QC codes codes, the parity check matrix (over $\mathcal{R}$) respectively have the form

$$\begin{pmatrix} 1 & h \end{pmatrix} \text{ with } h \in \mathcal{R}, \text{ and } \begin{pmatrix} 1 & 0 & h_0 \\ 0 & 1 & h_1 \end{pmatrix} \text{ with } h_0, h_1 \in \mathcal{R}.$$

In our case, we are interested only by those two types of QC codes and to the three related hard problems below:

Problem 3 ($(2,1)$-QC Syndrome Decoding – $(2,1)$-QCSD).
Instance: s, h *in* $\mathcal{R}$, *an integer* $t > 0$.
Property: *There exists* e_0, e_1 *in* $\mathcal{R}$ *such that* $|e_0| + |e_1| \leq t$ *and* $e_0 + e_1 h = s$.

Problem 4 ($(2,1)$-QC Codeword Finding – $(2,1)$-QCCF).
Instance: h *in* $\mathcal{R}$, *an integer* $t > 0$.
Property: *There exists* c_0, c_1 *in* $\mathcal{R}$ *such that* $|c_0| + |c_1| = t$ *and* $c_0 + c_1 h = 0$.

Problem 5 ($(3,1)$-QC Syndrome Decoding – $(3,1)$-QCSD).
Instance: s_0, s_1, h_0, h_1 *in* $\mathcal{R}$, *an integer* $t > 0$.
Property: *There exists* e_0, e_1, e_2 *in* $\mathcal{R}$ *such that* $|e_0| + |e_1| + |e_2| \leq t$, $e_0 + e_2 h_0 = s_0$ *and* $e_1 + e_2 h_1 = s_1$.

As they are presented, those problems have the appearance of *sparse polynomials* problem, but in fact they are equivalent to generic quasi-cyclic decoding and codeword finding problems.

In the current state of the art, the best known techniques for solving those problems are variants of Prange's Information Set Decoding (ISD) [33]. We remark that, though the best attacks consist in solving one of the search problems, the security reduction of our scheme requires the decision version of Problem 2.

5.2 Information Set Decoding

The best asymptotic variant of ISD is due to May and Ozerov [29], but it has a polynomial overhead which is difficult to estimate precisely. In practice, the BJMM variant [5] is probably the best for relevant cryptographic parameters. The work factor for classical (*i.e.* non quantum) computing of any variant $\mathcal{A}$ of ISD for decoding t errors (or finding a word of weight t) in a binary code of length n and dimension k can be written

$$\mathrm{WF}_{\mathcal{A}}(n,k,t) = 2^{ct(1+o(1))}$$

where c depends on the algorithm, on the code rate $R = k/n$ and on the error rate t/N. It has been proven in [35] that, asymptotically, for sublinear weight $t = o(n)$ (which is the case here as $w \approx t \approx \sqrt{n}$), we have $c = \log_2 \frac{1}{1-R}$ for all variants of ISD.

In practice, when t is small, using 2^{ct} with $c = \log_2 \frac{1}{1-R}$ gives a remarkably good estimate for the complexity. For instance, non asymptotic estimates derived

from [22] gives $\mathrm{WF}_{\mathrm{BJMM}}(65542, 32771, 264) = 2^{263.3}$ "column operations" which is rather close to 2^{264}. This closeness is expected asymptotically, but is circumstantial for fixed parameters. It only holds because various factors compensate, but it holds for most MDPC parameters of interest.

5.2.1 Exploiting the Quasi-Cyclic Structure.

Both codeword finding and decoding are a bit easier (by a polynomial factor) when the target code is quasi-cyclic. If there is a word of weight w in a QC code then its r quasi-cyclic shifts are in the code. In practice, this gives a factor r speedup compared to a random code. Similarly, using Decoding One Out of Many (DOOM) [34] it is possible to produce r equivalent instances of the decoding problem. Solving those r instances together saves a factor $\sqrt{r}$ in the workload.

5.2.2 Exploiting Quantum Computations.

Recall first that the NIST proposes to evaluate the quantum security as follows:

1. A quantum computer can only perform quantum computations of limited depth. They introduce a parameter, MAXDEPTH, which can range from 2^{40} to 2^{96}. This accounts for the practical difficulty of building a full quantum computer.

2. The amount (or bits) of security is not measured in terms of absolute time but in the time required to perform a specific task.

Regarding the second point, the NIST presents 6 security categories which correspond to performing a specific task. For example Task 1, related to Category 1, consists of finding the 128 bit key of a block cipher that uses AES-128. The security is then (informally) defined as follows:

Definition 5. *A cryptographic scheme is secure with respect to Category k iff. any attack on the scheme requires computational resources comparable to or greater than those needed to solve Task k.*

In what follows we will estimate that our scheme reaches a certain security level according to the NIST metric and show that the attack takes more quantum resources than a quantum attack on AES. We will use for this the following proposition.

Proposition 1. *Let f be a Boolean function which is equal to 1 on a fraction α of inputs which can be implemented by a quantum circuit of depth D_f and whose gate complexity is C_f. Using Grover's algorithm for finding an input x of f for which*

$f(x) = 1$ can not take less quantum resources than a Grover's attack on AES-N as soon as

$$\frac{D_f \cdot C_f}{\alpha} \geq 2^N D_{AES-N} \cdot C_{AES-N}$$

where D_{AES-N} and C_{AES-N} are respectively the depth and the complexity of the quantum circuit implementing AES-N.

This proposition is proved in Section B of the appendix. The point is that (essentially) the best quantum attack on our scheme consists in using Grover's search on the information sets computed in Prange's algorithm (this is Bernstein's algorithm [7]). Theoretically there is a slightly better algorithm consisting in quantizing more sophisticated ISD algorithms [23], however the improvement is tiny and the overhead in terms of circuit complexity make Grover's algorithm used on top of the Prange algorithm preferable in our case.

5.3 Defeating the GJS Reaction Attack

BIKE uses an ephemeral KEM key pair, i.e. a KEM key generation is performed for each key exchange. As a result, the GJS reaction attack is inherently defeated: a GJS adversary would have (at most) a single opportunity to observe decryption, thus not being able to create statistics about different error patterns. We note that, for efficiency purposes, an initiator may want to precompute KEM key pairs before engaging in key exchange sessions. We remark that policies to securely store the pregenerated KEM key pair must be in place, in order to avoid that an adversary access a KEM key pair to be used in a future communication.

5.4 Choice of Parameters

We denote $\mathrm{WF}(n, k, t)$ the workfactor of the best ISD variant for decoding t errors in a binary code of length n and dimension k. In the following we will consider only codes of transmission rate 0.5, that is length $n = 2r$ and dimension r. In a classical setting, the best solver for Problem 3 has a cost $\mathrm{WF}(2r, r, t)/\sqrt{r}$, the best solver for Problem 4 has a cost $\mathrm{WF}(2r, r, w)/r$, and the best solver for Problem 5 has a cost $\mathrm{WF}(3r, r, 3t/2)/\sqrt{r}$. As remarked above, with $\mathrm{WF}(n, k, \ell) \approx 2^{\ell \log_2 \frac{n}{n-k}}$ we obtain a crude but surprisingly accurate, parameter selection rule. We target security levels corresponding to AES λ with $\lambda \in \{128, 192, 256\}$. To reach λ bits of classical security, we choose w, t and r such that

- for BIKE-1 and BIKE-2, Problem 3 with block size r and weight t and Problem 4 with block size r and weight w must be hard enough, that is

$$\lambda \approx t - \frac{1}{2}\log_2 r \approx w - \log_2 r. \tag{2}$$

- for BIKE-3, Problem 5 with block size r and weight $3t/2$ and Problem 3 with block size r and weight w must be hard enough, that is

$$\lambda \approx \frac{3t}{2}\log_2\frac{3}{2} - \frac{1}{2}\log_2 r \approx w - \frac{1}{2}\log_2 r. \tag{3}$$

Those equation have to be solved in addition with the constraint that r must be large enough to decode t errors in $(2, 1, r, w)$-QC-MDPC code with a negligible failure rate. Finally, we choose r such that 2 is primitive modulo r. First, this will force r to be prime, thwarting the so-called squaring attack [25]. Also, it implies that $(X^r - 1)$ only has two irreducible factors (one of them being $X - 1$). This is an insurance against an adversary trying to exploit the structure of $\mathbb{F}_2[X]/\langle X^r - 1\rangle$ when $(X^r - 1)$ has small factors, other than $(X - 1)$. This produces the parameters proposed in the document.

The quantum speedup is at best quadratic for the best solvers of the problems on which our system, from the arguments of §5.2.2, it follows our set of parameters correspond the security levels 1, 3, and 5 described in the NIST call for quantum safe primitives.

6 Formal Security (2.B.4)

6.1 IND-CPA Security

We start with the following definition, where we denote by $\mathcal{K}$ the domain of the exchanged symmetric keys and by λ the security level of the scheme.

Definition 6. *A key-encapsulation mechanism is IND-CPA (passively) secure if the outputs of the two following games are computationally indistinguishable.*

Game $\mathbf{G}_{real}$	**Game $\mathbf{G}_{fake}$**
$(sk, pk) \leftarrow \textsc{Gen}(\lambda)$	$(sk, pk) \leftarrow \textsc{Gen}(\lambda)$
$(c, K) \leftarrow \textsc{Encaps}(pk)$	$(c, K) \leftarrow \textsc{Encaps}(pk)$
	$K^* \xleftarrow{\$} \mathcal{K}$
Output (pp, pk, c, K)	Output (pp, pk, c, K^*)

Rather than analyzing all three variants of BIKE separately, we state a single theorem, and highlight the differences in the proof.

Theorem 2. *BIKE is IND-CPA secure in the Random Oracle Model under the* $(2,1)$*-QCCF,* $(2,1)$*-QCSD and* $(3,1)$*-QCSD assumptions.*

Proof. To begin, note that we model the hash function $\mathbf{K}$ as a random oracle. The goal of our proof is to prove that an adversary distinguishing one game from another can be exploited to break one or more of the problems above in polynomial time (see Section 5.1 for definitions). Let $\mathcal{A}$ be a probabilistic polynomial time adversary against the IND-CPA of our scheme and consider the following games where we consider that $\mathcal{A}$ receives the encapsulation at the end of each game.

Game G_1: This corresponds to an honest run of the protocol, and is the same as Game $\mathbf{G}_{real}$. In particular, the simulator has access to all keys and randomness.

Game G_2: In this game, the simulator picks uniformly at random the public key, specifically (f_0, f_1) for BIKE-1 and BIKE-3, and h for BIKE-2. The rest of the game then proceeds honestly.

An adversary distinguishing between these two games is therefore able to distinguish between a well-formed public key and a randomly-generated one. Note that the public key in $\boldsymbol{G}_1$ corresponds to a valid $(2,1)$-QCCF instance for BIKE-1 and BIKE-2, and to a $(2,1)$-QCSD instance for BIKE-3, while it is random in $\boldsymbol{G}_2$. Thus we have respectively

$$\mathrm{Adv}^{G_1-G_2}(\mathcal{A}) \leq \mathrm{Adv}^{(2,1)\text{-QCCF}}(\mathcal{A}')$$

and

$$\mathrm{Adv}^{G_1-G_2}(\mathcal{A}) \leq \mathrm{Adv}^{(2,1)\text{-QCSD}}(\mathcal{A}')$$

where $\mathcal{A}'$ is a polynomial time adversary for the $(2,1)$-QCCF/ $(2,1)$-QCSD problem.

Game G_3: Now, the simulator also picks uniformly at random the ciphertext: again, this is (c_0, c_1) for BIKE-1 and BIKE-3, and c for BIKE-2. The encapsulated key K is still generated honestly.

If an adversary is able to distinguish game $\boldsymbol{G}_2$ from game $\boldsymbol{G}_3$, then it can solve one of the QCSD problems.
In fact, for BIKE-2, the ciphertext is exactly a syndrome that follows the $(2,1)$-QCSD distribution in game $\boldsymbol{G}_2$ and the uniform distribution in $\boldsymbol{G}_3$. The same can be easily shown for BIKE-1. Thus for both variants we have

$$\mathrm{Adv}^{G_2-G_3}(\mathcal{A}) \leq \mathrm{Adv}^{(2,1)\text{-QCSD}}(\mathcal{A}'')$$

where $\mathcal{A}''$ is a polynomial time adversary for the $(2,1)$-QCSD problem.

As we will see, a similar situation occurs for BIKE-3. In fact, the adversary has access to:

$$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & f_0 \\ 0 & 1 & f_1 \end{pmatrix} (e, e_0, e_1)^\top$$

Here (c_0, c_1) follows the $(3,1)$-QCSD distribution in game $\boldsymbol{G}_2$ and the uniform distribution over $(\mathbb{F}_2^n)^2$ in $\boldsymbol{G}_3$. Hence

$$\mathrm{Adv}^{G_2-G_3}(\mathcal{A}) \leq \mathrm{Adv}^{(3,1)\text{-QCSD}}(\mathcal{A}'')$$

where $\mathcal{A}''$ is a polynomial time adversary for the $(3,1)$-QCSD problem.

Game $\boldsymbol{G}_4$: Finally, we replace the value of K with a uniformly random value K^*. Since $\mathbf{K}$ is modeled as a random oracle, its output is pseudorandom, and an adversary only has negligible advantage ϵ, so for all three variants

$$\mathrm{Adv}^{G_3-G_4}(\mathcal{A}) \leq \epsilon.$$

Thus in the end we have

$$\mathrm{Adv}^{\text{IND-CPA}}(\mathcal{A}) \leq \mathrm{Adv}^{(2,1)\text{-QCCF}}(\mathcal{A}') + \mathrm{Adv}^{(2,1)\text{-QCSD}}(\mathcal{A}'') + \epsilon. \tag{4}$$

or

$$\mathrm{Adv}^{\text{IND-CPA}}(\mathcal{A}) \leq \mathrm{Adv}^{(2,1)\text{-QCSD}}(\mathcal{A}') + \mathrm{Adv}^{(3,1)\text{-QCSD}}(\mathcal{A}'') + \epsilon. \tag{5}$$

respectively for BIKE-1/BIKE-2 and BIKE-3. □

6.2 Public Keys and Subcodes

In this section, we prove that one can efficiently sample an *invertible* element from $\mathbb{F}_2[X]/\langle X^r-1\rangle$ by taking any polynomial $h \xleftarrow{\$} \mathbb{F}_2[X]/\langle X^r-1\rangle$ such that $|h|$ is odd. If this element was not invertible, the public code produced in BIKE-1 and BIKE-3 would be a subcode of the private one.

Lemma 1. *Let $h \in \mathbb{F}_2[X]$ have even weight. Then h is not invertible modulo X^r-1.*

Proof. We show that $(X-1) \mid h$ by induction on $|h|$. For $|h|=0$ trivially $(X-1) \mid h$. Assume that $(X-1) \mid h$ whenever $|h|=2k$ for some $k \geqslant 0$. Now consider any $h \in \mathbb{F}_2[X]$ with weight $|h|=2(k+1)$, and take two distinct terms X^i, X^j of h such that $i<j$. Define $h'=h-X^i-X^j$, so that $|h'|=2k$. Then $(X-1) \mid h'$ by induction, i.e. $h'=(X-1)h''$ for some $h'' \in \mathbb{F}_2[X]$. Hence $h=h'+X^i+X^j = (X-1)h''+X^i(X^{j-i}+1) = (X-1)h''+X^i(X-1)(X^{j-i-1}+\cdots+1) = (X-1)(h''+X^i(X^{j-i-1}+\cdots+1))$, and therefore $(X-1) \mid h$. □

Theorem 3. *Let r a prime such that $(X^r-1)/(X-1) \in \mathbb{F}_2[X]$ is irreducible. Then any $h \in \mathbb{F}_2[X]$ with $\deg(h)<r$ is invertible modulo X^r-1 iff $h \neq X^{r-1}+\cdots+1$ and $|h|$ is odd.*

Proof. Take a term X^i of h. Then $\left|h+X^i\right| = |h|-1$ is even, and by Lemma 1 $(X-1) \mid (h+X^i)$. Hence $h \bmod (X-1) = X^i \bmod (X-1) = 1$, meaning that h is invertible modulo $X-1$.

Now, since $(X^r-1)/(X-1) = X^{r-1}+\cdots+1$ is irreducible, if $\deg(h)<r-1$ then $\gcd(h, X^{r-1}+\cdots+1)=1$, and if $\deg(h)=r-1$, then $\gcd(h, X^{r-1}+\cdots+1) = \gcd(h+X^{r-1}+\cdots+1, X^{r-1}+\cdots+1)=1$, since $\deg(h+X^{r-1}+\cdots+1)<r-1$. Hence h is invertible modulo $X^{r-1}+\cdots+1$.

Therefore, the combination of the inverses of h modulo $X-1$ and modulo $X^{r-1}+\cdots+1$ via the Chinese remainder theorem is well defined, and by construction it is the inverse of h modulo $(X-1)(X^{r-1}+\cdots+1) = X^r-1$. □

Corollary 1. *One can efficiently sample an invertible element from $\mathbb{F}_2[X]/\langle X^r-1\rangle$ by taking any polynomial $h \xleftarrow{\$} \mathbb{F}_2[X]/\langle X^r-1\rangle$ such that $|h|$ is odd.*

7 Advantages and Limitations (2.B.6)

This document presents BIKE, a suite of IND-CPA secure key encapsulation mechanisms (KEM) composed by BIKE-1, BIKE-2 and BIKE-3. Each variant has its own pros and cons.

In common, all BIKE variants are based on quasi-cyclic moderate density parity-check (QC-MDPC codes), which can be efficiently decoded through bit flipping decoding techniques. This kind of decoder is extremely simple: it estimates what are the positions most likely in error, flip them and observes whether the result is better (smaller syndrome weight) than before or not. This process converges very quickly; in particular, Section 2.5 presents a 1-iteration bit flipping decoder.

Another characteristic in common to all BIKE variants is the fact that they rely on ephemeral keys. This leads to two things: at first, it inherently defeats the GJS reaction attack mentioned in section 5, which is an attack that needs to observe a large number of decodings for a same private key (something impossible when ephemeral keys are used). The other aspect of this choice is that key generation must be efficient since it is executed at every key encapsulation. Previous works based on QC-MDPC codes compute a polynomial inversion operation in order to obtain a QC-MDPC public key in systematic form. The polynomial inversion is an expensive operation. BIKE-1 completely avoids the polynomial inversion by not relying on public keys in systematic form. Instead, it hides the private sparse structure by multiplying it by a dense polynomial of odd weight sampled uniformly at random. This leads to an increased public key size but results in a very efficient key generation process (it becomes the fastest process among key generation, encapsulation and decapsulation operations). BIKE-2 uses public keys in systematic form, but thanks to our batch key generation technique discussed in Section 3.4, the amortized cost can decrease up to 84%, becoming less expensive than the bit flipping decoder. Besides the bit flipping algorithm and the eventual polynomial inversion (restricted to BIKE-2), all other operations in the BIKE suite consist of simple products of binary vectors, an operation that can be easily optimized for all sorts of hardware and software applications.

Regarding communication bandwidth, in BIKE-1 and BIKE-3 all public keys, private keys and cryptograms are n bits long, corresponding to the bandwidth of the messages exchanged by the parties. BIKE-2 offers smaller public keys and ciphertexts, r bits only, corresponding to the bandwidth of the messages exchanged by the parties as well. Two messages are exchanged per key encapsulation of same size (either n or r bits). In practice, these numbers range from 1.24 KB per message in BIKE-2 security level 1, up to 8.82 KB per message in BIKE-3 security level 5. These numbers seem fairly reasonable when compared to the the average size of a page in the Internet (currently near 2MB [2]), just as an example.

Regarding security, all BIKE variants rely their security on very well-known coding-theory problems: quasi-cyclic syndrome decoding and quasi-cyclic codeword finding problems. The best strategies to solve these problems are based on

Information Set Decoding (ISD) techniques, a research field that has a very long history (Prange's seminal work dates back 1962) and which has seem very little improvement along the years. Moreover, we show that in the quantum setting, Grover's algorithm used on top of the seminal Prange ISD algorithm is still the most preferable choice in our case.

One point of attention in BIKE is the fact that, nowadays, the bit flipping decoding techniques do not attain a negligible decoding failure rate. This makes it challenge to achieve higher security notions such as IND-CCA. This may also limits the usage of BIKE in certain applications such as, for instance, Hybrid Encryption, where both KEM and DEM need to satisfy IND-CCA security to guarantee chosen-ciphertext security for the hybrid encryption scheme. We stress however that it seems possible (although not simple) to prove that certain decoding techniques can in fact attain negligible decoding failure rates for QC-MDPC codes.

Regarding intellectual property, to the best of our knowledge, BIKE-1 and BIKE-2 are not covered by any patent. BIKE-3 is covered by a patent whose owners are willing to grant a non-exclusive license for the purpose of implementing the standard *without compensation* and under reasonable terms and conditions that are demonstrably free of any unfair discrimination, as denoted in the accompanying signed statements. We emphasize that BIKE-1 and BIKE-2 are **not covered** by the aforementioned patent, and that the BIKE team is willing to drop BIKE-3 if this ever becomes a disadvantage when comparing our suite with other proposals.

Overall, taking all these considerations into account, we believe that BIKE is a promising candidate for post-quantum key exchange standardization.

8 Acknowledgments

Shay Gueron, Tim Güneysu, Nicolas Sendrier and Jean-Pierre Tillich were supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO). Shay Gueron was also partially supported by the Israel Science Foundation (grant No. 1018/16). Paulo S. L. M. Barreto was partially supported by Intel and FAPESP through the project "Efficient Post-Quantum Cryptography for Building Advanced Security Applications" (grant No. 2015/50520-6). The logo presented in the cover page was designed by Szilard Nagy. The reference code was developed by Nir Druker, Shay Gueron, Rafael Misoczki, Tim Güneysu, Tobias Oder and Slim Bettaieb.

References

[1] Michael Alekhnovich. More on average case vs approximation complexity. In *FOCS 2003*, pages 298–307. IEEE, 2003.

[2] HTTP Archive. Http archive report, 2017. `http://httparchive.org/trends.php`.

[3] Elaine B Barker and John Michael Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2012.

[4] Paulo S. L. M. Barreto, Shay Gueron, Tim Guneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, and Jean-Pierre Tillich. CAKE: Code-based Algorithm for Key Encapsulation. Cryptology ePrint Archive, Report 2017/757, 2017. `https://eprint.iacr.org/2017/757.pdf`. To appear in the 16th IMA International Conference on Cryptography and Coding.

[5] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How 1+1=0 improves information set decoding. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 520–536. Springer, 2012.

[6] Elwyn Berlekamp, Robert J. McEliece, and Henk van Tilborg. On the inherent intractability of certain coding problems (corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978.

[7] Daniel J Bernstein. Grover vs. McEliece. In *International Workshop on Post-Quantum Cryptography*, pages 73–80. Springer, 2010.

[8] Céline Blondeau, Benoît Gérard, and Jean-Pierre Tillich. Accurate estimates of the data complexity and success probability for various cryptanalyses. *Des. Codes Cryptogr.*, 59(1-3):3–34, 2011.

[9] Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. Efficient implementation of a cca2-secure variant of McEliece using generalized Srivastava codes. In *Proceedings of PKC 2012, LNCS 7293, Springer-Verlag*, pages 138–155, 2012.

[10] Julia Chaulet. *Étude de cryptosystèmes à clé publique basés sur les codes MDPC quasi-cycliques*. Thèse de doctorat, University Pierre et Marie Curie, March 2017.

[11] Julia Chaulet and Nicolas Sendrier. Worst case QC-MDPC decoder for McEliece cryptosystem. In *Information Theory (ISIT), 2016 IEEE International Symposium on*, pages 1366–1370. IEEE, 2016.

[12] Tung Chou. Qcbits: Constant-time small-key code-based cryptography. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 280–300. Springer, 2016.

[13] Thomas M. Cover and Joy A. Thomas. *Information Theory*. Wiley Series in Telecommunications. Wiley, 1991.

[14] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1):167–226, January 2004.

[15] Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Ouroboros: A simple, secure and efficient key exchange protocol based on coding theory. In Tanja Lange and Tsuyoshi Takagi, editors, *PQCrypto 2017*, volume 10346 of *LNCS*, pages 18–34. Springer, 2017.

[16] Nir Drucker and Shay Gueron. A toolbox for software optimization of qc-mdpc code-based cryptosystems. Cryptology ePrint Archive, December 2017. `http://eprint.iacr.org/`.

[17] Shay Gueron. A j-lanes tree hashing mode and j-lanes SHA-256. *Journal of Information Security*, 4(01):7, 2013.

[18] Shay Gueron. Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. *Journal of Information Security*, 5(03):91, 2014.

[19] Shay Gueron. A-toolbox-for-software-optimization-of-qc-mdpc-code-based-cryptosystems, 2017. `https://github.com/Shay-Gueron/A-toolbox-for-software-optimization-of-QC-MDPC-code-based-cryptosystems`.

[20] Shay Gueron and Vlad Krasnov. Simultaneous hashing of multiple messages. *Journal of Information Security*, 3(04):319, 2012.

[21] Qian Guo, Thomas Johansson, and Paul Stankovski. *A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors*, pages 789–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[22] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. Cryptology ePrint Archive, Report 2013/162, 2013. `http://eprint.iacr.org/2013/162`.

[23] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. In Tanja Lange and Tsuyoshi Takagi, editors, *PQCrypto 2017*, volume 10346 of *LNCS*, pages 69–89. Springer, 2017.

[24] Gil Kalai and Nathan Linial. On the distance distribution of codes. *IEEE Trans. Inform. Theory*, 41(5):1467–1472, September 1995.

[25] Carl Löndahl, Thomas Johansson, Masoumeh Koochak Shooshtari, Mahmoud Ahmadian-Attari, and Mohammad Reza Aref. Squaring attacks on McEliece public-key cryptosystems using quasi-cyclic codes of even dimension. *Designs, Codes and Cryptography*, 80(2):359–377, 2016.

[26] Florence J. MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes*. North–Holland, Amsterdam, fifth edition, 1986.

[27] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology - EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397, Lofthus, Norway, May 1993. Springer.

[28] Ingo Von Maurich, Tobias Oder, and Tim Güneysu. Implementing qc-mdpc mceliece encryption. *ACM Trans. Embed. Comput. Syst.*, 14(3):44:1–44:27, April 2015.

[29] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 203–228. Springer, 2015.

[30] Daniele Micciancio. Improving lattice based cryptosystems using the hermite normal form. *Cryptography and lattices*, pages 126–145, 2001.

[31] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. L.S.M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *IEEE International Symposium on Information Theory – ISIT'2013*, pages 2069–2073, Istambul, Turkey, 2013. IEEE.

[32] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[33] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions*, IT-8:S5–S9, 1962.

[34] Nicolas Sendrier. Decoding one out of many. In B.-Y. Yang, editor, *PQCrypto 2011*, volume 7071 of *LNCS*, pages 51–67. Springer, 2011.

[35] Rodolfo Canto Torres and Nicolas Sendrier. Analysis of information set decoding for a sub-linear error weight. In Tsuyoshi Takagi, editor, *PQCrypto 2016*, volume 9606 of *LNCS*, pages 144–161. Springer, 2016.

[36] Christof Zalka. Grover's quantum searching algorithm is optimal. *Phys. Rev. A*, 60:2746–2751, October 1999.

A Proof of Theorem 1

Let us recall the theorem we want to prove.

Theorem 1. *Under assumption 1, the probability P_{err} that the bit flipping algorithm fails to decode with fixed threshold $\tau = \frac{1}{2}$ is upper-bounded by*

$$P_{err} \leq \frac{1}{\sqrt{\alpha \pi t}} e^{\frac{\alpha t w}{8} \ln\left(1-\varepsilon^2\right) + \frac{\alpha t}{8} \ln(n) + O(t)},$$

where $\varepsilon \stackrel{def}{=} e^{-\frac{2wt}{n}}$.

We will denote in the whole section by $h(x)$ the entropy (in nats) of a Bernoulli random variable of parameter x, that is $h(x) \stackrel{\text{def}}{=} -x \ln x - (1-x)\ln(1-x)$.

A.1 Basic tools

A particular quantity will play a fundamental role here, the Kullback-Leibler divergence (see for instance [13])

Definition 7. Kullback-Leibler divergence
Consider two discrete probability distributions $\mathbf{p}$ and $\mathbf{q}$ defined over a same discrete space $\mathcal{X}$. The Kullback-Leibler divergence between $\mathbf{p}$ and $\mathbf{q}$ is defined by

$$D(\mathbf{p} \| \mathbf{q}) = \sum_{x \in \mathcal{X}} p(x) \ln \frac{p(x)}{q(x)}.$$

We overload this notation by defining for two Bernoulli distributions $\mathcal{B}(p)$ and $\mathcal{B}(q)$ of respective parameters p and q

$$D(p\|q) \stackrel{def}{=} D(\mathcal{B}(p)\|\mathcal{B}(q)) = p \ln\left(\frac{p}{q}\right) + (1-p)\ln\left(\frac{1-p}{1-q}\right).$$

We use the convention (based on continuity arguments) that $0 \ln \frac{0}{p} = 0$ and $p \ln \frac{p}{0} = \infty$.

We will need the following approximations/results of the Kullback-Leibler divergence

Lemma 2. *For any $\delta \in (-1/2, 1/2)$ we have*

$$D\left(\frac{1}{2} \middle\| \frac{1}{2} + \delta\right) = -\frac{1}{2}\ln(1 - 4\delta^2). \tag{6}$$

For constant $\alpha \in (0,1)$ and δ going to 0 by staying positive, we have

$$D(\alpha\|\delta) = -h(\alpha) - \alpha \ln \delta + O(\delta). \tag{7}$$

For $0 < y < x$ and x going to 0 we have

$$D(x\|y) = x \ln \frac{x}{y} + x - y + O\left(x^2\right). \tag{8}$$

Proof. Let us first prove (6).

$$\begin{aligned} D\left(\frac{1}{2}\middle\|\frac{1}{2}+\delta\right) &= \frac{1}{2}\ln\frac{1/2}{1/2+\delta} + \frac{1}{2}\ln\frac{1/2}{1/2-\delta} \\ \mathbb{P} &= -\frac{1}{2}\ln(1+2\delta) - \frac{1}{2}\ln(1-2\delta) \\ &= -\frac{1}{2}\ln(1-4\delta^2). \end{aligned}$$

To prove (7) we observe that

$$\begin{aligned} D(\alpha\|\delta) &= \alpha\ln\left(\frac{\alpha}{\delta}\right) + (1-\alpha)\ln\left(\frac{1-\alpha}{1-\delta}\right) \\ &= -h(\alpha) - \alpha\ln\delta - (1-\alpha)\ln(1-\delta) \\ &= -h(\alpha) - \alpha\ln\delta + O(\delta). \end{aligned}$$

For the last estimate we proceed as follows

$$\begin{aligned} D(x\|y) &= x\ln\frac{x}{y} + (1-x)\ln\frac{1-x}{1-y} \\ &= x\ln\frac{x}{y} - (1-x)\left(-x+y+O\left(x^2\right)\right) \\ &= x\ln\frac{x}{y} + x - y + O\left(x^2\right). \end{aligned}$$

□

The Kullback-Leibler appears in the computation of large deviation exponents. In our case, we will use the following estimate which is well known and which can be found for instance in [8]

Lemma 3. *Let p be a real number in $(0,1)$ and $X_1, \ldots X_n$ be n independent Bernoulli random variables of parameter p. Then, as n tends to infinity:*

$$\mathbb{P}(X_1 + \ldots X_n \geq \tau n) = \frac{(1-p)\sqrt{\tau}}{(\tau-p)\sqrt{2\pi n(1-\tau)}} e^{-nD(\tau\|p)}(1+o(1)) \text{ for } p < \tau < 1, \tag{9}$$

$$\mathbb{P}(X_1 + \ldots X_n \leq \tau n) = \frac{p\sqrt{1-\tau}}{(p-\tau)\sqrt{2\pi n\tau}} e^{-nD(\tau\|p)}(1+o(1)) \text{ for } 0 < \tau < p. \tag{10}$$

A.2 Estimation of the probability that a parity-check equation of weight w gives an incorrect information

A.2.1 Main result

We start our computation by computing the probability that a parity-check equation gives an incorrect information about a bit. We say here that a parity-check equation $\mathbf{h}$ (viewed as a binary word) gives an incorrect information about an error bit e_i that is involved in $\mathbf{h}$ if $\langle \mathbf{h}, \mathbf{e} \rangle \neq e_i$, where $\mathbf{e}$ is the error. This is obtained through the following lemma.

Lemma 4. *Consider a word $\mathbf{h} \in \mathbb{F}_2^n$ of weight w and an error $\mathbf{e} \in \mathbb{F}_2^n$ of weight t chosen uniformly at random. Assume that both w and t are of order $\sqrt{n}$: $w = \Theta(\sqrt{n})$ and $t = \Theta(\sqrt{n})$. We have*

$$\mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1) = \frac{1}{2} - \frac{1}{2} e^{-\frac{2wt}{n}} \left(1 + O\left(\frac{1}{\sqrt{n}}\right)\right).$$

Remark 2. *Note that this probability is in this case of the same order as the probability taken over errors $\mathbf{e}$ whose coordinates are drawn independently from a Bernoulli distribution of parameter t/n. In such a case, from the piling-up lemma [27] we have*

$$\begin{aligned} \mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1) &= \frac{1 - \left(1 - \frac{2t}{n}\right)^w}{2} \\ &= \frac{1}{2} - \frac{1}{2} e^{w \ln(1 - 2t/n)} \\ &= \frac{1}{2} - \frac{1}{2} e^{-\frac{2wt}{n}} \left(1 + O\left(\frac{1}{\sqrt{n}}\right)\right). \end{aligned}$$

Let us bring now the following fundamental quantities for $b \in \{0, 1\}$

$$p_b \stackrel{\text{def}}{=} \mathbb{P}(\langle \mathbf{h}, \mathbf{e} \rangle = 1 | e_1 = b) \tag{11}$$

where without loss of generality we assume that $h_1 = 1$ and $\mathbf{e}$ is an error of weight t and length n chosen uniformly at random.

The proof of this lemma will be done in the following subsection. From this lemma it follows directly that

Corollary 2. *Assume that $w = \Theta(\sqrt{n})$ and $t = \Theta(\sqrt{n})$. Then*

$$p_b = \frac{1}{2} - (-1)^b \varepsilon \left(\frac{1}{2} + O\left(\frac{1}{\sqrt{n}}\right)\right), \tag{12}$$

where $\varepsilon \stackrel{def}{=} e^{-\frac{2wt}{n}}$.

A.2.2 Proof of Lemma 4

The proof involves properties of the Krawtchouk polynomials. We recall that the (binary) Krawtchouk polynomial of degree i and order n (which is an integer), $P_i^n(X)$ is defined for $i \in \{0, \cdots, n\}$ by:

$$P_i^n(X) \stackrel{\text{def}}{=} \frac{(-1)^i}{2^i} \sum_{j=0}^{i} (-1)^j \binom{X}{j} \binom{n-X}{i-j} \quad \text{where } \binom{X}{j} \stackrel{\text{def}}{=} \frac{1}{j!} X(X-1)\cdots(X-j+1). \tag{13}$$

Notice that it follows on the spot from the definition of a Krawtchouk polynomial that

$$P_k^n(0) = \frac{(-1)^k \binom{n}{k}}{2^k}. \tag{14}$$

Let us define the bias δ by

$$\delta \stackrel{\text{def}}{=} 1 - 2\mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1).$$

In other words $\mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1) = \frac{1}{2}(1-\delta)$. These Krawtchouk polynomials are readily related to δ. We first observe that

$$\mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1) = \frac{\sum_{\substack{j=1 \\ j \text{ odd}}}^{w} \binom{t}{j}\binom{n-t}{w-j}}{\binom{n}{w}}.$$

Moreover by observing that $\sum_{j=0}^{w} \binom{t}{j}\binom{n-t}{w-j} = \binom{n}{w}$ we can recast the following evaluation of a Krawtchouk polynomial as

$$\begin{aligned}
\frac{(-2)^w}{\binom{n}{w}} P_w^n(t) &= \frac{\sum_{j=0}^{w} (-1)^j \binom{t}{j}\binom{n-t}{w-j}}{\binom{n}{w}} \\
&= \frac{\sum_{\substack{j=0 \\ j \text{ even}}}^{w} \binom{t}{j}\binom{n-t}{w-j} - \sum_{\substack{j=1 \\ j \text{ odd}}}^{w} \binom{t}{j}\binom{n-t}{w-j}}{\binom{n}{w}} \\
&= \frac{\binom{n}{w} - 2\sum_{\substack{j=1 \\ j \text{ odd}}}^{w} \binom{t}{j}\binom{n-t}{w-j}}{\binom{n}{w}} \\
&= 1 - 2\mathbb{P}_{\mathbf{e}}(\langle \mathbf{h}, \mathbf{e} \rangle = 1) \\
&= \delta.
\end{aligned} \tag{15}$$

To simplify notation we will drop the superscript n in the Krawtchouk polynomial notation. It will be chosen as the length of the MDPC code when will use it in our case. An important lemma that we will need is the following one.

Lemma 5. *For all x in $\{1,\ldots,t\}$, we have*

$$\frac{P_w(x)}{P_w(x-1)} = \left(1+O\left(\frac{1}{n}\right)\right)\frac{n-2w+\sqrt{(n-2w)^2-4w(n-w)}}{2(n-w)}.$$

Proof. This follows essentially from arguments taken in the proof of [26][Lemma 36, §7, Ch. 17]. The result we use appears however more explicitly in [24][Sec. IV] where it is proved that if x is in an interval of the form $\left[0,(1-\alpha)\left(n/2-\sqrt{w(n-w)}\right)\right]$ for some constant $\alpha\in[0,1)$ independent of x, n and w, then

$$\frac{P_w(x+1)}{P_w(x)} = \left(1+O\left(\frac{1}{n}\right)\right)\frac{n-2w+\sqrt{(n-2w)^2-4w(n-w)}}{2(n-w)}.$$

For our choice of t this condition is met for x and the lemma follows immediately. □

We are ready now to prove Lemma 4.

Proof of Lemma 4. We start the proof by using (15) which says that

$$\delta = \frac{(-2)^w}{\binom{n}{w}}P_w^n(t).$$

We then observe that

$$\begin{aligned}
\frac{(-2)^w}{\binom{n}{w}}P_w^n(t) &= \frac{(-2)^w}{\binom{n}{w}}\frac{P_w^n(t)}{P_w^n(t-1)}\frac{P_w^n(t-1)}{P_w^n(t-2)}\cdots\frac{P_w^n(1)}{P_w^n(0)}P_w^n(0)\\
&= \frac{(-2)^w}{\binom{n}{w}}\left(\left(1+O\left(\frac{1}{n}\right)\right)\frac{n-2w+\sqrt{(n-2w)^2-4w(n-w)}}{2(n-w)}\right)^t P_w^n(0) \text{ (by Lemma 5)}\\
&= \left(1+O\left(\frac{1}{n}\right)\right)^t\left(\frac{n-2w+\sqrt{(n-2w)^2-4w(n-w)}}{2(n-w)}\right)^t \text{ (by (14))}\\
&= e^{t\ln\left(\frac{1-2\omega+\sqrt{(1-2\omega)^2-4\omega(1-\omega)}}{2(1-\omega)}\right)}\left(1+O\left(\frac{t}{n}\right)\right) \text{ where } \omega\stackrel{\text{def}}{=}\frac{w}{n}\\
&= e^{t\ln\left(\frac{1-2\omega+1-4\omega+O\left(\omega^2\right)}{2(1-\omega)}\right)}\left(1+O\left(\frac{t}{n}\right)\right)\\
&= e^{t\ln\left(\frac{1-3\omega+O\left(\omega^2\right)}{1-\omega}\right)}\left(1+O\left(\frac{t}{n}\right)\right)\\
&= e^{-2t\omega+O\left(\frac{tw^2}{n^2}\right)}\left(1+O\left(\frac{t}{n}\right)\right)\\
&= e^{-\frac{2wt}{n}}\left(1+O\left(\frac{1}{\sqrt{n}}\right)\right),
\end{aligned}$$

where we used at the last equation that $t = \theta(\sqrt{n})$ and $w = \theta(\sqrt{n})$. □

A.3 Estimation of the probability that a bit is incorrectly estimated by the first step of the bit flipping algorithm

We are here in the model where every bit is involved in $w/2$ parity-check equations and each parity-check equation is of weight w. We assume that the bit-flipping algorithm consists in computing for each bit i the syndrome bits corresponding to the parity-checks involving i and taking the majority vote of these syndrome bits. We model each vote of a parity-check by a Bernoulli variable equal to 1 if the information coming from this random variable says that the bit should be flipped. The parameter of this Bernoulli random variable depends on whether or not i is incorrect. When i is correct, then the Bernoulli random variable is of parameter p_0. When i is incorrect, then the Bernoulli random variable is of parameter p_1. We bring in the quantities

$$q_0 \stackrel{\text{def}}{=} \mathbb{P}(\text{flip the bit}|\text{bit was correct}) \tag{16}$$

$$q_1 \stackrel{\text{def}}{=} \mathbb{P}(\text{stay with the same value}|\text{bit was incorrect}) \tag{17}$$

Lemma 6. *For $b \in \{0,1\}$, we have*

$$q_b = O\left(\frac{(1-\varepsilon^2)^{w/4}}{\sqrt{\pi w}\varepsilon}\right).$$

Proof. For $b \in \{0,1\}$, we let X_1^b, $X_2^b, \ldots, X_{w/2}^b$ be independent random variables of parameter p_b. We obviously have

$$\begin{aligned} q_0 &\leq \mathbb{P}(\sum_{i=1}^{w/2} X_i^0 \geq w/4) \\ q_1 &\leq \mathbb{P}(\sum_{i=1}^{w/2} X_i^1 \leq w/4). \end{aligned}$$

By using Lemma 3 we obtain for q_0

$$\begin{aligned} q_0 &\leq \frac{(1-p_0)\sqrt{\frac{1}{2}}}{(\frac{1}{2}-p_0)\sqrt{2\pi\frac{w}{2}(1-\frac{1}{2})}} e^{-w/2D\left(\frac{1}{2}\|p_0\right)} \\ &\leq \frac{(1-p_0)}{\sqrt{\pi w}\varepsilon} e^{-w/2D\left(\frac{1}{2}\|\frac{1}{2}-\frac{1}{2}\varepsilon(1+O(1/w))\right)} \quad (18) \\ &\leq \frac{(1-p_0)}{\sqrt{\pi w}\varepsilon} e^{\frac{w\left(\ln(1-\varepsilon^2)+O\left(\frac{1}{w}\right)\right)}{4}} \quad (19) \\ &\leq O\left(\frac{(1-\varepsilon^2)^{w/4}}{\sqrt{\pi w}\varepsilon}\right) \quad (20) \end{aligned}$$

Whereas for q_1 we also obtain

$$\begin{aligned} q_1 &\leq \frac{p_1\sqrt{\frac{1}{2}}}{(p_1-\frac{1}{2})\sqrt{2\pi\frac{w}{2}\frac{1}{2}}} e^{-w/2D\left(\frac{1}{2}\|p_1\right)} \quad (21) \\ &\leq O\left(\frac{(1-\varepsilon^2)^{w/4}}{\sqrt{\pi w}\varepsilon}\right) \quad (22) \end{aligned}$$

□

A.4 Proof of Theorem 1

We are ready now to prove Theorem 1. We use here the notation of Assumption 1. Recall that e^0 denotes the true error vector. e^1 is the value of vector e after one round of iterative decoding in Algorithm 1. We let $\Delta e \stackrel{\text{def}}{=} e^0+e^1$. Call $X_1^0,\ldots,X_{n-t}^0$ the values after one round of iterative decoding of the $n-t$ bits which were without error initially (that is the bits i such that $e_i^0=0$) . Similarly let $X_1^1,\ldots,X_t^1$ be the values after one round of iterative decoding of the t bits which were initially in error (i.e. for which $e_i^0=1$). We let

$$\begin{aligned} S_0 &\stackrel{\text{def}}{=} X_1^0+\cdots+X_{n-t}^0 \\ S_1 &\stackrel{\text{def}}{=} X_1^1+\cdots+X_t^1 \end{aligned}$$

S_0 is the number of errors that were introduced after one round of iterative decoding coming from flipping the $n-t$ bits that were initially correct, that is the number of i's for which $e_i^0=0$ and $e_i^1=1$. Similarly S_1 is the number of errors that are left after one round of iterative decoding coming from not flipping the t bits that were initially incorrect, that is the number of i's for which $e_i^0=1$ and $e_i^1=0$.

Let S be the weight of Δe. By assumption 1 we have

$$P_{\text{err}} \leq \mathbb{P}(|\Delta e| \geq \alpha t) = \mathbb{P}(S \geq \alpha t),$$

for some α in $(0,1)$. We have

$$\begin{aligned} \mathbb{P}(S \geq \alpha t) &\leq \mathbb{P}(S_0 \geq \alpha t/2 \cup S_1 \geq \alpha t/2) \\ &\leq \mathbb{P}(S_0 \geq \alpha t/2) + \mathbb{P}(S_1 \geq \alpha t/2) \end{aligned}$$

By Assumption 1, S_0 is the sum of $n-t$ Bernoulli variables of parameter q_0. By applying Lemma 3 we obtain

$$\begin{aligned} \mathbb{P}(S_0 \geq \alpha t/2) &\leq \frac{(1-q_0)\sqrt{\frac{\alpha t}{2(n-t)}}}{(\frac{\alpha t}{2(n-t)} - q_0)\sqrt{2\pi(n-t)(1-\frac{\alpha t}{2(n-t)})}} e^{-(n-t)D\left(\frac{\alpha t}{2(n-t)} \middle\| q_0\right)} \\ &\leq \frac{1}{\sqrt{\alpha \pi t}} e^{-(n-t)D\left(\frac{\alpha t}{2(n-t)} \middle\| q_0\right)} \end{aligned} \tag{23}$$

We observe now that

$$D\left(\frac{\alpha t}{2(n-t)} \middle\| q_0\right) \geq D\left(\frac{\alpha t}{2(n-t)} \middle\| O\left(\frac{(1-\varepsilon^2)^{w/4}}{\sqrt{\pi w}\varepsilon}\right)\right) \tag{24}$$

where we used the upper-bound on q_0 coming from Lemma 6 and the fact that $D(x\|y) \geq D(x\|y')$ for $0 < y < y' < x < 1$. By using this and Lemma 2, we deduce

$$\begin{aligned} D\left(\frac{\alpha t}{2(n-t)} \middle\| q_0\right) &\geq \frac{\alpha t}{2(n-t)} \ln\left(\frac{\alpha t}{2(n-t)}\right) - \frac{\alpha t}{2(n-t)} \ln\left(O\left(\frac{(1-\varepsilon^2)^{w/4}}{\varepsilon\sqrt{w}}\right)\right) + O\left(\frac{\alpha t}{2(n-t)}\right) \\ &\geq \frac{\alpha t}{2(n-t)} \ln\left(\frac{t\sqrt{w}}{n}\right) - \frac{\alpha t w}{8(n-t)} \ln\left(1-\varepsilon^2\right) + O\left(\frac{t}{n}\right) \\ &\geq -\frac{\alpha t}{8(n-t)} \ln n - \frac{\alpha t w}{8(n-t)} \ln\left(1-\varepsilon^2\right) + O\left(\frac{t}{n}\right). \end{aligned}$$

By plugging in this expression in (23) we obtain

$$\mathbb{P}(S_0 \geq \alpha t/2) \leq \frac{1}{\sqrt{\alpha \pi t}} e^{\frac{\alpha t w}{8}\ln\left(1-\varepsilon^2\right) + \frac{\alpha t}{8}\ln(n) + O(t)}$$

On the other hand we have

$$\begin{aligned} \mathbb{P}(S_1 \geq \alpha t/2) &\leq \frac{(1-q_1)\sqrt{\frac{\alpha}{2}}}{(\frac{\alpha}{2} - q_1)\sqrt{2\pi t(1-\frac{\alpha}{2})}} e^{-tD\left(\frac{\alpha}{2} \| q_1\right)} \\ &\leq \frac{1}{\sqrt{\alpha \pi t}} e^{-tD\left(\frac{\alpha}{2} \| q_1\right)} \end{aligned} \tag{25}$$

Similarly to what we did above, by using the upper-bound on q_1 of Lemma 6 and $D(x\|y) \geq D(x\|y')$ for $0 < y < y' < x < 1$, we deduce that

$$D\Big(\frac{\alpha}{2}\Big\|q_1\Big) \geq D\left(\frac{\alpha}{2}\middle\|O\left(\frac{(1-\varepsilon^2)^{w/4}}{\varepsilon\sqrt{w}}\right)\right)$$

By using together with Lemma 2 we obtain

$$\begin{aligned} D\Big(\frac{\alpha}{2}\Big\|q_1\Big) &\geq -h(\alpha/2) - \frac{\alpha}{2}\ln\left(O\left(\frac{(1-\varepsilon^2)^{w/4}}{\varepsilon\sqrt{w}}\right)\right) + O\left(\frac{(1-4\varepsilon^2)^{w/4}}{\varepsilon\sqrt{w}}\right) \\ &\geq -\frac{\alpha w}{8}\ln\left(1-\varepsilon^2\right) + \frac{\alpha}{8}\ln n + O\left(1\right). \end{aligned}$$

By using this lower-bound in (25), we deduce

$$\mathbb{P}(S_1 \geq \alpha t/2) \leq \frac{1}{\sqrt{\alpha\pi t}} e^{\frac{\alpha t w}{8}\ln\left(1-\varepsilon^2\right)+\frac{\alpha t}{8}\ln(n)+O(t)}.$$

B Proof of Proposition 1

Let us recall first the proposition

Proposition 1. *Let f be a Boolean function which is equal to 1 on a fraction α of inputs which can be implemented by a quantum circuit of depth D_f and whose gate complexity is C_f. Using Grover's algorithm for finding an input x of f for which $f(x) = 1$ can not take less quantum resources than a Grover's attack on AES-N as soon as*

$$\frac{D_f \cdot C_f}{\alpha} \geq 2^N D_{AES-N} \cdot C_{AES-N}$$

where D_{AES-N} and C_{AES-N} are respectively the depth and the complexity of the quantum circuit implementing AES-N.

Proof. Following Zalka[36], the best way is to perform Grover's algorithm sequentially with the maximum allowed number of iterations in order not to go beyond MAXDEPTH. Grover's algorithm consists of iterations of the following procedure:

- Apply $U : |0\rangle|0\rangle \to \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}|x\rangle|f(x)\rangle$.
- Apply a phase flip on the second register to get $\sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}(-1)^{f(x)}|x\rangle|f(x)\rangle$.
- Apply $U^\dagger$.

If we perform I iterations of the above for $I \leq \frac{1}{\sqrt{\alpha}}$ then the winning probability is upper bounded by αI^2. In our setting, we can perform $I = \frac{\mathsf{MAXDEPTH}}{D_f}$ sequentially before measuring, and each iteration costs time C_f. At each iteration, we succeed with probability αI^2 and we need to repeat this procedure $\frac{1}{\alpha I^2}$ times to get a result with constant probability. From there, we conclude that the total complexity Q is:

$$Q = \frac{1}{\alpha I^2} \cdot I \cdot C_f = \frac{D_f \cdot C_f}{\alpha \mathsf{MAXDEPTH}}. \tag{26}$$

A similar reasoning performed on using Grover's search on AES-N leads to a quantum complexity

$$Q_{AES-N} = \frac{2^N D_{AES-N} \cdot C_{AES-N}}{\mathsf{MAXDEPTH}}. \tag{27}$$

The proposition follows by comparing (26) with (27). □

Classic McEliece: conservative code-based cryptography

29 November 2017

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Daniel J. Bernstein, University of Illinois at Chicago
- Tung Chou, Osaka University
- Tanja Lange, Technische Universiteit Eindhoven
- Ingo von Maurich, self
- Rafael Misoczki, Intel Corporation
- Ruben Niederhagen, Fraunhofer SIT
- Edoardo Persichetti, Florida Atlantic University
- Christiane Peters, self
- Peter Schwabe, Radboud University
- Nicolas Sendrier, Inria
- Jakub Szefer, Yale University
- Wen Wang, Yale University

E-mail address (preferred): `authorcontact-mceliece@box.cr.yp.to`

Telephone (if absolutely necessary): +1-312-996-3422

Postal address (if absolutely necessary): Daniel J. Bernstein, Department of Computer Science, University of Illinois at Chicago, 851 S. Morgan (M/C 152), Room 1120 SEO, Chicago, IL 60607–7053.

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: ×. See also printed version of "Statement by Each Submitter".

Document generated with the help of `pqskeleton` version 20171123. This submission is also known as "Boring Binary Goppa codes"; for the meaning of "boring" see https://cr.yp.to/talks/2015.10.05/slides-djb-20151005-a4.pdf and https://tinyurl.com/ydg55pta.

Contents

1 Introduction

The first code-based public-key cryptosystem was introduced in 1978 by McEliece [39]. The public key specifies a random binary Goppa code. A ciphertext is a codeword plus random errors. The private key allows efficient decoding: extracting the codeword from the ciphertext, identifying and removing the errors.

The McEliece system was designed to be one-way (OW-CPA), meaning that an attacker cannot efficiently find the codeword from a ciphertext and public key, when the codeword is chosen randomly. The security level of the McEliece system has remained remarkably stable, despite dozens of attack papers over 40 years. The original McEliece parameters were designed for only 2^{64} security, but the system easily scales up to "overkill" parameters that provide ample security margin against advances in computer technology, including quantum computers.

The McEliece system has prompted a tremendous amount of followup work. Some of this work improves efficiency while clearly preserving security:[1] this includes a "dual" PKE proposed by Niederreiter [42], software speedups such as [7], and hardware speedups such as [58].

Furthermore, it is now well known how to efficiently convert an OW-CPA PKE into a KEM that is IND-CCA2 secure against all ROM attacks. This conversion is tight, preserving the security level, under two assumptions that are satisfied by the McEliece PKE: first, the PKE is deterministic (i.e., decryption recovers all randomness that was used); second, the PKE has no decryption failures for valid ciphertexts. Even better, very recent work [48] suggests the possibility of achieving similar tightness for the broader class of QROM attacks. The risk that a hash-function-specific attack could be faster than a ROM or QROM attack is addressed by the standard practice of selecting a well-studied, high-security, "unstructured" hash function.

This submission *Classic McEliece* (CM) brings all of this together. It presents a KEM designed for IND-CCA2 security at a very high security level, even against quantum computers. The KEM is built conservatively from a PKE designed for OW-CPA security, namely Niederreiter's dual version of McEliece's PKE using binary Goppa codes. Every level of the construction is designed so that future cryptographic auditors can be confident in the long-term security of post-quantum public-key encryption.

[1] Other work includes McEliece variants whose security has not been studied as thoroughly. For example, many proposals replace binary Goppa codes with other families of codes, and lattice-based cryptography replaces "codeword plus random errors" with "lattice point plus random errors". Code-based cryptography and lattice-based cryptography are two of the main types of candidates identified in NIST's call for Post-Quantum Cryptography Standardization. This submission focuses on the classic McEliece system precisely because of how thoroughly it has been studied.

2 General algorithm specification (part of 2.B.1)

2.1 Notation

The list below introduces the notation used in this section. It is meant as a reference guide only; for complete definitions of the terms listed, refer to the appropriate text. Some other symbols are also used occasionally; they are introduced in the text where appropriate.

n	The code length	(part of the CM parameters)
k	The code dimension	(part of the CM parameters)
t	The guaranteed error-correction capability	(part of the CM parameters)
q	The size of the field used	(part of the CM parameters)
m	$\log_2 q$	(part of the CM parameters)
H	A cryptographic hash function	(part of the CM parameters)
ℓ	Length of a hash digest	(part of the CM parameters)
g	A polynomial in $\mathbb{F}_q[x]$	(part of the private key)
α_i	An element of the finite field $\mathbb{F}_q$	(part of the private key)
Γ	$(g, \alpha_1, \ldots, \alpha_n)$	(part of the private key)
s	A bit string of length n	(part of the private key)
(s, Γ)	A CM private key	
T	A CM public key	
e	A bit string of length n and Hamming weight t	
C	A ciphertext encapsulating a session key	
C_0	A bit string of length $n - k$	(part of the ciphertext)
C_1	A bit string of length ℓ	(part of the ciphertext)

Elements of $\mathbb{F}_2^n$, such as codewords and error vectors, are always viewed as column vectors. This convention avoids all transpositions. Beware that this differs from a common convention in coding theory, namely to write codewords as row vectors but to transpose the codewords for applying parity checks.

2.2 Parameters

The *CM parameters* are implicit inputs to the CM algorithms defined below. A CM parameter set specifies the following:

- A positive integer m. This also defines a parameter $q = 2^m$.
- A positive integer n with $n \leq q$.

- A positive integer $t \geq 2$ with $mt < n$. This also defines a parameter $k = n - mt$.
- A monic irreducible polynomial $f(z) \in \mathbb{F}_2[z]$ of degree m. This defines a representation $\mathbb{F}_2[z]/f(z)$ of the field $\mathbb{F}_q$.
- A positive integer ℓ, and a cryptographic hash function H that outputs ℓ bits.

2.3 Key generation

Given a set of CM parameters, a user generates a *CM key pair* as follows:

1. Generate a uniform random monic irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t.
2. Select a uniform random sequence $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of n distinct elements of $\mathbb{F}_q$.
3. Compute the $t \times n$ matrix $\tilde{H} = \{h_{i,j}\}$ over $\mathbb{F}_q$, where $h_{i,j} = \alpha_j^{i-1}/g(\alpha_j)$ for $i = 1, \ldots, t$ and $j = 1, \ldots, n$.
4. Form an $mt \times n$ matrix $\hat{H}$ over $\mathbb{F}_2$ by replacing each entry $c_0 + c_1 z + \cdots + c_{m-1} z^{m-1}$ of $\tilde{H}$ with a column of t bits $c_0, c_1, \ldots, c_{m-1}$.
5. Apply Gaussian elimination to $\hat{H}$ to reduce $\hat{H}$ to systematic form $(I_{n-k} \mid T)$, where I_{n-k} is an $(n-k) \times (n-k)$ identity matrix. If Gaussian elimination does not produce I_{n-k} (i.e., $\hat{H}$ cannot be transformed to systematic form), go back to Step 1.
6. Generate a uniform random n-bit string s.
7. Put $\Gamma = (g, \alpha_1, \alpha_2, \ldots, \alpha_n)$ and output (s, Γ) as private key and T as public key.

The second part of the private key, $\Gamma = (g, \alpha_1, \alpha_2, \ldots, \alpha_n)$, describes a binary Goppa code of length n and dimension $k = n - mt$. The public key T is a binary $(n-k) \times k$ matrix such that $H = (I_{n-k} \mid T)$ is a parity-check matrix for the same Goppa code.

2.4 Encoding subroutine

The encoding subroutine takes two inputs: a weight-t column vector $e \in \mathbb{F}_2^n$; and a public key T, i.e., an $(n-k) \times k$ matrix over $\mathbb{F}_2$. The subroutine returns a vector $C_0 \in \mathbb{F}_2^{n-k}$ defined as follows:

1. Define $H = (I_{n-k} \mid T)$.
2. Compute and return $C_0 = He \in \mathbb{F}_2^{n-k}$.

2.5 Decoding subroutine

The decoding subroutine decodes $C_0 \in \mathbb{F}_2^{n-k}$ to a word e of Hamming weight $\mathsf{wt}(e) = t$ with $C_0 = He$ if such a word exists; otherwise it returns failure.

Formally, this subroutine takes two inputs: a vector $C_0 \in \mathbb{F}_2^{n-k}$; and a private key (s, Γ). The subroutine has two possible return values, defined in terms of the public key T that corresponds to (s, Γ):

- If C_0 was returned by the encoding subroutine on input e and T, then the decoding subroutine returns e. In other words, if there exists a weight-t vector $e \in \mathbb{F}_2^n$ such that $C_0 = He$ with $H = (I_{n-k} \mid T)$, then the decoding subroutine returns e.
- If C_0 does not have the form He for any weight-t vector $e \in \mathbb{F}_2^n$, then the decoding subroutine returns $\perp$ (failure).

The subroutine works as follows:

1. Extend C_0 to $v = (C_0, 0, \ldots, 0) \in \mathbb{F}_2^n$ by appending k zeros.
2. Find the unique codeword c in the Goppa code defined by Γ that is at distance $\leq t$ from v. If there is no such codeword, return $\perp$.
3. Set $e = v + c$.
4. If $\mathsf{wt}(e) = t$ and $C_0 = He$, return e. Otherwise return $\perp$.

There are several standard algorithms for Step 2 of this subroutine. For references and speedups see generally [7] and [17].

To see why the subroutine works, note first that the "syndrome" Hv is C_0, because the first $n - k$ positions of v are multiplied by the identity matrix and the remaining positions are zero. If C_0 has the form He where e has weight t then $Hv = He$, so $c = v + e$ is a codeword. This codeword has distance exactly t from v, and it is the unique codeword at distance $\leq t$ from v since the minimum distance of Γ is at least $2t + 1$. Hence Step 2 finds c, Step 3 finds e, and Step 4 returns e. Conversely, if the subroutine returns e in Step 4 then e has been verified to have weight t and to have $C_0 = He$, so if C_0 does not have this form then the subroutine must return $\perp$.

The logic here relies on Step 2 always finding a codeword at distance t if one exists. It does not rely on Step 2 failing in the cases that a codeword does not exist: the subroutine remains correct if, instead of returning $\perp$, Step 2 chooses some vector $c \in \mathbb{F}_2^n$ and continues on to Step 3.

Implementors are cautioned that it is important to avoid leaking secret information through side channels, and that the distinction between success and failure in this subroutine is secret in the context of the Classic McEliece KEM. In particular, immediately stopping the

computation when Step 2 returns $\perp$ would reveal this distinction through timing, so it is recommended for implementors to have Step 2 always choose some $c \in \mathbb{F}_2^n$.

As a further implementation note: In order to test $C_0 = He$, the decoding subroutine does not need to recompute H from Γ as in key generation. Instead it can use any parity-check matrix H' for the same code. The computation uses $v = (C_0, 0, \ldots, 0)$ and compares $H'v$ to $H'e$. The results are equal if and only if $v + e = c$ is a codeword, which implies $He = H(v + c) = Hv + Hc = Hv = C_0$. There are various standard choices of H' related to $\hat{H}$ that are easily recovered from Γ, and that can be applied to vectors without using quadratic space.

Remark. Note that the triple of algorithms (Key Generation, Encoding, Decoding) is essentially Niederreiter's "dual" version [42] of the McEliece cryptosystem (plus a private string s not used in decoding; s is used in decapsulation below). We use the binary Goppa code family, as in McEliece's original proposal [39], rather than variants such as the GRS family considered by Niederreiter. See Section 4 for further history.

2.6 Encapsulation

The sender generates a session key K and its ciphertext C as follows:

1. Generate a uniform random vector $e \in \mathbb{F}_2^n$ of weight t.
2. Use the encoding subroutine on e and public key T to compute C_0.
3. Compute $C_1 = \mathsf{H}(2, e)$; see Section 2.8 for H input encodings. Put $C = (C_0, C_1)$.
4. Compute $K = \mathsf{H}(1, e, C)$; see Section 2.8 for H input encodings.
5. Output session key K and ciphertext C.

2.7 Decapsulation

The receiver decapsulates the session key K from ciphertext C as follows:

1. Split the ciphertext C as (C_0, C_1) with $C_0 \in \mathbb{F}_2^{n-k}$ and $C_1 \in \mathbb{F}_2^{\ell}$.
2. Set $b \leftarrow 1$.
3. Use the decoding subroutine on C_0 and private key Γ to compute e. If the subroutine returns $\perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $C_1' = \mathsf{H}(2, e)$; see Section 2.8 for H input encodings.
5. If $C_1' \neq C_1$, set $e \leftarrow s$ and $b \leftarrow 0$.

6. Compute $K = \mathsf{H}(b, e, C)$; see Section 2.8 for H input encodings.

7. Output session key K.

If C is a legitimate ciphertext then $C = (C_0, C_1)$ with $C_0 = He$ for some $e \in \mathbb{F}_2^n$ of weight t and $C_1 = \mathsf{H}(2, e)$. The decoding algorithm will return e as the unique weight-t vector and the $C_1' = C_1$ check will pass, thus $b = 1$ and K matches the session key computed in encapsulation.

As an implementation note, the output of decapsulation is unchanged if "$e \leftarrow s$" in Step 3 is changed to assign something else to e. Implementors may prefer, e.g., to set e to a fixed n-bit string, or a random n-bit string other than s. However, the definition of decapsulation does depend on e being set to s in Step 5.

Implementors are again cautioned that it is important to avoid leaking secret information through side channels. In particular, the distinction between failures in Step 3, failures in Step 5, and successes is secret information, and branching would leak this information through timing. It is recommended for implementors to always go through the same sequence of computations, using arithmetic to simulate tests and conditional assignments.

2.8 Representation of objects as byte strings

Vectors over $\mathbb{F}_2$. If r is a multiple of 8 then an r-bit vector $v = (v_0, v_1, \ldots, v_{r-1}) \in \mathbb{F}_2^r$ is represented as the following sequence of $r/8$ bytes:

$$(v_0+2v_1+4v_2+\cdots+128v_7, v_8+2v_9+4v_{10}+\cdots+128v_{15}, \ldots, v_{r-8}+2v_{r-7}+4v_{r-6}+\cdots+128v_{r-1}).$$

If r is not a multiple of 8 then an r-bit vector $v = (v_0, v_1, \ldots, v_{r-1}) \in \mathbb{F}_2^r$ is zero-padded to length between $r+1$ and $r+7$, whichever is a multiple of 8, and then represented as above.

Session keys. A session key K is an element of $\mathbb{F}_2^\ell$. It is represented as a $\lceil \ell/8 \rceil$-byte string.

Ciphertexts. A ciphertext C has two components: $C_0 \in \mathbb{F}_2^{n-k}$ and $C_1 \in \mathbb{F}_2^\ell$. The ciphertext is represented as the concatenation of the $\lceil mt/8 \rceil$-byte string representing C_0 and the $\lceil \ell/8 \rceil$-byte string representing C_1.

Hash inputs. There are three types of hash inputs: $(2, v)$; $(1, v, C)$; and $(0, v, C)$. Here $v \in \mathbb{F}_2^n$, and C is a ciphertext.

The initial 0, 1, or 2 is represented as a byte. The vector v is represented as the next $\lceil n/8 \rceil$ bytes. The ciphertext, if present, is represented as the next $\lceil mt/8 \rceil + \lceil \ell/8 \rceil$ bytes.

Public keys. The public key T, which is essentially a $mt \times (n - mt)$ matrix, is represented in a row-major fashion. Each row of T is represented as a $\lceil k/8 \rceil$-byte string, and the public key is represented as the $mt\lceil k/8 \rceil$-byte concatenation of these strings.

Field elements. Each element of $\mathbb{F}_q \cong \mathbb{F}_2[z]/f(z)$ has the form $\sum_{i=0}^{m-1} c_i z^i$ where $c_i \in \mathbb{F}_2$. The representation of the field element is the representation of the vector $(c_0, c_1, \ldots, c_{m-1}) \in \mathbb{F}_2^m$.

Private keys. A private key has the form $(s, g, \alpha_1, \alpha_2, \ldots, \alpha_n)$. This is represented as the concatenation of three parts:

- The $\lceil n/8 \rceil$-byte string representing $s \in \mathbb{F}_2^n$.
- The $t\lceil m/8 \rceil$-byte string representing $g = g_0 + g_1 x + \cdots + g_{t-1}x^{t-1} + x^t$, namely the concatenation of the representations of the field elements $g_0, g_1, \ldots, g_{t-1}$.
- The representation defined below of the sequence $(\alpha_1, \ldots, \alpha_n)$.

The obvious representation of $(\alpha_1, \ldots, \alpha_n)$ would be as a sequence of n field elements. We specify a different representation that simplifies fast constant-time decoding algorithms: $(\alpha_1, \ldots, \alpha_n)$ are converted into a $(2m-1)2^{m-1}$-bit vector of "control bits" defined below, and then this vector is represented as $\lceil (2m-1)2^{m-4} \rceil$ bytes as above.

Recall that a "Beneš network" is a series of $2m-1$ stages of swaps applied to an array of $q = 2^m$ objects $(a_0, a_1, \ldots, a_{q-1})$. The first stage conditionally swaps a_0 and a_1, conditionally swaps a_2 and a_3, conditionally swaps a_4 and a_5, etc., as specified by a sequence of $q/2$ control bits (1 meaning swap, 0 meaning leave in place). The second stage conditionally swaps a_0 and a_2, conditionally swaps a_1 and a_3, conditionally swaps a_4 and a_6, etc., as specified by the next $q/2$ control bits. This continues through the mth stage, which conditionally swaps a_0 and $a_{q/2}$, conditionally swaps a_1 and $a_{q/2+1}$, etc. The $(m+1)$st stage is just like the $(m-1)$st stage (with new control bits), the $(m+2)$nd stage is just like the $(m-2)$nd stage, and so on through the $(2m-1)$st stage.

Finally, $(\alpha_1, \ldots, \alpha_n)$ are represented as the control bits for a Beneš network that, when applied to all q field elements $(0, z^{12}, z^{11}, z^{12} + z^{11}, z^{10}, z^{12} + z^{10}, \ldots)$ in reverse lexicographic order, produces an array that begins $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and continues with the remaining field elements in some order. An algorithm by Lev, Pippenger, and Valiant [35] computes these control bits at reasonably high speed given the target array.

3 List of parameter sets (part of 2.B.1)

3.1 Parameter set `kem/mceliece6960119`

KEM with $m = 13$, $n = 6960$, $t = 119$, $\ell = 256$. Field polynomial $f(z) = z^{13}+z^4+z^3+z+1$. Hash function: SHAKE256 with 32-byte output.

3.2 Parameter set `kem/mceliece8192128`

KEM with $m = 13$, $n = 8192$, $t = 128$, $\ell = 256$. Field polynomial $f(z) = z^{13}+z^4+z^3+z+1$. Hash function: SHAKE256 with 32-byte output.

4 Design rationale (part of 2.B.1)

4.1 One-wayness

There is a long history of trapdoor systems (in modern terminology: PKEs) that are designed to be one-way (in modern terminology: OW-CPA). One-wayness means that it is difficult to invert the map from input to ciphertext, given the public key, when the input is chosen uniformly at random.

The McEliece system is one of the oldest proposals, almost as old as RSA. RSA has suffered dramatic security losses, while the McEliece system has maintained a spectacular security track record unmatched by any other proposals for post-quantum encryption. This is the reason that we have chosen to submit the McEliece system.

Here is more detail to explain what we mean by "spectacular security track record".

With the key-size optimizations discussed below, the McEliece system uses a key size of $(c_0 + o(1))b^2(\lg b)^2$ bits to achieve 2^b security against all inversion attacks that were known in 1978, when the system was introduced. Here lg means logarithm base 2, $o(1)$ means something that converges to 0 as $b \to \infty$, and $c_0 \approx 0.7418860694$.

The best attack at that time was from 1962 Prange [47]. After 1978 there were 25 publications studying the one-wayness of the system and introducing increasingly sophisticated non-quantum attack algorithms:

1. 1981 Clark–Cain [18], crediting Omura.
2. 1988 Lee–Brickell [33].
3. 1988 Leon [34].
4. 1989 Krouk [32].

5. 1989 Stern [52].
6. 1989 Dumer [24].
7. 1990 Coffey–Goodman [19].
8. 1990 van Tilburg [55].
9. 1991 Dumer [25].
10. 1991 Coffey–Goodman–Farrell [20].
11. 1993 Chabanne–Courteau [15].
12. 1993 Chabaud [16].
13. 1994 van Tilburg [56].
14. 1994 Canteaut–Chabanne [11].
15. 1998 Canteaut–Chabaud [12].
16. 1998 Canteaut–Sendrier [13].
17. 2008 Bernstein–Lange–Peters [8].
18. 2009 Bernstein–Lange–Peters–van Tilborg [10].
19. 2009 Finiasz–Sendrier [27].
20. 2011 Bernstein–Lange–Peters [9].
21. 2011 May–Meurer–Thomae [37].
22. 2012 Becker–Joux–May–Meurer [3].
23. 2013 Hamdaoui–Sendrier [29].
24. 2015 May–Ozerov [38].
25. 2016 Canto Torres–Sendrier [54].

What is the cumulative impact of all this work? Answer: With the same key-size optimizations, the McEliece system uses a key size of $(c_0 + o(1))b^2(\lg b)^2$ bits to achieve 2^b security against all non-quantum attacks known today, where c_0 is exactly the same constant. All of the improvements have disappeared into the $o(1)$.

This does not mean that the required key size is precisely the same—that dozens of attack papers over 40 years have not accomplished *anything*. What it means is that the required change in key size is below 1% once b is large enough; below 0.1% once b is large enough; etc. This is a remarkably stable security story.

What about quantum attacks? Grover's algorithm is applicable, reducing the attack cost to asymptotically its square root; see generally [5]. In other words, the key now needs $(4c_0 + o(1))b^2(\lg b)^2$ bits. As before, further papers on the topic have merely improved the $o(1)$.

All of the papers mentioned above are focusing on the most effective attack strategy known, namely "information-set decoding". This strategy does not exploit any particular structure of a generator matrix G: it recovers a low-weight error vector e given a *uniform random* matrix G and $Gm + e$ for some m. Experiments are consistent with the theory that McEliece's matrices G behave like uniform random matrices in this context.

There are also many papers studying attacks that instead recover McEliece's private key from the public key G. Recovering the private key also breaks one-wayness, since the attacker can then use the receiver's decryption algorithm. These attacks can be much faster than a brute-force search through private keys: for example, Sendrier's "support splitting" algorithm [49] quickly finds $\alpha_1, \ldots, \alpha_n$ given g provided that $n = 2^q$. More generally, whether or not $n = 2^q$, support splitting finds $\alpha_1, \ldots, \alpha_n$ given g and given the *set* $\{\alpha_1, \ldots, \alpha_n\}$. (This can be viewed as a reason to keep n somewhat smaller than 2^q, since then there are many possibilities for the set, along with many possibilities for g; one of our suggested parameter sets provides this extra defense.) However, despite this and other interesting speedups, the state-of-the-art key-recovery attacks are vastly slower than information-set decoding.

Various authors have proposed replacing the binary Goppa codes in McEliece's system with other families of codes: see, e.g., [2, 4, 40, 42, 44, 41]. Often these replacements are advertised as allowing smaller public keys. Unfortunately, many of these proposals have turned out to allow unacceptably fast recovery of the private key (or of something equivalent to the private key, something that allows fast inversion of the supposedly one-way function). Some small-key proposals are unbroken, but in this submission we focus on binary Goppa codes as the traditional, conservative, well-studied choice.

Authors of attacks on other codes often study whether binary Goppa codes are affected by their attacks. These studies consistently show that McEliece's system is far beyond all known attacks. For example, 2013 Faugère–Gauthier-Umaña–Otmani–Perret–Tillich [26] showed that "high-rate" binary Goppa codes can be distinguished from random codes. The worst-case possibility is that this distinguisher somehow allows an inversion attack faster than attacks for random codes. However, the distinguisher stops working

- at 8 errors for $n = 1024$ (where McEliece's original parameters used 50 errors),
- at 20 errors for $n = 8192$ (where our suggested parameters use more than 100 errors),

etc. As another example, the attack in [21] reaches degree $m = 2$ where McEliece's original parameters used degree $m = 10$ and where our suggested parameters use degree $m = 13$.

4.2 Better efficiency for the same one-wayness

The main focus of this submission is security, but we also take reasonable steps to improve efficiency when this clearly does not compromise security. In particular, we make the following two modifications suggested by Niederreiter [42].

First modification. The goal of the public key in McEliece's system is to communicate an $[n, k]$ linear code C over $\mathbb{F}_2$: a k-dimensional linear subspace of $\mathbb{F}_2^n$. This means communicating the ability to generate uniform random elements of C. McEliece accomplished this by choosing the public key to be a uniform random generator matrix G for C: specifically, multiplying any generator matrix for C by a uniform random invertible matrix.

Niederreiter accomplished this by instead choosing the public key to be the unique systematic-form generator matrix for C if one exists. This means a generator matrix of the form $\left(\frac{T}{I_k}\right)$ where T is some $(n-k) \times k$ matrix and I_k is the $k \times k$ identity matrix. Approximately 29% of choices of C have this form, so key generation becomes about 3.4× slower on average, but now the public key occupies only $k(n-k)$ bits instead of kn bits. Note that sending a systematic-form generator matrix also implies sending a parity-check matrix H for C, namely $(I_{n-k} \mid T)$.

Any attack against the limited set of codes allowed by Niederreiter implies an attack with probability 29% against the full set of codes allowed by McEliece; this is a security difference of at most 2 bits. Furthermore, any attack against Niederreiter's public key can be used to attack any generator matrix for the same code, and in particular McEliece's public key, since anyone given any generator matrix can quickly compute Niederreiter's public key by linear algebra.

Second modification. McEliece's ciphertext has the form $Ga + e$. Here G is a random $n \times k$ generator matrix for a code C as above; a is a column vector of length k; e is a weight-w column vector of length n; and the ciphertext is a column vector of length n. McEliece's inversion problem is to compute a uniform random input (a, e) given G and the ciphertext $Ga + e$.

Niederreiter's ciphertext instead has the form He. Here H is the unique systematic-form $(n-k) \times n$ parity-check matrix for C, and e is a weight-w column vector of length n, so the ciphertext is a column vector of length just $n-k$, shorter than McEliece's ciphertext. Niederreiter's inversion problem is to compute a uniform random input e given H and the ciphertext He.

Niederreiter's inversion problem is equivalent to McEliece's inversion problem for the same code. In particular, any attack recovering a random e from Niederreiter's He and H can be used with negligible overhead to recover a random (a, e) from McEliece's $Ga + e$ and G. Specifically, compute H from G, multiply H by $Ga + e$ to obtain $HGa + He = He$, apply the attack to recover e from He, subtract e from $Ga + e$ to obtain Ga, and recover a by linear algebra.

4.3 Indistinguishability against chosen-ciphertext attacks

Assume that McEliece's system is one-way. Niederreiter's system is then also one-way: the attacker cannot efficiently compute a uniform random weight-w vector e given Niederreiter's public key H and the ciphertext He.

What the user actually needs is more than one-wayness. The user is normally sending a plaintext with structure, perhaps a plaintext that can simply be guessed. Furthermore, the attacker can try modifying ciphertexts to see how the receiver reacts. McEliece's original PKE was not designed to resist, and does not resist, such attacks. In modern terminology, the user needs IND-CCA2 security.

There is a long literature studying the IND-CCA2 security of various PKE constructions, and in particular constructions built from an initial PKE assumed to have OW-CPA security. An increasingly popular simplification here is to encrypt the user's plaintext with an authenticated cipher such as AES-GCM. The public-key problem is then simply to send an unpredictable session key to use as the cipher key. Formally, our design goal here is to build a KEM with IND-CCA2 security; "KEM-DEM" composition [22] then produces a PKE with IND-CCA2 security, assuming a secure DEM. More complicated PKE constructions can pack some plaintext bytes into the ciphertext but are more difficult to audit and would be contrary to our goal of producing high confidence in security.

For our KEM construction we follow the best practices established in the literature:

- We use a uniform random input e. We compute the session key as a hash of e.
- Our ciphertext is the original ciphertext plus a "confirmation": another cryptographic hash of e.
- After using the private key to compute e from a ciphertext, we recompute the ciphertext (including the confirmation) and check that it matches.
- If decryption fails (i.e., if computing e fails or the recomputed ciphertext does not match), we do not return a KEM failure: instead we return a pseudorandom function of the ciphertext, specifically a cryptographic hash of a separate private key and the ciphertext.

We use a standard, thoroughly studied cryptographic hash function. We ensure that the three hashes mentioned above are obtained by applying this function to input spaces that are visibly disjoint. We choose the input details to simplify implementations that run in constant time, in particular not leaking whether decryption failed.

There are intuitive arguments for these practices, and to some extent there are also proofs. Specifically, a KEM construction 15 years ago from Dent [23, Section 6] features a tight proof of security against ROM attacks, assuming OW-CPA security of the underlying PKE; and a very recent KEM construction by Saito, Xagawa, and Yamakawa [48, Theorem 5.2] features a tight proof of security against the broader class of QROM attacks, under somewhat stronger

assumptions. Dent's theorem relies on the first three items in the list above, and the XYZ theorem from [48] relies on the first, third, and fourth items. Both theorems also rely on two PKE features that are provided by the PKE we use: the ciphertext is a *deterministic* function of the input e, and there are no decryption failures for legitimate ciphertexts. The theorems as stated do not apply directly to our KEM construction, but our preliminary analysis indicates that the proof ideas do apply; see Section 6. The deterministic PKE, the fact that decryption always works for legitimate ciphertexts, and the overall simplicity of the KEM construction should make it possible to formally verify complete proofs, building further confidence.

5 Detailed performance analysis (2.B.2)

5.1 Overview of implementations

We are supplying, as part of this submission, reference implementations for both of our parameter sets, `mceliece6960119` and `mceliece8192128`. Reference implementations are designed for clarity, not performance, so measuring their performance is not meaningful.

We are also supplying, as part of this submission, two additional software implementations for the larger parameter set, `mceliece8192128`. The `sse` implementation is (partially) vectorized using Intel's 128-bit (SSE4.1) vector instructions, and in particular provides much faster decapsulation performance than the `ref` implementation. The `avx` implementation is (partially) further vectorized using Intel's 256-bit vector instructions.

The `sse` and `avx` implementations are interoperable with the `ref` implementation, and produce identical test vectors. All three implementations are also designed to avoid all data flow from secrets to timing,[2] stopping timing attacks such as [53]. Formally verified protection against timing attacks can be provided by a combination of architecture documentation as recommended in [6] and [30], and timing-aware compilation as in [1].

We report measurements of the performance of `mceliece8192128/avx` as our speed estimate for `mceliece8192128` on the NIST PQC Reference Platform. To meet NIST's formal requirements, we also declare these numbers to be our current speed estimate for the smaller `mceliece6960119` parameter set. This is not an unreasonable estimate: the field size is the same, and other sizes are similar.

We also report preliminary measurements of key generation and decoding in hardware from an FPGA running a reference hardware design [58], for both parameter sets. The computations in McEliece's cryptosystem are particularly well suited for hardware implementations. The key generator is online at `http://caslab.csl.yale.edu/code/keygen/`.

[2] Each attempted key generation succeeds with probability about 29%, as mentioned earlier, so the total time for key generation varies. However, the final *successful* key generation takes constant time, and it uses separate random numbers from the unsuccessful key-generation attempts. In other words, the information about secrets that is leaked through timing is information about secrets that are not used.

5.2 Description of platforms

The software measurements were collected using `supercop-20171020` running on a computer named `hiphop`. The CPU on `hiphop` is an Intel Xeon E3-1220 v3 (Haswell) running at 3.10GHz. This CPU does not support hyperthreading. It does support Turbo Boost but `/sys/devices/system/cpu/intel_pstate/no_turbo` was set to 1, disabling Turbo Boost. `hiphop` has 32GB of RAM and runs Ubuntu 16.04. Benchmarks used `./do-part`, which ran on one core of the CPU. The compiler list was reduced to just `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv`.

NIST says that the "NIST PQC Reference Platform" is "an Intel x64 running Windows or Linux and supporting the GCC compiler." `hiphop` is an Intel x64 running Linux and supporting the GCC compiler. Beware, however, that different Intel CPUs have different cycle counts.

The hardware design was synthesized for and measured on a medium-sized Altera Stratix V FPGA (5SGXEA7N).

5.3 Time

`mceliece8192128` software: Encapsulation took slightly under 300000 cycles. Specifically, the median of 31 timings in the first run was 296036 cycles; the median of 31 timings in the second run was 295392 cycles; and the median of 31 timings in the third run was 295932 cycles.

Decapsulation took slightly over 450000 cycles. Specifically, the three medians were 458556 cycles, 458476 cycles, and 458340 cycles.

Key generation took billions of cycles, with medians of 4010278828 cycles, 6008245724 cycles (about 2 seconds), and 4005886024 cycles. Each key-generation attempt took about 2 billion cycles.

`mceliece8192128` hardware: Each key-generation attempt takes 1173750 cycles, which is 5.08ms with the FPGA running at 231MHz. Decoding takes 17140 cycles, which is 0.074ms with the FPGA running at 231MHz.

`mceliece6960119` hardware: Each key-generation attempt takes 966400 cycles, which is 3.85ms with the FPGA running at 248MHz. Decoding takes 17055 cycles, which is 0.060ms with the FPGA running at 248MHz.

5.4 Sizes of inputs and outputs

`mceliece8192128` uses 1357824-byte public keys, 14080-byte private keys, 240-byte ciphertexts, and 32-byte session keys.

`mceliece6960119` uses 1047319-byte public keys, 13908-byte private keys, 226-byte cipher-

texts, and 32-byte session keys.

5.5 Area

On the medium-sized Altera Stratix V FPGA described above, the `mceliece8192128` hardware design takes 227,750 registers (flip-flops), 129,059 ALMs (55% of available logic resources), 1,126 RAM blocks (44% of available on-chip RAM), and 4 DSP blocks (1.6% of available DSPs). The `mceliece6960119` hardware design takes 223,232 registers (flip-flops), 121,806 ALMs (52% of logic resources), 961 RAM blocks (38% of available on-chip RAM), and 6 DSP blocks (2.3% of available DSPs). Note that this includes only key generation and decoding; full decapsulation and encapsulation will use more space, for example for hashing.

5.6 How parameters affect performance

The ciphertext size is $n - k$ bits. Normally the rate $R = k/n$ is chosen around 0.8 (see Section 8), so the ciphertext size is around $0.2n$ bits, i.e., $n/40$ bytes, plus 32 bytes for confirmation.

The public-key size is $k(n-k)$ bits. For $R \approx 0.8$ this is around $0.16n^2$ bits, i.e., $n^2/50$ bytes.

Generating the public key uses $n^{3+o(1)}$ operations with standard Gaussian elimination. There are asymptotically faster matrix algorithms. Private-key operations use just $n^{1+o(1)}$ operations with standard algorithms.

6 Expected strength (2.B.4) in general

This submission is designed and expected to provide IND-CCA2 security.

See Section 7 for the quantitative security of our two suggested parameter sets, and Section 8 for analysis of known attacks. The rest of this section analyzes the KEM from a provable-security perspective.

6.1 Provable-security overview

In general, a security theorem for a cryptographic system C states that an attack $\mathcal{A}$ of type T against C implies an attack $\mathcal{A}'$ against an underlying problem P. Here are four important ways to measure the quality of a security theorem:

- The security of the underlying problem P. The theorem is useless if P is easy to break, and the value of the theorem is questionable if the security of P has not been thoroughly studied.

- The "tightness" of the theorem: i.e., the closeness of the efficiency of $\mathcal{A}'$ to the efficiency of $\mathcal{A}$. If $\mathcal{A}'$ is much less efficient than $\mathcal{A}$ then the theorem does not rule out the possibility that C is much easier to break than P.
- The type T of attacks covered by the theorem. The theorem does not rule out attacks of other types.
- The level of verification of the proof.

Our original plan was to present a KEM with a theorem of the folowing type:

- P is exactly the thoroughly studied inversion (OW-CPA) problem for McEliece's original 1978 system.
- The theorem is extremely tight.
- The theorem covers all IND-CCA2 "ROM" (Random-Oracle Model) attacks. Roughly, an attack of this type is an IND-CCA2 attack that works against any hash function H, given access to an oracle that computes H on any input.
- The proof was already published by Dent [23, Theorem 8] fourteen years ago. The proof is not very complicated, and should be within the range of current techniques for computer verification of proofs.

However, a very recent paper by Saito, Xagawa and Yamakawa [48] indicates that it is possible—without sacrificing tightness—to expand the attack type T from all IND-CCA2 ROM attacks to all IND-CCA2 "QROM" (Quantum Random-Oracle Model) attacks. Roughly, an attack of this type is an IND-CCA2 attack that works against any hash function H, given access to an oracle that computes H on a *quantum superposition* of inputs.

An obstacle here is that Dent's theorem and the Saito–Xagawa–Yamakawa theorem are stated for different KEMs. Another obstacle is that, while Dent's theorem is stated with OW-CPA as the sole assumption, the Saito–Xagawa–Yamakawa theorem is stated with additional assumptions.

To obtain the best of both worlds, we have designed a KEM that combines Dent's framework with the Saito–Xagawa–Yamakawa framework, with the goal of allowing *both* proof techniques to apply. This has created a temporary sacrifice in the level of verification, but we expect that complete proofs will be written and checked by the community in under a year.

6.2 Abstract conversion

Abstractly, we are building a correct KEM given a correct deterministic PKE. We want the KEM to achieve IND-CCA2 security, and we want this to be proven to the extent possible, assuming that the PKE achieves OW-CPA security.

The PKE functionality is as follows. There is a set of public keys, a set of private keys, a set of plaintexts, and a set of ciphertexts. There is a key-generation algorithm KeyGen that produces a public key and a private key. There is a deterministic encryption algorithm $\mathsf{Encrypt}$ that, given a plaintext and a public key, produces a ciphertext. There is a decryption algorithm $\mathsf{Decrypt}$ that, given a ciphertext and a private key, produces a plaintext or a failure symbol $\perp$ (which is not a plaintext). We require that $\mathsf{Decrypt}(\mathsf{Encrypt}(p, K), k) = p$ for every (K, k) output by $\mathsf{KeyGen}()$ and every plaintext p.

We emphasize that $\mathsf{Encrypt}$ is not permitted to randomize its output: in other words, any randomness used to produce a ciphertext must be in the plaintext recovered by decryption. We also emphasize that $\mathsf{Decrypt}$ is not permitted to fail on valid ciphertexts; even a tiny failure probability is not permitted. These requirements are satisfied by the PKE in this submission, and the literature indicates that these requirements are helpful for security proofs.

In this level of generality, our KEM is defined in two modular layers as follows, using three hash functions H_0, H_1, H_2. These hash functions can be modeled in proofs as independent random oracles. If the hash output spaces are the same then this is equivalent to defining $\mathsf{H}_i(x) = \mathsf{H}(i, x)$ for a single random oracle H, since the input spaces are disjoint.

First layer. Write X for the original correct deterministic PKE. We define a modified PKE $X_2 = \textsc{ConfirmPlaintext}(X, \mathsf{H}_2)$ as follows. This modified PKE is also a correct deterministic PKE.

The modified key-generation algorithm KeyGen_2 is the same as the original key-generation algorithm KeyGen. The set of public keys is the same, and the set of private keys is the same.

The modified encryption algorithm $\mathsf{Encrypt}_2$ is defined by $\mathsf{Encrypt}_2(p, K) = (\mathsf{Encrypt}(p, K), \mathsf{H}_2(p))$. The set of plaintexts is the same, and the modified set of ciphertexts consists of pairs of original ciphertexts and hash values.

Finally, the modified decryption algorithm $\mathsf{Decrypt}_2$ is defined by $\mathsf{Decrypt}_2((C, h), k) = \mathsf{Decrypt}(C, k)$.

Note that $\mathsf{Decrypt}_2$ does not check hash values: changing (C, h) to a different (C, h') produces the same output from $\mathsf{Decrypt}_2$. There was also no requirement for the original PKE X to recognize invalid ciphertexts.

Second layer. We define a KEM $\textsc{RandomizeSessionKeys}(X_2, \mathsf{H}_1, \mathsf{H}_0)$ as follows, given a correct deterministic PKE X_2 with algorithms $\mathsf{KeyGen}_2, \mathsf{Encrypt}_2, \mathsf{Decrypt}_2$. This KEM is a correct KEM.

Key generation:

1. Compute $(K, k) \leftarrow \mathsf{KeyGen}_2()$.

2. Choose a uniform random plaintext s.
3. Output K as the public key, and (k, K, s) as the private key.

Encapsulation, given a public key K:

1. Choose a uniform random plaintext p.
2. Compute $C \leftarrow \mathsf{Encrypt}_2(p, K)$.
3. Output C as the ciphertext, and $\mathsf{H}_1(p, C)$ as the session key.

Decapsulation, given a ciphertext C and a private key (k, K, s):

1. Compute $p' \leftarrow \mathsf{Decrypt}_2(C, k)$.
2. If $p' = \perp$, set $p' \leftarrow s$ and $b \leftarrow 0$. Otherwise set $b \leftarrow 1$.
3. Compute $C' \leftarrow \mathsf{Encrypt}_2(p', K)$.
4. If $C \neq C'$, set $p' \leftarrow s$ and $b \leftarrow 0$.
5. Output $\mathsf{H}_b(p', C)$ as the session key.

In other words:

- If there exists a plaintext p such that $C = \mathsf{Encrypt}_2(p, K)$, then decapsulation outputs $\mathsf{H}_1(p, C)$. Indeed, $p' = \mathsf{Decrypt}_2(C, k) = p$ by correctness, so $C' = \mathsf{Encrypt}_2(p, K) = C$ and $b = 1$ throughout, so the output is $\mathsf{H}_1(p, C)$.
- If there does *not* exist a plaintext p such that $C = \mathsf{Encrypt}_2(p, K)$, then decapsulation outputs $\mathsf{H}_0(s, C)$. Indeed, the only way for b to avoid being set to 0 in Step 4 is to have $C' = \mathsf{Encrypt}_2(p', K)$, contradiction; so that step sets p' to s and sets b to 0, and decapsulation outputs $\mathsf{H}_0(s, C)$.

6.3 Non-quantum reduction

The conversion by Dent requires nothing more than OW-CPA security for the underlying PKE, and has a tight IND-CCA2 ROM proof, but for a different KEM. Compared to Dent's KEM, the most significant change in our KEM is the replacement of the $\perp$ output for decapsulation errors with a pseudorandom value. This variant is not new and similar techniques have been used before for code-based schemes (e.g. [45, 46]). We expect that a theorem along the following lines can be proven for our KEM, showing that this difference does not have any sort of negative impact on the security proof.

Expected Theorem 1 *Let $\mathcal{A}$ be an IND-CCA2 adversary against the KEM, running in time t, with advantage ϵ, that performs at most q decapsulation queries and at most q_1 and q_2 queries to the independent uniform random oracles H_1 and H_2 respectively. Then there exists an OW-CPA adversary $\mathcal{A}'$ against the PKE, running in time t', which is successful with probability ϵ', where*

$$t' \leq t + (q + q_1 + q_2)T,$$
$$\epsilon' \geq \epsilon - \frac{q}{2^{\ell_2}} - \frac{q}{\#M},$$

where T is the running time of encapsulation, ℓ_2 is the number of bits of H_2 output, and $\#M$ is the size of the plaintext space.

We now indicate the modifications that need to be made in the proof of [23, Theorem 8]. First of all, the auxiliary table used by the algorithm simulating H_1 (called $KDFList$ in [23]) now contains entries of the type (x_0, x_1, x_2, K) to reflect the different form of the input. The simulator works in exactly the same way, checking the table for previously queried values and outputting a randomly-generated value for K otherwise. Then, we have to modify the response to decapsulation queries. These receive the same input as in [23], and the simulator behaves similarly. It first checks if there exists a preimage p that was already queried by the hash simulator for H_2 and is consistent with the ciphertext. But now, the simulator has to output a value for K even if this check fails: it will simply call the key-generating simulator for $\mathsf{H}_0(s, C)$ rather than $\mathsf{H}_1(p, C)$, where s is an independently generated element as in an honest run of the key generation algorithm. This modification has no impact on the simulation and the adversary learns no more than if it would have received $\perp$ instead. Note that the game is still halted if the adversary attempts to query the simulator on the challenge ciphertext.

Apart from these modifications, the proof is expected to proceed in the same way, generating the same probability bound. The probability bound is a consequence of one of two events occurring, none of which are impacted by the above modifications: the probability of the adversary querying the decapsulation oracle on the challenge ciphertext before this is generated, or querying it on the encapsulation of a string for which the hash oracle hasn't been queried.

6.4 Quantum reduction

As noted above, Saito, Xagawa, and Yamakawa very recently introduced a KEM construction "XYZ" with a tight QROM theorem [48, Theorem 5.2]. This theorem, like Dent's theorem, requires the underlying PKE to be correct (no decryption error) and deterministic. It also makes a stronger security assumption regarding the PKE: the PKE is required to satisfy a new notion of security called PR-CPA, which guarantees that encryption keys and ciphertexts can be indistinguishably replaced by "fake", randomly-generated equivalents. More precisely,

to be considered PR-CPA secure, an encryption scheme needs to satisfy the following three requirements:

- *PR-key security*: adversary has negligible advantage to distinguish a real public key from a fake one.
- *PR-ciphertext security*: adversary has negligible advantage to distinguish a real ciphertext from a fake one when using a fake public key.
- *Statistical disjointness*: negligible probability that a fake ciphertext is in the range of a real ciphertext obtained via a fake key.

See [48, Definition 3.1].

Our KEM construction has two differences from XYZ. First, there is an extra hash value in the ciphertext. Second, the ciphertext is an extra input to the hash used to compute the session key. We expect that a QROM theorem can be proven for our KEM as a composition of the following two steps.

Step 1: Reduce to passive attacks. The proof in [48] can be decomposed into two parts. The first part shows that decapsulation does not reveal any additional information: i.e., all attacks are as difficult as passive attacks.

The original proof of the first part proceeds as follows. If decryption fails or reencryption produces a different ciphertext, XYZ decapsulation outputs $\mathsf{H}_0(s, C)$. The proof simulates $\mathsf{H}_0(s, C)$ with $\mathsf{H}_q(C)$, where H_q (using the notation from [48]) is a random oracle.

If decryption succeeds and reencryption produces the same ciphertext, XYZ decapsulation outputs $\mathsf{H}_1(p)$. The proof redefines $\mathsf{H}_1(p)$ as $\mathsf{H}_q(\mathsf{Encrypt}(p, K))$; this does not change the attack success probability, since H_1 is again a random oracle. It is crucial to understand that this is valid only since the attack doesn't have access to H_q—except via decapsulation failures, but those are disjoint inputs to H_q.

Now decapsulation outputs $\mathsf{H}_q(C)$ for *all* ciphertexts C, whether C itself is valid or invalid. The attack using this decapsulation oracle has the same output as an attack that instead uses an oracle for its own randomly chosen H_q.

For our KEM construction, decapsulation outputs $\mathsf{H}_1(p, C)$ in the success case rather than $\mathsf{H}_1(p)$. We proceed analogously. First simulate $\mathsf{H}_0(s, C)$ with $\mathsf{H}_q(C, C)$, where H_q is a random oracle. Then redefine $\mathsf{H}_1(p, C)$ as $\mathsf{H}_q(\mathsf{Encrypt}(p, K), C)$; this is again a random oracle, and again the inputs to H_q are disjoint between the valid and invalid cases. Finally, decapsulation maps C to $\mathsf{H}_q(C, C)$ in all cases, regardless of the validity of C.

Step 2: Invoke the PR-CPA assumptions. The second part of the proof in [48] shows that, given the PR-CPA assumptions, passive attacks are infeasible. We expect this part of the proof to apply directly to our KEM construction, invoking the PR-CPA assumptions for the modified PKE.

We expect the PR-CPA assumptions for the modified PKE to be provable as follows from the same assumptions for the original PKE. PR-key security is the same property for the two PKEs, since $\mathsf{KeyGen}_2 = \mathsf{KeyGen}$. PR-ciphertext security for the modified PKE for a random oracle H_2 should follow from PR-ciphertext security for the original PKE. Statistical disjointness for the modified PKE is implied by statistical disjointness for the original PKE, since identical ciphertexts for the modified PKE begin with identical ciphertexts for the original PKE.

Plausibility of the PR-CPA assumptions for Classic McEliece. As noted in Section 4, there is a long literature on information-set decoding, the fastest inversion attack known against the McEliece PKE. This literature generally treats the problem of decoding *uniform random* codes, and frequently observes that—in experiments—the attacks behave the same way for uniform random binary Goppa codes. This behavior of attacks is sometimes formalized and generalized to a hypothesis about all fast algorithms: namely, the generator matrix (or parity-check matrix) for a uniform random binary Goppa code is hard to distinguish from the generator matrix (or parity-check matrix) for a uniform random code.

This hypothesis is the PR-key security assumption for this PKE. Cryptanalysis of this hypothesis has focused mainly on key-recovery attacks, although, as noted earlier, there is a paper [26] explicitly studying distinguishing attacks. None of these attacks threaten PR-key security for our suggested parameters. This is not the same as saying that PR-key security has been studied as thoroughly as OW-CPA security. Similarly, existing cryptanalysis of PR-ciphertext security has focused mainly on inversion attacks. Statistical disjointness, a statement about the sparsity of the range of the encryption function compared to the ciphertext space, may be provable: a similar property "γ-uniformity" was proved by Cayrel, Hoffmann, and Persichetti [14].

To summarize, there is already some work that can be viewed as studying the PR-CPA assumptions. On the other hand, the assumptions go beyond the thoroughly studied McEliece OW-CPA problem. A theorem assuming PR-CPA security, as in [48], is thus not a replacement for a theorem assuming merely OW-CPA security, as in [23, Theorem 8]. Note that the reduction to passive attacks is independent of this choice of assumption.

6.5 Relating the abstract conversion to the specification

The general specification in Section 2 can be viewed as the result of the following four steps:

- Start with the McEliece PKE. This PKE is correct and deterministic, and its OW-CPA security has been thoroughly studied.
- Switch to Niederreiter's dual PKE. This PKE is correct and deterministic, and its OW-CPA security is tightly implied by the OW-CPA security of the McEliece PKE.
- Obtain a KEM by applying the ConfirmPlaintext conversion, followed by the

RandomizeSessionKeys conversion. This KEM is correct, and its IND-CCA2 security is the topic of the previous subsections.

- Apply three further optimizations discussed below. These optimizations preserve correctness, and they do not affect the IND-CCA2 security analysis.

The first optimization is as follows. Checking whether $C = \mathsf{Encrypt}_2(p', K)$, with the knowledge that $p' = \mathsf{Decrypt}_2(C, k)$, does not necessarily require a full $\mathsf{Encrypt}_2$ computation. In particular, in Section 2, the decoding procedure is already guaranteed to output

- a weight-t vector whose syndrome is the input if such a vector exists, or
- $\perp$ otherwise.

Checking whether $C = \mathsf{Encrypt}_2(p', K)$ is thus a simple matter of checking $\mathsf{H}_2(p')$.

The second optimization is as follows. The KEM private key (k, K, s) does not necessarily need as much space as the space for k plus the space for K plus the space for s. For example, if K can be computed efficiently from k, then it can be recomputed on demand, or optionally cached. In Section 2, the situation is even simpler: decapsulation, with the first optimization, does not look at K, so K is simply eliminated from the KEM private key.

The third optimization is that s is generated from a larger space than the plaintext space: it is simpler to generate a uniform random n-bit string than to generate a uniform random weight-t n-bit string. The set of s enters into the security analysis solely for the indistinguishability of $\mathsf{H}_0(s, C)$ from uniform random.

7 Expected strength (2.B.4) for each parameter set

7.1 Parameter set `kem/mceliece6960119`

IND-CCA2 KEM, Category 5.

7.2 Parameter set `kem/mceliece8192128`

IND-CCA2 KEM, Category 5.

8 Analysis of known attacks (2.B.5)

8.1 Information-set decoding, asymptotically

There is a long literature studying algorithms to invert the McEliece PKE. See Section 4.1.

The fastest attacks known use information-set decoding (ISD). The simplest form of ISD, from 1962 Prange [47], tries to guess an error-free information set in the code. An information set is, by definition, a set of positions that determines an entire codeword. The set is error-free, by definition, if it avoids all of the error positions in the "received word", i.e., the ciphertext; then the ciphertext at those positions is exactly the codeword at those positions. The attacker determines the rest of the codeword by linear algebra, and sees whether the attack succeeded by checking whether the error weight is t.

One expects a random set of k positions to be an information set with reasonable probability, the same 29% mentioned earlier. However, the chance of the set being error-free drops rapidly as the number of errors increases. The following asymptotic statement holds for any real number R with $0 < R < 1$: if the code dimension k is $(R + o(1))n$, and the number of errors t is $\Theta(n/\log n)$, then the chance of a set being error-free is $(1 - R + o(1))^t$ as $n \to \infty$. The cost of ISD is thus $(1/(1 - R) + o(1))^t$.

Subsequent improvements to ISD have affected the $o(1)$ but have not changed the constant $1/(1 - R)$. See generally [10] and [54].

In the McEliece system, t is asymptotically $(1 - R + o(1))n/\lg n$, so the assumption $t \in \Theta(n/\log n)$ holds.[3] To summarize, the (OW-CPA) security level of the McEliece system against all of these attacks is the $n/\lg n$ power of $1/(1 - R)^{1-R} + o(1)$.

Meanwhile the ciphertext size is $(1 - R + o(1))n$ bits, and the key size is $(R(1 - R) + o(1))n^2$ bits. Security level 2^b thus uses key size $(C_0 + o(1))b^2(\lg b)^2$ where $C_0 = R/(1-R)(\lg(1-R))^2$. This C_0 reaches its minimum value, approximately 0.7418860694, when R is approximately 0.7968121300.

8.2 Information-set decoding, concretely

We emphasize that $o(1)$ does not mean 0: it means something that converges to 0 as $n \to \infty$. More detailed attack-cost evaluation is therefore required for any particular parameters.

Our smaller parameter set `mceliece6960119` takes $m = 13$, $n = 6960$, and $t = 119$. This parameter set was proposed in the attack paper [8] that broke the original McEliece parameters $(10, 1024, 50)$.

That paper reported that its attack uses $2^{266.94}$ bit operations to break the $(13, 6960, 119)$ parameter set. Subsequent ISD variants have reduced the number of bit operations considerably below 2^{256}. However:

- None of these analyses took into account the costs of memory access. A closer look shows that the attack in [8] is bottlenecked by random access to a huge array (much

[3] Beware that some ISD papers instead measure their results for much larger $t \in \Theta(n)$, such as "half of the GV distance". This dramatically increases cost from $2^{\Theta(n/\lg n)}$ to $2^{\Theta(n)}$. For example, [38] two years ago reports $\mathcal{O}(2^{0.0473n})$ when t is half of the GV distance, compared to $\mathcal{O}(2^{0.0576n})$ from Prange 55 years ago. As these numbers illustrate, this inflation of t also makes differences between algorithms more noticeable. Such large error rates are of interest in coding theory but are not relevant to the McEliece system.

larger than the public key being attacked), and that subsequent ISD variants use even more memory. The same amount of hardware allows much more parallelism in attacking, e.g., AES-256.

- Known quantum attacks multiply the security level of both ISD and AES by an asymptotic factor $0.5 + o(1)$, but a closer look shows that the application of Grover's method to ISD suffers much more overhead in the inner loop.

We expect that switching from a bit-operation analysis to a cost analysis will show that this parameter set is more expensive to break than AES-256 pre-quantum and much more expensive to break than AES-256 post-quantum.

8.3 Key recovery

A different inversion strategy is to find the private key $(g, \alpha_1, \ldots, \alpha_n)$. As noted earlier, one should not think that this is as difficult as a brute-force search: one can determine the sequence $(\alpha_1, \ldots, \alpha_n)$ from g and the *set* $\{\alpha_1, \ldots, \alpha_n\}$, or alternatively determine g from $(\alpha_1, \ldots, \alpha_n)$. See generally [36], [28], and [43]. The number of choices of g is more than 2^{1500} for our smaller parameter set and more than 2^{1600} for our larger parameter set. Known symmetries provide only a small speedup. The number of choices of $(\alpha_1, \ldots, \alpha_n)$ is much larger. Our smaller parameter set has an extra defense here, namely that there are a huge number of possibilities for the set $\{\alpha_1, \ldots, \alpha_n\}$.

In a multi-message attack scenario, the cost of finding the private key is spread across many messages. There are also faster multi-message attacks that do not rely on finding the private key; see, e.g., [31] and [51]. Rather than analyzing multi-message security in detail, we rely on the general fact that attacking T targets cannot gain more than a factor T. Our expected security levels are so high that this is not a concern for any foreseeable value of T.

8.4 Chosen-ciphertext attacks

A traditional approach to chosen-ciphertext attacks against the McEliece system is to add (say) two errors to a ciphertext $Gm + e$. This is equivalent to adding two errors to e. Decryption succeeds if and only if the resulting error vector has weight t, i.e., exactly one of the two error positions was already in e. It is straightforward to find e from the pattern of decryption failures. See, e.g., [57]. For a Niederreiter ciphertext He, one similarly adds two errors to e by adjusting He appropriately.

There are two reasons that these attacks do not work against our submission. First, KEM decapsulation forces the ciphertext to include a hash of e as a confirmation, and the attacker has no way to compute the hash of a modified version of e without knowing e in the first place. Second, the KEM does not reveal decryption failures: the modified ciphertext will produce an unpredictable session key, whether or not the modified error vector has weight t.

The confirmation allows attackers to check possibilities for e by checking their hashes. However, this is much less efficient than ISD.

9 Advantages and limitations (2.B.6)

The central advantage of this submission is security. See the design rationale.

Regarding efficiency, the use of random-looking linear codes with no visible structure forces public-key sizes to be on the scale of a megabyte for quantitatively high security: the public key is a full (generator/parity-check) matrix. Key-generation software is also not very fast. Applications must continue using each public key for long enough to handle the costs of generating and distributing the key.

There are, however, some compensating efficiency advantages. Encapsulation and decapsulation are reasonably fast in software, and impressively fast in hardware, due to the simple nature of the objects (binary vectors) and operations (such as binary matrix-vector multiplications). Key generation is also quite fast in hardware. The hardware speeds of key generation and decoding are already demonstrated by our FPGA implementation. Encapsulation takes only a single pass over a public key, allowing large public keys to be streamed through small coprocessors and small devices.

Furthermore, the ciphertexts are unusually small for post-quantum cryptography: under 256 bytes for our suggested high-security parameter sets. This allows ciphertexts to fit comfortably inside single network packets. The small ciphertext size can be much more important for total traffic than the large key size, depending on the ratio between how often keys are sent and how often ciphertexts are sent. System parameters can be adjusted for even smaller ciphertexts.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 53–70. USENIX Association, 2016.

[2] Marco Baldi, Franco Chiaraluce, Roberto Garello, and Francesco Mininni. Quasi-cyclic low-density parity-check codes in the McEliece cryptosystem. In *Proceedings of IEEE International Conference on Communications, ICC 2007, Glasgow, Scotland, 24-28 June 2007*, pages 951–956. IEEE, 2007.

[3] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1+1=0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT*

2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings, volume 7237 of *Lecture Notes in Computer Science*, pages 520–536. Springer, 2012.

[4] Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. Reducing key length of the McEliece cryptosystem. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 77–97. Springer, 2009.

[5] Daniel J. Bernstein. Grover vs. McEliece. In Sendrier [50], pages 73–80.

[6] Daniel J. Bernstein. Some small suggestions for the Intel instruction set, 2014. https://blog.cr.yp.to/20140517-insns.html.

[7] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013.

[8] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. In Johannes A. Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.

[9] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: Ball-collision decoding. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 743–760. Springer, 2011.

[10] Daniel J. Bernstein, Tanja Lange, Christiane Peters, and Henk C. A. van Tilborg. Explicit bounds for generic decoding algorithms for code-based cryptography. In *Pre-proceedings of WCC 2009*, pages 168–180, 2009.

[11] Anne Canteaut and Herve Chabanne. A further improvement of the work factor in an attempt at breaking McEliece's cryptosystem. In Pascale Charpin, editor, *Livre des résumés—EUROCODE 94, Abbaye de la Bussière sur Ouche, France, October 1994*, 1994. https://hal.inria.fr/inria-00074443.

[12] Anne Canteaut and Florent Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Trans. Information Theory*, 44(1):367–378, 1998.

[13] Anne Canteaut and Nicolas Sendrier. Cryptanalysis of the original McEliece cryptosystem. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology - ASIACRYPT*

'98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings, volume 1514 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 1998.

[14] Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. Efficient implementation of a CCA2-secure variant of McEliece using generalized Srivastava codes. In Marc Fischlin, Johannes A. Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2012.

[15] Herve Chabanne and B. Courteau. Application de la méthode de décodage itérative d'Omura à la cryptanalyse du système de McEliece, 1993. Université de Sherbrooke, Rapport de Recherche, number 122.

[16] Florent Chabaud. Asymptotic analysis of probabilistic algorithms for finding short codewords. In Paul Camion, Pascale Charpin, and Sami Harari, editors, *Eurocode '92: proceedings of the international symposium on coding theory and applications held in Udine, October 23–30, 1992*, pages 175–183. Springer, 1993.

[17] Tung Chou. McBits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017.

[18] George C. Clark, Jr. and J. Bibb Cain. *Error-correcting coding for digital communication*. Plenum, 1981.

[19] John T. Coffey and Rodney M. Goodman. The complexity of information set decoding. *IEEE Transactions on Information Theory*, 35:1031–1037, 1990.

[20] John T. Coffey, Rodney M. Goodman, and P. Farrell. New approaches to reduced complexity decoding. *Discrete and Applied Mathematics*, 33:43–60, 1991.

[21] Alain Couvreur, Ayoub Otmani, and Jean-Pierre Tillich. Polynomial time attack on Wild McEliece over quadratic extensions. *IEEE Trans. Information Theory*, 63(1):404–427, 2017.

[22] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1):167–226, January 2004.

[23] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2003.

[24] Ilya I. Dumer. Two decoding algorithms for linear codes. *Problemy Peredachi Informatsii*, 25:24–32, 1989.

[25] Ilya I. Dumer. On minimum distance decoding of linear codes. In Grigori A. Kabatianskii, editor, *Fifth joint Soviet-Swedish international workshop on information theory, Moscow, 1991*, pages 50–52, 1991.

[26] Jean-Charles Faugère, Valérie Gauthier-Umaña, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. A distinguisher for high-rate McEliece cryptosystems. *IEEE Trans. Information Theory*, 59(10):6830–6844, 2013.

[27] Matthieu Finiasz and Nicolas Sendrier. Security bounds for the design of code-based cryptosystems. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2009.

[28] J. K. Gibson. Equivalent Goppa codes and trapdoors to McEliece's public key cryptosystem. In Donald W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 517–521. Springer, 1991.

[29] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. *IACR Cryptology ePrint Archive*, 2013:162, 2013. https://eprint.iacr.org/2013/162.

[30] Gernot Heiser. For safety's sake: we need a new hardware-software contract! *IEEE Design and Test*, 2017. To appear.

[31] Thomas Johansson and Fredrik Jönsson. On the complexity of some cryptographic problems based on the general decoding problem. *IEEE Trans. Information Theory*, 48(10):2669–2678, 2002.

[32] Evgueni A. Krouk. Decoding complexity bound for linear block codes. *Problemy Peredachi Informatsii*, 25:103–107, 1989.

[33] Pil Joong Lee and Ernest F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In Christoph G. Günther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer, 1988.

[34] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Information Theory*, 34(5):1354–1359, 1988.

[35] Gavriela Freund Lev, Nicholas Pippenger, and Leslie G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. Computers*, 30(2):93–100, 1981.

[36] Pierre Loidreau and Nicolas Sendrier. Weak keys in the McEliece public-key cryptosystem. *IEEE Trans. Information Theory*, 47(3):1207–1211, 2001.

[37] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $2^{0.054n}$. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2011.

[38] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 203–228. Springer, 2015.

[39] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. Technical report, NASA, 1978. http://ipnpr.jpl.nasa.gov/progressreport2/42-44/44N.PDF.

[40] Rafael Misoczki and Paulo S. L. M. Barreto. Compact McEliece keys from Goppa codes. In Michael J. Jacobson Jr., Vincent Rijmen, and Rei Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2009.

[41] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*, pages 2069–2073. IEEE, 2013.

[42] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[43] Raphael Overbeck and Nicolas Sendrier. Code-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 95–145. Springer Berlin Heidelberg, 2009.

[44] Edoardo Persichetti. Compact McEliece keys based on quasi-dyadic Srivastava codes. *J. Mathematical Cryptology*, 6(2):149–169, 2012.

[45] Edoardo Persichetti. Secure and anonymous hybrid encryption from coding theory. In Philippe Gaborit, editor, *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, pages 174–187, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[46] Edoardo Persichetti. Code-based key encapsulation from McEliece's cryptosystem. 2017. http://arxiv.org/abs/1706.06306.

[47] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, IT-8:S5–S9, 1962.

[48] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. https://eprint.iacr.org/2017/1005.

[49] Nicolas Sendrier. Finding the permutation between equivalent linear codes: The support splitting algorithm. *IEEE Trans. Information Theory*, 46(4):1193–1203, 2000.

[50] Nicolas Sendrier, editor. *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, volume 6061 of *Lecture Notes in Computer Science*. Springer, 2010.

[51] Nicolas Sendrier. Decoding one out of many. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 2011.

[52] Jacques Stern. A method for finding codewords of small weight. In Gérard D. Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 1988.

[53] Falko Strenzke. A timing attack against the secret permutation in the McEliece PKC. In Sendrier [50], pages 95–107.

[54] Rodolfo Canto Torres and Nicolas Sendrier. Analysis of information set decoding for a sub-linear error weight. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, volume 9606 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2016.

[55] Johan van Tilburg. On the McEliece public-key cryptosystem. In Shafi Goldwasser, editor, *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1988, Proceedings*, volume 403 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 1988.

[56] Johan van Tilburg. *Security-analysis of a class of cryptosystems based on linear error-correcting codes*. PhD thesis, Technische Universiteit Eindhoven, 1994.

[57] Eric R. Verheul, Jeroen M. Doumen, and Henk C. A. van Tilborg. Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem. In Mario Blaum, Patrick G. Farrell, and Henk C. A. van Tilborg, editors, *Information, coding and mathematics*, volume 687 of *Kluwer International Series in Engineering and Computer Science*, pages 99–119. Kluwer, 2002.

[58] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter cryptosystem using binary Goppa codes. Paper in submission, http://caslab.csl.yale.edu/code/niederreiter/.

A Statements

These statements "must be mailed to Dustin Moody, Information Technology Laboratory, Attention: Post-Quantum Cryptographic Algorithm Submissions, 100 Bureau Drive – Stop 8930, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, or can be given to NIST at the first PQC Standardization Conference (see Section 5.C)."

First blank in submitter statement: full name. Second blank: full postal address. Third, fourth, and fifth blanks: name of cryptosystem. Sixth and seventh blanks: describe and enumerate or state "none" if applicable.

First blank in patent statement: full name. Second blank: full postal address. Third blank: enumerate. Fourth blank: name of cryptosystem.

First blank in implementor statement: full name. Second blank: full postal address. Third blank: full name of the owner.

A.1 Statement by Each Submitter

I, _____, of _____, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that (check one):

- *I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ OR (check one or both of the following):*
  - *to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ may be covered by the following U.S. and/or foreign patents:*

    _____
  - *I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations:*

    _____

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

A.2 Statement by Patent (and Patent Application) Owner(s)

If there are any patents (or patent applications) identified by the submitter, including those held by the submitter, the following statement must be signed by each and every owner, or each owner's authorized representative, of each patent and patent application identified.

I, _____, of _____, am the owner or authorized representative of the owner (print full name, if different than the signer) of the following patent(s) and/or patent application(s): _____

and do hereby commit and agree to grant to any interested party on a worldwide basis, if the cryptosystem known as _____ is selected for standardization, in consideration of its evaluation and selection by NIST, a non-exclusive license for the purpose of implementing the standard (check one):

- *without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination, OR*
- *under reasonable terms and conditions that are demonstrably free of any unfair discrimination.*

I further do hereby commit and agree to license such party on the same basis with respect to any other patent application or patent hereafter granted to me, or owned or controlled by me, that is or may be necessary for the purpose of implementing the standard.

I further do hereby commit and agree that I will include, in any documents transferring ownership of each patent and patent application, provisions to ensure that the commitments and assurances made by me are binding on the transferee and any future transferee.

I further do hereby commit and agree that these commitments and assurances are intended by me to be binding on successors-in-interest of each patent and patent application, regardless of whether such provisions are included in the relevant transfer documents.

I further do hereby grant to the U.S. Government, during the public review and the evaluation process, and during the lifetime of the standard, a nonexclusive, nontransferrable, irrevocable, paid-up worldwide license solely for the purpose of modifying my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability) for incorporation into the standard.

Signed:

Title:

Date:

Place:

A.3 Statement by Reference/Optimized Implementations' Owner(s)

The following must also be included:

I, _____, _____, am the owner or authorized representative of the owner _____ of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

CRYSTALS-Dilithium

Algorithm Specifications and Supporting Documentation

Léo Ducas[1], Eike Kiltz[2], Tancrède Lepoint[3], Vadim Lyubashevsky[4], Peter Schwabe[5], Gregor Seiler[6] and Damien Stehlé[7]

[1] CWI, Netherlands
[2] Ruhr Universität Bochum, Germany
[3] SRI International, USA
[4] IBM Research – Zurich, Switzerland
[5] Radboud University, Netherlands
[6] IBM Research – Zurich, Switzerland
[7] ENS de Lyon, France

November 30, 2017

1 Introduction

We present the digital signature scheme Dilithium, whose security is based on the hardness of finding short vectors in lattices. Our scheme was designed with the following criteria in mind:

Simple to implement securely. The most compact lattice-based signature schemes [DDLL13, DLP14] crucially require the generation of secret randomness from the discrete Gaussian distribution. Generating such samples in a way that is secure against side-channel attacks is highly non-trivial and can easily lead to insecure implementations, as demonstrated in [BHLY16, EFGT17, PBY17]. While it may be possible that a very careful implementation can prevent such attacks, it is unreasonable to assume that a universally-deployed scheme containing many subtleties will always be expertly implemented. Dilithium therefore only uses uniform sampling, as was originally proposed in [Lyu09, GLP12, BG14]. Furthermore all other operations (such as polynomial multiplication and rounding) are easily implemented in constant time.

Be conservative with parameters. Since we are aiming for long-term security, we have analyzed the applicability of lattice attacks from a very favorable, to the attacker, viewpoint. In particular, we are considering quantum algorithms that require virtually as much space as time. Such algorithms are currently unrealistic, and there seem to be serious obstacles in removing the space requirement, but we are allowing for the possibility that improvements may occur in the future.

Minimize the size of public key + signature. Since many applications require the transmission of both the public key and the signature (e.g. certificate chains), we designed our scheme to minimize the sum of these parameters. Under the restriction that we avoid (discrete) Gaussian sampling, to the best of our knowledge, Dilithium has the smallest combination of signature and public key sizes of any lattice-based scheme with the same security levels.

Be modular – easy to vary security. The two operations that constitute nearly the entirety of the signing and verification procedures are expansion of an XOF (we use SHAKE-128 and SHAKE-256), and multiplication in the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$. Highly efficient implementations of our algorithm will therefore need to optimize these operations and make sure that they run in constant time. For all security levels, our scheme uses the same ring with $q = 2^{23} - 2^{13} + 1$ and $n = 256$. Varying security simply involves doing more/less operations over this ring and doing more/less expansion of the XOF. In other words, once an optimized implementation is obtained for some security level, it is almost trivial to obtain an optimized implementation for a higher/lower level.

1.1 Overview of the Basic Approach

The design of the scheme is based on the "Fiat-Shamir with Aborts" approach [Lyu09] and bears most resemblance to the schemes proposed in [GLP12, BG14]. For readers who are unfamiliar with the general framework of such signature schemes, we present a simplified (and less efficient) version of our scheme in Fig. 1. This version is essentially a slightly modified version of the scheme from [BG14]. We will now go through each of its components to give the reader an idea of how such schemes work.

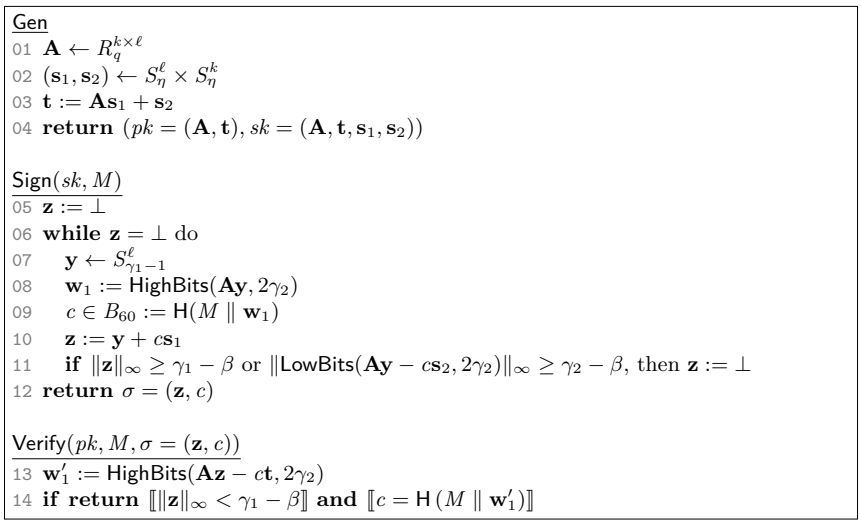


Figure 1: Template for our signature scheme.

Key Generation. The key generation algorithm generates a $k \times \ell$ matrix $\mathbf{A}$ each of whose entries is a polynomial in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. As previously mentioned, we will always have $q = 2^{23} - 2^{13} + 1$ and $n = 256$. Afterwards, the algorithm samples random secret key vectors $\mathbf{s}_1$ and $\mathbf{s}_2$. Each coefficient of these vectors is an element of R_q with small coefficients – of size at most η. Finally, the second part of the public key is computed as $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. All algebraic operations in this scheme are assumed to be over the polynomial ring R_q.

Signing Procedure. The signing algorithm generates a masking vector of polynomials $\mathbf{y}$ with coefficients less than γ_1. The parameter γ_1 is set strategically – it is large enough that the eventual signature does not reveal the secret key (i.e. the signing algorithm is zero-knowledge), yet small enough so that the signature is not easily forged. The signer then computes $\mathbf{A}\mathbf{y}$ and sets $\mathbf{w}_1$ to be the "high-order" bits of the coefficients in this vector. In particular, every coefficient w in $\mathbf{A}\mathbf{y}$ can be written in a canonical way as $w = w_1 \cdot 2\gamma_2 + w_0$ where $|w_0| \leq \gamma_2$; $\mathbf{w}_1$ is then the vector comprising all the w_1's. The challenge c is then created as the hash of the message and $\mathbf{w}_1$. The output c is a polynomial in R_q with exactly 60 ± 1's and the rest 0's. The reason for this distribution is that c has small norm and comes from a domain of size $> 2^{256}$. The potential signature is then computed as $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$.

If $\mathbf{z}$ were directly output at this point, then the signature scheme would be insecure due to the fact that the secret key would be leaked. To avoid the dependency of $\mathbf{z}$ on the secret key, we use rejection sampling. The parameter β is set to be the maximum possible coefficient of $c\mathbf{s}_i$. Since c has 60 ± 1's and the maximum coefficient in $\mathbf{s}_i$ is η, it's easy to see that $\beta \leq 60\eta$. If any coefficient of $\mathbf{z}$ is larger than $\gamma_1 - \beta$, then we reject and restart the signing procedure. Also, if any coefficient of the low-order bits of $\mathbf{A}\mathbf{z} - c\mathbf{t}$ is greater than $\gamma_2 - \beta$, we restart. The first check is necessary for security, while the second is necessary for both security and correctness. The while loop in the signing procedure keeps being repeated until the preceding two conditions are satisfied. The parameters are set such that the expected number of repetitions is not too high (in our instantiations, this number is between 4 and 7).

Verification. The verifier first computes $\mathbf{w}_1'$ to be the high-order bits of $\mathbf{Az} - c\mathbf{t}$, and then accepts if all the coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and if c is the hash of the message and $\mathbf{w}_1'$. Let us look at why verification works, in particular as to why $\mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) = \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$. The first thing to notice is that $\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} - c\mathbf{s}_2$. So all we really need to show is that

$$\mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2) = \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2). \tag{1}$$

The reason for this is that a valid signature will have $\|\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$. And since we know that the coefficients of $c\mathbf{s}_2$ are smaller than β, we know that adding $c\mathbf{s}_2$ is not enough to cause any carries by increasing any low-order coefficient to have magnitude at least γ_2. Thus Eq. (1) is true and the signature verifies correctly.

1.2 Dilithium

The basic template in Fig. 1 is rather inefficient, as is. The most glaring (but easily fixed) inefficiency is that the public key consists of a matrix of $k \cdot \ell$ polynomials, which could have a rather large representation. The obvious fix is to have $\mathbf{A}$ generated from some seed ρ using SHAKE-128, and this is a standard technique. The novelty of Dilithium over the previous schemes is that we also shrink the size of the public key by a factor of 2.5 at the expense of increasing the signature by around 150 bytes. For the recommended security level, the scheme has 2.7KB signatures and 1.5KB public keys.

The main observation for obtaining this very favorable trade-off is that when the verifier computes $\mathbf{w}_1'$ in Line 13, the high-order bits of $\mathbf{Az} - c\mathbf{t}$ do not depend too much on the *low order* bits of $\mathbf{t}$ because $\mathbf{t}$ is being multiplied by a very low-weight polynomial c. In our scheme, some low-order bits of $\mathbf{t}$ are not included in the public key, and so the verifier cannot always correctly compute the high-order bits of $\mathbf{Az} - c\mathbf{t}$. To make up for this, the signer includes some "hints" as part of the signature, which are essentially the carries caused by adding in the product of c with the missing low-order bits of $\mathbf{t}$. With this hint, the verifier is able to correctly compute $\mathbf{w}_1'$.

Additionally, we make our scheme deterministic using the standard technique of adding a seed to the secret key and using this seed together with the message to produce the randomness $\mathbf{y}$ in Line 07. The recent result of Kiltz et al. [KLS17] showed that the fewer different signatures the adversary sees for the same messages, the tighter the reduction is in the quantum random oracle model between the signature scheme and the underlying hardness assumptions. While it's not clear as to whether there is an improved quantum attack for randomized signatures, we suggest the deterministic version as the default option. Our full scheme in Fig. 4 also makes use of basic optimizations such as pre-hashing the message M so as to not rehash it with every signing attempt.

Implementation Considerations. The main algebraic operation performed in the scheme is a multiplication of a matrix $\mathbf{A}$, whose elements are polynomials in $\mathbb{Z}_q[X]/(X^{256}+1)$ by a vector of such polynomials. In our recommended parameter setting, $\mathbf{A}$ is a 5×4 matrix and therefore consists of 20 polynomials. Thus the multiplication $\mathbf{Av}$ involves 20 polynomial multiplications. As in most lattice-based schemes that are based on operations over polynomial rings, we have chosen our ring so that the multiplication operation has a very efficient implementation via the Number Theoretic Transform (NTT), which is just a version of FFT that works over the finite field $\mathbb{Z}_q$ rather than over the complex numbers. To enable the NTT, we needed to choose a prime q so that the group $\mathbb{Z}_q^*$ has an element of order $2n = 512$; or equivalently $q \equiv 1 \pmod{512}$. If r is such an element, then $X^{256}+1 = (X-r)(X-r^3)\cdots(X-r^{511})$ and thus one can equivalently represent any polynomial $a \in \mathbb{Z}_q[X]/(X^{256}+1)$ in its CRT (Chinese Remainder Theorem) form as $(a(r), a(r^3), \ldots, a(r^{2n-1}))$. The advantage of this representation is that the product of

two polynomials is coordinate-wise. Therefore the most expensive parts of polynomial multiplication are the transformations $a \to \hat{a}$ and the inverse $\hat{a} \to a$ – these are the NTT and inverse NTT operations.

The other major time-consuming operation is the expansion of a seed ρ into the polynomial matrix $\mathbf{A}$. The matrix $\mathbf{A}$ is needed for both signing and verification, therefore a good implementation of SHAKE-128 is important for the efficiency of the scheme.

For our AVX2 optimized implementation of the NTT we take a different approach than other fast implementations in that we use integer arithmetic. Although we pack only 4 coefficients into one vector register of 256 bits, which is the same density that is also used by floating point implementations, we can improve on the multiplication speed by about a factor of 2. We achieved this speed-up by carefully scheduling the instructions and interleaving the multiplications and reductions during the NTT so that parts of the multiplication latencies are hidden.

1.3 Comparisons to Other Post-Quantum Signature Schemes

We now give a brief comparison between our signature scheme and other post-quantum signature schemes that we are aware of.

1.3.1 Lattice Schemes

Schemes with Smaller Signature Sizes. The lattice-based digital signatures (with security reductions) that have the smallest signature sizes are [DDLL13, DLP14], which are based on the NTRU assumption. If one adjusts the parameters of [DDLL13] and [DLP14] so that the security is comparable to that of Dilithium[1], the signature sizes will be approximately 1.5KB and 1KB respectively (compared to 2.7KB in Dilithium), while the public key sizes will remain approximately the same as in Dilithium.

The main down-side of [DDLL13, DLP14] is that they intrinsically require the use of (discrete) Gaussian sampling in order to be efficient, which creates several serious real-world issues. The first is that it is not straight-forward to construct a constant-time discrete Gaussian sampler. The second is that mistakes in sampling (discrete) Gaussians are extremely hard to detect in testing; yet even small deviations from the right distribution can lead to complete signing key recovery by the adversary. For this reason, we believe that a universally-used scheme should be very simple to implement in a secure fashion, and we have thus eschewed using anything other than uniform sampling in Dilithium.

Schemes with (Quantum) Security Reductions from (Ring / Module)-LWE Only. The security of Dilithium is, in the quantum random oracle model (QROM), tightly based on the hardness of the standard MLWE and MSIS problems, as well as a "hybrid" SelfTargetMSIS problem that was defined in [KLS17]. For the latter problem, there is a reduction $\mathsf{MSIS} \leq \mathsf{SelfTargetMSIS}$ in the random oracle model via the forking lemma, but not in the quantum random oracle model.

One could construct a version of Dilithium whose security is based entirely on the hardness of Ring-LWE in the QROM, but this scheme would result in a 5X increase in the public key size and a 2X increase in the signature size [KLS17, Table 1]. Furthermore, we would not be able to work over a the ring that supports NTT, which would make signing and verification slower. As we explain in Section 5, the SelfTargetMSIS problem is in fact the lattice version of a problem upon which *tight* security proofs of today's signatures (e.g. Schnorr) using the Fiat-Shamir transform are based. We therefore believe that this assumption is a perfectly sound one to make and avoiding it is not worth the significant cost in speed and output size.

[1] The security in [DDLL13, DLP14] was not set as conservatively as for Dilithium – in particular, sieving attacks were not considered.

1.3.2 Other Post-Quantum Signatures.

Of all the non-lattice post-quantum schemes that we are aware of, Dilithium has the smallest combination of public key and signature size. There may be scenarios, however, that call for one of these values to be very small, while not caring too much about the other. If one would like to minimize the public key size, then hash-based signatures (e.g. [BHH+15]) are a good option because the public key is less than a hundred bytes. The signature length of such signatures, on the other hand, is between 30-40KB, and signing time is around 50X slower than Dilithium. If, on the other hand, one would like to minimize the signature size, then multivariate schemes (see the various comparisons in [CLP+17]) may be of interest. The signature sizes in these schemes are less than one hundred bytes and signing time is noticeably faster. The public keys, on the other hand, are often larger than 100KB.

2 Basic Operations

2.1 Ring Operations

We let R and R_q respectively denote the rings $\mathbb{Z}[X]/(X^n+1)$ and $\mathbb{Z}_q[X]/(X^n+1)$, for q an integer. Throughout this document, the value of n will always be 256 and q will be the prime $8380417 = 2^{23} - 2^{13} + 1$. Regular font letters denote elements in R or R_q (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent column vectors with coefficients in R or R_q. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $\mathbf{v}$, we denote by $\mathbf{v}^T$ its transpose. The boolean operator $[\![\text{statement}]\!]$ evaluates to 1 if statement is true, and to 0 otherwise.

Modular reductions. For an even (resp. odd) positive integer α, we define $r' = r \bmod^{\pm} \alpha$ to be the unique element r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$) such that $r' \equiv r \bmod \alpha$. We will sometimes refer to this as a *centered* reduction modulo q.[2] For any positive integer α, we define $r' = r \bmod^{+} \alpha$ to be the unique element r' in the range $0 \leq r' < \alpha$ such that $r' \equiv r \bmod \alpha$. When the exact representation is not important, we simply write $r \bmod \alpha$.

Sizes of elements. For an element $w \in \mathbb{Z}_q$, we write $\|w\|_\infty$ to mean $|w \bmod^{\pm} q|$. We define the ℓ_∞ and ℓ_2 norms for $w = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1} \in R$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \quad \|w\| = \sqrt{\|w_0\|_\infty^2 + \ldots + \|w_{n-1}\|_\infty^2}.$$

Similarly, for $\mathbf{w} = (w_1, \ldots, w_k) \in R^k$, we define

$$\|\mathbf{w}\|_\infty = \max_i \|w_i\|_\infty, \quad \|\mathbf{w}\| = \sqrt{\|w_1\|^2 + \ldots + \|w_k\|^2}.$$

We will write S_η to denote all elements $w \in R$ such that $\|w\|_\infty \leq \eta$.

2.2 NTT domain representation

Our modulus q is chosen such that there exists a 512-th root of unity r modulo q. Concretely, we always work with $r = 1753$. This implies that the cyclotomic polynomial $X^{256}+1$ splits into linear factors $X - r^i$ modulo q with $i = 1, 3, 5, \ldots, 511$. By the Chinese remainder theorem our cyclotomic ring R_q is thus isomorphic to the product of the rings

[2] We draw the reader's attention to the fact that for even α, the range includes $\alpha/2$ but not $-\alpha/2$. This is a somewhat less standard choice, but defining things in this way makes some parts of the scheme (in particular, the bit-packing of the public key) more efficient.

$\mathbb{Z}_q[X]/(X - r^i) \cong \mathbb{Z}_q$. In this product of rings it is easy to multiply elements since the multiplication is pointwise there. The isomorphism

$$a \mapsto \big(a(r), a(r^3), \ldots, a(r^{511})\big) : R_q \to \prod_i \mathbb{Z}_q[X]/(X - r^i)$$

can be computed quickly with the help of the Fast Fourier Transform. Since $X^{256} + 1 = X^{256} - r^{256} = (X^{128} - r^{128})(X^{128} + r^{128})$ one can first compute the map

$$\mathbb{Z}_q[X]/(X^{256} + 1) \to \mathbb{Z}_q[X]/(X^{128} - r^{128}) \times \mathbb{Z}_q[X]/(X^{128} + r^{128})$$

and then continue separately with the two reduced polynomials of degree less than 128 noting that $X^{128} + r^{128} = X^{128} - r^{384}$. The Fast Fourier Transform is also called Number Theory Transform (NTT) in this case where the ground field is a finite field. Natural fast NTT implementations do not output vectors with coefficients in the order $a(r), a(r^3), \ldots, a(r^{511})$. Therefore we define the NTT domain representation $\hat{a} = \mathtt{NTT}(a) \in \mathbb{Z}_q^{256}$ of a polynomial $a \in R_q$ to have coefficients in the order as output by our reference NTT. Concretely,

$$\hat{a} = \mathtt{NTT}(a) = (a(r_0), a(-r_0), \ldots, a(r_{127}), a(-r_{127}))$$

where $r_i = r^{\mathrm{brv}(128+i)}$ with $\mathrm{brv}(k)$ the bitreversal of the 8 bit number k. With this notation, and because of the isomorphism property, we have $ab = \mathtt{NTT}^{-1}(\mathtt{NTT}(a)\mathtt{NTT}(b))$. For vectors $\mathbf{y}$ and matrices $\mathbf{A}$, the representations $\hat{\mathbf{y}} = \mathtt{NTT}(\mathbf{y})$ and $\hat{\mathbf{A}} = \mathtt{NTT}(\mathbf{A})$ mean that every polynomial y_i and $a_{i,j}$ comprising $\mathbf{y}$ and $\mathbf{A}$ is in NTT domain representation. We give further detail about our NTT implementations in Section 4.5.

2.3 Hashing

Our scheme uses several different algorithms that hash strings in $\{0,1\}^*$ onto domains of various forms. Below we give the high level descriptions of these functions and defer the details of how exactly they are used in the signature scheme to Section 4.2.

Hashing to a Ball. Let B_h denote the set of elements of R that have h coefficients that are either -1 or 1 and the rest are 0. We have $|B_h| = 2^h \cdot \binom{n}{h}$. For our signature scheme, we will need a cryptographic hash function that hashes onto B_{60} (which has more than 2^{256} elements). The algorithm we will use to create a random element in B_{60} is sometimes referred to as an "inside-out" version of the Fisher-Yates shuffle [Knu97], and its high-level description is in Fig. 2.[3]

SampleInBall
01 Initialize $\mathbf{c} = c_0\, c_1 \ldots c_{255} = 00\ldots0$
02 **for** $i := 196$ to 255
03 $\quad j \leftarrow \{0, 1, \ldots, i\}$
04 $\quad s \leftarrow \{0,1\}$
05 $\quad c_i := c_j$
06 $\quad c_j := (-1)^s$
07 **return c**

Figure 2: Create a random 256-element array with 60 ± 1's and 196 $0's$

[3]Normally, the algorithm should begin at $i = 0$, but since there are 196 0's, the first 195 iterations would just be setting components of $\mathbf{c}$ to 0.

Expanding the Matrix A. The function ExpandA maps a uniform seed $\rho \in \{0,1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times l}$ in NTT domain representation. The matrix $\mathbf{A}$ is only needed for multiplication. Hence, for the sake of faster implementations, the expansion function ExpandA does not output $\mathbf{A} \in R_q^{k \times l} = (\mathbb{Z}_q[X]/(X^{256}+1))^{k \times l}$. Instead it outputs $\hat{\mathbf{A}} \in \mathbb{Z}_q^{256}$, which is interpreted as the NTT domain representation of $\mathbf{A}$. As $\mathbf{A}$ needs to be sampled uniformly and the NTT is an isomorphism, ExpandA also needs to sample uniformly in this representation. To be compatible to Dilithium, an implementation whose NTT produces differently ordered vectors than our reference NTT needs to sample coefficients in a non-consecutive order.

Sampling the vectors y. The function ExpandMask, used for deterministically generating the randomness of the signature scheme, maps $K \parallel \mu \parallel \kappa$ to $y \in S_{\gamma_1 - 1}^l$.

Collision resistant hash. The function CRH used in our signature scheme is a collision resistant hash function mapping to $\{0,1\}^{384}$.

2.4 High/Low Order Bits and Hints

To reduce the size of the public key, we will need some simple algorithms that extract "higher-order" and "lower-order" bits of elements in $\mathbb{Z}_q$. The goal is that when given an arbitrary element $r \in \mathbb{Z}_q$ and another small element $z \in \mathbb{Z}_q$, we would like to be able to recover the higher order bits of $r + z$ without needing to store z. We therefore define algorithms that take r, z and produce a 1-bit hint h that allows one to compute the higher order bits of $r + z$ just using r and h. This hint is essentially the "carry" caused by z in the addition.

There are two different ways in which we will break up elements in $\mathbb{Z}_q$ into their "high-order" bits and "low-order" bits. The first algorithm, $\mathsf{Power2Round}_q$, is the straightforward bit-wise way to break up an element $r = r_1 \cdot 2^d + r_0$ where $r_0 = r \bmod^{\pm} 2^d$ and $r_1 = (r - r_0)/2^d$.

Notice that if we choose the representatives of r_1 to be non-negative integers between 0 and $\lfloor q/2^d \rfloor$, then the distance (modulo q) between any two $r_1 \cdot 2^d$ and $r_1' \cdot 2^d$ is usually $\geq 2^d$, except for the border case. In particular, the distance modulo q between $\lfloor q/2^d \rfloor \cdot 2^d$ and 0 could be very small. This is problematic in the case that we would like to produce a 1-bit hint, as adding a small number to r can actually cause the high-order bits of r to change by more than 1.

We avoid having the high-order bits change by more than 1 with a simple tweak. We select an α that is a divisor of $q - 1$ and write $r = r_1 \cdot \alpha + r_0$ in the same way as before. For the sake of simplicity, we assume that α is even (which is possible, as q is odd). The possible $r_1 \cdot \alpha$'s are now $\{0, \alpha, 2\alpha, \ldots, q-1\}$. Note that the distance between $q - 1$ and 0 is 1, and so we remove $q - 1$ from the set of possible $r_1 \cdot \alpha$'s, and simply round the corresponding r's to 0. Because $q - 1$ and 0 differ by 1, all this does is possibly increase the magnitude of the remainder r_0 by 1. This procedure is called $\mathsf{Decompose}_q$. Using this procedure as a sub-routine, we can define the $\mathsf{MakeHint}_q$ and $\mathsf{UseHint}_q$ routines that produce a hint and, respectively, use the hint to recover the high-order bits of the sum. For notational convenience, we also define $\mathsf{HighBits}_q$ and $\mathsf{LowBits}_q$ routines that simply extract r_1 and r_0, respectively, from the output of $\mathsf{Decompose}_q$.

The below Lemmas state the crucial properties of these supporting algorithms that are necessary for the correctness and security of our scheme. Their proofs can be found in Appendix A.

Lemma 1. *Suppose that q and α are positive integers satisfying $q > 2\alpha$, $q \equiv 1 \pmod{\alpha}$ and α even. Let $\mathbf{r}$ and $\mathbf{z}$ be vectors of elements in R_q where $\|\mathbf{z}\|_\infty \leq \alpha/2$, and let $\mathbf{h}, \mathbf{h}'$*

$\underline{\mathsf{Power2Round}_q(r, d)}$
08 $r := r \bmod^+ q$
09 $r_0 := r \bmod^\pm 2^d$
10 **return** $\big((r - r_0)/2^d, r_0\big)$

$\underline{\mathsf{MakeHint}_q(z, r, \alpha)}$
11 $r_1 := \mathsf{HighBits}_q(r, \alpha)$
12 $v_1 := \mathsf{HighBits}_q(r + z, \alpha)$
13 **return** $[\![r_1 \neq v_1]\!]$

$\underline{\mathsf{UseHint}_q(h, r, \alpha)}$
14 $m := (q - 1)/\alpha$
15 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
16 **if** $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod^+ m$
17 **if** $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod^+ m$
18 **return** r_1

$\underline{\mathsf{Decompose}_q(r, \alpha)}$
19 $r := r \bmod^+ q$
20 $r_0 := r \bmod^\pm \alpha$
21 **if** $r - r_0 = q - 1$
22 **then** $r_1 := 0$; $r_0 := r_0 - 1$
23 **else** $r_1 := (r - r_0)/\alpha$
24 **return** (r_1, r_0)

$\underline{\mathsf{HighBits}_q(r, \alpha)}$
25 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
26 **return** r_1

$\underline{\mathsf{LowBits}_q(r, \alpha)}$
27 $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
28 **return** r_0

Figure 3: Supporting algorithms for Dilithium.

be vectors of bits. Then the $\mathsf{HighBits}_q$, $\mathsf{MakeHint}_q$, *and* $\mathsf{UseHint}_q$ *algorithms satisfy the following properties:*

1. $\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}, \alpha), \mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{z}, \alpha)$.
2. *Let* $\mathbf{v}_1 = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha)$. *Then* $\|\mathbf{r} - \mathbf{v}_1 \cdot \alpha\|_\infty \leq \alpha + 1$. *Furthermore, if the number of* 1*'s in* $\mathbf{h}$ *is* ω, *then all except at most* ω *coefficients of* $\mathbf{r} - \mathbf{v}_1 \cdot \alpha$ *will have magnitude at most* $\alpha/2$ *after centered reduction modulo* q.
3. *For any* $\mathbf{h}, \mathbf{h}'$, *if* $\mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha) = \mathsf{UseHint}_q(\mathbf{h}', \mathbf{r}, \alpha)$, *then* $\mathbf{h} = \mathbf{h}'$.

Lemma 2. *If* $\|\mathbf{s}\|_\infty \leq \beta$ *and* $\|\mathsf{LowBits}_q(\mathbf{r}, \alpha)\|_\infty < \alpha/2 - \beta$, *then*

$$\mathsf{HighBits}_q(\mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{s}, \alpha).$$

3 Signature

The Key Generation, Signing, and Verification algorithms for our signature scheme are presented in Fig. 4. We present the deterministic version of the scheme in which the randomness used in the signing procedure is generated (using SHAKE-256) as a deterministic function of the message and a small secret key. Since our signing procedure may need to be repeated several times until a signature is produced, we also append a counter in order to make the SHAKE-256 output differ with each signing attempt of the same message. Also due to the fact that each message may require several iterations to sign, we compute an initial digest of the message using a collision-resistant hash function, and use this digest in place of the message throughout the signing procedure.

As discussed in Section 1.2, the main design improvement of Dilithium over the scheme in Fig. 1 is that the public key size is reduced by a factor of around 2.5 at the expense of an additional hundred bytes in the signature. To accomplish the size reduction, the key generation algorithm outputs $\mathbf{t}_1 := \mathsf{Power2Round}_q(\mathbf{t}, d)$ as the public key instead of $\mathbf{t}$ as in Fig. 1. This means that instead of $\lceil \log q \rceil$ bits per coefficient, the public key requires $\lceil \log q \rceil - d$ bits. In our instantiation, $q \approx 2^{23}$ and $d = 14$, which means that instead of 23 bits in each public key coefficient, there are instead 9.

The main problem with not having the entire $\mathbf{t}$ in the public key is that the verification algorithm is no longer able to exactly compute $\mathbf{w}_1'$ in Line 13 in Fig. 1. In order to do this, the verification algorithm will need the high order bits of $\mathbf{Az} - c\mathbf{t}$, but it can only compute $\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0$. But since the product $c\mathbf{t}_0$ consists of only small numbers, and we only care about the high order bits, we really only need to know the carries that each coefficient of $c\mathbf{t}_0$ causes. These are the carries that the signer sends as a hint to the verifier. Heuristically, based on our parameter choices, there should not be more than ω positions in which a carry is caused. The signer therefore simply sends the positions in which these carries occur (this is the extra bytes in the signature), which allows the verifier to compute the high order bits of $\mathbf{Az} - c\mathbf{t}$.

3.1 Implementation Notes and Efficiency Trade-offs

To keep the size of the public (and secret) key small, both the Sign and Verify procedures begin with extracting the matrix $\mathbf{A}$ (or more accurately, its NTT domain representation $\hat{A}$) from the seed ρ. If storage space is not a factor, then $\hat{\mathbf{A}}$ can be pre-computed and be part of the secret/public key. The signer can additionally pre-compute the NTT domain representations of $\mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0$ to slightly speed up the signing operation. At the other extreme, if the signer wants to store as small a secret key as possible, he only needs to store ρ and K, and the random seed used to create $\mathbf{s}_1, \mathbf{s}_2$ in the key generation algorithm. All the other parts of the secret key can be recreated from these. Furthermore, one can also keep the memory for intermediate computations low by only keeping the parts of the NTT domain representation that one is currently working with.

Another possible change is to remove the strict deterministic nature of the digital signature. One may want to consider this option due to the recent side-channel attacks that exploit determinism [SBB+17, PSS+17]. An easy way in which to give an option of using randomized signatures is to allow the appending of some system randomness to the input of ExpandMask when generating $\mathbf{y}$. As we mentioned earlier, the security proof for Dilithium is "tight" according to [KLS17] for deterministic signatures and the bound gets gradually looser the more different signatures are seen per message. We therefore still recommend using deterministic signatures except in environments that may be vulnerable to the aforementioned side-channel attacks.

3.2 Correctness

In this section, we prove the correctness of the signature scheme.

If $\|c\mathbf{t}_0\|_\infty < \gamma_2$, then by Lemma 1 we know that

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\,.$$

Since $\mathbf{w} = \mathbf{Ay}$ and $\mathbf{t} = \mathbf{As}_1 + \mathbf{s}_2$, we have that

$$\mathbf{w} - c\mathbf{s}_2 = \mathbf{Ay} - c\mathbf{s}_2 = \mathbf{A}(\mathbf{z} - c\mathbf{s}_1) - c\mathbf{s}_2 = \mathbf{Az} - c\mathbf{t}, \tag{2}$$

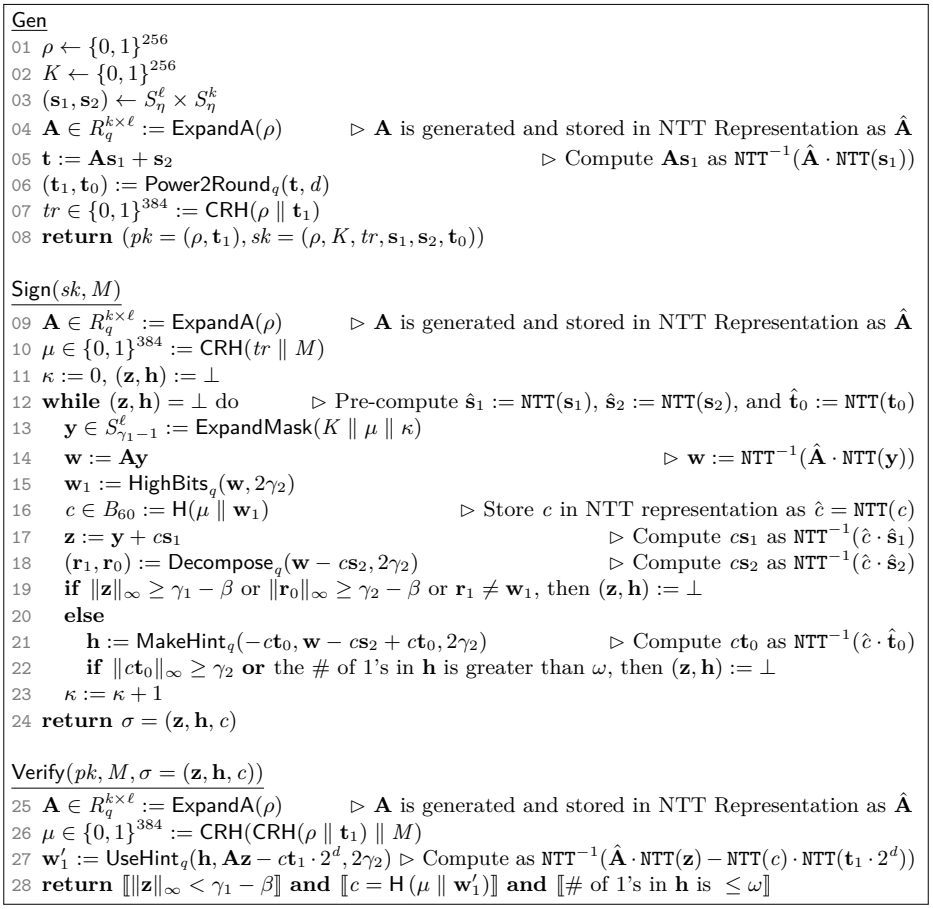
and $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0 = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$. Therefore the verifier computes

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\,.$$

Furthermore, because the signer also checks in Line 19 that $\mathbf{r}_1 = \mathbf{w}_1$, this is equivalent to

$$\mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2). \tag{3}$$

Therefore, the $\mathbf{w}_1$ computed by the verifier is the same as that of the signer, and the verification procedure will always accept.



Gen

01 $\rho \leftarrow \{0,1\}^{256}$
02 $K \leftarrow \{0,1\}^{256}$
03 $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
04 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
05 $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ ▷ Compute $\mathbf{A}\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{s}_1))$
06 $(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
07 $tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho \parallel \mathbf{t}_1)$
08 **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

Sign(sk, M)

09 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
10 $\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \parallel M)$
11 $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
12 **while** $(\mathbf{z}, \mathbf{h}) = \perp$ do ▷ Pre-compute $\hat{\mathbf{s}}_1 := \mathtt{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 := \mathtt{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathtt{NTT}(\mathbf{t}_0)$
13 $\quad \mathbf{y} \in S_{\gamma_1 - 1}^\ell := \mathsf{ExpandMask}(K \parallel \mu \parallel \kappa)$
14 $\quad \mathbf{w} := \mathbf{A}\mathbf{y}$ ▷ $\mathbf{w} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{y}))$
15 $\quad \mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
16 $\quad c \in B_{60} := \mathsf{H}(\mu \parallel \mathbf{w}_1)$ ▷ Store c in NTT representation as $\hat{c} = \mathtt{NTT}(c)$
17 $\quad \mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ ▷ Compute $c\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_1)$
18 $\quad (\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ ▷ Compute $c\mathbf{s}_2$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_2)$
19 $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$, then $(\mathbf{z}, \mathbf{h}) := \perp$
20 $\quad$ **else**
21 $\quad\quad \mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ ▷ Compute $c\mathbf{t}_0$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_0)$
22 $\quad\quad$ **if** $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$
23 $\quad \kappa := \kappa + 1$
24 **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Verify$(pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c))$

25 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
26 $\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{CRH}(\rho \parallel \mathbf{t}_1) \parallel M)$
27 $\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$ ▷ Compute as $\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{z}) - \mathtt{NTT}(c) \cdot \mathtt{NTT}(\mathbf{t}_1 \cdot 2^d))$
28 **return** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(\mu \parallel \mathbf{w}_1')]\!]$ **and** $[\![\#$ of 1's in $\mathbf{h}$ is $\leq \omega]\!]$

Figure 4: The signature scheme Dilithium.

3.3 Number of Iterations

We now want to compute the probability that Step 19 will set $(\mathbf{z}, \mathbf{h})$ to $\perp$. The probability that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ can be computed by considering each coefficient separately. For each coefficient σ of $c\mathbf{s}_1$, the corresponding coefficient of $\mathbf{z}$ will be between $-\gamma_1 + \beta + 1$ and $\gamma_1 - \beta - 1$ (inclusively) whenever the corresponding coefficient of $\mathbf{y}_i$ is between $-\gamma_1 + \beta + 1 - \sigma$ and $\gamma_1 - \beta - 1 - \sigma$. The size of this range is $2(\gamma_1 - \beta) - 1$, and the coefficients of $\mathbf{y}$ have $2\gamma_1 - 1$ possibilities. Thus the probability that every coefficient of $\mathbf{y}$ is in the good range is

$$\left(\frac{2(\gamma_1 - \beta) - 1}{2\gamma_1 - 1}\right)^{256 \cdot \ell} = \left(1 - \frac{\beta}{\gamma_1 - 1/2}\right)^{\ell n} \approx \mathrm{e}^{-256 \cdot \beta \ell / \gamma_1}, \tag{4}$$

where we used the fact that our values of γ_1 are large compared to $1/2$.

We now move to computing the probability that we have

$$\|\mathbf{r}_0\|_\infty = \|\mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta\,.$$

If we (heuristically) assume that the low order bits are uniformly distributed modulo $2\gamma_2$,

Table 1: Parameters for Dilithium. The formulas for the sizes of the public key and signature are given in Section 4.3. The explanations for the NIST security levels is in Section 5.3.

	I weak	II medium	III recommended	IV very high
q	8380417	8380417	8380417	8380417
d	14	14	14	14
weight of c	60	60	60	60
$\gamma_1 = (q-1)/16$	523776	523776	523776	523776
$\gamma_2 = \gamma_1/2$	261888	261888	261888	261888
(k, ℓ)	$(3, 2)$	$(4, 3)$	$(5, 4)$	$(6, 5)$
η	7	6	5	3
β	375	325	275	175
ω	64	80	96	120
pk size (bytes)	896	1184	1472	1760
sig size (bytes)	1487	2044	2701	3366
Exp. reps (from Eq. (5))	4.3	5.9	6.6	4.3
BKZ block-size b to break SIS	235	355	475	605
Best Known Classical bit-cost	68	103	138	176
Best Known Quantum bit-cost	62	94	125	160
BKZ block-size b to break LWE	200	340	485	595
Best Known Classical bit-cost	58	100	141	174
Best Known Quantum bit-cost	53	91	128	158
NIST Security Level	-	1	2	3
Gen cycles (Haswell)	169,972	269,844	382,756	512,116
Sign cycles (Haswell)	765,442	1,285,476	1,817,902	1,677,782
Verify cycles (Haswell)	196,048	296,920	395,936	548,558
Gen cycles (AVX2, Haswell)	104,128	156,432	225,432	292,404
Sign cycles (AVX2, Haswell)	338,922	493,332	673,144	711,018
Verify cycles (AVX2, Haswell)	105,584	150,228	207,164	288,398

then there is a

$$\left(\frac{2(\gamma_2 - \beta) - 1}{2\gamma_2}\right)^{256 \cdot k} \approx \mathrm{e}^{-256 \cdot \beta k/\gamma_2}$$

probability that all the coefficients are in the good range (using the fact that our values of β are large compared to $1/2$).

As we already mentioned, if $\|c\mathbf{s}_2\|_\infty \leq \beta$, then $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$ implies that $\mathbf{r}_1 = \mathbf{w}_1$. Thus the last check should succeed with overwhelming probability when the previous check passed. Therefore, the probability that Step 19 passes is

$$\approx \mathrm{e}^{-256 \cdot \beta(\ell/\gamma_1 + k/\gamma_2)}\,. \tag{5}$$

It is more difficult to formally compute the probability that Step 22 results in a restart. The parameters were set such that heuristically $(\mathbf{z}, \mathbf{h}) = \perp$ with probability less than 1%. Therefore the vast majority of the loop repetitions will be caused by Step 19.

4 Implementation Details

4.1 Bit-packing

We now describe how we encode vectors as byte strings. This is needed for absorbing them into SHAKE and defining the data layout of the keys and signature. To reduce the computation time spent on SHAKE and the sizes of keys and signatures, we use bit-packing.

We start with the vector $\mathbf{w}_1$ that, together with μ, is hashed to a ball. It consists of k polynomials $w_{1,1}, \ldots, w_{1,k}$ in R_q with coefficients that are roundings of elements in $\mathbb{Z}_q$ with respect to $\alpha = 2\gamma_2$. It follows that the coefficients lie in $\{0, \ldots, 15\}$ and can be represented by 4 bits each. This allows $\mathbf{w}_1$ to be packed in a string of $k \cdot 256 \cdot 4/8 = k \cdot 128$ bytes. Each byte encodes two consecutive coefficients of a polynomial $w_{1,i}$ in its low 4 bits and high 4 bits, respectively. See Figure 5 for an explanation of the exact bit packing.

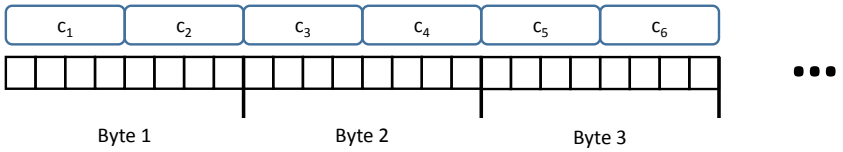


Figure 5: Bit-packing $\mathbf{w}_1$. The k polynomials comprising $\mathbf{w}_1$ are $w_{1,1}, \ldots, w_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $w_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $w_{1,2}$, etc.

Next we turn to the vector $\mathbf{t}_1$, which is the power-of-two rounding of $\mathbf{t}$. Note that $q - 1 = 2^{23} - 2^{13} = (2^9 - 1)2^{14} + 2^{13}$ which shows that the coefficients of the k polynomials of $\mathbf{t}_1$ lie in $\{0, \ldots, 2^9 - 1\}$ and can be represented by 9 bits each. These 9 bits per coefficient, in little-endian byte-order, are bit-packed. In total $\mathbf{t}_1$ needs $k \cdot 256 \cdot 9/8 = 288k$ bytes. See Figure 6 for an explanation of the exact bit packing.

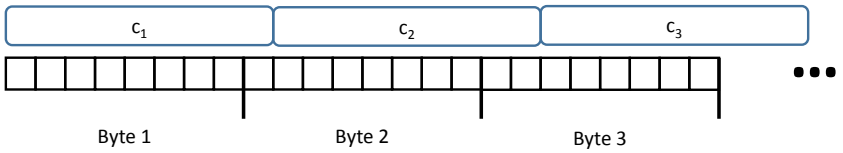


Figure 6: Bit-packing $\mathbf{t}_1$. The k polynomials comprising $\mathbf{t}_1$ are $t_{1,1}, \ldots, t_{1,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $t_{1,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $t_{1,2}$, etc.

The coefficients of the polynomials of $\mathbf{t}_0$ can be written in the form $q + 2^{13} - v$ with $v \in \{0, \ldots, 2^{14} - 1\}$. These v in little endian byte-order are bit-packed. This results in $256 \cdot 14/8$ bytes per polynomial and $k \cdot 256 \cdot 14/8 = 448k$ bytes for $\mathbf{t}_0$. See Figure 7 for an explanation of the exact bit packing.

The polynomials in $\mathbf{s}_1$ and $\mathbf{s}_2$ have coefficients with infinity norm at most η. So every coefficient of these polynomials is equivalent modulo q to $\eta - c$ with some $c \in \{0, \ldots, 2\eta\}$. In the bit packing the values for c are stored so that each polynomial needs $256\lceil \log 2\eta + 1 \rceil / 8$ bytes. This amounts to $256 \cdot 4/8 = 128$ bytes for the weak, medium and recommended security levels, and $256 \cdot 3/8 = 96$ bytes for the very high security level. The bit-packing is

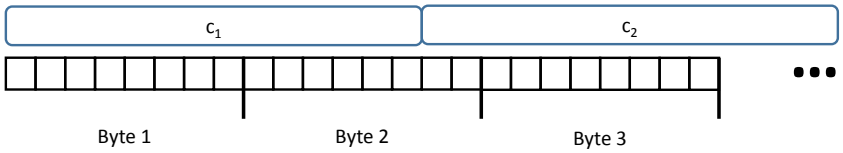


Figure 7: Bit-packing $\mathbf{t}_0$. The k polynomials comprising $\mathbf{t}_0$ are $t_{0,1}, \ldots, t_{0,k}$ and we let $c_1, \ldots, c_{256}$ be the coefficients of $t_{0,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $t_{0,2}$, etc.

done similarly to the case of $\mathbf{w}_1$, $\mathbf{t}_1$ and $\mathbf{t}_0$. See Figure 8 for an explanation of the exact bit packing.

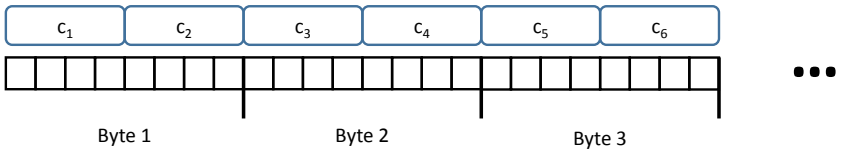


Figure 8: Bit-packing $\mathbf{s}_i$. The ℓ polynomials comprising $\mathbf{s}_1$ are $s_{1,1}, \ldots, s_{1,\ell}$ and we let $\eta - c_1, \ldots, \eta - c_{256}$ be the coefficients of $s_{1,1}$ (with the lower powers first), $\eta - c_{257}, \ldots, \eta - c_{512}$ be the coefficients of $s_{1,2}$, etc. where $c_i \in \{0, \ldots, 2\eta\}$. The k polynomials comprising $\mathbf{s}_2$ are $s_{2,1}, \ldots, s_{2,\ell}$ and we let $\eta - c_1, \ldots, \eta - c_{256}$ be the coefficients of $s_{2,1}$ (with the lower powers first), $c_{257}, \ldots, c_{512}$ be the coefficients of $s_{2,2}$, etc. $c_i \in \{0, \ldots, 2\eta\}$. The above picture is for parameter sets where c_i requires four bits per coefficient (i.e. when $4 \leq \eta \leq 15$). When $\eta < 4$, one would only use three bits per coefficient and pack in the obvious manner.

Finally, $\mathbf{z}$ contains polynomials whose coefficients are equivalent modulo q to $\gamma_1 - 1 - c$ with $c \in \{0, \ldots, 2\gamma_1 - 2\}$ and these values c are bit packed. Since $\lceil \log 2\gamma_1 - 1 \rceil = 20$, bit-packing $\mathbf{z}$ requires $l \cdot 256 \cdot 20/8 = 640l$ bytes and blocks of 2 coefficients are stored in 5 consecutive bytes. See Figure 9 for an explanation of the exact bit packing.

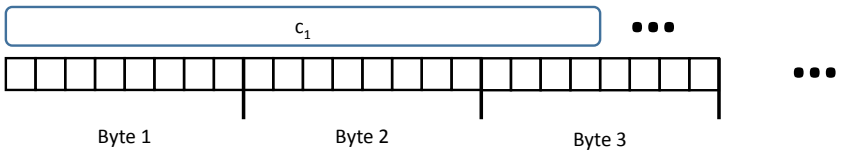


Figure 9: Bit-packing $\mathbf{z}$. The ℓ polynomials comprising $\mathbf{z}$ are $z_1, \ldots, z_\ell$ and we let $\gamma_1 - 1 - c_1, \ldots, \gamma_1 - 1 - c_{256}$ be the coefficients of z_1 (with the lower powers first), $\gamma_1 - 1 - c_{257}, \ldots, \gamma_1 - 1 - c_{512}$ be the coefficients of z_2, etc.

4.2 Hashing

Hashing to a Ball. We now precisely specify the operation of the function $\mathrm{H} : \mu \parallel \mathbf{w}_1 \mapsto c \in B_{60}$ described in Fig. 2 as it is used in our signature scheme. H absorbs the 48 bytes of μ immediately followed by the $128k$ bytes for the bit-packed representation of $\mathbf{w}_1$ into

SHAKE-256. Throughout its operations the function squeezes SHAKE-256 in order to obtain a stream of random bytes of variable length. The first 60 bits in the first 8 bytes of this random stream are interpreted as 60 random sign bits $s_i \in \{0,1\}$, $i = 0, \ldots, 59$. The remaining 4 bits are discarded. Then H uses Algorithm 2 to compute c. In each iteration of the for loop it uses rejection sampling on elements from $\{0, \ldots, 255\}$ until it gets a $j \in \{0, \ldots, i\}$. An element in $\{0, \ldots, 255\}$ is obtained by interpreting the next byte of the random stream from SHAKE-256 as a number in this set. For the sign s the corresponding s_{i-196} is used.

Expanding the Matrix A. The function ExpandA maps a uniform seed $\rho \in \{0,1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times l}$ in NTT domain representation. It computes each coefficient $\hat{a}_{i,j} \in R_q$ of $\hat{\mathbf{A}}$ separately. For the coefficient $\hat{a}_{i,j}$ it absorbs the 32 bytes of ρ immediately followed by one byte representing $0 \leq 2^4 \cdot j + i < 255$ into SHAKE-128. Next it uses consecutive blocks of 3 bytes of the variable-length output string in order to obtain a sequence of integers between 0 and $2^{23} - 1$. This is done by setting the highest bit of the third byte in each block to zero and interpreting the blocks in little endian byte order. So for example the three bytes b_0, b_1 and b_2 from SHAKE-128 are used to get the integer $0 \leq b_2' \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \leq 2^{23} - 1$ where b_2' is the logical AND of b_2 and $2^{128} - 1$. Finally, ExpandA performs rejection sampling on these 23-bit integers to sample the 256 coefficients $a_{i,j}(r_0), a_{i,j}(-r_0), \ldots, a_{i,j}(r_{127}) a_{i,j}(-r_{127})$ of $\hat{a}_{i,j}$ uniformly from the set $\{0, \ldots, q-1\}$ in the order of our NTT domain representation.

Sampling the vectors y. The function ExpandMask maps $K \parallel \mu \parallel \kappa$ to $y \in S^l_{\gamma_1 - 1}$, where $\kappa \geq 0$, and works as follows. It computes each of the l coefficients of y, which are polynomials in $S_{\gamma_1 - 1}$, independently. For the i-th polynomial, $0 \leq i < l$, it absorbs the 48 bytes of μ concatenated with the 32 bytes of K and two bytes representing $\kappa + i$ in little endian byte order into SHAKE-256. Then each block of 5 consecutive output bytes is used to get two 20 bit integers between 0 and $2^{20} - 1$. For this the first two bytes of each output block together with a third byte having as lower 4 bits the lower 4 bits of the third output byte and 4 high zero bits is interpreted in little endian order. Then the high 4 bits of the third output byte followed by the 16 bits of the fourth and and fifth byte are interpreted as the second 20 bit integer. As an example assume we have received the five bytes $b_0, \ldots, b_4$ from SHAKE-128. Then ExpandMask computes the two integers $0 \leq b_2' \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \leq 2^{20} - 1$ and $0 \leq b_4 \cdot 2^{12} + b_3 \cdot 2^4 + b_2'' \leq 2^{20} - 1$ where b_2' is the AND of b_2 and 15 and $b_2'' = \lfloor b_2 / 16 \rfloor$. On the resulting sequence of 20 bit integers rejection sampling is performed to get 256 values $v_j \in \{0, \ldots, 2\gamma_1 - 2\}$. From these the polynomial coefficients are computed in increasing order as $q + \gamma_1 - 1 - v_j$.

Collision resistant hash. The function CRH in Figure 4 is a collision resistant hash function. For this purpose 384 bits of the output of SHAKE-256 are used. CRH is called with two different sets of inputs. First it is called with $\rho \parallel \mathbf{t}_1$. The function then absorbs the 32 bytes of ρ followed by the $k \cdot 256 \cdot 9/8$ bytes for the bit-packed representation of $\mathbf{t}_1$ into SHAKE-256 and takes the first 48 bytes of the first output block of SHAKE-256 as the output hash. The second input is $\mu \parallel M$. Here the concatenation of the hash μ and the message string are absorbed into SHAKE-256 and the first 48 output bytes are used as the resulting hash.

4.3 Data layout of keys and signature

Public key. The public key, containing ρ and $\mathbf{t}_1$, is stored as the concatenation of the bit-packed representations of ρ and $\mathbf{t}_1$ in this order. Therefore, it has a size of $32 + 288k$ bytes.

Secret key. The secret key contains ρ, K, tr, $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{t}_0$ and is also stored as the concatenation of the bit-packed representation of these quantities in the given order. Consequently, a secret key requires $64 + 48 + 32((k + l) \cdot \lceil \log 2\eta + 1 \rceil + 14k)$ bytes. For the weak, medium and high security level this is equal to $112 + 576k + 128l$ bytes. With the very high security parameters one needs $112 + 544k + 96l = 3856$ bytes.

Signature. The signature byte string is the concatenation of a bit packed representation of $\mathbf{z}$ and encodings of h and c in this order. We describe the encoding of h, which needs $\omega + k$ bytes. Together all the polynomials in the vector $\mathbf{h}$ have at most ω non-zero coefficients. It is sufficient to store the locations of these non-zero coefficients. Each of the first ω bytes of the byte string representing $\mathbf{h}$ is the index i of the next non-zero coefficient in its polynomial, i.e. $0 \leq i \leq 255$, or zero if there are no more non-zero coefficients. The bytes numbers ω up to $\omega + k - 1$ record the k positions j of the polynomial boundaries in the string of ω coefficient indices, where $0 \leq j \leq \omega$. In the encoding of the challenge c, the first 256 bits are 0 or 1 when the corresponding coefficient of c is zero or non-zero, respectively. The next 60 bits are 0 or 1 if the corresponding non-zero coefficient is 1 or -1, respectively. Note that there are precisely 60 non-zero coefficients. The 4 bits up to the next byte boundary are zero.

Therefore, a signature requires $640l + \omega + k + 40$ bytes.

4.4 Constant time implementation

Our reference implementation does not branch depending on secret data and does not access memory locations that depend on secret data. For the modular reductions that are needed for the arithmetic in R_q we never use the '%' operator of the C programming language. Instead we use Montgomery reductions without the correction steps and special reduction routines that are specific to our modulus q. For computing the rounding functions described in Section 2.4, we have implemented branching-free algorithms. On the other hand, when it is safe to reveal information, we have not tried to make the code constant-time. This includes the computation of the challenges and the rejection conditions in the signing algorithm. When performing rejection sampling, our code reveals which of the conditions was the reason for the rejection, and in case of the norm checks, which coefficient violated the bound. This is safe since the rejection probabilities for each coefficient are independent of secret data. The challenges reveal information about $\mathsf{CRH}(\mu \parallel \mathbf{w}_1)$ also in the case of rejected $\mathbf{y}$, but this does not reveal any information about the secret key when CRH is modeled as a random oracle and $\mathbf{w}_1$ has high min-entropy.

4.5 Reference implementation

Our reference NTT is a natural iterative implementation for 32 bit unsigned integers that uses Cooley-Tukey butterflies in the forward transform and Gentleman-Sande butterflies in the inverse transform. For modular reductions after multiplying with a precomputed root of unity we use the Montgomery algorithm as was already done before in e.g. [ADPS16]. In order that the reduced values are correct representatives, the precomputed roots contain the Montgomery factor $2^{32} \bmod q$. We also use Montgomery reductions after the pointwise product of the polynomials in the NTT domain representations. Since we cannot get the Montgomery factor in at this point, these products are in fact Hensel remainders $r' \equiv r2^{32} \pmod q$. We then make use of the fact that the NTT transform is linear and multiply by an additional Montgomery factor after the inverse NTT when we divide out the factor 256.

The implementations of the functions $\mathsf{ExpandA}$ and $\mathsf{ExpandMask}$ initially squeeze a number of output blocks of SHAKE-256 and SHAKE-128 that gives enough randomness with high probability. In the case of $\mathsf{ExpandA}$, which samples uniform polynomials and hence needs at least $3 \cdot 256 = 768$ random bytes per polynomial, 5 blocks from SHAKE-128

of 168 bytes each are needed at least for one polynomial. They suffice with probability greater than $1-2^{-132}$. ExpandMask initially retrieves 5 blocks from SHAKE-256 that have 136 bytes. This is the minimum number of blocks and suffices with probability greater than $1-2^{-81}$.

As mentioned in the introduction our reference implementation is protected against timing attacks. For this reason the centralized remainders in the rounding functions given in Figure 3 are not computed with branchings. Instead we use the following well-known trick to compute the centralized remainder $r' = r \bmod^{\pm} \alpha$ where $0 \leq r < q$. Subtracting $\alpha/2+1$ from r yields a negative result if and only if $r \leq \alpha/2$. Therefore, shifting this result arithmetically to the right by 31 bits gives -1, i.e. the integer with all bits equal to 1, if $r \leq \alpha/2$ and 0 otherwise. Then the logical AND of the shifted value and α is added to r and $\alpha/2-1$ subtracted. This results in $r-\alpha$ if $r > \alpha$ and r if $r \leq \alpha/2$, i.e. the centralized remainder.

We make heavy use of lazy reduction in our implementation. In the NTT we do not reduce the results of additions and subtractions at all. For rounding and norm checking it is important to map to standard representatives. This freezing of the coefficients is achieved in constant-time by conditionally subtracting q with another instance of the arithmetic right shift trick.

4.6 AVX2 optimized implementation

We have written an optimized implementation of Dilithium for CPUs that support the AVX2 instruction set. Since the two most time-consuming operations are polynomial multiplication and the expansion of the matrix and vectors, the optimized implementation speeds up these two operations.

For polynomial multiplication, we use a vectorized version of the NTT. This NTT achieves a full multiplication of two polynomials including three NTTs and the pointwise multiplication in less than 5000 Haswell cycles and is about a factor of 4.5 faster than the reference C code compiled using gcc with full machine-specific optimizations turned on. Contrary to some other implementations (e.g. [ADPS16]), we do not use floating point instructions. When using floating point instructions, modular reductions are easily done by multiplying with a floating point inverse of q and rounding to get the quotient from which the remainder can be computed with another multiplication and a subtraction. Instead of this approach we use integer instructions only and the same Montgomery reduction methodology as in the reference C code. When compared to the floating point NTT from [ADPS16] applied to the Dilithium prime $q = 2^{23} - 2^{13} + 1$, our integer NTT is about two times faster.

At any time our AVX2 optimized NTT has 32 unsigned integer coefficients, of 32 bits each, loaded into 8 AVX2 vector registers. Each of these vector registers then contains 4 extended 64 bit coefficients. So after three levels of NTT the reduced polynomials fit completely into these 8 registers and we can transform them to linear factors without further loads and stores. In the second to last and last level the polynomials have degree less than 4. This means that every polynomial fits into one register but only half of the coefficients need to be multiplied by roots. For this reason we shuffle the vectors in order to group together coefficients that need to be multiplied. The instruction that we use for this task are `perm2i128` in the second last level and a combination of `vpshufd` and `vpblendd` in the last level. The multiplications with the constant roots of unity are performed using the `vpmuludq` instruction. This instruction computes a full 64 bit product of two 32 bit integers. It has a latency of 5 cycles on both Haswell and Skylake. In each level of the NTT half of the coefficients need to be multiplied. Therefore we can do four vector multiplications and Montgomery reductions in parallel. This hides some of the latency of the multiplication instructions.

For faster matrix and vector expansion, we use a vectorized SHAKE implementation

that operates on 4 parallel sponges and hence can absorb and squeeze blocks in and out of these 4 sponges at the same time. For sampling this means that up to four coefficients can be sampled simultaneously.

4.7 Computational Efficiency

We have performed timing experiments with our reference implementation on a Haswell CPU. The results are presented in Table 1. They include the number of CPU cycles needed by the three operations key generation, signing and signature verification. These numbers are the medians of 10000 operations each. Signing was performed with a message size of 32 bytes. The computer we have used is equipped with an Intel Core i7-4770K CPU running at the constant clock frequency of 3500 Mhz. Hyperthreading and Turbo Boost are switched off. The system runs Debian stable with Linux Kernel version 3.16.0 and the code was compiled with gcc 6.3.0.

5 Security Reductions

The standard security notion for digital signatures is UF-CMA security, which is security under chosen message attacks. In this security model, the adversary gets the public key and has access to a signing oracle to sign messages of his choice. The adversary's goal is to come up with a valid signature of a new message. A slightly stronger security requirement that is sometimes useful is SUF-CMA (Strong Unforgeability under Chosen Message Attacks), which also allows the adversary to win by producing a different signature of a message that he has already seen.

It can be shown that in the (classical) random oracle model, Dilithium is SUF-CMA secure based on the hardness of the standard MLWE and MSIS lattice problems. The reduction, however, is not tight. Furthermore, since we also care about quantum attackers, we need to consider the security of the scheme when the adversary can query the hash function on a superposition of inputs (i.e. security in the quantum random oracle model – QROM). Since the classical security proof uses the "forking lemma" (which is essentially rewinding), the reduction does not transfer over to the quantum setting.

There are no counter-examples of schemes whose security is actually affected by the non-tightness of the reduction. For example, schemes like Schnorr signatures [Sch89], GQ signatures [GQ88], etc. all set their parameters ignoring the non-tightness of the reduction. Furthermore, the only known uses of the additional power of quantum algorithms against schemes whose security is based on quantum-resistant problems under a classical reduction involve "Grover-type" algorithms that improve exhaustive search (although it has been shown that there cannot be a "black-box" proof that the Fiat-Shamir transform is secure in the QROM [ARU14]).

The reason that there haven't been any attacks taking advantage of the non-tightness of the reduction is because there is an intermediate problem which is *tightly* equivalent, even under quantum reductions, to the UF-CMA security of the signature scheme. This problem is essentially a "convolution" of the underlying mathematical problem (such as MSIS or discrete log) with a cryptographic hash function H. It would appear that as long as there is no relationship between the structure of the math problem and H, solving this intermediate problem is not easier than solving the mathematical problem.[4]

Below, we will introduce the hardness assumptions upon whose hardness the SUF-CMA security of our scheme is based. The first two assumptions, MLWE and MSIS, are standard lattice problems which are a generalization of LWE, Ring-LWE, SIS, and Ring-SIS. The third problem, SelfTargetMSIS is the aforementioned problem that's based on the combined

[4] In the ROM, there is indeed a (non-tight) reduction using the forking lemma that states that solving this problem is as hard as solving the underlying mathematical problem.

hardness of MSIS and the hash function H. In the classical ROM, there is a (non-tight) reduction from MSIS to SelfTargetMSIS.

5.1 Assumptions

The MLWE Problem. For integers m, k, and a probability distribution $D : R_q \to [0, 1]$, we say that the advantage of algorithm A in solving the decisional $\mathsf{MLWE}_{m,k,D}$ problem over the ring R_q is

$$\begin{aligned}\mathrm{Adv}^{\mathsf{MLWE}}_{m,k,D} := \big|&\Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{t} \leftarrow R_q^m; b \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{t})] \\ &- \Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{s}_1 \leftarrow D^k; \mathbf{s}_2 \leftarrow D^m; b \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)]\big| .\end{aligned}$$

The MSIS Problem. To an algorithm A we associate the advantage function $\mathrm{Adv}^{\mathsf{MSIS}}_{m,k,\gamma}$ to solve the (Hermite Normal Form) $\mathsf{MSIS}_{m,k,\gamma}$ problem over the ring R_q as

$$\mathrm{Adv}^{\mathsf{MSIS}}_{m,k,\gamma}(\mathsf{A}) := \Pr\left[0 < \|\mathbf{y}\|_\infty \le \gamma \wedge [\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y} = \mathbf{0} \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{y} \leftarrow \mathsf{A}(\mathbf{A})\right] .$$

The SelfTargetMSIS Problem. Suppose that $\mathsf{H} : \{0, 1\}^* \to B_{60}$ is a cryptographic hash function. To an algorithm A we associate the advantage function

$$\mathrm{Adv}^{\mathsf{SelfTargetMSIS}}_{\mathsf{H},m,k,\gamma}(\mathsf{A}) :=$$

$$\Pr\left[\begin{matrix}0 \le \|\mathbf{y}\|_\infty \le \gamma \\ \wedge\, \mathsf{H}([\ \mathbf{I} \mid \mathbf{A}\ ] \cdot \mathbf{y} \parallel M) = c\end{matrix} \;\middle|\; \mathbf{A} \leftarrow R_q^{m\times k}; \left(\mathbf{y} := \begin{bmatrix}\mathbf{r}\\ c\end{bmatrix}, M\right) \leftarrow \mathsf{A}^{|\mathsf{H}(\cdot)\rangle}(\mathbf{A})\right] .$$

5.2 Signature Scheme Security

The concrete security of Dilithium was analyzed in [KLS17], where it was shown that if H is a quantum random oracle (i.e., a quantum-accessible perfect hash function), the advantage of an adversary A breaking the SUF-CMA security of the signature scheme is

$$\mathrm{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Dilithium}}(\mathsf{A}) \le \mathrm{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathsf{B}) + \mathrm{Adv}^{\mathsf{SelfTargetMSIS}}_{\mathsf{H},k,\ell+1,\zeta}(\mathsf{C}) + \mathrm{Adv}^{\mathsf{MSIS}}_{k,\ell,\zeta'}(\mathsf{D}) + 2^{-254}\ ,^{5} \tag{6}$$

for D a uniform distribution over S_η, and

$$\zeta = \max\{\gamma_1 - \beta, 2\gamma_2 + 1 + 2^{d-1} \cdot 60\} \le 4\gamma_2, \tag{7}$$

$$\zeta' = \max\{2(\gamma_1 - \beta), 4\gamma_2 + 2\} \le 4\gamma_2 + 2. \tag{8}$$

Furthermore, if the running times and success probabilities (i.e. advantages) of $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}$ are $t_\mathsf{A}, t_\mathsf{B}, t_\mathsf{C}, t_\mathsf{D}, \epsilon_\mathsf{A}, \epsilon_\mathsf{B}, \epsilon_\mathsf{C}, \epsilon_\mathsf{D}$, then the lower bound on $t_\mathsf{A}/\epsilon_\mathsf{A}$ is within a small multiplicative factor of $\min t_i/\epsilon_i$ for $i \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}$.

Intuitively, the MLWE assumption is needed to protect against key-recovery, the SelfTargetMSIS is the assumption upon which new message forgery is based, and the MSIS assumption is needed for strong unforgeability. We will now sketch some parts of the security proof that are relevant to the concrete parameter setting.

[5] To simplify the concrete security bound, we assume that ExpandA produces a uniform matrix $\mathbf{A} \in R_q^{k\times\ell}$, $\mathsf{ExpandMask}(K, \cdot)$ is a perfect pseudo-random function, and CRH is a perfect collision-resistant hash function.

5.2.1 UF-CMA Security Sketch

It was shown in [KLS17] that for zero-knowledge deterministic signature schemes, if an adversary having quantum access to H and classical access to a signing oracle can produce a forgery of a new message, then there is also an adversary who can produce a forgery without access to the signing oracle (so he only gets the public key).[6] The latter security model is called $\mathsf{UF\text{-}NMA}$ – unforgeability under no-message attack. By the MLWE assumption, the public key $(\mathbf{A}, \mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)$ is indistinguishable from $(\mathbf{A}, \mathbf{t})$ where $\mathbf{t}$ is chosen uniformly at random. The proof that our signature scheme is zero-knowledge is fairly standard and follows the framework from [Lyu09, Lyu12, BG14]. It is formally proved in [KLS17][7] and we sketch the proof in Appendix B.

If we thus assume that the $\mathsf{MLWE}_{k,\ell,D}$ problem is hard, where D is the distribution that samples a uniform integer in the range $[-\eta, \eta]$, then to prove $\mathsf{UF\text{-}NMA}$ security, we only need to analyze the hardness of the experiment where the adversary receives a random $(\mathbf{A}, \mathbf{t})$ and then needs to output a valid message/signature pair $M, (\mathbf{z}, \mathbf{h}, c)$ such that

- $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$
- $\mathsf{H}(\mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) \| M) = c$
- # of 1's in $\mathbf{h}$ is $\leq \omega$

Lemma 1 implies that one can rewrite

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d + \mathbf{u}, \tag{9}$$

where $\|\mathbf{u}\|_\infty \leq 2\gamma_2 + 1$. Furthermore, only ω coefficients of $\mathbf{u}$ will have magnitude greater than γ_2. If we write $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$ where $\|\mathbf{t}_0\|_\infty \leq 2^{d-1}$, then we can rewrite Eq. (9) as

$$\mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d + \mathbf{u} = \mathbf{A}\mathbf{z} - c(\mathbf{t} - \mathbf{t}_0) + \mathbf{u} = \mathbf{A}\mathbf{z} - c\mathbf{t} + (c\mathbf{t}_0 + \mathbf{u}) = \mathbf{A}\mathbf{z} - c\mathbf{t} + \mathbf{u}'. \tag{10}$$

Note that the worst-case upper-bound for $\mathbf{u}'$ is

$$\|\mathbf{u}'\|_\infty \leq \|c\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq \|c\|_1 \cdot \|\mathbf{t}_0\|_\infty + \|\mathbf{u}\|_\infty \leq 60 \cdot 2^{d-1} + 2\gamma_2 + 1 < 4\gamma_2.$$

Thus a (quantum) adversary who is successful at creating a forgery of a new message is able to find $\mathbf{z}, c, \mathbf{u}', M$ such that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, $\|c\|_\infty = 1$, $\|\mathbf{u}'\|_\infty < 4\gamma_2$, and $M \in \{0,1\}^*$ such that

$$\mathsf{H}\left(\left[\mathbf{A} \mid \mathbf{t} \mid \mathbf{I}_k \right] \cdot \begin{bmatrix} \mathbf{z} \\ c \\ \mathbf{u}' \end{bmatrix} \parallel M \right) = c. \tag{11}$$

Since $(\mathbf{A}, \mathbf{t})$ is completely random, this is exactly the definition of the $\mathsf{SelfTargetMSIS}$ problem from above. A standard forking lemma argument can be used to show that an adversary solving the above problem in the (standard) random oracle model can be used to solve the MSIS problem. While giving a reduction using the forking lemma is a good "sanity check", it is not particularly useful for setting parameters due to its lack of tightness. So how does one set parameters? The Fiat-Shamir transform has been used for over 3 decades (and we have been aware of the non-tightness of the forking lemma for two of them), yet the parameter settings for schemes employing it have ignored this loss

[6] It was also shown in [KLS17] that the "deterministic" part of the requirement can be relaxed. The security proof simply loses a factor of the number of different signatures produced per message in its tightness. Thus, for example, if one were to implement the signature scheme (with the same secret key) on several devices with different random-number generators, the security of the scheme would not be affected much.

[7] In that paper, it is actually proved that the underlying zero-knowledge proof is zero-knowledge and then the security of the signature scheme follows via black box transformations.

in tightness. Implicitly, therefore, these schemes rely on the exact hardness of analogues (based on various assumptions such as discrete log [Sch89], one-wayness of RSA [GQ88], etc.) of the problem in Eq. (11).

The intuition for the security of the problem in Eq. (11) (and its discrete log, RSA, etc. analogues) is as follows: since H is a cryptographic hash function whose structure is completely independent of the algebraic structure of its inputs, choosing some M "strategically" should not help – so the problem would be equally hard if the M were fixed. Then, again relying on the independence of H and the algebraic structure of its inputs, the only approach for obtaining a solution appears to be picking some $\mathbf{w}$, computing $\mathsf{H}(\ \mathbf{w} \parallel M\ ) = c$, and then finding $\mathbf{z}, \mathbf{u}'$ such that $\mathbf{Az} + \mathbf{u}' = \mathbf{w} + c\mathbf{t}$.[8] The hardness of finding such $\mathbf{z}, \mathbf{u}'$ with ℓ_∞-norms less than $4\gamma_2$ such that

$$\mathbf{Az} + \mathbf{u}' = \mathbf{t}' \tag{12}$$

for some $\mathbf{t}'$ is the problem whose concrete security we will be analyzing. Note that this is conservative because in Eq. (11) $\|\mathbf{z}\|_\infty < \gamma_1 - \beta \approx 2\gamma_2$. Furthermore, only ω coefficients of $\mathbf{u}'$ can be larger than $2\gamma_2$.

5.2.2 The Addition of the Strong Unforgeabilty Property

To handle the strong-unforgeability requirement, one needs to handle an additional case. Intuitively, the reduction from UF-CMA to UF-NMA used the fact that a forgery of a new message will necessarily require the use of a challenge c for which the adversary has never seen a valid signature (i.e., $(\mathbf{z}, \mathbf{h}, c)$ was never an output by the signing oracle). To prove strong-unforgeability, we also have to consider the case where the adversary sees a signature $(\mathbf{z}, \mathbf{h}, c)$ for M and then only changes $(\mathbf{z}, \mathbf{h})$. In other words, the adversary ends up with two valid signatures such that

$$\mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathsf{UseHint}_q(\mathbf{h}', \mathbf{Az}' - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2).$$

By Lemma 1, the above equality can be shown to imply that there exist $\|\mathbf{z}\|_\infty \leq 2(\gamma_1 - \beta)$ and $\|\mathbf{u}\|_\infty \leq 4\gamma_2 + 2$ such that $\mathbf{Az} + \mathbf{u} = \mathbf{0}$.

5.3 Concrete Security Analysis

In Appendix C, we describe the best known lattice attacks against the problems in Eq. (6) upon which the security of our signature scheme is based. The best attacks involve finding short vectors in some lattice. The main difference between the MLWE and MSIS problems is that the MLWE problem involves finding a short vector in a lattice in which an "unusually short" vector exists. The MSIS problem, on the other hand, involves just finding a short vector in a random lattice. In knapsack terminology, the MLWE problem is a low-density knapsack, while MSIS is a high-density knapsack instance. The analysis for the two instances is slightly different and we analyze the MLWE problem in Appendix C.2 and the MSIS problem (as well as SelfTargetMSIS) in Appendix C.3.

We follow the general methodology from [ADPS16, BCD+16] to analyze the security of our signature scheme, with minor adaptations. This methodology is significantly more conservative than prior ones used in lattice-based cryptography. In particular, we assume the adversary can run the asymptotically best algorithms known, with no overhead compared to the asymptotic run-times. In particular, we assume the adversary can cheaply handle huge amounts of (possibly quantum) memory. This conservatism is in line with the goal of long-term post-quantum security. We note that despite this security analysis methodology, our schemes remain competitive in practice.

[8] This is indeed the (non-tight) proof sketch in the classical random oracle model.

The security parameters in Table 1 are based on this conservative methodology. Since we are making so many approximations (in favor of the adversary), it may seem a little strange that our numbers are so "precisely" stated. The purpose of such precision is to make it possible to compare between our scheme and other lattice-based ones based on the same conservative analysis. On the other hand, because our security levels understates the actual security of the schemes and the best cryptanalytic algorithms are extremely memory-intensive, we believe that our schemes still satisfy their stated "NIST Security Level" designation despite the security numbers appearing to be below the required levels.

While the MLWE and MSIS problems are defined over polynomial rings, we do not currently have any way of exploiting this ring structure, and therefore the best attacks are mounted by simply viewing these problems as LWE and SIS problems. The LWE and SIS problems are exactly as in the definitions of MLWE and MSIS in Section 5.1 with the ring R_q being replaced by $\mathbb{Z}_q$.

5.4 Changing Security Levels

The most straightforward way of raising/lowering the security of Dilithium is by changing the values of (k, ℓ) and then adapting the value of η (and then β and ω) accordingly as in Table 1. Increasing (k, ℓ) by 1 each results in the public key increasing by ≈ 300 bytes and the signature by ≈ 700 bytes; and increases security by ≈ 30 bits.

A different manner in which to increase security would be by lowering the values of γ_1 and/or γ_2. This would make forging signatures (whose hardness is based on the underlying SIS problem) more difficult. Rather than increasing the size of the public key / signature, the negative effect of lowering the γ_i is that signing would require more repetitions. One could similarly increase the value of η in order to make the LWE problem harder at the expense of more repetitions.[9] Because the increase in running time is rather dramatic (e.g. halving both γ_i would end up squaring the number of required repetitions as per Eq. (5)), we recommend increasing (k, ℓ) when needing to "substantially" increase security. Changing the γ_i should be reserved only for slight "tweaks" in the security levels.

References

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*, pages 327–343. USENIX Association, 2016. http://cryptojedi.org/papers/#newhope. 16, 17, 21, 28

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011. 29

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. Quantum attacks on classical proof systems: The hardness of quantum rewinding. In *FOCS*, pages 474–483, 2014. 18

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1006–1018. ACM Press, October 2016. 21

[9] The two changes in this paragraph may require lowering d so as to keep the value $\|c\mathbf{t}_0\|_\infty$ smaller than γ_2 with high probability. Lowering d will increase the size of the public key.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24. ACM-SIAM, January 2016. 27

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In *CT-RSA*, pages 28–47, 2014. 2, 20, 26

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Marc Fischlin and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015. http://cryptojedi.org/papers/#sphincs. 6

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In *CHES*, pages 323–345, 2016. 2

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003. 29

[BS16] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In Robert Krauthgamer, editor, *27th SODA*, pages 893–902. ACM-SIAM, January 2016. 30

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 559–585. Springer, Heidelberg, May 2016. 30

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to ideal-SVP. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 324–348, 2017. 30

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014. 30

[CLP+17] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure mpkc signatures. Cryptology ePrint Archive, Report 2017/636, 2017. https://eprint.iacr.org/2017/636. 6

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, Heidelberg, December 2011. 27

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO (1)*, pages 40–56, 2013. 2, 5

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In *ASIACRYPT*, pages 22–41, 2014. 2, 5

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures – exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. Cryptology ePrint Archive, Report 2017/505, 2017. https://eprint.iacr.org/2017/505 To appear in CCS 2017. 2

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *CHES*, pages 530–547, 2012. 2

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In *CRYPTO*, pages 216–231, 1988. 18, 21

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, Heidelberg, August 2011. 27

[KF17] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 3–26, 2017. 30

[KLS17] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. Cryptology ePrint Archive, Report 2017/916, 2017. https://eprint.iacr.org/2017/916. 4, 5, 10, 19, 20, 26

[Knu97] Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3 edition, 1997. page 145. 7

[Laa15] Thijs Laarhoven. *Search problems in cryptography.* PhD thesis, Eindhoven University of Technology, 2015. 27

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, pages 598–616, 2009. 2, 20

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *EUROCRYPT*, pages 738–755, 2012. 20, 26

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be - attacking strongswan's implementation of post-quantum signatures. Cryptology ePrint Archive, Report 2017/490, 2017. https://eprint.iacr.org/2017/490. To appear in CCS 2017. 2

[PSS+17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017. https://eprint.iacr.org/2017/1014. 10

[SBB+17] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. Cryptology ePrint Archive, Report 2017/985, 2017. https://eprint.iacr.org/2017/985. 10

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, pages 239–252, 1989. 18, 21

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994. 26

A Proofs for Rounding Algorithm Properties

The three lemmas below prove each of the three parts of Lemma 1.

Lemma 3. *Let $r, z \in \mathbb{Z}_q$ with $\|z\|_\infty \leq \alpha/2$. Then*

$$\mathsf{UseHint}_q\left(\mathsf{MakeHint}_q(z, r, \alpha), r, \alpha\right) = \mathsf{HighBits}_q(r + z, \alpha).$$

Proof. The output of $\mathsf{Decompose}_q$ is an integer r_1 such that $0 \leq r_1 < (q-1)/\alpha$ and another integer r_0 such that $\|r_0\|_\infty \leq \alpha/2$. Because $\|z\|_\infty \leq \alpha/2$, the integer $v_1 := \mathsf{HighBits}_q(r + z, \alpha)$ either stays the same as r_1 or becomes $r_1 \pm 1$ modulo $m = (q-1)/\alpha$. More precisely, if $r_0 > 0$, then $-\alpha/2 < r_0 + z \leq \alpha$. This implies that v_1 is either r_1 or $r_1 + 1 \bmod m$. If $r_0 \leq 0$, then we have $-\alpha \leq r_0 + z \leq \alpha/2$. In this case, we have $v_1 = r_1$ or $r_1 - 1 \bmod m$.

The $\mathsf{MakeHint}_q$ routine checks whether $r_1 = v_1$ and outputs 0 if this is so, and 1 if $r_1 \neq v_1$. The $\mathsf{UseHint}_q$ routine uses the "hint" h to either output r_1 (if $y = 0$) or, depending on whether $r_0 > 0$ or not, output either $r_1 + 1 \bmod^+ m$ or $r_1 - 1 \bmod^+ m$.

The lemma below shows that r is not too far away from the output of the $\mathsf{UseHint}_q$ algorithm. This will be necessary for the security of the scheme.

Lemma 4. *Let $(h, r) \in \{0,1\} \times \mathbb{Z}_q$ and let $v_1 = \mathsf{UseHint}_q(h, r, \alpha)$. If $h = 0$, then $\|r - v_1 \cdot \alpha\|_\infty \leq \alpha/2$; else $\|r - v_1 \cdot \alpha\|_\infty \leq \alpha + 1$.*

Proof. Let $(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$. We go through all three cases of the $\mathsf{UseHint}_q$ procedure.

Case 1 ($h = 0$): We have $v_1 = r_1$ and

$$r - v_1 \cdot \alpha = r_1 \cdot \alpha + r_0 - r_1 \cdot \alpha = r_0\,,$$

which by definition has absolute value at most $\alpha/2$.

Case 2 ($h = 1$ and $r_0 > 0$): We have $v_1 = r_1 + 1 - \kappa \cdot (q-1)/\alpha$ for $\kappa = 0$ or 1. Thus

$$\begin{aligned} r - v_1 \cdot \alpha &= r_1 \cdot \alpha + r_0 - (r_1 + 1 - \kappa \cdot (q-1)/\alpha) \cdot \alpha \\ &= -\alpha + r_0 + \kappa \cdot (q-1). \end{aligned}$$

After centered reduction modulo q, the latter has magnitude $\leq \alpha$.

Case 3 ($h = 1$ and $r_0 \leq 0$): We have $v_1 = r_1 - 1 + \kappa \cdot (q-1)/\alpha$ for $\kappa = 0$ or 1. Thus

$$\begin{aligned} r - v_1 \cdot \alpha &= r_1 \cdot \alpha + r_0 - (r_1 - 1 + \kappa \cdot (q-1)/\alpha) \cdot \alpha \\ &= \alpha + r_0 - \kappa \cdot (q-1). \end{aligned}$$

After centered reduction modulo q, the latter quantity has magnitude $\leq \alpha + 1$.

The next lemma will play a role in proving the strong existential unforgeability of our signature scheme. It states that two different h, h' cannot lead to $\mathsf{UseHint}_q(h, r, \alpha) = \mathsf{UseHint}_q(h', r, \alpha)$.

Lemma 5. *Let $r \in \mathbb{Z}_q$ and $h, h' \in \{0,1\}$. If $\mathsf{UseHint}_q(h, r, \alpha) = \mathsf{UseHint}_q(h', r, \alpha)$, then $h = h'$.*

Proof. Note that $\mathsf{UseHint}_q(0, r, \alpha) = r_1$ and $\mathsf{UseHint}_q(1, r, \alpha)$ is equal to $(r_1 \pm 1) \bmod^+ (q-1)/\alpha$. Since $(q-1)/\alpha \geq 2$, we have that $r_1 \neq (r_1 \pm 1) \bmod^+ (q-1)/\alpha$.

We now prove Lemma 2.

Proof. (Of Lemma 2) We prove the lemma for integers, rather than vectors of polynomials, since the HighBits function works independently on each coefficient. If $\|\mathsf{LowBits}_q(r,\alpha)\|_\infty < \alpha/2-\beta$, then $r = r_1\cdot\alpha + r_0$ where $-\alpha/2+\beta < r_0 \le \alpha/2+\beta$. Then $r+s = r_1\cdot\alpha + (r_0+s)$ and $-\alpha/2 < r_0+s \le \alpha/2$. Therefore $r+s \bmod^{\pm} \alpha = r_0+s$, and thus

$$(r+s) - ((r+s) \bmod^{\pm} \alpha) = r_1\cdot\alpha = r - (r \bmod^{\pm} \alpha),$$

and the claim in the Lemma follows.

B Zero-Knowledge Proof

The security of our scheme does not rely on the part of the public key $\mathbf{t}_0$ being secret and so we will be assuming that the public key is $\mathbf{t}$ rather than $\mathbf{t}_1$.

We want to first compute the probability that some particular $(\mathbf{z}, c)$ is generated in Step 17 taken over the randomness of $\mathbf{y}$ and the random oracle H which is modeled as a random function. We have

$$\Pr[\mathbf{z}, c] = \Pr[c]\cdot\Pr[\mathbf{y} = \mathbf{z} - c\mathbf{s}_1 \mid c].$$

Whenever $\mathbf{z}$ has all its coefficients less than $\gamma_1-\beta$ then the above probability is exactly the same for every such tuple $(\mathbf{z}, c)$. This is because $\|c\mathbf{s}_i\|_\infty \le \beta$ (with overwhelming probability), and thus $\|\mathbf{z} - c\mathbf{s}_1\|_\infty \le \gamma_1 - 1$, which is a valid value of $\mathbf{y}$. Therefore, if we only output $\mathbf{z}$ when all its coefficients have magnitudes less than $\gamma_1-\beta$, then the resulting distribution will be uniformly random over $S^{\ell}_{\gamma_1-\beta-1}\times B_{60}$.

The simulation of the signature follows [Lyu12, BG14]. The simulator picks a uniformly random $(\mathbf{z}, c)$ in $S^{\ell}_{\gamma_1-\beta-1}\times B_{60}$, after which it also makes sure that

$$\|\mathbf{r}_0\|_\infty = \|\mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta.$$

By Equation (2), we know that $\mathbf{w} - c\mathbf{s}_2 = \mathbf{A}\mathbf{z} - c\mathbf{t}$, and therefore the simulator can perfectly simulate this as well.

If $\mathbf{z}$ does indeed satisfy $\|\mathsf{LowBits}_q(\mathbf{w}-c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2-\beta$, then as long as $\|c\mathbf{s}_2\|_\infty \le \beta$, we will have

$$\mathbf{r}_1 = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2) = \mathbf{w}_1.$$

Since our β was chosen such that the probability (over the choice of $c, \mathbf{s}_2$) that $\|c\mathbf{s}_2\|_\infty < \beta$ is $> 1 - 2^{-128}$, the simulator does not need to perform the check that $\mathbf{r}_1 = \mathbf{w}_1$ and can always assume that it passes.

We can then program

$$\mathsf{H}(\mu \parallel \mathbf{w}_1) \leftarrow c\,.$$

Unless we have already set the value of $\mathsf{H}(\mu \parallel \mathbf{w}_1)$ to something else, the resulting pair $(\mathbf{z}, c)$ has the same distribution as in a genuine signature of μ. It was shown in [KLS17] that the probability, over the random choice of $\mathbf{A}$ and $\mathbf{y}$, that we already set the value of $\mathsf{H}(\mu \parallel \mathbf{w}_1)$ is less than 2^{-255}.

All the other steps (after Step 19) of the signing algorithm are performed using public information and are therefore simulatable.

C Concrete Security

C.1 Lattice Reduction and Core-SVP Hardness

The best known algorithm for finding very short non-zero vectors in Euclidean lattices is the Block–Korkine–Zolotarev algorithm (BKZ) [SE94], proposed by Schnorr and Euchner

in 1991. More recently, it was proven to quickly converge to its fix-point [HPS11] and improved in practice [CN11]. Yet, what it achieves asymptotically remains unchallenged.

BKZ with block-size b makes calls to an algorithm that solves the Shortest lattice Vector Problem (SVP) in dimension b. The security of our scheme relies on the necessity to run BKZ with a large block-size b and the fact that the cost of solving SVP is exponential in b. The best known classical SVP solver [BDGL16] runs in time $\approx 2^{c_C \cdot b}$ with $c_C = \log_2 \sqrt{3/2} \approx 0.292$. The best known quantum SVP solver [Laa15, Sec. 14.2.10] runs in time $\approx 2^{c_Q \cdot b}$ with $c_Q = \log_2 \sqrt{13/9} \approx 0.265$. One may hope to improve these runtimes, but going below $\approx 2^{c_P \cdot b}$ with $c_P = \log_2 \sqrt{4/3} \approx 0.2075$ would require a theoretical breakthrough. Indeed, the best known SVP solvers rely on covering the b-dimensional sphere with cones of center-to-edge angle $\pi/3$: this requires $2^{c_P \cdot b}$ cones. The subscripts C, Q, P respectively stand for Classical, Quantum and Paranoid.

The strength of BKZ increases with b. More concretely, given as input a basis $(\mathbf{c}_1, \ldots, \mathbf{c}_n)$ of an n-dimensional lattice, BKZ repeatedly uses the b-dimensional SVP-solver on lattices of the form $(\mathbf{c}_{i+1}(i), \ldots, \mathbf{c}_j(i))$ where $i \leq n$, $j = \min(n, i+b)$ and where $\mathbf{c}_k(i)$ denotes the projection of $\mathbf{c}_k$ orthogonally to the vectors $(\mathbf{c}_1, \ldots, \mathbf{c}_i)$. The effect of these calls is to flatten the curve of the $\ell_i = \log_2 \|\mathbf{c}_i(i-1)\|$'s (for $i = 1, \ldots, n$). At the start of the execution, the ℓ_i's typically decrease fast, at least locally. As BKZ preserves the determinant of the $\mathbf{c}_i$'s, the sum of the ℓ_i's remains constant throughout the execution, and after a (small) polynomial number of SVP calls, BKZ has made the ℓ_i's decrease less. It can be heuristically estimated that for sufficiently large b, the local slope of the ℓ_i's converges to

$$slope(b) = \frac{1}{b-1} \log_2 \left(\frac{b}{2\pi \mathrm{e}} (\pi \cdot b)^{1/b} \right),$$

unless the local input ℓ_i's are already too small or too large. The quantity $slope(b)$ decreases with b, implying that the larger b the flatter the output ℓ_i's.

In our case, the input ℓ_i's are of the following form (cf. Fig. 10). The first ones are all equal to $\log_2 q$ and the last ones are all equal to 0. BKZ will flatten the jump, decreasing ℓ_i's with small i's and increasing ℓ_i's with large i's. However, the local slope $slope(b)$ may not be sufficiently small to make the very first ℓ_i's decrease and the very last ℓ_i's increase. Indeed, BKZ will not increase (resp. increase) some ℓ_i's if these are already smaller (resp. larger) than ensured by the local slope guarantee. In our case, the ℓ_i's are always of the following form at the end of the execution:

- The first ℓ_i's are constant equal to $\log_2 q$ (this is the possibly empty Zone 1).
- Then they decrease linearly, with slope $slope(b)$ (this is the never-empty Zone 2).
- The last ℓ_i's are constant equal to 0 (this is the possibly empty Zone 3).

The graph is continuous, i.e., if Zone 1 (resp. Zone 3) is not empty, then Zone 2 starts with $\ell_i = \log_2 q$ (resp. ends with $\ell_i = 0$).

C.2 Solving MLWE

Any $\mathsf{MLWE}_{\ell,k,D}$ instance for some distribution D can be viewed as an LWE instance of dimensions $256 \cdot \ell$ and $256 \cdot k$. Indeed, the above can be rewritten as finding $\mathrm{vec}(\mathbf{s}_1), \mathrm{vec}(\mathbf{s}_2) \in \mathbb{Z}^{256 \cdot \ell} \times \mathbb{Z}^{256 \cdot k}$ from $(\mathrm{rot}(\mathbf{A}), \mathrm{vec}(\mathbf{t}))$, where $\mathrm{vec}(\cdot)$ maps a vector of ring elements to the vector obtained by concatenating the coefficients of its coordinates, and $\mathrm{rot}(\mathbf{A}) \in \mathbb{Z}_q^{256 \cdot k \times 256 \cdot \ell}$ is obtained by replacing all entries $a_{ij} \in R_q$ of $\mathbf{A}$ by the 256×256 matrix whose z-th column is $\mathrm{vec}\left(x^{z-1} \cdot a_{ij}\right)$.

Given an LWE instance, there are two lattice-based attacks. The primal attack and the dual attack. Here, the primal attack consists in finding a short non-zero vector in the

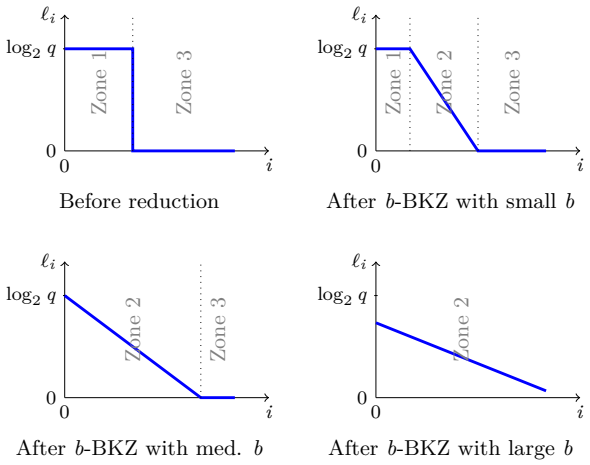


Figure 10: Evolution of Gram-Schmidt length in log-scale under BKZ reduction for various blocksizes. The area under the curves remains constant, and the slope in Zone 2 decrease with the blocksize. Note that Zone 3 may disappear before Zone 1, depending on the shape of the input basis.

lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^d : \mathbf{Mx} = \mathbf{0} \bmod \mathbf{q}\}$ where $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[1:m]}|\mathbf{I}_m|\mathrm{vec}(\mathbf{t})_{[1:m]})$ is an $m \times d$ matrix where $d = 256 \cdot \ell + m + 1$ and $m \leq 256 \cdot k$. Indeed, it is sometime not optimal to use all the given equations in lattice attacks.

We tried all possible number m of rows, and, for each trial, we increased the blocksize of b until the value ℓ_{d-b} obtained as explained above was deemed sufficiently large. As explained in [ADPS16, Sec. 6.3], if $2^{\ell_{d-b}}$ is greater than the expected norm of $(\mathrm{vec}(\mathbf{s}_1), \mathrm{vec}(\mathbf{s}_2))$ after projection orthogonally to the first $d - b$ vectors, it is likely that the MLWE solution can be easily extracted from the BKZ output.

The dual attack consists in finding a short non-zero vector in the lattice $\Lambda' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^d : \mathbf{M}^T\mathbf{x} + \mathbf{y} = 0 \bmod q)\}$, $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[1:m]})$ is an $m \times d$ matrix where $d = 256 \cdot \ell$ and $m \leq 256 \cdot k$. Again, for each value of m, we increased the value of b until the value ℓ_1 obtained as explained above was deemed sufficiently small according the analysis of [ADPS16, Sec. 6.3].

C.3 Solving MSIS and SelfTargetMSIS

As per the discussion in Section 5.2.1, the best known attack against the SelfTargetMSIS problem involves either breaking the security of H or solving the problem in Eq. (12). The latter amounts to solving the $\mathsf{MSIS}_{k,\ell+1,\zeta}$ problem for the matrix $[\ \mathbf{A} \mid \mathbf{t}'\ ]$. [10]

Note that the MSIS instance can be mapped to a $\mathsf{SIS}_{256\cdot k,256\cdot(\ell+1),\zeta}$ instance by considering the matrix $\mathrm{rot}(\mathbf{A}|\mathbf{t}') \in \mathbb{Z}_q^{256\cdot k \times 256\cdot(\ell+1)}$. The attack against the $\mathsf{MSIS}_{k,\ell,\zeta'}$ instance in Eq. (6) can similarly be mapped to a $\mathsf{SIS}_{256\cdot k,256\cdot\ell,\zeta'}$ instance by considering the matrix $\mathrm{rot}(\mathbf{A}) \in \mathbb{Z}_q^{256\cdot k \times 256\cdot\ell}$. The attacker may consider a subset of w columns, and let the solution coefficients corresponding to the dismissed columns be zero.

Remark 1. An unusual aspect here is that we are considering the infinity norm, rather than the Euclidean norm. Further, for our specific parameters, the Euclidean norms of the solutions are above q. In particular, the vector $(q, 0, \ldots, 0)^T$ belongs to the lattice, has

[10] Note that a solution to Eq. (12) would require the coefficient in from of $\mathbf{t}'$ to be ± 1, while we're allowing any small polynomial. Furthermore, as discussed after Eq. (12), some parts of the real solution are smaller than the bound ζ, but we're ignoring this for the sake of being conservative with our analysis.

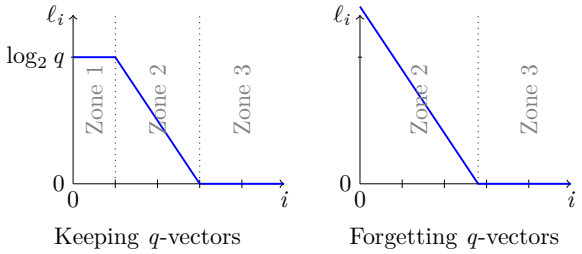


Figure 11: Effect of forgetting q-vectors by randomization, under the same BKZ-blocksize b.

Euclidean norm below that of the solution, but its infinity norm above the requirement. This raises difficulties in analyzing the strength of BKZ towards solving our infinity norm SIS instances: indeed, even with small values of b, the first ℓ_i's are short (they correspond to q-vectors), even though they are not solutions.

For each number w of selected columns and for each value of b, we compute the estimated BKZ output ℓ_i's, as explained above. We then consider the smallest i such that ℓ_i is below $\log_2 q$ and the largest j such that ℓ_j above 0. These correspond to the vectors that were modified by BKZ, with smallest and largest indices, respectively. In fact, for the same cost as a call to the SVP-solver, we can obtain $\sqrt{4/3}^b$ vectors with Euclidean norm $\approx 2^{\ell_i}$ after projection orthogonally to the first $i-1$ basis vectors. Now, let us look closely at the shape of such a vector. As the first $i-1$ basis vectors are the first $i-1$ canonical unit vectors multiplied by q, projecting orthogonally to these consists in zeroing the first $i-1$ coordinates. The remaining $w-i+1$ coordinates have total Euclidean norm $\approx 2^{\ell_i} \approx q$, and the last $w-j$ coordinates are 0. We heuristically assume that these coordinates have similar magnitudes $\sigma \approx 2^{\ell_i}/\sqrt{j-i+1}$; we model each such coordinate as a Gaussian of standard deviation σ. We assume that each one of our $\sqrt{4/3}^b$ vectors has its first $i-1$ coordinates independently uniformly distributed modulo q, and finally compute the probability that all coordinates in both ranges $[0, i-1]$ and $[i, j]$ are less than B in absolute value. Our cost estimate is the inverse of that probability multiplied by the run-time of our b-dimensional SVP-solver.

Forgetting q-vectors. For all the parameter sets in Table 1, the best parametrization of the attack above kept the basis in a shape with a non-trivial Zone 1. We note that the coordinates in this range have a quite lower probability of passing the ℓ_∞ constraint than coordinates in Zone 2. We therefore considered a strategy consisting of "forgetting" the q-vectors, by re-randomizing the input basis before running the BKZ algorithm. For the same blocksize b, this makes Zone 1 of the output basis disappear (BKZ does not find the q-vectors), at the cost of producing a basis with first vectors of larger Euclidean norms. This is depicted in Fig. 11.

It turns out that this strategy always improves over the previous strategy for the parameter ranges considered in Table 1. We therefore used this strategy for our security estimates.

C.4 On Other Attacks

For our parameters, the BKW [BKW03] and Arora–Ge [AG11] families of algorithms are far from competitive.

Algebraic attacks. One specificity of our LWE and SIS instances is that they are inherited from MLWE and MSIS instances. One may wonder whether the extra algebraic structure of the resulting lattices can be exploited by an attacker. The line of work of [CGS14, BS16, CDPR16, CDW17] did indeed find new cryptanalytic results on certain algebraic lattices, but [CDW17] mentions serious obstacles towards breaking cryptographic instances of Ring-LWE. By switching from Ring-LWE to MLWE, we get even further away from those weak algebraic lattice problems.

Dense sublattice attacks. Kirchner and Fouque [KF17] showed that the existence of many linearly independent and unexpectedly short lattice vectors (much shorter than Minkowski's bound) helps BKZ run better than expected in some cases. This could happen for our primal LWE attack, by extending $\mathbf{M} = (\mathrm{rot}(\mathbf{A})_{[\mathbf{1:m}]}|\mathbf{I_m}|\mathrm{vec}(\mathbf{t})_{[\mathbf{1:m}]})$ to $(\mathrm{rot}(\mathbf{A})_{[1:m]}|\mathbf{I}_m|\mathrm{rot}(\mathbf{t})_{[1:m]})$: the associated lattice now has 256 linearly independent short vectors rather than a single one. The Kirchner-Fouque analysis of BKZ works best if both q and the ratio between the number of unexpectedly short vectors and the lattice dimension are high. In the NTRU case, for example, the ratio is $1/2$, and, for some schemes derived from NTRU, the modulus q is also large. We considered this refined analysis of BKZ in our setup, but, to become relevant for our parameters, it requires a parameter b which is higher than needed with the usual analysis of BKZ. Note that [KF17] also arrived to the conclusion that this attack is irrelevant in the small modulus regime, and is mostly a threat to fully homomorphic encryption schemes and cryptographic multilinear maps.

Note that, once again, the switch from Ring-LWE to MLWE takes us further away from lattices admitting unconventional attacks. Indeed, the dimension ratio of the dense sub-lattice is $1/2$ in NTRU, at most $1/3$ in lattices derived from Ring-LWE, and at most $1/(\ell+2)$ in lattices derived from MLWE.

Specialized attack against ℓ_∞-SIS. At last, we would like to mention that it is not clear whether the attack sketched in Appendix C.3 above for SIS in infinity norm is optimal. Indeed, as we have seen, this approach produces many vectors, with some rather large uniform coordinates (at indices $1, \ldots, i$), and smaller Gaussian ones (at indices $i, \ldots, j$). In our current analysis, we simply hope that one of the vector satisfies the ℓ_∞ bound. Instead, one could combine them in ways that decrease the size of the first (large) coefficients, while letting the other (small) coefficients grow a little bit.

This situation created by the use of ℓ_∞-SIS (see Remark 1) has — to the best of our knowledge — not been studied in detail. After a preliminary analysis, we do not consider such an improved attack a serious threat to our concrete security claims, especially in light of the approximations already made in favor of the adversary.

CRYSTALS-Kyber

Algorithm Specifications And Supporting Documentation

Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint,
Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé

November 30, 2017

Contents

1 Written specification

KYBER is an IND-CCA2-secure key-encapsulation mechanism (KEM), which has first been described in [18]. The security of KYBER is based on the hardness of solving the learning-with-errors problem in module lattices (MLWE problem [51]). The construction of KYBER follows a two-stage approach: we first introduce an IND-CPA-secure public-key encryption scheme encrypting messages of a fixed length of 32 bytes, which we call KYBER.CPAPKE. We then use a slightly tweaked Fujisaki–Okamoto (FO) transform [35] to construct the IND-CCA2-secure KEM. Whenever we want to emphasize that we are speaking about the IND-CCA2-secure KEM, we will refer to it as KYBER.CCAKEM.

In Subsection 1.1 we give preliminaries and fix notation. In Subsection 1.2 we give a full specification of KYBER.CPAPKE. Subsection 1.3 gives details of the transform that we use in KYBER to obtain KYBER.CCAKEM from KYBER.CPAPKE. Subsection 1.4 lists the parameters that we propose for different security levels. Finally, Subsection 1.5 explains the design rationale behind KYBER.

1.1 Preliminaries and notation.

Bytes and byte arrays. Inputs and outputs to all API functions of KYBER are byte arrays. To simplify notation, we denote by by $\mathcal{B}$ the set $\{0, \ldots, 255\}$, i.e., the set of 8-bit unsigned integers (bytes). Consequently we denote by $\mathcal{B}^k$ the set of byte arrays of length k and by $\mathcal{B}^*$ the set of byte arrays of arbitrary length (or byte streams). For two byte arrays a and b we denote by $(a\|b)$ the concatenation of a and b. For a byte array a we denote by $a + k$ the byte array starting at byte k of a (with indexing starting at zero). For example, let a by a byte array of length ℓ, let b be another byte array and let $c = (a\|b)$ be the concatenation of a and b; then $b = a + \ell$. When it is more convenient to work with an array of bits than an array of bytes we make this conversion explicit via the BytesToBits function that takes as input an array of ℓ bytes and produces as output an array of 8ℓ bits. Bit β_i at position i of the output bit array is obtained from byte $b_{i/8}$ at position $i/8$ of the input array by computing $\beta_i = \left((b_{i/8}/2^{(i \mod 8)}) \mod 2\right)$.

Polynomial rings and vectors. We denote by R the ring $\mathbb{Z}[X]/(X^n+1)$ and by R_q the ring $\mathbb{Z}_q[X]/(X^n+1)$, where $n = 2^{n'-1}$ such that $X^n + 1$ is the $2^{n'}$-th cyclotomic polynomial. Throughout this document, the values of n, n' and q are fixed to $n = 256$, $n' = 9$, and $q = 7681$. Regular font letters denote elements in R or R_q (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in R or R_q. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $\mathbf{v}$ (or matrix $\mathbf{A}$), we denote by $\mathbf{v}^T$ (or $\mathbf{A}^T$) its transpose. For a vector $\mathbf{v}$ we write $\mathbf{v}[i]$ to denote it's i-th entry (with indexing starting at zero); for a matrix $\mathbf{A}$ we write $\mathbf{A}[i][j]$ to denote the entry in row i, column j (again, with indexing starting at zero).

Modular reductions. For an even (resp. odd) positive integer α, we define $r' = r \bmod^{\pm} \alpha$ to be the unique element r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$) such that $r' = r \bmod \alpha$. For any positive integer α, we define $r' = r \bmod^{+} \alpha$ to be the unique element r' in the range $0 \leq r' < \alpha$ such that $r' = r \bmod \alpha$. When the exact representation is not important, we simply write $r \bmod \alpha$.

Rounding. For an element $x \in \mathbb{Q}$ we denote by $\lceil x \rfloor$ rounding of x to the closest integer with ties being rounded up.

Sizes of elements. For an element $w \in \mathbb{Z}_q$, we write $\|w\|_\infty$ to mean $|w \bmod^{\pm} q|$. We now define the ℓ_∞ and ℓ_2 norms for $w = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1} \in R$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \quad \|w\| = \sqrt{\|w_0\|_\infty^2 + \ldots + \|w_{n-1}\|_\infty^2}.$$

Similarly, for $\mathbf{w} = (w_1, \ldots, w_k) \in R^k$, we define

$$\|\mathbf{w}\|_\infty = \max_i \|w_i\|_\infty, \quad \|\mathbf{w}\| = \sqrt{\|w_1\|^2 + \ldots + \|w_k\|^2}.$$

Sets and Distributions. For a set S, we write $s \leftarrow S$ to denote that s is chosen uniformly at random from S. If S is a probability distribution, then this denotes that s is chosen according to the distribution S.

Compression and Decompression. We now define a function $\mathsf{Compress}_q(x, d)$ that takes an element $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$. We furthermore define a function $\mathsf{Decompress}_q$, such that

$$x' = \mathsf{Decompress}_q\big(\mathsf{Compress}_q(x, d), d\big) \tag{1}$$

is an element close to x – more specifically

$$|x' - x \bmod^{\pm} q| \leq B_q \coloneqq \left\lceil \frac{q}{2^{d+1}} \right\rfloor .$$

The functions satisfying these requirements are defined as:

$$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod^{+} 2^d ,$$
$$\mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor .$$

When $\mathsf{Compress}_q$ or $\mathsf{Decompress}_q$ is used with $x \in R_q$ or $\mathbf{x} \in R_q^k$, the procedure is applied to each coefficient individually.

The main reason for defining the $\mathsf{Compress}_q$ and $\mathsf{Decompress}_q$ functions is to be able to discard some low-order bits in the public key and the ciphertext, which do not have much effect on the correctness probability of decryption – thus reducing the size of public keys and ciphertexts.

Yet, the $\mathsf{Compress}_q$ and $\mathsf{Decompress}_q$ are also used for another purpose than compression, namely to perform the usual LWE error correction during encryption and decryption. More precisely, in line 20 of the encryption procedure (Algorithm 5) the $\mathsf{Decompress}_q$ function is used to create error tolerance gaps by sending 0 to 0 and 1 to $\lceil q/2 \rfloor$. Later on, on line 4 of the decryption procedure (Algorithm 6), the $\mathsf{Compress}_q$ function is used to decrypt to a 1 if $v - \mathbf{s}^T\mathbf{u}$ is closer to $\lceil q/2 \rfloor$ than to 0, and decrypt to a 0 otherwise.

Symmetric primitives. The design of Kyber makes use of a pseudorandom function $\mathsf{PRF}\colon \mathcal{B}^{32} \times \mathcal{B} \to \mathcal{B}^*$ and of an extendable output function $\mathsf{XOF}\colon \mathcal{B}^* \to \mathcal{B}^*$. Kyber also makes use of two hash functions $\mathrm{H}\colon \mathcal{B}^* \to \mathcal{B}^{32}$ and $\mathrm{G}\colon \mathcal{B}^* \to \mathcal{B}^{32} \times \mathcal{B}^{32}$.

NTTs and bitreversed order. A very efficient way to perform multiplications in R_q is via the so-called *number-theoretic transform* (NTT). For a polynomial $g = \sum_{i=0}^{n-1} g_i X^i \in R_q$ we define the polynomial $\hat{g}$ *in* NTT *domain* as

$$\mathsf{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$
$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

where we fix the n-th primitive root of unity to $\omega = 3844$ and thus $\psi = \sqrt{\omega} = 62$. The motivation of choosing $\psi = 62$ is that it is the smallest integer that has multiplicative order 512 modulo 7681, i.e., the smallest integer whose square is an n-th root of unity.

The inverse NTT^{-1} of the function NTT is essentially the same as the computation of NTT, except that it uses $\omega^{-1} \bmod q = 6584$, multiplies by powers of $\psi^{-1} \bmod q = 1115$ after the summation, and also multiplies each coefficient by the scalar $n^{-1} \bmod q = 7651$, so that

$$\mathsf{NTT}^{-1}(\hat{g}) = g = \sum_{i=0}^{n-1} g_i X^i, \text{ with}$$
$$g_i = n^{-1}\psi^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}.$$

For the parameters of Kyber (see Subsection 1.4), both NTT and NTT^{-1} can be computed very efficiently *in place*, i.e., without requiring any additional memory. However, this means that the output coefficients will be stored in memory in a permuted order. More specifically, they will be in *bitreversed* order, i.e.,

coefficient $\hat{a}_i$ will be stored at position $\mathsf{br}_{256}(i)$, where br_{256} reverses the bits in an 8-bit integer. For example, $\mathsf{br}_{256}(142) = 113$. Our implementations of KYBER use an NTT which assumes inputs to be in "normal" (i.e., not bitreversed) order, returning output in NTT domain in bitreversed order and an NTT^{-1} which assumes inputs to be in bitreversed order and computes outputs in normal order.

Using NTT and NTT^{-1} we can compute the product $f \cdot g$ of two elements $f, g \in R_q$ very efficiently as $\mathsf{NTT}^{-1}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g))$, where $\circ$ denotes pointwise or coefficient-wise multiplication.

When we apply NTT or NTT^{-1} to a vector or matrix of elements of R_q, then this means that the respective operation is applied to each entry individually. When we apply $\circ$ to matrices or vectors it means that we perform a usual matrix multiplication, but that the individual products of entries are computed as pointwise multiplications of coefficients. Throughout the document we will write NTT and NTT^{-1} whenever we refer to the concrete functions as defined above and use normal-font NTT whenever we refer to the general technique.

Uniform sampling in R_q. KYBER uses a deterministic approach to sample elements in R_q that are statistically close to a uniformly random distribution. For this sampling we use a function $\mathsf{Parse}\colon \mathcal{B}^* \to R_q$, which receives as input a byte stream $B = b_0, b_1, b_2, \ldots$ and computes an element $\hat{a} = \hat{a}_0 + \hat{a}_1 X + \hat{a}_2 X^2 + \cdots + \hat{a}_{n-1} X^{n-1}$ in R_q (which is assumed to be in NTT domain) as described in Algorithm 1 (note that this description of Parse assumes $q = 7681$ and in particular $\lceil \log_2(q) \rceil = 13$).

Algorithm 1 $\mathsf{Parse}\colon \mathcal{B}^* \to R_q$

Input: Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
Output: Polynomial $\hat{a} \in R_q$, assumed to be in NTT domain

```
1  i := 0
2  j := 0
3  while j < n do
4      d := b_i + 256 · b_{i+1}
5      d := d mod+ 2^13
6      if d < q then
7          â_{br_256(j)} := d
8          j := j + 1
9      end if
10     i := i + 2
11 end while
12 return â_0 + â_1 X + ··· + â_{n-1} X^{n-1}
```

The intuition behind the function Parse is that if the input byte array is statistically close to a uniformly random byte array, then the output polynomial is statistically close to a uniformly random element of R_q. We can assume that this element is in NTT domain, because the NTT maps polynomials with uniformly random coefficients to polynomials with again uniformly random coefficients. Note that in line 7 we bitreverse the index; this ensures that when Parse writes output as a consecutive stream, the output is in NTT domain with bitreversed coefficient order; i.e., in precisely the format that typical implementations of NTT^{-1} take as input.

Sampling from a binomial distribution. Noise in KYBER is sampled from a centered binomial distribution B_η for some positive integer η. We define B_η as follows:

$$\text{Sample } (a_1, \ldots, a_\eta, b_1, \ldots, b_\eta) \leftarrow \{0,1\}^{2\eta}$$

$$\text{and output } \sum_{i=1}^{\eta} (a_i - b_i).$$

When we write that a polynomial $f \in R_q$ or a vector of such polynomials is sampled from B_η, we mean that each coefficient is sampled from B_η.

For the specification of KYBER we need to define how a polynomial $f \in R_q$ is sampled according to B_η deterministically from 64η bytes of output of a pseudorandom function (we fix $n = 256$ in this description). This is done by the function CBD (for "centered binomial distribution") defined as described in Algorithm 2.

Algorithm 2 $\mathsf{CBD}_\eta\colon \mathcal{B}^{64\eta} \to R_q$

Input: Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$
  $(\beta_0, \ldots, \beta_{512\eta-1}) \coloneqq \mathsf{BytesToBits}(B)$
  for i from 0 to 255 **do**
    $a \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
    $b \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
    $f_i \coloneqq a - b$
  end for
  return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

Encoding and decoding. There are two data types that Kyber needs to serialize to byte arrays: byte arrays and (vectors of) polynomials. Byte arrays are trivially serialized via the identity, so we need to define how we serialize and deserialize polynomials. In Algorithm 3 we give a pseudocode description of the function Decode_ℓ, which deserialize-serializes an array of 32ℓ bytes into a polynomial $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$ (we again fix $n = 256$ in this description) with each coefficient f_i in $\{0, \ldots, 2^\ell - 1\}$. We define the function Encode_ℓ as the inverse of Decode_ℓ. Whenever we apply Encode_ℓ to a vector of polynomials we encode each polynomial individually and concatenate the output byte arrays.

Algorithm 3 $\mathsf{Decode}_\ell\colon \mathcal{B}^{32\ell} \to R_q$

Input: Byte array $B \in \mathcal{B}^{32\ell}$
Output: Polynomial $f \in R_q$
  $(\beta_0, \ldots, \beta_{256\beta-1}) \coloneqq \mathsf{BytesToBits}(B)$
  for i from 0 to 255 **do**
    $f_i \coloneqq \sum_{j=0}^{\ell-1} \beta_{i\ell+j} 2^j$
  end for
  return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

1.2 Specification of Kyber.CPAPKE

Kyber.CPAPKE is essentially the LPR encryption scheme that was introduced (for Ring-LWE) by Lyubashevsky, Peikert, and Regev in the presentation of [54] at Eurocrypt 2010 [55]; the description is also in the full version of the paper [56, Sec. 1.1]. The roots of this scheme go back to the first LWE-based encryption scheme presented by Regev in [71, 72] and even further to the NTRU cryptosystem presented by Hoffstein, Pipher, and Silverman in [41].

The main modification we apply to the LPR encryption scheme is to use Module-LWE instead of Ring-LWE. Also, we adopt the approach taken by Alkım, Ducas, Pöppelmann and Schwabe in [5] for the generation of the public matrix $\mathbf{A}$. Furthermore, we shorten public keys and ciphertexts by "bit dropping" via learning-with-rounding [10, Eq. 2.1], which is a common technique for reducing the output sizes in lattice-based schemes (c.f. [63, 69]).

Parameters. Kyber.CPAPKE is parameterized by integers n, k, q, η, d_u, d_v, and d_t. As stated before, throughout this document n is always 256 and q is always 7681. Furthermore, throughout this document d_u and d_t will always be 11 and d_v will always be 3. The values of k and η vary for different security levels.

Using the notation of Subsection 1.1 we give the definition of key generation, encryption, and decryption of the Kyber.CPAPKE public-key encryption scheme in Algorithms 4, 5, and 6. A more high-level view of these algorithms is given in the comments.

Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation

Output: Secret key $sk \in \mathcal{B}^{13 \cdot k \cdot n/8}$
Output: Public key $pk \in \mathcal{B}^{d_t \cdot k \cdot n/8+32}$

$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) \coloneqq \mathrm{G}(d)$
$N \coloneqq 0$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho \| j \| i))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from B_η
  $\mathbf{s}[i] \coloneqq \mathsf{CBD}_\eta(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from B_η
  $\mathbf{e}[i] \coloneqq \mathsf{CBD}_\eta(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
end for
$\hat{\mathbf{s}} \coloneqq \mathsf{NTT}(\mathbf{s})$
$\mathbf{t} \coloneqq \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}) + \mathbf{e}$
$pk \coloneqq (\mathsf{Encode}_{d_t}(\mathsf{Compress}_q(\mathbf{t}, d_t)) \| \rho)$ ▷ $pk \coloneqq \mathbf{As} + \mathbf{e}$
$sk \coloneqq \mathsf{Encode}_{13}(\hat{\mathbf{s}} \mod^+ q)$ ▷ $sk \coloneqq \mathbf{s}$
return (pk, sk)

Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

Input: Public key $pk \in \mathcal{B}^{d_t \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$

$N \coloneqq 0$
$\mathbf{t} \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_t}(pk), d_t)$
$\rho \coloneqq pk + d_t \cdot k \cdot n/8$
for i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho \| i \| j))$
  end for
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from B_η
  $\mathbf{r}[i] \coloneqq \mathsf{CBD}_\eta(\mathsf{PRF}(r, N))$
  $N \coloneqq N + 1$
end for
for i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from B_η
  $\mathbf{e}_1[i] \coloneqq \mathsf{CBD}_\eta(\mathsf{PRF}(r, N))$
  $N \coloneqq N + 1$
end for
$e_2 \coloneqq \mathsf{CBD}_\eta(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from B_η
$\hat{\mathbf{r}} \coloneqq \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} \coloneqq \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} \coloneqq \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v \coloneqq \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{t})^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decode}_1(\mathsf{Decompress}_q(m, 1))$ ▷ $v \coloneqq \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
$c_1 \coloneqq \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 \coloneqq \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
return $c = (c_1 \| c_2)$ ▷ $c \coloneqq (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

Algorithm 6 KYBER.CPAPKE.Dec(sk, c): decryption

Input: Secret key $sk \in \mathcal{B}^{13 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$

$\mathbf{u} \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
$v \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
$\hat{\mathbf{s}} \coloneqq \mathsf{Decode}_{13}(sk)$
$m \coloneqq \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m \coloneqq \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
return m

1.3 Specification of KYBER.CCAKEM

We construct the KYBER.CCAKEM IND-CCA2-secure KEM from the IND-CPA-secure public-key encryption scheme described in the previous subsection via a slightly tweaked Fujisaki–Okamoto transform [35]. In Algorithms 7, 8, and 9 we define key generation, encapsulation, and decapsulation of KYBER.CCAKEM.

Algorithm 7 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{d_t \cdot k \cdot n/8+32}$
Output: Secret key $sk \in \mathcal{B}^{(13+d_t) \cdot k \cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \text{H}(pk) \| z)$
4: **return** (pk, sk)

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{d_t \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^{32}$
1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \text{H}(m)$ ▷ Do not send output of system RNG
3: $(\bar{K}, r) := \text{G}(m \| \text{H}(pk))$
4: $c := \text{KYBER.CPAPKE.Enc}(pk, m; r)$
5: $K := \text{H}(\bar{K} \| \text{H}(c))$
6: **return** (c, K)

Algorithm 9 KYBER.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{(13+d_t) \cdot k \cdot n/8+96}$
Output: Shared key $K \in \mathcal{B}^{32}$
1: $pk := sk + 13 \cdot k \cdot n/8$
2: $h := sk + (13 + d_t) \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
3: $z := sk + (13 + d_t) \cdot k \cdot n/8 + 64$
4: $m' := \text{KYBER.CPAPKE.Dec}(\mathbf{s}, (\mathbf{u}, v))$
5: $(\bar{K}', r') := \text{G}(m' \| h)$
6: $c' := \text{KYBER.CPAPKE.Enc}(pk, m', r')$
7: **if** $c = c'$ **then**
8: **return** $K := \text{H}(\bar{K}' \| \text{H}(c))$
9: **else**
10: **return** $K := \text{H}(z \| \text{H}(c))$
11: **end if**
12: **return** K

1.4 KYBER parameter sets

We define three parameter sets for KYBER, which we call KYBER512, KYBER768, and KYBER1024. The parameters are listed in Table 1. Note that the table also lists the derived parameter δ, which is the probability that decapsulation of a valid KYBER.CCAKEM ciphertext fails. The parameters were obtained via the following approach:

Table 1: Parameter sets for KYBER

	n	k	q	η	(d_u, d_v, d_t)	δ
KYBER512	256	2	7681	5	$(11, 3, 11)$	2^{-145}
KYBER768	256	3	7681	4	$(11, 3, 11)$	2^{-142}
KYBER1024	256	4	7681	3	$(11, 3, 11)$	2^{-169}

- n is set to 256 because the goal is to encapsulate 256-bit symmetric keys. Smaller values of n would require to encode multiple key bits into one polynomial coefficient, which requires lower noise levels and therefore lowers security. Larger values of n would reduce the capability to easily scale security via parameter k.
- q is set to the smallest prime satisfying $2n \mid (q-1)$; this is required to enable the fast NTT-based multiplication.
- k is selected to fix the lattice dimension as a multiple of n; changing k is the main mechanism in KYBER to scale security (and as a consequence, efficiency) to different levels.
- The remaining parameters η, d_u, d_v, and d_t were chosen to balance between security (see Section 4), public-key and ciphertext size, and failure probability. Note that all three parameter sets achieve a failure probability of $< 2^{-128}$ with some margin. We discuss this in more detail in Subsections 1.5 and 5.3. We decided to fix d_u and d_t to the same value, which slightly simplifies implementations.

The failure probability δ is computed with the help of the `Kyber.py` Python script which is available online at https://github.com/pq-crystals/kyber/tree/master/scripts/. For the theoretical background of that script see [18, Theorem 1].

Instantiating PRF, XOF, H, and G. What is still missing to complete the specification of KYBER is the instantiation of the symmetric primitives. We instantiate all of those primitives with functions from the FIPS-202 standard [60] as follows:

- We instantiate XOF with SHAKE-128;
- we instantiate H with SHA3-256;
- we instantiate G with SHA3-512; and
- we instantiate $\mathsf{PRF}(s, b)$ with SHAKE-256$(s||b)$.

1.5 Design rationale

The design of KYBER is based on the module version [51] of the Ring-LWE LPR encryption scheme [54] with bit-dropping [63, 69]. It is also enhanced by many of the improvements of preceding implementations of lattice-based encryption schemes such as NEWHOPE [5]. In NEWHOPE (and all other Ring-LWE schemes), operations were of the form $\mathbf{As}+\mathbf{e}$ where all the variables were polynomials in some ring. The main difference in KYBER is that $\mathbf{A}$ is now a matrix (with a small dimension like 3) over a constant-size polynomial ring and $\mathbf{s}, \mathbf{e}$ are vectors over the same ring. We refer to this as a scheme over "module lattices."

The use of Module-LWE. Previous proposals of LWE-based cryptosystems either used the very structured Ring-LWE problem (as, for example, NEWHOPE [5]) or standard LWE (as, for example, Frodo [17]). The main advantage of structured LWE variants based on polynomial rings is efficiency in terms of both speed and key and ciphertext sizes. The disadvantages are concerns that the additional structure might enable more efficient attacks and that tradeoffs between efficiency and security can be scaled only rather coarsely. The advantages of standard LWE is the lack of structure and easy scalability, but those come at the cost of significantly decreasing efficiency. Module-LWE offers a trade-off between these two extremes. In the specific case of the Module-LWE parameters used in KYBER, we obtain somewhat reduced structure compared to

Ring-LWE, much better scalability, and—when encrypting messages of a fixed size of 256 bits—performance very similar to Ring-LWE-based schemes.

Active security. In [19], Bos, Costello, Naehrig, and Stebila used a *passively secure* KEM to migrate TLS to transitional post-quantum security (i.e., post-quantum confidentiality, but only pre-quantum authentication). Subsequent work, like NewHope [5] or Frodo [17] followed up and proposed more efficient and more conservative instantiations of the underlying passively secure KEM. One advantage of passively secure KEMs is that they can accept a higher failure probability (which allows to either increase security by increasing noise or decreasing public-key and ciphertext size). The other advantage is that they do not require a CCA transform, and therefore come with faster decapsulation. Despite these advantages, Kyber is defined as an IND-CCA2 secure KEM only. For many applications like public-key encryption (via a KEM-DEM construction) or in authenticated key exchange active security is mandatory. However, also in use cases (like key exchange in TLS) that do not strictly speaking require active security, using a an actively secure KEM has advantages. Most notably, it allows (intentional or accidental) caching of ephemeral keys. Furthermore, the CCA transform of Kyber protects against certain bugs in implementations. Specifically, passively secure schemes will not notice if the communication partner uses "wrong" noise, for example, all-zero noise. Such a bug in the encapsulation of Kyber will immediately be caught by the re-encryption step during decapsulation. As a conclusion, we believe that the overhead of providing CCA security is not large enough to justify saving it and making the scheme less robust.

The role of the NTT. Multiplication in R_q based on the number-theoretic transform (NTT) has multiple advantages: it is extremely fast, does not require additional memory (like, for example, Karatsuba or Toom multiplication) and can be done in very little code space. Consequently, it has become common practice to choose parameters of lattice-based crypto to support this very fast multiplication algorithm. Some schemes go further and make the NTT part of the definition of the scheme. A prominent example is again NewHope, which samples the public value $\mathbf{a}$ in NTT domain and also sends messages in NTT domain to save 2 NTTs. NewHope was not the first scheme to do this; for earlier examples see [53, 69, 73].

In Kyber we also decided to make the NTT part of the definition of the scheme, but only in the sampling of $\mathbf{A}$. A consequence of this decision is that the NTT appears in the specification of Kyber.CPAPKE. Note that multiplications by $\mathbf{A}$ have to use the NTT, simply because $\hat{\mathbf{A}}$ is sampled in NTT domain[1]. As a consequence, implementations will also want to use the NTT for all other multiplications, so we make those invocations of NTT and NTT^{-1} also explicit in Alg. 4, Alg. 5, and Alg. 6. Note that also the secret key sk is stored in NTT domain.

We could have chosen to *not* make the NTT part of the definition of Kyber, which would have increased simplicity of the description. The cost for this increased simplicity would have been k^2 additional NTT operations in both key generation and encapsulation, which would result in a significant slowdown. We could have also chosen to send messages in NTT domain like NewHope does. However, this would prevent us from compressing public-key and ciphertext via the $\mathsf{Compress}_q$ function and thus result in larger ciphertexts and public keys.

Against all authority. For the generation of the public uniformly random matrix $\mathbf{A}$, we decided to adopt the "against-all-authority" approach of NewHope. This means that the matrix is not a system parameter but instead generated freshly as part of every public key. There are two advantages to this approach: First, this avoids discussions about how exactly a uniformly random system parameter was generated. Second, it protects against the all-for-the-price-of-one attack scenario of an attacker using a serious amount of computation to find a short basis of the lattice spanned by $\mathbf{A}$ *once* and then using this short basis to attack *all* users. The cost for this decision is the expansion of the matrix $\mathbf{A}$ from a random seed during key generation and encapsulation; we discuss this cost more in Subsection 2.1.

Binomial noise. Theoretic treatments of LWE-based encryption typically consider LWE with Gaussian noise, either rounded Gaussian [71] or discrete Gaussian [21]. As a result, many early implementations also sampled noise from a discrete Gaussian distribution, which turns out to be either fairly inefficient (see, for example, [19]) or vulnerable to timing attacks (see, for example, [22, 68, 33]). The performance of the best known attacks against LWE-based encryption does not depend on the exact distribution of noise, but rather on the standard deviation (and potentially the entropy). This motivates the use of noise distributions that

[1] An alternative would be to apply NTT^{-1} to $\hat{\mathbf{A}}$ but that would counteract the whole point of sampling $\mathbf{A}$ in NTT domain.

we can easily, efficiently, and securely sample from. One example is the centered binomial distribution used in [5]. Another example is the use of "learning-with-rounding" (LWR), which adds deterministic uniform noise by dropping bits as in KYBER's $\mathsf{Compress}_q$ function. In the design of KYBER we decided to use centered binomial noise and thus rely on LWE instead of LWR as the underlying problem. The compression of ciphertexts via $\mathsf{Compress}_q$ introduces additional noise (making the scheme more secure), but we do not consider this noise in our security analysis (this choice is motivated by the absence of a Ring/Module variant of a hardness reduction for LWR [10]).

Allowing decapsulation failures. Another interesting design decision is whether to allow decapsulation failures (i.e., decryption failures in KYBER.CPAPKE) or choose parameters that not only have a negligible, but a zero chance of failure. The advantages of zero failure probability are obvious: CCA transforms and security proofs become easier and we could have avoided a whole discussion of attacks exploiting decapsulation failures in Subsection 5.3. The disadvantage of designing LWE-based encryption with zero failures is that it means either decreasing security against attacks targeting the underlying lattice problem (by significantly decreasing the noise) or decreasing performance (by compensating for the loss in security via an increase of the lattice dimension). The decision to allow failure probabilities of $< 2^{-140}$ in all parameter sets of KYBER reflects the intuition that

- decapsulation failures are a problem if they appear with non-negligible probability; but
- attacks attempting to exploit failures that occur with extremely low probability as in KYBER are a much smaller threat than, for example, improvements to hybrid attacks [43] targeting schemes with very low noise.

Additional Hashes. In the CCA transform we hash the (hash of the) public key pk into the pre-key $\bar{K}$ and into the random coins r (see line 3 of Alg. 8), and we hash the (hash of the) ciphertext into the final key K. These hashes would not be necessary for the security reduction (see Section 4), but they add *robustness*. Specifically, the final shared key output by KYBER.CCAKEM depends on the full view of exchanged messages (public key and ciphertext), which means that the KEM is contributory and safe to use in authenticated key exchanges without additional hashing of context. Hashing pk also into the random coins r adds protection against a certain class of multi-target attacks that attempt to make use of protocol failures. This is discussed in more detail in Subsection 5.3.

Choice of symmetric primitives. In the design of KYBER we need an extendable output function (XOF), two hash functions, and a pseudorandom function. We decided to rely on only one underlying primitive for all those functions. This helps to reduce code size in embedded platforms and (for a conservative choice) reduces concerns that KYBER could be attacked by exploiting weaknesses in *one out of several* symmetric primitives. There are only relatively few extendable output functions described in the literature. The best known ones, which also coined the term XOF, are the SHAKE functions based on Keccak [15] and standardized in FIPS-202 [60]. This standard conveniently also describes hash functions with the output lengths we need; furthermore, SHAKE is designed to also work as a PRF. These properties of the FIPS-202 function family made the choice easy, but there are still two decisions that may need explanation:

- We could have chosen to instantiate all symmetric primitives with only *one* function (e.g., SHAKE-256) from the FIPS-202 standard. The choice of SHAKE-128 as instantiation of the XOF is actually important for performance; also we do not need any of the traditional security properties of hash functions from SHAKE-128, but rather that the output "looks uniformly random". In an earlier version of KYBER we instantiated H, G, and PRF all with SHAKE-256. We decided to change this to *different* functions from the FIPS-202 family to avoid any domain-separation discussion. Note that this decision increases code-size at most marginally: all 4 functions can be obtained by a call to a "Keccak" function with appropriate arguments (see, for example, [14]).
- We could have decided to use KMAC from NIST Special Publication 800-185 to instantiate the PRF. We decided against this, because it would increase the numbers of Keccak permutations required in the generation of the noise polynomials and thus noticeably and unnecessarily decrease performance.

Supporting non-incremental hash APIs. In line 3 of Alg. 8 we feed H(pk) (instead of pk) into G and in line 5 we feed H(c) (instead of c) into H. Using H(pk) in the call to G enables a small speedup for decapsulation as described in Subsection 2.1. However, there is another reason why we first hash pk and c, namely that it simplifies implementing KYBER with a non-incremental hash API. If KYBER is implemented in an environment which already offers a library for hashing, but only offers calls of the form $h = \mathrm{H}(m)$, then producing a hash of the form $h = \mathrm{H}(m_1 \| m_2)$ would first require copying m_1 and m_2 into one consecutive area of memory. This would require unnecessary copies and, more importantly, additional stack space. Such non-incremental hash APIs are not uncommon: one example is the API of NaCl [13].

Return value for decapsulation failure. Traditionally the FO transform returns $\perp$ (i.e., a special failure symbol) when decapsulation fails. We use a variant that instead sets the resulting shared key to a pseudo-random value computed as the hash of a secret z and the ciphertext c. This variant of the FO transform was proven secure in [42]. In practice it has the advantage that implementations of KYBER's decapsulation are safe to use even if higher level protocols fail to check the return value. In fact, it would be safe to always return "success" (i.e., `0` in the NIST API for KEMs). Our implementation of decapsulation returns a negative value on failure to allow the caller to abort early and not continue working with a key that would produce failures in later protocol stages (e.g., MAC verifications).

2 Performance analysis

In this section we consider implementational aspects of KYBER and report performance results of two implementations: the ANSI C reference implementation requested by NIST and an implementation optimized using AVX2 vector instructions included in the submission package under `Additional_Implementations/avx2/`. We remark that the optimized implementation in ANSI C in subdirectory `Optimized_Implementation/`, as requested by the Call for Proposals, is a copy of the reference implementation.

2.1 Implementation considerations and tradeoffs

Implementing the NTT. Many different tradeoffs are possible when implementing the number-theoretic transform. The most important ones are between code size (which becomes mainly relevant on embedded processors) and speed. The two implementations of KYBER included in the submission package have a dedicated forward NTT (from normal to bitreversed order) and inverse NTT (from bitreversed to normal order). Also, both implementations use precomputed tables of powers of ω and ψ. What is particularly interesting about using the NTT on embedded platforms is that the multiplication of two elements of R_q can be computed without any additional temporary storage. What is particularly interesting about using the NTT on large processors is that it is extremely efficiently vectorizable. Since 2013, the most efficient approach to compute the NTT on 64-bit Intel processors was to represent coefficients as double-precision floating-point values [39, 5]. In our AVX2-optimized implementation of KYBER, we show that carefully optimizing the NTT using AVX2 *integer* instructions results in much better performance. Specifically, on Intel Haswell CPUs one (forward or inverse) NTT in KYBER takes only about 480 cycles.

Keccak. The second speed-critical component inside KYBER are the symmetric primitives, i.e., SHA3-256, SHA3-512, SHAKE-128, and SHAKE-256, all based on the Keccak permutation. SHA3 has the reputation to not be the fastest hash function in software (see, for example, [50]). To some extent this is compensated by the fact that most calls to Keccak are parallel and thus very efficiently vectorizable. Our AVX2 implementation makes use of this fact. Also, ARM recently announced that future ARMv8 processors will have hardware support for SHA3 [38], so there is a good chance that at least on some architectures, software performance of SHA3 will not be an issue in the future.

Hardware-RNGs for key generation. During key generation, both our implementations use SHAKE-256 to generate the secret terms $\mathbf{s}$ and $\mathbf{e}$; however, this is not required. The choice of RNG during key generation is a local decision that any user and platform can make independently. In particular on platforms with fast hardware AES (like the AES-NI instructions on modern Intel processors), one can speed up key generation by using AES-256 in counter mode to generate the uniformly random noise that is then fed as input into CBD. We considered using this in our AVX2 implementation, but using this optimization means that testvectors

would not match between our two implementations. This is not an issue in actual deployments, where `randombytes` is not deterministic.

Caching of ephemeral keys. Applications that are even more conscious of key-generation time can decide to cache ephemeral keys for some time. This is enabled by the fact that KYBER is IND-CCA2 secure.

Tradeoffs between secret-key size and speed. It is possible to use different tradeoffs between secret-key size and decapsulation speed. If secret-key size is critical, it is of course possible to not store H(pk) and also to not store the public key as part of the secret key but instead recompute it during decapsulation. Furthermore, not keeping the secret key in NTT domain makes it possible to compress each coefficient to only 5 bits, resulting in a total size of only 320 bytes for the three polynomials. Finally, as all randomness in key generation is generated from two 32-byte seeds, it is also possible to only store these seeds and re-run key generation during decapsulation.

In the other direction, if secret-key size does not matter very much and decapsulation speed is critical, one might decide to store the expanded matrix $\mathbf{A}$ as part of the secret key and avoid recomputation from the seed ρ during the re-encapsulation part of decapsulation.

Both implementations included in the submission package use the secret-key format described in Algorithm 7, i.e., with polynomials in NTT domain, including the public key and H(pk), but not including the matrix $\mathbf{A}$.

Local storage format of static public keys. A user who is frequently encapsulating messages to the same public key can speed up encapsulation by locally storing an expanded public key containing the matrix $\mathbf{A}$ and H(pk). This saves the cost of expanding the matrix $\mathbf{A}$ from the seed ρ and the cost of hashing pk in every encapsulation.

2.2 Performance of reference and AVX2 implementations

Table 2 reports performance results of the reference implementation and of our implementation optimized using AVX2 vector instructions. All benchmarks were obtained on one core of an Intel Core i7-4770K (Haswell) processor clocked at 3491.789 MHz (as reported by `/proc/cpuinfo`) with TurboBoost and hyperthreading disabled. The benchmarking machine has 32 GB of RAM and is running Debian GNU/Linux with Linux kernel version 4.9.0. Both implementations were compiled with gcc version 6.3.0. We used compiler flags `-O3 -fomit-frame-pointer -march=native -fPIC`. to compile both implementations. All cycle counts reported are the median of the cycle counts of 10 000 executions of the respective function. The implementations are not optimized for memory usage, but generally KYBER has only very modest memory requirements. This means that in particular our implementations do not need to allocate any memory on the heap.

3 Known Answer Test values

All KAT values are included in subdirectories of the directory `KAT` of the submission package. Specifically, the KAT values of KYBER512 are in the subdirectory `KAT/kyber512`; the KAT values of KYBER768 are in the subdirectory `KAT/kyber768`; and the KAT values of KYBER1024 are in the subdirectory `KAT/kyber1024`. Each of those directories contains the KAT values as generated by the `PQCgenKAT_kem` program provided by NIST. Specifically, those files are:

- `KAT/kyber512/PQCkemKAT_1632.req`,
- `KAT/kyber512/PQCkemKAT_1632.rsp`,
- `KAT/kyber768/PQCkemKAT_2400.req`,
- `KAT/kyber768/PQCkemKAT_2400.rsp`,
- `KAT/kyber1024/PQCkemKAT_3168.req`, and
- `KAT/kyber1024/PQCkemKAT_3168.rsp`.

Table 2: Key and ciphertext sizes and cycle counts for all paramter sets of KYBER. Cycle counts were obtained on one core of an Intel Core i7-4770K (Haswell); "ref" refers to the C reference implementation, "AVX2" to the implementation using AVX2 vector instructions; **sk** stands for secret key, **pk** for public key, and **ct** for ciphertext.

KYBER512					
Sizes (in Bytes)		**Haswell Cycles (ref)**		**Haswell Cycles (AVX2)**	
sk:	1632	**gen:**	141 872	**gen:**	55 160
pk:	736	**enc:**	205 468	**enc:**	75 680
ct:	800	**dec:**	246 040	**dec:**	74 428
KYBER768					
Sizes (in Bytes)		**Haswell Cycles (ref)**		**Haswell Cycles (AVX2)**	
sk:	2400	**gen:**	243 004	**gen:**	85 472
pk:	1088	**enc:**	332 616	**enc:**	112 660
ct:	1152	**dec:**	394 424	**dec:**	108 904
KYBER1024					
Sizes (in Bytes)		**Haswell Cycles (ref)**		**Haswell Cycles (AVX2)**	
sk:	3168	**gen:**	368 564	**gen:**	121 056
pk:	1440	**enc:**	481 042	**enc:**	157 964
ct:	1504	**dec:**	558 740	**dec:**	154 952

4 Expected security strength

4.1 Security definition

KYBER.CCAKEM (or short, KYBER) is an IND-CCA2-secure key encapsulation mechanism, i.e., it fulfills the security definition stated in Section 4.A.2 of the Call for Proposals.

4.2 Rationale of our security estimates

Our estimates of the security strength for the three different parameter sets of KYBER—and consequently the classification into security levels as defined in Section 4.A.5 of the Call for Proposals—are based on the cost estimates of attacks against the underlying module-learning-with-errors (MLWE) problem as detailed in Subsection 5.1.

To justify this rationale, we will in the following give two reductions from MLWE: a tight reduction in the random-oracle model (ROM) in Theorem 2 and a non-tight reduction in the quantum-random-oracle model (QROM) in Theorem 3. With those reductions at hand, there remain two avenues of attack that would break KYBER without solving the underlying MLWE problem, namely

1. breaking one of the assumptions of the reductions, in particular attacking the symmetric primitives used in KYBER; or
2. exploiting the non-tightness of the QROM reduction.

We briefly discuss 1.) in Subsection 5.2. The discussion of 2.) requires considering two separate issues, namely

- a (quadratic) non-tightness in the decryption-failure probability of KYBER.CPAPKE, and
- a (quadratic) non-tightness between the advantage of the MLWE attacker and the quantum attacker against KYBER.

In Subsection 5.3 we discuss quantum attacks exploiting decryption failures and in the presentation of the non-tight QROM reduction we explain why the non-tightness between quantum attacks against MLWE and quantum attacks against KYBER is unlikely to matter in practice. More specifically, we show how to eliminate this non-tightness if we allow the reasonable, but non-standard, assumption that KYBER.CPAPKE ciphertexts are pseudorandom, even if all randomness is generated pseudorandomly from a hash of the encrypted message.

4.3 Security Assumption

The hard problem underlying the security of our schemes is Module-LWE [20, 51]. It consists in distinguishing uniform samples $(\mathbf{a}_i, b_i) \leftarrow R_q^k \times R_q$ from samples $(\mathbf{a}_i, b_i) \in R_q^k \times R_q$ where $\mathbf{a}_i \leftarrow R_q^k$ is uniform and $b_i = \mathbf{a}_i^T \mathbf{s} + e_i$ with $\mathbf{s} \leftarrow B_\eta^k$ common to all samples and $e_i \leftarrow B_\eta$ fresh for every sample. More precisely, for an algorithm A, we define $\mathbf{Adv}^{\mathrm{mlwe}}_{m,k,\eta}(\mathsf{A}) =$

$$\left| \Pr\left[b' = 1 \ : \ \begin{array}{l} \mathbf{A} \leftarrow R_q^{m \times k}; (\mathbf{s}, \mathbf{e}) \leftarrow \beta_\eta^k \times \beta_\eta^m; \\ \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}; b' \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{b}) \end{array} \right] - \Pr\left[b' = 1 \ : \ \mathbf{A} \leftarrow R_q^{m \times k}; \mathbf{b} \leftarrow R_q^m; b' \leftarrow \mathsf{A}(\mathbf{A}, \mathbf{b}) \ \right] \right|.$$

4.3.1 Tight reduction from MLWE in the ROM

We first note that KYBER.CPAPKE is tightly IND-CPA secure under the Module-LWE hardness assumption.

Theorem 1. *Suppose* XOF *and* G *are random oracles. For any adversary* A*, there exist adversaries* B *and* C *with roughly the same running time as that of* A *such that* $\mathbf{Adv}^{\mathrm{cpa}}_{\text{KYBER.CPAPKE}}(\mathsf{A}) \leq 2 \cdot \mathbf{Adv}^{\mathrm{mlwe}}_{k+1,k,\eta}(\mathsf{B}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{PRF}}(\mathsf{C})$.

The proof of this theorem is easily obtained by noting that, under the MLWE assumption, public-key and ciphertext are pseudo-random.

KYBER.CCAKEM is obtained via a slightly tweaked Fujisaki-Okamoto transform [42, 35] applied to KYBER.CPAPKE. The following concrete security statement proves KYBER.CCAKEM's IND-CCA2-security when the hash functions G and H are modeled as random oracles. It is obtained by combining the generic bounds from [42] with Theorem 1 (and optimizing the constants appearing in the bound).

Theorem 2. *Suppose* XOF*,* H*, and* G *are random oracles. For any classical adversary* A *that makes at most* q_{RO} *many queries to random oracles* XOF*,* H *and* G*, there exist adversaries* B *and* C *of roughly the same running time as that of* A *such that*

$$\mathbf{Adv}^{\mathrm{cca}}_{\text{KYBER.CCAKEM}}(\mathsf{A}) \leq 2\mathbf{Adv}^{\mathrm{mlwe}}_{k+1,k,\eta}(\mathsf{B}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{PRF}}(\mathsf{C}) + 4q_{RO}\delta.$$

Note that the security bound is tight. The negligible additive term $4q_{RO}\delta$ stems from KYBER.CPAPKE's decryption-failure probability δ.

4.3.2 Non-tight reduction from MLWE in the QROM

As for security in the quantum random oracle model (QROM), [42, 74] proved that KYBER.CCAKEM is IND-CCA2 secure in the QROM, provided that KYBER.CPAPKE is IND-CPA secure. A slightly tighter reduction can be obtained by requiring the base scheme KYBER.CPAPKE to be pseudo-random. Pseudo-randomness [74] requires that, for every message m, a (randomly generated) ciphertext $(c_1, c_2) \leftarrow \text{KYBER.CPAPKE.Enc}(pk, m)$ is computationally indistinguishable from a random ciphertext of the form $(\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$, for uniform $(\mathbf{u}, v)$. (We also require the property of "statistical disjointness" [74] which is trivially fulfilled for KYBER.CPAPKE.) The proof of KYBER.CPAPKE's IND-CPA security indeed shows that KYBER.CPAPKE is tightly pseudo-random under the Module-LWE hardness assumption.

Theorem 3. *Suppose* XOF*,* H*, and* G *are random oracles. For any quantum adversary* A *that makes at most* q_{RO} *many queries to quantum random oracles* XOF*,* H *and* G*, there exist quantum adversaries* B *and* C *of roughly the same running time as that of* A *such that*

$$\mathbf{Adv}^{\mathrm{cca}}_{\text{KYBER.CCAKEM}}(\mathsf{A}) \leq 4q_{RO} \cdot \sqrt{\mathbf{Adv}^{\mathrm{mlwe}}_{k+1,k,\eta}(\mathsf{B})} + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{PRF}}(\mathsf{C}) + 8q_{RO}^2\delta.$$

Table 3: Classical and quantum core-SVP hardness of the different proposed parameter sets of KYBER together with the claimed security level as defined in Section 4.A.5 of the Call for Proposals. Complexities are given in terms of the base-2 logarithm of the number of operations.

	core-SVP (classical)	core-SVP (quantum)	Claimed security level
KYBER512	112	102	**1** (AES-128)
KYBER768	178	161	**3** (AES-192)
KYBER1024	241	218	**5** (AES-256)

Unfortunately, the above security bound is non-tight and therefore can only serve as an asymptotic indication of KYBER.CCAKEM's CCA-security in the quantum random oracle model.

Tight reduction under non-standard assumption. We can use [42, 74] to derive a tight security bound in the QROM from a non-standard security assumption, namely that a deterministic version of KYBER.CPAPKE, called DKYBER.CPAPKE, is pseudo-random in the QROM. Deterministic KYBER.CPAPKE is defined as KYBER.CPAPKE, but the random coins r used in encryption are derived deterministically from the message m as $r := \mathrm{G}(m)$. Pseudo-randomness for deterministic encryption states that an encryption (c_1, c_2) of a randomly chosen message is computationally indistinguishable from a random ciphertext $(\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$, for uniform $(\mathbf{u}, v)$. In the classical ROM, pseudo-randomness of DKYBER.CPAPKE is tightly equivalent to MLWE but in the QROM the reduction is non-tight (and is the reason for the term $q_{RO} \cdot \sqrt{\mathbf{Adv}^{\mathrm{mlwe}}_{k+1,k,\eta}(\mathsf{B})}$ in Theorem 3). Concretely, we obtain the following bound:

$$\mathbf{Adv}^{\mathrm{cca}}_{\mathrm{KYBER.CCAKEM}}(\mathsf{A}) \leq 2\mathbf{Adv}^{\mathrm{mlwe}}_{k+1,k,\eta}(\mathsf{B}) + \mathbf{Adv}^{\mathrm{pr}}_{\mathrm{DKYBER.CPAPKE}}(\mathsf{C}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{PRF}}(\mathsf{D}) + 8q^2_{RO}\delta.$$

We remark that we are not aware of any quantum attack on deterministic KYBER.CPAPKE that performs better than breaking the MLWE problem.

4.4 Estimated security strength

Table 3 lists the security levels according to the definition in Section 4.A.5 of the Call for Proposals for the different parameter sets of KYBER. Our claims are based on the cost estimates of the best known attacks against the MLWE problem underlying KYBER as detailed in Subsection 5.1. Specifically we list the classical and the quantum *core-SVP hardness* and use those to derive security levels.

The impact of MAXDEPTH. The best known quantum speedups for the sieving algorithm, which we consider in our cost analysis (see Subsection 5.1.1), are only mildly affected by limiting the depth of a quantum circuit, because it uses Grover search on sets of small size (compared to searching through the whole keyspace of AES). For the core-SVP-hardness operation estimates to match the quantum gate cost of breaking AES at the respective security levels, a quantum computer would need to support a maximum depth of 70–80. When limiting the maximum depth to smaller values, or when considering classical attacks, the core-SVP-hardness estimates are smaller than the gate counts for attacks against AES. We discuss this difference in the following.

Gates to break AES vs. core-SVP hardness. The classical core-SVP hardness of the MLWE problem underlying KYBER differs by a factor of $\approx 2^{30}$ from the gate count to classically break the corresponding AES instances. The core-SVP hardness is a very conservative lower bound on the cost of an actual attack against the MLWE problem (for details, see Subsection 5.1). Specifically, the core-SVP-hardness ignores

- the (polynomial) number of calls to the SVP oracle that are required to solve the MLWE problem;
- the gate count required for one "operation";
- additional cost of sieving with asymptotically subexponential complexity;
- the cost of access into exponentially large memory; and

- the additional rounding noise (the LWR problem, see [10, 6]), *i.e.* the deterministic, uniformly distributed noise introduced in ciphertexts via the $\mathsf{Compress}_q$ function.

The state of research into SVP-solving algorithms is way too premature to assign meaningful cost estimates to each of those items. However, it seems clear that in any actual implementation of an attack algorithm the *product* of the cost of those items will exceed 2^{30}. See also the paragraph *"How conservative is this analysis?"* in Subsection 5.1.4.

4.5 Additional security properties

4.5.1 Forward secrecy.

Kyber has a very efficient key-generation procedure (see also Section 2) and is therefore particularly well suited for applications that use frequent key generations to achieve forward secrecy.

4.5.2 Side-channel attacks.

Timing attacks. Neither straight-forward reference implementations nor optimized implementations of Kyber use any secret-dependent branches or table lookups[2].This means that typical implementations of Kyber are free from the two most notorious sources of timing leakage. Another possible source of timing leakage are non-constant-time multipliers like the `UMULL` instruction on ARM Cortex-M3 processors, which multiplies two 32-bit integers to obtain a 64-bit result. However, multiplications in Kyber have only 16-bit inputs, and most non-constant-time multipliers show timing variation only for larger inputs. For example, on ARM Cortex-M3 processors the obvious way to implement multiplications in Kyber is through the constant-time `MUL` instruction, which multiplies two 32-bit integers, but returns only the bottom 32-bits of the result. What remains as a source of timing leakage are modular reductions, which are sometimes implemented via conditional statements. However, timing leakage in modular reductions is easily avoided by using (faster) Montgomery [59] and Barrett reductions [11] as illustrated in our reference and AVX2 implementations.

Differential attacks. We expect that any implementation of Kyber without dedicated protection against differential power or electromagnetic radiation (EM) attacks will be vulnerable to such attacks. This is true for essentially any implementation of a cryptographic scheme that uses long-term (non-ephemeral) keys. Deployment scenarios of Kyber in which an attacker is assumed to have the power to mount such an attack require specially protected—typically masked—implementations. In [62], Oder, Schneider, Pöppelmann, and Güneysu present such a masked implementation of Ring-LWE decryption with a CCA transform very similar to the one used in Kyber. The implementation targets Cortex-M4F microcontrollers; the conclusion of the work is that protecting the decryption (decapsulation) step against first-order DPA incurs an overhead of about about a factor of 5.5. The techniques presented in that paper also apply to Kyber and we expect that the overhead for protecting Kyber against differential attacks is in the same ballpark.

Template attacks. Protections against differential attacks do not help if an attacker is able to recover even ephemeral secrets from a single power or EM trace. At CHES 2017, Primas, Pessl, and Mangard presented such a single-trace attack against an implementation of Ring-LWE on a Cortex-M4F microcontroller [70]. The attacker model in this attack is rather strong: it is the typical setting of template attacks, which assumes an attacker who is able to generate template traces on known inputs on a device with leakage very similar to the actual target device. In [70], the authors used *the same device* for generating target traces and in the attack. The attack was facilitated (maybe even enabled) by the fact that the implementation under attack used variable-time modular reductions. Consequently, the paper states that *"One of the first measures to strengthen an implementation against SPA attacks is to ensure a constant runtime and control flow"*. This is the case for all implementations of Kyber. The attack from [70] would thus certainly not straight-forwardly apply to implementations of Kyber, but more research is required to investigate whether also constant-time implementations of Kyber (and other lattice-based schemes) succumb to template attacks, and what the cost of suitable countermeasures is.

[2]Note that the rejection sampling in generating the matrix $\mathbf{A}$ does not involve any secret data.

4.5.3 Multi-target attacks

Our security analysis makes no formal claims about security bounds in the multi-target setting. However, in the design of KYBER we made two decisions that aim at improving security against attackers targeting multiple users:

- We adopt the "against-all-authority" approach of re-generating the matrix $\mathbf{A}$ for each public key from NEWHOPE [5]. This protects against an attacker attempting to break *many keys at the cost of breaking one key*.
- In the CCA transform (see Alg. 8) we hash the public key into the pre-key $\bar{K}$ and the coins r. Making the coins r dependent of the public key protects against precomputation attacks that attempt to break *one out of many keys*. For details, see Subsection 5.3.

4.5.4 Misuse resilience

The first, and most important, line of defense against misuse is the decision to make IND-CCA2 security non-optional. As discussed in Subsection 1.5, it would have been possible to achieve slightly shorter public keys and ciphertexts, and faster decapsulation, in a CPA-secure variant of KYBER. Using IND-CCA2 security by default makes it safe to use KYBER with static keys and as a consequence also to re-use ephemeral keys for some time. What is *not* safe, is to reuse the same randomness in encapsulation, but that randomness is also not exposed to the outside by the API. The CCA transform has a second effect in terms of robustness: it protects against a broken implementation of the noise sampling. A rather peculiar aspect of LWE-based cryptography is that it will pass typical functional tests even if one communication partner does not add any noise (or by accident samples all-zero noise). The deterministic generation of noise via SHAKE-256 during encapsulation and the re-encryption step during decapsulation will reveal such an implementation mistake immediately.

An additional line of defense against misuse is to hash the public-key into the "pre-key" $\bar{K}$ and thus make sure that the KEM is contributory. Only few protocols require a KEM to be contributory and those protocols can always turn a non-contributory KEM into a contributory one by hashing the public key into the final key. Making this hash part of the KEM design in KYBER ensures that nothing will go wrong on the protocol level if implementers omit the hash there.

A similar statement holds for additionally hashing the ciphertext into the final key. Several protocols need to ensure that the key depends on the complete view of exchanged protocol messages. This is the case, for example, for the authenticated-key-exchange protocols described in the KYBER paper [18, Sec. 5]. Hashing the full protocol view (public key and ciphertext) into the final key already as part of the KEM makes it unnecessary (although of course still safe) to take care of these hashes on the higher protocol layer.

5 Analysis with respect to known attacks

5.1 Attacks against the underlying MLWE problem

MLWE as LWE. The best known attacks against the underlying MLWE problem in KYBER do not make use of the structure in the lattice. We therefore analyze the hardness of the MLWE problem as an LWE problem. We briefly discuss the current state of the art in *algebraic attacks*, i.e., attacks that exploit the structure of module lattices (or ideal lattices) at the end of this subsection.

5.1.1 Attacks against LWE.

Many algorithms exist for solving LWE (for a survey see [4]), but many of those are irrelevant for our parameter set. In particular, because there are only $m = (k+1)n$ LWE samples available to the attacker, we can rule out BKW types of attacks [45] and linearization attacks [8]. This essentially leaves us with two BKZ [75, 25] attacks, usually referred to as primal and dual attacks that we will recall in Subsections 5.1.2 and 5.1.3.

The algorithm BKZ proceeds by reducing a lattice basis using an SVP oracle in a smaller dimension b. It is known [40] that the number of calls to that oracle remains polynomial, yet concretely evaluating the number of calls is rather painful, and this is subject to new heuristic ideas [25, 24, 7]. We choose to ignore this polynomial factor, and rather evaluate only the *core SVP hardness*, that is the cost of *one call* to an SVP oracle in dimension b, which is clearly a pessimistic estimation (from the defender's point of view). This approach to deriving a conservative cost estimate for attacks against LWE-based cryptosystems was introduced in [5, Sec. 6].

Enumeration vs. sieving. There are two algorithmic approaches for the SVP oracle in BKZ: enumeration and sieving algorithms. These two classes of algorithms have very different performance characteristics and, in particular for sieving, it is hard to predict how *practical performance* scales from lattice dimensions that have been successfully tackled to larger dimensions that are relevant in attacks against cryptosystems like Kyber. The starting point of such an analysis is the fact that enumeration algorithms have super-exponential running time, while sieving algorithms have only exponential running time. Experimental evidence from typical implementations of BKZ [37, 25, 29] shows that enumeration algorithms are more efficient in "small" dimensions, so one question is at what dimension sieving becomes more efficient. So far it seems that sieving is slower in practice for accessible dimensions of up to $b \approx 130$. However, a recent work [31] showed (in the classical setting) that sieving techniques can be sped up *in practice* for exact-SVP, being now less than an order of magnitude slower than enumeration already in dimension 60 to 80.

The analysis is complicated by the fact that sieving algorithms are much more *memory intensive* than enumeration algorithms. Specifically, sieving algorithms have exponential complexity not only in time, but also in memory, while enumeration algorithms require only small amounts of memory. In practice, the cost of access to memory increases with the size of memory, which typically only becomes noticeable once the memory requirement exceeds fast local memory (RAM). There is no study, yet, that investigates the algorithmic optimization and practical performance of sieving using slow background storage.

We follow the approach of [5, Sec. 6] to obtain a conservative lower bound on the performance of both sieving and enumeration for the dimensions that are relevant for the cryptanalysis of Kyber. This approach works in the RAM model, i.e., it assumes that access into even exponentially large memory is free. Under this assumption sieving becomes more efficient than even sophisticated enumeration, with serious optimization as described in [25] and with quantum speedups, for dimensions larger than 250, quite possibly already earlier. The smallest dimension that we are interested in for the cryptanalysis of Kyber is 390, so that the performance of sieving in the RAM model serves as a conservative lower bound for the performance of both enumeration and sieving.

A lot of recent work has pushed the efficiency of the original lattice sieve algorithms [61, 58], improving the heuristic complexity from $(4/3)^{b+o(b)} \approx 2^{0.415b}$ down to $\sqrt{3/2}^{b+o(b)} \approx 2^{0.292b}$ using *Locality Sensitive Hashing* (LSH) techniques [48, 12]. The hidden sub-exponential factor is known to be much greater than one in practice. Again, we ignore this factor to arrive at a security estimate with a conservative margin. Most of the sieving algorithms have been shown [49, 47] to benefit from Grover's quantum search algorithm, bringing the complexity down to $2^{0.265b}$. We will use $2^{0.292b}$ as the classical and $2^{0.265b}$ and the quantum cost estimate of both the primal and dual attacks with block size (dimension) b. We recall those two attacks in the following.

5.1.2 Primal attack.

The primal attack consists of constructing a unique-SVP instance from the LWE problem and solving it using BKZ. We examine how large the block dimension b is required to be for BKZ to find the unique solution. Given the matrix LWE instance $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e})$ one builds the lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^{m+kn+1} : (\mathbf{A}| - \mathbf{I}_m| - \mathbf{b})\mathbf{x} = \mathbf{0} \bmod q\}$ of dimension $d = m + kn + 1$, volume q^m, and with a unique-SVP solution $\mathbf{v} = (\mathbf{s}, \mathbf{e}, 1)$ of norm $\lambda \approx \varsigma\sqrt{kn + m}$. Note that the number of used samples m may be chosen between 0 and $(k + 1)n$ in our case and we numerically optimize this choice.

Success condition. We model the behavior of BKZ using the geometric series assumption (which is known to be optimistic from the attacker's point of view), that finds a basis whose Gram-Schmidt norms are given by $\|\mathbf{b}_i^\star\| = \delta^{d-2i-1} \cdot \mathrm{Vol}(\Lambda)^{1/d}$, where $\delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)}$ [24, 4]. The unique short vector $\mathbf{v}$ will be detected if the projection of $\mathbf{v}$ onto the vector space spanned by the last b Gram-Schmidt vectors is shorter

than $\mathbf{b}^\star_{d-b}$. Its projected norm is expected to be $\varsigma\sqrt{b}$, that is the attack is successful if and only if

$$\varsigma\sqrt{b} \leq \delta^{2b-d-1} \cdot q^{m/d}. \tag{2}$$

We note that this analysis introduced in [5] differs and is more conservative than prior works, which were typically based on the hardness of unique-SVP estimates of [36]. The validity of the new analysis has been confirmed by further analysis and experiments in [3].

5.1.3 Dual attack

The dual attack consists of finding a short vector in the dual lattice $\mathbf{w} \in \Lambda' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^{kn} : \mathbf{A}^t\mathbf{x} = \mathbf{y} \bmod q\}$. Assume we have found a vector $(\mathbf{x}, \mathbf{y})$ of length ℓ and compute $z = \mathbf{v}^t \cdot \mathbf{b} = \mathbf{v}^t\mathbf{A}\mathbf{s} + \mathbf{v}^t\mathbf{e} = \mathbf{w}^t\mathbf{s} + \mathbf{v}^t\mathbf{e} \bmod q$, which is distributed as a Gaussian of standard deviation $\ell\varsigma$ if $(\mathbf{A}, \mathbf{b})$ is indeed an LWE sample (otherwise it is uniform mod q). Those two distributions have maximal variation distance bounded by $\epsilon = 4\exp(-2\pi^2\tau^2)$, where $\tau = \ell\varsigma/q$, that is, given such a vector of length ℓ one has an advantage ϵ against decision-LWE.

The length ℓ of a vector given by the BKZ algorithm is given by $\ell = \|\mathbf{b}_0\|$. Knowing that Λ' has dimension $d = m + kn$ and volume q^{kn} we get $\ell = \delta^{d-1}q^{kn/d}$. Therefore, obtaining an ϵ-distinguisher requires running BKZ with block dimension b, where

$$-2\pi^2\tau^2 \geq \ln(\epsilon/4). \tag{3}$$

Note that small advantages ϵ are not relevant since the agreed key is hashed: an attacker needs an advantage of at least $1/2$ to significantly decrease the search space of the agreed key. He must therefore amplify his success probability by building about $1/\epsilon^2$ many such short vectors. Because the sieve algorithms provides $2^{0.2075b}$ vectors, the attack must be repeated at least R times where

$$R = \max(1, 1/(2^{0.2075b}\epsilon^2)).$$

This makes the conservative assumption that all the vectors provided by the Sieve algorithm are as short as the shortest one.

5.1.4 Core-SVP hardness of Kyber

In Table 4 we list the classical and quantum core-SVP-hardness of the three parameter sets of Kyber. The lower bounds of the cost of the primal and dual attack were computed following the approach outlined above using the analysis script `Kyber.py` that is available online at https://github.com/pq-crystals/kyber/tree/master/scripts/.

How conservative is this analysis? The core-SVP-hardness estimates give a lower bound on the cost of actual attacks rather than attempting to assign costs to various building blocks that require further study. To give an idea of how far this lower bound is from recent estimates that attempt to find a tighter bound, consider the example of the "BCNS" key exchange [19]. In [5, Table 6], the NewHope paper computes the classical core-SVP hardness for the parameters used in [19] as 2^{86}. The claimed classical security level for those parameters in [19] is 2^{128}. Note that [5] does not contradict this claim, the factor of $2^{128}/2^{86} = 2^{42}$ rather indicates how conservative the core-SVP hardness estimate is. As a second example, consider another commonly used tool for estimating (classical) security of LWE-based cryptosystems, namely the `lwe-estimator` script by Albrecht [1]. Applied to the Kyber parameter sets it estimates a classical security of 2^{142} for Kyber512, 2^{209} for Kyber768, and 2^{273} for Kyber1024. Note that also these estimates count number of "operations" rather than gates, and are in the RAM model, i.e., ignore the cost of memory access for sieving.

5.1.5 Algebraic attacks.

While the best known attacks against the MLWE instance underlying Kyber do not make use of the structure in the lattice, we still discuss the current state of the art of such attacks. Most noticeably, several recent works propose new quantum algorithms against Ideal-SVP [32, 23, 16, 27, 28], i.e., solving the shortest vector problem in ideal lattices. The work of [28] mentions obstacles towards a quantum attack on Ring-LWE

Table 4: Classical and quantum core-SVP hardness of the MLWE problem (treated as LWE problem) underlying KYBER for different proposed parameter sets. The value b denotes the block dimension of BKZ (i.e., the dimension of the SVP considered in the core-SVP-hardness estimates), and m the number of used samples. Cost is given in $\log_2$ of operations and is the smallest cost for all possible choices of m and b.

	b	m	Core-SVP (classical)	Core-SVP (quantum)
KYBER512				
Primal attack:	390	455	114	103
Dual attack:	385	485	112	102
KYBER768				
Primal attack:	615	695	179	163
Dual attack:	610	690	178	161
KYBER1024				
Primal attack:	845	835	244	221
Dual attack:	825	850	241	218

from their new techniques, but nevertheless suggests using Module-LWE, as it plausibly creates even more obstacles. In [2], Albrecht and Deo establish a reduction from MLWE to RLWE, whose implication is that a polynomial-time algorithm against RLWE with certain parameters would translate to a polynomial-time algorithm against MLWE. In practical terms, however, this attack has a significant slow-down (and this is not just due to the proof) as the dimension of the module increases. This does suggest that increasing the dimension of the module may make the scheme more secure in concrete terms. In particular, going through this reduction to attack KYBER768 would lead to an RLWE problem with quite large modulus and error ($q' = q^3$, $\varsigma' > q^2\varsigma$), and therefore require the attacker to consider more than 1 sample: the underlying lattice remains a module with a rank strictly larger than 2.

5.2 Attacks against symmetric primitives

All symmetric building blocks of KYBER are instantiated with functions derived from Keccak [15]. In the deterministic expansion of $\mathbf{A}$ from ρ we essentially need SHAKE-128 to produce output that "looks uniformly random" and does not create any backdoors in the underlying lattice problem. In the noise generation we require that concatenating a secret and a public input and feeding this concatenation to SHAKE-256 as input results in a secure pseudorandom function. Breaking any of these properties of SHAKE would be a major breakthrough in the cryptanalysis of SHAKE, which would require replacing SHAKE inside KYBER by another XOF.

The security proofs model SHAKE-128, SHA3-256, and SHA3-512 as random oracles, i.e., they are subject to the standard limitations of proofs in the (quantum-)random-oracle model. Turning these limitations into an attack exploiting the instantiation of XOF, H, or G with SHAKE and SHA3 would again constitute a major breakthrough in the understanding of either Keccak or random-oracle proofs in general.

5.3 Attacks exploiting decryption failures

All parameter sets of KYBER have a decapsulation-failure probability δ of somewhat below 2^{-140}; for the examples in the remainder of this subsection we assume the failure probability of 2^{-142} of KYBER768. In Theorems 2 and 3 we see that this failure probability plays a role in the attacker's advantage: in the classical context in the term $4q_{RO}\delta$ and in the quantum context in the term $8q_{RO}^2\delta$, where q_{RO} is the number of queries to the (classical or quantum) random oracle.

Attacks exploiting failures. This term in the attacker's advantage is not merely a proof artifact, it can be explained by the following attack: An attacker searches through many different values of m (see line 1 of Alg. 8) until he finds one that produces random coins r (line 3 of Alg. 8) that lead to a decapsulation failure,

which will give the attacker information about the secret key. In the quantum setting the search through different values of m is accelerated by Grover's algorithm, which explains the square in the term q_{RO}^2. With this attack in mind note that with 2^{64} ciphertexts (cmp. Section 4.A.2 of the Call for Proposals), there is a chance of 2^{-78} of a decapsulation failure without any particular effort by the attacker.

The effect on KYBER**.** The attack sketched in the previous paragraph is based on two assumptions that do not hold for KYBER: First it requires the capability to determine *offline* (e.g., as part of the Grover oracle) if a certain value of r produces a decapsulation failure. Second it assumes that one decapsulation failure seriously threatens the secrecy of the private key. Concerning the first assumption, an attacker cannot determine offline whether a given value of r, or more specifically, the derived values $\mathbf{r}$ (line 9 of Alg. 5) and $\mathbf{e}_1$ (line 13 of Alg. 5), produce a decapsulation failure. The reason is that the probability of decapsulation failures largely depends on the products $\mathbf{s}^T\mathbf{e}_1$ and $\mathbf{e}^T\mathbf{r}$ and the attacker does not now the values of $\mathbf{s}$ and $\mathbf{e}$. A quantum attacker can try to use Grover search to precompute values of m that have a slightly higher chance to produce a failure; as the attacker does not know the signs of the coefficients of $\mathbf{e}_1$ and $\mathbf{s}$, the best strategy is probably to search for values of m that produce $\mathbf{e}_1$ and $\mathbf{r}$ with above-average norm. The gain achieved through such an approach is limited due to the fact that the distribution of a high-dimensional Gaussian is tightly concentrated around its expected value, while that of a 1-dimensional Gaussian is not as tightly concentrated around its mean.

The polynomial pair $(\mathbf{e}_1, \mathbf{r})$ can be seen as a vector in $\mathbb{Z}^{1536}$ distributed as a discrete Gaussian with standard deviation $\sigma = \sqrt{\eta/2} = \sqrt{2}$. By standard tail bounds on discrete Gaussians [9], we know that an m-dimensional vector $\mathbf{v}$ drawn from a discrete Gaussian of standard deviation σ will satisfy

$$\Pr[\|\mathbf{v}\| > \kappa\sigma\sqrt{m}] < \kappa^m \cdot e^{\frac{m}{2}(1-\kappa^2)}, \tag{4}$$

for any $\kappa > 1$.

So for example, the probability of finding a vector which is of length $1.33 \cdot \sigma\sqrt{1536}$ is already as small as 2^{-220}. Even if Grover's algorithm reduces the search space and increases the probability to 2^{-110}, finding such a vector merely increases the chances of getting a decryption error; and the probability increase is governed by the tail-bounds for 1-dimensional Gaussians.[3] For any vector $\mathbf{v}$, if $\mathbf{z}$ is chosen according to a Gaussian with standard deviation σ, then for any κ,

$$\Pr[\langle \mathbf{z}, \mathbf{v}\rangle| > \kappa\sigma\|\mathbf{v}\|] \leq 2e^{-\kappa^2/2}. \tag{5}$$

If originally, the above probability is set so that decryption errors occur with probability $\approx 2^{-140}$, then $\kappa \approx 14$.[4] If the adversary is then able to increase $\|\mathbf{v}\|$ by a factor of 1.33 (by being able to find larger $(\mathbf{e}_1, \mathbf{r})$), then we can decrease κ by a factor of 1.33 to ≈ 10.5 in (5), which would still give us a probability of a decryption error of less than 2^{-80}. However, finding such a large $\mathbf{v}$ would take at least 2^{110} time, which would make the whole attack cost at least 2^{190}.

Of course one can try to find a slightly smaller $\mathbf{v}$ in the first step so that the entire attack takes less time. If Grover's algorithm really saves a square-root factor, then the optimal value is ≈ 1.1 for κ in (4), which would allow us to lower κ by a factor of 1.1 in (5), and would still give a total time to find one decryption error $> 2^{128}$. This makes the attack completely impractical.

Furthermore, a single decapsulation failure in KYBER does not allow an attacker to recover much information about the secret key $\mathbf{s}$. To get an intuition for the amount of information obtained from failures, consider the attack described in [34]. This is the standard attack that exploits failures in RLWE key encapsulation schemes that reuse keys without the CCA transform. In this scenario the attacker can *adaptively choose arbitrary noise*, i.e., set failure probabilities to arbitrary values and maximize the information obtained from each failure or non-failure. The paper concludes that attacking RLWE key exchange in lattice dimension 1024 in this setting *"can be done with perhaps* $4,000$ *queries"*. It seems extremely unlikely that even 10 decapsulation failures in KYBER would allow an attacker to recover any meaningful information about the secret key $\mathbf{s}$. Note that the probability of f decapsulation failures in 2^{64} ciphertexts is about $2^{-(e-64)f}$ where

[3] The decryption noise is generated as an inner product of two vectors, and the distribution of this inner product closely resembles the Gaussian distribution.

[4] The above formula only roughly approximates how the decryption error is calculated where $\mathbf{z}$ corresponds to the secret key $(\mathbf{s}, \mathbf{e})$. We should also point out that a part of the decryption error in KYBER is caused by the rounding function Compress, which the adversary has no control over. Therefore this attack will be even less practical than what we describe.

2^{-e} is the largest probability of failure an attacker can achieve for one ciphertext. We have established that even with a Grover search making 2^{110} calls to the hash function, an attacker can only get $e > 80$. This very loose analysis shows that an attacker can't reasonably hope to produce more than two or three failures in less than 2^{128} time. We therefore conclude that the decryption failures do not introduce any weaknesses into KYBER.

Multitarget attacks using failures. Despite the limited gain, an attacker could consider using Grover's algorithm to precompute values of m that produce $\mathbf{r}$ and $\mathbf{e}_1$ with large norm and then use this precomputed set of values of m against many users. This multi-target attack is prevented by hashing the public key pk into the random coins r and thereby into $\mathbf{r}$ and $\mathbf{e}_1$ (line 3 of Alg. 8).

6 Advantages and limitations

6.1 Advantages

In addition to the very competitive speeds, small parameters, and being based on a well-studied problem, the unique advantages of KYBER are:

Ease of implementation: Optimized implementations only have to focus on a fast dimension-256 NTT and a fast Keccak permutation. This will give very competitive performance *for all parameter sets of* KYBER.

Scalability: Switching from one KYBER parameter set to another only requires changing the matrix dimension (i.e., a `#define` in most C implementations) and the noise sampling.

We will now give a brief comparison of KYBER to other types of post-quantum schemes (that we are aware of) and, more importantly, to other manners in which lattice-based schemes could be instantiated.

6.2 Comparison to SIDH

An interesting alternative to lattice-based KEMs is supersingular-isogeny Diffie-Hellman (SIDH) [44]. The obvious advantage of SIDH is the sizes of public keys and ciphertexts that—with suitable compression [26]—are about a factor of 3 smaller than KYBER's public keys and ciphertexts. The downside of SIDH is that it is more than 2 orders of magnitude slower than KYBER. The scheme is also rather new, which makes it hard to make definitive comparisons. In the coming years, both implementation speeds and (quantum) attacks against SIDH can improve which may result in faster schemes and/or larger parameters.

6.3 Comparison to code-based KEMs

When considering code-based KEMs, one needs to distinguish the "classical" McEliece and Niederreiter schemes based on binary Goppa codes, and schemes with a less conservative (but more efficient) choice of code. A KEM based on binary Goppa codes can reasonably claim to be a very conservative choice of post-quantum primitive; however, its deployment will, in many scenarios, be hampered by massive public-key size and key-generation time. Less conservative choices, like quasi-cyclic medium-density parity-check (QC-MDPC) codes, are a closer competition in terms of performance but suffer from the fact that for efficient parameters at high security levels they do not achieve (provably) negligible failure probability, which precludes their use in CCA-secure KEMs.

6.4 Comparison to other lattice-based schemes

There are certain design choices that one can make when designing lattice-based schemes, some of which can have significant effects on the efficiency of the resulting scheme and on the underlying security assumption. Below we list the most important ones and explain the advantages / disadvantages of them versus what we chose for KYBER.

6.4.1 Schemes that build a KEM directly

The KYBER KEM is constructed by encrypting a random message using the LPR encryption [54] (with "bit-dropping"). Another approach one could take is directly building a KEM using the slightly different ideas described in [30, 65]. The advantage of the constructions in [30, 65] over our approach is that if one were to construct a CPA-secure KEM transmitting a b-bit key, then the ciphertext would be b bits shorter, which is about a 3% saving for typical parameters [52]. If, however, one wishes to construct a CCA-secure KEM like KYBER, then this advantage disappears since transformations from CPA-secure KEMs to CCA-secure ones implicitly go through a CPA-secure encryption scheme, which will result in adding b bits to the KEM. This is why, in KYBER, we simply use the LPR encryption scheme (instead of the CPA-secure key encapsulation) to define KYBER.CPAPKE, and then use this as a building block to construct the IND-CCA2-secure KEM KYBER.CCAKEM. Since there is virtually no difference between the two approaches, we will not draw a distinction between schemes constructed in either manner throughout the rest of this section.

6.4.2 LWE based schemes

If one does not want to use any algebraic structure in the LWE problem (i.e. if one takes the MLWE problem over the ring $\mathbb{Z}$), then there are two possibilities for constructing encryption or key-exchange schemes. The first approach makes the public key and the secret key very large (on the order of Megabytes), while keeping the ciphertext at essentially the same size as in KYBER. This type of scheme is the [67] version of the original Regev scheme from [71]. Because of the very large public-key size, this scheme would be extremely inefficient as a key exchange. A scheme more amenable to key exchange is [17], whose public key and ciphertext sizes are both approximately 11 KB each, which is approximately 10 times larger than in KYBER. The running time of each party is also larger by a factor of at least 10. In short, LWE-based schemes do not have any ring structure but are an order of magnitude slower and larger than KYBER. They are good back-up schemes in case algebraic structure in lattice schemes could somehow be devastatingly exploited by attackers.

6.4.3 Ring-LWE based schemes

The other extreme in the LWE design space are Ring-LWE (RLWE) schemes based on [54] (e.g., [5]). RLWE is a special case of the MLWE problem where the width of the matrix $\mathbf{A}$ over the ring R is always 1 (and typically, its height would be 2 for a PKE or KEM scheme). Varying the hardness of an RLWE scheme therefore requires to change the dimension of the ring, whereas in KYBER, the ring is always the same and the dimension of the module is being varied. As we mentioned above, one advantage of the approach we chose for KYBER is that we only need to have one good implementation for operations over the ring; varying the dimension of the module simply involves doing more (or fewer) of the same ring operations. Changing the ring, on the other hand, would require completely re-implementing all the operations.

Another advantage of working with a constant-degree "small" ring is that it enables more fine-grained tradeoffs between performance and security. The simplest and most efficient way of implementing RLWE is to work over rings $\mathbb{Z}[X]/(X^n + 1)$ where n is a power of 2. Since n is the only parameter that determines the efficiency and security of RLWE schemes, limiting it to powers of 2 may require overshooting the needed security bound. For example, the dimension of KYBER768 is not reachable. One could of course work directly modulo a polynomial of any desired degree (with the main restriction being that it has to be irreducible over $\mathbb{Z}$), but then the security would decrease slightly due the geometry of non-power-of-2 number fields and the operations over the ring would be somewhat less trivial to implement (see [57]).

The one advantage of RLWE over KYBER is that if $\mathbf{A}$ is a $k \times k$ matrix, then extracting it from a seed requires k times more XOF output than for a 1×1 matrix.

6.4.4 NTRU

When compared to KYBER, NTRU [41] has all the advantages and disadvantages of RLWE, but in addition has two further negative points against it. First NTRU key generation is considerably more expensive than in RLWE when the ring does not support NTT. The reason is that NTRU key generation requires polynomial division, whereas RLWE key generation requires only multiplication (if the ring supports NTT, then division is not much slower than multiplication). The second possible downside of NTRU is that the

geometry of its underlying lattice leads to attacks that do not exist against RLWE or MLWE schemes [46]. While this property does not seem to aid in attacks against the small parameters that are used for defining NTRU cryptosystems, it may point to a possible weakness that could be further exploited. The one possible advantage of using NTRU is a small performance advantage during encryption (encapsulation), but given the disadvantages we do not consider this a good tradeoff. Furthermore, it is not possible to define an efficient version of "Module-NTRU" that would allow for the advantages of KYBER described above in Section 6.1.

6.4.5 Different Polynomial Rings

One could consider using KYBER with a ring that is not $\mathbb{Z}[X]/(X^n + 1)$. An argument that could be made for using different rings is that the rings currently used in KYBER have algebraic properties (e.g., subrings, large Galois groups, etc.) which may be exploited in attacks. We choose to work with $\mathbb{Z}[X]/(X^n + 1)$ for the following reasons:

- From a performance perspective there is no serious competition; the NTT-based multiplication supported by the parameters we chose for KYBER is at the same time very memory efficient and faster than any other algorithm for multiplication in polynomial rings.
- Lattice-based schemes using the ring $\mathbb{Z}[X]/(X^n + 1)$ have been studied since at least [53]. When the noise vectors are chosen as specified in [54], there have been no improved attacks against RLWE (or MLWE) that use the underlying algebraic structure [66]. Furthermore, being based on MLWE, the algebraic structure of KYBER is very different from that which was exploited in the attacks against *ideal* lattices in [16, 27, 28][5] – we emphasize that the lattice problems underlying the hardness of KYBER are *not* ideal lattices.
- Some of the additional algebraic structure of $\mathbb{Z}[X]/(X^n + 1)$ is actually *helpful against* certain possible attack vectors. As a simple example, it can be proved that when $X^n + 1$ fully splits modulo q, there do not exist polynomials in the ring that have small norm and many zeros in the NTT representation—the existence of such polynomials for *any* q would weaken the security of MLWE.
- Finally, $\mathbb{Z}[X]/(X^n + 1)$ is one of the most widely studied, and best understood, rings (along with other cyclotomic rings) in algebraic number theory. The fact that no attacks have been found against its use for cryptosystems like KYBER makes it a much more conservative choice than some ring that is harder to analyze and may show weaknesses only after many more years of study.

6.4.6 Deterministic Noise.

Instead of adding noise $\mathbf{e}, \mathbf{e}_1$, and e_2, one can add "deterministic" noise by simply dropping bits. This is the basis behind the "Learning with Rounding" (LWR) problem [10], which for certain parameters is as hard as the LWE problem. We believe that asymptotically this is a sound approach but the number of bits that can be dropped before significant decryption error is introduced is not very large (≈ 2) in certain places in the scheme. This may allow for a possibility of slightly improved attacks against the scheme. Since generating noise is not a particularly costly operation, we did not choose to potentially weaken the scheme to save a little time.

We point out that there is still bit-dropping in KYBER at exactly the same places that one would drop bits to create deterministic noise, but we only do this for reducing the output size. If one believes that deterministic noise adds some security (which we do), then KYBER also has the added security caused by the deterministic noise. We also point out that it very easy to create a version of KYBER that relies entirely on deterministic noise for security – one can simply remove the errors $\mathbf{e}, \mathbf{e}_1, e_2$ from the scheme description, while keeping the bit-dropping (and possibly increasing the number of dropped bits due to the fact that no noise was added).

[5] Also, like the attacks against NTRU, these do not apply for the small parameters used public key encryption schemes.

References

[1] Martin Albrecht. Security estimates for the learning with errors problem, 2017. Version 2017-09-27, https://bitbucket.org/malb/lwe-estimator. 21

[2] Martin Albrecht and Amit Deo. Large modulus Ring-LWE ≥ Module-LWE, 2017. To appear. https://eprint.iacr.org/2017/612. 22

[3] Martin R Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, volume 10211 of *LNCS*, pages 65–102. Springer, 2017. https://eprint.iacr.org/2017/815. 21

[4] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015. https://eprint.iacr.org/2015/046. 19, 20

[5] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*, pages 327–343. USENIX Association, 2016. http://cryptojedi.org/papers/#newhope. 6, 10, 11, 12, 13, 19, 20, 21, 25

[6] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *LNCS*, pages 57–74. Springer, 2013. https://eprint.iacr.org/2013/098. 18

[7] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi. Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9665 of *LNCS*, pages 789–819. Springer, 2016. https://eprint.iacr.org/2016/146. 20

[8] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzingeri, and Jiří Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *LNCS*, pages 403–415. Springer, 2011. https://www.cs.duke.edu/~rongge/LPSN.pdf. 19

[9] Wojciech Banaszczyk. New bounds in some transference theorems in the geometry of numbers. *Mathematische Annalen*, 296(1):625–635, 1993. 23

[10] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, 2012. http://www.iacr.org/archive/eurocrypt2012/72370713/72370713.pdf. 6, 12, 18, 26

[11] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag Berlin Heidelberg, 1987. https://link.springer.com/chapter/10.1007/3-540-47721-7_24. 18

[12] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA '16 Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*, pages 10–24. SIAM, 2016. https://eprint.iacr.org/2015/1128. 20

[13] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, 2012. http://cryptojedi.org/papers/#coolnacl. 13

[14] Daniel J. Bernstein, Peter Schwabe, and Gilles Van Assche. Tweetable FIPS 202, 2015. https://keccak.team/2015/tweetfips202.html (accessed 2017-11-29). 12

[15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference. Submission to the NIST SHA-3 competition, 2011. https://keccak.team/files/Keccak-reference-3.0.pdf. 12, 22

[16] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *SODA '16 Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*, pages 893–902. SIAM, 2016. http://fangsong.info/files/pubs/BS_SODA16.pdf. 21, 26

[17] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018. ACM, 2016. https://eprint.iacr.org/2016/659. 10, 11, 25

[18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. IEEE, 2018. To appear. https://eprint.iacr.org/2017/634. 3, 10, 19

[19] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, 2015. https://eprint.iacr.org/2014/599. 11, 21

[20] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS '12 Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012. https://eprint.iacr.org/2011/277. 16

[21] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *STOC '13 Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584. ACM, 2013. http://arxiv.org/pdf/1306.0281. 11

[22] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – a cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, 2016. https://eprint.iacr.org/2016/300. 11

[23] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014. https://docbox.etsi.org/workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf. 21

[24] Yuanmi Chen. *Lattice reduction and concrete security of fully homomorphic encryption*. PhD thesis, l'Université Paris Diderot, 2013. http://www.di.ens.fr/~ychen/research/these.pdf. 20

[25] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, 2011. http://www.iacr.org/archive/asiacrypt2011/70730001/70730001.pdf. 19, 20

[26] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of SIDH public keys. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 679–706. Springer, 2017. https://eprint.iacr.org/2016/963. 24

[27] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9666 of *LNCS*, pages 559–585. Springer, 2016. https://eprint.iacr.org/2015/313. 21, 26

[28] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to Ideal-SVP. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 324–348. Springer, 2017. https://eprint.iacr.org/2016/885. 21, 26

[29] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2017. 20

[30] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. IACR Cryptology ePrint Archive report 2012/688, 2012. https://eprint.iacr.org/2012/688. 25

[31] Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. IACR Cryptology ePrint Archive report 2017/999, 2017. https://eprint.iacr.org/2017/999. 20

[32] Kirsten Eisenträger, Sean Hallgren, Alexei Kitaev, and Fang Song. A quantum algorithm for computing the unit group of an arbitrary degree number field. In *STOC '14 Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 293–302. ACM, 2014. http://www.personal.psu.edu/kxe8/unitgroup.pdf. 21

[33] Thomas Espitau, Pierre-Alain Fouque, Benoït Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874. ACM, 2017. https://eprint.iacr.org/2017/505. 11

[34] Scott Fluhrer. Cryptanalysis of ring-LWE based key exchange with key share reuse. IACR Cryptology ePrint Archive report 2016/085, 2016. https://eprint.iacr.org/2016/085. 23

[35] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology - CRYPTO '99*, pages 537–554, 1999. https://link.springer.com/chapter/10.1007/3-540-48405-1_34. 3, 9, 16

[36] Nicolas Gama and Phong Nguyen. Predicting lattice reduction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, 2008. https://www.iacr.org/archive/eurocrypt2008/49650031/49650031.pdf. 21

[37] Nicolas Gama, Phong Q Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. http://www.iacr.org/archive/eurocrypt2010/66320257/66320257.pdf. 20

[38] Matthew Gretton-Dann. Introducing 2017's extensions to the Arm architecture, 2017. https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture. 13

[39] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *LNCS*, pages 67–82. Springer, 2013. http://cryptojedi.org/papers/#lattisigns. 13

[40] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Terminating BKZ. IACR Cryptology ePrint Archive report 2011/198, 2011. https://eprint.iacr.org/2011/198. 20

[41] Jeffrey Hoffstein, Jull Pipher, and Joseph H. Silverman. NTRU: a ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic number theory*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998. https://www.securityinnovation.com/uploads/Crypto/ANTS97.ps.gz. 6, 25

[42] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, LNCS, pages 341–371. Springer, 2017. https://eprint.iacr.org/2017/604. 13, 16, 17

[43] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169. Springer, 2007. http://www.iacr.org/archive/crypto2007/46220150/46220150.pdf. 12

[44] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography – PQCrypto 2011*, volume 7071 of *LNCS*, pages 19–34. Springer, 2011. https://eprint.iacr.org/2011/506. 24

[45] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 43–62. Springer, 2015. http://www.iacr.org/archive/crypto2015/92160264/92160264.pdf. 19

[46] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 3–26. Springer, 2017. https://www.di.ens.fr/~fouque/euro17a.pdf. 26

[47] Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015. http://www.thijs.com/docs/phd-final.pdf. 20

[48] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosiario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9216 of *LNCS*, pages 3–22. Springer, 2015. http://www.iacr.org/archive/crypto2015/92160123/92160123.pdf. 20

[49] Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, 2015. https://eprint.iacr.org/2014/907. 20

[50] Adam Langley. Maybe skip SHA-3. Blog post on ImperialViolet, 2017. https://www.imperialviolet.org/2017/05/31/skipsha3.html. 13

[51] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015. https://eprint.iacr.org/2012/090. 3, 10, 16

[52] Vadim Lyubashevsky. Standardizing lattice crypto and beyond. Slides of the talk given by Vadim Lyubashevsky at PQCrypto 2017, 2017. https://2017.pqcrypto.org/conference/slides/pqc_2017_lattice.pdf. 25

[53] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *Fast Software Encryption – FSE 2008,*, volume 5086 of *LNCS*, pages 54–72. Springer, 2008. https://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifft.pdf. 11, 26

[54] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010. http://www.iacr.org/archive/eurocrypt2010/66320288/66320288.pdf. 6, 10, 25, 26

[55] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Slides of the talk given by Chris Peikert at Eurocrypt 2010, 2010. http://crypto.rd.francetelecom.com/events/eurocrypt2010/talks/slides-ideal-lwe.pdf. 6

[56] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, 2013. http://www.cims.nyu.edu/~regev/papers/ideal-lwe.pdf. 6

[57] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for Ring-LWE cryptography. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, 2013. http://www.iacr.org/archive/eurocrypt2013/78810035/78810035.pdf. 25

[58] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA '10 Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1468–1480. SIAM, 2010. https://cseweb.ucsd.edu/~daniele/papers/Sieve.pdf. 20

[59] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf. 18

[60] National Institute of Standards and Technology. FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 10, 12

[61] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008. ftp://ftp.di.ens.fr/pub/users/pnguyen/JoMC08.pdf. 20

[62] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked Ring-LWE implementation. IACR Cryptology ePrint Archive report 2016/1109, 2016. https://eprint.iacr.org/2016/1109. 18

[63] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem, 2009. https://web.eecs.umich.edu/~cpeikert/pubs/svpcrypto.pdf (full version of [64]). 6, 10, 31

[64] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In *STOC '09 Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2009. See also full version [63]. 31

[65] Chris Peikert. Lattice cryptography for the Internet. In Michele Mosca, editor, *Post-Quantum Cryptography*, volume 8772 of *LNCS*, pages 197–219. Springer, 2014. http://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf. 25

[66] Chris Peikert. How (not) to instantiate Ring-LWE. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, volume 9841 of *LNCS*, pages 411–430. Springer, 2016. https://web.eecs.umich.edu/~cpeikert/pubs/instantiate-rlwe.pdf. 26

[67] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David A. Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, 2008. https://www.iacr.org/archive/crypto2008/51570556/51570556.pdf. 25

[68] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be – attacking strongSwan's implementation of post-quantum signatures. In *CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1843–1855. ACM, 2017. https://eprint.iacr.org/2017/490. 11

[69] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013. https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf. 6, 10, 11

[70] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, 2017. https://eprint.iacr.org/2017/594. 18

[71] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC '05 Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 84–93. ACM, 2005. Preliminary version of [72]. 6, 11, 25

[72] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34, 2009. http://www.cims.nyu.edu/~regev/papers/qcrypto.pdf. 6, 32

[73] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, 2014. http://www.iacr.org/archive/ches2014/87310183/87310183.pdf. 11

[74] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. IACR Cryptology ePrint Archive report 2017/1005, 2017. https://eprint.iacr.org/2017/1005. 16, 17

[75] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994. http://www.csie.nuk.edu.tw/~cychen/Lattices/Lattice%20Basis%20Reduction_%20Improved%20Practical%20Algorithms%20and%20Solving%20Subset%20Sum%20Problems.pdf. 19

DAGS: Key Encapsulation from Dyadic GS Codes

Gustavo Banegas[1], Paulo S. L. M. Barreto[2], Brice Odilon Boidje[3], Pierre-Louis Cayrel[4], Gilbert Ndollane Dione[3], Kris Gaj[7], Cheikh Thiécoumba Gueye[3], Richard Haeussler[7], Jean Belo Klamti[3], Ousmane N'diaye[3], Duc Tri Nguyen[7], Edoardo Persichetti[5], and Jefferson E. Ricardini[6]

[1]Technische Universiteit Eindhoven, The Netherlands
[2]University of Washington Tacoma, USA
[3]Université Cheikh Anta Diop, Dakar, Senegal.
[4]Laboratoire Hubert Curien, Saint-Etienne, France
[5]Florida Atlantic University, USA
[6]Universidade de São Paulo, Brazil
[7]George Mason University, USA

The team listed above is the principal submitter; there are no auxiliary submitters.

Owner, inventors and developers of this submission are the same as the principal submitter. Relevant prior work is credited where appropriate.

Email Address (preferred): epersichetti@fau.edu

Postal Address and Telephone (if absolutely necessary):
Edoardo Persichetti, Department of Mathematical Sciences, 777 Glades Rd, Boca Raton, FL, 33431, +1 561 297 4136.

Signature: ×. See also printed version of "Statement by Each Submitter".

Contents

1 Introduction

Code-based cryptography is one of the main candidates for post-quantum cryptography standardization. The area is largely based on the Syndrome Decoding Problem [8], which shows to be strong against quantum attacks. Over the years, since McEliece's seminal work [27], many cryptosystems have been proposed, trying to balance security and efficiency. In particular dealing with inherent flaws such as the large size of the public keys. In fact, while McEliece's cryptosystem, which is based binary Goppa codes, is still unbroken, it features a key of several kilobytes, which has effectively prevented its use in many applications.

There are currently two main trends to deal with this issue, and they both involve structured matrices: the first, is based on "traditional" algebraic codes such as Goppa or Srivastava codes; the second suggests to use sparse matrices as in LDPC/MDPC codes. This work builds on the former approach, initiated in 2009 by Berger et al. [7], who proposed Quasi-Cyclic (QC) codes, and Misoczki and Barreto [28], suggesting Quasi-Dyadic (QD) codes instead (later generalized to Quasi-Monoidic (QM) codes [6]). Both proposals feature very compact public keys due to the introduction of the extra algebraic structure, but unfortunately this also leads to a vulnerability. Indeed, Faugère, Otmani, Perret and Tillich [17] devised a clever attack (known simply as FOPT) which exploits the algebraic structure to build a system of equations, which can successively be solved using Gröbner bases techniques. As a result, the QC proposal is definitely compromised, while the QD/QM approach needs to be treated with caution. In fact, for a proper choice of parameters, it is still possible to design secure schemes using for instance binary Goppa codes, or Generalized Srivastava (GS) codes as suggested by Persichetti in [32].

In this document, we present DAGS, a Key Encapsulation Mechanism (KEM) that follows the QD approach using GS codes. To the best of our knowledge, this is the first code-based KEM that uses structured algebraic codes. The KEM achieves IND-CCA security following the recent framework by Kiltz et al. [23], and features compact public keys and efficient encapsulation and decapsulation algorithms. We modulate our parameters to achieve the most efficient scheme, while at the same time avoiding the FOPT attack.

2 Notation

This section describes the notation used in this document.

a	a constant
$\mathbf{a}$	a vector
A	a matrix
$\mathcal{A}$	an algorithm or hash function
A	a set
Diag($\mathbf{a}$)	the diagonal matrix formed by the vector $\mathbf{a}$
I_n	the $n \times n$ identity matrix
$\xleftarrow{\$}$	choosing a random element from a set or distribution

2.1 Formats and Conventions

DAGS operates on vectors of elements of the finite field $\mathbb{F}_q$, where q is a power of 2 as specified by the choice of parameters.

1. Finite field elements are represented as bit strings using standard log/antilog tables (see for instance [26, Ch. 4, §5]) which are stored in the memory.

2. Field operations are performed using the log/antilog tables, and implemented in an isochronous way.

3. Every vector or matrix defined over an extension field $\mathbb{F}_{q^m}$ can be *projected* onto the base field $\mathbb{F}_q$ by replacing each element with the (column) vector formed by the coefficients of its representation over $\mathbb{F}_q$.

4. We use the hash function SHA3-512 with 256 bits input for the random oracles $\mathcal{G}, \mathcal{H}$ and $\mathcal{K}$ (see Section 3.1).

3 Full Protocol Specification (2.B.1)

3.1 Design Rationale

In this section we introduce the three algorithms that form DAGS. System parameters are the code length n and dimension k, the values s and t which define a GS code, the cardinality of the base field q and the degree of the field extension m. In addition, we have $k = k' + k''$, where k' is set to be "small". In practice, k' is such that a vector of length k' can be efficiently stored in 256 bits, depending on the base

field. This makes the hash functions (see below) easy to compute, and minimizes the overhead due to the IND-CCA2 security in the QROM.

The key generation process uses the following fundamental equation

$$\frac{1}{h_{i\oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}. \tag{1}$$

to build the vector $\mathbf{h} = (h_0, \ldots, h_{n-1})$ of elements of $\mathbb{F}_{q^m}$, which is known as *signature* of a dyadic matrix. This is then used to form a Cauchy matrix, i.e. a matrix $C(\mathbf{u}, \mathbf{v})$ with components $C_{ij} = \frac{1}{u_i - v_j}$. The matrix is then successively powered (element by element) forming several blocks which are superimposed and then multiplied by a random diagonal matrix. Finally, the resulting matrix is projected onto the base field and row-reduced to systematic form. The overall process is described below.

3.1.1 Key Generation

1. Generate dyadic signature $\mathbf{h}$. To do this:
   i. Choose random non-zero distinct h_0 and h_j for $j = 2^l, l = 0, \ldots, \lfloor \log q^m \rfloor$.
   ii. Form the remaining elements using (1).
   iii. Return a selection[1] of blocks of dimension s up to length n.
2. Build the Cauchy support. To do this:
   i. Choose a random[2] offset $\omega \xleftarrow{\$} \mathbb{F}_{q^m}$.
   ii. Set $u_i = 1/h_i + \omega$ and $v_j = 1/h_j + 1/h_0 + \omega$ for $i = 0, \ldots, s-1$ and $j = 0, \ldots, n-1$.
   iii. Set $\mathbf{u} = (u_0, \ldots, u_{s-1})$ and $\mathbf{v} = (v_0, \ldots, v_{n-1})$.
3. Form Cauchy matrix $\hat{H}_1 = C(\mathbf{u}, \mathbf{v})$.
4. Build blocks $\hat{H}_i$, $i = 2, \ldots t$, by raising each element of $\hat{H}_1$ to the power of i.
5. Superimpose blocks to form matrix $\hat{H}$.
6. Choose random elements $z_i \xleftarrow{\$} \mathbb{F}_{q^m}$ such that $z_{is+j} = z_{is}$ for $i = 0, \ldots, n_0 - 1$, $j = 0, \ldots, s-1$.

[1] Making sure to exclude any block containing an undefined entry.
[2] See Appendix A for restrictions about the choice of the offset.

7. Form $H = \hat{H} \cdot \mathrm{Diag}(\mathbf{z})$.
8. Transform H into alternant form[3]: call this H'.
9. Project H onto $\mathbb{F}_q$ using the co-trace function: call this H_{base}.
10. Write H_{base} in systematic form $(M \mid I_{n-k})$.
11. The public key is the generator matrix $G = (I_k \mid M^T)$.
12. The private key is the alternant matrix H'.

The encapsulation and decapsulation algorithms make use of two hash functions[4] $\mathcal{G} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^k$ and $\mathcal{H} : \mathbb{F}_q^{k'} \to \mathbb{F}_q^{k'}$, the former with the task of generating randomness for the scheme, the latter to provide "plaintext confirmation" as in [23]. The shared symmetric key is obtained via another hash function $\mathcal{K} : \{0,1\}^* \to \{0,1\}^\ell$, where ℓ is the desired key length.

3.1.2 Encapsulation

1. Choose $\mathbf{m} \xleftarrow{\$} \mathbb{F}_q^{k'}$.
2. Compute $\mathbf{r} = \mathcal{G}(\mathbf{m})$ and $\mathbf{d} = \mathcal{H}(\mathbf{m})$.
3. Parse $\mathbf{r}$ as $(\boldsymbol{\rho} \parallel \boldsymbol{\sigma})$ then set $\boldsymbol{\mu} = (\boldsymbol{\rho} \parallel \mathbf{m})$.
4. Generate error vector $\mathbf{e}$ of length n and weight w from $\boldsymbol{\sigma}$.
5. Compute $\mathbf{c} = \boldsymbol{\mu} G + \mathbf{e}$.
6. Compute $\mathbf{k} = \mathcal{K}(\mathbf{m})$.
7. Output ciphertext $(\mathbf{c}, \mathbf{d})$; the encapsulated key is $\mathbf{k}$.

The decapsulation algorithm consists mainly of decoding the noisy codeword received as part of the ciphertext. This is done using the alternant decoding algorithm described in [26, Ch. 12, §9] and requires the parity-check matrix to be in alternant form (hence the nature of the private key).

[3] See §2 and §6 of [26, Ch. 12].
[4] As specified in Section 2.1

3.1.3 Decapsulation

1. Input private key, i.e. parity-check matrix H' in alternant form.
2. Use H' to decode $\mathbf{c}$ and obtain codeword $\boldsymbol{\mu}'G$ and error $\mathbf{e}'$.
3. Output $\perp$ if decoding fails or $\mathsf{wt}(\mathbf{e}') \neq w$
4. Recover $\boldsymbol{\mu}'$ and parse it as $(\boldsymbol{\rho}' \parallel \mathbf{m}')$.
5. Compute $\mathbf{r}' = \mathcal{G}(\mathbf{m}')$ and $\mathbf{d}' = \mathcal{H}(\mathbf{m}')$.
6. Parse $\mathbf{r}'$ as $(\boldsymbol{\rho}'' \parallel \boldsymbol{\sigma}')$.
7. Generate error vector $\mathbf{e}''$ of length n and weight w from $\boldsymbol{\sigma}'$.
8. If $\mathbf{e}' \neq \mathbf{e}'' \vee \rho' \neq \rho'' \vee \mathbf{d} \neq \mathbf{d}'$ output $\perp$.
9. Else compute $\mathbf{k} = \mathcal{K}(\mathbf{m}')$.
10. The decapsulated key is $\mathbf{k}$.

DAGS is built upon the McEliece encryption framework, with a notable exception. In fact, we incorporate the "randomized" version of McEliece by Nojima et al. [31] into our scheme. This is extremely beneficial for two distinct aspects: first of all, it allows us to use a much shorter vector $\mathbf{m}$ to generate the remaining components of the scheme, greatly improving efficiency. Secondly, it allows us to get tighter security bounds. Note that our protocol differs slightly from the paradigm presented in [23], in the fact that we don't perform a full re-encryption in the decapsulation algorithm. Instead, we simply re-generate the randomness and compare it with the one obtained after decoding. This is possible since, unlike a generic PKE, McEliece decryption reveals the randomness used, in our case $\mathbf{e}$ (and $\boldsymbol{\rho}$). It is clear that if the re-generated randomness is equal to the retrieved one, the resulting encryption will also be equal. This allows us to further decrease computation time.

4 Security (2.B.4)

In this section, we discuss some aspects of provable security, and in particular we show that DAGS satisfies the notion of IND-CCA security for KEMs. Before discussing the IND-CCA security of DAGS, we show that the underlying PKE (i.e. Randomized McEliece) satisfies the γ-spread property. This will allow us to get better security bounds in our reduction.

Definition 1 *Consider a probabilistic PKE with randomness set* R*. We say that PKE is* γ-spread *if for a given key pair* (sk, pk)*, a plaintext* m *and an element* y *in the ciphertext domain, we have*

$$\mathsf{Pr}[r \xleftarrow{\$} \mathsf{R} \mid y = \mathsf{Enc}_{pk}(m, r)] \leq 2^{-\gamma},$$

for a certain $\gamma \in \mathbb{R}$*.*

The definition above is presented as in [23], but note that in fact this corresponds to the notion of γ-*uniformity* given by Fujisaki and Okamoto in [20], except for a change of constants. In other words, a scheme is γ-spread if it is $2^{-\gamma}$-uniform.

It was proved in [14] that a simple variant of the (classic) McEliece PKE is γ-uniform for $\gamma = 2^{-k}$, where k is the code dimension as usual (more in general, $\gamma = q^{-k}$ for a cryptosystem defined over $\mathbb{F}_q$). We can extend this result to our scheme as follows.

Lemma 1 *Randomized McEliece is* γ*-uniform for* $\gamma = \dfrac{q^{-k''}}{\binom{n}{w}}$.

Proof Let $\mathbf{y}$ be a generic vector of $\mathbb{F}_q^n$. Then either $\mathbf{y}$ is a word at distance w from the code, or it isn't. If it isn't, the probability of $\mathbf{y}$ being a valid ciphertext is clearly exactly 0. On the other hand, suppose $\mathbf{y}$ is at distance w from the code; then there is only one choice of $\boldsymbol{\rho}$ and one choice of $\mathbf{e}$ that satisfy the equation (since w is below the GV bound), i.e. the probability of $\mathbf{y}$ being a valid ciphertext is exactly $1/q^{k''} \cdot 1/\binom{n}{w}$, which concludes the proof. □

We are now ready to present the security results.

Theorem 1 *Let* $\mathcal{A}$ *be an IND-CCA adversary against DAGS that makes at most* $q_{RO} = q_{\mathcal{G}} + q_{\mathcal{K}}$ *total random oracle queries*[5] *and* q_D *decryption queries. Then there exists an IND-CPA adversary* $\mathcal{B}$ *against PKE, running in approximately the same time as* $\mathcal{A}$*, such that*

$$\mathsf{Adv}_{KEM}^{IND-CCA}(\mathcal{A}) \leq q_{RO} \cdot 2^{-\gamma} + 3 \cdot \mathsf{Adv}_{PKE}^{IND-CPA}(\mathcal{B}).$$

[5]Respectively $q_{\mathcal{G}}$ queries to the random oracle $\mathcal{G}$ and $q_{\mathcal{K}}$ queries to the random oracle $\mathcal{K}$.

Proof The thesis is a consequence of the results presented in Section 3.3 of [23]. In fact, our scheme follows the $\mathsf{KEM}_m^{\perp}$ framework that consists of applying two generic transformations to a public-key encryption scheme. The first step consists of transforming the IND-CPA encryption scheme into a OW-PCVA (i.e. Plaintext and Validity Checking) scheme. Then, the resulting scheme is transformed into a KEM in a "standard" way. Both proofs are obtained via a sequence of games, and the combination of them shows that breaking IND-CCA security of the KEM would lead to break the IND-CPA security of the underlying encryption scheme. Note that Randomized McEliece, instantiated with Quasi-Dyadic GS codes, presents no correctness error (the value δ in [23]), which greatly simplifies the resulting bound. □

The value $\mathbf{d}$ included in the KEM ciphertext does not contribute to the security result above, but it is a crucial factor to provide security in the Quantum Random Oracle Model (QROM). We present this in the next theorem.

Theorem 2 *Let $\mathcal{A}$ be a quantum IND-CCA adversary against DAGS that makes at most $q_{RO} = q_{\mathcal{G}} + q_{\mathcal{K}}$ total quantum random oracle queries*[6] *and q_D (classical) decryption queries. Then there exists a OW-CPA adversary $\mathcal{B}$ against PKE, running in approximately the same time as $\mathcal{A}$, such that*

$$\mathsf{Adv}_{KEM}^{IND-CCA}(\mathcal{A}) \leq 8q_{RO} \cdot \sqrt{q_{RO} \cdot \sqrt{\mathsf{Adv}_{PKE}^{OW-CPA}(\mathcal{B})}}.$$

Proof The thesis is a consequence of the results presented in Section 4.4 of [23]. In fact, our scheme follows the $\mathsf{QKEM}_m^{\perp}$ framework that consists of applying two generic transformations to a public-key encryption scheme. The first step transforming the IND-CPA encryption scheme into a OW-PCVA (i.e. Plaintext and Validity Checking) scheme, is the same as in the previous case. Now, the resulting scheme is transformed into a KEM with techniques suitable for the QROM. The combination of the two proofs shows that breaking IND-CCA security of the KEM would lead to break the OW-CPA security of the underlying encryption scheme. Note, therefore, that the IND-CPA security of the underlying PKE has in this case no further effect on the final result, and can be considered instead just a guarantee that the scheme is indeed OW-CPA secure. The bound obtained is a "simplified" and "concrete" version (as presented by the authors) and, in particular, it is easy to notice that it does not depend on the number of queries $q_{\mathcal{H}}$ presented to the random oracle $\mathcal{H}$. The bound is further simplified since, as above, the underlying PKE presents no correctness error. □

[6] Same as above.

5 Known Attacks and Parameters (2.B.5/2.B.1)

We start by briefly presenting the hard problem on which DAGS is based, and then discuss the main attacks on the scheme and related security concerns.

5.1 Hard Problems from Coding Theory

Most of the code-based cryptographic constructions are based on the hardness of the following problem, known as the (q-ary) *Syndrome Decoding Problem (SDP).*

Problem 1 *Given an $(n-k) \times n$ full-rank matrix H and a vector $\mathbf{y}$, both defined over $\mathbb{F}_q$, and a non-negative integer w, find a vector $\mathbf{e} \in \mathbb{F}_q^n$ of weight w such that $H\mathbf{e}^T = \mathbf{y}$.*

The corresponding decision problem was proved to be NP-complete in 1978 [8], but only for binary codes. In 1994, A. Barg proved that this result holds for codes over all finite fields ([3], in Russian, and [4, Theorem 4.1]).

In addition, many schemes (including the original McEliece proposal) require the following computational assumption.

Assumption 1 *The public matrix output by the key generation algorithm is computationally indistinguishable from a uniformly chosen matrix of the same size.*

The assumption above is historically believed to be true, except for very particular cases. For instance, there exists a distinguisher (Faugère et al. [16]) for cryptographic protocols that make use of high-rate Goppa codes (like the CFS signature scheme [15]). Moreover, it is worth mentioning that the "classical" methods for obtaining an indistinguishable public matrix, such as the use of scrambling matrices S and P, are rather outdated and unpractical and can introduce vulnerabilities to the scheme as per the work of Strenzke et al. ([36, 37]). Thus, traditionally, the safest method (Biswas and Sendrier, [11]) to obtain the public matrix is simply to compute the systematic form of the private matrix.

5.2 Decoding Attacks

The main approach for solving SDP is the technique known as Information Set Decoding (ISD), first introduced by Prange [35]. Among several variants and generalizations, Peters showed [34] that it is possible to apply Prange's approach to

generic q-ary codes. Other approaches such as Statistical Decoding [24, 29] are usually considered less efficient. Thus, when choosing parameters, we will focus mainly on defeating attacks of the ISD family.

Hamdaoui and Sendrier in [22] provide non-asymptotic complexity estimates for ISD in the binary case. For codes over $\mathbb{F}_q$, instead, a bound is given in [30], which extends the work of Peters. For a practical evaluation of the ISD running times and corresponding security level, we used Peters's ISDFQ script[1].

Quantum Speedup. Bernstein in [9] shows that Grover's algorithm applies to ISD-like algorithms, effectively halving the asymptotic exponent in the complexity estimates. Later, it was proven in [25] that several variants of ISD have the potential to achieve a better exponent, however the improvement was disappointingly away from the factor of 2 that could be expected. For this reason, we simply treat the best quantum attack on our scheme to be "traditional" ISD (Prange) combined with Grover search.

5.3 FOPT

While, as we discussed above, recovering a private matrix from a public one can be in general a very difficult problem, the presence of extra structure in the code properties can have a considerable effect in lowering this difficulty.

A very effective structural attack was introduced by Faugère, Otmani, Perret and Tillich in [17]. The attack (for convenience referred to as FOPT) relies on the simple property (valid for every linear code) $H \cdot G^T = 0$ to build an algebraic system, using then Gröbner bases techniques to solve it. The special properties of alternant codes are fundamental, as they contribute to considerably reduce the number of unknowns of the system.

The attack was originally aimed at two variants of McEliece, introduced respectively in [7] and [28]. The first variant, using quasi-cyclic codes, was completely broken, and falls out of the scope of this paper. The second variant, instead, only considered quasi-dyadic Goppa codes. In this case, most of the parameters proposed have also been broken very easily, except for the binary case (i.e. base field $\mathbb{F}_2$). This was, in truth, not connected to the base field per se, but rather depended on the fact that, with a smaller base field, the authors provided a much higher extension degree m, as they were keeping constant the value $q^m = 2^{16}$. As it turns out, the extension degree m plays a key role in the attack, as it defines the dimension of the

solution space, which is equal, in fact, exactly to $m - 1$. In a successive paper [18], the authors provide a theoretical complexity bound for the attack, and point out that any scheme for which this dimension is less or equal to 20 should be within the scope of the attack.

Since GS codes are also alternant codes, the attack can be applied to our proposal as well. In the case of GS codes, though, there is one important difference to keep in mind. In fact, as shown in [32], the dimension of the solution space is defined by $mt - 1$, rather than $m - 1$ as for Goppa codes. This provides greater flexibility when designing parameters for the code, and it allows, for example, to keep the extension degree m small.

Recently, an extension of the FOPT attack appeared in [19]. The authors introduce a new technique called "folding", and show that it is possible to reduce the complexity of the FOPT attack to the complexity of attacking a much smaller code (the "folded" code), thanks to the strong properties of the automorphism group of the alternant codes in use. The attack turns out to be very efficient against Goppa codes, as it is possible to recover a folded code which is also a Goppa code. The paper features two tables with several sets of parameters, respectively for signature schemes, and encryption schemes. The parameters are either taken from the original papers, or generated ad hoc. While codes designed to work for signature schemes turn out to be very easy to attack (due to their particular nature), the situation for encryption is more complex. Despite a refinement in the techniques used to solve the algebraic system, some of the parameters could not be solved in practice, and even the binary Goppa codes of [28], with their relatively low dimension of 15, require a considerably high computational effort (at least 2^{150} operations).

It is not clear how the attack performs against GS codes, since the authors didn't present any explicit result against this particular family of codes, nor attempted to decode GS codes specifically. Thus, an attack against GS codes would use generic techniques for Alternant codes, and wouldn't benefit from the speedups which are specific to (binary) Goppa codes. Furthermore, the authors do not propose a concrete bound, but only provide experimental results. For these reasons, and until an accurate complexity analysis for an attack on GS codes is available, we choose to attain to the latest measurable guidelines (those suggested in [18]) and choose our parameters such that the dimension of the solution space for the algebraic system is strictly greater than 20.

5.4 Parameter Selection

To choose our parameters, we have to first keep in mind all of the remarks from the previous sections about decoding and structural attacks. For FOPT, we have the condition $mt \geq 21$. This guarantees at least 128 bits of security according to the bound presented in [18]. On the other hand, for ISD to be computationally intensive we require a sufficiently large number w of errors to decode: this is given by $st/2$ according to the minimum distance of GS codes.

In addition, we tune our parameters to optimize performance. In this regard, the best results are obtained when the extension degree m is as small as possible. This, however, requires the base field to be large enough to accommodate sufficiently big codes (against ISD attacks), since the maximum size for the code length n is capped by $q^m - s$. Realistically, this means we want q^m to be at least 2^{12}, and an optimal choice in this sense seems to be $q = 2^6, m = 2$. Finally, note that s is constrained to be a power of 2, and that odd values of t seem to offer best performance.

Putting all the pieces together, we are able to present three set of parameters, in the following table. These correspond to three of the security levels indicated by NIST (first column), which are related to the hardness of performing a key search attack on three different variants of a block cipher, such as AES (with key-length respectively 128, 192 and 256). As far as quantum attacks are concerned, we claim that ISD with Grover (see above) will usually require more resources than a Grover search attack on AES for the circuit depths suggested by NIST (parameter MAXDEPTH). Thus, classical security bits are the bottleneck in our case, and as such we choose our parameters to provide 128, 192 and 256 bits of classical security for security levels 1, 3 and 5 respectively.

We also included the estimated complexity of the structural attack (column FOPT), which is at least greater than 128 bits in all cases.

Table 1: Suggested DAGS Parameters.

Security Level	FOPT	q	m	n	k	k'	s	t	w
1	≥ 128	2^5	2	832	416	32	2^4	13	104
3	≥ 128	2^6	2	1216	512	32	2^5	11	176
5	≥ 128	2^6	2	2112	704	32	2^6	11	352

For practical reasons, during the rest of the paper we will refer to these parameters respectively as DAGS_1, DAGS_3 and DAGS_5.

6 Implementation and Performance Analysis (2.B.2)

6.1 Components

DAGS computations are detailed as follows:

Key generation:

1. Two polynomial multiplications in $\mathbb{F}_{q^m}$ and two in $\mathbb{F}_q$.
2. Six polynomial inversions in $\mathbb{F}_{q^m}$ and four in $\mathbb{F}_q$.
3. Two polynomial squarings in $\mathbb{F}_{q^m}$ and two in $\mathbb{F}_q$.
4. Two polynomial additions in $\mathbb{F}_{q^m}$ and two in $\mathbb{F}_q$.
5. One random generation of a polynomial in $\mathbb{F}_{q^m}$.

Encapsulation:

1. One polynomial multiplication in $\mathbb{F}_q$.
2. One polynomial addition in $\mathbb{F}_q$.
3. One random generation of a polynomial in $\mathbb{F}_q$.
4. One hash function computation.

Decapsulation:

1. Three polynomial multiplications in $\mathbb{F}_{q^m}$.
2. One polynomial power in $\mathbb{F}_{q^m}$.
3. One polynomial addition in $\mathbb{F}_{q^m}$.
4. One random generation of a polynomial in $\mathbb{F}_q$.
5. One hash function computation.

For DAGS_3 and DAGS_5, the finite field $\mathbb{F}_{2^6}$ is built using the polynomial x^6+x+1 and then extended to $\mathbb{F}_{2^{12}}$ using $x^2+x+\alpha^{34}$, where α is a primitive element of $\mathbb{F}_{2^6}$. For DAGS_1, we build $\mathbb{F}_{2^5}$ using x^5+x^2+1 and then extend it to $\mathbb{F}_{2^{10}}$ via $x^2+\alpha^4 x+\alpha$.

The three main functions from DAGS are defined as:

Key generation: the key generation algorithm *key_gen* is composed by three main functions: *binary_quasi_dyadique_sig*, *cauchy_support* and *key_pair*. The first two first functions are in charge of generating the signature and the Cauchy matrix respectively. The *key_pair* function generates public key and private key which is stored in memory for a better performance.

Encapsulation: the encapsulation algorithm is essentially composed of the function *encapsulation* in the file *encapsulation.c*, where it computes the expansion of the message and the McEliece-like encryption. In the end, the function computes the hash function $\mathcal{K}$ to get the shared secret.

Decapsulation: the decapsulation algorithm consists mainly of the function *decapsulation* in the file *decapsulation.c*, where we essentially run the decoding algorithm plus a few comparisons. In the end, we compute the hash function $\mathcal{K}$ to get the shared secret.

6.2 Time and Space Requirements

The implementation is in ANSI C. For the measurements we used a processor x64 Intel core i5-5300U@2.30GHz with 16GiB of RAM compiled with GCC version 6.3.020170516 without any optimization and running on Debian 9.2.

We start by considering space requirements. In Table 2 we recall the flow between two parties P_1 and P_2 in a standard Key Exchange protocol derived from a KEM.

Table 2: KEM-based Key Exchange flow

P_1		P_2
$(pk, sk) \leftarrow KEM.KeyGen$		
	$\xrightarrow{pk}$	
		$(\mathbf{k}, \mathbf{c}) \leftarrow KEM.Encaps(pk)$
	$\xleftarrow{C}$	
$\mathbf{k}/\perp \leftarrow KEM.Decaps(\mathbf{c}, sk)$		
	Shared Key := $\mathbf{k}$	

When instantiated with DAGS, the public key is given by the generator matrix G. The non-identity block M^T is $k \times (n-k) = k \times mst$ and is dyadic of order s, thus requires only $kmst/s = kmt$ elements of the base field for storage. The private key is given by the alternant matrix H' which is composed of stn elements of $\mathbb{F}_{q^m}$. Finally, the ciphertext C is the pair $(\mathbf{c}, \mathbf{d})$, that is, a q-ary vector of length n plus 256 bits. This leads to the following measurements (in bytes).

Table 3: Memory Requirements.

Parameter Set	Public Key	Private Key	Ciphertext
DAGS_1	6760	432640	552
DAGS_3	8448	1284096	944
DAGS_5	11616	2230272	1616

Table 4: Communication Bandwidth.

Message Flow	Transmitted Message	Size		
		DAGS_1	DAGS_3	DAGS_5
$\mathsf{P}_1 \rightarrow \mathsf{P}_2$	G	6760	8448	11616
$\mathsf{P}_2 \rightarrow \mathsf{P}_1$	$(\mathbf{c}, \mathbf{d})$	552	944	1616

Note that in our reference code, which is not optimized, we currently allocate a full byte for each element of $\mathbb{F}_{2^6}$ and two bytes for each element of $\mathbb{F}_{2^{12}}$ thus effectively wasting some memory. However, we expect to be able to represent elements more efficiently, namely using three bytes to store either four elements of $\mathbb{F}_{2^6}$ or two elements of $\mathbb{F}_{2^{12}}$. The measurements in Tables 3 and 4, above, are taken with respect to the latter method.

Furthermore, we would like to mention that the representation of the private key offers a significant tradeoff between time and space. In fact, it would be possible to store as private key the (quasi-dyadic) matrix H or even the defining vectors $\mathbf{u}, \mathbf{v}$ and $\mathbf{z}$, and then compute the alternant form during decapsulation (following [26, Ch. 12, §2,6]); this, however, would significantly slow down the decapsulation algorithm. Thus, we have opted to store H' instead and save computation time, although this obviously results in a very large private key. It is debatable which of the two routes is preferable, and we signal this as an implementor's choice.

We now move on to analyze time measurements. We are using x64 architecture and our measurements use an assembly instruction to get the time counter. We do this by calling "rdtsc" before and after the instruction, which gives us the cycles used by each function. Table 5 gives the results of our measurements represented by the mean after running the code 50 times.

Table 5: Timings.

Algorithm	Cycles		
	DAGS_1	DAGS_3	DAGS_5
Key Generation	49394032811	106876216775	136497712522
Encapsulation	20109354	26109354	49029613
Decapsulation	23639371	24639371	260829051

Note About Implementations. Our reference implementation and code have been compiled for DAGS_5. However, it is possible to adapt both to run with any set of parameters simply by editing the file *param.h*.

We would like to remark that our reference implementation is designed for clarity, rather than performance. However, we found that, as a consequence of NIST's platform and language of choice, there wouldn't be many significant performance differences by presenting an optimized version of our reference code. Thus, our Optimized Implementation is the same as the Reference Implementation, and all the measurements presented in this section refer to the reference code.

Our team is currently at work to complete additional implementations that could better showcase the potential of DAGS in terms of performance. These include code prepared with x86 assembly instructions (AVX) as well as a hardware implementation (FPGA) etc. We plan to include such additional implementations in time for the second evaluation period. A hint at the effectiveness of DAGS can be had by looking at the performance of the scheme presented in [14], which also features an implementation for embedded devices. In particular, we expect DAGS to perform especially well in hardware, due to the nature of the computations of the McEliece framework.

7 Advantages and Limitations (2.B.6)

We presented DAGS, a Key Encapsulation Mechanism based on Quasi-Dyadic Generalized Srivastava codes. We proved that DAGS is IND-CCA secure in the Random Oracle Model, and in the Quantum Random Oracle Model. Thanks to this feature, it is possible to employ DAGS not only as a key-exchange protocol (for which IND-CPA would be a sufficient requirement), but also in other contexts such as Hybrid Encryption, where IND-CCA is of paramount importance.

Like any scheme based on structured algebraic codes, DAGS is susceptible to the FOPT attack and its successive improvements; this can be seen as a limitation of the scheme. In fact, to defeat the attack, we need to respect stringent conditions on the minimal choices of values for the scheme, in particular the extension degree m and the value t. We remark that an accurate complexity analysis of the attack is, to date, not available, and a version of the attack targeting GS codes hasn't yet been provided. This forces us to choose conservative parameters, according to the theoretical bound of [18].

Nevertheless, DAGS is competitive and compares well with other other code-based schemes. These include the well-known McBits [10], as well as more recent proposals such as CAKE [5]. The former follows the work of Persichetti [33], and is built using binary Goppa codes, thus benefiting from a well-understood security assessment. The scheme however suffers from the same public key size issue as "classic" McEliece-like cryptosystems. On the other hand, CAKE, a protocol based on QC-MDPC codes, possesses some very nice features like compact keys and an easy implementation approach, but the QC-MDPC encryption scheme on which it is based suffers from a security-related drawback. This means that, in order to circumvent the Guo-Johansson-Stankovski (GJS) attack [21], the protocol is forced to employ ephemeral keys. While CAKE key generation is indeed very fast, this still causes an increase in computation time. Moreover, due to its non-trivial Decoding Failure Rate (DFR), achieving IND-CCA security becomes very hard, so that the CAKE protocol only claims to be IND-CPA secure.

Indeed, another advantage of our proposal is that it doesn't involve any decoding error. This is particularly favorable in a comparison with some lattice-based schemes like [13], [2] and [12], as well as CAKE. No decoding error allows for a simpler formulation and better security bounds in the IND-CCA security proof.

Our public key is much smaller than the McBits family, and of the same order of magnitude of CAKE. With respect to CAKE, it is possible to notice that, for the

same security level, DAGS requires lower overall communication bandwidth. This is because, while the size of a CAKE public key is slightly less than a DAGS key, DAGS uses much shorter codes, and as a consequence the ciphertext is quite small compared to a CAKE ciphertext. All the objects involved in the scheme are vectors of finite fields elements, which in turn are represented as binary strings; thus computations are very fast. The cost of computing the hash functions is minimized thanks to the parameter choice that makes sure the input $\mathbf{m}$ is only 256 bits. As a result, we expect that it will be possible to implement our scheme efficiently on multiple platforms.

Finally, we would like to highlight that a DAGS-based Key Exchange features an "asymmetric" structure, where the bandwidth cost and computational effort of the two parties are considerably different. In particular, in the flow described in Table 2, the party P_2 benefits from a much smaller message and faster computation (the encapsulation operation), whereas P_1 has to perform a key generation and a decapsulation (the most expensive operations in the scheme), and transmit a larger message (the public matrix). This is suitable for traditional client-server applications where the server side is usually expected to respond to a large number of requests and thus benefits from a lighter computational load. On the other hand, it is easy to imagine an instantiation, with reversed roles, which could be suitable for example in Internet-of-Things (IoT) applications, where it would be beneficial to lesser the burden on the client side, due to its typical processing, memory and energy constraints. All in all, our scheme offers great flexibility in key exchange applications, which is not the case for traditional key exchange protocols like Diffie-Hellman.

8 Acknowledgments

Cheikh Thiecoumba Gueye, Brice Odilon Boidje, Jean Belo Klamti, Ousmane Ndiaye and Gilbert Ndollane Dione were supported by the National Commission of Cryptology via the ISPQ project and by the CEA-MITIC via the CBC project. Jefferson E. Ricardini is Supported by the joint São Paulo Research Foundation (FAPESP)/Intel Research grant 2015/50520-6 "Efficient Post-Quantum Cryptography for Building Advanced Security". Gustavo Banegas has received funding under the European Union's Horizon 2020 research and innovation program (Marie Sklodowska-Curie grant agreement 643161 ECRYPT-NET).

References

[1] http://christianepeters.wordpress.com/publications/tools/.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. Cryptology ePrint Archive, Report 2015/1092, 2015. `http://eprint.iacr.org/2015/1092`.

[3] A. Barg. Some new NP-complete coding problems. *Probl. Peredachi Inf.*, 30:23–28, 1994. (in Russian).

[4] A. Barg. Complexity issues in coding theory. *Electronic Colloquium on Computational Complexity (ECCC)*, 4(46), 1997.

[5] Paulo SLM Barreto, Shay Gueron, Tim Gueneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, and Jean-Pierre Tillich. Cake: Code-based algorithm for key encapsulation.

[6] Paulo SLM Barreto, Richard Lindner, and Rafael Misoczki. Monoidic codes in cryptography. *PQCrypto*, 7071:179–199, 2011.

[7] T. P. Berger, P.-L. Cayrel, P. Gaborit, and A. Otmani. Reducing Key Length of the McEliece Cryptosystem. In *AFRICACRYPT*, pages 77–97, 2009.

[8] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978.

[9] Daniel J. Bernstein. *Grover vs. McEliece*, pages 73–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8086 LNCS, pages 250–272, 12 2013.

[11] B. Biswas and N. Sendrier. Mceliece cryptosystem implementation: Theory and practice. In *PQCrypto*, pages 47–62, 2008.

[12] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. Cryptology ePrint Archive, Report 2016/659, 2016. `http://eprint.iacr.org/2016/659`.

[13] Joppe W Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 553–570. IEEE, 2015.

[14] Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. Efficient implementation of a cca2-secure variant of McEliece using generalized Srivastava codes. In *Proceedings of PKC 2012, LNCS 7293, Springer-Verlag*, pages 138–155, 2012.

[15] N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a mceliece-based digital signature scheme. In *ASIACRYPT*, pages 157–174, 2001.

[16] J.-C. Faugère, V. Gauthier-Umaña, A. Otmani, L. Perret, and J.-P. Tillich. A distinguisher for high rate mceliece cryptosystems. In *Information Theory Workshop (ITW), 2011 IEEE*, pages 282 –286, oct. 2011.

[17] J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic cryptanalysis of mceliece variants with compact keys. In *EUROCRYPT*, pages 279–298, 2010.

[18] J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic Cryptanalysis of McEliece Variants with Compact Keys – Towards a Complexity Analysis. In *SCC '10: Proceedings of the 2nd International Conference on Symbolic Computation and Cryptography*, pages 45–55, RHUL, June 2010.

[19] Jean-Charles Faugere, Ayoub Otmani, Ludovic Perret, Frédéric De Portzamparc, and Jean-Pierre Tillich. Structural cryptanalysis of mceliece schemes with compact keys. *Designs, Codes and Cryptography*, 79(1):87–112, 2016.

[20] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999.

[21] Qian Guo, Thomas Johansson, and Paul Stankovski. *A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors*, pages 789–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[22] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. Cryptology ePrint Archive, Report 2013/162, 2013. `http://eprint.iacr.org/2013/162`.

[23] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. Cryptology ePrint Archive, Report 2017/604, 2017. `http://eprint.iacr.org/2017/604`.

[24] A. Al Jabri. *A Statistical Decoding Algorithm for General Linear Block Codes*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[25] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. In Tanja Lange and Tsuyoshi Takagi, editors, *PQCrypto 2017*, volume 10346 of *LNCS*, pages 69–89. Springer, 2017.

[26] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16. North-Holland Mathematical Library, 1977.

[27] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[28] R. Misoczki and P. S. L. M. Barreto. Compact mceliece keys from goppa codes. In *Selected Areas in Cryptography*, pages 376–392, 2009.

[29] R. Niebuhr. *Statistical Decoding of Codes over* $\mathbb{F}_q$, pages 217–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[30] R. Niebuhr, E. Persichetti, P.-L. Cayrel, S. Bulygin, and J. Buchmann. On lower bounds for information set decoding over $\mathfrak{U}_q$ and on the effect of partial knowledge. *Int. J. Inf. Coding Theory*, 4(1):47–78, January 2017.

[31] R. Nojima, H. Imai, K. Kobara, and K. Morozov. Semantic security for the McEliece cryptosystem without random oracles. *Des. Codes Cryptography*, 49(1-3):289–305, 2008.

[32] Edoardo Persichetti. Compact mceliece keys based on quasi-dyadic srivastava codes. *Journal of Mathematical Cryptology*, 6(2):149–169, 2012.

[33] Edoardo Persichetti. Secure and anonymous hybrid encryption from coding theory. In Philippe Gaborit, editor, *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, pages 174–187, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[34] C. Peters. Information-set decoding for linear codes over $\mathbb{F}_q$. In *PQCrypto*, LNCS, pages 81–94, 2010.

[35] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions*, IT-8:S5–S9, 1962.

[36] F. Strenzke. A timing attack against the secret permutation in the mceliece pkc. In *PQCrypto*, pages 95–107, 2010.

[37] F. Strenzke, E. Tews, H. G. Molter, R. Overbeck, and A. Shoufan. Side channels in the mceliece pkc. In *PQCrypto*, pages 216–229, 2008.

A Note on the choice of ω

In this section we point out some considerations about the choice of the offset ω during the key generation process.

The usual decoding algorithm for alternant codes, for example as in [26], relies on the special form of the parity-check matrix ($H_{ij} = y_j x_j^{i-1}$). The first step is to recover the error locator polynomial $\sigma(x)$, by means of the euclidean algorithm for polynomial division; then it proceeds by finding the roots of σ. There is a 1-1 correspondence between these roots and the error positions: in fact, there is an error in position i if and only if $\sigma(1/x_i) = 0$.
Of course, if one of the x_i's is equal to 0, it is not possible to find the root, and to detect the error.

Now, the generation of the error vector is random, hence we can assume the probability of having an error in position i to be around $st/2n$; since the codes give the best performance when mst is close to $n/2$, we can estimate this probability as $1/4m$, which is reasonably low for any nontrivial choice of m; however, we still argue that the code is not fully decodable and we now explain how to adapt the key generation algorithm to ensure that all the x_i's are nonzero.

As part of the key generation algorithm we assign to each x_i the value L_i, hence it is enough to restrict the possible choices for ω to the set $\{\alpha \in \mathbb{F}_{q^m} | \alpha \neq 1/h_i + 1/h_0, i = 0, \ldots, n-1\}$. In doing so, we considerably restrict the possible choices for ω but we ensure that the decoding algorithm works properly.

FrodoKEM
Learning With Errors Key Encapsulation

Algorithm Specifications And Supporting Documentation

November 30, 2017

Contents

1 Introduction and design rationale

This submission defines a family of key-encapsulation mechanisms (KEMs), collectively called FrodoKEM. The FrodoKEM schemes are designed to be *conservative* yet *practical* post-quantum constructions whose security derives from cautious parameterizations of the well-studied *learning with errors* problem, which in turn has close connections to conjectured-hard problems on *generic*, "algebraically unstructured" lattices.

Concretely, FrodoKEM is designed for IND-CCA security at two levels:

- FrodoKEM-640, which targets Level 1 in the NIST call for proposals (matching or exceeding the brute-force security of AES-128), and
- FrodoKEM-976, which targets Level 3 in the NIST call for proposals (matching or exceeding the brute-force security of AES-192).

Two variants of each of the above schemes are provided:

- FrodoKEM-640-AES and FrodoKEM-976-AES, which use AES128 to pseudorandomly generate a large public matrix (called $\mathbf{A}$).
- FrodoKEM-640-cSHAKE and FrodoKEM-976-cSHAKE, which use cSHAKE128 to pseudorandomly generate the matrix.

The AES variants are particularly suitable for devices having AES hardware acceleration (such as AES-NI on Intel platforms), while the cSHAKE variants generally provide competitive or better performance in comparison with the AES variants in the absence of hardware acceleration.

In the remainder of this section, we outline FrodoKEM's scientific lineage, briefly explain our design choices (with further details appearing in subsequent sections), and describe other features of our proposal beyond those explicitly requested by NIST.

1.1 Pedigree

The core of FrodoKEM is a public-key encryption scheme called FrodoPKE,[1] whose IND-CPA security is tightly related to the hardness of a corresponding *learning with errors* problem. Here we briefly recall the scientific lineage of these systems. See the surveys [82, 103, 92] for further details.

The seminal works of Ajtai [3] (published in 1996) and Ajtai–Dwork [4] (published in 1997) gave the first cryptographic constructions whose security properties followed from the conjectured *worst-case* hardness of various problems on point *lattices* in $\mathbb{R}^n$. In subsequent years, these works were substantially refined and improved, e.g., in [57, 30, 81, 101, 84]. Notably, in work published in 2005, Regev [102] defined the *learning with errors* (LWE) problem, proved the hardness of (certain parameterizations of) LWE assuming the hardness of various worst-case lattice problems against *quantum* algorithms, and defined a public-key encryption scheme whose IND-CPA security is tightly related to the hardness of LWE.[2]

Regev's initial work on LWE was followed by much more, which, among other things:

- provided additional theoretical support for the hardness of various LWE parameterizations (e.g., [88, 13, 28, 48, 83, 94]),
- extensively analyzed the concrete security of LWE and closely related lattice problems (e.g., [85, 40, 76, 8, 39, 6, 7, 69, 65, 10, 11, 24, 5, 9], among countless others), and
- constructed LWE-based cryptosystems with improved efficiency or additional functionality (e.g., [96, 95, 55, 32, 29, 56, 22, 58]).

In particular, in work published in 2011, Lindner and Peikert [75] gave a more efficient LWE-based public-key encryption scheme that uses a square public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ instead of an oblong rectangular one.

The FrodoPKE scheme from this submission is an instantiation and implementation of the Lindner–Peikert scheme [75] with some modifications, such as: pseudorandom generation of the public matrix $\mathbf{A}$ from a small seed, more balanced key and ciphertext sizes, and new LWE parameters.

[1] FrodoPKE is an intermediate building block used to create FrodoKEM, but is not a submission to the NIST competition.

[2] As pointed out in [89], Regev's encryption scheme implicitly contains an (unauthenticated) "approximate" key-exchange protocol analogous to the classic Diffie–Hellman protocol [46].

Frodo. FrodoPKE is closely related to an earlier work [24], called "Frodo," by a subset of the authors of this submission, which appeared at the 2016 ACM CCS conference. For clarity, we refer to the conference version as FrodoCCS, and the KEM defined in this submission as FrodoKEM. The main differences are as follows:

- FrodoCCS was described as an unauthenticated key-exchange protocol, which can equivalently be viewed as an IND-CPA-secure KEM, whereas FrodoKEM is designed to be an IND-CCA-secure KEM.
- FrodoCCS used a "reconciliation mechanism" to extract shared-key bits from approximately equal values (similarly to [47, 91, 25, 11]), whereas FrodoKEM uses simpler key transport via public-key encryption (as in [102, 75]).
- FrodoKEM uses significantly "wider" LWE error distributions than FrodoCCS does, which conform to certain worst-case hardness theorems (see below).
- FrodoKEM uses different symmetric-key primitives than FrodoCCS does.

Chosen-ciphertext security. FrodoKEM achieves IND-CCA security by way of a transformation of the IND-CPA-secure FrodoPKE. In work published in 1999, Fujisaki and Okamoto [51] gave a generic transform from an IND-CPA PKE to an IND-CCA PKE, in the random-oracle model. At a high level, the Fujisaki–Okamoto transform derives encryption coins pseudorandomly, and decryption regenerates these coins to re-encrypt and check that the ciphertext is well-formed. In 2016, Targhi and Unruh [111] gave a modification of the Fujisaki–Okamoto transform that achieves IND-CCA security in the *quantum* random-oracle model. In 2017, Hofheinz, Hövelmanns, and Kiltz [61] gave several variants of the Fujisaki–Okamoto and Targhi–Unruh transforms that in particular convert an IND-CPA-secure PKE into an IND-CCA-secure KEM, and analyzed them in both the classical and quantum random-oracle models, even for PKEs with non-zero decryption error. FrodoKEM is constructed from FrodoPKE using a slight variant of one of the constructions from [61], that includes additional values in hash computations to avoid multi-target attacks.

1.2 Design overview and rationale

Given the high cost and slow deployment of entirely new cryptographic systems, the desired decades-long lifetime of such systems, and the unpredictable trajectory of quantum computing technology and quantum cryptanalysis over the coming years, we argue that any post-quantum standard should follow a *conservative* approach that errs comfortably on the side of *security* and *simplicity* over performance and (premature) optimization. This principle permeates the design choices behind FrodoKEM, as we now describe.

1.2.1 Generic, algebraically unstructured lattices

The security of every public-key cryptosystem depends on the presumed intractability of one or more computational problems. In lattice-based cryptography, the (plain) LWE problem [102] relates to solving a "noisy" linear system modulo a known integer; it can also be interpreted as the problem of decoding a random "unstructured" lattice from a certain class. There are also "algebraically structured" variants, called Ring-LWE [78, 94] and Module-LWE [27, 72], and problems associated with the classic NTRU cryptosystem [60], which are more compact and computationally efficient, but also have the potential for weaknesses due to the extra structure.

After a good deal of investigation, the state of the art for recommended parameterizations of algebraic LWE variants does not indicate any particular weaknesses in comparison to plain LWE. However, at present there appear to be some gaps between the (quantum) complexity of some *related*, seemingly weaker problems on certain kinds of algebraic lattices and their counterparts on general lattices. (See below for details.) Of course, this only represents our current understanding of these problems, which could potentially change with further cryptanalytic effort.

Given the unpredictable long-term outlook for algebraically structured lattices, and because any post-quantum standard should remain secure for decades into the future—including against new quantum attacks—we have based our proposal on the algebraically unstructured, plain LWE problem with conservative parameterizations (see Section 1.2.2). While this choice comes at some cost in efficiency versus algebraic lattice problems, our proposal is still eminently practical for the vast majority of today's devices, networks, and applications, and will become only more so in the coming years.

Algebraic lattices. Ring-LWE, Module-LWE, and NTRU-related problems can be viewed as decoding (or in the case of NTRU, shortest vector) problems on random "algebraically structured" lattices over certain polynomial rings. (Formally, the lattices are modules of a certain rank over the ring.) Similarly to LWE, various parameterizations of Ring-LWE and Module-LWE, and even some non-standard versions of NTRU [110], have been proven hard assuming the *worst-case* quantum hardness of certain problems on lattices corresponding to *ideals* or *modules* over the ring [78, 72, 94].

For recommended parameterizations of Ring- and Module-LWE, the current best attacks perform essentially the same as those for plain LWE, apart from some obvious linear-factor (in the ring dimension) savings in time and memory; the same goes for the underlying worst-case problems on ideal and module lattices, versus generic lattices [40, 107, 62, 26, 70].[3] However, some conventional NTRU parameterizations admit specialized attacks with significantly better asymptotic performance than on generic lattices with the same parameters [65, 66]. In addition, a series of recent works [31, 42, 43] has yielded a quantum polynomial-time algorithm for very large but *subexponential* $2^{\tilde{O}(\sqrt{n})}$ approximations to the worst-case Shortest Vector Problem on *ideal* lattices over a widely used class of rings (in contrast to just slightly subexponential $2^{O(n \log\log n/\log n)}$ factors obtainable for general lattices [74, 108]). Note that these subexponential approximation factors are still much larger than the small *polynomial* factors that are typically used in cryptography (so the reductions have not been made vacuous), and the algorithms from [31, 42, 43] do not yet have any impact on Ring- or Module-LWE themselves.

1.2.2 Parameters from worst-case reductions and conservative cryptanalysis

Like all cryptographic problems, LWE is an *average-case* problem, i.e., input instances are chosen *at random* from a prescribed probability distribution. As already mentioned, some parameterizations of LWE admit (quantum or classical) reductions from *worst-case* lattice problems. That is, any algorithm that solves n-dimensional LWE (with some non-negligible advantage) can be converted with some polynomial overhead into a (quantum) algorithm that solves certain short-vector problems on *any* n-dimensional lattice (with high probability). Therefore, if the latter problems have *some* (quantumly) hard instances, then *random* instances of LWE are also hard.

Worst-case/average-case reductions help guide the search for cryptographically hard problems in a large design space, and offer (at minimum) evidence that the particular distribution of inputs does not introduce any structural weaknesses. This is in contrast to several lattice-based proposals that lacked such reductions, and turned out to be insecure because their distributions made "planted" secrets easy to recover, e.g., [109, 86, 31, 42]. Indeed, Micciancio and Regev [85] argue that a reduction from a hard worst-case problem

> "... assures us that attacks on the cryptographic construction are likely to be effective only for small choices of parameters and not asymptotically. In other words, it assures us that there are no fundamental flaws in the design of our cryptographic construction... In principle the worst-case security guarantee can help us in choosing concrete parameters for the cryptosystem, although in practice this leads to what seems like overly conservative estimates, and ... one often sets the parameters based on the best known attacks."

Not all LWE parameterizations admit reductions from worst-case lattice problems. For example, the iterative quantum reductions from [102, 94] require the use of Gaussian error having standard deviation at least $c\sqrt{n}$ for an arbitrary constant $c > 1/(2\pi)$, where n is the dimension of the LWE secret. In practice, a drawback of using such "wide" error distributions for cryptography is the relatively large modulus required to avoid decryption error, which leads to larger dimensions n and sizes of keys and ciphertexts for a desired level of concrete security. Subsequent works like [48, 83] provided weaker reductions for "narrower" error distributions, such as uniform over a small set (even $\{0, 1\}$), but only by restricting the number of LWE samples available to the attacker—to fewer than the number exposed by LWE-based cryptosystems [102, 75], in the case of moderately small errors. Note that some limitation on the number of samples is necessary,

[3] Some unconventional parameterizations of Ring-LWE were specifically devised to be breakable by certain algebraic attacks [50, 37, 33, 38]. However, it was later shown that their error distributions are insufficiently "wide" relative to the ring, so they reveal errorless (or nearly so) linear equations and can therefore be broken even more efficiently using elementary, non-algebraic means [33, 93].

because LWE with errors bounded by (say) a constant is solvable in polynomial time, given a large enough polynomial number of samples [14, 6]. Currently, there is still a sizeable gap between small-error LWE parameters that are known to be vulnerable, and those conforming to a worst-case reduction. Most proposed implementations use parameters that lie within this gap.

Error width and an alternative worst-case reduction. In keeping with our philosophy of using conservative choices of hard problems that still admit practical implementations, our proposal uses "moderately wide" Gaussian error of standard deviation $\sigma \geq 2.3$, and (automatically) limits the number of LWE samples available to the adversary. Although such parameters do not conform to the full quantum reductions from [102, 94] for our choices of n, we show that they *do* conform to an alternative, classical worst-case reduction that can be extracted from those works. (We note that the original version of Frodo from [24] used a smaller σ that is not compatible with any of these reductions.)

In a little more detail, the alternative reduction is from a worst-case lattice problem we call "Bounded Distance Decoding with Discrete Gaussian Samples" (BDDwDGS), which has been closely investigated (though not under that name) in several works [2, 77, 102, 45]. Along with being classical (non-quantum), a main advantage of the alternative reduction is that it works for LWE with (1) Gaussian error whose width only needs to exceed the "smoothing parameter" [84] of the integer lattice $\mathbb{Z}$ for tiny enough $\varepsilon > 0$, and (2) a correspondingly bounded number of samples. We view this reduction as evidence that the smoothing parameter of $\mathbb{Z}$ is an important qualitative threshold for LWE error, which is why we use a standard deviation σ which is comfortably above it. We also view the reduction as narrowing the gap between the known weakness of small-error LWE with a large number of samples, and its apparent hardness with a small number of samples. See Section 5.1.5 for full details.

We stress that we use the worst-case reduction only for guidance in choosing a narrow enough error distribution for practice that still has some theoretical support, and not for any concrete security claim. As alluded to in the above quote from [85] (see also [36]), the known worst-case reduction does not yield any meaningful "end-to-end" security guarantee for our concrete parameters based on the conjecture hardness of a worst-case problem, because the reduction is *non-tight*: it has some significant polynomial overhead in running time and number of discrete Gaussian samples used, versus the number of LWE samples it produces. (Improving the tightness of worst-case reductions is an interesting problem.) Instead, as stated in the above quote from [85], we choose concrete parameters using a conservative analysis of the best known cryptanalytic attacks, as described next.

Concrete cryptanalysis using core-SVP hardness. Our concrete security estimates are based on a conservative methodology, as previously used for NewHope [11] and Frodo [24] and detailed in Section 5.2.1, that estimates the "core-SVP hardness" of solving the underlying LWE problem. This methodology builds on the extensive prior cryptanalysis of LWE and related lattice problems, and was further validated by recent work [9], which concluded that its experimental results "confirm that lattice-reduction largely follows the behavior expected from the 2016 estimate [11]." The core-SVP methodology counts only the *first-order* exponential cost of just *one* (quantum) shortest-vector computation on a lattice of appropriate dimension to solve the relevant LWE problem. Because it ignores lower-order terms like the significant subexponential factors in the runtime, as well as the large exponential memory requirements, it significantly underestimates the actual cost of known attacks, and allows for significant future improvement in these attacks.

1.2.3 Simplicity of design and implementation

Using plain LWE allows us to construct encryption and key-encapsulation schemes that are *simple* and *easy to implement*, reducing the potential for errors. Wherever possible, design decisions were made in favor of simplicity over more sophisticated mechanisms.

Modular arithmetic. Our LWE parameters use an integer modulus $q \leq 2^{16}$ that is always a power of two. This ensures that only single-precision arithmetic is needed, and that reduction modulo q can be computed almost for free by bit-masking. (Reduction modulo 2^{16} is even entirely free when 16-bit data types are used.) Modular arithmetic is thus easy to implement correctly and in a way that is resistant to cache and timing side-channel attacks.

Error sampling. Although our "ideal" LWE error distribution is a Gaussian with an appropriate standard deviation, our implementation actually uses a distribution that is very close to it. Sampling from the distribution is quite simple via a small lookup table and a few random bits, and is resistant to cache and timing side-channels. (See Section 2.2.4 for details.) Using this alternative error distribution comes at very little expense in the concrete security of FrodoKEM, which we show by analyzing the Rényi divergence between the two distributions, following [15]. See Section 5.1.3 for full details.

Matrix-vector operations. Apart from error sampling and calls to symmetric primitives like AES or cSHAKE, the main operations in our schemes are simple matrix-vector products. Compared to systems like NewHope [11] or Kyber [23] that use algebraically structured LWE variants, our system has moderately larger running times and bandwidth requirements, but is also significantly simpler, because there is no need to implement fast polynomial multiplication algorithms (like the number-theoretic transform for a prime modulus) to exploit the algebraic structure.

Encryption and key encapsulation without reconciliation. Our PKE and KEM use the original method from Regev's encryption scheme [102] of transmitting secret bits by simply adding their $(q/2)$-multiples to pseudorandom values that the receiver can (approximately) subtract away. We do not need or use any of the more complicated reconciliation mechanisms that were developed in the context of key-exchange protocols (as mentioned above in Section 1.1).

In addition, unlike the Ring-LWE-based NewHope scheme [11], which transmits data using non-trivial lattice codes to make up for bandwidth losses arising from a sparse set of friendly ring dimensions, plain-LWE-based constructions do not have such bandwidth losses because the dimensions can be set freely. Therefore, we also have no need for complex bandwidth-saving optimizations.

Simple and compact code base. Our focus on simplicity is manifested in the FrodoKEM code base. For example, our x64 implementation of the full FrodoKEM scheme consists of only about 250 lines of plain C code (not including header files and code for symmetric primitives). Moreover, the exact same code can be used for other LWE parameters and security levels, solely by changing compile-time constants.

1.3 Other features

Flexible, fine-grained choice of parameters. The plain LWE problem imposes very few requirements on its parameters, which makes it possible to rather tightly meet almost any desired security target in an automated way, using the methodology described in Section 5.2.1. Alternative parameters can be selected to reflect future advances in cryptanalysis, or to support other features beyond basic encryption and key encapsulation. For example, by using a larger LWE modulus (e.g., $q = 2^{32}$ or $q = 2^{64}$) and appropriate dimensions for a desired security level, FrodoPKE can easily support a large number of *homomorphic* additions, or multiplications by (small) public scalars, on ciphertexts. Using even larger moduli, it can even be made into a leveled or fully homomorphic encryption scheme, following [29].

Dynamically generated public matrices. To reduce the size of public keys and accelerate encryption, the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ could potentially be a fixed value that is chosen in a "nothing-up-my-sleeve" fashion [18] and used for all keys (see [25] for an example of this in a Ring-LWE-based system). However, to avoid the possibility of backdoors and all-for-the-price-of-one attacks [1], following prior work [11, 24] we dynamically and pseudorandomly generate a fresh matrix $\mathbf{A}$ for every generated key. The pseudorandom derivation is defined in a way that allows for fast generation of the entire matrix, or row-by-row generation on devices that cannot store the entire matrix in memory. See Section 2.2.5 for details.

2 Written specification

2.1 Background

This defines the cryptographic primitives and security notions that are relevant to FrodoPKE and FrodoKEM, as well as the mathematical background required to analyze their security.

2.1.1 Notation

We use the following notation throughout this document.

- Vectors are denoted with bold lower-case letters (e.g., $\mathbf{a}, \mathbf{b}, \mathbf{v}$), and matrices are denoted with bold upper-case letters (e.g., $\mathbf{A}, \mathbf{B}, \mathbf{S}$). For a set D, the set of m-dimensional vectors with entries in D is denoted by D^m, and the set of m-by-n matrices with entries in D is denoted by $D^{m \times n}$.
- For an n-dimensional vector $\mathbf{v}$, its ith entry for $0 \leq i < n$ is denoted by $\mathbf{v}_i$.
- For an m-by-n matrix $\mathbf{A}$, its (i, j)th entry (i.e., the entry in the ith row and jth column) for $0 \leq i < m$ and $0 \leq j < n$ is denoted by $\mathbf{A}_{i,j}$, and its ith row is denoted by $\mathbf{A}_i = (\mathbf{A}_{i,0}, \mathbf{A}_{i,1}, \ldots, \mathbf{A}_{i,n-1})$.
- An m-bit string $\mathbf{k} \in \{0,1\}^m$ is written as a vector over the set $\{0,1\}$ and its ith bit for $0 \leq i < m$ is denoted by $\mathbf{k}_i$.
- The ring of integers is denoted by $\mathbb{Z}$, and, for a positive integer q, the quotient ring of integers modulo q is denoted by $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$.
- For a probability distribution χ, the notation $e \leftarrow\!\!\$\ \chi$ denotes drawing a value e according to χ. The n-fold product distribution of χ with itself is denoted by χ^n.
- For a finite set S, the uniform distribution on S is denoted by $U(S)$.
- The floor of a real number a, i.e., the largest integer less than or equal to a, is denoted by $\lfloor a \rfloor$.
- The closest integer to a real number a (with ties broken upward) is denoted by $\lfloor a \rceil = \lfloor a + 1/2 \rfloor$.
- For a real vector $\mathbf{v} \in \mathbb{R}^n$, its Euclidean (i.e., ℓ_2) norm is denoted by $\|\mathbf{v}\|$.
- For two n-dimensional vectors $\mathbf{a}, \mathbf{b}$ over a common ring R, their inner product is denoted by $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=0}^{n-1} \mathbf{a}_i \mathbf{b}_i \in R$.

2.1.2 Cryptographic definitions

This section states definitions of the cryptographic primitives that are specified in this document, along with their correctness and security notions. This document specifies a key encapsulation mechanism (KEM), formally defined by three algorithms as follows.

Definition 2.1 (Key encapsulation mechanism). A *key encapsulation mechanism* KEM is a tuple of algorithms (KeyGen, Encaps, Decaps) along with a finite keyspace $\mathcal{K}$:

- KeyGen() $\$\!\!\rightarrow (pk, sk)$: A probabilistic *key generation algorithm* that outputs a public key pk and a secret key sk.
- Encaps(pk) $\$\!\!\rightarrow (c, \mathbf{ss})$: A probabilistic *encapsulation algorithm* that takes as input a public key pk, and outputs an encapsulation c and a shared secret $\mathbf{ss} \in \mathcal{K}$. The encapsulation is sometimes called a ciphertext.
- Decaps(c, sk) $\rightarrow \mathbf{ss}'$: A (usually deterministic) *decapsulation algorithm* that takes as input an encapsulation c and a secret key sk, and outputs a shared secret $\mathbf{ss}' \in \mathcal{K}$.

The notion of δ-correctness gives a bound on the probability of a legitimate protocol execution producing different keys in encapsulation and decapsulation.

Definition 2.2 (δ-correctness for KEMs). A key encapsulation mechanism KEM is δ-*correct* if

$$\Pr\left[\mathbf{ss}' \neq \mathbf{ss} : (pk, sk) \leftarrow\!\!\$\ \mathsf{KEM.KeyGen}(); (c, \mathbf{ss}) \leftarrow\!\!\$\ \mathsf{KEM.Encaps}(pk); \mathbf{ss}' \leftarrow \mathsf{KEM.Decaps}(c, sk)\right] \leq \delta \ .$$

The following defines IND-CCA security for a key encapsulation mechanism.

Definition 2.3 (IND-CCA for KEMs). Let KEM be a key encapsulation mechanism with keyspace $\mathcal{K}$, and let $\mathcal{A}$ be an algorithm. The security experiment for *indistinguishability under adaptive chosen ciphertext*

*attack (*IND-CCA2*, or just* IND-CCA*)* of KEM is $\mathrm{Exp}^{\mathsf{ind\text{-}cca}}_{\mathsf{KEM}}(\mathcal{A})$ shown in Figure 1. The advantage of $\mathcal{A}$ in the experiment is

$$\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{KEM}}(\mathcal{A}) := \left| \Pr\left[\mathrm{Exp}^{\mathsf{ind\text{-}cca}}_{\mathsf{KEM}}(\mathcal{A}) \Rightarrow 1\right] - \frac{1}{2} \right| \ .$$

Note that $\mathcal{A}$ can be a classical or quantum algorithm. If $\mathcal{A}$ is a quantum algorithm, then we only consider the model in which the adversary makes classical queries to its $\mathcal{O}_{\mathrm{Decaps}}$ oracle.

Experiment $\mathrm{Exp}^{\mathsf{ind\text{-}cca}}_{\mathsf{KEM}}(\mathcal{A})$:

1: $(pk, sk) \leftarrow\!\!\$\ \mathsf{KEM}.\mathrm{KeyGen}()$
2: $b \leftarrow\!\!\$\ \{0,1\}$
3: $(c^*, \mathbf{ss}_0) \leftarrow\!\!\$\ \mathsf{KEM}.\mathrm{Encaps}(pk)$
4: $\mathbf{ss}_1 \leftarrow\!\!\$\ U(\mathcal{K})$
5: $b' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}_{\mathrm{Decaps}}(\cdot)}(pk, \mathbf{ss}_b, c^*)$
6: **if** $b' = b$ **then**
7: **return** 1
8: **else**
9: **return** 0

Oracle $\mathcal{O}_{\mathrm{Decaps}}(c)$:

1: **if** $c = c^*$ **then**
2: **return** $\perp$
3: **else**
4: **return** $\mathsf{KEM}.\mathrm{Decaps}(c, sk)$

Figure 1: Security experiment for indistinguishability under adaptive chosen ciphertext attack (IND-CCA2, or just IND-CCA) of a key encapsulation mechanism KEM for an adversary $\mathcal{A}$.

The key encapsulation mechanism specified in this document is obtained by a transformation from a public-key encryption (PKE) scheme; a PKE scheme is formally defined as follows.

Definition 2.4 (Public key encryption scheme). A *public key encryption scheme* PKE is a tuple of algorithms (KeyGen, Enc, Dec) along with a message space $\mathcal{M}$:

- $\mathrm{KeyGen}() \xrightarrow{\$} (pk, sk)$: A probabilistic *key generation algorithm* that outputs a public key pk and a secret key sk.
- $\mathrm{Enc}(m, pk) \xrightarrow{\$} c$: A probabilistic *encryption algorithm* that takes as input a message $m \in \mathcal{M}$ and public key pk, and outputs a ciphertext c. The deterministic form is denoted $\mathrm{Enc}(m, pk; r) \rightarrow c$, where the randomness $r \in \mathcal{R}$ is passed as an explicit input; $\mathcal{R}$ is called the *randomness space* of the encryption algorithm.
- $\mathrm{Dec}(c, sk) \rightarrow m'$ or $\perp$: A deterministic *decryption algorithm* that takes as input a ciphertext c and secret key sk, and outputs a message $m' \in \mathcal{M}$ or a special error symbol $\perp \notin \mathcal{M}$.

The notion of δ-correctness captures an upper bound on the probability of decryption failure in a legitimate execution of the scheme.

Definition 2.5 (δ-correctness for PKEs [61]). A public key encryption scheme PKE with message space $\mathcal{M}$ is *δ-correct* if

$$\mathbb{E}\left[\max_{m \in \mathcal{M}} \Pr\left[\mathsf{PKE}.\mathrm{Dec}(c, sk) \neq m : c \leftarrow\!\!\$\ \mathsf{PKE}.\mathrm{Enc}(m, pk)\right]\right] \leq \delta \ , \tag{1}$$

where the expectation is taken over $(pk, sk) \leftarrow\!\!\$\ \mathsf{PKE}.\mathrm{KeyGen}()$.

In our PKE, the probability expression in Equation (1) has no dependence on m, so the condition simplifies to

$$\Pr\left[\mathsf{PKE}.\mathrm{Dec}(c, sk) \neq m : (pk, sk) \leftarrow \mathsf{PKE}.\mathrm{KeyGen}(); c \leftarrow\!\!\$\ \mathsf{PKE}.\mathrm{Enc}(m, pk)\right] \leq \delta \ , \tag{2}$$

which is what we analyze when calculating the probability of decryption failure (see Section 2.2.7).

The PKE scheme we use as the basis for the KEM transformation in Section 2.2.8 is required to satisfy the notion of IND-CPA security, which is defined as follows.

Definition 2.6 (IND-CPA for PKE). Let PKE be a public key encryption scheme, and let $\mathcal{A}$ be an algorithm. The security experiment for *indistinguishability under chosen plaintext attack (*IND-CPA*)* of PKE is $\mathrm{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{A})$ shown in Figure 2. The advantage of $\mathcal{A}$ in the experiment is

$$\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{A}) := \left| \Pr\left[\mathrm{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{A}) \Rightarrow 1\right] - \frac{1}{2} \right|.$$

Note that $\mathcal{A}$ can be a classical or quantum algorithm.

Experiment $\mathrm{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{A})$:

1: $(pk, sk) \leftarrow_{\$} \mathsf{PKE.KeyGen}()$
2: $(m_0, m_1, st) \leftarrow_{\$} \mathcal{A}(pk)$
3: $b \leftarrow_{\$} \{0,1\}$
4: $c^* \leftarrow_{\$} \mathsf{PKE.Enc}(m_b, pk)$
5: $b' \leftarrow_{\$} \mathcal{A}(pk, c^*, st)$
6: **if** $b' = b$ **then**
7: **return** 1
8: **else**
9: **return** 0

Figure 2: Security experiment for indistinguishability under chosen plaintext attack (IND-CPA) of a public key encryption scheme PKE against an adversary $\mathcal{A}$.

2.1.3 Learning With Errors

The security of our proposed PKE and KEM relies on the hardness of the *Learning With Errors* (LWE) problem, a generalization of the classic Learning Parities with Noise problem (see, e.g., [20]) first defined by Regev [102]. This section defines the LWE probability distributions and computational problems.

Definition 2.7 (LWE distribution). Let n, q be positive integers, and let χ be a distribution over $\mathbb{Z}$. For an $\mathbf{s} \in \mathbb{Z}_q^n$, the *LWE distribution* $A_{\mathbf{s},\chi}$ is the distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ obtained by choosing $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random and an integer error $e \in \mathbb{Z}$ from χ, and outputting the pair $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e \bmod q) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$.

There are two main kinds of computational LWE problem: *search*, which is to recover the secret $\mathbf{s} \in \mathbb{Z}_q^n$ given a certain number of samples drawn from the LWE distribution $A_{\mathbf{s},\chi}$; and *decision*, which is to distinguish a certain number of samples drawn from the LWE distribution from uniformly random samples. For both variants, one often considers two distributions of the secret $\mathbf{s} \in \mathbb{Z}_q^n$: the uniform distribution, and the distribution $\chi^n \bmod q$ where each coordinate is drawn from the error distribution χ and reduced modulo q. The latter is often called the "normal form" of LWE.

Definition 2.8 (LWE Search Problem). Let n, m, q be positive integers, and let χ be a distribution over $\mathbb{Z}$. The *uniform-secret* (respectively, *normal-form*) learning with errors *search* problem with parameters (n, m, q, χ), denoted by $\mathsf{SLWE}_{n,m,q,\chi}$ (respectively, $\mathsf{nf\text{-}SLWE}_{n,m,q,\chi}$), is as follows: given m samples from the LWE distribution $A_{\mathbf{s},\chi}$ for uniformly random $\mathbf{s}$ (resp, $\mathbf{s} \leftarrow_{\$} \chi^n \bmod q$), find $\mathbf{s}$. More formally, for an adversary $\mathcal{A}$, define (for the uniform-secret case)

$$\mathrm{Adv}^{\mathsf{slwe}}_{n,m,q,\chi}(\mathcal{A}) = \Pr\left[\mathcal{A}((\mathbf{a}_i, b_i)_{i=1,\ldots,m}) \Rightarrow \mathbf{s} : \mathbf{s} \leftarrow_{\$} U(\mathbb{Z}_q^n), (\mathbf{a}_i, b_i) \leftarrow_{\$} A_{\mathbf{s},\chi} \text{ for } i = 1, \ldots, m\right] .$$

Similarly, define (for the normal-form case) $\mathrm{Adv}^{\mathsf{nf\text{-}slwe}}_{n,m,q,\chi}(\mathcal{A})$, where $\mathbf{s} \leftarrow_{\$} \chi^n \bmod q$ instead of $\mathbf{s} \leftarrow_{\$} U(\mathbb{Z}_q^n)$.

Definition 2.9 (LWE Decision Problem). Let n, m, q be positive integers, and let χ be a distribution over $\mathbb{Z}$. The *uniform-secret* (respectively, *normal-form*) *learning with errors decision problem* with parameters (n, m, q, χ), denoted $\mathsf{DLWE}_{n,m,q,\chi}$ (respectively, $\mathsf{nf\text{-}DLWE}_{n,m,q,\chi}$), is as follows: distinguish m samples drawn

from the LWE distribution $A_{\mathbf{s},\chi}$ from m samples drawn from the uniform distribution $U(\mathbb{Z}_q^n \times \mathbb{Z}_q)$. More formally, for an adversary $\mathcal{A}$, define (for the uniform-secret case)

$$\mathrm{Adv}^{\mathsf{dlwe}}_{n,m,q,\chi}(\mathcal{A}) = \Big| \Pr\big[\mathcal{A}((\mathbf{a}_i, b_i)_{i=1,\ldots,m}) \Rightarrow 1 : \mathbf{s} \leftarrow\!\!\$\, U(\mathbb{Z}_q^n), (\mathbf{a}_i, b_i) \leftarrow\!\!\$\, A_{\mathbf{s},\chi} \text{ for } i = 1, \ldots, m\big] - \Pr\big[\mathcal{A}((\mathbf{a}_i, b_i)_{i=1,\ldots,m}) \Rightarrow 1 : (\mathbf{a}_i, b_i) \leftarrow\!\!\$\, U(\mathbb{Z}_q^n \times \mathbb{Z}_q) \text{ for } i = 1, \ldots, m\big] \Big| \ .$$

Similarly, define (for the normal-form case) $\mathrm{Adv}^{\mathsf{nf\text{-}dlwe}}_{n,m,q,\chi}(\mathcal{A})$, where $\mathbf{s} \leftarrow\!\!\$\, \chi^n \bmod q$ instead of $\mathbf{s} \leftarrow\!\!\$\, U(\mathbb{Z}_q^n)$.

For all of the above problems, when $\chi = \Psi_{\alpha q}$ is the continuous Gaussian of parameter αq, rounded to the nearest integer (see Definition 2.11 below), we sometimes replace the subscript χ by α.

2.1.4 Gaussians

For any real $s > 0$, the (one-dimensional) *Gaussian function* with parameter (or width) s is the function $\rho_s \colon \mathbb{R} \to \mathbb{R}^+$, defined as

$$\rho_s(\mathbf{x}) := \exp(-\pi \|\mathbf{x}\|^2 / s^2) \ .$$

Definition 2.10 (Gaussian distribution). For any real $s > 0$, the (one-dimensional) *Gaussian distribution* with parameter (or width) s, denoted D_s, is the distribution over $\mathbb{R}$ having probability density function $D_s(x) = \rho_s(x)/s$.

Note that D_s has standard deviation $\sigma = s/\sqrt{2\pi}$.

Definition 2.11 (Rounded Gaussian distribution). For any real $s > 0$, the *rounded Gaussian distribution* with parameter (or width) s, denoted Ψ_s, is the distribution over $\mathbb{Z}$ obtained by rounding a sample from D_s to the nearest integer:

$$\Psi_s(x) = \int_{\{z : \lfloor z \rceil = x\}} D_s(z)\, dz \ . \tag{3}$$

2.1.5 Lattices

Here we recall some background on lattices that will be used when relating LWE to lattice problems.

Definition 2.12 (Lattice). A (full-rank) *n-dimensional lattice* $\mathcal{L}$ is a discrete additive subset of $\mathbb{R}^n$ for which $\mathrm{span}_{\mathbb{R}}(\mathcal{L}) = \mathbb{R}^n$. Any such lattice can be generated by a (non-unique) *basis* $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subset \mathbb{R}^n$ of linearly independent vectors, as

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) := \mathbf{B} \cdot \mathbb{Z}^n = \Big\{ \sum_{i=1}^{n} z_i \cdot \mathbf{b}_i : z_i \in \mathbb{Z} \Big\} \ .$$

The *volume*, or *determinant*, of $\mathcal{L}$ is defined as $\mathrm{vol}(\mathcal{L}) := |\det(\mathbf{B})|$. An *integer lattice* is a lattice that is a subset of $\mathbb{Z}^n$. For an integer q, a *q-ary lattice* is an integer lattice that contains $q\mathbb{Z}^n$.

Definition 2.13 (Minimum distance). For a lattice $\mathcal{L}$, its *minimum distance* is the length (in the Euclidean norm) of a shortest non-zero lattice vector:

$$\lambda_1(\mathcal{L}) = \min_{\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\mathbf{v}\| \ .$$

More generally, its *ith successive minimum* $\lambda_i(\mathcal{L})$ is the smallest real $r > 0$ such that $\mathcal{L}$ has i linearly independent vectors of length at most r.

Definition 2.14 (Discrete Gaussian). For a lattice $\mathcal{L} \subset \mathbb{R}^n$, the *discrete Gaussian distribution* over $\mathcal{L}$ with parameter s, denoted $D_{\mathcal{L},s}$, is defined as $D_s(\mathbf{x}) = \rho_s(\mathbf{x})/\rho_s(\mathcal{L})$ for $\mathbf{x} \in \mathcal{L}$ (and $D_s(\mathbf{x}) = 0$ otherwise), where $\rho_s(\mathcal{L}) = \sum_{\mathbf{v} \in \mathcal{L}} \rho_s(\mathbf{v})$ is a normalization factor.

We now recall various computational problems on lattices. We stress that these are *worst-case* problems, i.e., to solve such a problem an algorithm must succeed on *every* input. The following two problems are parameterized by an *approximation factor* $\gamma = \gamma(n)$, which is a function of the lattice dimension n.

Definition 2.15 (Decisional approximate shortest vector problem (GapSVP_γ)). Given a basis $\mathbf{B}$ of an n-dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, where $\lambda_1(\mathcal{L}) \leq 1$ or $\lambda_1(\mathcal{L}) > \gamma(n)$, determine which is the case.

Definition 2.16 (Approximate shortest independent vectors problem (SIVP_γ)). Given a basis $\mathbf{B}$ of an n-dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, output a set $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\} \subset \mathcal{L}$ of n linearly independent lattice vectors where $\|\mathbf{v}_i\| \leq \gamma(n) \cdot \lambda_n(\mathcal{L})$ for all i.

The following problem is parameterized by a function φ from lattices to positive real numbers.

Definition 2.17 (Discrete Gaussian Sampling (DGS_φ)). Given a basis $\mathbf{B}$ of an n-dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$ and a real number $s \geq \varphi(L)$, output a sample from the discrete Gaussian distribution $D_{L,s}$.

2.2 Algorithm description

This section specifies the algorithms comprising the FrodoKEM key encapsulation mechanism. FrodoKEM is built from a public key encryption scheme, FrodoPKE, as well as several other components.

Notation. The algorithms in this document are described in terms of the following parameters:

- χ, a probability distribution on $\mathbb{Z}$;
- $q = 2^D$, a power-of-two integer modulus with exponent $D \leq 16$;
- $n, \overline{m}, \overline{n}$, integer matrix dimensions with $n \equiv 0 \pmod 8$;
- $B \leq D$, the number of bits encoded in each matrix entry;
- $\ell = B \cdot \overline{m} \cdot \overline{n}$, the length of bit strings that are encoded as $\overline{m}$-by-$\overline{n}$ matrices;
- $\mathsf{len}_\mathbf{A}$, the bit length of seeds used for pseudorandom matrix generation;
- $\mathsf{len}_\mathbf{E}$, the bit length of seeds used for pseudorandom bit generation for error sampling.

Additional parameters for specific algorithms accompany the algorithm description.

2.2.1 Matrix encoding of bit strings

This subsection describes how bit strings are encoded as mod-q integer matrices. Recall that $2^B \leq q$. The encoding function $\mathrm{ec}(\cdot)$ encodes an integer $0 \leq k < 2^B$ as an element in $\mathbb{Z}_q$ by multiplying it by $q/2^B = 2^{D-B}$:

$$\mathrm{ec}(k) := k \cdot q/2^B \ .$$

Using this function, the function Frodo.Encode encodes bit strings of length $\ell = B \cdot \overline{m} \cdot \overline{n}$ as $\overline{m}$-by-$\overline{n}$-matrices with entries in $\mathbb{Z}_q$ by applying $\mathrm{ec}(\cdot)$ to B-bit sub-strings sequentially and filling the matrix row by row entry-wise. The function Frodo.Encode is shown in Algorithm 1. Each B-bit sub-string is interpreted as an integer $0 \leq k < 2^B$ and then encoded by $\mathrm{ec}(k)$, which means that B-bit values are placed into the B most significant bits of the corresponding entry modulo q.

The corresponding decoding function Frodo.Decode is defined as shown in Algorithm 2. It decodes the $\overline{m}$-by-$\overline{n}$ matrix $\mathbf{K}$ into a bit string of length $\ell = B \cdot \overline{m} \cdot \overline{n}$. It extracts B bits from each entry by applying the function $\mathrm{dc}(\cdot)$:

$$\mathrm{dc}(c) = \lfloor c \cdot 2^B/q \rceil \bmod 2^B \ .$$

That is, the $\mathbb{Z}_q$-entry is interpreted as an integer, then divided by $q/2^B$ and rounded. This amounts to rounding to the B most significant bits of each entry. With these definitions, it is the case that $\mathrm{dc}(\mathrm{ec}(k)) = k$ for all $0 \leq k < 2^B$.

2.2.2 Packing matrices modulo q

This section specifies packing and unpacking algorithms to transform matrices with entries in $\mathbb{Z}_q$ to bit strings and vice versa. The algorithm Frodo.Pack packs a matrix into a bit string by simply concatenating the D-bit matrix coefficients, as shown in Algorithm 3. Note that in the software implementation, the resulting bit string is stored as a byte array, padding with zero bits to make the length a multiple of 8. The reverse operation Frodo.Unpack is shown in Algorithm 4.

Algorithm 1 Frodo.Encode

Input: Bit string $\mathbf{k} \in \{0,1\}^{\ell}$, $\ell = B \cdot \overline{m} \cdot \overline{n}$.
Output: Matrix $\mathbf{K} \in \mathbb{Z}_q^{\overline{m} \times \overline{n}}$.

1: **for** $(i = 0; i < \overline{m}; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < \overline{n}; j \leftarrow j + 1)$ **do**
3: $k \leftarrow \sum_{l=0}^{B-1} \mathbf{k}_{(i \cdot \overline{n} + j)B + l} \cdot 2^l$
4: $\mathbf{K}_{i,j} \leftarrow \mathrm{ec}(k) = k \cdot q / 2^B$
5: **return** $\mathbf{K} = (\mathbf{K}_{i,j})_{0 \le i < \overline{m}, 0 \le j < \overline{n}}$

Algorithm 2 Frodo.Decode

Input: Matrix $\mathbf{K} \in \mathbb{Z}_q^{\overline{m} \times \overline{n}}$.
Output: Bit string $\mathbf{k} \in \{0,1\}^{\ell}$, $\ell = B \cdot \overline{m} \cdot \overline{n}$.

1: **for** $(i = 0; i < \overline{m}; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < \overline{n}; j \leftarrow j + 1)$ **do**
3: $k \leftarrow \mathrm{dc}(\mathbf{K}_{i,j}) = \lfloor \mathbf{K}_{i,j} \cdot 2^B / q \rceil \bmod 2^B$
4: $k = \sum_{l=0}^{B-1} k_l \cdot 2^l$ where $k_l \in \{0,1\}$
5: **for** $(l = 0; l < B; l \leftarrow l + 1)$ **do**
6: $\mathbf{k}_{(i \cdot \overline{n} + j)B + l} \leftarrow k_l$
7: **return** $\mathbf{k}$

Algorithm 3 Frodo.Pack

Input: Matrix $\mathbf{C} \in \mathbb{Z}_q^{n_1 \times n_2}$.
Output: Bit string $\mathbf{b} \in \{0,1\}^{D \cdot n_1 \cdot n_2}$.

1: **for** $(i = 0; i < n_1; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < n_2; j \leftarrow j + 1)$ **do**
3: $\mathbf{C}_{i,j} = \sum_{l=0}^{D-1} c_l \cdot 2^l$ where $c_l \in \{0,1\}$
4: **for** $(l = 0; l < D; l \leftarrow l + 1)$ **do**
5: $\mathbf{b}_{(i \cdot n_2 + j)D + l} \leftarrow c_{D-1-l}$
6: **return** $\mathbf{b}$

Algorithm 4 Frodo.Unpack

Input: Bit string $\mathbf{b} \in \{0,1\}^{D \cdot n_1 \cdot n_2}$, n_1, n_2.
Output: Matrix $\mathbf{C} \in \mathbb{Z}_q^{n_1 \times n_2}$.

1: **for** $(i = 0; i < n_1; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < n_2; j \leftarrow j + 1)$ **do**
3: $\mathbf{C}_{i,j} \leftarrow \sum_{l=0}^{D-1} \mathbf{b}_{(i \cdot n_2 + j)D + l} \cdot 2^{D-1-l}$
4: **return** $\mathbf{C}$

2.2.3 Deterministic random bit generation

FrodoKEM requires the deterministic generation of random bit sequences from a random seed value. This is done using the SHA-3-derived extendable output function cSHAKE [64]. The function cSHAKE is taken as either cSHAKE128 or cSHAKE256 (indicated below for each parameter set of FrodoKEM), and takes as input a bit string X, a requested output bit length L, and a 16-bit customization value c as a domain separator. Interpreting c as a 16-bit integer, it is converted to an array of two bytes $[c_0, c_1]$, where $c = c_0 + 2^8 \cdot c_1$. The call to the function on these inputs is written $\mathrm{cSHAKE}(X, L, c)$, and returns a string of L bits as output.

2.2.4 Sampling from the error distribution

The error distribution χ used in FrodoKEM is a discrete, symmetric distribution on $\mathbb{Z}$, centered at zero and with small support, which approximates a rounded continuous Gaussian distribution.

The support of χ is $S_\chi = \{-s, -s+1, \ldots, -1, 0, 1, \ldots, s-1, s\}$ for a positive integer s. The probabilities $\chi(z) = \chi(-z)$ for $z \in S_\chi$ are given by a discrete probability density function, which is described by a table

$$T_\chi = (T_\chi(0), T_\chi(1), \ldots, T_\chi(s))$$

of $s+1$ positive integers related to the cumulative distribution function. For a certain positive integer len_χ, the table entries satisfy the following conditions:

$$T_\chi(0)/2^{\mathsf{len}_\chi} = \chi(0)/2 \qquad \text{and} \qquad T_\chi(z)/2^{\mathsf{len}_\chi} = \chi(0)/2 + \sum_{i=1}^{z} \chi(i) \text{ for } 1 \le z \le s \ .$$

Therefore, $T_\chi(s) = 2^{\mathsf{len}_\chi - 1}$.

Sampling from χ via inversion sampling is done as shown in Algorithm 5. Given a string of len_χ uniformly random bits $\mathbf{r} \in \{0,1\}^{\mathsf{len}_\chi}$ and a distribution table T_χ, the algorithm Frodo.Sample returns a sample e from the distribution χ. We emphasize that it is important to perform this sampling in constant time to avoid exposing timing side-channels, which is why Step 3 of the algorithm does a complete loop through the entire table T_χ. Note also that the comparison in Step 4 needs to be implemented in a constant-time manner.

Algorithm 5 Frodo.Sample

Input: A (random) bit string $\mathbf{r} = (\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{\mathsf{len}_\chi - 1}) \in \{0,1\}^{\mathsf{len}_\chi}$, the table $T_\chi = (T_\chi(0), T_\chi(1), \ldots, T_\chi(s))$.
Output: A sample $e \in \mathbb{Z}$.

$t \leftarrow \sum_{i=1}^{\mathsf{len}_\chi - 1} \mathbf{r}_i \cdot 2^{i-1}$
$e \leftarrow 0$
for $(z = 0;\ z < s;\ z \leftarrow z + 1)$ **do**
  if $t > T_\chi(z)$ **then**
    $e \leftarrow e + 1$
$e \leftarrow (-1)^{\mathbf{r}_0} \cdot e$
return e

An n_1-by-n_2 matrix of $n_1 n_2$ samples from the error distribution is sampled by expanding a random seed to an $(n_1 n_2 \cdot \mathsf{len}_\chi)$-bit string, here written as a sequence of $n_1 n_2$ bit vectors $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(n_1 n_2 - 1)})$ of length len_χ, using the extendable output function cSHAKE, and then sampling $n_1 n_2$ error terms by calling Frodo.Sample on a corresponding len_χ-bit substring $\mathbf{r}^{(i \cdot n_2 + j)}$ and the distribution table T_χ to sample the matrix entry $\mathbf{E}_{i,j}$. The algorithm Frodo.SampleMatrix is shown in Algorithm 6.

Algorithm 6 Frodo.SampleMatrix

Input: Seed $\mathsf{seed}_\mathbf{E} \in \{0,1\}^{\mathsf{len}_\mathbf{E}}$, dimensions n_1, n_2, the table T_χ, domain separator value c.
Output: A sample $\mathbf{E} \in \mathbb{Z}^{n_1 \times n_2}$.

$(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(n_1 n_2 - 1)}) \leftarrow \text{cSHAKE}(\mathsf{seed}_\mathbf{E}, n_1 n_2 \cdot \mathsf{len}_\chi, c)$ (here, each $\mathbf{r}^{(i)}$ is a vector of len_χ bits)
for $(i = 0;\ i < n_1;\ i \leftarrow i + 1)$ **do**
  for $(j = 0;\ j < n_2;\ j \leftarrow j + 1)$ **do**
    $\mathbf{E}_{i,j} \leftarrow \mathsf{Frodo.Sample}(\mathbf{r}^{(i \cdot n_2 + j)}, T_\chi)$
return $\mathbf{E}$

2.2.5 Pseudorandom matrix generation

The algorithm Frodo.Gen takes as input a seed $\mathsf{seed}_\mathbf{A} \in \{0,1\}^{\mathsf{len}_\mathbf{A}}$ and a dimension $n \in \mathbb{Z}$, and outputs a pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. There are two options for instantiating Frodo.Gen. The first one uses AES128 and is shown in Algorithm 7; the second uses cSHAKE128 and is shown in Algorithm 8.

Using AES128**.** Algorithm 7 generates a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ as follows. For each row index $i = 0, 1, \ldots, n-1$ and column index $j = 0, 8, \ldots, n-8$, the algorithm generates a 128-bit block, which it uses to set the matrix entries $\mathbf{A}_{i,j}, \mathbf{A}_{i,j+1}, \ldots, \mathbf{A}_{i,j+7}$ as follows. It applies AES128 with key $\mathsf{seed}_\mathbf{A}$ to the input block $\langle i \rangle \| \langle j \rangle \| 0 \cdots 0 \in \{0,1\}^{128}$, where i, j are encoded as 16-bit strings. It then splits the 128-bit AES output block into eight 16-bit strings, which it interprets as nonnegative integers $c_{i,j+k}$ for $k = 0, 1, \ldots, 7$. Finally, it sets $\mathbf{A}_{i,j+k} = c_{i,j+k} \bmod q$ for all k.

Algorithm 7 Frodo.Gen using AES128

Input: Seed $\mathsf{seed}_\mathbf{A} \in \{0,1\}^{\mathsf{len}_\mathbf{A}}$.
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

for $(i = 0;\ i < n;\ i \leftarrow i + 1)$ **do**
  for $(j = 0;\ j < n;\ j \leftarrow j + 8)$ **do**
    $\mathbf{b} \leftarrow \langle i \rangle \| \langle j \rangle \| 0 \cdots 0 \in \{0,1\}^{128}$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^{16}$
    $\langle c_{i,j} \rangle \| \langle c_{i,j+1} \rangle \| \cdots \| \langle c_{i,j+7} \rangle \leftarrow \text{AES128}_{\mathsf{seed}_\mathbf{A}}(\mathbf{b})$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$.
    for $(k = 0;\ k < 8;\ k \leftarrow k + 1)$ **do**
      $\mathbf{A}_{i,j+k} \leftarrow c_{i,j+k} \bmod q$
return $\mathbf{A}$

Using cSHAKE128**.** Algorithm 8 generates a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ as follows. For each row index $i = 0, 1, \ldots, n-1$, it calls cSHAKE128 with a main input of $\mathsf{seed}_{\mathbf{A}}$ and customization value $2^8 + i$ to produce a $16n$-bit output string. It splits this output into 16-bit integers $c_{i,j}$ for $j = 0, 1, \ldots, n-1$, and sets $\mathbf{A}_{i,j} = c_{i,j} \bmod q$ for all j. Note that the offset of 2^8 in the customization value is used for domain separation between the generation of $\mathbf{A}$ and other uses of cSHAKE128 in the key encapsulation functions below, which have customization values smaller than 2^8.

Algorithm 8 Frodo.Gen using cSHAKE128

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathbf{A}}}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

1: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
2: $\quad \langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow \text{cSHAKE128}(\mathsf{seed}_{\mathbf{A}}, 16n, 2^8 + i)$ where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$.
3: $\quad$ **for** $(j = 0; j < n; j \leftarrow j + 1)$ **do**
4: $\quad\quad \mathbf{A}_{i,j} \leftarrow c_{i,j} \bmod q$
5: **return** $\mathbf{A}$

2.2.6 FrodoPKE: IND-CPA-secure public key encryption scheme

This section describes FrodoPKE, a public-key encryption scheme with fixed-length message space, targeting IND-CPA security, that will be used as a building block for FrodoKEM. FrodoPKE is based on the public-key encryption scheme presented by Lindner and Peikert in [75], with the following adaptations and specializations:

- The matrix $\mathbf{A}$ is generated from a seed using the function Gen specified in Section 2.2.5.
- Several ($\overline{m}$) ciphertexts are generated at once.
- The same Gaussian-derived error distribution is used for both key generation and encryption.

The PKE scheme is given by three algorithms (FrodoPKE.KeyGen, FrodoPKE.Enc, FrodoPKE.Dec), defined respectively in Algorithm 9, Algorithm 10, and Algorithm 11. FrodoPKE is parameterized by the following parameters:

- $q = 2^D$, a power-of-two integer modulus with exponent $D \leq 16$;
- $n, \overline{m}, \overline{n}$, integer matrix dimensions with $n \equiv 0 \pmod 8$;
- $B \leq D$, the number of bits encoded in each matrix entry;
- $\ell = B \cdot \overline{m} \cdot \overline{n}$, the length of bit strings that are encoded as $\overline{m}$-by-$\overline{n}$ matrices;
- $\mathsf{len}_{\mu} = \ell$, the bit length of messages;
- $\mathcal{M} = \{0,1\}^{\mathsf{len}_{\mu}}$, the message space;
- $\mathsf{len}_{\mathbf{A}}$, the bit length of seeds used for pseudorandom matrix generation;
- $\mathsf{len}_{\mathbf{E}}$, the bit length of seeds used for pseudorandom bit generation for error sampling;
- Gen, the matrix-generation algorithm, either Algorithm 7 or Algorithm 8;
- T_{χ}, the distribution table for sampling.

In the notation of [75], their n_1 and n_2 both equal n here, and their dimension ℓ is $\overline{n}$ here.

Algorithm 9 FrodoPKE.KeyGen.

Input: None.
Output: Key pair $(pk, sk) \in (\{0,1\}^{\mathsf{len}_{\mathbf{A}}} \times \mathbb{Z}_q^{n \times \overline{n}}) \times \mathbb{Z}_q^{n \times \overline{n}}$.

1: Choose a uniformly random seed $\mathsf{seed}_{\mathbf{A}} \leftarrow\!\!\$\ U(\{0,1\}^{\mathsf{len}_{\mathbf{A}}})$
2: Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
3: Choose a uniformly random seed $\mathsf{seed}_{\mathbf{E}} \leftarrow\!\!\$\ U(\{0,1\}^{\mathsf{len}_{\mathbf{E}}})$
4: Sample error matrix $\mathbf{S} \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, n, \overline{n}, T_{\chi}, 1)$
5: Sample error matrix $\mathbf{E} \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, n, \overline{n}, T_{\chi}, 2)$
6: Compute $\mathbf{B} = \mathbf{AS} + \mathbf{E}$
7: **return** public key $pk \leftarrow (\mathsf{seed}_{\mathbf{A}}, \mathbf{B})$ and secret key $sk \leftarrow \mathbf{S}$

Algorithm 10 FrodoPKE.Enc.

Input: Message $\mu \in \mathcal{M}$ and public key $pk = (\mathsf{seed}_{\mathbf{A}}, \mathbf{B}) \in \{0,1\}^{\mathsf{len}_{\mathbf{A}}} \times \mathbb{Z}_q^{n \times \overline{n}}$.
Output: Ciphertext $c = (\mathbf{C}_1, \mathbf{C}_2) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}}$.

1: Generate $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
2: Choose a uniformly random seed $\mathsf{seed}_{\mathbf{E}} \leftarrow\!\!\$\ U(\{0,1\}^{\mathsf{len}_{\mathbf{E}}})$
3: Sample error matrix $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, \overline{m}, n, T_\chi, 4)$
4: Sample error matrix $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, \overline{m}, n, T_\chi, 5)$
5: Sample error matrix $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, \overline{m}, \overline{n}, T_\chi, 6)$
6: Compute $\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'$ and $\mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E}''$
7: **return** ciphertext $c \leftarrow (\mathbf{C}_1, \mathbf{C}_2) = (\mathbf{B}', \mathbf{V} + \mathsf{Frodo.Encode}(\mu))$

Algorithm 11 FrodoPKE.Dec.

Input: Ciphertext $c = (\mathbf{C}_1, \mathbf{C}_2) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}}$ and secret key $sk = \mathbf{S} \in \mathbb{Z}_q^{n \times \overline{n}}$.
Output: Decrypted message $\mu' \in \mathcal{M}$.

1: Compute $\mathbf{M} = \mathbf{C}_2 - \mathbf{C}_1\mathbf{S}$
2: **return** message $\mu' \leftarrow \mathsf{Frodo.Decode}(\mathbf{M})$

2.2.7 Correctness of IND-CPA PKE

The next lemma states bounds on the size of errors that can be handled by the decoding algorithm.

Lemma 2.18. *Let $q = 2^D$, $B \leq D$. Then* $\mathrm{dc}(\mathrm{ec}(k) + e) = k$ *for any $k, e \in \mathbb{Z}$ such that $0 \leq k < 2^B$ and $-q/2^{B+1} \leq e < q/2^{B+1}$.*

Proof. This follows directly from the fact that $\mathrm{dc}(\mathrm{ec}(k) + e) = \lfloor k + e2^B/q \rceil \bmod 2^B$. □

Correctness of decryption: The decryption algorithm FrodoPKE.Dec computes

$$\begin{aligned}
\mathbf{M} &= \mathbf{C}_2 - \mathbf{C}_1\mathbf{S} \\
&= \mathbf{V} + \mathsf{Frodo.Encode}(\mu) - (\mathbf{S}'\mathbf{A} + \mathbf{E}')\mathbf{S} \\
&= \mathsf{Frodo.Encode}(\mu) + \mathbf{S}'\mathbf{B} + \mathbf{E}'' - \mathbf{S}'\mathbf{A}\mathbf{S} - \mathbf{E}'\mathbf{S} \\
&= \mathsf{Frodo.Encode}(\mu) + \mathbf{S}'\mathbf{A}\mathbf{S} + \mathbf{S}'\mathbf{E} + \mathbf{E}'' - \mathbf{S}'\mathbf{A}\mathbf{S} - \mathbf{E}'\mathbf{S} \\
&= \mathsf{Frodo.Encode}(\mu) + \mathbf{S}'\mathbf{E} + \mathbf{E}'' - \mathbf{E}'\mathbf{S} \\
&= \mathsf{Frodo.Encode}(\mu) + \mathbf{E}'''
\end{aligned}$$

for some error matrix $\mathbf{E}''' = \mathbf{S}'\mathbf{E} + \mathbf{E}'' - \mathbf{E}'\mathbf{S}$. Therefore, any B-bit substring of the message μ corresponding to an entry of $\mathbf{M}$ will be decrypted correctly if the condition in Lemma 2.18 is satisfied for the corresponding entry of $\mathbf{E}'''$.

Failure probability. Each entry in the matrix $\mathbf{E}'''$ is the sum of $2n$ products of two independent samples from χ, and one more independent sample from χ. Denote the distribution of this sum by χ'. In the case of a power-of-2 modulus q, the probability of decryption failure for any single symbol is therefore the sum

$$p = \sum_{e \notin [-q/2^{B+1}, q/2^{B+1})} \chi'(e) \ .$$

The probability of decryption failure for the entire message can then be obtained using the union bound.

For the distributions χ we use, which have rather small support S_χ, the distribution χ' can be efficiently computed exactly. The probability that a product of two independent samples from χ equals e (modulo q) is simply

$$\sum_{(a,b) \in S_\chi \times S_\chi \,:\, ab = e \bmod q} \chi(a) \cdot \chi(b) \ .$$

Similarly, the probability that the sum of two entries assumes a certain value is given by the standard convolution sum. Section 2.4.3 reports the failure probability for each of the selected parameter sets.

2.2.8 Transform from IND-CPA PKE to IND-CCA KEM

The Fujisaki–Okamoto transform [51] constructs an IND-CCA2-secure public key encryption scheme from a one-way-secure public key encryption scheme in the classical random oracle model (with an assumption on the distribution of ciphertexts for each plaintext being sufficiently close to uniform). Targhi and Unruh [111] gave a variant of the Fujisaki–Okamoto transform and showed its IND-CCA2 security against a quantum adversary in the quantum random oracle model under similar assumptions. The results of both FO and TU proceed under the assumption that the public key encryption scheme has perfect correctness, which is not the case for lattice-based schemes. Hofheinz, Hövelmanns, and Kiltz [61] gave a variety of constructions in a modular fashion. We apply their $\mathsf{QFO}_m^{\not\perp}$ ("Quantum FO with implicit rejection and K being a hash of m") transform which constructs an IND-CCA-secure key encapsulation mechanism from an IND-CPA public key encryption scheme and three hash functions; following [23], we make the following modifications (see Figure 3 for notation), denoting the resulting transform $\mathsf{QFO}_m^{\not\perp\prime}$:

- A single hash function (with longer output) is used to compute $\mathbf{r}$, $\mathbf{k}$, and $\mathbf{d}$.
- The computation of $\mathbf{r}$, $\mathbf{k}$, and $\mathbf{d}$ also takes the public key pk as input.
- The computation of the shared secret $\mathbf{ss}$ also takes the encapsulation c as input.

Definition 2.19 ($\mathsf{QFO}_m^{\not\perp\prime}$ transform). Let $\mathsf{PKE} = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$ be a public key encryption scheme with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$, where the randomness space of Enc is $\mathcal{R}$. Let $\mathsf{len_s}, \mathsf{len_k}, \mathsf{len_d}, \mathsf{len_{ss}}$ be parameters. Let $G : \{0,1\}^* \to \mathcal{R} \times \{0,1\}^{\mathsf{len_k}} \times \{0,1\}^{\mathsf{len_d}}$ and $F : \{0,1\}^* \to \{0,1\}^{\mathsf{len_{ss}}}$ be hash functions. Define $\mathsf{QKEM}_m^{\not\perp\prime} = \mathsf{QFO}_m^{\not\perp\prime}[\mathsf{PKE}, G, F]$ be the key encapsulation mechanism with $\mathsf{QKEM}_m^{\not\perp\prime}$.KeyGen, $\mathsf{QKEM}_m^{\not\perp\prime}$.Encaps and $\mathsf{QKEM}_m^{\not\perp\prime}$.Decaps as shown in Figure 3.

$\underline{\mathsf{QKEM}_m^{\not\perp\prime}.\mathrm{KeyGen}():}$
1: $(pk, sk) \leftarrow_{\$} \mathsf{PKE.KeyGen}()$
2: $\mathbf{s} \leftarrow_{\$} \{0,1\}^{\mathsf{len_s}}$
3: $sk' \leftarrow (sk, \mathbf{s})$
4: **return** (pk, sk')

$\underline{\mathsf{QKEM}_m^{\not\perp\prime}.\mathrm{Encaps}(pk):}$
1: $\mu \leftarrow_{\$} \mathcal{M}$
2: $(\mathbf{r}, \mathbf{k}, \mathbf{d}) \leftarrow G(pk \| \mu)$
3: $c \leftarrow \mathsf{PKE.Enc}(\mu, pk; \mathbf{r})$
4: $\mathbf{ss} \leftarrow F(c \| \mathbf{k} \| \mathbf{d})$
5: **return** $((c, \mathbf{d}), \mathbf{ss})$

$\underline{\mathsf{QKEM}_m^{\not\perp\prime}.\mathrm{Decaps}((c, \mathbf{d}), (sk, \mathbf{s}, pk)):}$
1: $\mu' \leftarrow \mathsf{PKE.Dec}(c, sk)$
2: $(\mathbf{r}', \mathbf{k}', \mathbf{d}') \leftarrow G(pk \| \mu')$
3: **if** $c = \mathsf{PKE.Enc}(\mu', pk; \mathbf{r}')$ **and** $\mathbf{d} = \mathbf{d}'$ **then**
4: **return** $\mathbf{ss}' \leftarrow F(c \| \mathbf{k}' \| \mathbf{d})$
5: **else**
6: **return** $\mathbf{ss}' \leftarrow F(c \| \mathbf{s} \| \mathbf{d})$

Figure 3: Construction of an IND-CCA-secure key encapsulation mechanism $\mathsf{QKEM}_m^{\not\perp\prime} = \mathsf{QFO}_m^{\not\perp\prime}[\mathsf{PKE}, G, F]$ from a public key encryption scheme PKE and hash functions G and F.

Remark. In October and November 2017, Jiang et al. posted an eprint giving a proof for two FO-like transforms that yield an IND-CCA-secure KEM in the quantum random oracle model and do not require the extra hash value d introduced by Targhi and Unruh [63]. As of this writing, it is too early to know if the results in the Jiang et al. eprint are correct. If they are, then we can remove the d value in the $\mathsf{QKEM}_m^{\not\perp\prime}$ transform of Figure 3, and from the resulting FrodoKEM in Algorithm 13 and Algorithm 14. The transform becomes sufficiently similar to Jiang et al.'s FO-I transform to have its security implied by their result; the differences are: the computation of r and K also take the public key pk as input, and the value random coins μ are hashed before they are input to the hash function that computes the shared secret.

2.2.9 FrodoKEM: IND-CCA-secure key encapsulation mechanism

This section describes FrodoKEM, a key encapsulation mechanism that is derived from FrodoPKE by applying the $\mathsf{QFO}_m^{\not\perp\prime}$ transform. FrodoKEM is parameterized by the following parameters:

- $q = 2^D$, a power-of-two integer modulus with exponent $D \leq 16$;
- $n, \overline{m}, \overline{n}$, integer matrix dimensions with $n \equiv 0 \pmod 8$;
- $B \leq D$, the number of bits encoded in each matrix entry;
- $\ell = B \cdot \overline{m} \cdot \overline{n}$, the length of bit strings to be encoded in an $\overline{m}$-by-$\overline{n}$ matrix;
- $\mathsf{len}_\mu = \ell$, the bit length of messages;
- $\mathcal{M} = \{0,1\}^{\mathsf{len}_\mu}$, the message space;
- $\mathsf{len}_{\mathbf{A}}$, the bit length of seeds used for pseudorandom matrix generation;
- $\mathsf{len}_{\mathbf{E}}$, the bit length of seeds used for pseudorandom bit generation for error sampling;
- Gen, pseudorandom matrix generation algorithm, either Algorithm 7 or Algorithm 8;
- T_χ, distribution table for sampling;
- $\mathsf{len}_{\mathbf{s}}$, the length of the bit vector $\mathbf{s}$ used for pseudorandom shared secret generation in the event of decapsulation failure in the $\mathsf{QFO}_m^{\not\perp\prime}$ transform;
- $\mathsf{len}_{\mathbf{z}}$, the bit length of seeds used for pseudorandom generation of $\mathsf{seed}_{\mathbf{A}}$;
- $\mathsf{len}_{\mathbf{k}}$, the bit length of intermediate shared secret $\mathbf{k}$ in the $\mathsf{QFO}_m^{\not\perp\prime}$ transform;
- $\mathsf{len}_{\mathbf{d}}$, the bit length of check value $\mathbf{d}$ in the $\mathsf{QFO}_m^{\not\perp\prime}$ transform;
- $\mathsf{len}_{\mathbf{ss}}$, the bit length of shared secret $\mathbf{ss}$ in the $\mathsf{QFO}_m^{\not\perp\prime}$ transform;
- $H : \{0,1\}^* \to \{0,1\}^{\mathsf{len}_{\mathbf{A}}}$, a hash function, either cSHAKE128$(\cdot, \mathsf{len}_{\mathbf{A}}, 0)$ or cSHAKE256$(\cdot, \mathsf{len}_{\mathbf{A}}, 0)$;
- $G : \{0,1\}^* \to \{0,1\}^{\mathsf{len}_{\mathbf{E}}} \times \{0,1\}^{\mathsf{len}_{\mathbf{k}}} \times \{0,1\}^{\mathsf{len}_{\mathbf{d}}}$, a hash function, either cSHAKE128$(\cdot, \mathsf{len}_{\mathbf{E}} + \mathsf{len}_{\mathbf{k}} + \mathsf{len}_{\mathbf{d}}, 3)$ or cSHAKE256$(\cdot, \mathsf{len}_{\mathbf{E}} + \mathsf{len}_{\mathbf{k}} + \mathsf{len}_{\mathbf{d}}, 3)$;
- $F : \{0,1\}^* \to \{0,1\}^{\mathsf{len}_{\mathbf{ss}}}$, a hash function, either cSHAKE128$(\cdot, \mathsf{len}_{\mathbf{ss}}, 7)$ or cSHAKE256$(\cdot, \mathsf{len}_{\mathbf{ss}}, 7)$.

Algorithm 12 FrodoKEM.KeyGen.

Input: None.
Output: Key pair (pk, sk') with $pk \in \{0,1\}^{\mathsf{len}_{\mathbf{A}} + D \cdot n \cdot \overline{n}}$, $sk' \in \{0,1\}^{\mathsf{len}_{\mathbf{s}} + \mathsf{len}_{\mathbf{A}} + D \cdot n \cdot \overline{n}} \times \mathbb{Z}_q^{n \times \overline{n}}$.

1: Choose uniformly random seeds $\mathbf{s} \| \mathsf{seed}_{\mathbf{E}} \| \mathbf{z} \leftarrow\!\!\$\ U(\{0,1\}^{\mathsf{len}_{\mathbf{s}} + \mathsf{len}_{\mathbf{E}} + \mathsf{len}_{\mathbf{z}}}$
2: Generate pseudorandom seed $\mathsf{seed}_{\mathbf{A}} \leftarrow H(\mathbf{z})$
3: Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
4: Sample error matrix $\mathbf{S} \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, n, \overline{n}, T_\chi, 1)$
5: Sample error matrix $\mathbf{E} \leftarrow \mathsf{Frodo.SampleMatrix}(\mathsf{seed}_{\mathbf{E}}, n, \overline{n}, T_\chi, 2)$
6: Compute $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
7: Compute $\mathbf{b} \leftarrow \mathsf{Frodo.Pack}(\mathbf{B})$
8: **return** public key $pk \leftarrow \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}$ and secret key $sk' \leftarrow (\mathbf{s} \| \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}, \mathbf{S})$

Algorithm 13 FrodoKEM.Encaps.

Input: Public key $pk = \mathsf{seed}_{\mathbf{A}} \| \mathbf{b} \in \{0,1\}^{\mathsf{len}_{\mathbf{A}} + D \cdot n \cdot \overline{n}}$.
Output: Ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{d} \in \{0,1\}^{(\overline{m} \cdot n + \overline{m} \cdot \overline{n}) D + \mathsf{len}_{\mathbf{d}}}$ and shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathbf{ss}}}$.

1: Choose a uniformly random key $\mu \leftarrow_{\$} U(\{0,1\}^{\mathsf{len}_{\mu}}$
2: Generate pseudorandom values $\mathsf{seed}_{\mathbf{E}} \| \mathbf{k} \| \mathbf{d} \leftarrow G(pk \| \mu)$
3: Sample error matrix $\mathbf{S}' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}_{\mathbf{E}}, \overline{m}, n, T_\chi, 4)$
4: Sample error matrix $\mathbf{E}' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}_{\mathbf{E}}, \overline{m}, n, T_\chi, 5)$
5: Generate $\mathbf{A} \leftarrow$ Frodo.Gen$(\mathsf{seed}_{\mathbf{A}})$
6: Compute $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
7: Compute $\mathbf{c}_1 \leftarrow$ Frodo.Pack$(\mathbf{B}')$
8: Sample error matrix $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}_{\mathbf{E}}, \overline{m}, \overline{n}, T_\chi, 6)$
9: Compute $\mathbf{B} \leftarrow$ Frodo.Unpack$(\mathbf{b}, n, \overline{n})$
10: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
11: Compute $\mathbf{C} \leftarrow \mathbf{V} +$ Frodo.Encode(μ)
12: Compute $\mathbf{c}_2 \leftarrow$ Frodo.Pack$(\mathbf{C})$
13: Compute $\mathbf{ss} \leftarrow F(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{k} \| \mathbf{d})$
14: **return** ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{d}$ and shared secret $\mathbf{ss}$

Algorithm 14 FrodoKEM.Decaps.

Input: Ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{d} \in \{0,1\}^{(\overline{m} \cdot n + \overline{m} \cdot \overline{n}) D + \mathsf{len}_{\mathbf{d}}}$, secret key $sk' = (\mathbf{s} \| \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}, \mathbf{S}) \in \{0,1\}^{\mathsf{len}_{\mathbf{s}} + \mathsf{len}_{\mathbf{A}} + D \cdot n \cdot \overline{n}} \times \mathbb{Z}_q^{n \times \overline{n}}$.
Output: Shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathbf{ss}}}$.

1: $\mathbf{B}' \leftarrow$ Frodo.Unpack$(\mathbf{c}_1)$
2: $\mathbf{C} \leftarrow$ Frodo.Unpack$(\mathbf{c}_2)$
3: Compute $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
4: Compute $\mu' \leftarrow$ Frodo.Decode$(\mathbf{M})$
5: Parse $pk \leftarrow \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}$
6: Generate pseudorandom values $\mathsf{seed}'_{\mathbf{E}} \| \mathbf{k}' \| \mathbf{d}' \leftarrow G(pk \| \mu')$
7: Sample error matrix $\mathbf{S}' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}'_{\mathbf{E}}, \overline{m}, n, T_\chi, 4)$
8: Sample error matrix $\mathbf{E}' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}'_{\mathbf{E}}, \overline{m}, n, T_\chi, 5)$
9: Generate $\mathbf{A} \leftarrow$ Frodo.Gen$(\mathsf{seed}_{\mathbf{A}})$
10: Compute $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
11: Sample error matrix $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix$(\mathsf{seed}'_{\mathbf{E}}, \overline{m}, \overline{n}, T_\chi, 6)$
12: Compute $\mathbf{B} \leftarrow$ Frodo.Unpack$(\mathbf{b}, n, \overline{n})$
13: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
14: Compute $\mathbf{C}' \leftarrow \mathbf{V} +$ Frodo.Encode(μ')
15: **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ and $\mathbf{d} = \mathbf{d}'$ **then**
16: **return** shared secret $\mathbf{ss} \leftarrow F(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{k}' \| \mathbf{d})$
17: **else**
18: **return** shared secret $\mathbf{ss} \leftarrow F(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{s} \| \mathbf{d})$

2.2.10 Correctness of IND-CCA KEM

The failure probability δ of FrodoKEM is the same as the failure probability of the underlying FrodoPKE as computed in Section 2.2.7.

2.2.11 Interconversion to IND-CCA PKE

FrodoKEM can be converted to an IND-CCA-secure public key encryption scheme using standard conversion techniques as specified by NIST. In particular, shared secret **ss** can be used as the encryption key in

an appropriate data encapsulation mechanism in the KEM/DEM (key encapsulation mechanism / data encapsulation mechanism) framework [44].

2.3 Cryptographic primitives

In FrodoKEM we use the following generic cryptographic primitives. We describe their security requirements and instantiations with NIST-approved cryptographic primitives. In what follows, we use cSHAKE128/256 to denote the use of either cSHAKE128 or cSHAKE256; which one is used with which parameter set for FrodoKEM is indicated in Table 3. Every use of cSHAKE in different contexts listed below is performed with a different domain separator.

- Gen: The security requirement on Gen is that it is a public random function that generates pseudorandom matrices $\mathbf{A}$. Gen is instantiated using either AES128 (as in Algorithm 7) or cSHAKE128 (as in Algorithm 8).
- H: The security requirement on H is that it is a pseudorandom generator. H is instantiated using cSHAKE128/256 with the input being a uniformly random seed $\mathbf{z}$.
- G: The security requirement on G is that it is a pseudorandom function. G is instantiated using cSHAKE128/256 with the input being the public key pk concatenated with a random key μ.
- F: The security requirement on F is that it is a pseudorandom function. F is instantiated using cSHAKE128/256 with the input being the ciphertext components c_1 and c_2, concatenated (during encapsulation and successful decapsulation) with the intermediate shared secret $\mathbf{k}$ or (during failed decapsulation) the seed $\mathbf{s}$.
- cSHAKE: We also use cSHAKE directly in the SampleMatrix algorithm as a pseudorandom function. cSHAKE128/256 is used with the input being a seed $\mathsf{seed}_{\mathbf{E}}$ and distinct context strings for the different matrices generated from the same seed.

2.4 Parameters

This section outlines our methodology for choosing tunable parameters of the proposed algorithms.

2.4.1 High-level overview

Recall the main FrodoPKE parameters defined in Section 2.2:

- χ, a probability distribution on $\mathbb{Z}$;
- $q = 2^D$, a power-of-two integer modulus with exponent $D \leq 16$;
- $n, \overline{m}, \overline{n}$, integer matrix dimensions with $n \equiv 0 \bmod 8$;
- $B \leq D$, the number of bits encoded in each matrix entry;
- $\ell = B \cdot \overline{m} \cdot \overline{n}$ the length of bit strings to be encoded in an $\overline{m}$-by-$\overline{n}$ matrix.

The task of parameter selection is framed as a combinatorial optimization problem, where the objective function is the ciphertext's size, and the constraints are dictated by the target security level, probability of decryption failure, and computational efficiency. The optimization problem is solved by sweeping the parameter space, subject to simple pruning techniques. We perform this sweep of the parameter space using the Python scripts that accompany the submission, in the folder `Additional_Implementations/Parameter_Search_Scripts`.

2.4.2 Parameter constraints

Implementation considerations limit q to be at most 2^{16} and n to be a multiple of 8. Our cost function is the bit length of the FrodoPKE ciphertext, which is $D \cdot \overline{m} \cdot (n + \overline{n})$.

The standard deviation σ of the Gaussian error distribution is bounded from below by the constant 2.12. This bound is chosen following the argument in Section 5.1.5, which demonstrates that $\sigma > 2.12$ conforms to a reduction from the worst-case BDDwDGS problem to the LWE decision problem.

The complexity of the error-sampling algorithm (Section 2.2.4) depends on the support of the distribution and the number of uniformly random bits per sample. We bound the number of bits per sample by 16. Since the distribution is symmetric, the sample's sign ($\mathbf{r}_0$ in Algorithm 5) can be chosen independently from its magnitude e, which leaves 15 bits for sampling from the non-negative part of the support. For each

setting of the variance σ^2 we find a discrete distribution subject to the above constraints that minimizes its Rényi divergence (for several integral orders) from the target "ideal" distribution, which is the rounded Gaussian $\Psi_{\sigma\sqrt{2\pi}}$.

We estimate the concrete security of parameters for our scheme based on cryptanalytic attacks (Section 5.2), accounting for the loss due to substitution of a rounded Gaussian with its discrete approximation (Section 5.1.3). The probability of decryption failure is computed according to the procedure outlined in Section 2.2.6.

In case of ties, i.e., when different parameter sets result in identical ciphertext sizes (i.e., the same q and n), we chose the smaller σ for FrodoKEM-640 (minimizing the probability of decryption failure), and the larger σ for FrodoKEM-976 (prioritizing security).

2.4.3 Selected parameter sets

We present two parameter sets for FrodoKEM. The first, denoted Frodo-640, targets Level 1 in the NIST call for proposals (matching or exceeding the brute-force security of AES-128). The second, denoted Frodo-976, targets Level 3 (matching or exceeding the brute-force security of AES-192). At present, we are not proposing parameters for Level 5, the highest security target (matching or exceeding the brute-force security of AES-256).

In addition to targeting Level 5 and other security objectives, the procedures outlined in this section can be adapted to support alternative cost functions and constraints. For instance, an objective function that takes into account computational costs or penalizes the public key size would lead to a different set of outcomes. For example, constraints can be also chosen to guarantee error-free decryption, or to select parameters that allow for a bounded number of homomorphic operations.

The two parameter sets are given in Table 1. The corresponding error distributions are given in Table 2. Security columns C and Q respectively denote security, in bits, for classical and quantum attacks as estimated by the methodology of Section 5.2.

Table 1: **Parameter sets**

	n	q	σ	**support of χ**	B	$\bar{m} \times \bar{n}$	**failure prob.**	c **size (bytes)**	**Security C**	**Q**
Frodo-640	640	2^{15}	2.75	$[-11 \dots 11]$	2	8×8	$2^{-148.8}$	9,736	143	103
Frodo-976	976	2^{16}	2.3	$[-10 \dots 10]$	3	8×8	$2^{-199.6}$	15,768	209	150

Table 2: **Error distributions**

	σ	**Probability of (in multiples of 2^{-15})** 0	±1	±2	±3	±4	±5	±6	±7	±8	±9	±10	±11	**Rényi order**	**Rényi divergence**
$\chi_{\text{Frodo-640}}$	2.75	9456	8857	7280	5249	3321	1844	898	384	144	47	13	3	500.0	0.72×10^{-4}
$\chi_{\text{Frodo-976}}$	2.3	11278	10277	7774	4882	2545	1101	396	118	29	6	1		500.0	0.14×10^{-4}

2.5 Summary of parameters

Table 3 summarizes all cryptographic parameters for Frodo-640 and Frodo-976. FrodoKEM-640-AES and FrodoKEM-976-AES use AES128 as Gen for generation of $\mathbf{A}$; FrodoKEM-640-cSHAKE and FrodoKEM-976-cSHAKE use cSHAKE as Gen for generation of $\mathbf{A}$.

Table 4 summarizes the sizes, in bytes, of the different inputs and outputs required by FrodoKEM. Note that we also include the size of the public key in the secret key sizes, in order to comply with NIST's API guidelines. Specifically, since NIST's decapsulation API does not include an input for the public key, it needs to be included as part of the secret key.

Table 3: **Cryptographic parameters for** Frodo-640 **and** Frodo-976

	Frodo-640	Frodo-976
D	15	16
q	32768	65536
n	640	976
$\overline{m} = \overline{n}$	8	8
B	2	3
$\mathsf{len}_{\mathbf{A}}$	128	128
$\mathsf{len}_{\mu} = \ell$	128	192
$\mathsf{len}_{\mathbf{E}}$	128	192
$\mathsf{len}_{\mathbf{z}}$	128	192
$\mathsf{len}_{\mathbf{s}}$	128	192
$\mathsf{len}_{\mathbf{k}}$	128	192
$\mathsf{len}_{\mathbf{d}}$	128	192
$\mathsf{len}_{\mathbf{ss}}$	128	192
len_{χ}	16	16
χ	$\chi_{\text{Frodo-640}}$	$\chi_{\text{Frodo-976}}$
H	cSHAKE128$(\cdot, 128, 0)$	cSHAKE256$(\cdot, 128, 0)$
G	cSHAKE128$(\cdot, 384, 3)$	cSHAKE256$(\cdot, 576, 3)$
F	cSHAKE128$(\cdot, 128, 7)$	cSHAKE256$(\cdot, 192, 7)$

Table 4: **Size (in bytes) of inputs and outputs of** FrodoKEM**.** Secret key size is the sum of the sizes of the actual secret value and of the public key (the NIST API does not include the public key as explicit input to decapsulation).

Scheme	**secret key** sk	**public key** pk	**ciphertext** c	**shared secret** **ss**
FrodoKEM-640	19,872 (10,256 + 9,616)	9,616	9,736	16
FrodoKEM-976	31,272 (15,640 + 15,632)	15,632	15,768	24

2.6 Provenance of constants and tables

Constants used as domain separators in calls to cSHAKE are integers selected incrementally starting at 0.

The constants in Table 2 and Table 1 were generated by search scripts following the methodology described in Section 2.4.

3 Performance analysis

3.1 Associated implementations

The submission package includes:

- a reference implementation written exclusively in portable C,
- an optimized implementation written exclusively in portable C that includes efficient algorithms to generate the matrix $\mathbf{A}$ and to compute the matrix operations $\mathbf{AS} + \mathbf{E}$ and $\mathbf{S}'\mathbf{A} + \mathbf{E}'$, and
- an additional, optimized implementation for x64 platforms that exploits Advanced Vector Extensions 2 (AVX2) intrinsic instructions.

The implementations in the submission package support all four schemes: FrodoKEM-640-AES, FrodoKEM-640-cSHAKE, FrodoKEM-976-AES, and FrodoKEM-976-cSHAKE. The only difference between the reference and the optimized implementation is that the latter includes two efficient functions to generate the public matrix $\mathbf{A}$ and to compute the matrix operations $\mathbf{AS} + \mathbf{E}$ and $\mathbf{S}'\mathbf{A} + \mathbf{E}'$. Similarly, the only difference between the optimized and the additional implementation is that the latter uses AVX2 intrinsic instructions to speed up the implementation of the aforementioned functions. Hence, the different implementations share most of their codebase: this illustrates the simplicity of software based on FrodoKEM.

All our implementations avoid the use of secret address accesses and secret branches and, hence, are protected against timing and cache attacks.

3.2 Performance analysis on x64 Intel

In this section, we summarize results of our performance evaluation using a machine equipped with a 3.4GHz Intel Core i7-6700 (Skylake) processor and running Ubuntu 16.04.3 LTS. As standard practice, TurboBoost was disabled during the tests. For compilation we used GNU GCC version 7.2.0 with the command `gcc -O3 -march=native`. The generation of the matrix $\mathbf{A}$ is the most expensive part of the computation. As described in Section 2.2.5, we support two ways of generating $\mathbf{A}$: one using AES128 and one using cSHAKE128.

3.2.1 Performance using AES128

Table 5 details the performance of the optimized implementations and the additional x64 implementations when using AES128 for the generation of the matrix $\mathbf{A}$. The top two sets of results correspond to performance when using OpenSSL's AES implementation[4] and the bottom set presents the results when using a standalone AES implementation using Intel's Advanced Encryption Standard New Instructions (AES-NI).

As can be observed, the different implementation variants have similar performance, even when using hand-optimized AVX2 intrinsic instructions. This illustrates that FrodoKEM's algorithms, which are mainly based on matrix operations, facilitate automatic parallelization using vector instructions. Hence, the compiler is able to achieve close to "optimal" performance with little intervention from the programmer. The best results for FrodoKEM-640-AES and FrodoKEM-976-AES (i.e., 1.1 ms and 2.1 ms, respectively, obtained by adding the times for encapsulation and decapsulation) are achieved by the optimized implementation using C only and by the additional implementation using AVX2 intrinsic instructions, respectively. However, the difference in performance between the different implementations reported in Table 5 is, in all the cases, less than 1%.

We note that the performance of FrodoKEM using AES on Intel platforms greatly depends on AES-NI instructions. For example, when turning off the use of these instructions the computing cost of the optimized implementation of FrodoKEM-640-AES (resp. FrodoKEM-976-AES) is 26.4 ms (resp. 61.2 ms), which is roughly a 24-fold (resp. 29-fold) degradation in performance.

3.2.2 Performance using cSHAKE128

Table 6 outlines the performance figures of the optimized implementations and the additional x64 implementations when using cSHAKE128 for the generation of the matrix $\mathbf{A}$. The top set of results shows the performance of the optimized implementation written in C only, while the bottom set presents the results

[4] Note that in order to enable AES-NI instructions in OpenSSL, we use the `EVP_aes_128_ecb` interface in OpenSSL.

Table 5: **Performance (in thousands of cycles) of FrodoKEM on a 3.4GHz Intel Core i7-6700 (Skylake) processor with matrix A generated using** AES128**.** Results are reported using OpenSSL's AES implementation and using a standalone AES implementation, all of which exploit AES-NI instructions. Cycle counts are rounded to the nearest 10^3 cycles.

Scheme	KeyGen	Encaps	Decaps	Total (Encaps + Decaps)
Optimized Implementation (AES from OpenSSL)				
FrodoKEM-640-AES	1,287	1,810	1,811	3,621
FrodoKEM-976-AES	2,715	3,572	3,588	7,160
Additional implementation using AVX2 intrinsic instructions (AES from OpenSSL)				
FrodoKEM-640-AES	1,293	1,828	1,829	3,657
FrodoKEM-976-AES	2,663	3,565	3,580	7,145
Additional implementation using AVX2 intrinsic instructions (standalone AES)				
FrodoKEM-640-AES	1,288	1,834	1,837	3,671
FrodoKEM-976-AES	2,677	3,577	3,580	7,157

Table 6: **Performance (in thousands of cycles) of FrodoKEM on a 3.4GHz Intel Core i7-6700 (Skylake) processor with matrix A generated using** cSHAKE128**.** Results are reported for two test cases: (i) using a cSHAKE implementation written in plain C and, (ii) using a 4-way implementation of cSHAKE using AVX2 instructions. Cycle counts are rounded to the nearest 10^3 cycles.

Scheme	KeyGen	Encaps	Decaps	Total (Encaps + Decaps)
Optimized Implementation (plain C cSHAKE)				
FrodoKEM-640-cSHAKE	8,297	9,082	9,077	18,159
FrodoKEM-976-cSHAKE	17,798	19,285	19,299	38,584
Additional implementation using AVX2 intrinsics (cSHAKE4x using AVX2)				
FrodoKEM-640-cSHAKE	4,212	4,671	4,672	9,343
FrodoKEM-976-cSHAKE	8,822	9,749	9,720	19,469

when using a 4-way implementation of cSHAKE using AVX2 instructions ("cSHAKE4x using AVX2"). Note that the use of such vectorized implementation of cSHAKE is necessary to boost the practical performance. In our use-case, the usage of such a vectorized implementation results in a two-fold speedup when compared to the version using a cSHAKE implementation written in plain C.

Comparing Table 5 and Table 6, FrodoKEM using AES, when implemented with AES-NI instructions, is around 2.6–2.7× faster than the vectorized cSHAKE implementation. Nevertheless, this comparative result could change drastically if hardware-accelerated instructions such as AES-NI are not available on the targeted platform, or if support for hardware-accelerated instructions for SHA-3 is added in the future.

3.2.3 Memory analysis

Table 7 shows the peak usage of stack memory per function. In addition, in the right-most column we show the size of the produced static libraries.

In order to determine the memory usage we ran `valgrind` (`http://valgrind.org/`) to obtain "memory use snapshots" during execution of the test program:

```
$ valgrind --tool=massif --stacks=yes --detailed-freq=1 ./frodo/test_KEM
```

This command produces a file of the form `massif.out.xxxxx`. We then ran `massif-cherrypick` (https://github.com/lnishan/massif-cherrypick), which is an extension that outputs memory usage per function:

```
$ ./massif-cherrypick massif.out.xxxxx kem_function
```

The results are summarized in Table 7. Note that in our implementations the use of cSHAKE for generating **A** reduces peak memory usage in up to 26%. However, the vectorized AVX2 implementation of cSHAKE increases the size of the produced static libraries significantly (implementations based on AES-NI instructions are indeed very compact).

Table 7: **Peak usage of stack memory (in bytes) and static library size (in bytes)** of the optimized and additional implementations of FrodoKEM on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Compilation with GNU GCC version 7.2.0 using flags `-O3 -march=native`. Matrix **A** is generated with either cSHAKE128 or AES128 (using OpenSSL's AES implementation or the standalone AES implementation).

Scheme	Peak stack memory usage			Static library size
	KeyGen	Encaps	Decaps	
Optimized Implementation (AES from OpenSSL)				
FrodoKEM-640-AES	72,192	103,072	123,968	81,836
FrodoKEM-976-AES	111,424	159,136	189,176	79,700
Additional implementation using AVX2 intrinsics (standalone AES)				
FrodoKEM-640-AES	72,192	103,088	124,016	81,184
FrodoKEM-976-AES	111,424	157,832	189,176	79,048
Additional implementation using AVX2 intrinsics (cSHAKE4x using AVX2)				
FrodoKEM-640-cSHAKE	70,184	81,832	101,016	257,654
FrodoKEM-976-cSHAKE	105,560	124,856	156,248	255,398

3.3 Performance analysis on ARM

In this section, we summarize results of our performance evaluation using a device powered by a 1.992GHz 64-bit ARM Cortex-A72 (ARMv8) processor and running Ubuntu 16.04.2 LTS. For compilation we used GNU GCC version 5.4.0 with the command `gcc -O3 -march=native`.

Table 8 details the performance of the optimized implementations when using AES128 and cSHAKE128. Similar to the case of the x64 Intel platform, the overall performance of FrodoKEM is highly dependent on the performance of the primitive that is used for the generation of the matrix **A**. Hence, the best performance in this case is achieved when using OpenSSL's AES implementation, which exploits the efficient NEON engine. On the other hand, cSHAKE performs significantly better when there is no support for specialized instructions in the targeted platform: using a plain C version of cSHAKE is more than 3 times faster than using a plain C version of AES.

Table 8: **Performance (in thousands of cycles) of the optimized implementations of** FrodoKEM **on a 1.992GHz 64-bit ARM Cortex-A72 (ARMv8) processor.** Results are reported for three test cases: (i) using OpenSSL's AES implementation, (ii) using an AES implementation written in plain C, and (iii) using a cSHAKE implementation written in plain C. Results have been scaled to cycles using the nominal processor frequency. Cycle counts are rounded to the nearest 10^3 cycles.

Scheme	**KeyGen**	**Encaps**	**Decaps**	**Total** (Encaps + Decaps)
Optimized Implementation (AES from OpenSSL)				
FrodoKEM-640-AES	3,343	4,082	4,082	8,164
FrodoKEM-976-AES	7,056	8,428	8,382	16,810
Optimized implementation (plain C AES)				
FrodoKEM-640-AES	44,007	45,013	44,967	89,980
FrodoKEM-976-AES	100,996	102,074	102,323	204,397
Optimized implementation (plain C cSHAKE)				
FrodoKEM-640-AES	12,081	13,458	13,446	26,904
FrodoKEM-976-AES	26,125	28,735	28,506	57,241

4 Known Answer Test (KAT) values

The submission includes KAT values with tuples containing secret key (sk), public key (pk), ciphertext (c) and shared secret (**ss**) values for the proposed KEM schemes. The KAT files can be found in the KAT folder of the submission:

Scheme	**KAT file**
FrodoKEM-640-AES	`\KAT\PQCkemKAT_19872.rsp`
FrodoKEM-976-AES	`\KAT\PQCkemKAT_31272.rsp`
FrodoKEM-640-cSHAKE	`\KAT\PQCkemKAT_19872_cshake.rsp`
FrodoKEM-976-cSHAKE	`\KAT\PQCkemKAT_31272_cshake.rsp`

In addition, we provide a test suite that can be used to verify the KAT values against any of the implementations. Instructions to compile and run the KAT test suite can be found in the README file (see Section 2, "Quick Instructions").

5 Justification of security strength

The security of FrodoKEM is supported both by security reductions and by analysis of the best known cryptanalytic attacks.

5.1 Security reductions

A summary of the reductions supporting the security of FrodoKEM is as follows:

1. FrodoKEM is an IND-CCA-secure KEM under the assumption that FrodoPKE is an IND-CPA-secure public-key encryption scheme, and where G and F are modeled as random oracles. Theorem 5.1 gives a tight, classical reduction against classical adversaries in the classical random oracle model. Theorem 5.2 gives a non-tight, classical reduction against quantum adversaries in the quantum random oracle model.
2. FrodoPKE is an IND-CPA secure public key encryption scheme under the assumption that the corresponding normal-form learning with errors decision problem is hard. Theorem 5.3 gives a tight, classical reduction against classical or quantum adversaries in the standard model.
3. Section 5.1.3 provides justification and bounds exact security loss for substituting exact rounded Gaussian distributions with distributions from Table 2.
4. Section 5.1.4 provides an overview of the security reduction when replacing $\mathbf{A}$ sampled from a truly uniform distribution with one generated in a pseudorandom fashion from a seed. The reduction models AES128 as an ideal cipher when considering Frodo.Gen with AES128 (Algorithm 7) and cSHAKE128 as a random oracle when considering Frodo.Gen with cSHAKE128 (Algorithm 8) and preserve the security up to a small multiplicative loss in the number of samples of the underlying LWE problem.
5. The normal-form learning with errors decision problem is hard under the assumption that the uniform-secret learning with errors decision problem is hard for the same parameters, except for a small additive loss in the number of samples. Theorem 5.4 gives a tight, classical reduction against classical or quantum adversaries (in the standard model).
6. The (average-case) uniform-secret learning with errors decision problem, with the particular values of σ from Table 1 and an appropriate bound on the number of samples, is hard under the assumption that the *worst-case* bounded distance decoding problem with discrete Gaussian samples problem (BDDwDGS, Definition 5.7) is hard for related parameters. Theorem 5.8 gives a non-tight classical reduction against classical or quantum adversaries (in the standard model).

5.1.1 IND-CCA Security of KEM

Theorem 5.1 (IND-CPA PKE $\implies$ IND-CCA KEM in classical ROM). *Let* PKE *be a public key encryption scheme with algorithms* $(\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$*, message space* $\mathcal{M}$*, and which is* δ*-correct. Let* G *and* F *be independent random oracles. Let* $\mathsf{QKEM}_m^{\not\perp\prime} = \mathsf{QFO}_m^{\not\perp\prime}[\mathsf{PKE}, G, F]$ *be the KEM obtained by applying the* $\mathsf{QFO}_m^{\not\perp\prime}$ *transform as in Definition 2.19. For any classical algorithm* $\mathcal{A}$ *against the* IND-CCA *security of* $\mathsf{QKEM}_m^{\not\perp\prime}$ *that makes* q_G *and* q_F *queries to its* G *and* F *oracles, there exists a classical algorithm* $\mathcal{B}$ *against the* IND-CPA *security of* PKE *such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{QKEM}_m^{\not\perp\prime}}(\mathcal{A}) \leq \frac{4 \cdot q_{\mathsf{RO}} + 1}{|\mathcal{M}|} + q_{\mathsf{RO}} \cdot \delta + 3 \cdot \mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{B})$$

where $q_{\mathsf{RO}} = q_G + q_F$*. Moreover, the running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$*.*

Theorem 5.1 follows from Theorems 3.2 and 3.4 of Hofheinz, Hövelmanns, and Kiltz (HHK) [61], with the following modifications. In the application of HHK's Theorem 3.2, we take $q_V = 0$. Note that Theorems 3.2 and 3.4 of HHK are about the $\mathsf{FO}^{\not\perp}$ transform, which differs from the $\mathsf{QFO}_m^{\not\perp\prime}$ in the following ways.

1. $\mathsf{QFO}_m^{\not\perp\prime}$ uses a single hash function (with longer output) to compute r and K whereas $\mathsf{FO}^{\not\perp}$ uses two; but this is equivalent in the random oracle model with appropriate output lengths.
2. $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of r and K also takes the public key pk as input whereas $\mathsf{FO}^{\not\perp}$ does not; this does not negatively affect any of the theorems, and has the potential to provide multi-target security.
3. $\mathsf{QFO}_m^{\not\perp\prime}$ includes the d value in the ciphertext, whereas $\mathsf{FO}^{\not\perp}$ does not; since d is computed by applying a random oracle G to the secret $\mu \in \mathcal{M}$, taking advantage of d requires querying G on μ, which occurs with the additional $q_{\mathsf{RO}}/|\mathcal{M}|$ probability term added in the theorem.

Theorem 5.2 (IND-CPA PKE $\implies$ IND-CCA KEM in quantum ROM). *Let* PKE *be a public key encryption scheme with algorithms* (KeyGen, Enc, Dec)*, message space $\mathcal{M}$, and which is δ-correct. Let G and F be independent random oracles. Let* $\mathsf{QKEM}_m^{\not\perp\prime} = \mathsf{QFO}_m^{\not\perp\prime}[\mathsf{PKE}, G, F]$ *be the KEM obtained by applying the* $\mathsf{QFO}_m^{\not\perp\prime}$ *transform as in Definition 2.19. For any quantum algorithm $\mathcal{A}$ against the* IND-CCA *security of* $\mathsf{QKEM}_m^{\not\perp\prime}$ *that makes q_G and q_F queries to its quantum G and F oracles, there exists a quantum algorithm $\mathcal{B}$ against the* IND-CPA *security of* PKE *such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{QKEM}_m^{\not\perp\prime}}(\mathcal{A}) \leq 9 \cdot q_{\mathsf{RO}} \cdot \sqrt{q_{\mathsf{RO}}^2 \cdot \delta + q_{\mathsf{RO}} \cdot \sqrt{\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{B}) + \frac{1}{|\mathcal{M}|}}}$$

where $q_{\mathsf{RO}} = q_G + q_F$. Moreover, the running time of $\mathcal{B}$ is about that of $\mathcal{A}$.

Theorem 5.2 follows from Lemma 2.3 and Theorems 4.4 and 4.6 of Hofheinz, Hövelmanns, and Kiltz [61], with the following modifications. Note that Theorems 4.4 and 4.6 of HHK are about the $\mathsf{QFO}_m^{\not\perp}$ transform, which differs from the $\mathsf{QFO}_m^{\not\perp\prime}$ in the following ways. 1) $\mathsf{QFO}_m^{\not\perp\prime}$ uses a single hash function (with longer output) to compute r, K, and d whereas $\mathsf{FO}^{\not\perp}$ uses two; but this is equivalent in the random oracle model with appropriate output lengths. 2) $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of r, K, and d also takes the public key pk as input whereas $\mathsf{FO}^{\not\perp}$ does not; this does not negatively affect any of the theorems, and has the potential to provide multi-target security. 3) $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of the shared secret k also takes the encapsulation c as input; this does not negatively affect any of the theorems, and provides robustness against ciphertext modification.

Note that Theorem 5.2 is far from being tight due to the fourth-root between the $\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{B})$ term and the $\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{QKEM}_m^{\not\perp\prime}}(\mathcal{A})$ term. As of writing there is no known attack that takes advantage of the tightness gap, and so it seems to be an artifact of the proof technique. As noted in Section 2.2.8, a recent eprint of Jiang et al. [63] presents an alternative transform from IND-CPA PKE to IND-CCA KEM in the quantum random oracle model, and their theorems give only a square-root gap between the advantages; our IND-CPA PKE satisfies the conditions needed for their Theorem 1 to yield IND-CCA security of the resulting KEM. In our parameter selection, we ignore the tightness gap arising from Theorem 5.2.

5.1.2 IND-CPA Security of PKE

Theorem 5.3 (Normal Form DLWE $\implies$ IND-CPA security of FrodoPKE). *Let $n, q, \overline{m}, \overline{n}$ be positive integers, and χ be a probability distribution on $\mathbb{Z}$. For any quantum algorithm $\mathcal{A}$ against the* IND-CPA *security of* FrodoPKE *(with a uniformly random $\mathbf{A}$), there exist quantum algorithms $\mathcal{B}_1$ and $\mathcal{B}_2$ against the normal-form LWE decision problem such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{FrodoKEM}}(\mathcal{A}) \leq \overline{n} \cdot \mathrm{Adv}^{\mathsf{nf\text{-}dlwe}}_{n,n,q,\chi}(\mathcal{B}_1) + \overline{m} \cdot \mathrm{Adv}^{\mathsf{nf\text{-}dlwe}}_{n,n+\overline{n},q,\chi}(\mathcal{B}_2) \ .$$

Moreover, the running times of $\mathcal{B}_1$ and $\mathcal{B}_2$ are approximately that of $\mathcal{A}$.

The proof of Theorem 5.3 is the same as that of [75, Theorem 3.2] or [24, Theorem 5.1].

Theorem 5.4 (uniform-secret DLWE $\implies$ normal-form DLWE). *Let n, q, m be integers, and χ be a probability distribution on $\mathbb{Z}$. For any quantum algorithm $\mathcal{A}$ against the normal-form LWE decision problem, there exists a quantum algorithm $\mathcal{B}$ against the uniform-secret LWE decision problem such that*

$$\mathrm{Adv}^{\mathsf{nf\text{-}dlwe}}_{n,m,q,\chi}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{dlwe}}_{n,m+O(n),q,\chi}(\mathcal{B}) \ .$$

Moreover, the running time of $\mathcal{B}$ is about that of $\mathcal{A}$.

The proof of Theorem 5.4 is the same as that of [13, Lemma 2].

5.1.3 Approximating the error distribution

The discrete Gaussian distribution (Definition 2.14), whose properties are key to the worst-to-average-case reduction, is difficult to sample from on a finite computer (and impossible to do so in constant time). Following Langlois et al. [73], we replace an infinite-precision distribution with its discrete approximation and quantify the loss of security by computing the Rényi divergence between the two distributions.

Definition 5.5 (Rényi divergence). Rényi divergence of order α between two discrete distributions P and Q is defined as

$$D_\alpha(P\|Q) = \frac{1}{\alpha - 1} \ln \sum_{x \in \operatorname{supp} P} P(x) \left(\frac{P(x)}{Q(x)}\right)^{\alpha-1} .$$

(Note that our definition differs from Langlois et al. in that we take the logarithm of the sum.)

The following theorem relates probabilities of observing a certain event under two distributions as a function of their Rényi divergence.

Theorem 5.6 ([73, Lemma 4.1]). *If there is an event S defined in a game G_Q where n samples are drawn from distribution Q, the probability of S in the same game where Q is replaced with P is bounded as follows:*

$$\Pr[G_P(S)] \leq (\Pr[G_Q(S)] \cdot \exp(n \cdot D_\alpha(P\|Q)))^{1-1/\alpha}. \tag{4}$$

Reduction to any *search* problem, such as the ones that appear in the proof of Theorem 5.3 (specifically, winning in the OW-PCVA game as defined in [61]) are preserved subject to the relaxation (4). For each exact security argument, and any concrete choice of the two distributions P and Q, the bound can be minimized by choosing an optimal value of the Rényi order α.

For a concrete example of application of Theorem 5.6 consider the distribution $\chi_{\mathsf{Frodo\text{-}640}}$ specified according to Table 2. The distribution approximates the rounded Gaussian $\Psi_{2.75/\sqrt{2\pi}}$ as defined in Section 2.1.4. During a single run of FrodoKEM the parties sample from the distribution $(8+8)\times640+64 = 10{,}304$ times. Assume that the adversary attacking the underlying search problem (of recovering the shared secret key *before* applying the random oracle) has probability of winning 2^{-145} when the parties sample from $\Psi_{2.75/\sqrt{2\pi}}$. According to Table 2 the Rényi divergence between the two distributions is $D_{500}(\chi_{\mathsf{Frodo\text{-}640}}\|\Psi_{2.75/\sqrt{2\pi}}) = .000074$. Substituting the rounded Gaussian distribution with $\chi_{\mathsf{Frodo\text{-}640}}$ and applying Theorem 5.6 will lead to the following bound on classical security of the resulting protocol (cf. Table 1): $(2^{-145} \cdot \exp(10304 \cdot .000074))^{.998} \approx 2^{-143.6}$.

5.1.4 Deterministic generation of A

The matrix $\mathbf{A}$ in FrodoKEM is deterministically expanded from a short random seed in the function Frodo.Gen either using AES128 or cSHAKE128. In order to relate FrodoKEM's security to the hardness of the learning with errors problem, we argue that we can replace a uniformly sampled $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ with matrices sampled according to Frodo.Gen. Although the matrix appears pseudorandom under standard security assumptions to an adversary without access to the seed, we argue security of this step against a stronger (and more realistic) adversary via the indifferentiability framework [80, 41].

Informally, a construction $\mathcal{C}$ with access to an ideal primitive $\mathcal{G}$ is said to be ε-indifferentiable from an ideal primitive $\mathcal{F}$ if there exists a simulator $\mathcal{S}$ such that for any polynomial time distinguisher $\mathcal{D}$ it holds that $\left|\Pr\left[\mathcal{D}^{\mathcal{C},\mathcal{G}} = 1\right] - \Pr\left[\mathcal{D}^{\mathcal{F},\mathcal{S}} = 1\right]\right| < \varepsilon$. An indifferentiability argument implies that any cryptosystem secure in the $\mathcal{F}$-model remains secure (in a tight sense) in the $\mathcal{G}$-model with $\mathcal{F}$ instantiated as $\mathcal{C}^{\mathcal{G}}$ [80]. In what follows, we consider the ideal primitive $\mathcal{F}$ to be an ideal "domain expansion" function expanding a small seed to a matrix $\mathbf{A}$. Criticially, the security of the step depends on the properties of $\mathcal{G}$ rather than randomness of the seed. The construction $\mathcal{C}$ and primitive $\mathcal{G}$ depend on whether we use AES128 or cSHAKE128, modeled below as an ideal cipher and an ideal extendable-output function (XOF) respectively.

Using AES128 to generate A. Algorithm 7 generates the entries of $\mathbf{A}$ as 16-bit values and then reduces each one modulo q. For simplicity, we assume that $\mathbf{A}$ consists of $N = 16n^2$ bits and we set $M = N/128$. This means that $\mathbf{A}$ consists of M 128-bit AES128 blocks. The pseudorandom bits in the ith block are generated by encrypting a fixed index idx_i with a uniformly random $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{128}$ as the key. Throughout, we refer to the set $\mathrm{Idx} := \{\mathsf{idx}_1, \ldots, \mathsf{idx}_M\}$ as the set of indices used in the pseudorandom generation of $\mathbf{A}$.

The ideal domain expansion primitive $\mathcal{F}$ expands a uniformly random seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{128}$ to a larger bit string $s_1\|s_2\|\cdots\|s_M \in \{0,1\}^{128M}$ subject to the condition that $s_i \neq s_j$ for any distinct pair of i, j. Observe that a uniformly sampled $\mathbf{A}$ satisfies this condition with probability at least $1 - M^2/2^{128}$. In our security reductions, the matrix $\mathbf{A}$ is constructed through $m = n$ calls to the LWE oracle (Definition 2.9). By increasing the number of calls to this oracle marginally, by setting $m = 1.01n > n \cdot (1 - M^2/2^{128})^{-1}$, we

can construct an LWE matrix $\mathbf{A}$ sampled from the same distribution as the output of $\mathcal{F}$ with overwhelming probability without affecting its underlying security.

When Frodo.Gen uses AES128, we consider a construction $\mathcal{C}^{\mathcal{G}}$ in the Ideal Cipher model implementing $\mathcal{F}$ as $\text{AES128}_{\mathsf{seed}_\mathbf{A}}(\mathsf{idx}_1)\| \cdots \|\text{AES128}_{\mathsf{seed}_\mathbf{A}}(\mathsf{idx}_M)$. We show that $\mathcal{C}^{\mathcal{G}}$ is indifferentiable from $\mathcal{F}$ as follows. Consider the two worlds with which $\mathcal{D}$ interacts to make queries on the construction $\mathcal{C}$ and $\mathcal{G}$:

- **REAL.** In the real world, upon query $\mathcal{C}(k)$, $\mathcal{D}$ receives $\text{AES128}_k(\mathsf{idx}_1)\| \cdots \|\text{AES128}_k(\mathsf{idx}_M)$. Queries to $\mathcal{G}$ are answered naturally with $\text{AES128}_{(\cdot)}(\cdot)$ or $\text{AES128}^{-1}_{(\cdot)}(\cdot)$ as required.
- **IDEAL.** In the ideal world, upon query $\mathcal{C}(k)$, the simulator $\mathcal{S}$ simulates $\mathcal{F}$ as follows. $\mathcal{S}$ samples M uniformly random strings $s_1, \ldots, s_M$ subject to no collisions and outputs $\mathcal{F}(k) = s_1\| \cdots \|s_M$. It additionally stores a mapping M_k from $\{\mathsf{idx}_1, \ldots, \mathsf{idx}_M\}$ to $S = \{s_1, \ldots, s_M\}$. These will be used to answer $\mathcal{G}$ queries. Without loss of generality, we assume that whenever $\mathcal{G}$ is queried on a key k, $\mathcal{S}$ pretends that $\mathcal{C}(k)$ has been queried and sets up M_k.
  $\mathcal{D}$ can now effectively simulate an ideal cipher $\mathcal{G}$ as follows. For forward queries with an input in Idx or backward queries with an input in S_k, $\mathcal{S}$ uses the mapping M_k to answer the query in a manner consistent with $\mathcal{C}(\cdot)$ simulation. For all other queries, the simulator maintains an on-the-fly table to simulate an ideal cipher. It samples independent uniformly random responses for each input query (forward or backward) subject to the fact that the resulting table of input/output pairs (x, y) combined with (idx_i, s_i) pairs remains a permutation over $\{0,1\}^{128}$ for every key k.

It is easy to see that the simulator is efficient. Indifferentiability of the two worlds follows by construction as $\text{AES128}(\cdot,\cdot)$ is modeled as an ideal cipher. Thus, in generating $\mathbf{A}$ starting with a seed $\mathsf{seed}_\mathbf{A}$ using AES128, we can effectively replace the ideal domain extension primitive $\mathcal{F}$ with our construction in the ideal cipher model.

Using cSHAKE128 to generate A. An argument in using cSHAKE128 to expand $\mathsf{seed}_\mathbf{A}$ to the matrix $\mathbf{A}$ is significantly simpler. In the random oracle model, cSHAKE128 is an ideal XOF [49]. In fact, for every customization string *str*, we can model $\text{cSHAKE128}(\cdot, \ell, str)$ as an independent hash function mapping $\{0,1\}^{128}$ to $\{0,1\}^{\ell}$. The domain expansion step is constructed by computing $\text{cSHAKE128}(\mathsf{seed}_\mathbf{A}, 16n, 2^8 + i)$ for $1 \le i \le n$ where each step fills up the ith row of the matrix $\mathbf{A}$. As each row is independently constructed via an ideal hash function, this construction maps a uniformly random seed $\mathsf{seed}_\mathbf{A}$ to a much larger uniformly random matrix $\mathbf{A}$ thereby implementing the ideal functionality $\mathcal{F}$ perfectly.

Reusing A. Finally, we point out that generating $\mathbf{A}$ from $\mathsf{seed}_\mathbf{A}$ can be a significant computational burden, but this cost can be amortized by relaxing the requirement that a fresh $\mathsf{seed}_\mathbf{A}$ be used for every instance of key encapsulation, e.g., by caching and reusing $\mathbf{A}$ for a small period of time. In this case, we observe that the cost of generating $\mathbf{A}$ represents roughly 40% of the cost of encapsulation and decapsulation on the targeted x64 Intel machine used in Section 3. A straightforward argument shows that the amortization above is compatible with all the security reductions in this section. But importantly, it now allows for an all-for-the-price-of-one attack against those key encapsulations that share the same $\mathbf{A}$. This can be mitigated by making sure that we cache and reuse $\mathbf{A}$ only for a small number of uses, but we need to do this in a very careful manner.

Generating A from joint randomness. It is also possible to generate $\mathbf{A}$ from joint randomness or using protocol random nonces. For example, when integrating FrodoKEM into the TLS protocol, $\mathbf{A}$ could be generated from a seed consisting of the random nonces `client_random` and `server_random` sent by the client and server in their `ClientHello` and `ServerHello` messages in the TLS handshake protocol. This functionality does not match the standard description of a KEM and the API provided by NIST, but is possible in general. A design with both parties contributing entropy to the seed might better protect against all-for-the-price-of-one attacks by being more robust to faulty random number generation at one of the parties.

5.1.5 Reductions from worst-case lattice problems

When choosing parameters for LWE, one needs to choose an error distribution, and in particular its "width." Certain choices (e.g., sufficiently wide Gaussians) are supported by *reductions* from worst-case lattice problems

to LWE; see, e.g., [102, 88, 28, 94]. At a high level, such a reduction transforms any algorithm that solves LWE *on the average*—i.e., for random instances sampled according to the prescribed distribution—into an algorithm of related efficiency that solves *any instance* of certain lattice problems (not just random instances).

The original work of [102] and a follow-up work [94] gave quantum polynomial-time reductions, from the worst-case GapSVP_γ (Definition 2.15), SIVP_γ (Definition 2.16), and DGS_φ (Definition 2.17) problems on n-dimensional lattices, to n-dimensional LWE (for an unbounded polynomial $m = \mathrm{poly}(n)$ number of samples) with Gaussian error of standard deviation $\sigma \geq c\sqrt{n}$. The constant factor c was originally stated as $c = \sqrt{2/\pi}$, but can easily be improved to any $c > 1/(2\pi)$ via a tighter analysis of essentially the same proof.[5] However, for efficiency reasons our choices of σ (see Table 2) are somewhat smaller than required by these reductions.

Instead, following [102, Section 1.1], below we obtain an alternative *classical* (i.e., non-quantum) reduction from a variant of the worst-case bounded-distance decoding (BDD) problem to our LWE parameterizations. In contrast to the quantum reductions described above, which requires Gaussian error of standard deviation $\sigma = \Theta(\sqrt{n})$, the alternative reduction supports a smaller error width—as small as the "smoothing parameter" [84] of the lattice of integers $\mathbb{Z}$. For the BDD variant we consider, which we call "BDD with Discrete Gaussian Samples" ($\mathsf{BDDwDGS}$), the input additionally includes discrete Gaussian samples over the dual lattice, but having a larger width than known algorithms are able to exploit [77, 45]. Details follow.

Bounded-distance decoding with discrete Gaussian samples. We first define a variant of the bounded-distance decoding problem, which is implicit in prior works that consider "BDD with preprocessing," [2, 77, 45] and recall the relevant aspects of known algorithms for the problem.

Definition 5.7 (Bounded-distance decoding with discrete Gaussian samples). For a lattice $\mathcal{L} \subset \mathbb{R}^n$ and positive reals $d < \lambda_1(\mathcal{L})/2$ and $r > 0$, an instance of the *bounded-distance decoding with discrete Gaussian samples* problem $\mathsf{BDDwDGS}_{\mathcal{L},d,r}$ is a point $\mathbf{t} \in \mathbb{R}^n$ such that $\mathrm{dist}(\mathbf{t}, \mathcal{L}) \leq d$, and access to an oracle that samples from $D_{\mathcal{L}^*,s}$ for any (adaptively) queried $s \geq r$. The goal is to output the (unique) lattice point $\mathbf{v} \in \mathcal{L}$ closest to $\mathbf{t}$.

Remark. For a given distance bound d, known $\mathsf{BDDwDGS}$ algorithms use discrete Gaussian samples that all have the same width parameter s. However, the reduction to LWE will use the ability to vary s. Alternatively, we mention that when $r \geq \eta_\varepsilon(\mathcal{L}^*)$ for some very small $\varepsilon > 0$ (which will always be the case in our setting), we can replace the variable-width DGS oracle from Definition 5.7 with a fixed-width one that samples from $D_{\mathbf{w}+\mathcal{L}^*,r}$ for any queried coset $\mathbf{w} + \mathcal{L}^*$, always for the same width r. This is because we can use the latter oracle to implement the former one (up to statistical distance 8ε), by sampling $\mathbf{e}$ from the continuous Gaussian of parameter $\sqrt{s^2 - r^2}$ and then adding a sample from $D_{\mathcal{L}^*-\mathbf{e},r}$. See [90, Theorem 3.1] for further details.

The state-of-the-art algorithms for solving $\mathsf{BDDwDGS}$ [2, 77, 45] deal with a certain $\mathcal{L}$-periodic function $f_{\mathcal{L},1/r} \colon \mathbb{R}^n \to [0, 1]$, defined as

$$f_{\mathcal{L},1/r}(\mathbf{x}) := \frac{\rho_{1/r}(\mathbf{x} + \mathcal{L})}{\rho_{1/r}(\mathcal{L})} = \mathop{\mathbb{E}}_{\mathbf{w} \sim D_{\mathcal{L}^*,r}} [\cos(2\pi \langle \mathbf{w}, \mathbf{x} \rangle)] \ , \tag{5}$$

where the equality on the right follows from the Fourier series of $f_{\mathcal{L},1/r}$ (see [2]). To solve $\mathsf{BDDwDGS}$ for a target point $\mathbf{t}$, the algorithms use several discrete Gaussian samples $\mathbf{w}_i \sim D_{\mathcal{L}^*,r}$ to estimate the value of $f_{\mathcal{L},1/r}$ at $\mathbf{t}$ and nearby points via Equation (5), to "hill climb" from $\mathbf{t}$ to the nearest lattice point. For the relevant points $\mathbf{t}$ we have the (very sharp) approximation

$$f_{\mathcal{L},1/r}(\mathbf{t}) \approx \exp(-\pi r^2 \cdot \mathrm{dist}(\mathbf{t}, \mathcal{L})^2) \ ,$$

so by the Chernoff-Hoeffding bound, approximating $f_{\mathcal{L},1/r}(\mathbf{t})$ to within (say) a factor of two uses at least

$$\frac{1}{f_{\mathcal{L},1/r}(\mathbf{t})^2} \approx \exp(2\pi r^2 \cdot \mathrm{dist}(\mathbf{t}, \mathcal{L})^2)$$

[5] The approximation factor γ for GapSVP and SIVP is $\tilde{O}(qn/\sigma) = (qn/\sigma) \log^{O(1)} n$, and the parameter φ for DGS is $\Theta(q\sqrt{n}/\sigma)$ times the "smoothing parameter" of the lattice.

samples.[6] Note that without enough samples, the "signal" of $f_{\mathcal{L},1/r}(\mathbf{t})$ is overwhelmed by measurement "noise," which prevents the hill-climbing from making progress toward the answer.

In summary, when limited to N discrete Gaussian samples, the known approaches to solving BDDwDGS are limited to distance

$$\mathrm{dist}(\mathbf{t}, \mathcal{L}) \leq r^{-1}\sqrt{\ln(N)/(2\pi)} \ , \tag{6}$$

and having such samples does not seem to provide any speedup in decoding at somewhat distances that are larger by some constant factor greater than one. In particular, if $d \cdot r \geq \omega(\sqrt{\log n})$ (which is the smoothing parameter of the integer lattice $\mathbb{Z}$ for negligible error ε), then having $N = \mathrm{poly}(n)$ samples does not seem to provide any help in solving $\mathsf{BDDwDGS}_{\mathcal{L},d,r}$ (versus having no samples at all).

Reduction from BDDwDGS to LWE. We now recall the following result from [94], which generalizes a key theorem from [102] to give a reduction from BDDwDGS to the LWE decision problem.

Theorem 5.8 (BDDwDGS hard $\implies$ decision-LWE hard [94, Lemma 5.4]). *Let $\varepsilon = \varepsilon(n)$ be a negligible function and let $m = poly(n)$ and $C = C(n) > 1$ be arbitrary. There is a probabilistic polynomial-time (classical) algorithm that, given access to an oracle that solves $\mathsf{DLWE}_{n,m,q,\alpha}$ with non-negligible advantage and input a number $\alpha \in (0,1)$, an integer $q \geq 2$, a lattice $\mathcal{L} \subset \mathbb{R}^n$, and a parameter $r \geq Cq \cdot \eta_\varepsilon(\mathcal{L}^*)$, solves $\mathsf{BDDwDGS}_{\mathcal{L},d,r}$ using $N = m \cdot poly(n)$ samples, where $d = \sqrt{1 - 1/C^2} \cdot \alpha q/r$.*

Remark. The above statement generalizes the fixed choice of $C = \sqrt{2}$ in the original statement (inherited from [102, Section 3.2.1]), using [102, Corollary 3.10]. In particular, for any constant $\delta > 0$ there is a constant $C > 1$ such that $d = (1 - \delta) \cdot \alpha q/r$.

In particular, by Equation (6), if the Gaussian parameter αq of the LWE error sufficiently exceeds $\sqrt{\ln(N)/(2\pi)}$ (e.g., by a constant factor greater than one), then the $\mathsf{BDDwDGS}_{\mathcal{L},d,r}$ problem is plausibly hard in the worst case, hence so is the corresponding LWE problem from Theorem 5.8.

Concretely, if we use an extremely large bound $N \leq 2^{256}$ on the number of discrete Gaussian samples, then the threshold for Gaussian parameters αq that conform to Theorem 5.8 is $\sqrt{\ln(N)/(2\pi)} \approx 5.314$, which corresponds to a standard deviation threshold of $\sqrt{\ln(N)}/(2\pi) \approx 2.120$. Our FrodoPKE parameters, which use standard deviation $\sigma \geq 2.3$ (see Table 2), are above this threshold by a comfortable margin. (Note that a standard deviation threshold of 2.3 corresponds to $N \approx 2^{300}$.)

5.2 Cryptanalytic attacks

In this section, we explain our methodology to estimate the security level of our proposed parameters. The methodology is similar to the one proposed in [11], with slight modifications taking into account the fact that some quasi-linear accelerations [107, 26] over sieving algorithms [16, 69] are not available without the ring structure.

We also remark that this methodology is significantly more conservative than what is usually used in the literature [10], at least since recently. Indeed, we must acknowledge that lattice cryptanalysis is far less mature than that for factoring and computing discrete logarithms, for which the best-known attacks can more safely be considered best-possible attacks.

5.2.1 Methodology: the core-SVP hardness

In this section, let m_{samp} denote the number of LWE samples available to the attacker. Due to the small number of samples (i.e., $m_{\mathsf{samp}} \approx n$ in our schemes) we are not concerned with either BKW types of attacks [65] or linearization attacks [14]. This essentially leaves us with two BKZ [40] attacks, usually referred to as primal and dual attacks that we will briefly recall below.

Formally, BKZ with block-size b requires up to polynomially many calls to an SVP oracle in dimension b, but some heuristics allow to decrease the number of calls to be essentially linear [39]. To account for further improvement, we shall count only the cost of one such call to the SVP oracle: the core-SVP hardness. Such

[6] In fact, the algorithms need approximation factors much better than two, so the required number of samples is even larger by a sizable constant factor. However, the above crude bound will be sufficient for our purposes.

precaution is motivated by the fact that there are ways to amortize the cost of SVP calls inside BKZ, especially when sieving is to be used as the SVP oracle. Such a strategy was suggested in a talk, but has so far not been experimentally tested, as more implementation effort is required to integrate sieving within BKZ.

Even evaluating the concrete cost of one SVP oracle call in dimension b is difficult, because the numerically optimized pruned enumeration strategy does not yield a closed formula [53, 40]. Yet, asymptotically, enumeration is super-exponential (even with pruning), while sieving algorithms are exponential $2^{cb+o(b)}$ with a well understood constant c in the exponent. A sound and simple strategy is therefore to give a lower bound for the cost of an attack by 2^{cb} vector operations (i.e. about $b2^{cb}$ CPU cycles[7]), and to make sure that the block-size b is in a range where enumeration costs more than 2^{cb}. From the estimates of [40], it is argued in [11] that this is the case both classically and quantumly whenever $b \geq 200$.

The best known constant in the exponent is for classical algorithms is $c_{\mathrm{C}} = \log_2 \sqrt{3/2} \approx 0.292$, as provided by the sieve algorithm of [16]. For quantum algorithms it isq $c_{\mathrm{Q}} = \log_2 \sqrt{13/9} \approx 0.265$ [69, Sec. 14.2.10]. Because all variants of the sieve algorithm require building a list of $\sqrt{4/3}^b$ many vectors, the constant $c_{\mathrm{P}} = \log_2 \sqrt{4/3} \approx .2075$ can plausibly serve as a "worst-possible" lower bound for sieving algorithm.

Conservatism: lower bounds vs. experiments. These estimates are very conservative compared to the state of the art implementation of [79], which has practical complexity of about $2^{0.405b+11}$ cycles in the range $b = 60 \ldots 80$. The classical lower bound of $2^{0.292b}$ corresponds to a margin factor of 2^{20} at blocksize $b = 80$, and this margin should continue increasing with the blocksize (abusing the linear fit suggests a margin of 2^{45} at blocksize $b = 300$).

Conservatism: future improvements. Of course, one could assume further improvements on known techniques. At least asymptotically, it may be reasonable to assume that $2^{0.292b+o(b)}$ is optimal for SVP considering that the underlying technique of [16] has been shown to reach lower bounds for the generic nearest-neighbor search problem [12]. As for concrete improvements, we note that this algorithm has already been subject to some fine-tuning in [79], so we may conclude that there is not much more to be gained without introducing new ideas. We therefore consider our margin sufficient to absorb such future improvements.

Conservatism: cost models. The NIST call for proposals suggested a particular cost model, inspired by the estimates of a Grover search attack on AES, essentially accounting for the quantum gate count. In comparison, the literature on sieving algorithms mostly focuses on analysis in the RAM model and quantumly accessible RAM models, and considers the amount of memory they use. Their cost in the area-time model should be higher by polynomial, if not exponential, factors.

Firstly, our model accounts for arithmetic operations rather than gates (used to compute inner products and evaluate norms of vectors). The conversion to gate count may not be trivial as it is unclear how many bits of precision are required.

Secondly, even in the classical setting, the cost of sieving in large dimensions may not be accurately captured by the count of elementary operations in the RAM model, as those algorithms use an exponential amount of memory. Admittedly, the most basic sieve algorithm (with theoretical complexity $2^{0.415b+o(b)}$) has sequential memory access, and can therefore be efficiently implemented by a large circuit without memory access delays. But more advanced ones [16] have much less predictable memory access patterns, and memory complexities as large as time complexities ($2^{0.292b+o(b)}$). It is unclear if they can be adapted to reach a complexity $2^{0.292b+o(b)}$ in the area-time model; one might expect extra polynomial factors to appear. (Following an idea of [17], Becker et al. [16] also claims a version that only requires $2^{0.2015b+o(b)}$ memory, but we suspect this would come at some hidden cost on the running time.)

Moreover, the quantum versions of all sieving algorithms work in the quantumly accessible RAM model [71]. Again, the conversion to an efficient quantum circuit will induce extra costs—at least polynomial ones.

[7] Because of the additional ring-structure, [11] chooses to ignore this factor b to the advantage of the adversary, assuming the techniques of [107, 26] can be adapted to more advanced sieve algorithms [11]. But for plain LWE, we can safely include this factor.

5.2.2 Primal attack

The primal attack consists of constructing a unique-SVP instance from the LWE problem and solving it using BKZ. We examine how large the block dimension b is required to be for BKZ to find the unique solution. Given the matrix LWE instance $(\mathbf{A}, \mathbf{b} = \mathbf{As}+\mathbf{e})$ one builds the lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^{m+n+1} : (\mathbf{A}|-\mathbf{I}_m|-\mathbf{b})\mathbf{x} = 0 \bmod q\}$ of dimension $d = m + n + 1$, volume q^m, and with a unique-SVP solution $\mathbf{v} = (\mathbf{s}, \mathbf{e}, 1)$ of norm $\lambda \approx \sigma\sqrt{n+m}$. The number of used samples m may be chosen between 0 and m_{samp} in our case and we numerically optimize this choice.

Using the typical models of BKZ (geometric series assumption, Gaussian heuristic [39, 10]) one concludes that the primal attack is successful if and only if

$$\sigma\sqrt{b} \leq \delta^{2b-d-1} \cdot q^{m/d} \quad \text{where } \delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)} \ . \tag{7}$$

We note that this condition, introduced in [11], is substantially different from the one suggested in [52] and is used in many previous security analyses, such as [10]. The recent study [9] showed that this new condition predicts significantly smaller security levels than the older, and is corroborated by extensive experiments.

5.2.3 Dual attack

The dual attack searches for a short vector in the dual lattice $\mathbf{w} \in \hat{\Lambda} = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \mathbf{A}^t\mathbf{x} = \mathbf{y} \bmod q\}$ and intends to use it as a distinguisher for LWE. The algorithm BKZ with block size b will output such a vector of length $\ell = \delta^{d-1}q^{n/d}$.

Having found $(\mathbf{x}, \mathbf{y}) \in \hat{\Lambda}$ of length ℓ, an attacker computes $z = \mathbf{v}^t \cdot \mathbf{b} = \mathbf{v}^t\mathbf{As} + \mathbf{v}^t\mathbf{e} = \mathbf{w}^t\mathbf{s} + \mathbf{v}^t\mathbf{e} \bmod q$ which is distributed as a Gaussian of standard deviation $\ell\sigma$ if $(\mathbf{A}, \mathbf{b})$ is indeed an LWE sample (otherwise it is uniform mod q). Those two distributions have maximal variation distance bounded by $\varepsilon = \exp(-\pi\tau^2)$ where $\tau = \ell\sigma/q$: given such a vector of length ℓ the attacker may distinguish LWE samples from random with advantage at most ε.

Note that small advantages ε are not meaningful to attack a key exchange protocol: as the agreed key is to be used as a symmetric cipher key, any advantage below $1/2$ does not significantly decrease the search space to bruteforce the symmetric cipher.[8]

We therefore require the attacker to amplify his success probability by building about $1/\varepsilon^2$ many such short vectors. Because the sieve algorithms provide $2^{.2075b}$ vectors, the attack must be repeated at least R times where

$$R = \max(1, 1/(2^{.2075b}\varepsilon^2)).$$

This is again quite pessimistic, as the other vectors outputted by the sieving algorithm are a bit larger than the shortest one.

[8] To make this more formal, one may simply hash the agreed key with a random oracle before using it for any other purposes.

6 Advantages and limitations

6.1 Ease of implementation

One of the features of FrodoKEM is that it is easy to implement and naturally facilitates writing implementations that are compact and run in constant-time. This latter feature aids to avoid common cryptographic implementation mistakes which can lead to key-extraction based on, for instance, timing differences when executing the code. For example, the additional x64 implementation of the full KEM scheme accompanying this submission consists of slightly more than 250 lines of plain C code.[9] This same code is used for the two security levels FrodoKEM-640 and FrodoKEM-976, with parameters changed by a small number of macros at compile-time.

Computing on matrices —the basic operation in FrodoKEM— allows for easy scaling to different dimensions n. In addition, FrodoKEM uses a modulus q that is always equal or less than 2^{16}. These two combined aspects allow for the full reuse of the matrix functions for the different security levels by instantiating them with the right parameters at build time. Since the modulus q used is always a power of two, implementing arithmetic modulo q is simple, efficient and ease to do in constant-time in modern computer architectures: for instance, computing modulo 2^{16} comes for free when using 16-bit data-types. Moreover, the dimension values were chosen to be divisible by 16 in order to facilitate vectorization optimizations and to simplify the use of AES128 for the generation of the matrix $\mathbf{A}$.

Also the error sampling is designed to be simple and facilitates code reuse: for any security level, FrodoKEM requires 16 bits per sample, and the tables T_χ corresponding to the discrete cumulative density functions always consist of values that are less than 2^{15}. Hence, a simple function applying inversion sampling (see Algorithm 5) can be instantiated using different precomputed tables T_χ. Moreover, due to the small sizes of these pre-computed tables constant-time table lookups, needed to protect against attacks based on timing differences, can be implemented almost for free in terms of effort and performance impact.

6.2 Compatibility with existing deployments and hybrid schemes

FrodoKEM does have larger public key / encapsulation sizes than traditional RSA and elliptic curve cryptosystems, and some other post-quantum candidates such as ring-LWE-based schemes. Nonetheless, FrodoKEM's communication size is sufficiently small that it is still compatible with many existing deployments. In our original research paper on FrodoCCS [24], we integrated FrodoCCS as well as several other key encapsulation mechanisms into OpenSSL v1.0.1f and added ciphersuites, both hybrid and non-hybrid, to the TLS 1.2 implementation in OpenSSL. We compiled the Apache httpd v2.4.20 web server against our modified OpenSSL, and tested compatibility and performance of the web server. We encountered no problems with existing clients despite using larger ephemeral public keys / encapsulations, and did not need to make any modifications to data structures (e.g., existing 16-bit length fields were large enough to hold our values).

We measured throughput (connections per second) for a variety of page sizes, and latency (connection establishment time) for a server with or without heavy load, of both hybrid and non-hybrid ciphersuites. Detailed results including the exact methodology can be found in [24]. To highlight a few results: the connection time of an ECDHE (nistp256) ciphersuite with an RSA certificate on an unloaded server was 16.1 milliseconds (over a network with ping time 0.62 ms); it was 20.7 ms for FrodoCCS, and 24.5 ms for hybrid FrodoCCS+ECDHE[10]. The number of connections (with 1 KiB HTTP payload) supported per second with an ECDHE ciphersuite with an RSA certificate was 810, compared to 700 for FrodoCCS and 551 for hybrid FrodoCCS+ECDHE. These results indicate that, despite its larger communication sizes, FrodoKEM remains practical for Internet applications.

In our experience with testing the performance of the original Frodo construction in an end-to-end testbed OpenSSL deployment, we observed a few trends that let us extrapolate these results to our current proposal. First, we note that even with the significantly larger bandwidth of the original FrodoCCS proposal, as compared to the original NewHope proposal, we observed a slowdown of less than 1.6× when comparing

[9] This count does not include header files and the additional symmetric primitives.

[10] Note that the results in [24] use a different parameter set than in this proposal which had slightly larger communication (22.1 KiB in [24] versus 18.9 KiB for FrodoKEM-640 in this proposal; the IND-CCA-secure FrodoKEM-640 in this proposal has an additional runtime cost in decapsulation due to the application of the FO transform compared to the IND-CPA-secure scheme in [24]; and used somewhat different symmetric primitives. Nonetheless the results provide some indication of suitability.

connection times for 1 KiB webpages. This slowdown factor only decreases with increasing sizes of webpages and considering our smaller bandwidth (18.9 KiB for FrodoKEM-640 versus 22.1 KiB for the original FrodoCCS construction) we expect to be competitive for typical connection sizes.

Moreover, we can state with some measure of confidence that the additional costs when applying the FO transform will have a very small impact on the connection throughput as well as on the connection times. We state this with two supporting arguments. First, with a microbenchmark a whole order of magnitude faster than the original FrodoCCS construction, the original NewHope construction only improves connection times and throughputs by 30–50% and we expect various other bottlenecks in the entire Web serving ecosystem to have a larger impact. To compare, our FO-transformed implementations run in time a small constant factor larger than the microbenchmarks of FrodoCCS. Second—and as stated previously to support the practical application of the original FrodoCCS construction [24]—deployments in the near-term will necessarily involve both a post-quantum and a traditional EC-based construction which would result in any drastic improvements in post-quantum microbenchmarks having a small or even negligible impact in practical deployments. The costs of these small impacts are well worth the long-term post-quantum security afforded by a conservative scheme based only on generic lattices.

6.3 Hardware implementations

Hardware implementations of lattice-based cryptographic schemes have mainly considered the ring learning with errors based schemes (see, e.g., [59, 97, 98, 106]) since these schemes allow to compute polynomial multiplication with the number-theoretic transform, e.g. the discrete Fourier transform over a finite field. Computing the fast Fourier transform (FFT) is a well-known primitive for hardware implementations.

Schemes based on the original learning with errors problem work with matrices instead. Fortunately, the FPGA design and implementation of, for instance, matrix multiplication architectures is a well-studied area and very efficient (in terms of either area, energy or performance) implementations are known (cf. [100] and the related literature mentioned therein). Hence, the proposed schemes FrodoKEM-640 and FrodoKEM-976 are a natural fit for hardware implementations.

6.4 Side-channel resistance

Side-channel attacks are a family of attacks which use meta-information such as power consumption (e.g, in a differential power analysis (DPA) attack [68]) or electromagnetic usage (e.g., in a differential electromagnetic analysis (DEMA) attack [54]) in a statistical analysis by correlating this information obtained when executing a cryptographic primitive to a key-dependent guess. Besides such passive side-channel attacks (cf. [67]) there are also active attacks which might *inject faults* [21, 19] and use the potentially corrupted output to obtain information about the secret key used.

This is a well-studied and active research area used to protect software and hardware implementations where such attacks are realistic. In the setting of implementations based on the ring LWE problems not much work has been done yet. For ring LWE masking techniques [34] have been studied to protect implementations such as in [87, 104, 105].

In a more recent work [99] it is shown how to perform a single trace attack on ring LWE encryption using side-channel template matching [35]. Hence, it can also be applied to attack masked implementations. This single trace behaviour makes it immediately applicable to key-exchange algorithms.

No side-channel attacks nor countermeasures are currently known for LWE key encapsulation mechanisms but the generic attacks methods as well as the countermeasures which apply to ring LWE also do apply to LWE. However, since our LWE-based schemes FrodoKEM-640 and FrodoKEM-976 do not use FFT-based multiplication techniques (the point of attack used in [99]), the attack surface against FrodoKEM is significantly reduced. This might result in cheap and easy-to-apply countermeasures against a large set of the known side-channel attacks applied in practice.

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 5–17. ACM Press, Oct. 2015.

[2] D. Aharonov and O. Regev. Lattice problems in NP ∩ coNP. *Journal of the ACM*, 52(5):749–765, 2005. Preliminary version in FOCS 2004.

[3] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *28th Annual ACM Symposium on Theory of Computing*, pages 99–108. ACM Press, May 1996.

[4] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *29th Annual ACM Symposium on Theory of Computing*, pages 284–293. ACM Press, May 1997.

[5] M. R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 103–129. Springer, Heidelberg, May 2017.

[6] M. R. Albrecht, C. Cid, J.-C. Faugère, and L. Perret. Algebraic algorithms for LWE. Cryptology ePrint Archive, Report 2014/1018, 2014. http://eprint.iacr.org/2014/1018.

[7] M. R. Albrecht, J.-C. Faugère, R. Fitzpatrick, and L. Perret. Lazy modulus switching for the BKW algorithm on LWE. In H. Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 429–445. Springer, Heidelberg, Mar. 2014.

[8] M. R. Albrecht, R. Fitzpatrick, and F. Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In H.-S. Lee and D.-G. Han, editors, *ICISC 13: 16th International Conference on Information Security and Cryptology*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310. Springer, Heidelberg, Nov. 2014.

[9] M. R. Albrecht, F. Göpfert, F. Virdia, and T. Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. Cryptology ePrint Archive, Report 2017/815, 2017. http://eprint.iacr.org/2017/815.

[10] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, Nov 2015.

[11] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - a new hope. In *25th USENIX Security Symposium*, pages 327–343, 2016.

[12] A. Andoni, T. Laarhoven, I. P. Razenshteyn, and E. Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In P. N. Klein, editor, *28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 47–66. ACM-SIAM, Jan. 2017.

[13] B. Applebaum, D. Cash, C. Peikert, and A. Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, Heidelberg, Aug. 2009.

[14] S. Arora and R. Ge. New algorithms for learning in presence of errors. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, Heidelberg, July 2011.

[15] S. Bai, A. Langlois, T. Lepoint, D. Stehlé, and R. Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. In T. Iwata and J. H. Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 3–24. Springer, Heidelberg, Nov. / Dec. 2015.

[16] A. Becker, L. Ducas, N. Gama, and T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In R. Krauthgamer, editor, *27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 10–24. ACM-SIAM, Jan. 2016.

[17] A. Becker, N. Gama, and A. Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522, 2015. http://eprint.iacr.org/2015/522.

[18] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, E. Lambooij, T. Lange, R. Niederhagen, and C. van Vredendaal. How to manipulate curve standards: A white paper for the black hat. In L. Chen and S. Matsuo, editors, *Security Standardisation Research (SSR) 2015*, volume 9497 of *Lecture Notes in Computer Science*, pages 109–139. Springer, 2015.

[19] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, Heidelberg, Aug. 1997.

[20] A. Blum, M. L. Furst, M. J. Kearns, and R. J. Lipton. Cryptographic primitives based on hard learning problems. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 278–291. Springer, Heidelberg, Aug. 1994.

[21] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, Heidelberg, May 1997.

[22] D. Boneh, C. Gentry, S. Gorbunov, S. Halevi, V. Nikolaenko, G. Segev, V. Vaikuntanathan, and D. Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 533–556. Springer, Heidelberg, May 2014.

[23] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[24] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 1006–1018. ACM Press, Oct. 2016.

[25] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society Press, May 2015.

[26] J. W. Bos, M. Naehrig, and J. van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. *International Journal of Applied Cryptography*, 2016. to appear, http://eprint.iacr.org/2014/880.

[27] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computer Theory*, 6(3):13, 2014. Preliminary version in ITCS 2012.

[28] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. Classical hardness of learning with errors. In D. Boneh, T. Roughgarden, and J. Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 575–584. ACM Press, June 2013.

[29] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In R. Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 97–106. IEEE Computer Society Press, Oct. 2011.

[30] J. Cai and A. Nerurkar. An improved worst-case to average-case connection for lattice problems. In *38th Annual Symposium on Foundations of Computer Science*, pages 468–477. IEEE Computer Society Press, Oct. 1997.

[31] P. Campbell, M. Groves, and D. Shepherd. Soliloquy: a cautionary tale. ETSI 2nd Quantum-Safe Crypto Workshop, 2014. http://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.

[32] D. Cash, D. Hofheinz, E. Kiltz, and C. Peikert. Bonsai trees, or how to delegate a lattice basis. *Journal of Cryptology*, 25(4):601–639, Oct. 2012.

[33] W. Castryck, I. Iliashenko, and F. Vercauteren. Provably weak instances of ring-LWE revisited. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 147–167. Springer, Heidelberg, May 2016.

[34] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, Heidelberg, Aug. 1999.

[35] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. Kaliski Jr., Çetin Kaya. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, Heidelberg, Aug. 2003.

[36] S. Chatterjee, N. Koblitz, A. Menezes, and P. Sarkar. Another look at tightness II: Practical issues in cryptography. In *Paradigms in Cryptology – Mycrypt 2016*, volume 10311 of *Lecture Notes in Computer Science*, pages 21–25. Springer, Heidelberg, 2017.

[37] H. Chen, K. Lauter, and K. E. Stange. Attacks on search RLWE. Cryptology ePrint Archive, Report 2015/971, 2015. http://eprint.iacr.org/2015/971.

[38] H. Chen, K. Lauter, and K. E. Stange. Vulnerable Galois RLWE families and improved attacks. Cryptology ePrint Archive, Report 2016/193, 2016. http://eprint.iacr.org/2016/193.

[39] Y. Chen. *Lattice reduction and concrete security of fully homomorphic encryption*. PhD thesis, l'Université Paris Diderot, 2013. http://www.di.ens.fr/~ychen/research/these.pdf.

[40] Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, Heidelberg, Dec. 2011.

[41] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, Heidelberg, Aug. 2005.

[42] R. Cramer, L. Ducas, C. Peikert, and O. Regev. Recovering short generators of principal ideals in cyclotomic rings. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 559–585. Springer, Heidelberg, May 2016.

[43] R. Cramer, L. Ducas, and B. Wesolowski. Short Stickelberger class relations and application to ideal-SVP. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 324–348. Springer, Heidelberg, May 2017.

[44] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[45] D. Dadush, O. Regev, and N. Stephens-Davidowitz. On the closest vector problem with a distance guarantee. In *IEEE Conference on Computational Complexity*, pages 98–109, 2014.

[46] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[47] J. Ding, X. Xie, and X. Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Report 2012/688, 2012. http://eprint.iacr.org/2012/688.

[48] N. Döttling and J. Müller-Quade. Lossy codes and a new variant of the learning-with-errors problem. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 18–34. Springer, Heidelberg, May 2013.

[49] M. J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards (FIPS) 202, National Institute of Standards and Technology, Aug. 2015.

[50] Y. Elias, K. E. Lauter, E. Ozman, and K. E. Stange. Provably weak instances of ring-LWE. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 63–92. Springer, Heidelberg, Aug. 2015.

[51] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, Heidelberg, Aug. 1999.

[52] N. Gama and P. Q. Nguyen. Predicting lattice reduction. In N. P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, Heidelberg, Apr. 2008.

[53] N. Gama, P. Q. Nguyen, and O. Regev. Lattice enumeration using extreme pruning. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 257–278. Springer, Heidelberg, May 2010.

[54] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, Heidelberg, May 2001.

[55] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In R. E. Ladner and C. Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 197–206. ACM Press, May 2008.

[56] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, Heidelberg, Aug. 2013.

[57] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. Cryptology ePrint Archive, Report 1996/009, 1996. http://eprint.iacr.org/1996/009.

[58] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Predicate encryption for circuits from LWE. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 503–523. Springer, Heidelberg, Aug. 2015.

[59] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, Heidelberg, Sept. 2012.

[60] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In *ANTS*, pages 267–288, 1998.

[61] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Y. Kalai and L. Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, Heidelberg, Nov. 2017.

[62] T. Ishiguro, S. Kiyomoto, Y. Miyake, and T. Takagi. Parallel Gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In H. Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 411–428. Springer, Heidelberg, Mar. 2014.

[63] H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma. Post-quantum IND-CCA-secure KEM without additional hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[64] J. Kelsey, S. Chang, and R. Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. NIST Special Publication 800-185. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf.

[65] P. Kirchner and P.-A. Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 43–62. Springer, Heidelberg, Aug. 2015.

[66] P. Kirchner and P.-A. Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 3–26. Springer, Heidelberg, May 2017.

[67] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, Heidelberg, Aug. 1996.

[68] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, Heidelberg, Aug. 1999.

[69] T. Laarhoven. *Search problems in cryptography.* PhD thesis, Eindhoven University of Technology, 2015.

[70] T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 3–22. Springer, Heidelberg, Aug. 2015.

[71] T. Laarhoven, M. Mosca, and J. van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2-3):375–400, 2015.

[72] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[73] A. Langlois, D. Stehlé, and R. Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 239–256. Springer, Heidelberg, May 2014.

[74] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, December 1982.

[75] R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In A. Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, Heidelberg, Feb. 2011.

[76] M. Liu and P. Q. Nguyen. Solving BDD by enumeration: An update. In E. Dawson, editor, *Topics in Cryptology – CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 293–309. Springer, Heidelberg, Feb. / Mar. 2013.

[77] Y.-K. Liu, V. Lyubashevsky, and D. Micciancio. On bounded distance decoding for general lattices. In *APPROX-RANDOM*, volume 4110 of *Lecture Notes in Computer Science*, pages 450–461. Springer, Heidelberg, 2006.

[78] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, November 2013. Preliminary version in Eurocrypt 2010.

[79] A. Mariano, T. Laarhoven, and C. Bischof. A parallel variant of LDSieve for the SVP on lattices. In *25th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing (PDP)*, pages 23–30. IEEE, 2017.

[80] U. M. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, Heidelberg, Feb. 2004.

[81] D. Micciancio. Improved cryptographic hash functions with worst-case/average-case connection. In *34th Annual ACM Symposium on Theory of Computing*, pages 609–618. ACM Press, May 2002.

[82] D. Micciancio. Cryptographic functions from worst-case complexity assumptions. Information Security and Cryptography, pages 427–452. Springer, Heidelberg, 2010.

[83] D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 21–39. Springer, Heidelberg, Aug. 2013.

[84] D. Micciancio and O. Regev. Worst-case to average-case reductions based on Gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007. Preliminary version in FOCS 2004.

[85] D. Micciancio and O. Regev. Lattice-based cryptography. In *Post Quantum Cryptography*, pages 147–191. Springer, February 2009.

[86] P. Q. Nguyen and O. Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. *Journal of Cryptology*, 22(2):139–160, Apr. 2009.

[87] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-secure and masked ring-LWE implementation. Cryptology ePrint Archive, Report 2016/1109, 2016. http://eprint.iacr.org/2016/1109.

[88] C. Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In M. Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 333–342. ACM Press, May / June 2009.

[89] C. Peikert. Some recent progress in lattice-based cryptography. In O. Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, page 72. Springer, Heidelberg, Mar. 2009. Invited talk. Slides available at http://web.eecs.umich.edu/~cpeikert/pubs/slides-tcc09.pdf.

[90] C. Peikert. An efficient and parallel Gaussian sampler for lattices. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, Heidelberg, Aug. 2010.

[91] C. Peikert. Lattice cryptography for the Internet. In M. Mosca, editor, *PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 197–219. Springer, Heidelberg, 2014.

[92] C. Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.

[93] C. Peikert. How (not) to instantiate ring-LWE. In V. Zikas and R. De Prisco, editors, *SCN 16: 10th International Conference on Security in Communication Networks*, volume 9841 of *Lecture Notes in Computer Science*, pages 411–430. Springer, Heidelberg, Aug. / Sept. 2016.

[94] C. Peikert, O. Regev, and N. Stephens-Davidowitz. Pseudorandomness of ring-LWE for any ring and modulus. In H. Hatami, P. McKenzie, and V. King, editors, *49th Annual ACM Symposium on Theory of Computing*, pages 461–473. ACM Press, June 2017.

[95] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571. Springer, Heidelberg, Aug. 2008.

[96] C. Peikert and B. Waters. Lossy trapdoor functions and their applications. In R. E. Ladner and C. Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 187–196. ACM Press, May 2008.

[97] T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In A. Hevia and G. Neven, editors, *Progress in Cryptology - LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, Heidelberg, Oct. 2012.

[98] T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, Heidelberg, Aug. 2014.

[99] R. Primas, P. Pessl, and S. Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, Heidelberg, Sept. 2017.

[100] S. M. Qasim, A. A. Telba, and A. Y. AlMazroo. FPGA design and implementation of matrix multiplier architectures for image and signal processing applications. *International Journal of Computer Science and Network Security*, 10(2):168–176, 2010.

[101] O. Regev. New lattice-based cryptographic constructions. *J. ACM*, 51(6):899–942, 2004. Preliminary version in STOC 2003.

[102] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34, 2009. Preliminary version in STOC 2005.

[103] O. Regev. The learning with errors problem (invited survey). In *IEEE Conference on Computational Complexity*, pages 191–204, 2010.

[104] O. Reparaz, R. de Clercq, S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. Additively homomorphic ring-LWE masking. In T. Takagi, editor, *PQCrypto 2016*, volume 9606 of *Lecture Notes in Computer Science*, pages 233–244. Springer, Heidelberg, 2016.

[105] O. Reparaz, S. Sinha Roy, R. de Clercq, F. Vercauteren, and I. Verbauwhede. Masking ring-LWE. *Journal of Cryptographic Engineering*, 6(2):139–153, 2016.

[106] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-LWE cryptoprocessor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, Heidelberg, Sept. 2014.

[107] M. Schneider. Sieving for shortest vectors in ideal lattices. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *AFRICACRYPT 13: 6th International Conference on Cryptology in Africa*, volume 7918 of *Lecture Notes in Computer Science*, pages 375–391. Springer, Heidelberg, June 2013.

[108] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.

[109] A. Shamir. A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology – CRYPTO'82*, pages 279–288. Plenum Press, New York, USA, 1982.

[110] D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, Heidelberg, May 2011.

[111] E. E. Targhi and D. Unruh. Post-quantum security of the Fujisaki-Okamoto and OAEP transforms. In M. Hirt and A. D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 192–216. Springer, Heidelberg, Oct. / Nov. 2016.

Name of Proposal:

Gui

Principal Submitter:
Jintai Ding

email: jintai.ding@gmail.com
phone: 513 556 - 4024
organization: University of Cincinnati
postal address: 4314 French Hall, OH 45221 Cincinnati, USA

Auxiliary Submitters: Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang

Inventors: c.f. Submitters

Owners: c.f. Submitters

Jintai Ding (Signature)

Additional Point of Contact:
Bo-Yin Yang
email: by@crypto.tw
phone: 886-2-2788-3799
Fax: 886-2-2782-4814
organization: Academia Sinica
postal address: 128 Academia Road, Section 2
Nankang, Taipei 11529, Taiwan

Gui - Algorithm Specification and Documentation

Type: Signature scheme

Family: Multivariate Cryptography, BigField schemes

The Gui signature scheme as described in this proposal is based on the HFEv- signature scheme, which was first proposed by Patarin, Courtois and Goubin in [12]. Similar to Gui, their QUARTZ scheme uses a specially designed signature generation process which allows to reduce key and signature sizes compared to the original HFEv- design. However, while QUARTZ uses an HFE polynomial of high degree as well as small values for the numbers of minus equations and vinegar variables, Gui follows another approach. By decreasing the degree of the HFE polynomial in use while increasing the numbers of minus equations and vinegar variables, we can speed up the signature generation process of the scheme drastically without weakening its security.

1 Algorithm Specification

In this section we present the Gui signature scheme as proposed in [15].

1.1 Parameters

- $\mathbb{F} = \mathbb{F}_q$: finite field with q elements, $q = 2^e$
- $\mathbb{E} = \mathbb{F}_{q^n}$: degree n extension field of $\mathbb{F}$

- $\phi : \mathbb{F}^n \to \mathbb{E}$: isomorphism between the vector space $\mathbb{F}^n$ and the extension field $\mathbb{E}$
- D: degree of the HFE polynomial, set $r = \lfloor \log_q(D-1) \rfloor + 1$.
- a: number of minus equations
- v: number of vinegar variables
- k: repetition factor (used in signature generation)
- number of equations: $n - a$
- number of variables: $n + v$

1.2 Key Generation

Private Key. The private key consists of the three maps

- $\mathcal{S} : \mathbb{F}^n \to \mathbb{F}^{n-a}$: affine transformation of maximal rank
- $\mathcal{T} : \mathbb{F}^{n+v} \to \mathbb{F}^{n+v}$: invertible affine transformation
- central map $\mathcal{F} : \mathbb{E} \times \mathbb{F}^v \to \mathbb{E}$

$$\mathcal{F}(X) = \sum_{\substack{0 \le i \le j \\ q^i + q^j \le D}} \alpha_{i,j} X^{q^i + q^j} + \sum_{\substack{0 \le i \\ q^i \le D}} \beta_i(v_1, \ldots, v_v) \cdot X^{q^i} + \gamma(v_1, \ldots, v_v). \quad (1)$$

Here, the $\beta_i : \mathbb{F}^v \to \mathbb{E}$ are linear (affine) functions in the vinegar variables $v_1, \ldots, v_v$, while $\gamma : \mathbb{F}^v \to \mathbb{E}$ is a quadratic function in $v_1, \ldots, v_v$.

Due to the special structure of $\mathcal{F}$, the map $\bar{\mathcal{F}} = \phi^{-1} \circ \mathcal{F} \circ (\phi \times id_v)$ is a quadratic multivariate map from $\mathbb{F}^{n+v}$ to $\mathbb{F}^n$.

The size of the private key is

$$\underbrace{(n-a)\cdot(n+1)}_{\text{affine map } \mathcal{S}} + \underbrace{(n+v)\cdot(n+v+1)}_{\text{affine map } \mathcal{T}} + \underbrace{n \cdot \left(\#\alpha + (\lfloor \log_q D \rfloor + 1) \cdot (v+1) + \frac{(v+1)\cdot(v+2)}{2} \right)}_{\text{central map } \mathcal{F}}$$

$\mathbb{F}_q$-elements. Here, $\#\alpha$ denotes the number of non-zero coefficients α in equation (1), which is upper bounded by $\frac{r\cdot(r+1)}{2}$.

Public Key. The public key of Gui is the composed map

$$\mathcal{P} = \mathcal{S} \circ \phi^{-1} \circ \mathcal{F} \circ (\phi \times id_v) \circ \mathcal{T} : \mathbb{F}^{n+v} \to \mathbb{F}^{n-a}.$$

consisting of $n-a$ quadratic poltynomials in $n+v$ variables. The size of the public key is

$$(n-a) \cdot \frac{(n+v+1) \cdot (n+v+2)}{2}$$

$\mathbb{F}_q$-elements.

1.3 Signature Generation

Given a document d to be signed, we first compute hash values $\mathbf{d}_1, \ldots, \mathbf{d}_k \in \mathbb{F}_q^{n-a}$ as follows. For a standard hash function $\mathcal{H} : \{0,1\}^\star \to \{0,1\}^{n'}$ we compute a bitstring

$$\tilde{\mathbf{h}} = \mathcal{H}(d)||\mathcal{H}(\mathcal{H}(d))|| \ldots ||\mathcal{H}^\ell(d)$$

of length $k \cdot \log_2 q \cdot (n-a) \leq |\tilde{\mathbf{h}}| < (k+1) \cdot \log_2 q \cdot (n-a)$. Here, $\mathcal{H}^\ell(d)$ denotes the ℓ-times repeated application of the hash function $\mathcal{H}$. We set

$$\mathbf{d}_i = (\tilde{\mathbf{h}}_{(i-1)\cdot\log_2 q\cdot(n-a)+1}|| \ldots ||\tilde{\mathbf{h}}_{i\cdot\log_2 q\cdot(n-a)}) \quad (i = 1, \ldots, k)$$

and transform each $\mathbf{d}_i$ into a vector of $(n-a)$ $\mathbb{F}_q$-elements.
The last $\ell \cdot n' - k \cdot \log_2 q \cdot (n-a)$ bits of $\tilde{\mathbf{h}}$ are skipped.

After having computed the hash values $\mathbf{d}_i \in \mathbb{F}^{n-a}$ $(i = 1, \ldots, k)$, we obtain a Gui signature for the message d as follows.

We set $S_0 = \mathbf{0}^{n-a}$ and perform for $i = 1$ to k the following steps

1. Compute a preimage $\mathbf{x} \in \mathbb{F}^n$ of $\mathbf{d}_i \oplus S_{i-1}$ under the affine map $\mathcal{S}$ and lift it to the extension field, obtaining $X \in \mathbb{E}$.

2. Choose random values for the vinegar variables $v_1, \ldots, v_v$ and substitute them into the central map to obtain the parametrized map $\mathcal{F}_V : \mathbb{E} \to \mathbb{E}$.

3. Find a solution to the equation $\mathcal{F}_V(Y) = X$ by performing the first step of the Cantor-Zassenhaus algorithm, i.e. compute $\tilde{Y} = \gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y})$. For efficiency reasons, we repeat step 2 and 3 until the gcd is a linear polynomial. Denote the unique root of $\tilde{Y}$ by $Y \in \mathbb{E}$.

4. Move Y down to the vector space $\mathbb{F}^n$, obtaining $\mathbf{y}' = (y_1, \ldots, y_n)$, and append the vinegar variables of step 2, obtaining $\mathbf{y} = (\mathbf{y}'||v_1|| \ldots ||v_v) \in \mathbb{F}^{n+v}$.

5. Compute $\mathbf{z} = (z_1, \ldots, z_{n+v}) = \mathcal{T}^{-1}(\mathbf{y})$ and set $S_i = (z_1, \ldots, z_{n-a})$, $X_i = (z_{n-a+1}, \ldots, z_{n+v})$.

The final Gui signature of the message d has the form $\sigma = (S_k||X_k|| \ldots ||X_1) \in \mathbb{F}^{(n-a)+k\cdot(a+v)}$.

1.4 Signature Verification

In order to check the authenticity of a signature $\sigma = (S_k||X_k||\ldots||X_1) \in \mathbb{F}^{(n-a)+k\cdot(a+v)}$, we first compute the hash values $\mathbf{d}_i \;\; (i = 1, \ldots, k)$ as described in the previous section.
After that, we perform for $i = k-1, \ldots, 0$ the two steps

1. Evaluate the public key at $(S_{i+1}||X_{i+1})$. Denote the result by $\mathbf{w} \in \mathbb{F}^{n-a}$.
2. Set $S_i = \mathbf{w} \oplus \mathbf{d}_{i+1}$.

The signature σ is accepted, if and only if $S_0 = \mathbf{0}^{n-a}$ holds.

In order to find a good balance between security and efficiency, the Gui (and HFEv-) signature scheme is mainly used over small finite fields. Since the complexity of the signature generation process is $O(n \cdot D^3)$, one aims at choosing the degree D of the HFE polynomial in use as small as possible. On the other hand, for security reasons, the number of quadratic terms in the HFE polynomial (1) and therefore the value $r = \lfloor \log_q(D-1) \rfloor + 1$ should not be too small. For a fixed D, we therefore can increase r and the number of quadratic terms in the HFEv-polynomial by simply decreasing q. So, a small value of q allows to achieve both good performance and high security. In order to reduce the number of equations in the public system and therefore key and signature sizes, one introduces a repetition factor k. To reach a security level of ℓ bits under collision attacks, one would need, for the pure HFEv- scheme ($k = 1$), $2 \cdot \ell / \log_2(q)$ equations in the public key, leading to a very large public key. By choosing $k > 1$, we can omit this problem by using a hash of effective length $k \cdot |\mathcal{H}|$. However, choosing $k > 2$ brings no advantage, but increases the signature size.

The following algorithms `GuiKeyGen`, `GuiSign` and `GuiVer` show the key generation, signature generation and signature verification processes of Gui in algorithmic form.

Algorithm 1 `GuiKeyGen`: Key Generation of Gui

Input: Gui parameters (q, n, D, a, v), isomorphism $\phi : \mathbb{F}_q^n \to \mathbb{E}$
Output: Gui key pair (sk, pk)

1: **repeat**
2: $\quad M_S \leftarrow \texttt{Matrix}(q, n, n)$
3: **until** IsInvertible(M_S) == **TRUE**
4: $c_S \leftarrow_R \mathbb{F}^n$
5: $\mathcal{S} \leftarrow \texttt{Aff}(M_S, c_S)$
6: $InvS \leftarrow M_S^{-1}$
7: **repeat**
8: $\quad M_T \leftarrow \texttt{Matrix}(q, n+v, n+v)$
9: **until** IsInvertible(M_T) == **TRUE**
10: $c_T \leftarrow_R \mathbb{F}^{n+v}$
11: $\mathcal{T} \leftarrow \texttt{Aff}(M_T, c_T)$
12: $InvT \leftarrow M_T^{-1}$
13: $\mathcal{F} \leftarrow \texttt{HFEvmap}(q, n, D, a, v)$
14: $\mathcal{P} \leftarrow \mathcal{S} \circ \phi^{-1} \circ \mathcal{F} \circ (\phi \times id_v) \circ \mathcal{T}$
15: $sk \leftarrow (InvS, c_S, \mathcal{F}, InvT, c_T)$
16: $pk \leftarrow \mathcal{P}$
17: **return** (sk, pk)

The possible input values of algorithm `GuiKeyGen` are specified in Section 1.8. The function $\texttt{Matrix}(q, m, n)$ returns an $m \times n$ matrix with coefficients chosen uniformly at random in $\mathbb{F}_q$. $\texttt{Aff}(M, c)$ returns the affine map $M \cdot x + c$. $\texttt{HFEvmap}(q, D, a, v, \beta_i, \gamma)$ outputs an HFEv central map (see equation (1)) with randomly chosen coefficients $\alpha \in \mathbb{E}$ and randomly chosen vinegar maps β_i $(i = 0, \ldots, \lfloor \log_q D \rfloor)$ and γ.

The algorithm `GuiKeyGen` makes use of

$$\underbrace{n \cdot (n+1)}_{\text{affine map } \mathcal{S}} + \underbrace{(n+v) \cdot (n+v+1)}_{\text{affine map } \mathcal{T}} + \underbrace{n \cdot \left(\#\alpha + (\lfloor \log_q D \rfloor + 1) \cdot (v+1) + \frac{(v+1) \cdot (v+2)}{2} \right)}_{\text{central map } \mathcal{F}}$$

randomly generated field elements. Here, $\#\alpha$ denotes the number of non-zero coefficients α in equation (1), which is upper bounded by $\frac{r \cdot (r+1)}{2}$. In contrast to the description in Section 1.2, we use here and in our implementation an invertible matrix $M_S \in \mathbb{F}_q^{n \times n}$. While this increases the private key size slightly, it speeds up the signature generation significantly.

Algorithm 2 `GuiSign`: Signature Generation Process of Gui

Input: Gui private key ($InvS$, c_S, $\mathcal{F}$, $InvT$, c_T), message d, repetition factor k
Output: signature $\sigma \in \mathbb{F}^{(n-a)+k\cdot(a+v)}$

1: $\ell \leftarrow \lceil k \cdot \log_2 q \cdot (n-a)/|\mathcal{H}| \rceil$
2: $\tilde{\mathbf{h}} \leftarrow \mathcal{H}(d)||\mathcal{H}(\mathcal{H}(d))|| \ldots ||\mathcal{H}^{\ell}(d)$
3: $S_0 \leftarrow \mathbf{0}^{n-a}$
4: **for** $i = 1$ to k **do**
5: $\quad \mathbf{d}_i \leftarrow \mathbb{F}^{n-a}!(\tilde{\mathbf{h}}_{(i-1)\cdot\log_2 q\cdot(n-a)+1}, \ldots, \tilde{\mathbf{h}}_{i\cdot\log_2 q\cdot(n-a)})$
6: $\quad (S_i, X_i) \leftarrow \texttt{InvHFEv-}(\mathbf{d}_i \oplus S_{i-1})$
7: **end for**
8: $\sigma \leftarrow (S_k||X_k|| \ldots ||X_1)$
9: **return** σ

Algorithm 3 `InvHFEv-`: Inversion of the HFEv- public key

Input: Gui private key ($InvS$, c_S, $\mathcal{F}$, $InvT$, c_T), isomorphism $\phi : \mathbb{F}_q^n \to \mathbb{E}$, vector $\mathbf{w} \in \mathbb{F}^{n-a}$,
Output: vector $\mathbf{z} \in \mathbb{F}^{n+v}$ such that $\mathcal{P}(\mathbf{z}) = \mathbf{w}$

1: $r_1, \ldots, r_a \leftarrow_R \mathbb{F}$
2: $\mathbf{x} \leftarrow InvS \cdot ((\mathbf{w}||r_1|| \ldots ||r_a) - c_S)$
3: $X \leftarrow \phi(\mathbf{x})$
4: **repeat**
5: $\quad v_1, \ldots, v_v \leftarrow_R \mathbb{F}$
6: $\quad \mathcal{F}_V \leftarrow \mathcal{F}(v_1, \ldots, v_v)$
7: $\quad Y \leftarrow \gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y})$
8: **until** $\deg(Y) == 1$
9: $\mathbf{y} \leftarrow \phi^{-1}(\texttt{root}(Y))$
10: $\mathbf{z} \leftarrow InvT \cdot ((\mathbf{y}||v_1|| \ldots ||v_v) - c_T)$
11: **return z**

In line 1 of Algorithm `GuiSign`, $|\mathcal{H}|$ denotes the output length of the hash function in use. $\mathbb{F}^{n-a}!(\mathbf{h})$ coerces the binary vector $\mathbf{h} \in \mathrm{GF}(2)^{\log_2 q\cdot(n-a)}$ into a vector in $\mathbb{F}^{n-a}$. [1]

In Algorithm `InvHFEv-`, we perform the loop (line 4 to 8) until the computed gcd is a linear polynomial (i.e. if the equation $\mathcal{F}_V(\hat{Y}) = X$ has a unique solution). In this case, the function $\texttt{root}(Y)$ returns the unique root of the linear polynomial $Y \in \mathbb{E}[\hat{Y}]$. Otherwise, we choose other values for the vinegar variables $v_1, \ldots, v_v$ and try again. Though this check requires more gcd computations, it is still more efficient than computing the roots of a polynomial Y of higher degree.

During the signature generation of Gui, we make use of approximately $k\cdot(a+e\cdot v)$ randomly chosen field elements (in line 2 and 6 of algorithm `InvHFEv-`). The reason for this is that, in order to find a unique solution of $\mathcal{F}_V(\hat{Y}) = X$ (see

[1] In the case of $\mathbb{F} = \mathrm{GF}(2)$, this function does not do anything.

line 8 of Algorithm `InvHFEv-`), we have to run the loop about e times, while the whole algorithm is performed k times..

Algorithm 4 `GuiVer`: Signature Verification Process of Gui

Input: Gui public key $\mathcal{P}$, message d, repetition factor k, signature $\sigma \in \mathbb{F}^{(n-a)+k(a+v)}$

Output: boolean value **TRUE** or **FALSE**

1: $\ell \leftarrow \lceil k \cdot \log_2 q \cdot (n-a)/|\mathcal{H}| \rceil$
2: $\tilde{\mathbf{h}} \leftarrow \mathcal{H}(d)||\mathcal{H}(\mathcal{H}(d))|| \ldots ||\mathcal{H}^{\ell}(d)$
3: **for** $i = 1$ to k do **do**
4: $\quad \mathbf{d}_i \leftarrow \mathbb{F}^{n-a}!(\tilde{\mathbf{h}}_{(i-1)\cdot\log_2 q\cdot(n-a)+1}, \ldots, \tilde{\mathbf{h}}_{i\cdot\log_2 q\cdot(n-a)})$
5: **end for**
6: **for** $i = k-1$ to 0 **do**
7: $\quad S_i \leftarrow \mathcal{P}(S_{i+1}||X_{i+1}) \oplus \mathbf{d}_{i+1}$
8: **end for**
9: **if** $S_0 = \mathbf{0}$ **then**
10: $\quad$ **return TRUE**
11: **else**
12: $\quad$ **return FALSE**
13: **end if**

1.5 Remark on Correctness

In the signature generation process, we start with $S_0 = \mathbf{0}$ and set recursively $(S_i||X_i) = \texttt{InvHFEv}-(\mathbf{d}_i \oplus S_{i-1})$ until we finally obtain the signature $\sigma = (S_k||X_k|| \ldots ||X_1)$.
During the signature verification process, we start with $\tilde{S}_k = S_k$ and compute recursively $\tilde{S}_{i-1} \leftarrow \texttt{HFEv}-(\tilde{S}_i||X_i) \oplus \mathbf{d}_i$ until we get $\tilde{S}_0$. We find

$$\begin{aligned} \tilde{S}_{k-1} &= \texttt{HFEv}-(S_k||X_k) \oplus \mathbf{d}_k = \texttt{HFEv}-(\texttt{InvHFEv}-(\mathbf{d}_k \oplus S_{k-1})) \oplus \mathbf{d}_k \\ &= \mathbf{d}_k \oplus S_{k-1} \oplus \mathbf{d}_k = S_{k-1}. \end{aligned}$$

Analogously we obtain $\tilde{S}_{k-2} = S_{k-2}, \ldots, \tilde{S}_0 = S_0 = \mathbf{0} \in \mathbb{F}^{n-a}$. Therefore, a honestly generated signature will always be accepted.

1.6 Changes needed to achieve EUF-CMA Security

The standard Gui signature scheme as described above provides only universal unforgeability. In order to obtain EUF-CMA security, we apply a transformation similar to that in [16]. The main difference is the use of a random binary vector r called salt. Instead of generating a signature for the hash value $\mathbf{h} = \mathcal{H}(d)$, we generate a signature for $\mathcal{H}(\mathcal{H}(d)||r)$. The resulting signature has the form $\sigma^{\star} = (\sigma, r)$, where σ is a standard Gui signature. By doing so, we ensure that an attacker is not able to forge any hash/signature pair.

In particular, we apply the following changes to the algorithms `GuiKeyGen`, `GuiSign` and `GuiVer`.

- In the algorithm `GuiKeyGen`*, we choose an integer $\bar{\ell}$ as the length of the random salt; $\bar{\ell}$ is appended both to the private and public key.
- In the algorithm `GuiSign`*, we choose first randomly the values of the $k \cdot (a + v)$ random variables (the random values for the affine map $\mathcal{S}$ and the vinegar variables of the central map); after that, we choose a random salt $r \in \{0,1\}^{\bar{\ell}}$ and perfom the standard Gui signature generation process to obtain a signature $\sigma^\star = (S_k||X_k||\ldots||X_1||r)$ for the message $\mathcal{H}(d)||r$. If there appears an error (i.e. one of the equations $\mathcal{F}_V(\hat{Y}) = X$ is not uniquely solvable), we choose a new value for the salt r and try again.
- The verification algorithm `GuiVer`* returns **TRUE** if we obtain $S_0 = \mathbf{0}^{n-a}$, and **FALSE** otherwise

Algorithms `GuiKeyGen`*, `GuiSign`* and `GuiVer`* show the modified key generation, signing and verification algorithms.

Algorithm 5 `KeyGen`*: Modified Key Generation Algorithm for Gui

Input: Gui parameters (q, n, D, a, v), length $\bar{\ell}$ of the random salt
Output: Gui key pair (sk, pk)

1: $sk, pk \leftarrow$ `GuiKeyGen`(q, n, D, a, v)
2: $sk \leftarrow sk,\ \bar{\ell}$
3: $pk \leftarrow pk,\ \bar{\ell}$
4: **return** (sk, pk)

The value of $\bar{\ell}$ is specified at the end of this section.

Algorithm 6 `GuiSign`*: Modified signature generation process for Gui

Input: document d, Gui private key $(InvS, c_S, \mathcal{F}, InvT, c_T)$, length $\bar{\ell}$ of salt
Output: Gui signature $\sigma = (S_k||X_k||\dots||X_1||r) \in \mathbb{F}^{(n-a)+k\cdot(a+v)} \times \{0,1\}^{\bar{\ell}}$

1: $\ell \leftarrow \lceil k \cdot \log_2 q \cdot (n-a)/|\mathcal{H}| \rceil$
2: $r_1^{(1)}, \dots, r_a^{(1)}, r_1^{(2)}, \dots, r_a^{(k)}, v_1^{(1)}, \dots, v_v^{(1)}, v_1^{(2)}, \dots, v_v^{(k)} \leftarrow_R \mathbb{F}$
3: **repeat**
4: $S_0 \leftarrow \mathbf{0}$
5: $r \leftarrow \{0,1\}^{\bar{\ell}}$
6: $\tilde{\mathbf{h}} \leftarrow \mathcal{H}(\mathcal{H}(d)||r)||\mathcal{H}(\mathcal{H}(\mathcal{H}(d)||r))||\dots||\mathcal{H}^\ell(\mathcal{H}(d)||r)$
7: **for** $i = 1$ to k do **do**
8: $\mathbf{d}_i \leftarrow \mathbb{F}^{n-a}!(\tilde{\mathbf{h}}_{(i-1)\cdot\log_2 q\cdot(n-a)+1}, \dots, \tilde{\mathbf{h}}_{i\cdot\log_2 q\cdot(n-a)})$
9: $t, S_i, X_i \leftarrow$ `InvHFEv-`$^\star(\mathbf{d}_i \oplus S_{i-1}, r_1^{(i)}, \dots, r_a^{(i)}, v_1^{(i)}, \dots, v_v^{(i)})$
10: **if** $t ==$ **FALSE then**
11: **break** and **go to** 4
12: **end if**
13: **end for**
14: **until** $t ==$ **TRUE**
15: $\sigma^\star \leftarrow (S_k||X_k||\dots||X_1||r)$
16: **return** $\sigma^\star$

Note that, in algorithm `GuiSign`* we do not generate a signature for $\mathcal{H}(d||r)$, but for $\mathcal{H}(\mathcal{H}(d)||r)$. In case we have to run the loop in the algorithm several times, this speeds up the signature generation of our scheme significantly (at least for long messages d).
In algorithm `GuiSign`*, we make use of approximately $k \cdot \log_2 q \cdot (a+v) + e \cdot k \cdot \bar{\ell}$ random bits.

Algorithm 7 `InvHFEv-`*: Modified Inversion of the HFEv- public key

Input: Gui private key $(InvS, c_S, \mathcal{F}, InvT, c_T)$, isomorphism $\phi : \mathbb{F}^n \to \mathbb{E}$, hash value $\mathbf{w} \in \mathbb{F}^{n-a}$, random values $r_1, \dots, r_a$, vinegar values $v_1, \dots, v_v$
Output: boolean value t, vector $\mathbf{z} \in \mathbb{F}^{n+v}$

1: $\ell \to \lceil k \cdot \log_2 \cdot (n-a)/|\mathcal{H}| \rceil$
2: $\mathbf{x} \leftarrow InvS \cdot ((\mathbf{w}||r_1||\dots||r_a) - c_S)$
3: $X \leftarrow \phi(\mathbf{x})$
4: $\mathcal{F}_V \leftarrow \mathcal{F}(v_1, \dots, v_v)$
5: $Y \leftarrow \gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y})$
6: **if** $\deg(Y) == 1$ **then**
7: $\mathbf{y} \leftarrow \phi^{-1}(\mathtt{root}(Y))$
8: $\mathbf{z} \leftarrow InvT((\mathbf{y}||v_1||\dots||v_v) - c_T)$
9: **return TRUE**, $\mathbf{z}$
10: **else**
11: **return FALSE**, $\mathbf{0}^{n+v}$
12: **end if**

In line 6 of algorithm `InvHFEv-`* we check if the computed gcd is a linear

polynomial in $Y \in \mathbb{E}[\hat{Y}]$. If so, the function $\texttt{root}(Y)$ returns the unique root of Y and the algorithm returns **TRUE** as well as the unique solution of $\mathcal{P}(\mathbf{z}) = \mathbf{w}$. Otherwise, the algorithm returns **FALSE** (and the zero vector $\mathbf{0} \in \mathbb{F}^{n+v}$). Though this check makes it necessary to run algorithm $\texttt{InvHFEv}-^{\star}$ more often, it is still more efficient than computing the root of a polynomial Y of higher degree.

Algorithm 8 $\texttt{GuiVer}^{\star}$: Modified signature verification process for Gui

Input: Gui public key $\mathcal{P}$, document d, signature $\sigma = (S_k||X_k|| \dots ||X_1||r) \in \mathbb{F}^{(n-a)+k\cdot(a+v)} \times \{0,1\}^{\bar{\ell}}$

Output: boolean value **TRUE** or **FALSE**

$\ell \leftarrow \lceil k \cdot \log_2 q \cdot (n-a)/|\mathcal{H}| \rceil$
$\tilde{\mathbf{h}} \leftarrow \mathcal{H}(\mathcal{H}(d)||r)||\mathcal{H}(\mathcal{H}(\mathcal{H}(d)||r))|| \dots ||\mathcal{H}^{\ell}(\mathcal{H}(d)||r)$
for $i = 1$ to k do **do**
    $\mathbf{d}_i \leftarrow \mathbb{F}^{n-a}!(\tilde{\mathbf{h}}_{(i-1)\cdot\log_2 q\cdot(n-a)+1}, \dots, \tilde{\mathbf{h}}_{i\cdot\log_2 q\cdot(n-a)})$
end for
for $i = k-1$ to 0 **do**
    $S_i \leftarrow \mathcal{P}(S_{i+1}||X_{i+1}) \oplus \mathbf{d}_{i+1}$
end for
if $S_0 = \mathbf{0}$ **then**
    return TRUE
else
    return FALSE
end if

Similar to [16] we find that every attacker, who can break the EUF-CMA security of the modified scheme, can also break the standard Gui signature scheme.

In order to get a secure scheme, we must ensure that every signature is generated using a different random seed. Under the assumption of maximal 2^{64} signatures being generated with the system [11], a random salt of length 128 bit seems reasonable.

1.7 Note on the generation of random field elements

During the key and signature generation of Gui, we make use of a large number of random field elements. These are obtained by calling a cryptographic random number generator such as that from the OpenSSL library. In our implementation we use the `AES CTR_DRBG` function. In debug mode, our software can either generate random bits and store them in a file, or read in the required random bits from a file (for Known Answer Tests).

1.8 Parameter Choice

We choose GF(2) as the underlying finite field and choose the repetition factor k to be 2. We propose the following three parameter sets for Gui

Gui-184 Gui(GF(2),184,33,16,16,2) with 168 equations in 200 variables

Gui-312 Gui(GF(2),312,129,24,20,2) with 288 equations in 332 variables

Gui-448 Gui(GF(2),448,513,32,28,2) with 416 equations in 476 variables

The reason for restricting our scheme to the field GF(2) is to find a good balance between security and efficiency. During the signature generation process, we have to invert a univariate polynomial of degree D over the extension field $\mathbb{E}$. The complexity of this step can be estimated as

$$\text{Complexity}_{\texttt{SignGen}} = O(\log_q(|\mathbb{E}|) \cdot D^3) = O(n \cdot D^3).$$

For efficiency reasons, we therefore aim at decreasing D as far as possible.
On the other hand we want, for security reasons, the HFE polynomial (1) to contain not too few quadratic terms. Since this number directly depends on $r = \lfloor \log_q(D-1) \rfloor + 1$, we choose q as small as possible, i.e $q = 2$.
The idea behind the choice of D, a and v is that we want HFE, the minus method and the vinegar modification to play a more or less equally important role in enhancing the security of our scheme. Therefore, we choose D, a and v to increase more or less proportionally with increasing security. By doing so, Gui can be seen as balanced version of HFEv-, where HFE, the minus method and the vinegar modification are equally important. Due to the vast improvements in implementation efficiency, our solution comes without much sacrifice of efficiency, but reduces the risk of possible future attacks against Gui significantly.
For ease of implementation, we furthermore choose the parameters n, a and v of Gui in such a way that the lengths of the hash values $\mathbf{d}_i$ $(i = 1, \ldots, k)$ and the resulting Gui signatures are multiples of 8 bit. Moreover, for efficiency reasons (special processor instructions), the size n of the extension fields is chosen to be close to multiples of 64.
The resulting key and signature sizes can be found in Section 4.1.

Additionally to the three parameter proposals Gui-184, Gui-312 and Gui-448, we give here two more parameter sets to illustrate certain aspects of the parameter choice.

Gui-160 Gui(GF(2),160,33,16,16,2) The value $n = 160$ is not large enough to prevent quantum brute force attacks (see Section attacks 6.2) against the scheme. While the complexity of a classical brute force attack of 2^{147} gates is acceptable for NIST security category I, the complexity of a quantum brute force attackagainst the scheme is only 2^{94} (logical) quantum gates.

Gui-192 Gui(GF(2), 192,9,8,8,2) While the number of equations in the scheme is large enough to prevent brute force attacks, the scheme is, due to the small values of D, a and v, vulnerable by direct attacks (see Section 6.3). The degree of regularity of a direct attack against this scheme can be estimated as $(4 + 8 + 8 + 7)/3 = 9$, leading to a complexity of the direct attack of about 2^{116} gates.

1.8.1 Note on the used hash functions

We use SHA-2 as the hash function underlying our Gui instances. The SHA-2 hash function family comprises the four hash functions SHA224, SHA256, SHA384 and SHA512 with output lengths of 224, 256, 384 and 512 bits respectively. We use

- SHA256 for Gui-184
- SHA-384 for Gui-312 and
- SHA-512 for Gui-448.

An analysis of the security of the proposed Gui instances against collision attacks can be found in Section 5.3.

2 Key Storage

2.1 Representation of Finite Field Elements

2.1.1 Elements of $\mathbb{F}_2$

The field of two elements, denoted as $\mathbb{F}_2$, is the set $\{0, 1\}$. The multiplication of elements in $\mathbb{F}_2$ is a logic `AND` and addition is a logic `XOR`. Each element of $\mathbb{F}_2$ is stored in one bit.

2.1.2 Vectors over $\mathbb{F}_2$

A vector of l elements in $\mathbb{F}_2$ is represented as a bit sequence of length l. Vectors are the basic building blocks of our implementations. All components in the key file are in the form of vectors over $\mathbb{F}_2$ of different length.
Suppose $\mathbf{v} = (v_0, \ldots, v_{l-1}) \in \mathbb{F}_2^l$ and $v_i \in \mathbb{F}_2$.

- v_0 corresponds to the least significant bit of the l bit sequence.
- If l is a multiple of 8, $\mathbf{v}$ is stored in $l/8$ bytes.
- If l is not a multiple of 8, $\mathbf{v}$ is stored in $\lfloor (l + 7)/8 \rfloor$ bytes and the bits with index $> l$ are padded with 0.

2.2 Public Key

The public key $\mathcal{P}$ of Gui is a set of m multivariate quadratic polynomials in n variables (we write $\mathcal{P} := \mathcal{MQ}(m, n)$). The monomials are ordered according to the "graded-reverse-lexicographic" order.
Therefore, the system $\mathcal{P}$ looks as follows.

$$
\begin{aligned}
y_1 &= q_{2,1,1} \cdot x_2x_1 + q_{3,1,1} \cdot x_3x_1 + q_{3,2,1} \cdot x_3x_2 + q_{4,1,1} \cdot x_4x_1 + q_{4,2,1} \cdot x_4x_2 \\
&\quad + q_{4,3,1} \cdot x_4x_3 + \cdots + q_{n,n-1,1} \cdot x_nx_{n-1} + l_{1,1}x_1 + l_{2,1}x_2 + \cdots + l_{n,1}x_n + c_1 \\
y_2 &= q_{2,1,2} \cdot x_2x_1 + q_{3,1,2} \cdot x_3x_1 + q_{3,2,2} \cdot x_3x_2 + q_{4,1,2} \cdot x_4x_1 + q_{4,2,2} \cdot x_4x_2 \\
&\quad + q_{4,3,2} \cdot x_4x_3 + \cdots + q_{n,n-1,2} \cdot x_nx_{n-1} + l_{1,2}x_1 + l_{2,2}x_2 + \cdots + l_{n,2}x_n + c_2 \\
&\quad \vdots \\
y_m &= q_{2,1,m} \cdot x_2x_1 + q_{3,1,m} \cdot x_3x_1 + q_{3,2,m} \cdot x_3x_2 + q_{4,1,m} \cdot x_4x_1 + q_{4,2,m} \cdot x_4x_2 \\
&\quad + q_{4,3,m} \cdot x_4x_3 + \cdots + q_{n,n-1,m} \cdot x_nx_{n-1} + l_{1,m}x_1 + l_{2,m}x_2 + \cdots + l_{n,m}x_n + c_m
\end{aligned}
$$

Here, $q_{i,j,k}$ is the coefficient of the quadratic monomial x_ix_j of the polynomial y_k, $l_{i,k}$ the coefficient of the linear monomial x_i in y_k and c_k the constant coefficient of y_k ($1 \leq j < i \leq n, 1 \leq k \leq m$). Note that there are no x_ix_j terms with $i = j$, since the polynomials are defined over $\mathbb{F}_2$.
We define the vectors $\mathbf{q}_{i,j} = (q_{i,j,k})_{k=1}^m \in \mathbb{F}_2^m$, $\mathbf{l}_i = (l_{i,k})_{k=1}^m \in \mathbb{F}_2^m$, and $\mathbf{c} = (c_1, \ldots, c_m) \in \mathbb{F}_2^m$. Using this notation, the public key $\mathcal{P}$ is stored as a byte sequence

$$[\mathbf{l}_1, \ldots, \mathbf{l}_n, \mathbf{q}_{2,1}, \mathbf{q}_{3,1}, \mathbf{q}_{3,2}, \ldots, \mathbf{q}_{n,n-2}, \mathbf{q}_{n,n-1}, \mathbf{c}].$$

2.3 Secret Key

The secret key comprises the three components $\mathcal{T}, \mathcal{S}$, and $\mathcal{F}$. These components are stored in the order $\mathcal{T}, \mathcal{S}$, and $\mathcal{F}$.

2.3.1 The affine maps $\mathcal{T}$ and $\mathcal{S}$

Suppose the affine map $\mathcal{T}(\mathbf{x}) : \mathbb{F}^{n+v} \to \mathbb{F}^{n+v}$ is given by

$$\mathcal{T}(\mathbf{x}) = \begin{bmatrix} t_{1,1} & t_{1,2} & \ldots & t_{1,n+v} \\ & \vdots & \ddots & \\ t_{n+v,1} & t_{n+v,2} & \ldots & t_{n+v,n+v} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n+v} \end{bmatrix} + \begin{bmatrix} c_1 \\ \vdots \\ c_{n+v} \end{bmatrix}.$$

We store the matrix in column-major form. Define, for $i := 1, \ldots, n+v$, the column vector $\mathbf{t}_i = (t_{1,i}, \ldots, t_{n+v,i}) \in \mathbb{F}_2^{n+v}$, as well as the vector of the constant terms $\mathbf{c} = (c_1, \ldots, c_{n+v}) \in \mathbb{F}_2^{n+v}$. With this, the affine map $\mathcal{T}$ is stored as the sequence

$$[\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_{n'}, \mathbf{c}].$$

The affine map $\mathcal{S}$ is stored in the same manner.

2.3.2 The central map $\mathcal{F}$

Recall that the central map of Gui is a map from $\mathbb{E} \times \mathbb{F}^v$ to $\mathbb{E}$ of the form

$$\mathcal{F}(X) = \sum_{0 \leq i \leq j}^{q^i + q^j \leq D} \alpha_{i,j} X^{q^i + q^j} + \sum_{0 \leq i}^{q^i \leq D} \beta_i(v_1, \ldots, v_v) X^{q^i} + \gamma(v_1, \ldots, v_v).$$

An element in $\mathbb{E} := \mathbb{F}_{2^n}$ is stored as an n dimensional vector in $\mathbb{F}_2^n$. The field isomorphism $\phi : \mathbb{F}_2^n \to \mathbb{E}$ is described in the next section.
The coefficients of $\mathcal{F}(X)$ are stored in the order $\alpha_{i,j}$, β_i, and γ.

$\alpha_{i,j}$ **:** The coefficients $(\alpha_{i,j})_{0\le i\le j}^{2^i+2^j\le D} \in \mathbb{E}$ are divided into 2 groups $A : (\alpha_{i,j})_{0\le i=j}^{2^i+2^j\le D}$ and $B : (\alpha_{i,j})_{0\le i<j}^{2^i+2^j\le D}$. These elements are stored in the order A, B. Within A and B, the coefficients $\alpha_{i,j}$ are stored in ascending order (with respect to the exponent $2^i + 2^j$).
The reason for partitioning the coefficients $\alpha_{i,j}$ into the two groupt A and B is that the elements of A a later combined with the maps β_i to compute the coefficients of the linear terms of the parametrized map $\mathcal{F}_V$.

$\beta_i(v_1, \ldots, v_v)$ **:** Each β_i contains v $\mathbb{E}$-elements corresponding to the coefficients of the linear monomials $v_1, \ldots, v_v$. We denote the elements of the vector β_i by $(\beta_{i,1}, \ldots, \beta_{i,v})$. The β part of $\mathcal{F}(X)$ is stored in the order

$$[\beta_{0,1}, \ldots, \beta_{0,v}, \beta_{1,1}, \ldots, \beta_{\lfloor \log_2 D \rfloor, v}].$$

$\gamma(v_1, \ldots, v_v)$ **:** The map $\gamma : \mathbb{F}^v \to \mathbb{E}$ is a multivariate quadratic polynomial in the n variables $v_1, \ldots, v_n$. Using the field equations $x_i^2 = x_i$ $(i = q, \ldots, v)$, we find that we can interpret the linear terms of γ as quadratic ones. Therefore, we can consider γ as a homogeneous quadratic polynomial. We store the coefficients of γ according to the graded-reverse-lexicographic order. Therefore we get $\gamma(v_1, \ldots, v_v) = \gamma_{1,1}v_1^2 + \gamma_{2,1} \cdot v_2 \cdot v_1 + \gamma_{2,2} \cdot v_2 \cdot v_2 + \cdots + \gamma_{v,v} \cdot v_v \cdot v_v$. The γ part of $\mathcal{F}(X)$ stores the $v(v+1)/2$ coefficients of $\gamma(v_1, \ldots, v_v)$ as a sequence

$$[\gamma_{1,1}, \gamma_{2,1}, \gamma_{2,2}, \gamma_{3,1}, \ldots, \gamma_{v,v})].$$

Note that each of the $\gamma_{i,j}$ is a vector of n bits.

Example (central map of degree 17): The quadratic monomials in a degree-17 central map are $\{X^2 = X^{2^0+2^0}, X^3 = X^{2^0+2^1}, X^4 = X^{2^1+2^1}, X^5 = X^{2^2+2^0}, X^6 = X^{2^2+2^1}, X^8 = X^{2^2+2^2}, X^9 = X^{2^3+2^0}, X^{10} = X^{2^3+2^1}, X^{12} = X^{2^3+2^2}, X^{16} = X^{2^3+2^3}, X^{17} = X^{2^4+2^0}\}$. These are divided into the sets $A = \{X^2, X^4, X^8, X^{16}\}$ and $B = \{X^3, X^5, X^6, X^9, X^{10}, X^{12}, X^{17}\}$. We denote the coefficient of $X^{q^i+q^j}$ by $\alpha_{i,j}$.
The set of linear monomials is given by $\{X, X^2, X^4, X^8, X^{16}\}$. Note that the monomials X^2, X^4, X^8 and X^{16} are also contained in the set A of quadratic monomials. That is why we do not need constant terms in the linear functions β_i. We denote the linear function multiplied to X^{q^i} by β_i and the coefficient of v_j in β_i by $\beta_{i,j}$.
Finally, we have to append the coefficients of the map γ.

The coefficients of $\mathcal{F}(X)$ are therefore stored in the order

$$[\ \underbrace{\underbrace{\alpha_{0,0},\ \alpha_{1,1},\ \alpha_{2,2},\ \alpha_{3,3}}_{setA}\ ,\ \underbrace{\alpha_{0,1},\ \alpha_{0,2},\ \alpha_{1,2},\ \alpha_{0,3},\ \alpha_{1,3},\ \alpha_{2,3},\ \alpha_{0,4}}_{setB}}_{\text{coefficients } \alpha_{i,j} \text{ of quadratic terms}}\ ,$$
$$\underbrace{\beta_{0,1},\ \ldots,\ \beta_{0,v},\beta_{1,1}\ ,\ldots,\ \beta_{4,v}}_{\text{coefficients of the linear maps } \beta_i}\ ,\quad \underbrace{\gamma_{1,1},\ \gamma_{2,1},\ \gamma_{2,2},\ \ldots,\ \gamma_{v,v}}_{\text{coefficients of the quadratic map } \gamma}\ ].$$

2.4 The field isomorphism ϕ

We use the field of 256 elements, $\mathbb{F}_{256}$, each element occupying 1 byte, as a basic operating unit. The larger extension fields are further extended from $\mathbb{F}_{256}$.

We first describe here the field isomorphism $\phi_{256} : \mathbb{F}_2^8 \to \mathbb{F}_{256}$ and then describe the isomorphism ϕ mapping vectors over $\mathbb{F}_2$ to $\mathbb{E}$.

2.4.1 The isomorphism $\phi_{256} : \mathbb{F}_2^8 \to \mathbb{F}_{256}$

We use the *tower field* representation of $\mathbb{F}_{256}$ which considers an element in $\mathbb{F}_{256}$ as a linear polynomial over $\mathbb{F}_{16}$. Elements of GF(16) are viewed as linear polynomials over GF(4) and so on. The sequence of tower fields from which we build $\mathbb{F}_{256}$ looks like

$$\begin{aligned} \mathbb{F}_4 &:= \mathbb{F}_2[e_1]/(e_1^2 + e_1 + 1), \\ \mathbb{F}_{16} &:= \mathbb{F}_4[e_2]/(e_2^2 + e_2 + e_1), \\ \mathbb{F}_{256} &:= \mathbb{F}_{16}[e_3]/(e_3^2 + e_3 + e_2 e_1)\ . \end{aligned}$$

We use $\mathbf{b}_{256} = (1, e_1, e_2, e_1e_2, e_3, e_1e_3, e_2e_3, e_1e_2e_3) \in \mathbb{F}_{256}^8$ as a basis for $\mathbb{F}_{256}$. Such, the $\mathbb{F}_{256}$-element encoded as `0x2` is e_1, `0x4` is e_2, `0x8` is e_1e_2, `0x16` is e_3 etc., and numbers up to `0xff` are their combinations, for example `0x1d` $= e_3 + e_1e_2 + e_2 + 1$.

The field isomorphism ϕ_{256} maps a vector $\mathbf{v} = (v_1, \ldots, v_8) \in \mathbb{F}_2^8$ to the $\mathbb{F}_{256}$ element $v_1 + v_2 \cdot e_1 + \cdots + v_8 \cdot e_1e_2e_3$.

2.4.2 The isomorphism $\phi : \mathbb{F}_2^n \to \mathbb{E}$

A field element in $\mathbb{F}_{2^{184}}$ is represented as a degree 22 polynomial in $\mathbb{F}_{256}[X]$. We define

$$\mathbb{F}_{2^{184}} := \mathbb{F}_{256}[X]/X^{23} + X^3 + X + \texttt{0x2}\ .$$

The isomorphism $\phi : \mathbb{F}_2^{184} \to \mathbb{F}_{2^{184}}$ is defined as

$$\begin{aligned} \phi : \mathbf{a} =& (a_1, \ldots, a_{184}) \in \mathbb{F}_{2^{184}} \mapsto \\ & (b_0, \ldots, b_{22}) = (\phi_{256}(a_1, \ldots, a_8), \ldots, \phi_{256}(a_{177}, \ldots, a_{184})) \in \mathbb{F}_{256}^{22} \mapsto \\ & b_0 + b_1 \cdot X + b_2 \cdot X^2 \ldots + b_{22} \cdot X^{22} \in \mathbb{F}_{256}[X]\ . \end{aligned}$$

The isomorphisms for the other extension fields used for Gui differ from the above construction only by the use of different irreducible polynomials. We define

$$\begin{aligned} \mathbb{F}_{2^{184}} &:= \mathbb{F}_{256}[X]/X^{23} + X^3 + X + \texttt{0x2}, \\ \mathbb{F}_{2^{312}} &:= \mathbb{F}_{256}[X]/X^{39} + X^2 + X + \texttt{0x2}, \\ \mathbb{F}_{2^{448}} &:= \mathbb{F}_{256}[X]/X^{56} + \texttt{0x2} \cdot X^3 + X + \texttt{0x10}. \end{aligned} \tag{2}$$

3 Implementation Details

In this section we present the details of our implementation of the Gui signature scheme.

3.1 Arithmetic Over Finite Fields

The first step in our implementation of the Gui signature scheme is to provide efficient arithmetic operations over the large binary fields in use. To do this, we use a set of new processor instructions for carry-less multiplication: PCLMULQDQ [18].
The instruction set PCLMULQDQ allows the efficient multiplication of two degree 64 polynomials over GF(2), resulting in a polynomial of degree 128. The PCLMULQDQ instructions are available on most new processors of Intel and AMD. Performance data of PCLMULQDQ can be found in Table 1.

Processor type	Latency (cycles)	Throughput (cycles/multiplication)
Intel Sandy Bridge	14	8
Ivy Bridge	14	8
Haswell	7	2
Skylake	7	1
AMD Bulldozer	12	7
Piledriver	12	7
Steamroller	11	7

Table 1: Performance of PCLMULQDQ on different platforms [8]

In the case of Gui, we represent an element of the field $\mathbb{E}$ as a polynomial over GF(2) of degree 184, 312 or 448. These polynomials can be divided into 3–7 polynomials of degree 64, which then can be used as input values for PCMULQDQ. A multiplication over the large field $\mathbb{E}$ is divided into two phases, namely a multiplication and a reduction phase.

In the *multiplication phase*, the multiplication of two 184-bit polynomials can be performed by 6 calls of PCLMULQDQ. With the help of the Karatsuba algorithm, we can avoid 3 calls of PCLMULQDQ and therefore its long latency

(see Table 1).
To square an element of $\mathbb{E}$, we need only 3 calls of PCLMULQDQ since we are operating over a field of characteristic 2.

The *reduction phase* of the field multiplication heavily depends on the representation of the extension fields. The baseline for ths step is 9 calls of PCLMULQDQ, since, after the multiplication phase, the degree of the polynomial will be greater than $5 \cdot 64$.
The irreducible polynomials used to define the extension fields (see equation (2)) are chosen to contain only few terms of low degree. With few terms in the irreducible polynomials, we can replace the use of PCLMULQDQ by a few logic shifts and XOR instructions.

Since, regardless of the input, the same operations are performed, our implementation provides time-constant multiplication for preventing side channel leakage. The same strategy is also applied to the calculation of multiplicative inverses. For example, for the sake of time-constant arithmetics, the inverse of an element $x \in \mathrm{GF}(2^{184})$ is calculated by raising x to $x^{2^{184}-2}$ instead of the faster extended Euclidean algorithm.

3.2 Inverting the HFEv- Core

In this section we describe how we can perform the inversion of the central HFEv- equation $\mathcal{F}_V(\hat{Y}) = X$ efficiently. To invert the central HFEv- equation, we have to run the Berlekamp or Cantor-Zassenhaus algorithms to find the roots of the polynomial $\mathcal{F}_V(\hat{Y}) - X$. In order to speed up the computations, we restrict to polynomials $\mathcal{F}_V(\hat{Y}) - X$ having a unique solution (see Algorithms `GuiSign` and `GuiSign`*). Therefore, we only have to perform the first step of the algorithm, i.e. the computation of

$$\gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y}). \tag{3}$$

In case that this gcd is not a linear polynomial, we choose another value for the random seed r and try again. We have

$$\begin{aligned} & \gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y}) \\ = \; & \gcd(\mathcal{F}_V(\hat{Y}) - X, \prod_{i \in \mathbb{F}_{2^n}} (\hat{Y} - i)) = \prod_{i \in \mathbb{F}_{2^n} : \mathcal{F}_V(i) = X} (\hat{Y} - i). \end{aligned}$$

Therefore the most costly step in generating a signature consists in computing $\gcd(\mathcal{F}_V(\hat{Y}) - X, \hat{Y}^{2^n} - \hat{Y})$. The number of roots of $\mathcal{F}_V(\hat{Y}) - X$ (as well as the only solution when that happens) can obviously be read off from the result.

Probability of a Unique Root. Every time we choose the values of Minus equations and Vinegar variables (respectively, when we pick a salt), we basically pick a random central equation $\mathcal{F}_V(\hat{Y}) - X = 0$. The probability of this

equation having a unique solution is about $1/e$. Therefore, in order to invert the HFEv- central equation k times successfully, we have to perform the gcd computation about e^k times.
The repeated computation of the gcd is probably the most detectable side channel leakage of our scheme. However, there are no known side channel attacks on big field schemes or HFEv- which use the information that one particular equation in the big field has no, respectively two or more solutions.

How Do We Optimize the Computation of the GCD? The most costly step in the computation of the gcd is the division of the extreme high power polynomial $\hat{Y}^{2^n} - \hat{Y} \bmod \mathcal{F}_V(\hat{Y})$. A naive long division is unacceptable for this purpose due to its slow reduction phase. Instead of this, we choose to recursively raise the lower degree polynomial $\hat{Y}^{2^m}$ to the power of 2.

$$\begin{aligned} & \left(\hat{Y}^{2^m} \bmod \mathcal{F}_V(\hat{Y})\right)^2 \\ = & \left(\sum_{i \le D} b_i \hat{Y}^i\right)^2 \bmod \mathcal{F}_V(\hat{Y}) = \left(\sum_{i \le D} b_i^2 \hat{Y}^{2i}\right) \bmod \mathcal{F}_V(\hat{Y}) \end{aligned}$$

By multiplying $\hat{Y}$ to the naive relation

$$\hat{Y}^D = \sum_{0 \le i \le j, 2^i + 2^j < D} \alpha_{ij} \hat{Y}^{2^i + 2^j},$$

we can prepare a table for $\hat{Y}^{2^i} \bmod \mathcal{F}_V(\hat{Y})$ first. The remaining computation of the raising process consists in squaring all the coefficients b_i in $\hat{Y}^{2^m} \bmod \mathcal{F}_V(\hat{Y})$ and multiply them to the $\hat{Y}^{2i}$'s in the table.
Although the starting relation

$$\mathcal{F}_V(\hat{Y}) = \hat{Y}^D + \sum_{0 \le i \le j,\ 2^i + 2^j < D} \alpha_{ij} \hat{Y}^{2^i + 2^j}$$

is a sparse polynomial, the polynomials become dense quickly in the course of the raising process. However, the number of terms in the polynomials is restricted by D because of mod $\mathcal{F}_V(\hat{Y})$. We expect the number of terms to be in average D during the computation.

We implemented the simplified Cantor-Zassenhaus algorithm in such a way that it takes, independently of the input, always the same number of iterations in the main gcd loop and the same number of operations in the big field. Therefore the algorithm runs, independently from the input, at constant time.

The number of field multiplications needed to compute the $\hat{Y}^{2^i}$ table is $\mathcal{O}(2 \cdot D^2)$. To raise $\hat{Y}^{2^m}$ to $\hat{Y}^{2^n}$, we need $\mathcal{O}((n-m) \cdot D)$ squarings and $\mathcal{O}((n-m) \cdot D^2)$

multiplications.
It is possible to reduce the number of computations needed for computing $\hat{Y}^{2^m}$ further by using a higher degree $\hat{Y}^i$ table. For example, if one raises $\hat{Y}^{2^m}$ to $\hat{Y}^{2^{4m}}$ in one step, one only needs $\mathcal{O}((n-m)\cdot D)$ squarings and $\mathcal{O}((n-m)\cdot 2\cdot D^2)$ multiplications. However, the computational effort of preparing the $\hat{Y}^{2^i}$ table increases.

4 Performance Analysis

4.1 Key and Signature Sizes

The following table shows the key and signature sizes of our proposed Gui-instances.

	parameters (n, D, a, v, k)	public key size (kB)	private key size (kB)	signature size (bit)[1]
Gui-184	$(184, 33, 16, 16, 2)$	416.3	19.1	360
Gui-312	$(312, 129, 24, 20, 2)$	1,955.1	59.3	504
Gui-448	$(448, 513, 32, 28, 2)$	5,789.2	155.9	664

[1] including 128 bit salt

Table 2: Key and Signature sizes of the proposed Gui instances

4.2 Performance on the NIST Reference Platform

Processor: Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz (Skylake)
Clock Speed: 3.30GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
No use of special processor instructions

scheme	parameters (n, D, a, v, k)		key generation	signature generation	signature verification
Gui-184	$(184, 33, 16, 16, 2)$	cycles	2,408M	1,910M	152k
		time (ms)	213	10.4	0.051
		memory	3.5MB	3.4MB	3.3MB
Gui-312	$(312, 129, 24, 20, 2)$	cycles	43,817M	25,436M	846k
		time (ms)	13,227	7,707	0.256
		memory	5.4MB	3.8MB	5.0MB
Gui-448	$(448, 513, 32, 28, 2)$	cycles	239,502M	872,949M	1,787k
		time (ms)	71,485	264,530	0.542
		memory	17.7MB	10.7MB	8.7MB

Table 3: Performance of Gui on the NIST Reference Platform (Linux/Skylake)

4.3 Performance on Other Platforms

Processor: Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz (Skylake)
Clock Speed: 3.30GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2,133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
Use of PCLMULQDQ instructions

scheme	parameters (n, D, a, v, k)		key generation	signature generation	signature verification
Gui-184	$(184, 33, 16, 16, 2)$	cycles	704M	34M	169k
		time (ms)	213	10.4	0.051
		memory	3.5MB	3.4MB	3.3MB
Gui-312	$(312, 129, 24, 20, 2)$	cycles	4,790M	1,757M	595k
		time (ms)	1,452	532	0.181
		memory	5.4MB	3.6MB	5.0MB
Gui-448	$(448, 513, 32, 28, 2)$	cycles	32,247M	86,086M	3,385k
		time (ms)	9,772	26,086	1.025
		memory	9.2MB	10.7MB	8.7MB

Table 4: Performance of Gui on Linux/Skylake (PCLMULQDQ)

Processor: Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz (Broadwell)
Clock Speed: 2.1GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
Use of PCLMULQDQ instructions

scheme	parameters (n, D, a, v, k)		key generation	signature generation	signature verification
Gui-184	$(184, 33, 16, 16, 2)$	cycles	721M	34M	141k
		time (ms)	343	16.3	0.067
		memory	3.7MB	3.2MB	3.2MB
Gui-312	$(312, 129, 24, 20, 2)$	cycles	4,955M	1,815M	371k
		time (ms)	2,360	864	0.018
		memory	5.3MB	3.5MB	4.8MB
Gui-448	$(448, 513, 32, 28, 2)$	cycles	30,025M	88,528M	3,307k
		time (ms)	71,485	42,156	1.575
		memory	17.5MB	10.7MB	8.6MB

Table 5: Performance of Gui on Linux/Broadwell (PCLMULQDQ)

4.4 Note about the measurements

Turboboost is disabled on our platforms. The main compilation flags are `gcc -O3 -std=c99 -Wall -Wextra (-mclmul)`. The used memory is measured during an actual run using `/usr/bin/time -f "%M"` (average of 10 runs). For key generation, signatures and verification we take the average of 10 runs.

4.5 Trends as the number n of variables increases

Key Generation: Like other large field multivariate schemes, HFEv-/Gui uses interpolation to generate the keys, which takes about n^2 times the time needed to evaluate the central map once. This is often said to be an $O(n^6)$ operation. In practice, the HFE polynomial has $O((\log_q D)^2)$ terms, each of which takes $O(\log_q D)$ square-and multiplies in the big field, so the complexity is closer to $O(n^4 \log_q D^3)$ in our range.

Verification/Public Map: This is a straightforward MQ evaluation and is $O(n^3)$,

Signature/Secret Map: The simplified Cantor-Zassenhaus algorithm takes $O(nD)$ big field multiplications, but is in our implementation in practice closer to $O(nD^2)$. Similarly, the big field multiplication is $O(n \log n)$ but is in practice is closer to $O(n^2)$, so in our range the time used increase more like $O(n^3 D^2)$.

5 Expected Security Strength

In the NIST call for proposals [11], the security of a scheme is measured in the number of classical or quantum gates that an attack against the scheme requires. Hereby, the number of quantum gates contains a factor `MAXDEPTH`, which, according to [11], can take the values 2^{40}, 2^{64}, or 2^{96}. According to this, the proposed 6 NIST security categories are defined as shown in Table 6.

category	$\log_2$ classical gates	$\log_2$ (logical) quantum gates
I	143	130 / **106** / 74
II	146	
III	207	193 / **169** / 137
IV	210	
V	272	258 / **234** / 202
VI	274	

Table 6: NIST security categories

In this proposal, we use a value of `MAXDEPTH` of 2^{64}. This seems plausible to us since, in a quantum setting, the best known attack against our scheme is the quantum brute force attack, for which we assume `MAXDEPTH` to be more or less in the middle of the given extremas. This choice results in the values printed in **bold** in Table 6.

5.1 General Remarks

The Gui signature scheme as described in Section 1.6 of this proposal fulfills the requirements of the EUF-CMA security model (existential unforeability under chosen message attacks). The parameters of the scheme (in particular the length of the random salt) are chosen in a way that up to 2^{64} messages can be signed with one key pair. The scheme can generate signatures for messages of arbitrary length (as long as the underlying hash function can process them).

5.2 Practical Security

In this section we study the practical security of the proposed instances of the Gui signature scheme. Since the security of Gui can not be reduced directly to a hard mathematical problem such as the $\mathcal{MQ}$-Problem, we choose the parameters of Gui in a way that the complexities of all known attacks against Gui are beyond the required levels of security. We note that, despite of the altered signature generation process, all of the known attacks against Gui are attacks against the underlying HFEv- scheme. These include (see Section 6):

- brute force attacks (see Section 6.2)
- direct algebraic attacks (see Section 6.3)

- rank attacks of the Kipnis-Shamir type (see Section 6.4) and
- the distinguishing attack (see Section 6.5).

While the first two of these attacks are signature forgery attacks, which have to be performed for every message separately, the Kipnis-Shamir attack and the distinguishing attack are key recovery attacks. After having recovered the HFEv- private key using one of these attacks, the attacker can generate signatures for arbitrary messages in the same way as a legitimate user. Further note that, due to the special signature generation process of Gui, the attacker has, in order to forge a Gui signature, to run the HFEv- signature forgery attacks k times.

The following table shows the complexity of these known attacks against our Gui instances. In each cell, the first number shows the number of classical gates required to perform the attack, while the second number shows the required number of quantum gates. In each row, the number printed in **bold** shows the complexity of the best attack against the scheme.

	parameters	$\log_2$(#gates)			
	(n, D, a, v, k)	direct[1]	brute force	MinRank[1]	distinguisher
Gui-184	$(184, 33, 16, 16, 2)$	**156.8**	171.9	323.6	246.4
		156.8	**108.2**	323.6	191.4
Gui-312	$(312, 129, 24, 20, 2)$	**221.6**	292.0	480.5	370.7
		221.6	**170.5**	480.5	280.3
Gui-448	$(448, 513, 32, 28, 2)$	**293.3**	420.1	665.2	509.9
		293.3	**236.1**	665.2	381.9

[1] As our analysis (see Section 6) shows, there is no difference between the number of classical and quantum gates for direct and rank attacks.

Table 7: Estimated attack complexities against the proposed Gui instances

Based on the above security evaluation (and the results of Section 5.3 below) , we propose Gui-184 for the security categories I and II (see Table 4). Gui-312 meets the requirements of security categories III and IV, whereas Gui-448 is suitable for the security Categories V and VI.

5.3 Security against collision attacks

Additionally to the attacks against the Gui scheme itself, we also have to study the security of our schemes under collision attacks against the underlying hash functions. As already mentioned in Section 1.8.1, we use

- SHA256 for Gui-184
- SHA384 for Gui-312 and

- SHA512 for Gui-448

as hash functions underlying the proposed Gui instances. For all of the proposed schemes, we use $k = 2$ as the repetition factor of our scheme. Therefore, we obtain for the parameter ℓ used in Algorithms 2, 4, 6 and 8

$$\ell = \lceil k \cdot (n-a)/|\mathcal{H}| \rceil = 2,$$

which means that the vector $\tilde{\mathbf{h}}$ used in these algorithms is given by

$$\tilde{\mathbf{h}} = \mathcal{H}(d)||\mathcal{H}(\mathcal{H}(d)) \quad (\text{rsp. } \mathcal{H}(\mathcal{H}(d)||r)||\mathcal{H}(\mathcal{H}(\mathcal{H}(d)||r)).$$

The first hash value used in the signature generation process of Gui, $\mathbf{d}_1$ consists of the first $(n-a)$ bits of $\tilde{\mathbf{h}}$, whereas $\mathbf{d}_2 = \tilde{\mathbf{h}}_{n-a+1}, \ldots, \tilde{\mathbf{h}}_{2\cdot(n-a)}$.
In any case, all the bits of $\mathcal{H}(d)$ are contained either in $\mathbf{d}_1$ or $\mathbf{d}_2$, which means that a collision attack against our scheme is at least as hard as a collision attack against the hash function $\mathcal{H}$. This justifies the classification of our Gui instances into the corresponding security categories.

5.4 Side Channel Resistance

In our implementation of the Gui signature scheme (see Section 3), all key dependent operations (in particular Gaussian Elimination, Exponentiation) are performed in a time-constant manner. Therefore, our implementation is immune against timing attacks.

6 Analysis of Known Attacks

Despite of the modified signature generation process, the security of Gui is based on that of the HFEv- scheme and all known attacks against Gui are attacks against HFEv-. These include

- collision attacks against the hash function (Section 6.1)
- brute force attacks (Section 6.2)
- direct attacks (Section 6.3)
- rank attacks of the Kipnis-Shamir type (Section 6.4)
- the distinguishing attack of Perlner et al. (Section 6.5).

Brute force and direct attacks are signature forgery attacks, which means that they have to be performed for each message separately. Furthermore, due to the special signature generation process of Gui, these attacks have to be performed k times in order to forge a Gui signature. On the other hand, the rank attack of Kipnis and Shamir and the distinguishing attack are key recovery attacks. After having recovered the HFEv- private key using one of these attacks, the adversary can sign messages in the same way as a legitimate user.

6.1 Collision attacks against the hash function

Since the Gui signature scheme follows the Hash then Sign approach, we have to choose the parameters of Gui in such a way that collision attacks against the underlying hash function are infeasible. However, due to the specially designed signature generation process of Gui, this does not have a direct influence on the number of equations in the public system. Since a Gui signature depends on the k hash values $\mathbf{d}_1, \ldots, \mathbf{d}_k \in \mathbb{F}^{n-a}$, the resulting effective length of the hash value is $k \cdot (n-a) \cdot \log_2 q$ bit. Therefore, the complexity of a collision attack against our scheme is about

$$\text{Complexity}_{\text{collision}} = 2^{k \cdot (n-a)}.$$

By choosing the repetition factor k in an appropriate way, it is therefore easy to prevent collision attacks (even for a relatively small number $(n-a)$ of equations).
An analysis of the security of our proposed Gui instances against collision attacks can be found in Section 5.3.

6.2 Brute Force Attacks

Since the public system of Gui is defined over the field GF(2) with two elements, the parameters of the scheme have to be chosen in a way that prevents brute force attacks against binary $\mathcal{MQ}$-systems. In the classical world, we have to

mention here the Gray Code enumeration of [3]. In order to solve a public HFEv- system using this technique, one first fixes $v + a$ variables to get a determined system. The resulting system of $n - a$ equations in $n - a$ variables can then be evaluated for every possible input using $2^{n-a+2} \cdot \log_2(n - a)$ bit operations. In order to forge a Gui signature, we have to perform this step k times. Therefore, the complexity of this attack can be estimated as

$$\text{Complexity}_{\text{brute force; classical}} = k \cdot 2^{n-a+2} \cdot \log_2(n - a)$$

bit operations.
In the quantum world, brute force attacks can be sped up using Grover's algorithm. As shown in [17], we can find the solution of a binary $\mathcal{MQ}$-system of $n - a$ equations in $n - a$ variables using $2^{(n-a)/2} \cdot 2 \cdot (n - a)^3$ quantum bit operations. Again, in order to forge a Gui signature, an attacker has to perform this step k times. Therefore, we can estimate the complexity of a quantum brute force attack against our scheme as

$$\text{Complexity}_{\text{brute force; quantum}} = k \cdot 2^{(n-a)/2} \cdot 2 \cdot (n - a)^3$$

(quantum) bit operations.

6.3 Direct Attacks

Similar to the brute force attacks (see previous section), a direct attack considers the public equation $\mathcal{P}(\mathbf{z}) = \mathbf{w}$ as an instance of the $\mathcal{MQ}$-Problem. Since the public system of HFEv- is an underdetermined system (more variables than equations), the most efficient way to solve this equation is to fix $a + v$ variables to create a determined system before applying an algorithm like XL or a Gröbner Basis technique such as F_4 or F_5 [6]. It can be expected that the resulting determined system has exactly one solution. In some cases, one obtains even better results when guessing additional variables before solving the system (hybrid approach) [1].

Experiments [10, 7] have shown that the public systems of HFEv- can be solved significantly faster than random systems. The reason for this is that these systems have a significantly lower degree of regularity. In [5] it was shown that the degree of regularity of an HFEv- system is upper bounded by

$$\mathrm{d}_{\text{reg}} \leq \begin{cases} \frac{(q-1)\cdot(r+a+v-1)}{2} + 2 & q \text{ even and } r + a \text{ odd,} \\ \frac{(q-1)\cdot(r+a+v)}{2} + 2 & \text{otherwise.} \end{cases} \tag{4}$$

with $r = \lfloor \log_q(D - 1) \rfloor + 1$. Since the upper bound on the degree of regularity given by equation (4) does not really help to estimate the complexity of direct attacks against HFEv- schemes in practice, we follow here the analysis of [14]. In [14], Petzoldt derived from experiments the following lower bound for the degree of regularity of an HFEv- system

$$d_{\text{reg}} = \lfloor \frac{a + r + v + 7}{3} \rfloor \tag{5}$$

Furthermore, as shown in [14], the hybrid approach does not help to speed up direct attacks against HFEv- schemes. In our security analysis (see previous section), we therefore estimate the complexity of a direct attack against an HFEv-(n, D, a, v) instance as

$$\text{Complexity}_{\text{direct attack}} = 2 \cdot k \cdot 3 \cdot \binom{n-a}{d_{\text{reg}}}^2 \cdot \binom{n-a}{2}$$

bit operations, where d_{reg} is given by formula (5).
Since there is no guessing step in the attack, we can not reduce its complexity by the use of Grover's algorithm.

6.4 Rank attacks of the Kipnis Shamir type

In [9], Kipnis and Shamir proposed a rank attack against the HFE cryptosystem. The key idea of this attack is to consider the public and private maps of HFE as univariate polynomial maps over the extension field. Due to the special structure of the HFE central map, the rank of the corresponding matrix is limited by $r = \lfloor \log_2(D-1) \rfloor + 1$. It is therefore possible to recover the affine transformation $\mathcal{S}$ by solving an instance of the MinRank problem.

In [2], Bouillaget et al. improved this attack by showing that the map $\mathcal{S}$ can be found by computing a Gröbner Basis over the base field GF(2) (Minors Modelling). By doing so, they could speed up the attack of Kipnis and Shamir significantly. The complexity of the MinRank attack against HFE using the Minors Modelling approach can be estimated as

$$\text{Complexity}_{\text{MinRank; HFE}} = \binom{n+r}{r}^{\omega},$$

where $r = \lfloor \log_2(D-1) \rfloor + 1$ is the rank of the matrix corresponding to the central map and $2 < \omega \leq 3$ is the linear algebra factor. In our security analysis (see Secton 5.2), we choose the value of ω to be 2.3.
In the case of HFEv-, the rank of the matrix is given by $r + a + v$ [2]. Therefore, we can estimate the complexity of our attack by

$$\text{Complexity}_{\text{KS attack; HFEv}-} = \begin{cases} \binom{n+r+a+v}{r+a+v}^{2.3} & r+a+v \text{ even} \\ \binom{n+r+a+v-1}{r+a+v-1}^{2.3} & r+a+v \text{ odd} \end{cases}.$$

Since the quadratic systems to be solved during this attack are highly overdetermined, the attack can not be sped up with the help of Grover's algorithm.

[2] Since we work over fields of even chatacteristic, the rank of the matrix corresponding to the central map $\mathcal{F}$ is always even. Therefore, in the case of $r + a + v$ being odd, the rank of this matrix is given by $r + a + v - 1$.

6.5 The Distinguishing attack of Perlner et al.

The distinguishing attack of Perlner et al. [13] uses the fact that the behavior of a direct attack depends on the number of vinegar variables in the HFEv- system. By using this fact, it is possible to remove the vinegar variables one by one from the system. The resulting HFE- system can then be solved much easier than the original system. The most costly step in the attack is hereby to remove the first vinegar variable, i.e. the reduction of an HFEv-(n, D, a, v) to an HFEv-$(n, D, a, v-1)$ scheme. The complexity of this first step can be estimated as

$$\text{Complexity}_{\text{Distinguisher; classical}} = 2^{n-k} \cdot 3 \cdot \binom{n+v-k}{d_{\text{reg}}}^2 \cdot \binom{n+v-k}{2},$$

where

- $n + v - k$ is the maximal number of variables for which a direct attack against the projected systems HFEv-(n, D, a, v) and HFEv-$(n, D, a, v-1)$ behaves differently; $n+v-k$ is given as the maximal number n' for which the degree of regularity of a direct attack against a projected random system of $(n - a)$ quadratic equations in n' variables is below d_{reg}.
- d_{reg} is the degree of regularity of a direct attack against the unprojected HFEv-(n, D, a, v) system; according to [14], d_{reg} can be estimated by

$$d_{\text{reg}} = \lceil \frac{r + a + v + 7}{3} \rceil.$$

In the presence of quantum computers, we can speed up the searching step of this attack using Grover's algorithm. We then get

$$\text{Complexity}_{\text{Distinguisher; quantum}} = 2^{(n-k)/2} \cdot 3 \cdot \binom{n+v-k}{d_{\text{reg}}}^2 \cdot \binom{n+v-k}{2}.$$

6.6 Differential attacks

Differential attacks against the HFEv- scheme were intensively studied in [4]. In this paper, the authors proved that HFEv- has no differential symmetries or invariants which could be used for differential attacks.

7 Advantages and Limitations

The main advantages of the Gui signature scheme are

- **Very short signatures**. The signatures produced by the Gui signature scheme are of size about two times the corresponding security level. Therefore, Gui produces the shortest signatures of all existing digital signature schemes (both classical and post-quantum).

- **Security**. Though there exists no formal security proof which connects the security of the Gui signature scheme to a hard mathematical problem such as MQ, we are quite confident about the security of our scheme. The Gui signature scheme is based on the HFEv- signature scheme, which is one of the best known and most analyzed multivariate schemes. The only recent advance in the cryptanalysis of HFEv- like schemes is the Minors modelling of the MinRank attack [2] from 2013. However, as shown in [15], this attack can be easily prevented by increasing the numbers of minus equations and vinegar variables. Moreover, while the behaviour of direct attacks against Gui/HFEv-had long been a mystery, this problem could be solved by the works of [5, 15, 14].
  In general we can say that, in the case of the Gui signature scheme, the experimental data follow closely the theoretical complexity estimations of the known attacks. This is fundamentally different than for many other cryptographic schemes, e.g. lattice based constructions, and gives us additional confidence in the security of Gui.

- **Modest computational requirements**. Since Gui only requires simple linear algebra operations over a small finite field, it can be efficiently implemented on low cost devices, without the need of a cryptographic coprocessor [15].

- **Efficiency**. Though Gui is not one of the fastest multivariate schemes, its performance is highly comparable with that of RSA and ECC (see [15] and Section 4.2). Especially for high levels of security, the parameters of Gui and therefore the running time do not increase as drastically as in the case of RSA. We further mention here that, in the last years, the performance of schemes like Gui has improved dramatically due to new processor instructions such as `PCLMULQDQ`.

On the other hand, the main disadvantage of Gui is its **Large Public Key Size**. The public key size of Gui lies, for the parameter sets recommended in this proposal, in the range of 400 kB to 5MB and is therefore much larger than that of many classical signature schemes such as RSA and DSA and e.g. lattice based signature schemes.
On the other hand, the private key of Gui is much smaller than the public key, which allows to store the private key on small devices such as smartcards.

References

[1] L. Bettale, J.C. Faugére, L. Perret: Hybrid approach for solving multivariate systems over finite fields. J. Math. Cryptol. 3, pp. 177 - 197 (2009).

[2] L. Bettale, J.C. Faugére, L. Perret: Cryptanalysis of HFE, multi-HFE and variants for odd and even characteristic. Designs Codes and Cryptography 69 (2013), pp. 1 - 52.

[3] C. Bouillaguet, H.-C. Chen, C.-M. Cheng, T. Chou, R. Niederhagen, A. Shamir, B.-Y. Yang: Fast exhaustive search for polynomial systems in F2. CHES 2010, LNCS vol. 6225, pp. 203 - 218. Springer, 2010.

[4] R. Cartor, R. Gipson, D. Smith-Tone, J. Vates: On the Differential Security of the HFEv- Signature Primitive. PQCrypto 2016, LNCS vol. 9606, pp. 162 - 181. Springer, 2016.

[5] J. Ding, B.Y. Yang: Degree of Regularity for HFEv and HFEv-. PQCrypto 2013, LNCS vol. 7932, pp. 52 - 66. Springer, 2013.

[6] J.C. Faugére: A new efficient algorithm for computing Gröbner bases (F4). J. Pure Appl. Algebra 139, pp. 61 - 88 (1999).

[7] J.C. Faugére: Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. CRYPTO 2003, LNCS vol. 2729, pp. 44 - 60. Springer, 2003.

[8] A. Fog: Instruction tables: Lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs, 7 December 2014. `http:// www.agner.org/optimize/`

[9] A. Kipnis, A. Shamir: Cryptanalysis of the HFE public key cryptosystem by Relinearization. CRYPTO 1999, LNCS vol. 1666, pp. 19 - 30. Springer, 1999.

[10] M.S.E. Mohamed, J. Ding, J. Buchmann: Towards algebraic cryptanalysis of HFE challenge 2. ISA 2011. CCIS vol. 200, pp. 123 - 131. Springer, 2011.

[11] NIST: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/submission-requirements`

[12] J. Patarin, N.T. Courtois, L. Goubin: QUARTZ, 128-bit long digital signatures. CT-RSA 2001, LNCS, vol. 2020, pp. 282 - 297. Springer, 2001.

[13] R. Perlner, A. Petzoldt, D. Smith - Tone: Improved Cryptanalysis of HFEv- via Projection. Submitted to PQCrypto 2018. Available as IACR eprint report 2017/1149.

[14] A. Petzoldt: On the Complexity of the Hybrid Approch on HFEv-. IACR eprint report 2017/1135.

[15] A. Petzoldt, M.S. Chen, B.Y. Yang, C. Tao, J. Ding: Design principles for HFEv- based multivariate signature schemes. ASIACRYPT 2015 (Part 1), LNCS vol. 9742 , pp. 311 - 334. Springer, 2015.

[16] K. Sakumoto, T. Shirai, H. Hiwatari: On Provable Security of UOV and HFE Signature Schemes against Chosen-Message Attack. PQCrypto 2011, LNCS vol. 7071, pp 68 - 82. Springer, 2011.

[17] P. Schwabe, B. Westerbaan: Solving Binary *MQ* with Grover's Algorithm. SPACE 2016, LNCS vol. 10076, pp. 303 - 322. Springer 2016.

[18] J. Taverne, A. Faz-Hernandez, D.F. Aranha, F. Rodrguez-Henrquez, D. Hankerson, , J. Lopez: Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication. CHES 2011. LNCS vol. 6917, pp. 108123. Springer, 2011.

[19] J. Vates, D. Smith-Tone: Key recovery for all parameters of HFE-. PQCrypto 2017, LNCS 10346, pp. 272 -288. Springer, 2017.

KINDI

20171130 Submission

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Rachid El Bansarkhani, TU Darmstadt

E-mail address (preferred): `elbansarkhani@cdc.informatik.tu-darmstadt.de`

Telephone (if absolutely necessary): +49-6151-16-20667

Postal address (if absolutely necessary): Rachid El Bansarkhani, Department of Computer Science, TU Darmstadt, Hochschulstraße 10, 64289 Darmstadt.

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature:

See also printed version of "Statement by Each Submitter".

Document generated with the help of `pqskeleton` version .

Contents

1 Introduction

Lattices as mathematical objects have been studied by early mathematicians such as Gauss or Dirichlet due to its extremely rich combinatorial structure appearing in many areas of mathematics. But in the last 2 decades they have also extensively been utilized in cryptography to build powerful cryptosystems, where the security stems from the worst-case hardness of well studied lattice problems.

Beside the NTRU assumption the main computational assumptions exploited to build those cryptosystems are the hardness of the problems LWE/SIS [1, 16, 12], ring-LWE/ring-SIS [11, 14, 7, 13] and recently also MLWE/MSIS [10], which are equipped with security guarantees based on worst-case lattice problems. However, the efficiency of cryptosystems increases by imposing more structure. Thus, one almost only finds ring instantiations of the respective schemes for use in practice.

The decision problems are widely used to build lattice-based encryption schemes, where the public key and ciphertexts can be represented as LWE instances $\mathbf{A} \cdot \mathbf{s} + \mathbf{e}$. CPA-security is thus obtained almost for free.

We propose trapdoor-based encryption schemes, where the message is injected into the error term. By use of the trapdoor, the secret vector and error terms are recovered during decryption and are thus available for inspection. In many encryption schemes, this is indeed not possible. Since the message is embedded in the error term, small expansion factors can be realized at competitive parameters, i.e. we can encrypt more keys or data per (small) ciphertext bit (e.g. for sign-then-encrypt). Furthermore, it can easily be transformed to ensure CCA security. This work is based on [2, 15].

2 General algorithm specification (part of 2.B.1)

Parameter	Definition
n	power of two
$x^n + 1$	cyclotomic polynomial
$\mathbb{Z}[x]$	set of polynomials with integer coefficients
$\mathbb{Z}_b[x]$	set of polynomials with integer coefficients modulo b
$\mathcal{R}$	$\mathbb{Z}[x]/\langle x^n + 1\rangle$
$\mathcal{R}_b$	$\mathbb{Z}_b[x]/\langle x^n + 1\rangle$
$\mathcal{R}_b^d$	set of d polynomials from $\mathcal{R}_b$
q	modulus
ℓ	module rank
k	$\log q$
$[x]$	represents a polynomial in $\mathcal{R}$ with all coefficients equal to x.
$\lfloor x \rceil$	rounds x to the nearest integer.
$\mathbf{g}$	gadget $\mathbf{g} = 2^{k-1}$
$\mathbf{g}'$	gadget $\mathbf{g} = 2^{k-2}$ used for higher security levels
rsec	coefficient range $\{-\mathsf{rsec}, \ldots, \mathsf{rsec} - 1\}$ of the secrets and error used
p	$p = \mathsf{rsec}$ for simplicity
t	number of truncated bits per coefficient of the public key
$\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell}$	public uniform matrix
$\mathbf{r} \in \mathcal{R}^{\ell}$	secret key
$\bar{\mathbf{p}} \in \mathcal{R}_q^{\ell}$	public key
$\bar{\mathbf{b}} \in \mathcal{R}_q^{\ell}$	compressed public key
λ	security parameter
δ	decryption failure
μ	seed of size 2λ for $\mathbf{A} \in \mathcal{R}_q^{\ell \times \ell}$
γ	seed of size 2λ for $\mathbf{r}, \mathbf{r}'$
$\mathsf{MLWE}_{x,y,z}$	MLWE instances over a module of rank x with y samples having uniform errors in $\{-z, \ldots, z-1\}$
Secret key size	$n\ell \log 2p + n\ell(k-t) + 2\lambda$ bits
Public key size	$n\ell(k-t) + 2\lambda$ bits
Message size	$n(\ell+1)\log 2p$ bits

2.1 Trapdoor based Encryption $\mathsf{Kindi}_{\mathsf{CPA}}$ with Uniform Errors

In this section, we describe our CPA-secure Module-LWE/SIS based encryption scheme $\mathsf{KINDI}_{\mathsf{CPA}}$. It is based on the works [2, 15] and employs trapdoors in order to recover the secret vector and error term from Module-LWE instances. In fact, the scheme embeds the message into the error term and further encrypts a random string (similar to a KEM) in the secret vector, which can be exploited as a key for a symmetric key cipher. We note that our encryption scheme can be seen in some sense as a "simplified" KEM, where $\mathbf{c} = \mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg}) = (s_1 \leftarrow \mathcal{R}_2, \mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg}||s_1, \mathsf{G}(s_1)), \mathsf{msg}||s_1 \leftarrow \mathsf{Decrypt}(\mathsf{sk}, \mathbf{c})$ and $\mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Decrypt}$ just outputs msg. One part of the message, namely the random string, is always hashed with a random oracle in order to deterministically generate the secret and error term. The encryption engine $\mathsf{Encrypt}(\cdot)$ thus coincides with the deterministic encryption scheme in [9], if msg is for instance set to $\perp$. In our KEM construction we need s_1 in order to finally deduce the key, thus we take $s_1 \leftarrow \mathcal{R}_2$ out of $\mathsf{Kindi}_{\mathsf{CPA}}.\mathsf{Encrypt}$.

We now give a specification of the parameter space and the algorithms.

2.1.1 Parameter Space and Notation

We operate with the rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where $n, q = 2^k$ are powers of two and k is a positive integer. In general, we define $\mathcal{R}_b := \mathbb{Z}_b[x]/\langle x^n + 1\rangle$ for some positive integer b. Furthermore, we introduce the gadget polynomial $\mathbf{g} = 2^{k-1}$ with all coefficients being zero except for the constant. By ℓ we denote the module rank and the message size per coefficient amounts to $\log_2 \mathsf{rsec}$ bits. Let λ denote the bit security level and define $p := \mathsf{rsec}$ for simplicty. In the implementation, we use rsec instead. By $[x]$ we denote the polynomial with all coefficients equal to x.

2.1.2 Secret and Public Keys

Two seeds of size 2λ bits are generated, where $\lambda \geq 128$. The first seed μ is used to generate the public matrix $\mathbf{A} \in \mathcal{R}_q^{\ell\times\ell}$ by use of a $\mathsf{PRNG} \in \{\mathsf{Shake128}, \mathsf{Shake256}\}$, which consists of ℓ^2 uniformly distributed ring elements modulo q. This seed is public. The second seed γ of size 2λ bits is secret and serves to generate the private key $\mathbf{r} \in \mathcal{R}_q^\ell$ and the error term $\mathbf{r}'$ each consisting of ℓ ring elements with coefficients sampled uniformly at random from $\{-\mathsf{p}, \ldots, \mathsf{p}-1\}$. In particular, Shake_p generates uniform random integers from $\{0, \ldots, 2p-1\}$ with Shake and substracts p. The uncompressed public key part $\mathbf{b}$ is a module-LWE instance $\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{r}' \in \mathcal{R}_q^\ell$. The public key thus consists of $\mathsf{pk} := (\bar{\mathbf{b}}, \mu)$ and the secret key $\mathsf{sk} := (\mathbf{r}, \bar{\mathbf{p}}, \mu)$ also contains the public key required for the decryption engine when recovering the secret and error terms.

Algorithm 1: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{KeyGen}(1^n, p, k, t, \ell)$:

$\gamma, \mu \leftarrow \{0,1\}^n$
$\mathbf{A} \in \mathcal{R}_q^{\ell\times\ell} \leftarrow \mathsf{Shake}(\gamma)$
$\mathbf{r}, \mathbf{r}' \in \mathcal{R}_q^\ell \leftarrow \mathsf{Shake}_p(\mu)$
$\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{r}'$
$\bar{\mathbf{b}} = \mathsf{Compress}(\mathbf{b}, t)$
$\mathsf{pk} := (\bar{\mathbf{b}}, \mu),\ \mathsf{sk} := (\mathbf{r}, \bar{\mathbf{b}}, \mu)$
return $(\mathsf{pk}, \mathsf{sk})$

Algorithm 2: $\mathsf{Compress}(\mathbf{x} \in \mathcal{R}_q^{\ell+1}, t \in \mathbb{N})$:

Truncate the t least significant bits of each coefficient in $\mathbf{x}$.
$\bar{\mathbf{b}} = \lfloor \mathbf{x}/2^t \rfloor \in \mathcal{R}_{2^{k-t}}^{\ell+1}$.
return $\bar{\mathbf{b}}$

Compressing the public key just requires to truncate the least t significant bits. Thus, if the public key is uniform random then so the compressed one.

2.1.3 Encryption

If $\mathsf{coins} = \perp$ (required for the KEM) the encryptor samples a secret binary polynomial s_1 with coefficients from $\{0,1\}$ uniformly at random, otherwise coins already contains s_1 (for the KEM). In what follows, the matrix $\mathbf{A} \in \mathcal{R}_q^\ell$ is deterministically generated from μ. The public key is retrieved, decompressed by multiplication with 2^t and modified in such a way that the constant polynomial $\mathbf{g}$ is added to $\mathbf{p}_1$. The random binary polynomial s_1 is extended via the random oracle G implemented as Shake to $\ell - 1$ uniform random polynomials $(\mathbf{s}_2, \ldots, \mathbf{s}_\ell)$ with coefficients in the range $\{0, \ldots, 2p-1\}$, one random polynomial $\bar{\mathbf{s}}_1$ in the range $\{0, \ldots, p-1\}$, i.e. one bit per coefficient less than the other polynomials, and a bit string of size $n(\ell+1)\log 2p$ bits. To obtain a secret polynomial $\mathbf{s}_1$ over the full range $\mathcal{R}_{2p}$, we shift $\bar{\mathbf{s}}_1$ by one bit and add s_1. The message msg is xored with the uniform random string $\bar{u}$ and finally split into $n \log 2p$ bit chunks that are encoded as polynomials $\mathbf{u}_i$ from $\mathcal{R}_{2p}^n$ with $\log 2p$ bits per entry. The error terms are just $\mathbf{u}_i$ translated by p in each coefficient. The ciphertext is computed as a module-LWE instances with the same secret $\mathbf{s} = (\mathbf{s}_1 - [p], \ldots, \mathbf{s}_\ell - [p])$, where each coefficient is translated by p. To enable recovery of s_1 we adjust the last ciphertext sample via subtraction of $\mathbf{g} \cdot [p]$, which vanishes modulo q for $\mathbf{g} = 2^{k-1}$ and $p = 2^x$ with $x \geq 1$.

For $n = 256$ at a post quantum security level of 256 bits, we need $s_1 \leftarrow \mathcal{R}_4$ and $\mathbf{g} = 2^{k-2}$. The coefficients are recovered with the alternative subroutine $\mathsf{Recover}'$.

Algorithm 3: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, \mathsf{msg} = \{0,1\}^{n(\ell+1)\log 2p}; \mathsf{coins} = \perp \text{ or } s_1 \in \mathcal{R}_2)$:

$s_1 \leftarrow \mathcal{R}_2$
$\mathbf{A} \leftarrow \mathsf{Shake}(\mu)$
$\bar{\mathbf{p}} = \mathsf{Decompress}(\bar{\mathbf{b}}, t)$
$\mathbf{p} = (\bar{\mathbf{p}}_1 + \mathbf{g}, \bar{\mathbf{p}}_2, \ldots, \bar{\mathbf{p}}_\ell)$
$\bar{u} \in \{0,1\}^{n\ell \log p}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \in \{0,1\}^{n(\ell+1)\log 2p} \times \mathcal{R}_p \times \mathcal{R}_{2p}^{\ell-1} \leftarrow \mathsf{G}(s_1) := \mathsf{Shake}(s_1)$
$\mathbf{s} = (s_1 + 2 \cdot \bar{\mathbf{s}}_1 - [p], \mathbf{s}_2 - [p], \ldots, \mathbf{s}_\ell - [p])^\top$
$u = \bar{u} \oplus \mathsf{msg}$
$\mathbf{u} = \mathsf{Encode}(u)$
$\mathbf{e} = (\mathbf{u}_1 - [p], \ldots, \mathbf{u}_\ell - [p])^\top,\ e = \mathbf{u}_{\ell+1} - [p]$
$(\mathbf{c}, c)^\top = (\mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e}, \mathbf{p} \cdot \mathbf{s} + \mathbf{g} \cdot [p] + e) \in \mathcal{R}_q^{\ell+1}$
return $(\mathbf{c}, c)$

Algorithm 4: $\mathsf{Decompress}(\mathbf{x} \in \mathcal{R}_q^{\ell+1}, t \in \mathbb{N})$:

$\mathbf{b} = 2^t \cdot \mathbf{x}$

Algorithm 5: $\mathsf{Encode}(u \in \{0,1\}^{n(\ell+1)\log 2p})$:

1 Pack $\log 2p$ bits of u into each coefficient of $\mathbf{u}_i \in \mathcal{R}$ for $1 \leq i \leq \ell+1$.
2 Each $\mathbf{u}_i \in \mathcal{R}$ contains $n \cdot \log 2p$ bits.
3 $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_{\ell+1})$
4 **return u**

Theorem 2.1 *In the random oracle model, assume that there exists a PPT-adversary $\mathcal{A}$ against the scheme, then there exists a reduction $\mathcal{D}$ that breaks $\mathsf{MLWE}_{\ell,\ell+1,p}$ such that* $\mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{Kindi}(\mathcal{A})} \leq 3\mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D})$.

Proof. We proceed in a sequence of hybrids. Thus, let $\mathcal{H}_0$ be the real $\mathsf{IND}-\mathsf{CPA}$ game. In $\mathcal{H}_1$, the MLWE instance $\mathbf{b} = \mathbf{A} \cdot \mathbf{r} + \mathbf{e}$ in the key generation step is changed to a uniform random value. If there exists an adversary that can distinguish the hybrids $\mathcal{H}_0$ and $\mathcal{H}_1$, then there exists a reduction $\mathcal{D}_0$ that can distinguish $\mathsf{MLWE}_{\ell,\ell,p}$ from uniform such that $\mathsf{Adv}_{\mathcal{H}_0,\mathcal{H}_1}(\mathcal{D}_0) \leq \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell,p}(\mathcal{D}_0)$. In the hybrid $\mathcal{H}_2$, the elements $\bar{u}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell)$ are replaced by uniform random elements (RO) such that $\mathbf{e}, e, \mathbf{s}$ are again uniform random. Here too, we obtain $\mathsf{Adv}_{\mathcal{H}_1,\mathcal{H}_2}(\mathcal{D}_1) \leq \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell,p}(\mathcal{D}_1)$ for the chosen parameters. Finally, in $\mathcal{H}_3$ the ciphertexts $\mathbf{c} = \mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e}$ and $c = \mathbf{p} \cdot \mathbf{s} + \mathbf{g} \cdot [p] + e$ are replaced by uniform random elements. If there exists an adversary that can distinguish the hybrids $\mathcal{H}_2$ and $\mathcal{H}_3$, then there exists a reduction $\mathcal{D}_2$ that can distinguish $\mathsf{MLWE}_{\ell,\ell+1,p}$ from uniform such that $\mathsf{Adv}_{\mathcal{H}_2,\mathcal{H}_3}(\mathcal{D}_2) \leq \mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D}_2)$.

We now analyze the advantage of an adversary in $\mathcal{H}_0$, which is given by

$$\begin{aligned} \mathsf{Adv}_{\mathcal{H}_0}(\mathcal{A}) := \mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{Kindi}}(\mathcal{A}) &= |P[b = b'] \text{ in } \mathcal{H}_0 - 1/2| \\ &\leq \mathsf{Adv}_{\mathcal{H}_0,\mathcal{H}_1}(\mathcal{D}) + \mathsf{Adv}_{\mathcal{H}_1,\mathcal{H}_2}(\mathcal{D}) + \mathsf{Adv}_{\mathcal{H}_2,\mathcal{H}_3}(\mathcal{D}) \leq 3\mathsf{Adv}^{\mathsf{MLWE}}_{\ell,\ell+1,p}(\mathcal{D}) . \end{aligned}$$

2.1.4 Decryption

The decryption engine works similar to the encryption engine. First, the least significant bit of the coefficients of $\mathbf{s}_1$ are recovered via $s_1 = \mathsf{Recover}(\mathbf{v}) \in \mathcal{R}_2$ and $\mathbf{v} = c - \mathbf{c} \cdot \mathbf{r}^\top = 2^{k-1}\mathbf{s}_1 + \mathbf{d} \bmod q = 2^{k-1}s_1 + \mathbf{d} \bmod q$ for some small $\|\mathbf{d}\|_\infty$. This recovery function has also been used for instance in [4] avoiding if-else checks. From s_1 the vectors $\bar{u}$ and $\mathbf{s}_i$ are derived. We obtain $(\mathbf{u}_1 - [p], \ldots, \mathbf{u}_{\ell+1} - [p]) = (\mathbf{e}, e) = (\mathbf{c} - \mathbf{A}^\top \cdot \mathbf{s}, c - \mathbf{p} \cdot \mathbf{s}) \bmod q$.. The decoder $\mathsf{Decode}(\mathbf{u})$ maps the set of polynomials with coefficients in the range $[0, 2p]$ to a bit string such that the bit string $\mathbf{msg} = \mathsf{Decode}(\mathbf{u}) \oplus \bar{u}$ returns the message.

Algorithm 6: $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}(\mathsf{sk}, (\mathbf{c}, c))$:

$\mathbf{A} \leftarrow \mathsf{Shake}(\mu)$
$\bar{\mathbf{p}} = \mathsf{Decompress}(\bar{\mathbf{b}}, t)$
$\mathbf{p} = (\bar{\mathbf{p}}_1 + \mathbf{g}, \bar{\mathbf{p}}_2, \ldots, \bar{\mathbf{p}}_\ell)$
$\mathbf{v} = c - \mathbf{c} \cdot \mathbf{r}^\top$
$s_1 = \mathsf{Recover}(\mathbf{v}) \in \mathcal{R}_2$
$\bar{\mathbf{u}}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \in \{0,1\}^{n(\ell+1)\log 2p} \times \mathcal{R}_p \times \mathcal{R}_p^{\ell-1} \leftarrow \mathsf{Shake}(s_1)$
$\mathbf{s} = (s_1 + 2 \cdot \bar{\mathbf{s}}_1 - [p], \mathbf{s}_2 - [p], \ldots, \mathbf{s}_\ell - [p])^\top$
$(\mathbf{e}, e) = (\mathbf{u}_1 - [p], \ldots, \mathbf{u}_{\ell+1} - [p]) = (\mathbf{c} - \mathbf{A}^\top \cdot \mathbf{s}, c - \mathbf{p} \cdot \mathbf{s}) \bmod q$
$\mathsf{msg} = \mathsf{Decode}(\mathbf{u}) \oplus \bar{u}$
return (msg, s_1)

Algorithm 7: $\mathsf{Decode}(\mathbf{u} \in \mathcal{R}_{2p}^{\ell+1})$:

Concatenate the least significant $\log 2p$ bits of all coefficients in $\mathbf{u}$ into u.
return $u \in \{0,1\}^{n(\ell+1)\log 2p}$

Algorithm 8: $\mathsf{Recover}(\mathbf{v} \in \mathcal{R}_q)$:

Let v_i be in $\{0, \ldots, q-1\}$.
For $i = 1$ to n **do**
    $b_i = \lfloor v_i / 2^{k-1} \rceil \bmod 2$
return $b \in \mathcal{R}_2$

Algorithm 9: $\mathsf{Recover}'(\mathbf{v} \in \mathcal{R}_q)$:

Let v_i be in $\{0, \ldots, q-1\}$.
For $i = 1$ to n **do**
    $b_i = \lfloor v_i / 2^{k-2} \rceil \bmod 4$
return $b \in \mathcal{R}_4$

Theorem 2.2 *Let the coefficients of* $\mathbf{r}_j, \mathbf{r}'_j, \mathbf{s}_j - [p]$ *and* $\mathbf{e}_i = \mathbf{u}_i - [p]$ *be uniformly distributed from* $\{-p, \ldots, p-1\}$ *for* $1 \le j \le \ell$ *and* $1 \le i \le \ell + 1$. *Then, for*

$$\delta := P[\|\mathbf{w}^\top \cdot \mathbf{s} + e - \mathbf{e} \cdot \mathbf{r}^\top\|_\infty \ge q/4]$$

the algorithm is $(1 - \delta)$ *correct, where* $\mathbf{w} = \mathsf{Decompress}(\mathsf{Compress}(\mathbf{A} \cdot \mathbf{r} + \mathbf{r}')) - \mathbf{A} \cdot \mathbf{r}$.

Proof. We choose the parameters p and q such that s_1 is correctly recovered. Let $\mathbf{s} =$

$(\mathbf{s}_1 - [p], \ldots, \mathbf{s}_\ell - [p])$, then

$$\begin{aligned} \|\mathbf{v} - 2^{k-1}s_1\|_\infty &= \|c - \mathbf{c}^\top \cdot \mathbf{r} - 2^{k-1}s_1\|_\infty \\ &= \|\mathbf{p}\cdot\mathbf{s} + \mathbf{g}\cdot[p] + e - (\mathbf{A}^\top\cdot\mathbf{s} + \mathbf{e})^\top\cdot\mathbf{r} - 2^{k-1}s_1\|_\infty \\ &= \|\mathbf{g}\cdot(\mathbf{s}_1 - [p]) + \mathbf{g}\cdot[p] + e + \mathbf{w}^\top\cdot\mathbf{s} - \mathbf{e}\cdot\mathbf{r}^\top - 2^{k-1}s_1\|_\infty \\ &= \|\mathbf{w}^\top\cdot\mathbf{s} + e - \mathbf{e}\cdot\mathbf{r}^\top\|_\infty < q/4\,. \end{aligned}$$

We note that in case $p = 2^x$, then the term $\mathbf{g}\cdot[p]$ vanishes and is not needed in the computation.

For $n = 256$ and $\lambda = 256$ (key size 2λ bits resisting Grover's search), we have $\mathbf{g} = 2^{k-2}$. We define the correctness requirement with respect to the bound $q/8$ rather than $q/4$, i.e.

$$\delta := P[\|\mathbf{w}^\top\cdot\mathbf{s} + e - \mathbf{e}\cdot\mathbf{r}^\top\|_\infty \geq q/8]\,.$$

2.2 Trapdoor-based CCA-secure KEM $\mathsf{KINDI}_{\mathsf{KEM}}$ with Uniform Errors

The key encapsulation mechanism $\mathsf{KINDI}_{\mathsf{KEM}}$ has the same parameter space as $\mathsf{KINDI}_{\mathsf{CPA}}$. We adopt the transformation [9] in order to realize a KEM by our construction. In fact, we already indicated in Section 2.1 that some of the transformations are already encompassed in our construction. Thus, the construction gets very simple.

The generic construction secure in the quantum random oracle model is given by the following two algorithms, where $\mathsf{G}, \mathsf{H}, \mathsf{H}'$ denote random oracles.

Algorithm 10: $\mathsf{QEncaps}(\mathsf{pk})$:

$m \leftarrow \mathsf{M}$
$\mathbf{c} := \mathsf{Enc}(\mathsf{pk}, m, \mathsf{G}(m))$
$K := \mathsf{H}(m)$
$d := \mathsf{H}'(m, \mathbf{c})$
return $(K, \mathbf{c}, d)$

We state the theorem for tight security, when the computation and check of d is omitted. For that we combine the security implications [9] $\mathsf{IND-CPA} \implies \mathsf{OW-PCVA}$ and $\mathsf{OW-PCVA} \implies \mathsf{IND-CCA}$.

Theorem 2.3 *Let* M *denote the message space. Furthermore, for any* $\mathsf{IND-CCA}$ *adversary that makes* q_{G} *queries to the random oracle* G*,* q_{H} *queries to the random oracle* H*, and* q_D

Algorithm 11: $\mathsf{QDecaps}(\mathsf{sk}, \mathbf{c})$:

$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathbf{c})$
$\mathbf{c}' := \mathsf{Enc}(\mathsf{pk}, m', \mathsf{G}(m'))$
if $\mathbf{c}' = \mathbf{c} \wedge \mathsf{H}'(m', \mathbf{c}) = d$
  return $K := \mathsf{H}(m', \mathbf{c})$
else
  return $K := \mathsf{H}(s, \mathbf{c})$

queries to the decapsulation oracle, there exists an $\mathsf{IND}-\mathsf{CPA}$ *adversary such that*

$$\mathsf{Adv}^{\mathsf{IND-CCA}}_{\mathsf{KEM}}(\mathcal{B}) \leq q_G \cdot \delta + \frac{2 \cdot q_G + q_H}{|\mathsf{M}|} + 3\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathsf{PKE}}(\mathcal{A}) \tag{1}$$

and the running time of A *is about that of* B.

This reduction is tight. Thus, we can tightly reduce it from $\mathsf{MLWE}_{\ell,\ell+1,p}$.

For security in the quantum random oracle model, which requires d, there is an alternative theorem in [9].

2.3 Key Generation

The key generation step just outputs the keys of $\mathsf{KINDI}_{\mathsf{CPA}}$.

Algorithm 12: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{KeyGen}(1^n, p, k, d, \ell)$:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{KeyGen}(1^n, p, k, d, \ell)$
return $(\mathsf{pk}, \mathsf{sk})$

2.4 Encapsulation

The encapsulation mechanism slightly differs from the generic construction. We do not need to input $\mathsf{G}(s_1)$ but rather just s_1. In fact, the encryption engine $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}$ does this implicitly within the algorithm as it applies $\mathsf{G}(s_1)$ to deterministically deduce the secret and error polynomials. At the same time s_1 is encrypted (see $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}$). As in the generic construction, we compute the key $K \in \{0,1\}^{2\lambda}$ and $d \in \{0,1\}^{2\lambda}$. Due to the fact that KINDI has a large message container, we can also encrypt d and send a ciphertext that is as large as in $\mathsf{KINDI}_{\mathsf{CPA}}$. Finally, the ciphertext is output. We implement the different random oracles as $\mathsf{H}(s_1) := \mathsf{Shake}(s_1||\mathsf{padding}), \mathsf{H}'(s_1, \mathbf{c}) := \mathsf{Shake}(s_1, \mathbf{c})$ and $\mathsf{G}(s_1) := \mathsf{Shake}(s_1)$, where we use a one byte $\mathsf{padding} = 4$.

Algorithm 13: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{Encaps}(\mathsf{pk})$:

$s_1 \leftarrow \{0,1\}^n$
$d \leftarrow \mathsf{H}(s_1)$
$(\mathbf{c}, c)^\top \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, d; s_1)$
$K \leftarrow \mathsf{H}'(s_1, (\mathbf{c}, c))$
return K

2.5 Decapsulation

The decapsulation mechanism implicitly performs many steps of the generic construction within $\mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}$. For instance, it is not required to encrypt s_1' again once recovered from the ciphertext as we prove below. It is only necessary, to check that the decrypted d is equal to the computed d'. In case, the check is correct the key is deduced, otherwise it outputs a random key for some uniform random $s \in \{0,1\}^{2\lambda}$.

Algorithm 14: $\mathsf{KINDI}_{\mathsf{CCA-KEM}}.\mathsf{Decaps}(\mathsf{sk}, (\mathbf{c}, c, d))$:

$(d', s_1') \leftarrow \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Decrypt}(\mathsf{sk}, (\mathbf{c}, c))$
if $d' = d$
    return $\mathsf{H}'(s_1', (\mathbf{c}, c))$
else
    return $\mathsf{H}'(s, (\mathbf{c}, c))$

In the following lemma we show that it suffices to only check $d' = d$ in order to satisfy the conditions from [9] for key decapsulation.

Lemma 2.4 *If $d' = d$ is satisfied, then $s_1' = s_1$ and $(\mathbf{c}, c) = \mathsf{KINDI}_{\mathsf{CPA}}.\mathsf{Encrypt}(\mathsf{pk}, d; s_1')$.*

Proof. If $d' = d$, then $\mathsf{G}(s_1) = \mathsf{G}(s_1')$, which means that s_1' has been correctly recovered. As a result, we have that $\mathbf{c}, c$ is uniquely generated from $\mathbf{s}$ and $\mathbf{u} = \mathsf{Encode}(\bar{u} \oplus d)$ with

$$\bar{\mathbf{u}}, \bar{\mathbf{s}}_1, (\mathbf{s}_2, \ldots, \mathbf{s}_\ell) \leftarrow \mathsf{G}(s_1')$$

and $\mathbf{s}_1 = s_1 + 2\bar{\mathbf{s}}_1$.

We note that if $\mathsf{KINDI}_{\mathsf{CPA}}$ is $(1-\delta)$ correct, then so is the resulting $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$.

3 List of parameter sets (part of 2.B.1)

3.1 Parameter set $\mathsf{encrypt/KINDI-256-3-4-2}$

Public key encryption with $n = 256,\ \ell = 3,\ p = 4,\ t = 2$ and $q = 2^{14}$.

3.2 Parameter set encrypt/KINDI − 512 − 2 − 2 − 2

Public key encryption with $n = 512,\ \ell = 2,\ p = 2,\ t = 2$ and $q = 2^{13}$.

3.3 Parameter set encrypt/KINDI − 512 − 2 − 4 − 1

Public key encryption with $n = 512,\ \ell = 2,\ p = 4,\ t = 1$ and $q = 2^{14}$.

3.4 Parameter set encrypt/KINDI − 256 − 5 − 2 − 2

Public key encryption with $n = 256,\ \ell = 5,\ p = 2,\ t = 2$ and $q = 2^{14}$.

3.5 Parameter set encrypt/KINDI − 512 − 3 − 2 − 1

Public key encryption with $n = 512,\ \ell = 3,\ p = 2,\ t = 1$ and $q = 2^{13}$.

3.6 Parameter set kem/KINDI − 256 − 3 − 4 − 2

Key encapsulation mechanism with $n = 256,\ \ell = 3,\ p = 4,\ t = 2$ and $q = 2^{14}$.

3.7 Parameter set kem/KINDI − 512 − 2 − 2 − 2

Key encapsulation mechanism with $n = 512,\ \ell = 2,\ p = 2,\ t = 2$ and $q = 2^{13}$.

3.8 Parameter set kem/KINDI − 512 − 2 − 4 − 1

Key encapsulation mechanism with $n = 512,\ \ell = 2,\ p = 4,\ t = 1$ and $q = 2^{14}$.

3.9 Parameter set kem/KINDI − 256 − 5 − 2 − 2

Key encapsulation mechanism with $n = 256,\ \ell = 5,\ p = 2,\ t = 2$ and $q = 2^{14}$.

3.10 Parameter set kem/KINDI − 512 − 3 − 2 − 1

Key encapsulation mechanism with $n = 512,\ \ell = 3,\ p = 2,\ t = 1$ and $q = 2^{13}$.

4 Design rationale (part of 2.B.1)

We propose a simple and highly efficient trapdoor-construction, where the public key $\mathbf{B}$ represents an MLWE instance endowed with a trapdoor $\mathbf{T}$. Roughly spoken, ciphertexts are generated as MLWE instances $\mathbf{B}^\top \cdot \mathbf{s} + \mathbf{e}$, where $\mathbf{s}$ and $\mathbf{e} = \mathbf{u} \oplus \mathsf{msg}$ are vectors of uniform random polynomials. The message is simply xored to the error polynomials such that large amounts of data can be encrypted at very competitive parameters, for instance useful in sign-then-encrypt scenarios or for the transmission of encrypted key bundles etc. Different to other proposals, the decryption engine can recover all $\mathbf{s}, \mathbf{e} = \mathbf{u} \oplus \mathsf{msg}$ and thus msg by means of the trapdoor $\mathbf{T}$. This further allows to inspect the secret and error polynomials and use all of the information stored therein. Our proposal not only encrypts arbitrary messages, but also outputs by construction a uniform random string s_1 for free that can act as a key for a symmetric key cipher as required in a KEM. In other words, the random coins used to encrypt the message can be recovered by use of the trapdoor.

We choose to implement random oracles with the FIPS 202 standardized Shake. It is also used in other lattice-based schemes such as Frodo and Kyber. The matrix $\mathbf{A}$ is generated by use of a $\mathsf{PRNG} \in \{\mathsf{Shake128}, \mathsf{Shake256}\}$ and a uniform random input string μ of size 2λ bits. In fact, we only use $\mathsf{Shake256}$ except for one parameter set namely $n = 256$ and $\ell = 3$. For the optimized variants we use the Keccak code package[1] that allows via AVX2 to compute 4 independent streams of random values on 4 inputs of the same length. Each input $\mathsf{input}_i = \mu || i$ is obtained by the seed concatenated with a one byte number $0 \leq i \leq 3$ resulting in independent uniform random streams. Thus, we do not store $\mathbf{A}$ but rather derive it from μ each time we need it. Since we work modulo 2^k, each k bit chunk is considered as a little endian integer representing one coefficient. Similarly, we generate uniform secrets and errors just from $\mathsf{Shake}(s_1)$. Our choice for p to be a power of two allows us to proceed as with the matrix $\mathbf{A}$ taking the required bits from Shake for $\bar{u}$,$\bar{\mathbf{s}}_\mathbf{1}$ and $\mathbf{s}_2, \ldots, \mathbf{s}_\ell$. For the random oracle G we use the same padding scheme in our optimized variant. The message is xored to $\bar{u}$ prior to encoding.

However, for the computation of $d \in \{0,1\}^{2\lambda}$ in the KEM we append 4 to s_1 before invoking $\mathsf{H} := \mathsf{PRNG}$. We implement $\mathsf{H}' := \mathsf{PRNG}$ without any padding in the reference implementation. For the optimized variant we split the large ciphertext into 4 inputs and invoke Shake outputting 4 streams of size 2λ bits each. The outputs are subsequently concatenated to s_1 serving as input to one regular Shake call.

We mark the end of a message by a one byte padding. For polynomial multiplication in $O(n \log n)$ we make use of the FFT transformation, which is a divide-and-conquer algorithm. Our AVX2 optimized variant processes 4 coefficients at once. Furthermore, we precompute tables containing powers of the complex root of unity. Modulo $q = 2^k$ operations are obtained almost for free as it just requires to take the k least significant bits.

The ciphertexts, compressed public keys and secret keys are stored in little endian format. The $k - t$ bit coefficients of the compressed public key are appended to each other before

[1] https://keccak.team/

the seed μ is concatenated to the resulting string. For the secret key we proceed similarly.

5 Detailed performance analysis (2.B.2)

5.1 Description of platform

We implemented both our CPA/CCA secure schemes on a machine that is specified by an Intel Core i5-6200U processor (Skylake) operating at 2.3GHz and 8GB of RAM running on one core. We used Ubuntu 17.10 64-bit (Kernel 4.13.0-17) and gcc version 7.2.0 with compilation flags

- Reference: `-fomit-frame-pointer -Ofast -march=native`
- AVX Version: `-fomit-frame-pointer -Ofast -msse2avx -mavx2 -march=native`

5.2 Time

The following measurements are for `kem` and `encrypt`. The difference in running times between `kem` and `encrypt` stems from 3 additional invocations of Shake for `kem`. We took the average over 1 Mio measurements.

`Kindi-256-3-4-2`:

- Reference Implementation:
  - Key generation in cycles: 203096
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(247793,260137)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(312211,323947)
- AVX Implementation:
  - Key generation in cycles: 104308
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(122648,133888)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(151723,162070)

We note that in case we use Shake256 key generation increases by about 3000-6000 cycles, encryption by about 3000-8000 cycles, encaps by about 8000-10000, decryption by 5000-6000 cycles and decaps by about 7000-10000 cycles (for the reference implementation and AVX implementation). The decryption failure rate is here $\delta = 2^{-192}$. In the AVX implementation, encryption is carried out at a speed of 320 cycles per message byte or 68 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 396 cycles per message byte or 84 cycles per ciphertext byte.

`Kindi-512-2-2-2`:

- Reference Implementation:
  - Key generation in cycles: 214064
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(280420,306043)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(377962,397147)
- AVX Implementation:
  - Key generation in cycles: 113082
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(142950,160150)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(187097,202458)

The decryption failure rate is here $\delta = 2^{-284}$. In the AVX implementation, encryption is carried out at a speed of 373 cycles per message byte or 57 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 488 cycles per message byte or 75 cycles per ciphertext byte.

`Kindi-512-2-4-1`:

- Reference Implementation:
  - Key generation in cycles: 215542
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(285832,307999)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(382958,402041)
- AVX Implementation:
  - Key generation in cycles: 114356
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(142961,160905)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(186397,202330)

The decryption failure rate is here $\delta = 2^{-165}$. In the AVX implementation, encryption is carried out at a speed of 248 cycles per message byte or 53 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 324 cycles per message byte or 69 cycles per ciphertext byte.

`Kindi-256-5-2-2`:

- Reference Implementation:
  - Key generation in cycles: 519010
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(595043,623436)

  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(701763,723922)

- AVX Implementation:
  - Key generation in cycles: 249776
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(280265,298163)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(328537,342016)

The decryption failure rate is here smaller than $\delta = 2^{-216}$.In the AVX implementation, encryption is carried out at a speed of 731 cycles per message byte or 104 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 857 cycles per message byte or 122 cycles per ciphertext byte.

`Kindi-512-3-2-1`:

- Reference Implementation:
  - Key generation in cycles: 429952
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(530173,562640)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(672720,698041)
- AVX Implementation:
  - Key generation in cycles: 216600
  - Encryption/Encaps in cycles: (`encrypt`,`kem`)=(256730,282120)
  - Decryption/Decaps in cycles: (`encrypt`,`kem`)=(325113,339830)

The decryption failure rate is here $\delta = 2^{-276}$. In the AVX implementation, encryption is carried out at a speed of 502 cycles per message byte or 77 cycles per ciphertext byte, whereas decryption is accomplished at a rate of 636 cycles per message byte or 97 cycles per ciphertext byte.

5.3 Space

The secret key, public key and ciphertext sizes can be computed straightforwardly. They are the same for the encryption scheme and the KEM.

The ciphertext size is $n(\ell+1)k/8$ bytes, whereas the public key pk amounts to $(n\ell(k-t)+2\lambda)/8$ bytes including the seed μ for the matrix $\mathbf{A}$. The secret key amounts to $(n\ell(k-t+\log 2p)+2\lambda)/8$ bytes including the size of the public key. The message size for encryption amounts to $n\ell(\log 2p)/8-1$ bytes, where one byte is used for the padding. Thus, we obtain the following.

`Kindi-256-3-4-2`:

- Ciphertext size: 1792 bytes
- Public key size: 1184 bytes
- Secret key size: 1472 bytes
- Message size: 383 bytes
- Message expansion factor: 4.7

`Kindi-512-2-2-2`:

- Ciphertext size: 2496 bytes
- Public key size: 1456 bytes
- Secret key size: 1712 bytes
- Message size: 383 bytes
- Message expansion factor: 6.5

`Kindi-512-2-4-1`:

- Ciphertext size: 2688 bytes
- Public key size: 1728 bytes
- Secret key size: 2112 bytes
- Message size: 575 bytes
- Message expansion: 4.7

`Kindi-256-5-2-2`:

- Ciphertext size: 2688 bytes
- Public key size: 1984 bytes
- Secret key size: 2304 bytes
- Message size: 383 bytes
- Message expansion factor: 7

`Kindi-512-3-2-3`:

- Ciphertext size: 3328 bytes
- Public key size: 2368 bytes
- Secret key size: 2752 bytes
- Message size: 511 bytes
- Message expansion: 6.5

5.4 How parameters affect performance

The main parameters governing the performance and security level of the schemes are n, ℓ, q and $p = \mathsf{rsec}$. For increasing parameters n, p or ℓ the security of the overall system is increased while simultaneously decreasing the performance level via n and ℓ or increasing the secret key size at a higher decyption failure rate via p. For increasing q and all other parameters being fixed, the decryption failure rate and the security of the system decrease while the ciphertext and public key sizes increase.

5.5 Optimizations

In pricipal, it is possible to generate the private and public keys just by use of the secret seed γ and the public seed μ. If in applications, the running time is of interest, then all keys and the matrix $\mathbf{A}$ are stored rather than the seeds. In case, key sizes are more important than running time, then one may store only the seeds and generate the respective keys during decryption or decapsulation. Furthermore, it is possible to compress the ciphertext in case the message container is not fully exhausted, i.e. one can compress the coefficients of $\mathbf{c}_i$ if the respective error terms do not contain message bits. For the simplicity of our construction, we did not include these modifications.

6 Expected strength (2.B.4) in general

6.1 Security definitions

The KEM is designed for IND-CCA2 security and PKE ensures CPA security. See Section 7 for quantitative estimates of the security of specific parameter sets.

6.2 Rationale

See Section 8 for an analysis of known attacks. This analysis also presents the rationale for these security estimates.

7 Expected strength (2.B.4) for each parameter set

7.1 Parameter set encrypt/KINDI − 256 − 3 − 4 − 2

Classical security	PQ-security	Category
181	164	2

7.2 Parameter set encrypt/KINDI − 512 − 2 − 2 − 2

Classical security	PQ-security	Category
229	207	4

7.3 Parameter set encrypt/KINDI − 512 − 2 − 4 − 1

Classical security	PQ-security	Category
255	232	4

7.4 Parameter set encrypt/KINDI − 256 − 5 − 2 − 2

Classical security	PQ-security	Category
270	251	5

7.5 Parameter set encrypt/KINDI − 512 − 3 − 2 − 1

Classical security	PQ-security	Category
365	330	5

7.6 Parameter set kem/KINDI − 256 − 3 − 4 − 2

Classical security	PQ-security	Category
181	164	2

7.7 Parameter set kem/KINDI − 512 − 2 − 2 − 2

Classical security	PQ-security	Category
229	207	4

7.8 Parameter set kem/KINDI − 512 − 2 − 4 − 1

Classical security	PQ-security	Category
255	232	4

7.9 Parameter set kem/KINDI − 256 − 5 − 2 − 2

Classical security	PQ-security	Category
270	251	5

7.10 Parameter set kem/KINDI − 512 − 3 − 2 − 1

Classical security	PQ-security	Category
365	330	5

8 Analysis of known attacks (2.B.5)

We give a summary of the most relevant attacks against our MLWE based encryption schemes. To this end, MLWE instances are considered as regular LWE instances of dimension $\ell \cdot n$ with $(\ell + 1) \cdot n$ samples. To date, there exist no better cryptanalytic algorithms to attack MLWE for concrete parameters than the ones on regular LWE. The best way to attack our encryption schemes is to mount a key recovery attack or to consider attacks against the ciphertext. Since we chose the same parameter for the ciphertext and public key, we need only to consider attacks against the ciphertext since it further contains an additional ring sample as compared to the public key. We apply the conservative methodology of [3] in order to estimate the attack complexity or to choose reasonable parameters. Currently, the best way to attack the system is carried out with the primal and dual attacks using BKZ. This lattice reduction algorithm reduces the basis of the lattice using polynomial calls to an SVP oracle in a smaller dimension. For the computation of the attack complexity only one call to the SVP oracle is taken into account. All other factors are also removed leading to very conservative estimates.

In the classical setting the best-known attack bound is $2^{0.292b}$ deduced from lattice sieving whereas in the post-quantum setting the SVP solver requires $2^{0.265b}$ with quantum sieving. Here b denotes the block size. The best plausible security estimates rely on building lists of

size $2^{0.2075b}$.

The primal attack on our cryptosystem consists in finding a unique solution $(\mathbf{s}, \mathbf{e}, 1)$ for the SIS instance $[\ \mathbf{P}^\top \mid \mathbf{I} \mid\ -\mathbf{c}] \cdot \mathbf{x} \equiv \mathbf{0} \bmod q$ for $\mathbf{x} \in \mathbb{Z}^{n(2\ell+1)+1)}$ and pk considered as a matrix $\mathbf{P} = [\ \mathbf{A} \mid \mathbf{p}^\top] \in \mathbb{Z}_q^{n\ell \times (n\ell+1)}$ and $\mathbf{c} \in \mathbb{Z}_q^{n(\ell+1)}$.

For the dual attack the attacker tries to find a short vector in the dual lattice that is employed to distinguish MLWE samples from uniform random samples in order to break decision-MLWE.

We do not need to take (quantum) hybrid attacks [8] into account as they often only get significant for sparse binary or trinary secrets and errors, which is never the case for the chosen parameters. Those attacks are based on Howgrave-Graham's Hybrid Attack combining lattice reduction with guessing techniques such as brute-force or meet-in-the-middle attacks.

Algebraic attacks such as finding short generators do not apply in our setting as the parameters required for a successful attack are far from being practical [5, 6].

9 Advantages and limitations (2.B.6)

The encryption scheme KINDI is a simple and flexible trapdoor-based encryption scheme, which by use of the trapdoor allows to retrieve back the error term and secret key from Module-LWE based ciphertexts. By this, it is possible to inspect all the constituents, if they comply with the allowed parameters in order to detect, for instance, inadmissible error terms. Furthermore, lattice-based trapdoor-constructions are used in many areas of cryptography, not only for encryption or KEMs. Thus, KINDI may serve as a basis for new primitives. For instance, when using a slightly modified $\mathsf{KINDI}_{\mathsf{CPA}}$ in combination with a random oracle tag $\mathsf{mac} = H(\mathbf{s}, \mathbf{e})$, we already obtain a scheme that can be employed in CCA2-secure scenarios, since an adversary needs to know the unique inputs in order compute mac or differently spoken a correct ciphertext requires already to show knowledge of all its inputs via the mac.

In addition, KINDI allows to encrypt huge amount of data at once resulting in low message expansion factors as compared to other proposals since the error serves to transport the message. This is particularly interesting for sign-then-encrypt scenarios or for the transport of key bundles etc. For appropriate parameters signatures (uniform or Gauss) could also act as the error term, in this case the encryption scheme needs not to compute $\bar{u}$. Our proposal always by construction encrypts both a uniform random key s_1 and arbitrary messages. Thus, it inherently tends to act as a KEM. Due to this, we see that many steps from [9] are already implicit in our CPA-secure construction resulting in very small overhead. In fact, even the generation of s_1 in $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$ is implicitly accomplished in the encryption engine. Due to uniform random secrets and error vectors generated by SHAKE and the applied operations our implementations are constant-time.

There exist a wide range of parameters for various security levels. Increasing the parameters allows to encrypt more data at once without loosing efficiency. For $\mathsf{rsec} = 1$ we obtain binary errors, in this case we can even apply the NTT transform.

Our $\mathsf{KINDI}_{\mathsf{CCA-KEM}}$ can easily be deployed into the TLS protocol as shown by Google for `NewHope` or in constrained devices or can be transformed into an authenticated key exchange protocol using known transformations.

References

[1] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.

[2] Rachid El Bansarkhani, Özgür Dagdelen, and Johannes A. Buchmann. Augmented learning with errors: The untapped potential of the error term. *IACR Cryptology ePrint Archive*, 2014:733, 2014.

[3] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1006–1018, 2016.

[4] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570, 2015.

[5] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. *IACR Cryptology ePrint Archive*, 2015:313, 2015.

[6] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 324–348, 2017.

[7] Henri Gilbert, editor. *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[8] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. In *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, pages 184–202, 2017.

[9] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. *IACR Cryptology ePrint Archive*, 2017:604, 2017.

[10] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptography*, 75(3):565–599, 2015.

[11] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. *Electronic Colloquium on Computational Complexity (ECCC)*, (142), 2005.

[12] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 372–381, 2004.

[13] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-lwe for any ring and modulus. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 461–473, 2017.

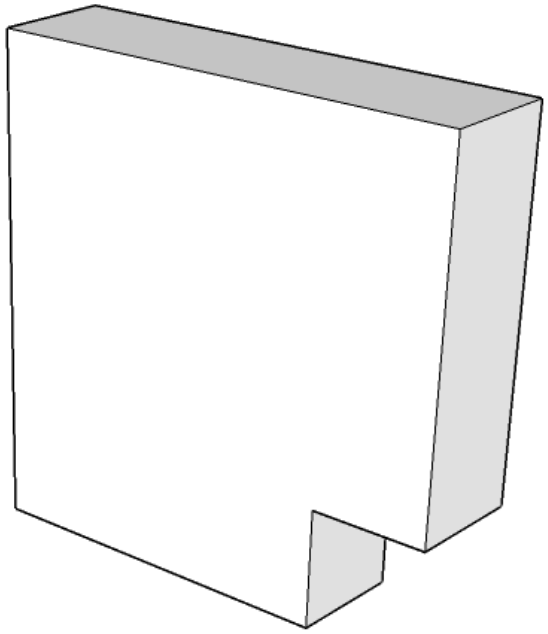
[14] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. *Electronic Colloquium on Computational Complexity (ECCC)*, (158), 2005.

[15] El Bansarkhani Rachid. Lara - a design concept for lattice-based encryption. Cryptology ePrint Archive, Report 2017/049, 2017. https://eprint.iacr.org/2017/049.

[16] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93, 2005.

LUOV

Signature Scheme proposal for NIST PQC Project



Principal submitter	Ward Beullens, imec-COSIC KU Leuven `ward.beullens@esat.kuleuven.be` +32471 12 64 57 Afdeling ESAT - COSIC, Kasteelpark Arenberg 10 - bus 2452, 3001 Heverlee, Belgium
Auxiliary submitters	Bart Preneel, imec-COSIC KU Leuven Alan Szepieniec, imec-COSIC KU Leuven Frederik Vercauteren, imec-COSIC KU Leuven
Inventors/developers	The same as the principal submitter. Relevant prior work is credited below where appropriate.
Owner	Same as submitter
Signature	

Contents

1 Introduction

One of the major candidates for providing secure cryptographic primitives in a post-quantum world is Multivariate Cryptography. Multivariate Cryptography is based on the hardness of problems related to multivariate polynomials over finite fields, such as solving systems of multivariate polynomial equations. In general, Multivariate Cryptography is very fast and requires only moderate computational resources, which makes it attractive for applications in low-cost devices. In the field of Multivariate Cryptography, the Unbalanced Oil and Vinegar signature scheme (UOV) is one of the oldest and best studied cryptosystems. Since the proposal of the Oil and Vinegar scheme in 1997 by Patarin [15], UOV has successfully withstood almost two decades of cryptanalysis. The UOV scheme is very simple, has small signatures and is fast. The main disadvantage of UOV is arguably that its public keys are quite large. This document presents the Lifted Unbalanced Oil and Vinegar signature scheme (LUOV), which is a simple improvement of the UOV scheme that greatly reduces the size of the public keys.

2 Algorithm specification (part of 2.B.1)

2.1 Overview of the scheme

The LUOV signature scheme uses a one-way function $\mathcal{P} : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^m$, which is a multivariate quadratic polynomial map in $n = m + v$ variables with coefficients in the binary subfield $\mathbb{F}_2 \subset \mathbb{F}_{2^r}$. The trapdoor is a factorization $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$, where $\mathcal{T} : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^n$ is an invertible linear map, and $\mathcal{F} : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^m$ is a quadratic map whose components $f_1, \cdots, f_m$ are of the form

$$f_k(\mathbf{x}) = \sum_{i=1}^{v} \sum_{j=i}^{n} \alpha_{i,j,k} x_i x_j + \sum_{i=1}^{n} \beta_{i,k} x_i + \gamma_k \,,$$

where the $\alpha_{i,j,k}, \beta_{i,k}$ and γ are chosen randomly from $\mathbb{F}_2$ and $v = n - m$. We say that the first v variables $x_1, \cdots, x_v$ are the *vinegar* variables, whereas the remaining m variables are the *oil* variables. Equivalently, the components of $\mathcal{F}$ are quadratic polynomials with random binary coefficients in the variables x_i such that there are no quadratic terms which contain two oil variables. One could say that the vinegar variables and the oil variables are not fully mixed, which is where their names come from.

How does the trapdoor $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$ help to invert the function $\mathcal{P}$? Given a target $\mathbf{x} \in \mathbb{F}_{2^r}^m$ a solution $\mathbf{y}$ for $\mathcal{P}(\mathbf{y}) = \mathbf{x}$ can be found by first solving $\mathcal{F}(\mathbf{y}') = \mathbf{x}$ for $\mathbf{y}'$ and then computing $\mathbf{y} = \mathcal{T}^{-1}(\mathbf{y}')$. The system $\mathcal{F}(\mathbf{y}') = \mathbf{x}$ can be solved efficiently by fixing the vinegar variables to some pseudo-randomly chosen values. If we substitute these values in the equations the remaining system only contains linear equations, because every quadratic term contains at least one vinegar variable and thus turns into a linear or constant term after substitution. The remaining linear system can be solved using linear algebra. In the event that there are no solutions we can simply try again with a different assignment to the vinegar variables.

The trapdoor function is then combined with a collision resistant hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{F}_{2^r}^m$ into a signature scheme using the standard hash-and-sign paradigm. The resulting key generation, signature generation and verification algorithms are described in the next few sections.

A large part of the coefficients of $\mathcal{P}$ is generated from a seed. This seed is included in the public key and replaces all the generated coefficients to make the public key much smaller. In order to reduce the size of the secret key we do not store $\mathcal{F}$ nor $\mathcal{T}$. Instead, we only store a private seed that was used to generate the public seed and $\mathcal{T}$.

The LUOV scheme can be used in two modes. One option is the usual appended signature mode where a message is authenticated by appending a signature. A different option is the message recovery mode, which can be used to reduce the size of a message-signature pair. In message recovery mode (part of) the message is not transmitted but recovered from the signature.

2.2 Relation to the UOV scheme

The LUOV scheme is an adaptation of the Unbalanced Oil and Vinegar signature scheme. It differs from the original UOV scheme in a number of ways. The first modification, due to Petzoldt [16], changes the key generation algorithm to make it possible to choose a large part of the public key. One can then choose this part to correspond with the output of a pseudo-random number generator and replace a large part of the public key by a seed. The modified key generation algorithm generates a distribution of public polynomial maps $\mathcal{P}$ that is indistinguishable from the original signature scheme if we assume the output of the PRNG (we have used the Keccak1600 Sponge construction) is indistinguishable from true randomness.

A second modification is that a public key $\mathcal{P} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ for the UOV scheme over $\mathbb{F}_2$ is used as a public key for the UOV scheme over a large extension field $\mathbb{F}_{2^r}$. The public key is 'lifted' to the extension field by just extending the polynomial map $\mathcal{P}$ to a map from $\mathbb{F}_{2^r}^n$ to $\mathbb{F}_{2^r}^m$. This is were the Lifted UOV scheme gets its name from. The advantage of this approach is that the public key remains small (since the coefficients of the public key are 0 or 1), while solving the system $\mathcal{P}(x) = y$ for some y in $\mathbb{F}_{2^r}^m$ becomes more difficult compared to the case where y is in $\mathbb{F}_2^m$. This adaptation is due to Beullens and Preneel. [5].

Thirdly, the linear map $\mathcal{T}$ is chosen to have a matrix representation of the form

$$\begin{pmatrix} \mathbf{1}_v & \mathbf{T} \\ 0 & \mathbf{1}_m \end{pmatrix},$$

where $\mathbf{T}$ is a v-by-m matrix. This choice makes the key generation algorithm and the signing algorithm much faster, but does not affect the security of the scheme because for a random public key there exists an equivalent private key with $\mathcal{T}$ of this form with high probability [18]. This implies that if there is an attack against the modified signature scheme, the same attack would work on nearly all public keys of the original UOV scheme. This choice of $\mathcal{T}$ was first

proposed by Czypek [7], where it was used to speed up the key generation algorithm. LUOV makes the same choice of $\mathcal{T}$, but uses different key generation and signature generation algorithms that are even faster.

Lastly, in the signing algorithm, instead of choosing the assignments to the vinegar variables truly randomly, the assignments are deterministically generated from the message M and the private key. This ensures that when a message is signed multiple times, the generated signatures will be identical. If the vinegar variables were chosen at random, an attacker could query many different signatures for the same message. We are not aware of an attack that exploits this fact, but it is cautious to block this kind of attack anyway.

2.3 Parameter space

The parameters for the LUOV algorithm are :

- r — The degree of the field extension $\mathbb{F}_2 \subset \mathbb{F}_{2^r}$.
- m — The number of polynomials in the public key, also the number of oil variables.
- v — The number of vinegar variables.
- $n = m + v$ — The total number of variables
- SHAKE — The extendable output function that is used, either SHAKE128 or SHAKE256.

2.4 Key Generation Algorithm

The key generation algorithm (Alg. 4) first uses a private seed to pseudo-randomly generate a seed that will be published, as well as the v-by-m matrix that determines the linear map $\mathcal{T}$. Then, the public seed is used to generate $\mathbf{C} \in \mathbb{F}_2^m$, the constant part of the public map $\mathcal{P}$, $\mathbf{L} \in \mathbb{F}_2^{m \times n}$, the linear part of $\mathcal{P}$ and $\mathbf{Q}_1 \in \mathbb{F}_2^{m \times \frac{v(v+1)}{2} + vm}$, the first $\frac{v(v+1)}{2} + vm$ columns of the Macaulay matrix of the quadratic part of $\mathcal{P}$ in the lexicographic ordering. Then $\mathbf{Q}_2 \in \mathbb{F}_2^{m \times \frac{m(m+1)}{2}}$, the remaining part of the Macaulay matrix of the quadratic part of $\mathcal{P}$ is calculated (see Sect. 2.4.1). The public key consists of the public seed and $\mathbf{Q}_2$. The private key is simply the seed that was used as input for the key generation algorithm. The details of how the different objects are sampled from the SHAKE function are described in Sect. 2.9.

2.4.1 Finding the remaining coefficients of $\mathcal{P}$

For each polynomial p_k in the public map $\mathcal{P}$ there is a uniquely determined upper triangular matrix $\mathbf{P}_k \in \mathbb{F}_2^{n \times n}$, such that $\mathbf{x}^\top \mathbf{P}_k \mathbf{x}$ is equal to the evaluation of the quadratic part of p_k

at $\mathbf{x}$. The matrix corresponding to the polynomial f_k in the secret map $\mathcal{F}$ is then, up to the addition of a skew-symmetric matrix, equal to

$$\begin{pmatrix} \mathbf{1}_v & \mathbf{0} \\ -\mathbf{T}^\top & \mathbf{1}_m \end{pmatrix} \begin{pmatrix} \mathbf{P}_{k,1} & \mathbf{P}_{k,2} \\ \mathbf{0} & \mathbf{P}_{k,3} \end{pmatrix} \begin{pmatrix} \mathbf{1}_v & -\mathbf{T} \\ \mathbf{0} & \mathbf{1}_m \end{pmatrix} = \begin{pmatrix} \mathbf{P}_{k,1} & -\mathbf{P}_{k,1}\mathbf{T} + \mathbf{P}_{k,2} \\ -\mathbf{T}^\top\mathbf{P}_{k,1} & \mathbf{T}^\top\mathbf{P}_{k,1}\mathbf{T} - \mathbf{T}^\top\mathbf{P}_{k,2} + \mathbf{P}_{k,3} \end{pmatrix},$$

where we have split up the matrix $\mathbf{P}_k$, into $\mathbf{P}_{k,1} \in \mathbb{F}_2^{v\times v}$, $\mathbf{P}_{k,2} \in \mathbb{F}_2^{v\times m}$ and $\mathbf{P}_{k,3} \in \mathbb{F}_2^{m\times m}$. The terms of f_k that are quadratic in the vinegar variables have to vanish, so

$$\mathbf{P}_{k,3} = -\mathbf{T}^\top\mathbf{P}_{k,1}\mathbf{T} + \mathbf{T}^\top\mathbf{P}_{k,2}\,,$$

up to the addition of a skew-symmetric matrix. This formula completely determines the upper triangular matrix $\mathbf{P}_{k,3}$. The entries of the $\mathbf{P}_{k,1}$ and $\mathbf{P}_{k,2}$ are generated from the public seed and the matrix $\mathbf{T}$ is known, so the matrices $\mathbf{P}_{k,3}$ can easily be computed. The entries of the matrices $\mathbf{P}_{k,3}$ are then arranged in the Macaulay matrix $\mathbf{Q}_2$. A detailed implementation of this procedure is shown in Alg. 3.

Algorithm findPk1

input: k — An integer between 1 and m.
$\mathbf{Q}_1$ — First part of Macaulay matrix of the quadratic part of $\mathcal{P}$

output: $\mathbf{P}_{k,1}$ — The v-by-v matrix representing the part of p_k that is quadratic in the vinegar variables.

```
1:  P_{k,1} ← 0_v
2:  column ← 1
3:  for i from 1 to v do
4:  |  for j from i to v do
5:  |  |  P_{k,1}[i,j] ← Q_1[k, column]
6:  |  |  column ← column + 1                    ▷ move to the next term
7:  |  end for
8:  |  column ← column + m                       ▷ Skip the terms x_i x_{v+1} up to x_i x_{v+m}
9:  end for
10: return P_{k,1}
```

Alg. 1: Algorithm for reading $\mathbf{P}_{k,1}$ from $\mathbf{Q}_1$.

2.5 Signature Generation Algorithm

The signature generation algorithm first generates $\mathbf{C}, \mathbf{L}, \mathbf{Q}_1$ and $\mathbf{T}$ from the private seed in the same way as the key generation algorithm. Then, it calculates $\mathbf{h}$, the hash digest of the message that will be signed, concatenated with a zero. Concatenating the message with zero is done to make signatures generated in appended signature mode unrelated to signatures generated in message recovery mode (see Sect. 2.7). Then, the algorithm produces

Algorithm findPk2

input: k — An integer between 1 and m.
$\mathbf{Q}_1$ — First part of Macaulay matrix of quadratic part of $\mathcal{P}$

output: $\mathbf{P}_{k,2}$ — The v-by-m matrix representing the part of p_k that is bilinear in the vinegar variables and the oil variables.

$\mathbf{P}_{k,2} \leftarrow \mathbf{0}_{v\times m}$
column $\leftarrow 1$
for i from 1 to v **do**
  column $\leftarrow$ column $+ v - i + 1$ ▷ Skip terms x_i^2 up to $x_i x_v$
  for j from 1 to m **do**
    $\mathbf{P}_{k,2}$[i,j] $\leftarrow \mathbf{Q}_1[k,$ column$]$
    column $\leftarrow$ column $+ 1$ ▷ Move to the next term
  end for
end for
return $\mathbf{P}_{k,2}$

Alg. 2: Algorithm for reading $\mathbf{P}_{k,2}$ from $\mathbf{Q}_1$.

Algorithm findQ2

input: $\mathbf{Q}_1$ — First part of Macaulay matrix of quadratic part of $\mathcal{P}$
$\mathbf{T}$ — A v-by-m matrix

output: $\mathbf{Q}_2$ — The second part of Macaulay matrix for quadratic part of $\mathcal{P}$

$\mathbf{Q}_2 \leftarrow 0_{m\times D_2}$
for k from 1 to m **do**
  $\mathbf{P}_{k,1} \leftarrow$findPk1$(k, \mathbf{Q}_1)$
  $\mathbf{P}_{k,2} \leftarrow$findPk2$(k, \mathbf{Q}_1)$
  $\mathbf{P}_{k,3} \leftarrow -\mathbf{T}^\top \mathbf{P}_{k,1}\mathbf{T} + \mathbf{T}^\top \mathbf{P}_{k,2}$ ▷ Compute $\mathbf{P}_{k,3}$ up to skew-symmetric matrix
  column $\leftarrow 1$
  for i from 1 to m **do** ▷ Read off $\mathbf{Q}_2$
    $\mathbf{Q}_2[k,$ column$] \leftarrow \mathbf{P}_{k,3}[i,i]$
    column $\leftarrow$ column $+ 1$
    for j from $i+1$ to m **do**
      $\mathbf{Q}_2[k,$ column$] \leftarrow \mathbf{P}_{k,3}[i,j] + \mathbf{P}_{k,3}[j,i]$
      column $\leftarrow$ column $+ 1$
    end for
  end for
end for
return $\mathbf{Q}_2$

Alg. 3: Algorithm for determining $\mathbf{Q}_2$ from $\mathbf{Q}_1$ and $\mathbf{T}$.

Algorithm KeyGen

input: private_seed — seed to generate a key-pair

output: (public_seed, $\mathbf{Q}_2$) — A public key
private_seed — A corresponding private key

1: private_sponge ← InitializeAndAbsorb(private_seed)
2: public_seed ← SqueezePublicSeed(private_sponge)
3: $\mathbf{T}$ ← SqueezeT(private_sponge)
4: public_sponge ← InitializeAndAbsorb(public_seed)
5: $\mathbf{C}, \mathbf{L}, \mathbf{Q}_1$ ← SqueezePublicMap (public_sponge)
6: $\mathbf{Q}_2$ ← FindQ2($\mathbf{Q}_1, \mathbf{T}$)
7: **return** (public_seed, $\mathbf{Q}_2$) and private_seed

Alg. 4: The key generation algorithm

a signature in two steps. First, the special structure of $\mathcal{F}$ is exploited to produce a solution $\mathbf{s}'$ to the equation $\mathcal{F}(\mathbf{s}') = \mathbf{h}$. Then, the signature $\mathbf{s}$ is calculated as

$$\mathbf{s} = \begin{pmatrix} \mathbf{1}_v & -\mathbf{T} \\ \mathbf{0} & \mathbf{1}_m \end{pmatrix} \mathbf{s}' .$$

Solving $\mathcal{F}(\mathbf{s}') = \mathbf{h}$ is done by repeatedly substituting pseudo-randomly generated values into the vinegar variables and trying to solve the resulting linear system until a unique solution is found. A unique solution is almost always found on the first try, the probability of failing being roughly 2^{-r}. For a particular assignment to the vinegar variables $\mathbf{v} \in \mathbb{F}_{2^r}^v$, the augmented matrix for the linear system $\mathcal{F}((\mathbf{v}||\mathbf{o})^\top) = \mathbf{h}$ can be derived as in Alg. 5. This algorithm relies on the fact that after fixing the vinegar variables to $\mathbf{v}$, the map $\mathcal{F}$ is a linear map with constant part

$$\mathbf{C} + \mathbf{L} \begin{pmatrix} \mathbf{v} \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{v}^\top \mathbf{P}_{1,1} \mathbf{v} \\ \cdots \\ \mathbf{v}^\top \mathbf{P}_{m,1} \mathbf{v} \end{pmatrix} ,$$

and a linear part with the matrix representation

$$\mathbf{L} \begin{pmatrix} -\mathbf{T} \\ \mathbf{1}_m \end{pmatrix} + \begin{pmatrix} \mathbf{v}^\top [(\mathbf{P}_{1,1} + \mathbf{P}_{1,1}^\top)\mathbf{T} + \mathbf{P}_{1,2}] \\ \cdots \\ \mathbf{v}^\top [(\mathbf{P}_{m,1} + \mathbf{P}_{m,1}^\top)\mathbf{T} + \mathbf{P}_{m,2}] \end{pmatrix} .$$

Pseudocode for the signature generation algorithm is provided in Alg. 6.

2.6 Signature Verification Algorithm

First, the signature verification algorithm uses the public seed to generate $\mathbf{C}, \mathbf{L}$ and $\mathbf{Q}_1$. Together with $\mathbf{Q}_2$, which is included in the public key, this completely determines the public

Algorithm BuildAugmentedMatrix

input: $\mathbf{C} \in \mathbb{F}_{2^r}^m$ — The constant part of the public map $\mathcal{P}$
$\mathbf{L} \in \mathbb{F}_{2^r}^{m \times n}$ — The linear part of $\mathcal{P}$
$\mathbf{Q}_1 \in \mathbb{F}_{2^r}^{m \times \frac{v(v+1)}{2} + vm}$ — The first part of quadratic part of $\mathcal{P}$
$\mathbf{T} \in \mathbb{F}_2^{v \times m}$ — The matrix that determines the linear transformation $\mathcal{T}$.
$\mathbf{h} \in \mathbb{F}_{2^r}^m$ — The hash digest to target.
$\mathbf{v} \in \mathbb{F}_{2^r}^v$ — An assignment to the vinegar variables.

output: $\mathbf{LHS}||\mathbf{RHS} \in \mathbb{F}_{2^r}^{m \times m+1}$ — The augmented matrix for $\mathcal{F}(\mathbf{v}||\mathbf{o}) = \mathbf{h}$

$\mathbf{RHS} \leftarrow \mathbf{h} - \mathbf{C} - \mathbf{L}_s(\mathbf{v}||0)^\top$ ▷ Right hand side of linear system
$\mathbf{LHS} \leftarrow \mathbf{L} \begin{pmatrix} -\mathbf{T} \\ \mathbf{1}_m \end{pmatrix}$ ▷ Left hand side of linear system
for k from 1 to m **do**
  $\mathbf{P}_{k,1} \leftarrow \text{findPk1}(k, \mathbf{Q}_1)$
  $\mathbf{P}_{k,2} \leftarrow \text{findPk2}(k, \mathbf{Q}_1)$
  $\mathbf{RHS}[k] \leftarrow \mathbf{RHS}[k] - \mathbf{v}^\top \mathbf{P}_{k,1} \mathbf{v}$ ▷ evaluation of terms of f_k that are quadratic in vinegar variables
  $\mathbf{F}_{k,2} \leftarrow -(\mathbf{P}_{k,1} + \mathbf{P}_{k,1}^\top)\mathbf{T} + \mathbf{P}_{k,2}$ ▷ Terms of f_k that are bilinear in the vinegar and the oil variables
  $\mathbf{LHS}[k] \leftarrow \mathbf{LHS}[k] + \mathbf{v}\mathbf{F}_{k,2}$ ▷ Insert row in the left hand side
end for
return LHS||RHS

Alg. 5: Builds the augmented matrix for the linear system $\mathcal{P}(\mathbf{v}||\mathbf{o}) = \mathbf{h}$ after fixing the vinegar variables.

Algorithm Sign

input: private_seed — A private key
M — A message to sign
output: $\mathbf{s}$ — A signature for the message M

sponge ← InitializeAndAbsorb(private_seed)
public_seed ← SqueezePublicSeed(sponge)
$\mathbf{T}$ ← SqueezeT(sponge)
public_sponge ← InitializeAndAbsorb(public_seed)
$\mathbf{C}, \mathbf{L}, \mathbf{Q}_1$ ← SqueezePublicMap (public_sponge)
hash_sponge ←InitializeAndAbsorb($M||$0x00)
$\mathbf{h}$ ←SqueezeHashDigest(hash_sponge) ▷ Calculate hash digest
vinegar_sponge ←InitializeAndAbsorb($M||$0x00$||$private_seed) ▷ Sponge for determining vinegar variables
while No solution $\mathbf{s}'$ to the system $\mathcal{F}(\mathbf{s}') = \mathbf{h}$ is found **do**
  $\mathbf{v}$ ← SqueezeVinegar (vinegar_sponge)
  $\mathbf{A}$ ← BuildAugmentedMatrix ($\mathbf{C}, \mathbf{L}, \mathbf{Q}_1, \mathbf{T}, \mathbf{h}, \mathbf{v}$)
    ▷ Build the augmented matrix for the linear system $\mathcal{F}(\mathbf{v}||\mathbf{o}) = \mathbf{h}$
  GaussianElimination($\mathbf{A}$)
  if $\mathcal{F}(\mathbf{v}||\mathbf{o}) = \mathbf{h}$ has a unique solution $\mathbf{o}$ **then**
    $\mathbf{s}' \leftarrow (\mathbf{v}||\mathbf{o})^\top$
  end if
end while
$\mathbf{s} \leftarrow \begin{pmatrix} \mathbf{1}_v & -\mathbf{T} \\ \mathbf{0} & \mathbf{1}_m \end{pmatrix} \mathbf{s}'$
return $\mathbf{s}$

Alg. 6: The signature generation algorithm

map $\mathcal{P}$. To verify a signature $\mathbf{s}$ for a message M, the verification algorithm simply checks whether $\mathcal{P}(\mathbf{s})$ is equal to the rm-bit long hash digest of the message M, appended with `0x00`. Pseudocode for this algorithm is provided in Alg. 9.

Algorithm EvaluatePublicMap

input: $(\mathsf{public\_seed}, \mathbf{Q}_2)$ — A public key
$\mathbf{s}$ — A candidate–signature
output: The evaluation of $\mathcal{P}$ at $\mathbf{s}$

```
1: sponge ← InitializeAndAbsorb(public_seed)
2: C, L, Q1 ← SqueezePublicMap (sponge)
3: Q ← Q1||Q2
4: e ← Cs + Ls                          ▷ Evaluate constant and linear part of P at s
5: column ← 1
6: for i from 1 to n do                 ▷ Evaluate quadratic parts of P at s
7:     for j from i to n do
8:         for k from 1 to m do
9:             e[k] ← e[k] + Q[k, column]s[i]s[j]      ▷ Evaluate terms in x_i x_j
10:        end for
11:        column ← column + 1
12:    end for
13: end for
14: return e
```

Alg. 7: The algorithm for evaluating the public map at a point

2.7 Signatures with message recovery

It is possible to use the signature scheme in a message recovery mode. Whether or not message recovery is used does not affect the signature generation algorithm. The same key pair can be used to sign messages in message recovery mode and in appended signature mode, a signature for M in appended signature mode is unrelated to a signature for the same message in message recovery mode, because a different byte is appended to the message in each mode. The signing algorithm in message recovery mode differs from the signing algorithm in appended signature mode (Alg. 6) because the message is padded with `0x01` instead of `0x00` in lines 6 and 8. Furthermore, the procedure to determine the target of the public map is altered to make message recovery possible. In the appended signature mode, the target was determined by interpreting the $\frac{r}{8}m$ byte long output of a SHAKE function as a vector of m elements of $\mathbb{F}_{2^r}$. In message recovery mode, the target is obtained by interpreting

$$\text{SHAKE}(M||\texttt{0x01}, l_1)||\text{SHAKE}(\text{SHAKE}(M||\texttt{0x01}, l_1), l_2) \oplus M'$$

as a vector of m elements in $\mathbb{F}_{2^r}$, where l_1 is equal to 256 if SHAKE128 is used, or equal to 512 if SHAKE265 is used, and l_2 is equal to $\frac{r}{8}m - l_1$, and M' is formed by taking the last

Algorithm Verify

input: $(\mathsf{public\_seed}, \mathbf{Q}_2)$ — A public key
M — A message
$\mathbf{s}$ — A candidate–signature

output: **Accept** if $\mathbf{s}$ is a valid signature for M, **Reject** otherwise

1: $\mathsf{sponge} \leftarrow$InitializeAndAbsorb($M||\mathtt{0x00}$)
2: $\mathbf{h} \leftarrow$SqueezeHashDigest(sponge)
3: $\mathbf{e} \leftarrow$EvaluatePublicMap($(\mathsf{public\_seed}, \mathbf{Q}_2), \mathbf{s}$)
4: **if** $\mathbf{e} = \mathbf{h}$ **then** ▷ Check if $\mathcal{P}(\mathbf{s}) = \mathbf{h}$
5: **return Accept**
6: **else**
7: **return Reject**
8: **end if**

Alg. 8: The signature verification algorithm in appended signature mode

$l_2 - 1$ bytes of the message M , appending the byte `0x01` from the right, and padding with zeros in the case that the message M is shorter that $l_2 - 1$ bytes.

The signature verification algorithm evaluates the public map $\mathcal{P}$ at the signature $\mathbf{s}$, and interprets the output as a sequence $\mathsf{first\_bytes}$ of l_1 bytes, concatenated with a sequence $\mathsf{last\_bytes}$ of l_2 bytes. The signature verification algorithm recovers up to $l_2 - 1$ bytes of the message M, by calculating

$$M' = \mathsf{last\_bytes} \oplus \mathrm{SHAKE}(\mathsf{first\_bytes}, l_2)$$

and removing the padding. If the computed value of M' does not end in a `0x01`, followed by a (possibly empty) sequence of `0x00`s, the signature is rejected. Otherwise, the signature is accepted if t_1 is equal to $\mathrm{SHAKE}(M||\mathtt{0x01}, l_1)$.

2.8 Encoding of objects

2.8.1 Encoding of finite field elements

The finite fields that are used by the various instantiations of the LUOV signature scheme are $\mathbb{F}_{2^8}, \mathbb{F}_{2^{16}}, \mathbb{F}_{2^{48}}, \mathbb{F}_{2^{64}}$ and $\mathbb{F}_{2^{80}}$.

Field of size 2^8. Field elements in the field $\mathbb{F}_{2^8}$ are represented as binary polynomials modulo the irreducible polynomial $f_8 = x^8 + x^4 + x^3 + x + 1$. This choice is arbitrary and does not affect the security of the scheme. An element of $\mathbb{F}_2[x]/(f_8)$ is encoded as the byte obtained by concatenating its coefficients, where the least significant bits correspond to the lowest degree terms.

Algorithm Verify

input: $(\mathsf{public\_seed}, \mathbf{Q}_2)$ — A public key
M — The first part of a message (possibly the empty string)
$\mathbf{s}$ — A candidate–signature

output: The full message M if $\mathbf{s}$ is a valid signature, **Reject** otherwise

$\mathbf{e} \leftarrow$EvaluatePublicMap$((\mathsf{public_seed}, \mathbf{Q}_2), \mathbf{s})$
$\mathsf{first_bytes}, \mathsf{last_bytes} \leftarrow$Enc$(\mathbf{e})$ ▷ Split $\mathbf{e}$ into l_1 and l_2 bytes
$\mathsf{padded_message} \leftarrow \mathsf{last_bytes} \oplus$SHAKE$(\mathsf{first_bytes}, l_2)$
if $\mathsf{padded\_message}$ is **not** properly padded **then**
  return Reject ▷ Reject if $\mathsf{padded\_message}$ doesn't end in $\mathtt{0x01}\,\mathtt{0x00}\cdots\mathtt{0x00}$
end if
$M \leftarrow M||$RemovePadding$(\mathsf{padded_message})$
$\mathsf{hash_digest} \leftarrow$SHAKE$(M||\mathtt{0x01}, l_1)$
if $\mathsf{first\_bytes} = \mathsf{hash\_digest}$ **then**
  return M
else
  return Reject
end if

Alg. 9: The signature verification algorithm in message recovery mode

Example.

$$\begin{aligned}\mathsf{Enc}(1) &= \mathtt{0x01}\\ \mathsf{Enc}(x^6) &= \mathtt{0x40}\\ \mathsf{Enc}(x + x^5 + x^7) &= \mathtt{0xa2}\end{aligned}$$

Field of size 2^{16}. Field elements in the field $\mathbb{F}_{2^{16}}$ are represented as binary polynomials modulo the irreducible polynomial $f_{16} = x^{16} + x^{12} + x^3 + x + 1$. This choice is arbitrary and does not affect the security of the scheme. An element of $\mathbb{F}_2[x]/(f_{16})$ is encoded as the two bytes obtained by concatenating its coefficients. The first byte represents the terms of degree 0 op to 7, the second byte represents the terms of degree 8 up to 15.

Example.

$$\begin{aligned}\mathsf{Enc}(1) &= \mathtt{0x01}\,\mathtt{0x00}\\ \mathsf{Enc}(x^8 + x^9) &= \mathtt{0x00}\,\mathtt{0x03}\\ \mathsf{Enc}(x + x^5 + x^7 + x^{15}) &= \mathtt{0xa2}\,\mathtt{0x80}\end{aligned}$$

Larger fields. The larger fields used by the scheme are seen as simple field extensions of $\mathbb{F}_{2^{16}}$. The irreducible polynomials of these field extensions are given in Table 1. If F is

Table 1: Irreducible polynomials used for representing finite fields.

Finite Field	Irreducible polynomial in $\mathbb{F}_2[X,x]/(f_{16})$
$\mathbb{F}_{2^{48}}$	X^3+X+1
$\mathbb{F}_{2^{64}}$	X^4+X^2+xX+1
$\mathbb{F}_{2^{80}}$	X^5+X^2+1

such an irreducible polynomial of degree d, an element of $\mathbb{F}_2[X,x]/(f_{16},F)$ is encoded by the $2d$ bytes obtained by concatenating the encodings of its coefficients in order of increasing degrees, i.e.

$$\mathsf{Enc}(c_0+c_1X+\cdots+c_{d-1}X^{d-1}) = \mathsf{Enc}(c_0)\cdots\mathsf{Enc}(c_{d-1})$$

2.8.2 Encoding of private key

A private key for the LUOV signature scheme is a sequence of 256 random bits (used to seed a Keccak1600 Sponge) and is simply encoded as a sequence of 32 bytes.

2.8.3 Encoding of public key

A public key of the LUOV signature scheme consists of a sequence of 32 bytes (which are used to seed a Keccak Sponge) and an m-by-$m(m+1)/2$ matrix with binary entries. The matrix is encoded by concatenating the columns and padding the result with zero bits to get a sequence of bits of length divisible by 8. Then, the sequence is interpreted as a sequence of bytes, where the first bits have the least significant values. The encoding of a public keys is $32+\lceil\frac{m^2(m+1)}{2}\frac{1}{8}\rceil$ bytes large.

Example. *For a parameter set with $m=3$, the public key could contain the matrix*

$$\mathbf{Q}_2 = \begin{pmatrix} 010111 \\ 111001 \\ 000101 \end{pmatrix}.$$

Concatenating its columns gives 010110010101100111, which results in the 3 bytes (01011001) (01011001) (11000000), so

$$\mathsf{Enc}(\texttt{0x36}\cdots\texttt{0x5d},\mathbf{Q}_2) = \underbrace{\texttt{0x36}\cdots\texttt{0x5d}}_{\textit{32-byte Public seed}}\ \underbrace{\texttt{0x9a 0x9a 0x03}}_{T}.$$

2.8.4 Encoding of signature

A signature of the UOV signature scheme consists of a vector $\mathbf{s}\in\mathbb{F}_{2^r}^n$ of $n=v+m$ field elements. The encoding of the signature consists of the concatenation of the encodings of

these n field elements. The encoding of a signature is $\frac{nr}{8}$ bytes large. (r is always divisible by 8)

$$\mathsf{Enc}(\mathbf{s}) = \mathsf{Enc}(\mathbf{s}[0])\mathsf{Enc}(\mathbf{s}[1]) \cdots \mathsf{Enc}(\mathbf{s}[n-1])$$

2.9 Sampling objects with the SHAKE function

The LUOV signature scheme uses the SHAKE extendable-output functions to provide cryptographically secure pseudorandom bit-streams. First, a seed is fed into the Keccak1600 sponge construction. Then output bytes are squeezed from the sponge and interpreted as some mathematical object. This approach is used to generate the following objects:

- $\mathsf{public\_seed}$ — The public seed used to generate a large part of the public map $\mathcal{P}$.
- $\mathbf{T}$ — The matrix that determines the linear transformation that hides the UOV structure of the secret map $\mathcal{F}$.
- $\mathbf{h}$ — The hash digest of a message.
- $\mathbf{v}$ — An assignment to the vinegar variables.
- $\mathbf{C}, \mathbf{L}, \mathbf{Q}_1$ — A large part of the public map $\mathcal{P}$.

Before sampling objects from a Keccak sponge, the sponge has to be initialized to the all-zero state and used to absorb a seed. In our pseudocode description of the LUOV algorithm we refer to this operation as InitializeAndAbsorb, which receives a sequence of bytes as input, and outputs a Keccak sponge object that was initialized and has absorbed the input sequence. The sponge can then provide an arbitrarily long sequence of pseudorandom bytes with the Squeeze operation, which takes a sponge object and an integer b as input, outputs a sequence of b bytes and updates the state of the sponge, such that it can be used to squeeze more bytes if needed.

2.9.1 Squeezing public seed

A public seed, represented by 32 bytes, is simply obtained from a sponge by squeezing out the 32 bytes. This operation is called SqueezePublicSeed.

2.9.2 Squeezing T

The matrix $\mathbf{T} \in \mathbb{F}_2^{v \times m}$ is squeezed out of a sponge by squeezing $\lceil \frac{m}{8} \rceil v$ bytes from the sponge, and interpreting the bytes $(i-1)\lceil \frac{m}{8} \rceil + 1$ up to $i\lceil \frac{m}{8} \rceil$ as the i-th row of $\mathbf{T}$. If m is not divisible by 8, the most significant bits of the last byte (i.e. $i\lceil \frac{m}{8} \rceil$-th byte in the sequence) are ignored. This operation is referred to as SqueezeT.

Example. *Suppose $m = 3$, $v = 4$ and the following 4 bytes are squeezed from the Keccak sponge :*

```
0x49 0xa2 0x86 0x4d .
```

Then, the matrix $\mathbf{T} \in \mathbf{F}_2^{v \times m}$ is equal to

$$\begin{pmatrix} 001 \\ 010 \\ 110 \\ 101 \end{pmatrix} .$$

2.9.3 Squeezing hash digest and vinegar variables

The hash digest and the assignment to the vinegar variables are vectors over $\mathbb{F}_{2^r}$ of length $n = m + v$ and length v respectively. They are obtained by squeezing $n\frac{r}{8}$ and $v\frac{r}{8}$ bytes from the sponge and interpreting these as the encoding of n, respectively v elements of $\mathbb{F}_{2^r}$. These operations are referred to as SqueezeHashDigest and SqueezeVinegar.

2.9.4 Squeezing most part of the public map

The matrices $\mathbf{C} \in \mathbb{F}_2^{m \times 1}, \mathbf{L} \in \mathbb{F}_2^{m \times n}$ and $\mathbf{Q}_1 \in \mathbb{F}_2^{m \times (\frac{o(o+1)}{2} + mo)}$ are squeezed column by column from the Keccak sponge. Each column is obtained by squeezing $\lceil \frac{m}{8} \rceil$ bytes from the sponge, and interpreting these as m-bit long columns, ignoring the most significant bits of the last byte in the case that m is not divisible by 8. The process of sampling columns of coefficients of $\mathcal{P}$ is identical to the process of sampling rows of $\mathbf{T}$.

In total, $1+n+\frac{o(o+1)}{2}+mo$ columns are sampled from the sponge. The first column represents $\mathbf{C}$, the next n columns represent $\mathbf{L}$, and the remaining $\frac{o(o+1)}{2}+mo$ columns represent $\mathbf{Q}_1$. The entire operation is called SqueezePublicMap, it takes a sponge object as input and returns the matrices $\mathbf{C}, \mathbf{L}$ and $\mathbf{Q}_1$.

3 List of parameter sets (part of 2.B.1)

We define two sets of parameter choices. The first set aims to provide small signatures, which is suitable for applications where many signatures are communicated. The second set of parameter choices aims to minimize the combined cost of a signature and a public key and is more suitable when the signatures and the public key are both communicated, such as a chain of signatures anchored to a root certificate authority.

Table 2: Different parameter choices for the LUOV signature scheme. The first 3 choices provide small signatures, the last three choices give small public keys at the cost of larger signatures.

	claimed security level	r	m	v	SHAKE	|sig|	|pk|	|sk|	message recovery (optional)
LUOV-8-63-256	lvl 2	8	63	256	128	319 B	15.5 KB	32B	30 B
LUOV-8-90-351	lvl 4	8	90	351	256	441 B	45.0 KB	32B	25 B
LUOV-8-117-404	lvl 5	8	117	404	256	521 B	98.6 KB	32B	52 B
LUOV-48-49-242	lvl 2	48	49	242	128	1.7 KB	7.3 KB	32B	261 B
LUOV-64-68-330	lvl 4	64	68	330	256	3.1 KB	19.5 KB	32B	479 B
LUOV-80-86-399	lvl 5	80	86	399	256	4.7 KB	39.3 KB	32B	795 B

4 Detailed performance analysis (2.B.2)

4.1 Description of platform

The following measurements were collected using `supercop-20171020` running on a computer named `bas`. The CPU on `bas` is an Intel® Core™ i5-7500T running at 3.3 GHz. `bas` has 7.5GB of RAM and runs CentOS Linux release 7.4.1708. Benchmarks used `crypto_sign`, which ran on one core of the CPU. The gcc version 4.8.5 20150623 (Red Hat 4.8.5-16) was used.

4.2 Time

The median number of cycles consumed by the different algorithms are reported in Table 3. The measurements are made in appended signature mode, but there is no noticeable difference between the cycle count in appended signature mode and in message recovery mode. A more optimized implementation that uses vectorization instructions is likely to reduce the cycle counts significantly.

4.3 Space

For all parameter choices, the secret key consists of a **32-byte** seed.

The public key consists of a 4 byte seed, and the remaining $\frac{m^2(m+1)}{2}$ coefficients of the public map $\mathcal{P}$. This makes a total of $\mathbf{4} + \lceil \frac{\mathbf{m^2(m+1)}}{\mathbf{16}} \rceil$ **bytes**. If message recovery is used, the messages can be shortened by roughly 15% of the signature size.

Table 3: Median cycle counts of optimized implementation. Measured with `supercop20171020`. The SUPERCOP output files with the compiler flags that were used and the exact cycle counts for various message sizes are included in the Supporting_Documentation folder.

	claimed security level	Key generation (million cycles)	Signing (million cycles)	Verification (million cycles)
LUOV-8-63-256	lvl 2	21.0	5.87	4.93
LUOV-8-90-351	lvl 4	81.8	21.6	17.3
LUOV-8-117-404	lvl 5	146	36.5	29.7
LUOV-48-49-242	lvl 2	14.8	34.1	23.6
LUOV-64-68-330	lvl 4	50.8	111	66.1
LUOV-80-86-399	lvl 5	96.8	216	124

A signature consists of $v + m$ elements of the field $\mathbb{F}_{2^r}$, good for a total of $\frac{\mathbf{r(v+m)}}{\mathbf{8}}$ **bytes**.

The concrete sizes for the proposed parameter choices are displayed in Table 2.

When implemented properly, the signing and verification algorithms require very little RAM memory. The RAM usage of the signing algorithm is dominated by storing the augmented matrix for the linear system after fixing the vinegar variables. This requires storing $m(m+1)$ elements of $\mathbb{F}_{2^r}$. For the LUOV-8-63-256 parameter set this is 4032 bytes. Besides storing the public key and a signature, the memory requirements of the verification algorithm is dominated by the state of the Keccak sponge (i.e. 200 bytes), or storing the evaluation of the public map $\mathcal{P}$, (i.e. $rm/8$ bytes).

4.4 How parameters affect performance

Table 3 shows that key generation is faster for the parameter sets with large extension fields. This is so because key generation benefits from the smaller polynomial systems, without paying the price of more complex field arithmetic, since key generation works in $\mathbb{F}_2$.

In contrast, in our implementation of the signing and verification algorithms, the smaller size of the polynomial systems does not make up for the increased complexity of the field arithmetic. Therefore, signing and verification is faster for the parameter sets with smaller field extensions.

The size of the public key is only impacted by the parameter m, and scales as $O(m^3)$, therefore to keep the public key small m should not be too large. By increasing r, the degree of the field extension $\mathbb{F}_2 \subset \mathbb{F}_{2^r}$, the required value of m to achieve a fixed security level decreases. However, increasing r also increases the size of the signatures. Therefore, it is possible to make a trade-off between small public keys (i.e. large r) or small signatures (i.e. small r). We propose two sets of parameter choices, one aiming at small signatures, the other aiming at small public keys. By varying the parameter r it is possible to interpolate

between these parameter sets.

Example. *One might want a signature scheme that attains security level 2 with signatures as small as possible, subject to the condition that the public key is smaller than* $10KB$*. The best option from the proposed parameter sets would be LUOV-48-49-242, having signatures of* $1.7KB$ *and public keys of* $7.3KB$*. We can do better by adjusting the parameter* r*. For the choice* $r = 28$*, the python script that is included in the submission proposes the parameters* $m = 54, v = 247$*, resulting in signatures of* $1.0KB$ *and public keys of just under* $10KB$*.*

4.5 Optimizations

4.5.1 Bit slicing

The i-th row of $\mathbf{Q}_2$ is calculated using only the data $\mathbf{T}$ and the i-th row of $\mathbf{Q}_1$ and this calculation is exacly the same for each row. This is an ideal situation for using bit slicing. The bits in the columns of $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are packed into words and the computation is performed for all rows simultaneously. This greatly speeds up the key generation algorithm. This optimization is included in the reference implementation, because it does not affect the legibility of the code.

4.5.2 Precomputing $\mathcal{P}$ and $\mathcal{F}$

With each verification of a signature a lot of coefficients of the public map $\mathcal{P}$ have to be generated with the SHAKE function. According to the gprof profiler, this computation is responsible for roughly 75% of the cycle usage of the verification algorithm in our optimized implementation of the first parameter set. If enough memory is available (e.g. roughly 380 KB for the first parameter set) the coefficients of $\mathcal{P}$ can be precomputed and stored to speed up the verification of signatures. Similarly, the coefficients of the secret map $\mathcal{F}$ can be precomputed to speed up the signing algorithm. This optimization was not used in the reference or optimized implementation.

5 Expected strength (2.B.4)

The LUOV signature system is designed for EUF-CMA security. The parameters of the LUOV scheme are chosen such that lower bounds to the bit complexity of all the known attacks exceed the required complexity level by a margin of 10 percent to account for possible future improvements in the attacks. The process of choosing the parameters is implemented in a python script which is included in the submission package. The designer specifies the desired security level and chooses the size of the field extension, then the script determines the parameters m and v to reach the required security level. Larger field extensions lead to smaller public keys at the cost of larger signatures. Table 4 summarizes the lower bounds

to the complexity of the various attacks. An overview of the known attacks and what the lower bounds to their complexities are is given in section 6.

To reach security level 2 i.e. "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 256-bit hash function (e.g. SHA256/ SHA3-256)" we assure that all known attacks (except hash collision attacks) require at least 2^{160} operations. This number was determined by considering the estimated number of gates required to find a hash collision in SHA3-256 (i.e. 2^{146}), and increasing the exponent by a margin of 10 percent to allow for future improvements of the attacks. Similarly, to reach security level 4, we require that all known attacks require at least $2^{231} = 2^{210 \times 1.1}$ operations.

To reach security level 5, i.e. "any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256)" we require that all classical attacks require at least 2^{299} operations, and all quantum attacks require at least 2^{257} operations. These numbers are obtained by considering the estimated number of classical gates (i.e. 2^{272}) or quantum gates (i.e. 2^{234}) and increasing their exponent by 10 percent to allow for future improvements of the attacks. In all attack scenarios the depth of a quantum computation is assumed to be bounded by 2^{64} quantum gates.

Table 4: Summary of attacks against our parameters. The table reports $\log_2$ of a lower bound to the number of operations required for each attack. Quantum computations are bounded to a depth of 2^{64} field operations.

		Direct forgery		UOV attack		Reconciliation attack	
(r, m, v)	security	optimal k	complexity	classical	quantum	classical	quantum
$(8, 63, 256)$	lvl 2	2	161	225	161	192	192
$(8, 90, 351)$	lvl 4	3	231	295	231	263	287
$(8, 117, 404)$	lvl 5	4	300	322	258	303	340
$(48, 49, 242)$	lvl 2	1	165	224	160	181	178
$(64, 68, 330)$	lvl 4	1	235	295	231	247	266
$(80, 86, 399)$	lvl 5	1	300	347	283	299	335

6 Analysis of known attacks (2.B.5)

The signature scheme is an adaptation of Oil and Vinegar [15] scheme that was proposed by Patarin in 1997. The Oil and Vinegar scheme is one of the best studied multivariate signature schemes which has, with the right parameter choices, withstood all cryptanalysis since 1997.

All the adaptations that LOUV makes to the Unbalanced Oil and Vinegar scheme (see Sect. 2.2) can be shown not to impact the security of the scheme (assuming the output of the Keccak1600 sponge construction is indistinguishable from random bits), an exception being the adaptation of lifting a public key of UOV over $\mathbb{F}_2$ to a large extension field. It requires some argument to show that a direct signature forgery against the modified scheme is as difficult as a direct signature forgery against UOV over the extension field. However, since the key generation algorithm is not changed by this adaptation, it is clear that a key recovery attack against LUOV is equivalent to a key recovery attack against UOV over $\mathbb{F}_2$.

We now give an overview of known attacks. The overview is based on the overview given in [5]; We have adapted the example to match one of the proposed parameter sets.

6.1 Direct attack

This attack tries to forge a signature for a certain message M by trying to find a solution $\mathbf{s} \in \mathbb{F}_{2^r}^n$ for the system $\mathcal{F}(\mathbf{s}) = \mathcal{H}(M)$. This is an instance of the MQ (Multivariate Quadratic) problem.

MQ Problem. Given a quadratic polynomial map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ over a finite field $\mathbb{F}_q$, find $\mathbf{x} \in \mathbb{F}_q^n$ that satisfies $\mathcal{P}(\mathbf{x}) = \mathbf{0}$.

Thomae and Wolf showed that finding a solution for an underdetermined system with $n = \alpha m$ can be reduced to finding a solution of a determined system with only $m + 1 - \lfloor \alpha \rfloor$ equations [17]. This means that as a system becomes more underdetermined it becomes easier to solve.

For all but very small values of q, (e.g. $q = 2, q = 3$), the best known classical algorithms to solve the MQ-problem for generic determined systems over finite fields use a hybrid approach [3, 4] that combines exhaustive search with Gröbner basis computations. In this approach k variables are fixed to random values and the remaining $n-k$ variables are found with a Gröbner basis algorithm such as F_4, F_5 or XL. If no assignment to the remaining $n-k$ variables exists that solves the system, the procedure starts again with a different guess for the first k variables. We require on average roughly q^k Gröbner basis computations until a solution is found. As a result, the optimal value of k decreases as q increases. The complexity of computing a Gröbner basis for a system of polynomials depends critically on the degree of regularity (d_{reg}) of that system. We refer to Bardet [1] for a precise definition of the degree of regularity.

The most costly part of the F_5 algorithm is doing Gaussian elimination on a large matrix with roughly $\binom{n+d_{reg}}{d_{reg}}$ rows and columns. The complexity of the F_5 algorithm is thus given by

$$C_{F_5}(n, d_{reg}) = O\left(\binom{n + d_{reg}}{d_{reg}}^{\omega}\right),$$

where $2 \leq \omega < 3$ is the constant in the complexity of doing Gaussian reduction on the matrices constructed in the Gröbner basis computation. These matrices are structured and sparse, which can be exploited to make Gaussian elimination more efficient [9]. The complexity of the hybrid approach is

$$C_{\text{Hybrid}F_5(n,d_{reg},k)} = O\left(q^k \binom{n - k + d_{reg}(k)}{d_{reg}(k)}^{\omega}\right), \tag{1}$$

where $d_{reg}(k)$ stand for the degree of regularity of the system after fixing the values of k variables.

Determining the degree of regularity for a specific polynomial system is difficult, but for a certain class of systems, called semi-regular systems, it is known that the degree of regularity can be deduced from the number m of equations and the number n of variables [1, 8]. In particular, for quadratic semi-regular systems the degree of regularity is the degree of the first term in the power series of

$$S_{m,n}(x) = \frac{(1 - x^2)^m}{(1 - x)^n}$$

that has a non-positive coefficient. This gives a practical method to calculate the degree of regularity of any semi-regular system. Empirically, polynomial systems that are randomly chosen have a very large probability of being semi-regular and it is conjectured that most systems are semi-regular systems. For the definition and the theory of semi-regular systems we refer to chapter 3 of the PhD thesis of Bardet [1].

In a direct attack against the LUOV scheme all the coefficients of the system that needs to be solved lie in $\mathbb{F}_2$, except those of the constant terms, because those coefficients come from the message digest. We claim that this property does not significantly reduce the hardness of finding solutions relative to the case where the coefficients are generic elements of $\mathbb{F}_{2^r}$. By definition [1], the degree of regularity of a polynomial system does only depend on its quadratic part, and it is apparent that lifting a polynomial system to a field extension does not affect its degree of regularity. Therefore, the degree of regularity of a LUOV public key follows the same distribution of a UOV public key over the field $\mathbb{F}_2$, even after fixing a number of variables. It has been observed by Faugère and Perret [10] that polynomial systems that result from fixing $\approx v$ variables in a UOV system behave like semi-regular systems, whose degree of regularity does not depend on q. Therefore, the degree of regularity of a LUOV public polynomial system is distributed identically to that of a UOV public polynomial system, independently of the size q of the finite field that is used.

Since the degree of regularity, in combination with the number of variables, determines the complexity of a Gröbner basis computation (measured in number of field operations), a Gröbner basis computation on the LUOV polynomial system is not significantly more efficient than a Gröbner basis computation against regular UOV with the same parameters. This argument is confirmed by the experimental data in Table 5. There we see that a direct attack is slightly faster against the modified scheme than against the original UOV scheme, but only by a small constant factor. Even though the Gröbner basis is computed over $\mathbb{F}_{2^r}$, the largest part of the arithmetic only involves the field elements 0 and 1, so the arithmetic is faster than with generic elements of $\mathbb{F}_{2^r}$. This is where the difference observed in Table 5 comes from. If we do the same experiment with a smaller extension field such as $\mathbb{F}_{2^8}$ there is no observed difference between the running time of a direct attack against a regular UOV scheme and our modified scheme.

Remark. *In a direct attack one fixes $\approx v$ variables randomly to make the system a slightly overdetermined system. In our experiments we have fixed these variables to values in $\mathbb{F}_2$ to make sure that we do not introduce linear terms with coefficients in $\mathbb{F}_{2^r}$ instead of $\mathbb{F}_2$ in the case of the modified UOV scheme.*

Table 5: Running time of a direct attack against the regular UOV scheme over $\mathbb{F}_{2^{64}}$ and the modified UOV scheme, with the MAGMA v2.22-10 implementation of the F4 algorithm. We did not implement the method of Thomae and Wolf [17].

(m, v)	Regular UOV (s)	Lifted UOV (s)	difference
(7,35)	0.43	0.21	-52%
(8,40)	1.56	0.76	-51%
(9,45)	7.00	3.21	-54%
(10,50)	33.50	17.44	-48%
(11,55)	132.88	76.60	-42%
(12,60)	828.31	588.33	-29%

To obtain a lower bound to the complexity of a Gröbner basis computation we assume that the parameter ω in the complexity of Gaussian elimination on the matrices constructed

in the Gröbner basis algorithm is equal to 2 and that the constant factor hidden by the big O notation is equal to 1. That is, in Eqn. (1) we put $\omega = 2$ and we drop the big O notation to get a concrete lower bound to the number of bit operations of a hybrid attack. Even though this is a generous lower bound, we require that this lower bound exceeds the required bit complexity by 10 percent when choosing parameters. This is done to allow for future improvements in algorithms that find solutions to polynomial equations.

Example. *We will estimate the complexity of a direct attack against LUOV with the parameter set* $(r = 8, m = 63, v = 256)$*; this set is proposed as a set that achieves security level 2. Using the method of Thomae and Wolf. we can reduce finding a solution to this underdetermined system to finding a solution of a determined system with* $63+1-\lfloor(63+256)/63\rfloor = 59$ *equations. We assume this system, and the systems that are derived by fixing a number of variables, to be semi-regular. If we fix* k *extra variables the degree of regularity is equal to the degree of the first term in the power series of*

$$S_{59,59-k}(x) = \frac{(1-x^2)^{59}}{(1-x)^{59-k}}$$

which has a non-positive coefficient. For $k = 0$ *we have* $S_{59,59}(x) = (1+x)^{59}$*, so the degree of regularity is* 60*. For* $k = 1$ *we have*

$$S_{59,58}(x) = 1 + 58x + 1652x^3 + \cdots + 3814986502092304x^{29} + 0x^{30} + O(x^{31})\,,$$

where all the omitted terms have positive coefficients, so the degree of regularity is 30*. We can now use (1) to obtain a lower bound to the complexity of the hybrid approach. For* k *equal to* 0 *and* 1 *this is equal to*

$$\binom{59+60}{60}^2 \approx 2^{230.4} \qquad \textit{and} \qquad 2^8\binom{59-1+30}{30}^2 \approx 2^{164.0}$$

respectively. Repeating this calculation for higher values of k *we eventually see that the optimal value of* k *is 2, the corresponding degree of regularity is* 27 *and the complexity of the direct attack is estimated as* $2^{161.3}$*. Thus, this lower bound exceeds* $2^{146\times 1.1}$*, as required.*

In theory, a quantum attacker could use Grover search instead of the brute force part of the hybrid approach to speed up a direct attack. The complexity of this attack would be

$$C_{\mathrm{Hybrid}F_5(n,d_{reg},k)} = O\left(q^{k/2}\binom{n-k+d_{reg}(k)}{d_{reg}(k)}^{\omega}\right)\,, \tag{2}$$

where the only difference with (1) is that the factor q^k is replaced by $q^{k/2}$. However, this attack is not possible if the depth of a quantum computation is limited to, say, 2^{64} operations. For all our parameter choices and all practical values of k, the complexity of even a single Gröbner basis computation is beyond 2^{64}, and the Grover algorithm should do a large number of these computations sequentially in order to enjoy a noticeable speedup over the classical brute force search.

6.2 Key recovery attacks.

Since the key pair generation algorithm used by the LUOV scheme is identical to that of the original UOV scheme over the field $\mathbb{F}_2$ it is clear that a key recovery attack against the Lifted UOV scheme is equivalent to a key recovery attack against a regular UOV scheme over $\mathbb{F}_2$. Key recovery attacks against UOV have been investigated ever since the invention of the Oil and Vinegar scheme in 1997 [15], so it is well understood which attacks are possible and what the complexities of these attacks are. It is also clear that we can make key recovery attacks harder by increasing the number of vinegar variables.

6.2.1 UOV attack

Patarin [15] suggested in the original version of the Oil and Vinegar scheme to choose the same number of vinegar and oil variables, or $v = m$. This choice was cryptanalyzed by Kipnis and Shamir [14]: they showed that an attacker can find the inverse image of the oil variables under the map $\mathcal{T}$. This is enough information to find an equivalent secret key, so this breaks the scheme. This approach generalizes for the case $v > m$; the complexity then increases to $O(q^{v-m}n^4)$ [13] and is thus exponential in $v - m$. Since a UOV attack on the Lifted UOV scheme is equivalent to a UOV attack over $\mathbb{F}_2$, we have that the complexity of a UOV attack against the Lifted UOV scheme is approximately $2^{v-m-1} \cdot n^4$ binary operations.

The generalized UOV attack chooses a random linear combination of the matrices that represent the quadratic parts of the polynomials in the public system and computes the minimal eigenspaces of the matrix. With probability 2^{m-v+1} this computation yields a vector in the oil subspace. This means that a quantum attacker can use the Grover search algorithm [11] to look for a random linear combination that will yield a vector in the oil subspace. Ignoring issues of 'Groverizing' the algorithm such as making the computation reversible and the probabilistic nature of the eigenspace computation, the complexity of a quantum attack becomes $2^{\frac{v-m-1}{2}}n^4$. If we limit the depth of a quantum computation to 2^{depth}, and we ignore the depth of the eigenspace-finding subroutine, the complexity of an attack is at least $\max(2^{\frac{v-m-1}{2}}n^4, 2^{v-m-1}n^4/2^{depth})$.

6.2.2 Reconciliation attack

The reconciliation attack against the lifted UOV scheme is equivalent to the UOV reconciliation attack against UOV over the field $\mathbb{F}_2$. A lower bound on the complexity of this attack is given by the complexity of solving a quadratic system of v variables and v equations over $\mathbb{F}_2$, but the problem is expected to be harder [5]. There exists specialized algorithms for solving polynomial systems over $\mathbb{F}_2$ that are more efficient than the generic hybrid approach. One method is a smart exhaustive search, which requires approximately $log_2(n)2^{n+2}$ bit operations [6]. The BooleanSolve algorithm [2] combines an exhaustive search with sparse linear algebra to achieve a complexity of $O(2^{0.792n})$. However the method only becomes faster than

the exhaustive search method when $n > 200$. Recently, Joux and Vitse proposed a new algorithm that was able to solve a Boolean system of 146 quadratic equations in 73 variables in one day [12]. The algorithm beats the exhaustive search algorithm, even for small systems. The complexity of this algorithm is still under investigation, but a rough estimate based on the reported experiments suggests that the number of operations scales like $2^{\alpha n}$ with α between 0.8 and 0.85 and with a constant factor between 2^7 and 2^{10}. For choosing the parameters of the LUOV signature scheme, we have assumed that finding a solution to a determined system of n quadratic Boolean equations requires $2^{0.75n}$ operations in $\mathbb{F}_2$, even though this is likely to seriously overestimate the capabilities of the state of the art algorithms.

Due to the limit on the circuit depth of quantum computations, the Gröbner based methods of solving a Boolean system cannot be 'Groverised'. In contrast, quantum attackers can still use a brute force Grover search to solve systems over $\mathbb{F}_2$ with $2^{n/2}$ sequential evaluations of the polynomials in the system. However, if the depth of a quantum computation is restricted to 2^{depth} evaluations of the polynomials, the required number of polynomial evaluations in a Grover search is at least $\max(2^{n-depth}, 2^{n/2})$. Asymptotically this is worse than the classical Gröbner basis based methods, which is why the reported hardness of a quantum reconciliation attack in Table 6 is higher than the hardness of the classical reconciliation attack. One would expect quantum attacks to be at least as efficient as classical attacks, because a quantum computer can simulate a classical computer. In our analysis this is not the case, because the depth of a quantum computation is assumed to be limited, which is not the case for a classical computation.

6.3 Hash collision attack

As is the case for all hash-and-sign digital signature algorithms, a hash collision can be exploited to break the EUF-CMA security definition. The SHAKE extendable output functions are used to generate a hash digest of the required length. The parameter sets claiming a security level 2 use SHAKE-128, those claiming security level 4 or 5 use SHAKE-256. In each proposed parameter set the output length (i.e. rm bits) is large enough to reach the required hardness of finding collisions. Therefore, a hash collision attack does not threaten the claimed security levels.

7 Advantages and limitations (2.B.6)

7.1 Advantages

- **Small signatures.** Like many other MQ signature schemes, the signatures of the LUOV scheme are very small. For security level 2 the signatures are only 319 bytes long.

- **A wide security margin** Instead of trying to estimate the complexity of existing attacks and choosing the parameters such that these estimates match the required security level we have formulated conservative lower bounds to plausible attacks. For example, we have assumed that a classical attacker can solve a determined system of n Boolean quadratic polynomials with only $2^{0.75n}$ bit operations, whereas the best known algorithms seem to require $2^{0.80n+7}$ operations at best. On top of our conservative lower bounds, we require the $\log_2$ of this lower bound to exceed the $\log_2$ of the required number of operations by 10% (see Sect. 5).

- **Simple arithmetic.** The scheme only uses SHA-3 and simple arithmetic operations over $\mathbb{F}_2$ or over an extension field. Arithmetic over $\mathbb{F}_2$ translates to the operations AND and XOR, while the arithmetic over an extension field can be implemented with XOR, additions and table lookups in small tables. This makes the algorithm very suitable for hardware implementations.

- **Message recovery.** It is possible to use the LUOV scheme in a message recovery mode. In this mode, a part of the message can be recovered from the signature and does not need to be communicated. This can reduce the size of a message-signature pair by up to 15 percent of the signature size.

- **Deterministic signatures.** The generation of a signature does not require any external source of randomness. This makes a secure implementation easier and excludes any attack that might exploit the usage of a poor source of randomness.

- **Stateless.** The signing algorithm does not need to maintain a state between signing sessions and can sign an unbounded number of messages. This makes a secure implementation of the algorithm easier.

- **Flexible.** The parameters of the signature are easily adjustable to reach a specific security level. It is also possible to choose parameters to make a trade-off between small signatures and small public keys.

- **Diversity.** Multivariate cryptography relies on a different hard problem than other branches such as lattice cryptography or hash-based cryptography. It is prudent to have cryptographic algorithms that rely on a diverse set of hard problems such that if one hard problem is broken and wipes out a branch of cryptography, there are alternative algorithms available.

7.2 Limitations

- **Public key size.** Even though the public key size of the LUOV scheme is much smaller than the public key size of other MQ signature schemes, it remains larger than the public key size of some other post quantum signature schemes. It is possible to mitigate this problem by making a trade-off for a smaller public key at the cost of larger signatures.

- **No encryption or KEM.** The LUOV scheme is a digital signature scheme. This submission does not include an encryption scheme or a key encapsulation mechanism.

References

[1] Magali Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2004.

[2] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the complexity of solving quadratic Boolean systems. *Journal of Complexity*, 29(1):53–75, 2013.

[3] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, 3(3):177–197, 2009.

[4] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Solving polynomial systems over finite fields: Improved analysis of the hybrid approach. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, pages 67–74. ACM, 2012.

[5] Ward Beullens and Bart Preneel. Field lifting for smaller UOV public keys. In *Progress in Cryptology–INDOCRYPT 2017: 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2016, Proceedings 18*. Springer, 2017.

[6] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 203–218. Springer, 2010.

[7] Peter Czypek. *Implementing Multivariate Quadratic Public Key Signature Schemes on Embedded Devices*. PhD thesis, Diploma Thesis, Chair for Embedded Security, Ruhr-Universität Bochum, 2012.

[8] Claus Diem. The XL-algorithm and a conjecture from commutative algebra. In *Asiacrypt*, volume 4, pages 338–353. Springer, 2004.

[9] Jean-Charles Faugère and Sylvain Lachartre. Parallel Gaussian elimination for Gröbner bases computations in finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 89–97. ACM, 2010.

[10] Jean-Charles Faugère and Ludovic Perret. On the security of UOV. *IACR Cryptology ePrint Archive*, 2009:483, 2009.

[11] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[12] Antoine Joux and Vanessa Vitse. A crossbred algorithm for solving Boolean polynomial systems. *IACR Cryptology ePrint Archive*, 2017:372, 2017.

[13] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 206–222. Springer, 1999.

[14] Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil and Vinegar signature scheme. In *Annual International Cryptology Conference*, pages 257–266. Springer, 1998.

[15] Jacques Patarin. The Oil and Vinegar signature scheme. In *Dagstuhl Workshop on Cryptography1997*, 1997.

[16] Albrecht Petzoldt. *Selecting and Reducing Key Sizes for Multivariate Cryptography.* PhD thesis, TU Darmstadt, July 2013. Referenten: Professor Dr. Johannes Buchmann, Professor Jintai Ding, Ph.D.

[17] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *International Workshop on Public Key Cryptography*, pages 156–171. Springer, 2012.

[18] Christopher Wolf and Bart Preneel. Equivalent keys in multivariate quadratic public key systems. *Journal of Mathematical Cryptology*, 4(4):375–415, 2011.

A Statements

These statements "must be mailed to Dustin Moody, Information Technology Laboratory, Attention: Post-Quantum Cryptographic Algorithm Submissions, 100 Bureau Drive – Stop 8930, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, or can be given to NIST at the first PQC Standardization Conference (see Section 5.C)."

First blank in submitter statement: full name. Second blank: full postal address. Third, fourth, and fifth blanks: name of cryptosystem. Sixth and seventh blanks: describe and enumerate or state "none" if applicable.

First blank in patent statement: full name. Second blank: full postal address. Third blank: enumerate. Fourth blank: name of cryptosystem.

First blank in implementor statement: full name. Second blank: full postal address. Third blank: full name of the owner.

A.1 Statement by Each Submitter

I, Ward Beullens, of Afdeling ESAT - COSIC, Kasteelpark Arenberg 10 - bus 2452, 3001 Heverlee, Belgium, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LUOV, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that (check one):

☑ *I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LUOV OR (check one or both of the following):*

☐ *to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LUOV may be covered by the following U.S. and/or foreign patents: None*

☐ *I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations: None*

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed: Ward Beullens

Title:
Date:
Place:

A.2 Statement by Reference/Optimized Implementations' Owner(s)

I, Ward Beullens, Afdeling ESAT - COSIC, Kasteelpark Arenberg 10 - bus 2452, 3001 Heverlee, Belgium, am the owner or authorized representative of the owner Ward Beullens of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed: Ward Beullens
Title:
Date:
Place:

Ming-Shing Chen
Andreas Hülsing
Joost Rijneveld
Simona Samardjiska
Peter Schwabe

MQDSS specifications

Version 1.0 November 2017

Introduction

This document is a detailed specification of the design and security arguments of the digital signature scheme MQDSS. It is divided in two main parts:

- **Part I Backbone results** - contains:
  - an analysis of the hardness of the underlying hard problem with respect to both classical and quantum algorithms - Chapter 2,
  - a description of the underlying Identification scheme - Chapter 3,
  - a description and proof of security of the underlying construction - Chapter 5.
- **Part II MQDSS Specifications** - contains:
  - a detailed description of MQDSS - Chapter 7 and Chapter 9,
  - proposed and additional parameter sets - Chapter 8,
  - security analysis of MQDSS- Chapter 10,
  - justification of the design choices - Chapter 11,
  - a detailed performance analysis of the reference implementation using the proposed parameter sets - Chapter 12.
  - a discussion on the security vs performance tradeoffs - Chapter 13,
  - a summary of the strengths and weaknesses - Chapter 14, and
  - a short description of an additional AVX2 implementation - Chapter 15.

Contents

Part II MQDSS Specifications

Part I

Backbone Results
- Underlying Construction and Security Arguments -

1

Preliminaries

1.1 Notations and Conventions

Let $A(\cdot,\cdot,\dots)$ be a randomized algorithm. We write $y \leftarrow A(x_1, x_2, \dots)$ for the output of the algorithm on input $x_1, x_2, \dots$. The same notation is used for the output of a function. If S is a set, then $s \leftarrow_R S$ denotes that s is drawn uniformly at random from S.

Furthermore, let $\mathbb{F}_q$ denote the finite field of order q. We use boldface letters $\mathbf{u}$ to denote vectors over a finite field, i.e. $\mathbf{u} \in \mathbb{F}_q^n$, for some positive integer $n \in \mathbb{N}$. We call a function $\mathbb{F}_q^n \to \mathbb{F}_q^m$ a vectorial function.

1.2 Security Notions and Definitions

In the following we provide basic security related definitions used throughout these specifications.

A function μ is called negligible (in k) if for every positive polynomial p, and sufficiently large k it holds that $\mu(k) < 1/p(k)$. For better readability we sometimes denote negligible functions by $\mathrm{negl}(k)$.

We say that two distribution ensembles $\{X_k\}_{k\in\mathbb{N}}$ and $\{Y_k\}_{k\in\mathbb{N}}$ indexed by a security parameter k are computationally indistinguishable if for any non-uniform probabilistic polynomial time algorithm $\mathcal{A}$

$$|\mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(X_k\right)\right] - \mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(Y_k\right)\right]| = \mathrm{negl}(k)\,.$$

1.2.1 Digital Signatures

This specification describes a construction of digital-signature schemes. These are defined as follows.

Definition 1.1 (Digital signature scheme). *A digital-signature scheme with security parameter k, denoted $\mathsf{Dss}(1^k)$ is a triplet of polynomial-time algorithms $\mathsf{Dss} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ defined as follows:*

- *The key-generation algorithm* KGen *is a probabilistic algorithm that outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$.
- *The signing algorithm* Sign *is a possibly probabilistic algorithm that on input a secret key* sk *and a message M outputs a signature σ.*

- *The verification algorithm* Vf *is a deterministic algorithm that on input a public key* pk*, a message* M *and a signature* σ *outputs a bit* b*, where* $b = 1$ *indicates that the signature is accepted and* $b = 0$ *indicates a reject.*

We write Dss instead of $\mathsf{Dss}(1^k)$, whenever the security parameter k is clear from context or irrelevant. For correctness of a Dss, we require that for all $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}()$, all messages M and all signatures $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, M)$, we get $\mathsf{Vf}(\mathsf{pk}, M, \sigma) = 1$, i.e., that correctly generated signatures are accepted.

Existential Unforgeability under Adaptive Chosen Message Attacks.
The standard security notion for digital signature schemes is existential unforgeability under adaptive chosen message attacks ($\mathsf{EU\text{-}CMA}$) [34], defined as follows.

Experiment $\mathsf{Exp}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}()$
$(M^\star, \sigma^\star) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$
Let $\{(M_i)\}_1^{Q_s}$ be the queries to $\mathsf{Sign}(\mathsf{sk}, \cdot)$.
Return 1 iff $\mathsf{Vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$ and $M^\star \notin \{M_i\}_1^{Q_s}$.

For the success probability of an adversary $\mathcal{A}$ in the above experiment we write

$$\mathrm{Succ}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = \mathsf{Pr}\left[\mathsf{Exp}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = 1\right].$$

A signature scheme is called $\mathsf{EU\text{-}CMA}$-secure if any PPT algorithm $\mathcal{A}$ has only negligible success probability in the $\mathsf{Exp}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A})$ experiment. More formally, we have the following definition.

Definition 1.2 ($\mathsf{EU\text{-}CMA}$ security). *Let* $k \in \mathbb{N}$ *and* Dss *a digital signature scheme with security parameter* k*. We call* Dss *existentially unforgeable under chosen message attacks or* $\mathsf{EU\text{-}CMA}$*-secure if for all* $Q_s, t = \mathrm{poly}(k)$ *the success probability of any PPT algorithm* $\mathcal{A}$ *(the adversary) running in time* $\leq t$*, making at most* Q_s *queries to* Sign *in the* $\mathsf{Exp}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A})$ *experiment, is negligible in* k*:*

$$\mathrm{Succ}^{\mathsf{eu\text{-}cma}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = \mathrm{negl}(k).$$

In the security proof of our signature scheme, we will also make use of the weaker notion of security against key-only attacks (KOA). The difference from $\mathsf{EU\text{-}CMA}$ security is that the adversary is given no access to the signing oracle, i.e., $Q_s = 0$. More formally, we define the following experiment.

Experiment $\mathsf{Exp}^{\mathsf{koa}}_{\mathsf{Dss}(1^k)}(\mathcal{A})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}()$
$(M^\star, \sigma^\star) \leftarrow \mathcal{A}(\mathsf{pk})$
Return 1 iff $\mathsf{Vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$.

Definition 1.3 (KOA security). *Let* $k \in \mathbb{N}$ *and* Dss *a digital signature scheme with security parameter* k*. We call* Dss *secure under key only attacks or* KOA*-secure if for all* $t = \mathrm{poly}(k)$ *the success probability of any PPT adversary* $\mathcal{A}$ *running in time* $\leq t$ *in the* $\mathsf{Exp}^{\mathsf{koa}}_{\mathsf{Dss}(1^k)}(\mathcal{A})$ *experiment, is negligible in* k*:*

$$\mathrm{Succ}^{\mathsf{koa}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = \mathrm{negl}(k),$$

where $\mathrm{Succ}^{\mathsf{koa}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = \mathsf{Pr}\left[\mathsf{Exp}^{\mathsf{koa}}_{\mathsf{Dss}(1^k)}(\mathcal{A}) = 1\right]$.

1.2.2 Identification Schemes

An identification scheme (IDS) is a protocol that allows a prover $\mathcal{P}$ to prove its identity to a verifier $\mathcal{V}$. More formally:

Definition 1.4 (Identification scheme). *An identification scheme with security parameter k, denoted $\mathsf{IDS}(1^k)$, is a triplet of PPT algorithms $\mathsf{IDS} = (\mathsf{KGen}, \mathcal{P}, \mathcal{V})$ such that:*

- *the key generation algorithm KGen outputs a key pair $(\mathsf{sk}, \mathsf{pk})$.*
- *$\mathcal{P}$ and $\mathcal{V}$ are interactive algorithms, executing a common protocol. The prover $\mathcal{P}$ takes as input a secret key sk and the verifier $\mathcal{V}$ takes as input a public key pk. At the conclusion of the protocol, $\mathcal{V}$ outputs a bit b with $b = 1$ indicating "accept" and $b = 0$ indicating "reject".*

We write IDS instead of $IDS(1^k)$, if the security parameter k is clear from context or irrelevant. For correctness of an IDS, we require that for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}()$ we have

$$\mathsf{Pr}\left[\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk}) \rangle = 1\right] = 1,$$

where $\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk}) \rangle$ refers to the common execution of the protocol between $\mathcal{P}$ with input sk and $\mathcal{V}$ on input pk. In this case we say that the IDS is perfectly correct.

For the following definitions we need the notion of a transcript. A transcript of an execution of an identification scheme IDS refers to all the messages exchanged between $\mathcal{P}$ and $\mathcal{V}$ and is denoted by $\mathsf{trans}(\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk}) \rangle)$.

We will focus on canonical $2n+1$-pass IDS, where the prover and the verifier exchange $2n+1$ messages, n challenges and n replies. These IDS are defined as follows.

Definition 1.5 (Canonical $2n+1$-pass identification schemes). *Consider $\mathsf{IDS} = (\mathsf{KGen}, \mathcal{P}, \mathcal{V})$, a $2n+1$-pass identification scheme with n challenge spaces $\mathsf{C}_1, \ldots, \mathsf{C}_n$. We call IDS a canonical $2n+1$-pass identification scheme if the prover can be split into $n+1$ subroutines $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the verifier into $n+1$ subroutines $\mathcal{V} = (\mathsf{ChS}_1, \ldots, \mathsf{ChS}_n, \mathsf{Vf})$ such that:*

- *$\mathcal{P}_0(\mathsf{sk})$ computes the initial commitment com sent as the first message and a state state fed forward to $\mathcal{P}_1$.*
- *ChS_1, computes the first challenge message $\mathsf{ch}_1 \leftarrow_R \mathsf{C}_1$, sampling at random from the challenge space C_1.*
- *$\mathcal{P}_1(\mathsf{state}, \mathsf{ch}_1)$, computes the first response resp_1 of the prover (and updates the state state) given access to the state and the first challenge.*
- *For every $i \in \{2, \ldots, n\}$*
  - *ChS_i, computes the i-th challenge message $\mathsf{ch}_i \leftarrow_R \mathsf{C}_i$.*
  - *$\mathcal{P}_i(\mathsf{state}, \mathsf{ch}_i)$, computes the i-th response resp_i of the prover given access to the state and the i-th challenge.*
- *$\mathsf{Vf}(\mathsf{pk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{ch}_n, \mathsf{resp}_n)$, upon access to the public key and the whole transcript outputs $\mathcal{V}$'s final decision.*

Note that the state forwarded among the prover algorithms can contain all inputs to previous prover algorithms if they are needed later. We also assume that the verifier keeps all sent and received messages to feed them to Vf.

Our construction uses the special case of canonical 5-pass IDS (where $n = 2$). On the other hand, standard choice in the literature for building signatures is the special case

$n = 1$. For comparison, we will use both in these specifications, and for completeness and clarity we provide figures of both. Figure 1.1 describes a canonical 3-pass IDS, and Figure 1.2 a canonical 5-pass IDS.

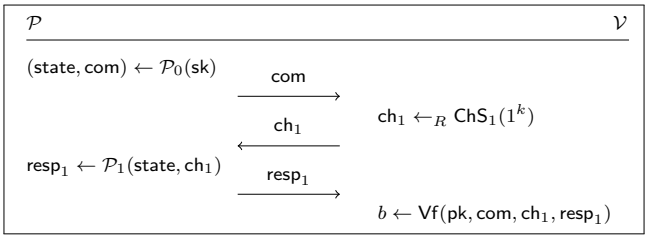


Fig. 1.1: Canonical 3-pass IDS

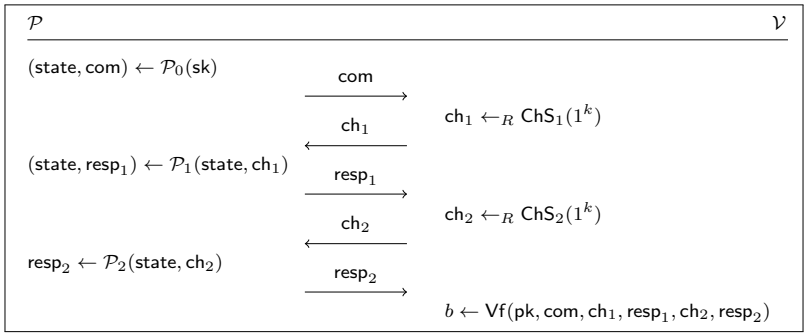


Fig. 1.2: Canonical 5-pass IDS

Furthermore, we will consider a particular type of canonical 5-pass IDS where the size of the two challenge spaces is restricted to q and 2.

Definition 1.6 ($q2$-Identification scheme). *A $q2$-Identification scheme* IDS *is a canonical 5-pass identification scheme where for the challenge spaces* C_1 *and* C_2 *it holds that* $|\mathsf{C}_1| = q$ *and* $|\mathsf{C}_2| = 2$.

Security against Impersonation under Passive Attack.
The standard security notion for identification schemes is security against impersonation. Here, the goal of the adversary - the impersonator $\mathcal{I}$, is to impersonate the prover in an interaction with an honest verifier without the knowledge of the secret key. When talking about passive attacks, the impersonator, besides the public key, might have access to polynomially many valid interactions between the prover and the verifier (via eavesdropping for example), i.e., access to a transcript oracle $\mathsf{Trans}(\mathsf{pk}, \mathsf{sk}, \cdot)$ that outputs valid transcripts of honest executions.

For a canonical $2n + 1$-pass IDS we consider the following experiment:

Experiment $\mathsf{Exp}^{\mathsf{imp\text{-}pa}}_{\mathsf{IDS}(1^k)}(\mathcal{I})$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}()$
$(\mathsf{state}, \mathsf{com}) \leftarrow \mathcal{I}^{\mathsf{Trans}(\mathsf{pk},\mathsf{sk},\cdot)}(\mathsf{pk})$
For every $i \in \{1, \ldots, n\}$
$\mathsf{ch}_i \leftarrow_R \mathsf{ChS}_i(1^k)$
$(\mathsf{state}, \mathsf{resp}_i) \leftarrow \mathcal{I}^{\mathsf{Trans}(\mathsf{pk},\mathsf{sk},\cdot)}(\mathsf{pk})$
Return 1 iff $\mathsf{Vf}(\mathsf{pk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{ch}_n, \mathsf{resp}_n) = 1$.

For the success probability of the impersonator $\mathcal{I}$ in the above experiment we write

$$\mathrm{Succ}^{\mathsf{imp\text{-}pa}}_{\mathsf{IDS}(1^k)}(\mathcal{I}) = \mathsf{Pr}\left[\mathsf{Exp}^{\mathsf{imp\text{-}pa}}_{\mathsf{IDS}(1^k)}(\mathcal{I}) = 1\right].$$

Definition 1.7 (IMP-PA security). *Let $k \in \mathbb{N}$ and* IDS *a canonical $2n+1$ identification scheme with security parameter k. We say* IDS *is secure against impersonation under passive attacks or IMP-PA-secure if for all $Q_t, t = \mathrm{poly}(k)$ the success probability of any PPT impersonator $\mathcal{I}$ running in time $\leq t$, making at most Q_t queries to* Trans *in the $\mathsf{Exp}^{\mathsf{imp\text{-}pa}}_{\mathsf{IDS}(1^k)}(\mathcal{I})$ experiment, is negligible in k:*

$$\mathrm{Succ}^{\mathsf{imp\text{-}pa}}_{\mathsf{IDS}(1^k)}(\mathcal{I}) = \mathrm{negl}(k).$$

Security Properties of Identification Schemes.
The properties of identification schemes interesting in our context are those that provide passive security. We next give the necessary definitions.

First of all, it must be hard for any cryptographic scheme to derive a valid secret key given a public key. To formally capture this intuition, we need to define what valid means. For this we define the notion of a key relation.

Definition 1.8 (Key relation). *Let* IDS *be an identification scheme and R some relation. We say* IDS *has key relation R if*

$$\forall(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}() : (\mathsf{pk},\mathsf{sk}) \in R$$

Now that we have defined what valid means, we can define key-one-wayness.

Definition 1.9 (Key-One-Wayness). *Let $k \in \mathbb{N}$ be the security parameter, $\mathsf{IDS}(1^k)$ be an identification scheme with key relation R. We call* IDS *key-one-way (KOW) (with respect to key relation R) if for all polynomial time algorithms $\mathcal{A}$,*

$$\mathrm{Succ}^{pq-kow}_{\mathsf{IDS}(1^k)}(\mathcal{A}) = \mathsf{Pr}\left[(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}(), \mathsf{sk}' \leftarrow \mathcal{A}(\mathsf{pk}) : (\mathsf{pk},\mathsf{sk}') \in R\right] = \mathrm{negl}(k)$$

Definition 1.10 (Soundness (with soundness error κ)). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k) = (\mathsf{KGen}, \mathcal{P}, \mathcal{V})$ an identification scheme with security parameter k. We say that* IDS *is sound, with soundness error κ, if for every PPT algorithm $\mathcal{A}$ (the adversary),*

$$\mathsf{Pr}\left[\begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}() \\ \mathcal{A}(1^k,\mathsf{pk}), \mathcal{V}(\mathsf{pk})\rangle = 1\end{array}\right] \leq \kappa + \mathrm{negl}(k).$$

Definition 1.11 ((computational) Honest-verifier zero-knowledge). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k) = (\mathsf{KGen}, \mathcal{P}, \mathcal{V})$ an identification scheme with security parameter k. We say that* IDS *is computational honest-verifier zero-knowledge (HVZK) if there exists a probabilistic polynomial time algorithm $\mathcal{S}$, called the simulator, such that for any polynomial time algorithm $\mathcal{A}$ and $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}()$:*

$$\mathrm{Succ}^{pq-hvzk}_{\mathsf{IDS}(1^k)}(\mathcal{A}) =$$
$$\left|\mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(\mathsf{sk},\mathsf{pk},\mathsf{trans}(\langle\mathcal{P}(\mathsf{sk}),\mathcal{V}(\mathsf{pk})\rangle)\right)\right] - \mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(\mathsf{sk},\mathsf{pk},\mathcal{S}(\mathsf{pk})\right)\right]\right| = \mathrm{negl}(k).$$

Definition 1.12 (Special soundness). *Let* $\mathsf{IDS}(1^k)$ *be a* 3*-pass Identification scheme with key relation* R *and* $\mathcal{A}$ *a polynomial time algorithm that upon input of security parameter* 1^k *and an* $\mathsf{IDS}(1^k)$ *public key* pk *outputs, with non-negligible probability, four valid transcripts with respect to* pk*:*

$$\begin{array}{l} \mathsf{trans} = (\mathsf{com},\ \mathsf{ch}_1,\ \mathsf{resp}_1), \\ \mathsf{trans}' = (\mathsf{com},\ \mathsf{ch}'_1,\ \mathsf{resp}'_1), \end{array} \tag{1.1}$$

where $\mathsf{ch}_1 \neq \mathsf{ch}'_1$*.*

We say that $\mathsf{IDS}(1^k)$ *is special sound if there exists a polynomial time algorithm* $\mathcal{K}_{IDS}$*, the extractor, that, given a public key* pk *and access to* $\mathcal{A}$*, outputs a secret key* sk *such that* $(\mathsf{pk}, \mathsf{sk}) \in R$ *with non-negligible success probability in* k*.*

To prove security of our signature scheme, we will make use of the existence of so called $q2$-extractor which is a variant of special soundness. This is combined with a notion of key-one-wayness to later be able to argue about security.

Definition 1.13 ($q2$**-Extractor).** *Let* $\mathsf{IDS}(1^k)$ *be a* $q2$*-Identification scheme with key relation* R *and* $\mathcal{A}$ *a polynomial time algorithm that upon input of security parameter* 1^k *and an* $\mathsf{IDS}(1^k)$ *public key* pk *outputs, with non-negligible probability, four valid transcripts with respect to* pk*:*

$$\begin{array}{ll} \mathsf{trans}^{(1)} = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \mathsf{ch}_2, \mathsf{resp}_2), & \mathsf{trans}^{(3)} = (\mathsf{com}, \mathsf{ch}'_1, \mathsf{resp}'_1, \mathsf{ch}_2, \mathsf{resp}_2), \\ \mathsf{trans}^{(2)} = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \mathsf{ch}'_2, \mathsf{resp}'_2), & \mathsf{trans}^{(4)} = (\mathsf{com}, \mathsf{ch}'_1, \mathsf{resp}'_1, \mathsf{ch}'_2, \mathsf{resp}'_2). \end{array} \tag{1.2}$$

where $\mathsf{ch}_1 \neq \mathsf{ch}'_1$ *and* $\mathsf{ch}_2 \neq \mathsf{ch}'_2$*.*

We say that $\mathsf{IDS}(1^k)$ *has a* $q2$-Extractor *if there exists a polynomial time algorithm* $\mathcal{K}_{IDS}$*, the extractor, that, given a public key* pk *and access to* $\mathcal{A}$*, outputs a secret key* sk *such that* $(\mathsf{pk}, \mathsf{sk}) \in R$ *with non-negligible success probability in* k*.*

Security Properties of Commitments.
The security of identification schemes relies on the properties of the underlying commitment scheme. The goal of "commiting" to a certain value is twofold: It should not be feasible for anyone to discover this value before the prover opens the commitment, but also, it should not be feasible for the prover to open the commitment in multiple ways. These two properties are known as hiding and binding. They come in different flavors - perfect, statistical and computational, depending on what "feasible" means.

For our purposes, the weakest version of computational hiding and binding will suffice. These are formally defined as follows.

Definition 1.14 (Computationally hiding commitments). *Let* $k \in \mathbb{N}$*,* $\mathsf{Com}(1^k)$ *a commitment scheme with security parameter* k*. We say that* Com *is computationally hiding if for any two messages* M, M' *and random string* ρ*, for any polynomial time algorithm* $\mathcal{A}$*:*

$$\left|\mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(\mathsf{Com}(\rho, M)\right)\right] - \mathsf{Pr}\left[1 \leftarrow \mathcal{A}\left(\mathsf{Com}(\rho, M')\right)\right]\right| = \mathrm{negl}(k)\,.$$

Definition 1.15 (Computationally binding commitments). *Let* $k \in \mathbb{N}$*,* $\mathsf{Com}(1^k)$ *a commitment scheme with security parameter* k*. We say that* Com *is computationally binding if for any polynomial time algorithm* $\mathcal{A}$

$$\mathsf{Pr}\left[\mathsf{Com}(\rho, M) = \mathsf{Com}(\rho', M') \wedge M \neq M' : (\rho, M, \rho', M') \leftarrow \mathcal{A}(1^k)\right] = \mathrm{negl}(k)\,.$$

2

The $\mathcal{MQ}$ Problem

2.1 Multivariate Quadratic ($\mathcal{MQ}$) Functions and the $\mathcal{MQ}$ Problem

Let $m, n, q \in \mathbb{N}$, $\mathbf{x} = (x_1, \ldots, x_n)$ and let $\mathcal{MQ}(n, m, \mathbb{F}_q)$ denote the family of vectorial functions $\mathbf{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ of degree 2 over $\mathbb{F}_q$:

$$\mathcal{MQ}(n, m, \mathbb{F}_q) = \{\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))| \\ f_s(\mathbf{x}) = \sum_{i,j} a_{i,j}^{(s)} x_i x_j + \sum_i b_i^{(s)} x_i, s \in \{1, \ldots, m\}\}.$$

We will refer to $\mathbf{F} \in \mathcal{MQ}(n, m, \mathbb{F}_q)$ as multivariate quadratic ($\mathcal{MQ}$) function. Given $\mathbf{v} \in \mathbb{F}_q^m$ we will refer to $\mathbf{F}(\mathbf{x}) = \mathbf{v}$ as system of m quadratic equations in n variables.

We will omit m, n, q whenever they are clear from the context.

Definition 2.1. *Let $\mathbf{F} \in \mathcal{MQ}(n, m, \mathbb{F}_q)$. The function $\boldsymbol{G}(\mathbf{x}, \mathbf{y}) = \boldsymbol{F}(\mathbf{x} + \mathbf{y}) - \boldsymbol{F}(\mathbf{x}) - \boldsymbol{F}(\mathbf{y})$ is called the polar form of the function $\boldsymbol{F}$.*

It is not hard to verify that the polar form is bilinear, i.e., for every $\mathbf{a}_1, \mathbf{a}_2, \mathbf{b} \in \mathbb{F}_q^n$ it holds

$$\mathbf{G}(\mathbf{a}_1 + \mathbf{a}_2, \mathbf{b}) = \mathbf{G}(\mathbf{a}_1, \mathbf{b}) + \mathbf{G}(\mathbf{a}_2, \mathbf{b}) \text{ and}$$
$$\mathbf{G}(\mathbf{b}, \mathbf{a}_1 + \mathbf{a}_2) = \mathbf{G}(\mathbf{b}, \mathbf{a}_1) + \mathbf{G}(\mathbf{b}, \mathbf{a}_2).$$

Definition 2.2 ($\mathcal{MQ}$ relation). *The $\mathcal{MQ}$ relation is the binary relation defined as: $R_{\mathcal{MQ}(m,n,q)} \subseteq (\mathcal{MQ}(n, m, \mathbb{F}_q) \times \mathbb{F}_q^m) \times \mathbb{F}_q^n : ((\mathbf{F}, \mathbf{v}), \mathbf{s}) \in R_{\mathcal{MQ}(m,n,q)}$ iff $\mathbf{F}(\mathbf{s}) = \mathbf{v}$.*

We relate the following problem to the family $\mathcal{MQ}(n, m, \mathbb{F}_q)$ of $\mathcal{MQ}$ functions:

Definition 2.3 ($\mathcal{MQ}$ problem (search version)). *Let $m, n, q \in \mathbb{N}$. An instance $\mathcal{MQ}(\mathbf{F}, \mathbf{v})$ of the $\mathcal{MQ}$ (search) problem is defined as:*

Given $\mathbf{F} \in \mathcal{MQ}(n, m, \mathbb{F}_q), \mathbf{v} \in \mathbb{F}_q^m$ find, if any, $\mathbf{s} \in \mathbb{F}_q^n$ such that

$$((\mathbf{F}, \mathbf{v}), \mathbf{s}) \in R_{\mathcal{MQ}(m,n,q)}.$$

The decisional version of the $\mathcal{MQ}$ problem is known to be NP-complete [32][1]. It is widely believed that the $\mathcal{MQ}$ problem is intractable even for quantum computers in the average case. We formalize the intractability of the $\mathcal{MQ}$ problem through the following.

[1] Note that the $\mathcal{MQ}$ problem is a special case of the more general problem of solving a system of equations over a finite field of degree $deg \geq 2$, known as $\mathcal{PoSSo}$. The decisional version of the $\mathcal{PoSSo}$ problem is NP-complete [32].

Assumption 2.4 ($\mathcal{MQ}$ assumption) *Let $m, n, q \in \mathbb{N}$, $\mathbf{F} \leftarrow_R \mathcal{MQ}(n, m, \mathbb{F}_q)$ and $\mathbf{s} \leftarrow_R \mathbb{F}_q^n$. For every polynomial time quantum algorithm $\mathcal{A}$ given $\mathbf{F}$ and $\mathbf{v} = \mathbf{F}(\mathbf{s})$ it is computationally hard to find a solution $\mathbf{s}'$ to the $\mathcal{MQ}(\mathbf{F}, \mathbf{v})$ problem. More formally,*

$$\mathsf{Pr}\left[\left((\mathbf{F}, \mathbf{v}), \mathbf{s}'\right) \in R_{\mathcal{MQ}(m,n,q)} : \begin{array}{l} \mathbf{F} \leftarrow_R \mathcal{MQ}(n, m, \mathbb{F}_q) \\ \mathbf{s} \leftarrow_R \mathbb{F}_q^n \\ ((\mathbf{F}, \mathbf{v}), \mathbf{s}) \in R_{\mathcal{MQ}(m,n,q)} \\ \mathbf{s}' \leftarrow \mathcal{A}(1^k, \mathbf{F}, \mathbf{v}) \end{array}\right] = \mathrm{negl}(k)\,.$$

2.2 Classical Algorithms for Solving the $\mathcal{MQ}$ Problem

The difficulty of solving the $\mathcal{MQ}$ problem is strongly dependent on the ratio between the number of variables n and number of equations m. It is known that when $m \geqslant n(n+1)/2$ (overdetermined systems) and when $n \geqslant m(m+1)$ (underdetermined systems) the $\mathcal{MQ}$ problem is solvable in polynomial time.

The first case is simply a result of replacing all monomials by a new variable, and solving a linear system in $n(n+1)/2$ variables and at least as many equations. The second case was solved by Kipnis, Patarin and Goubin [40] and later [53] Thomae and Wolf showed that the complexity gradually increases to exponential when $m \approx n$. Indeed, the most interesting case is when $m = n$: Adding more equations gives away more information about the system; On the other hand, if there are more variables, we can simply fix the excess of them, and end up with a system of the same number of variables and equations.

In the rest of this section we will assume that $m \geqslant n$, but also that $m = \mathcal{O}(n)$. For this range of parameters, the state of the art algorithms employ algebraic techniques that analyze the properties of the ideal generated by the given polynomials. The most important are the algorithms from the F4/F5 family [26, 27, 5, 12], and the variants of the XL algorithm [21, 24, 59, 58]. Although different in description, the two families bear many similarities, which results in similar complexity [60]. Therefore, in our analysis we will not consider the algorithms from the XL family.

In the Boolean case, today's state of the art algorithms BooleanSolve [6] and FXL [58], provide improvement over exhaustive search, with an asymptotic complexity of $\Theta(2^{0.792n})$ and $\Theta(2^{0.875n})$ for $m = n$, respectively. Practically, the improvement is visible for polynomials with more than 200 variables. A very recent algorithm, the Crossbred algorithm [37] over $\mathbb{F}_2$, is likely to further improve the asymptotic complexity, as the authors report that it passes the exhaustive search barrier already for 37 Boolean variables.

Interestingly, the current best known algorithms, BooleanSolve [6], FXL [58, 59], the Crossbred algorithm [37] and the Hybrid approach [12] all combine algebraic techniques with exhaustive search. This immediately allows for improvement in their quantum version using Grover's quantum search algorithm [36], provided the cost of implementing them on a quantum computer does not diminish the gain from Grover. These algorithms will be subject to our interest in the rest of the section. Their implementation on a quantum computer and the speed up from using Grover's algorithm will be discussed in Section 2.3.

For comparison reasons, in our analysis we will also consider exhaustive search performed through fast enumeration techniques [13]. We will not consider a probabilistic method recently proposed by Lokshtanov et al. [43]. Although it is provably faster than exhaustive search, the improvement in the case of odd characteristic fields is not comparable to the best algebraic methods. Furthermore, it has not been studied enough and has

not been implemented (to the best of our knowledge), so even in the Boolean case where the asymptotic complexity is $\mathcal{O}(2^{0.8765n})$ it is not clear for what values of n this algorithm outperforms exhaustive search.

In the rest of this section, let $\mathbf{F} = (f_1, \ldots, f_m), f_i \in \mathbb{F}_q[x_1, \ldots, x_n]$. Without loss of generality, the equation system that we want to solve is $\mathbf{F}(\mathbf{x}) = \mathbf{0}$.

2.2.1 Exhaustive search

A natural and simple way of obtaining a solution of the given system is to try out all possible values $\mathbf{x} \in \mathbb{F}_q^n$ until the system is satisfied. A naïve implementation would require $2\binom{n}{2}$ additions and multiplications for a single polynomial, and m times more for the entire system, amounting to a complexity of $\mathcal{O}(mn^2q^n)$ field operations. However, in [13], Bouillaguet *et al.* introduced a technique for fast enumeration in $\mathbb{F}_2$ that needs only $\log_2(n)2^{n+2}$ Boolean operations. The technique uses Gray codes enumeration and partial derivatives of the polynomials. Although [13] considers only the Boolean case, the technique can be extended to larger fields by using q-ary Gray codes. So, for simplicity we will assume that fast enumeration can be performed in $\log_q(n)q^n$ operations over a field of size q.

2.2.2 The HybridF5 algorithm

Currently, the standard algorithms for solving generic instances of the $\mathcal{MQ}$ problem are the algorithms for computing a Gröbner basis of the ideal generated by the set of $\mathcal{MQ}$ polynomials. The idea was first introduced by Buchberger [15] and later further developed by Lazard [42] who established the link between computing the Gröbner basis and performing Gaussian elimination on the Macaulay matrices (of degree up to a sufficiently large integer D) of the given polynomials.[2] The algorithm was improved several times by Buchberger himself in order to reduce the number of unnecessary reductions to 0 during the Gröbner basis computation. A significant improvement was done in the variant proposed by Faugère [26] known as the F4-algorithm. The main improvement comes from the introduced strategy to reduce all critical pairs of minimal degree at once (instead of one by one) using the Macaulay matrix and sparse matrix algebra techniques. Later, Faugère completely removed the reductions to zero for semi-regular sequences in the improved F5-algorithm [27, 7, 8].

The semi-regularity assumption is crucial in this algorithm (as it will be in the other algebraic methods we consider). Informally (which is enough for our purposes), a sequence of polynomials $(f_1, \ldots, f_m), m \geq n$ is semi-regular if the only relations (dependencies) among the polynomials are the trivial ones generated by $f_i f_j - f_j f_i = 0$. Note that the regularity assumption is a very plausible one for randomly generated polynomials, and has been experimentally supported (see for example [6]). We will also assume that the instances generated in our signature scheme are semi-regular.

The main complexity in the F5 algorithm (and also in the BooleanSolve and the Crossbread algorithm) comes from performing Gauss elimination on the Macaulay matrix $Mac_D(\mathbf{F})$ of degree D (the matrix whose rows are formed by the coefficients of monomials

[2] In essence, it can be considered as a generalization of Gaussian elimination for nonlinear polynomials. One important distinction is that, unlike in Gaussian elimination, in Gröbner basis algorithms the order in which the variables are eliminated and generally, the ordering of the monomials, is very important (see [27, 7] for example).

uf_i of maximal degree D). The degree D should be big enough so that a Gröbner basis of the ideal generated by the polynomials can be obtained by row-reducing the Macaulay matrix. The smallest such D is called the degree of regularity D_{reg}, and for semi-regular systems it is given by $D_{reg}(n,m) = 1 + deg(HS_q(t))$, where

$$HS_q(t) = \left[\frac{(1-t^2)^m}{(1-t)^n}\right]_+, \text{for } q > 2, \text{ and } HS_2(t) = \left[\frac{(1+t)^n}{(1+t^2)^m}\right]_+,$$

and the $_+$ subscript denotes that the series has been truncated before the first non-positive coefficient.

Now, for the case of $q > 2$ the Macaulay matrix $Mac_D(\mathbf{F})$ has $\binom{n+D-1}{D}$ columns and $m\binom{n+D-3}{D-2}$ rows, so computing the row-echelon form requires $\Theta(m\binom{n+D-3}{D-2}\binom{n+D-1}{D}^{\omega-1})$ operations, where $2 \leqslant \omega \leqslant 3$ is the linear algebra constant. The computation is repeated for every $D \in \{2, \ldots, D_{reg}\}$, which amounts to a total of $\Theta\left(m\sum_{D=2}^{D_{reg}}\binom{n+D-3}{D-2}\binom{n+D-1}{D}^{\omega-1}\right)$ field operations. In the case of $q = 2$, the logic is the same, except that now we use plain combinations (instead of combinations with repetition as for $q > 2$), so the formula becomes $\Theta\left(m\sum_{D=2}^{D_{reg}}\binom{n}{D-2}\binom{n}{D}^{\omega-1}\right)$. More compactly, the complexity of the F5 algorithm is:

$$\begin{aligned} C_{F5}(q,m,n) &= \mathcal{O}\left(mD_{reg}\binom{n+D_{reg}(n,m)-1}{D_{reg}(n,m)}^{\omega}\right), \text{ for } q > 2, \text{ and} \\ C_{F5}(2,m,n) &= \mathcal{O}\left(mD_{reg}\binom{n}{D_{reg}(n,m)}^{\omega}\right) \end{aligned} \tag{2.1}$$

field operations [8]. The value of the linear algebra constant ω depends on the algorithm used, and it ranges from $\omega = 3$ for naïve Gauss elimination down to $\omega = 2.376$ for Coppersmith-Winograd algorithm [19], and even further to $\omega < 2.373$ due to improvements by Vassilevska-Williams [57]. However these algorithms are extremely complex and with a huge constant factor to be actually useful in practice. For cryptanalysis purposes, the best we can hope for is $\omega = \log_2(7)$, obtained using Strassen algorithm [51].

The Hybrid approach introduced by Bettale *et al.*[12], tries to reduce the complexity of F5 by introducing a trade-off between brute-forcing and the F5 algorithm for smaller $\mathcal{MQ}$ instances. Namely, the algorithms first fixes $n - k$ variables, so the reduction is now performed on $Mac_{D_{reg}}(\tilde{\mathbf{F}})$, where $\tilde{\mathbf{F}} = (\tilde{f}_1, \ldots, \tilde{f}_m)$ and $\tilde{f}_i(x_1, \ldots, x_k) = f_i(x_1, \ldots, x_k, a_{k+1}, \ldots, a_n)$, for every $(a_{k+1}, \ldots, a_n) \in \mathbb{F}_2^{n-k}$. The value of k is chosen such that the overall complexity is minimized. In total the complexity of the Hybrid approach for solving systems of n equations in n variables over $\mathbb{F}_q$ is

$$C_{Hyb}(n,k) = Guess(q, n-k) \cdot C_{F5}(q,k,n), \tag{2.2}$$

where $Guess(q, n-k) = \mathcal{O}(\log(n-k)q^{n-k})$ is the cost of the exhaustive search over all q^{n-k} possibilities, including partially evaluating $n - k$ variables in field operations, and $C_{F5}(k,n)$ is given in (2.1).

Note that the technique of fixing variables had already been used in the XL algorithm [21], and this version is known as FXL [58, 59].

2.2.3 The BooleanSolve algorithm

In the case of $\mathbb{F}_2$, the BooleanSolve algorithm [6] performs better than the Hybrid approach. Similar to the Hybrid approach, it requires a semi-regularity assumption on the $\mathcal{MQ}$

instance. Also as in the Hybrid approach, it first fixes some optimal amount $n-k$ of the variables and then performs some tests on the smaller instance.

Then, the problem of finding a solution is basically reduced to testing the consistency of a related linear system

$$\mathbf{u} \cdot Mac_{D_{reg}}(\tilde{\mathbf{F}}) = (0, \ldots, 0, 1) \tag{2.3}$$

where $Mac_{D_{reg}}(\tilde{\mathbf{F}})$ is defined in the previous paragraph. If the system (2.3) is consistent, then the original system does not have a solution. This allows for pruning of all the inconsistent branches corresponding to some $\mathbf{a} \in \mathbb{F}_2^{n-k}$. A simple exhaustive search is then performed on the remaining branches. It can be shown that the running time of the algorithm is dominated by the first part of the algorithm (this holds true even in the quantum version of the algorithm, although in the quantum case the difference is not as big, as a consequence of the reduced complexity of the first part). Therefore, for simplicity, we omit the exhaustive search on the remaining branches from our analysis. The complexity of the BooleanSolve algorithm is given by

$$C_{Bool}(n,k) = Guess(2, n-k) \cdot C_{cons}(Mac_{D_{reg}}(\tilde{\mathbf{F}})), \tag{2.4}$$

where $Guess(2,k)$ is defined the same as in the Hybrid approach, and

$$C_{cons}(Mac_{D_{reg}}(\tilde{\mathbf{F}})) = \Theta(N^2 \log^2 N \log\log N), \quad N = \sum_{i=0}^{D_{reg}(k,n)} \binom{k}{i}$$

is the complexity of testing consistency of the system (2.3), using the sparse linear algebra algorithm from [33].

2.2.4 The Crossbread algorithm

Recently, Joux and Vitse [37] proposed a new algebraic method for solving quadratic systems over $\mathbb{F}_2$ called the Crossbred algorithm. Although originally only $\mathbb{F}_2$ was considered, the algorithm works the same for any field, so we will assume an arbitrary field $\mathbb{F}_q$. We will also assume that the given system is semi-regular.

The main idea of this approach is to first perform some operations on the Macaulay matrix of degree $D \geqslant D_{reg}(k,n)$ of the given system, and only afterwards to fix variables. Again, as in the previous algorithms, k is a suitably chosen optimization parameter such that the overall complexity is minimized. Furthermore, let $d \leqslant D$ be a small integer and $deg_k u$ denote the degree of the monomial u in the first k variables. Let $Mac^{(k)}_{D,d}(\mathbf{F})$ be the submatrix of $Mac_D(\mathbf{F})$ consisting of the rows indexed by uf_i, where $deg_k u \geqslant d-1$, and let $M^{(k)}_{D,d}(\mathbf{F})$ be the submatrix of $Mac^{(k)}_{D,d}(\mathbf{F})$ consisting of the columns indexed by u, where $deg_k u > d$.

The algorithm works as follows: In the first part, we try to find enough linearly independent elements v_i (in particular for $q > 2$ at least $\sum_{i=0}^{d} \binom{k+i-1}{i}$ including the original m when $d > 1$) in the kernel of $M^{(k)}_{D,d}(\mathbf{F})$, that are not in the kernel of $Mac^{(k)}_{D,d}(\mathbf{F})$. Next we find the set of polynomials corresponding to $v_i Mac^{(k)}_{D,d}(\mathbf{F})$. These polynomials, (possibly together with the original when $d > 1$) form a new system $\mathbf{P}$ that will be of interest in the second part of the algorithm.

In this part, for each $(a_{k+1}, \ldots, a_n) \in \mathbb{F}_q^{n-k}$ we form the system $\tilde{\mathbf{P}}(x_1, \ldots, x_k) = \mathbf{P}(x_1, \ldots, x_k, a_{k+1}, \ldots, a_n)$. It is crucial to observe that $\tilde{\mathbf{P}}$ is of degree d and the system contains $\sum_{i=0}^{d} \binom{k+i-1}{i}$ equations when $q > 2$ ($\sum_{i=0}^{d} \binom{k}{i}$ when $q = 2$), which means it is possible to solve it easily by linearization, i.e. by considering each monomial as new variable, and solving the resulting linear system.

The advantage here comes from using sparse linear algebra algorithms on $Mac_D(\mathbf{F})$ for the first part and dense linear algebra only on the smaller matrix in the second part. Note that, as long as the number of the remaining k variables is small, the sparse linear algebra part takes much less time, since in this case $D_{reg}(k, n)$ is also small. It turns out that actually it is more efficient to work with a Mac_D, with $D > D_{reg}(k, n)$, but not too large so that the cost of the first part becomes significant. The complexity thus, is dominated by enumeration of $n-k$ variables in a system of n variables of degree D over $\mathbb{F}_q$ $q > 2$, and checking whether the obtained system has a valid solution. In total, under the condition that:

$$Dim(Ker(M_{D,d}^{(k)}(\mathbf{F})) \setminus Ker(Mac_{D,d}^{(k)}(\mathbf{F}))) \geqslant \sum_{i=0}^{d} \binom{k+i-1}{i}, \tag{2.5}$$

the complexity of the Crossbread algorithm for $q > 2$ is given by[3]

$$C_{Cross}(n, k, d) = Sparse(M_{D,d}^{(k)}(\mathbf{F})) + Guess(q, n-k) \cdot \binom{k+d-1}{d}^{\omega}, \tag{2.6}$$

where $Guess(q, k)$ is defined the same as in the Hybrid approach, and $Sparse(M_{D,d}^{(k)}(\mathbf{F})) = \mathcal{O}(\binom{n+D-1}{D}^2)$ is the complexity for finding the kernel vectors using for example the block Lanczos algorithm [44] or the block Wiedemann algorithm [20] for sparse matrices (or their improvements). Note that an external specialization of variables is also possible, but we have verified that this does not bring any improvement in the number of operations. However it is useful for parallelization of the algorithm. [4]

At the end of this section, we provide the number of field operations of the described algorithms for solving $\mathcal{MQ}$ instances for various fields $\mathbb{F}_q$ and different values of $m = n$ that are interesting for practical use. Table 2.1 summarizes the Boolean case, and Table 2.2 lists the values for some common choices of $q > 2$. Note that the cost of the algorithms is given in field operations as is common for such algorithms. We did not use the gate count metric from the NIST call for proposals [46], but it is not difficult to convert to number of gates if necessary.

[3] The case of $q = 2$ is similar except that the system is of size $\approx \binom{k}{d}$.

[4] The preprint [37] does not contain a complexity analysis of the Crossbread algorithm, nor are all the choices in the algorithm described. What is written here is our interpretation of the algorithm.

	CrossBread ($d=1$)		BooleanSolve		HybridF5		FastEnum
n	k	Field op.	k	Field op.	k	Field op.	Field op.
128	28	2^{118}	40	2^{135}	16	2^{137}	2^{133}
144	30	2^{130}	52	2^{150}	17	2^{153}	2^{149}
160	30	2^{148}	55	2^{164}	18	2^{168}	2^{165}
192	31	2^{179}	80	2^{191}	35	2^{198}	2^{197}
224	32	2^{210}	96	2^{219}	38	2^{228}	2^{229}
256	33	2^{241}	102	2^{246}	55	2^{259}	2^{261}
296	34	2^{280}	139	2^{280}	59	2^{296}	2^{301}

Table 2.1: Comparison of the time complexity of the Crossbread algorithm [37], the BooleanSolve algorithm [6], the Hybrid Approach [12] and exhaustive search through fast enumeration [13] in terms of field operations for $\mathbb{F}_2$. The parameter k denotes the number of remaining variables in the specialization process in each of the algorithms respectively.

		HybridF5		CrossBread ($d=1$)		FastEnum
q	n	k	Field op.	k	Field op.	Field op.
4	80	32	2^{160}	21	2^{134}	2^{164}
4	96	35	2^{188}	21	2^{166}	2^{195}
4	112	44	2^{215}	22	2^{196}	2^{228}
4	128	53	2^{242}	23	2^{226}	2^{260}
4	144	51	2^{269}	24	2^{257}	2^{292}
4	160	60	2^{296}	25	2^{287}	2^{324}
16	48	36	2^{147}	18	2^{135}	2^{194}
16	64	41	2^{190}	19	2^{196}	2^{258}
16	72	49	2^{210}	20	2^{224}	2^{290}
16	96	66	2^{273}	21	2^{316}	2^{386}
31	40	32	2^{134}	17	2^{129}	2^{200}
31	48	39	2^{159}	18	2^{164}	2^{240}
31	64	49	2^{205}	19	2^{238}	2^{319}
31	88	71	2^{274}	20	2^{353}	2^{438}
31	96	72	2^{297}	21	2^{388}	2^{478}
32	48	39	2^{159}	18	2^{165}	2^{242}
32	64	52	2^{206}	19	2^{240}	2^{322}
32	88	71	2^{274}	20	2^{356}	2^{442}
32	96	72	2^{298}	21	2^{391}	2^{482}
64	40	32	2^{143}	17	2^{153}	2^{242}
64	64	52	2^{217}	19	2^{286}	2^{386}
128	40	37	2^{143}	17	2^{176}	2^{281}
128	64	59	2^{222}	19	2^{330}	2^{450}
256	40	37	2^{146}	17	2^{199}	2^{321}

Table 2.2: Comparison of the time complexity of the the Hybrid Approach [12], the Crossbread algorithm [37], and exhaustive search through fast enumeration [13] in terms of field operations for common choices of the field $\mathbb{F}_q$, $q > 2$. The parameter k denotes the number of remaining variables in the specialization process in each of the algorithms respectively.

2.3 Using Grover's Algorithm for Solving the $\mathcal{MQ}$ Problem

In this section we will investigate the cost of the quantum versions of the known classical algorithms that we described in the previous Section. To the best of our knowledge, there are no dedicated quantum algorithms for solving the $\mathcal{MQ}$ problem. We are only aware of the work of Westerbaan and Schwabe [56], who investigate the cost of exhaustive search using Grover's algorithm against the $\mathcal{MQ}$ problem. In our paper [16], where we first introduce MQDSS, we briefly analyze the gain of applying Grover on the Hybrid approach.

Here, we will use a more accurate metric, and following NIST's recommendations [46] we will express the cost of the algorithms in terms of number of fault-tolerant quantum gates and quantum circuit depth.

2.3.1 Finite Field Arithmetic on Quantum Computers

Fault-Tolerant quantum gates.
In quantum computing, similarly as in classical computing, there is a need for fixed small universal set of instructions that can be used to express any type of reversible quantum operation. Furthermore, such universal sets need to have fault-tolerant implementations to reduce pilling up of noise and thus errors in quantum computation. Recent work [4, 3] has identified "Clifford+T" as the standard universal fault-tolerant gate set. It is the set of gates generated by $= \{H, CNOT, T\}$ where,

$$\begin{aligned} H &: |x\rangle \mapsto \frac{|0\rangle + (-1)^x|1\rangle}{\sqrt{2}}, \\ CNOT &: |x\rangle|y\rangle \mapsto |x\rangle|x \oplus y\rangle, \\ T &: |x\rangle \mapsto e^{\frac{i\pi x}{4}}|x\rangle. \end{aligned}$$

We will also need the Toffoli gate:

$$Toffoli : |x\rangle|y\rangle|z\rangle \mapsto |x\rangle|y\rangle|z \oplus xy\rangle.$$

which is also a common gate in designing circuits. Many implementations of the Toffoli gate using Clifford+T gate are known, depending on whether the goal is to minimize the number of ancilla qubits used, the gate count or circuit depth. In our evaluation we have chosen a balanced metric, assuming sufficient number of ancilla qubits. In other words, we are interested in implementations that minimize at the same time the T-count and T-depth (of T gates only) but more importantly, the overall gate count and depth including Clifford gates. Thus we will use the implementation from Amy *et al.*[4] that requires 7 T-gates and 8 Clifford gates, and has T-depth 4, and overall depth 8.

In what follows, we will use the same metric to evaluate larger quantum circuits.

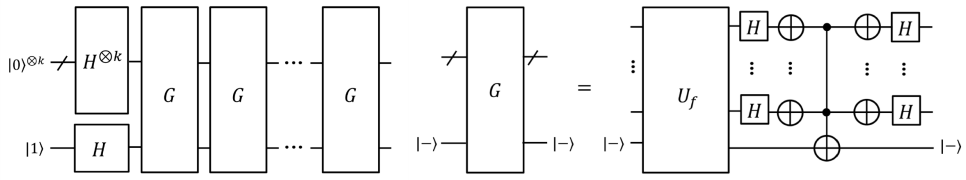
Cost of finite field addition and multiplication.
The algorithms we are interested in, are all performed over finite fields $\mathbb{F}(2^s)$ or $\mathbb{F}_p$ for p prime. Therefore we need an estimate for the cost of the arithmetic operations over these fields. We use the results from [9, 18, 38]:

- Addition over $\mathbb{F}(2^s)$ can be implemented using s parallel $CNOT$ (Clifford) gates (so the overall depth is 1).

- Multiplication over $\mathbb{F}(2^s)$ (using Karatsuba's algorithm) can be implemented using $7s^{\log_2(3)}$ T-gates and $10s^{\log_2(3)} - 2s$ Clifford gates, with T-depth of $4s$ and overall depth of $9s$
- Addition over $\mathbb{F}(p)$ can be implemented using approximately $180\log_2(p)$ Clifford gates and $140\log_2(p)$ T-gates with the same depth.
- Multiplication over $\mathbb{F}(p)$ can be implemented using approximately $2{\cdot}180\log_2^2(p)$ Clifford gates and $2 \cdot 140\log_2^2(p)$ T-gates with T-depth of $2 \cdot 140\log_2(p)$ and overall depth of $2 \cdot 320\log_2(p)$.

2.3.2 Grover's Quantum Search Algorithm

Grover's algorithm [36] searches for an item in an unordered list of size $N = 2^k$ that satisfies a certain condition given in a form of a quantum black-box function $f : \{0,1\}^k \to \{0,1\}$ and realized as a unitary circuit $U_f : |x\rangle|y\rangle \mapsto |x\rangle|x \oplus f(y)\rangle$ - the "oracle". If the condition is satisfied for an item x_0, then $f(x_0) = 1$, otherwise $f(x_0) = 0$. The algorithm consists of applying an optimal number of times the operator $G = U_f\left((H^{\otimes k}(2|0\rangle\langle 0| - 1_{2^k})H^{\otimes k}) \otimes 1_2\right)$ on a state $|\psi\rangle \otimes |\phi\rangle$ where the first register has been prepared in an equal superposition of all $|x\rangle$, i.e., $|\psi\rangle = \frac{1}{\sqrt{2^k}}\sum_{x\in\{0,1\}^k}|x\rangle$, and $\phi = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ (see Figure 2.1). The operator G needs to be iteratively repeated $\left\lfloor \frac{\pi}{4}\sqrt{N/M} \right\rfloor$ times, where M is the number of items that satisfy the condition f [14]. In this case, if $M \ll N$, the algorithm fails with negligible probability $\leqslant M/N$. Note that, even if the number of solutions M is unknown, a slight modification of the algorithm from [14], again guarantees that a solution will be found with overwhelming probability after $\left\lfloor \frac{9}{4}\sqrt{N/M} \right\rfloor$ Grover iterations. In the next subsection, we will elaborate on the number of solutions of the $\mathcal{MQ}$ problem.



(a) (b)

Fig. 2.1: Quantum circuit that implements Grover's algorithm for a search space of size $N = 2^k$. Figure from [35]. (a) The full algorithm where G is the Grover iterate that represents one round of the algorithm. (b) One round of Grover's algorithm (detailed view of the operator G).

From the above, and assuming we know the number of solutions, the cost of Grover's algorithm can be expressed as:

$$Cost(Grover) = \left\lfloor \frac{\pi}{4}\sqrt{2^k/M} \right\rfloor \cdot (Cost(U_f) + Cost(U_s)) \tag{2.7}$$

where $U_s = 2|s\rangle\langle s| - 1_{2^k}$ is the Grover diffusion operator. Here $Cost$ can be any metric of choice, such as quantum gate count or quantum circuit depth.

In [35] it was calculated that the diffusion operator U_s can be implemented as a k-fold $CNOT$ gate which requires $8k - 24$ Toffoli gates, which easily translates to Clifford+T gates (see previous Section).

In the next subsection we will consider several different instantiations of the function U_f leading to a solution for the $\mathcal{MQ}$ problem $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. The oracle U_f can be any of the following:

- The $\mathcal{MQ}$ oracle $U_{\mathcal{MQ}}$: $U_{\mathcal{MQ}}(\mathbf{a}) = 1$ for $\mathbf{a} \in \mathbb{F}_q^n$ if $\mathbf{F}(\mathbf{a}) = \mathbf{0}$ (cf. Subsection (2.2.1));
- The BooleanSolve oracle U_{Bool}: $U_{Bool}(\mathbf{a}) = 1$ for $\mathbf{a} \in \mathbb{F}_2^{n-k}$ if the system (2.3) is inconsistent and the second part of the BooleanSolve algorithm on $\tilde{\mathbf{F}}$ outputs $\mathbf{b} \in \mathbb{F}_q^k$ such that $\tilde{\mathbf{F}}(\mathbf{b}) = \mathbf{0}$ (cf. Subsection (2.2.3));
- The Hybrid F5 oracle U_{HybF5}: $U_{HybF5}(\mathbf{a}) = 1$ for $\mathbf{a} \in \mathbb{F}_q^{n-k}$ if the F5 algorithm on $\tilde{\mathbf{F}}$ outputs $\mathbf{b} \in \mathbb{F}_q^k$ such that $\tilde{\mathbf{F}}(\mathbf{b}) = \mathbf{0}$ (cf. Subsection (2.2.2));
- The Crossbread oracle U_{Cross}: $U_{Cross}(\mathbf{a}) = 1$ for $\mathbf{a} \in \mathbb{F}_q^{n-k}$ if the Crossbread algorithm on $\tilde{\mathbf{P}}$ outputs a solution $\mathbf{b} \in \mathbb{F}_q^k$ such that $\tilde{\mathbf{P}}(\mathbf{b}) = \mathbf{0}$ (cf. Subsection (2.2.4)).

2.3.3 Resource Estimates of Grover Enhanced Quantum Algorithms for Solving the $\mathcal{MQ}$ Problem

On the number of solutions of the $\mathcal{MQ}$ problem.
As already mentioned, our proposal uses randomly generated instances $\mathbf{F} \in \mathcal{MQ}(n, m, \mathbb{F}_q)$, where the number of polynomials m is the same as the number of variables n. The goal of the adversary will be to find one solution of a system $\mathbf{F}(\mathbf{x}) = \mathbf{v}$, for a given public value $\mathbf{v} \in \mathbb{F}^n$. While our key generation mechanism guarantees that this system will have at least one solution, we don't know the exact number of solutions which is an important parameter in Grover's algorithm. In the previous section we saw that it is possible to overcome this problem by adapting Grover's algorithm to such a setting. But, we can actually argue that there is no need for that, and that it is safe to assume in our analysis that the number of solutions is $M = 1$. Indeed, in [30], it was shown that the number of solutions of a system of n equations in n variables follows the Poisson distribution with parameter $\lambda = 1$ (the expected value is 1), i.e. the probability that the system has exactly M solutions is $\frac{1}{eM!}$. Furthermore, the probability that there are more than M solutions can be estimated as the tail probability of a Poisson distribution which is negligible in M. This means that with overwhelming probability, the number of solutions is very small, and we can simply run Grover first assuming $M = 1$, then $M = 2$ and so on, until the algorithm succeeds. In particular, since we know that the system has at least one solution, the probability that it is the only solution is $\frac{1}{e-1} \approx 0.58$, and that there are at most 2 solutions $\frac{5}{4(e-1)} \approx 0.73$. Hence, the adversary has a good chance to succeed already in the first two runs, and the probability quickly rises with each additional run. In our analysis, we will assume that it is enough to run Grover only for $M = 1$ (as a lower bound of the cost of the algorithm).

The $\mathcal{MQ}$ oracle.
In [56], Westerbaan and Schwabe constructed two oracles for evaluation of $\mathcal{MQ}$ polynomials over $\mathbb{F}_2$ and estimated the cost of Grover's algorithm using these oracles. Here we will adapt their estimates for the case of any field $\mathbb{F}_q$. As our metrics is mainly circuit size and depth (and not number of qubits) we will focus on their approach for the first

oracle. Their second oracle uses approximately half the number of qubits of their first oracle (with a small overhead), but double the circuit size.

Following [56] we estimate that the $\mathcal{MQ}$ oracle $U_{\mathcal{MQ}}$ over $\mathbb{F}_q$ requires approximately $4n^2m$ field multiplications and as many field additions. The total depth required for the multiplications is approximately $4n$, as is for the additions. Using the formulas:

$$\begin{aligned} Gates(\mathcal{MQ}Grover) &= \left\lfloor \tfrac{\pi}{4} 2^{\lfloor \log_2 q \rfloor \frac{n}{2}} \right\rfloor \cdot (Gates(U_{\mathcal{MQ}}) + Gates(U_s)), \\ Depth(\mathcal{MQ}Grover) &= \left\lfloor \tfrac{\pi}{4} 2^{\lfloor \log_2 q \rfloor \frac{n}{2}} \right\rfloor \cdot (Depth(U_{\mathcal{MQ}}) + Depth(U_s)), \end{aligned} \tag{2.8}$$

and the results from Subsections 2.3.1 and 2.3.2 we obtain an estimate of the cost of Exhaustive search with Grover's algorithm. The results are summarized in Table 2.3.

		Gates		Depth	
q	n	T	Clifford	T	Total
2	128	$2^{89.46}$	$2^{89.82}$	$2^{76.54}$	$2^{77.63}$
2	192	$2^{123.21}$	$2^{123.58}$	$2^{109.13}$	$2^{110.23}$
2	224	$2^{139.88}$	$2^{140.24}$	$2^{125.36}$	$2^{126.45}$
2	256	$2^{156.46}$	$2^{156.82}$	$2^{141.55}$	$2^{142.64}$
4	72	$2^{96.55}$	$2^{96.97}$	$2^{85.02}$	$2^{86.15}$
4	96	$2^{121.80}$	$2^{122.21}$	$2^{109.44}$	$2^{110.57}$
4	112	$2^{138.47}$	$2^{138.88}$	$2^{125.66}$	$2^{126.79}$
4	128	$2^{155.04}$	$2^{155.46}$	$2^{141.85}$	$2^{142.98}$
16	32	$2^{86.63}$	$2^{87.08}$	$2^{77.21}$	$2^{78.38}$
16	40	$2^{103.60}$	$2^{104.04}$	$2^{93.53}$	$2^{94.70}$
16	48	$2^{120.38}$	$2^{120.83}$	$2^{109.80}$	$2^{110.96}$
16	64	$2^{153.63}$	$2^{154.08}$	$2^{142.22}$	$2^{143.38}$
31	24	$2^{87.77}$	$2^{88.13}$	$2^{78.60}$	$2^{79.80}$
31	32	$2^{108.83}$	$2^{109.19}$	$2^{98.84}$	$2^{100.03}$
31	40	$2^{129.61}$	$2^{129.97}$	$2^{118.98}$	$2^{120.17}$
31	48	$2^{150.22}$	$2^{150.58}$	$2^{139.06}$	$2^{140.25}$
31	56	$2^{170.70}$	$2^{171.06}$	$2^{159.09}$	$2^{160.29}$
32	32	$2^{103.14}$	$2^{103.60}$	$2^{93.66}$	$2^{94.84}$
32	40	$2^{124.11}$	$2^{124.56}$	$2^{113.99}$	$2^{115.16}$
32	48	$2^{144.89}$	$2^{145.35}$	$2^{134.25}$	$2^{135.43}$
32	56	$2^{165.56}$	$2^{166.02}$	$2^{154.47}$	$2^{155.65}$
64	24	$2^{94.31}$	$2^{94.78}$	$2^{85.62}$	$2^{86.80}$
64	32	$2^{119.56}$	$2^{120.02}$	$2^{110.04}$	$2^{111.22}$
64	40	$2^{144.52}$	$2^{144.99}$	$2^{134.36}$	$2^{135.54}$
64	48	$2^{169.31}$	$2^{169.77}$	$2^{158.62}$	$2^{159.81}$
128	24	$2^{106.66}$	$2^{107.13}$	$2^{97.94}$	$2^{99.13}$
128	32	$2^{135.91}$	$2^{136.38}$	$2^{126.36}$	$2^{127.54}$
128	40	$2^{164.87}$	$2^{165.34}$	$2^{154.67}$	$2^{155.87}$
256	16	$2^{85.22}$	$2^{85.69}$	$2^{77.63}$	$2^{78.82}$
256	24	$2^{118.97}$	$2^{119.44}$	$2^{110.21}$	$2^{111.41}$
256	32	$2^{152.21}$	$2^{152.69}$	$2^{142.63}$	$2^{143.83}$

Table 2.3: Cost of Exhaustive search on the $\mathcal{MQ}$ problem using Grover's algorithm

The HybridF5 oracle.
The Hybrid Approach includes partial evaluation of $n-k$ variables. The cost of this part can be computed similarly as for the $\mathcal{MQ}$ oracle. We found that for this part we need

$6(n-k)km$ multiplications and the same amount of additions. The depth of this part of the circuit is $6(n-k)$ times the depth of a multiplication and $6(n-k)$ times the depth of an addition.

The rest of the cost of the Hybrid Approach comes from implementing Strassen's algorithm on the Macaulay matrix of size $\binom{k+D_{reg}-1}{D_{reg}}$ in a quantum circuit. The cost is approximately $5\binom{k+D_{reg}-1}{D_{reg}}^{\log_2 7}$ field additions which can be realized in circuit depth $3\log_2\binom{k+D_{reg}-1}{D_{reg}}$ times the depth of an addition.

Using the formulas:

$$\begin{aligned} Gates(HybF5Grover) &= \left\lfloor \tfrac{\pi}{4} 2^{\lfloor \log_2 q \rfloor \frac{n-k}{2}} \right\rfloor \cdot (Gates(U_{HybF5}) + Gates(U_s)), \\ Depth(HybF5Grover) &= \left\lfloor \tfrac{\pi}{4} 2^{\lfloor \log_2 q \rfloor \frac{n-k}{2}} \right\rfloor \cdot (Depth(U_{HybF5}) + Depth(U_s)), \end{aligned} \tag{2.9}$$

and the results from Subsections 2.3.1 and 2.3.2 we obtain an estimate of the cost of the Hybrid F5 algorithm with Grover's search algorithm. The results are summarized in Table 2.4.

The Crossbread oracle.
Similarly as the HybridF5 oracle, the Crossbread oracle includes partial evaluation of $n-k$ variables, so this cost is the same. For each enumeration, the Strassen's algorithm is applied on a matrix of size $\binom{k+d-1}{d}$ for a small d which costs $5\binom{k+d-1}{d}^{\log_2 7}$ field additions and total depth $3\log_2\binom{k+d-1}{d}$ times the depth of an addition. An important feature of the algorithm is that it can be split into two distinct parts: Sparse linear algebra on the Macaulay matrix, and enumeration plus dense linear algebra to check the consistency of the smaller system obtained from the kernel elements of $M_{D,d}^{(k)}$. The first part, that is more memory demanding can always be performed on a classical computer, and the second part which can make use of Grover's algorithm can be performed on a quantum computer. This is undoubtedly a big advantage over the quantum version of the Hybrid Approach, albeit the later is in theory faster.

The cost of the entire algorithm for various parameters is given in Table 2.5.

The BooleanSolve oracle.
For the BooleanSolve oracle, the cost of the partial evaluation of $n-k$ is the same as for the previous oracles. The rest comes from consistency checking of a system of size $\binom{k}{D_{reg}}$ using a sparse linear algebra technique from [33]. For simplicity, we will lower bound the cost of this part by $\binom{k}{D_{reg}}^2 \log^2\binom{k}{D_{reg}}$ additions (xor) in $\mathbb{F}_2$ of linear depth $\binom{k}{D_{reg}}$.

The cost of the entire algorithm for various parameters is given in Table 2.6.

			Gates		Depth	
q	n	k	T	Clifford	T	Total
2	120	59	$2^{54.23}$	$2^{139.87}$	$2^{42.50}$	$2^{46.14}$
2	128	63	$2^{56.56}$	$2^{143.84}$	$2^{44.59}$	$2^{48.26}$
2	168	83	$2^{67.73}$	$2^{174.26}$	$2^{54.99}$	$2^{59.02}$
2	192	95	$2^{74.31}$	$2^{197.09}$	$2^{61.18}$	$2^{65.52}$
2	232	115	$2^{85.12}$	$2^{226.97}$	$2^{71.46}$	$2^{76.10}$
4	72	35	$2^{60.16}$	$2^{134.03}$	$2^{49.27}$	$2^{52.01}$
4	96	47	$2^{73.40}$	$2^{164.67}$	$2^{61.69}$	$2^{64.75}$
4	104	51	$2^{77.74}$	$2^{171.29}$	$2^{65.80}$	$2^{68.89}$
4	128	63	$2^{90.64}$	$2^{211.72}$	$2^{78.10}$	$2^{81.69}$
4	136	67	$2^{94.90}$	$2^{218.18}$	$2^{82.19}$	$2^{85.80}$
16	48	23	$2^{73.00}$	$2^{119.41}$	$2^{62.70}$	$2^{64.37}$
16	64	31	$2^{90.24}$	$2^{151.34}$	$2^{79.11}$	$2^{81.02}$
16	72	35	$2^{98.74}$	$2^{172.03}$	$2^{87.28}$	$2^{89.40}$
16	88	43	$2^{115.61}$	$2^{203.83}$	$2^{103.57}$	$2^{105.92}$
31	40	18	$2^{118.76}$	$2^{119.12}$	$2^{75.98}$	$2^{77.17}$
31	48	20	$2^{136.74}$	$2^{137.10}$	$2^{91.18}$	$2^{92.37}$
31	56	22	$2^{154.25}$	$2^{154.61}$	$2^{106.37}$	$2^{107.52}$
31	64	28	$2^{172.07}$	$2^{172.44}$	$2^{112.07}$	$2^{113.26}$
31	72	30	$2^{189.42}$	$2^{189.78}$	$2^{127.15}$	$2^{128.34}$
32	40	19	$2^{75.22}$	$2^{118.45}$	$2^{65.26}$	$2^{66.81}$
32	56	27	$2^{96.67}$	$2^{154.85}$	$2^{85.74}$	$2^{87.49}$
32	64	31	$2^{107.24}$	$2^{168.16}$	$2^{95.94}$	$2^{97.71}$
32	72	35	$2^{117.75}$	$2^{190.85}$	$2^{106.10}$	$2^{108.06}$
64	40	19	$2^{86.14}$	$2^{129.21}$	$2^{76.02}$	$2^{77.50}$
64	48	23	$2^{98.92}$	$2^{145.00}$	$2^{88.29}$	$2^{89.78}$
64	56	27	$2^{111.59}$	$2^{169.62}$	$2^{100.51}$	$2^{102.16}$
64	64	31	$2^{124.16}$	$2^{184.92}$	$2^{112.70}$	$2^{114.37}$
128	32	15	$2^{82.03}$	$2^{113.34}$	$2^{72.43}$	$2^{73.72}$
128	40	19	$2^{96.99}$	$2^{139.93}$	$2^{86.75}$	$2^{88.17}$
128	48	23	$2^{111.78}$	$2^{157.72}$	$2^{101.01}$	$2^{102.45}$
128	56	27	$2^{126.44}$	$2^{184.34}$	$2^{115.23}$	$2^{116.81}$
256	32	15	$2^{90.84}$	$2^{122.03}$	$2^{81.12}$	$2^{82.39}$
256	40	19	$2^{107.80}$	$2^{150.62}$	$2^{97.44}$	$2^{98.82}$
256	48	23	$2^{124.58}$	$2^{170.41}$	$2^{113.70}$	$2^{115.10}$
256	56	27	$2^{141.24}$	$2^{199.03}$	$2^{129.93}$	$2^{131.45}$

Table 2.4: Cost of applying the Hybrid F5 algorithm on the $\mathcal{MQ}$ problem using Grover's algorithm

			Classical part	Gates		Depth	
q	n	k	Field. op.	T	Clifford	T	Total
2	128	21	$2^{69.79}$	$2^{76.69}$	$2^{77.05}$	$2^{66.20}$	$2^{67.31}$
2	160	25	$2^{91.87}$	$2^{91.60}$	$2^{91.96}$	$2^{80.54}$	$2^{81.64}$
2	192	27	$2^{105.09}$	$2^{107.26}$	$2^{107.62}$	$2^{95.82}$	$2^{96.92}$
2	224	30	$2^{118.36}$	$2^{122.36}$	$2^{122.73}$	$2^{110.55}$	$2^{111.65}$
2	296	35	$2^{154.47}$	$2^{156.92}$	$2^{157.28}$	$2^{144.46}$	$2^{145.57}$
4	88	19	$2^{92.65}$	$2^{92.46}$	$2^{92.88}$	$2^{82.45}$	$2^{83.59}$
4	96	19	$2^{88.24}$	$2^{100.74}$	$2^{101.16}$	$2^{90.59}$	$2^{91.74}$
4	120	23	$2^{115.69}$	$2^{121.67}$	$2^{122.09}$	$2^{110.93}$	$2^{112.07}$
4	128	24	$2^{124.76}$	$2^{128.93}$	$2^{129.34}$	$2^{118.02}$	$2^{119.17}$
4	160	28	$2^{154.32}$	$2^{157.81}$	$2^{158.23}$	$2^{146.36}$	$2^{147.50}$
16	48	17	$2^{85.02}$	$2^{84.87}$	$2^{85.33}$	$2^{75.77}$	$2^{76.93}$
16	56	19	$2^{94.79}$	$2^{97.51}$	$2^{97.96}$	$2^{88.01}$	$2^{89.18}$
16	72	23	$2^{123.38}$	$2^{122.54}$	$2^{123.00}$	$2^{112.41}$	$2^{113.57}$
16	80	25	$2^{133.03}$	$2^{134.98}$	$2^{135.43}$	$2^{124.57}$	$2^{125.74}$
16	96	28	$2^{156.92}$	$2^{161.71}$	$2^{162.16}$	$2^{150.86}$	$2^{152.03}$
31	48	20	$2^{97.57}$	$2^{99.27}$	$2^{99.63}$	$2^{89.33}$	$2^{90.52}$
31	56	22	$2^{112.01}$	$2^{114.76}$	$2^{115.13}$	$2^{104.47}$	$2^{105.66}$
31	64	24	$2^{126.43}$	$2^{130.17}$	$2^{130.54}$	$2^{119.57}$	$2^{120.76}$
31	72	27	$2^{140.84}$	$2^{143.07}$	$2^{143.43}$	$2^{132.12}$	$2^{133.31}$
31	80	28	$2^{151.00}$	$2^{160.81}$	$2^{161.18}$	$2^{149.67}$	$2^{150.86}$
32	48	19	$2^{93.55}$	$2^{95.95}$	$2^{96.41}$	$2^{86.65}$	$2^{87.83}$
32	64	24	$2^{126.43}$	$2^{124.65}$	$2^{125.12}$	$2^{114.60}$	$2^{115.78}$
32	72	26	$2^{136.64}$	$2^{140.14}$	$2^{140.60}$	$2^{129.79}$	$2^{130.97}$
32	80	28	$2^{151.00}$	$2^{155.57}$	$2^{156.03}$	$2^{144.96}$	$2^{146.14}$
32	88	29	$2^{156.56}$	$2^{173.44}$	$2^{173.90}$	$2^{162.63}$	$2^{163.81}$
64	40	18	$2^{86.85}$	$2^{89.13}$	$2^{89.60}$	$2^{80.17}$	$2^{81.34}$
64	48	21	$2^{105.20}$	$2^{104.90}$	$2^{105.37}$	$2^{95.45}$	$2^{96.63}$
64	56	24	$2^{119.83}$	$2^{120.56}$	$2^{121.03}$	$2^{110.69}$	$2^{111.87}$
64	64	27	$2^{138.24}$	$2^{136.13}$	$2^{136.60}$	$2^{125.89}$	$2^{127.07}$
64	72	27	$2^{140.84}$	$2^{160.58}$	$2^{161.05}$	$2^{150.14}$	$2^{151.32}$
128	40	19	$2^{90.48}$	$2^{96.99}$	$2^{97.47}$	$2^{87.93}$	$2^{89.11}$
128	56	26	$2^{130.80}$	$2^{129.93}$	$2^{130.41}$	$2^{119.94}$	$2^{121.12}$
128	64	27	$2^{138.24}$	$2^{154.98}$	$2^{155.45}$	$2^{144.71}$	$2^{145.89}$
128	72	27	$2^{140.84}$	$2^{183.43}$	$2^{183.90}$	$2^{172.96}$	$2^{174.15}$
256	32	15	$2^{68.54}$	$2^{90.84}$	$2^{91.32}$	$2^{82.39}$	$2^{83.58}$
256	48	23	$2^{112.38}$	$2^{124.58}$	$2^{125.06}$	$2^{114.96}$	$2^{116.14}$
256	64	27	$2^{138.24}$	$2^{173.79}$	$2^{174.26}$	$2^{163.48}$	$2^{164.67}$

Table 2.5: Cost of applying the Crossbread algorithm on the $\mathcal{MQ}$ problem using Grover's algorithm

		Gates		Depth	
n	k	T	Clifford	T	Total
144	71	$2^{61.06}$	$2^{112.88}$	$2^{48.76}$	$2^{69.46}$
160	79	$2^{65.52}$	$2^{119.58}$	$2^{52.92}$	$2^{74.75}$
192	95	$2^{74.31}$	$2^{138.99}$	$2^{61.18}$	$2^{88.25}$
208	103	$2^{78.66}$	$2^{151.94}$	$2^{65.30}$	$2^{96.57}$
256	127	$2^{91.55}$	$2^{177.46}$	$2^{77.60}$	$2^{115.13}$
264	131	$2^{93.68}$	$2^{180.51}$	$2^{79.65}$	$2^{117.65}$

Table 2.6: Cost of BooleanSolve on the $\mathcal{MQ}$ problem using Grover's algorithm

3

The Sakumoto-Shirai-Hiwatari (SSH) 5-pass IDS scheme

3.1 Description of the SSH 5-pass IDS

In [41], Sakumoto, Shirai, and Hiwatari proposed two new identification schemes, a 3-pass and a 5-pass IDS, based on the intractability of the $\mathcal{MQ}$ problem. In these specifications, and documents related to this submission, we will refer to identification schemes from [41] as the SSH 3-pass and 5-pass schemes.

Unlike previous public key schemes, the SSH schemes provably rely only on the $\mathcal{MQ}$ problem (and the security of the commitment scheme), and not on other related problems in multivariate cryptography such as the Isomorphism of Polynomials (IP) [48], the related Extended IP [25] and IP with partial knowledge [52] problems or the MinRank problem [22, 28].

The main idea from [41] is a clever splitting of the secret, that relies on the polar form of the function $\mathbf{F}$. With this technique, the secret $\mathbf{s}$ is split into $\mathbf{s} = \mathbf{r}_0 + \mathbf{r}_1$, and the public $\mathbf{v} = \mathbf{F}(\mathbf{s})$ can be represented as $\mathbf{v} = \mathbf{F}(\mathbf{r}_0) + \mathbf{F}(\mathbf{r}_1) + \mathbf{G}(\mathbf{r}_0, \mathbf{r}_1)$. In order for the polar form not to depend on both shares of the secret, $\mathbf{r}_0$ and $\mathbf{F}(\mathbf{r}_0)$ are further split as $\alpha\mathbf{r}_0 = \mathbf{t}_0 + \mathbf{t}_1$ and $\alpha\mathbf{F}(\mathbf{r}_0) = \mathbf{e}_0 + \mathbf{e}_1$. Now, because of the bilinearity of the polar form it holds that $\alpha\mathbf{v} = (\mathbf{e}_1 + \alpha\mathbf{F}(\mathbf{r}_1) + \mathbf{G}(\mathbf{t}_1, \mathbf{r}_1)) + (\mathbf{e}_0 + \mathbf{G}(\mathbf{t}_0, \mathbf{r}_1))$, and from only one of the two summands, represented by $(\mathbf{r}_1, \mathbf{t}_1, \mathbf{e}_1)$ and $(\mathbf{r}_1, \mathbf{t}_0, \mathbf{e}_0)$, nothing can be learned about the secret $\mathbf{s}$.

Let $(\mathsf{pk}, \mathsf{sk}) = ((\mathbf{F}, \mathbf{v}), \mathbf{s}) \in R_{\mathcal{MQ}}$ be the public and private keys of the prover $\mathcal{P}$ (i.e., key generation just samples from the $\mathcal{MQ}$ relation). The SSH 5-pass IDS from [41] is given in Figure 3.1.

3.2 Properties of the SSH 5-pass IDS

The following theorem summarizes the properties of the SSH 5-pass IDS.

Theorem 3.1. *The 5-pass identification scheme of Sakumoto, Shirai, and Hiwatari [41]:*

1. *Has key relation $R_{\mathcal{MQ}(m,n,q)}$,*
2. *Is KOW if the $\mathcal{MQ}$ search problem is hard on average,*
3. *Is perfectly correct,*
4. *Is computationally HVZK when the commitment scheme Com is computationally hiding,*
5. *Is argument of knowledge for $R_{\mathcal{MQ}(m,n,q)}$ with knowledge error $\frac{1}{2} + \frac{1}{2q}$ when the commitment scheme Com is computationally binding,*

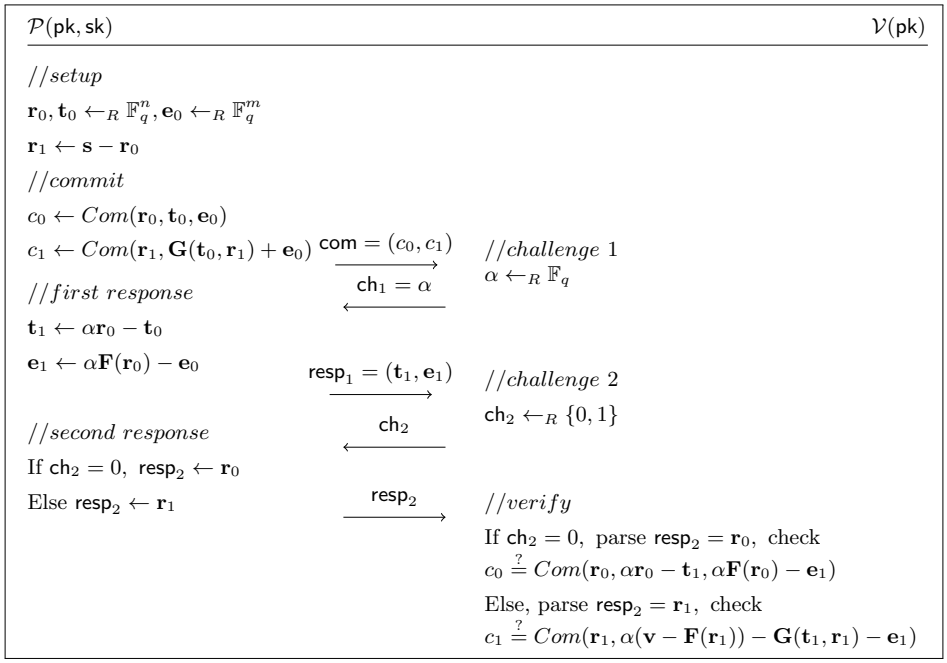


Fig. 3.1: The SSH 5-pass IDS by Sakumoto, Shirai, and Hiwatari [41]

6. *Is sound with soundness error* $\frac{1}{2} + \frac{1}{2q}$ *when the commitment scheme Com is computationally binding,*
7. *Has a q2-Extractor when the commitment scheme Com is computationally binding.*

The first statement holds by construction. The second statement follows directly from the first. The third, a stronger version of the fourth[1] and the fifth were proven in [41]. The last two statements were proven in [16].

[1] Sakumoto et al. [41] proved that their 5-pass scheme is statistically zero knowledge when the commitment scheme *Com* is statistically hiding which implies (honest-verifier) zero knowledge. Relaxing the requirements of *Com* to computationally hiding, weakens the result to computationally HVZK, since now, it is possible to distinguish (albeit only with negligible probability) whether the commitment was produced in a valid run of the protocol.

4

The Fiat-Shamir Transform

The Fiat-Shamir paradigm [29] for transforming canonical 3-pass identification schemes to signatures has been one the most popular methods for obtaining classically secure signature schemes. In this chapter, we present the transform, known results about its security, as well as its limitations.

4.1 Description of the Fiat-Shamir Transform

In what follows, let $\mathsf{IDS}^r = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}^r, \mathcal{V}^r)$ denote the parallel composition of r rounds of the identification scheme $\mathsf{IDS} = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}, \mathcal{V})$.

Construction 4.1 (Fiat-Shamir transform [29]) *Let $k \in \mathbb{N}$ the security parameter, $\mathsf{IDS} = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}, \mathcal{V})$, where $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1)$, $\mathcal{V} = (\mathsf{ChS}, \mathsf{Vf}_{\mathsf{IDS}})$ a canonical 3-pass Identification scheme that achieves soundness with soundness error κ. Select r, the number of (parallel) rounds of IDS, such that $\kappa^r = \mathrm{negl}(k)$, and that the challenge space C^r of the composition IDS^r, has exponential size in k. Moreover, select a cryptographic hash function $H : \{0,1\}^* \to \mathsf{C}^r$.*

A signature scheme derived from IDS via the Fiat-Shamir transform is a triplet of algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ defined as in Figures 4.1,4.2 and 4.3.

$\mathsf{KGen}()$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}_{\mathsf{IDS}}$
Return $(\mathsf{pk}, \mathsf{sk})$

Fig. 4.1: Fiat-Shamir key generation

$\mathsf{Sign}(\mathsf{sk}, M)$

For $j \in \{1, \ldots, r\}$ **do**
  $(\mathsf{state}^{(j)}, \mathsf{com}^{(j)}) \leftarrow \mathcal{P}_0(\mathsf{sk})$
$\mathsf{state} := (\mathsf{state}^{(1)}, \ldots, \mathsf{state}^{(r)})$
$\mathsf{com} := (\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(r)})$
$\sigma_0 := \mathsf{com}$
$\mathsf{ch} \leftarrow H(\mathsf{pk}, M, \sigma_0)$
Parse ch as $\mathsf{ch} = (\mathsf{ch}^{(1)}, \mathsf{ch}^{(2)}, \ldots, \mathsf{ch}^{(r)})$, $\mathsf{ch}^{(j)} \in \mathsf{C}$
For $j \in \{1, \ldots, r\}$ **do**
  $\mathsf{resp}^{(j)} \leftarrow \mathcal{P}_1(\mathsf{state}^{(j)}, \mathsf{ch}^{(j)})$
$\mathsf{resp} := (\mathsf{resp}^{(1)}, \ldots, \mathsf{resp}^{(r)})$
$\sigma_1 := \mathsf{resp}$
Return $\sigma = (\sigma_0, \sigma_1)$

Fig. 4.2: Fiat-Shamir signature generation

$\mathsf{Vf}(\mathsf{pk}, \sigma, M)$

Parse $\sigma = (\sigma_0, \sigma_1)$
Parse σ_0 as $\sigma_0 = (\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(r)})$
$\mathsf{ch} \leftarrow H(\mathsf{pk}, M, \sigma_0)$
Parse ch as $\mathsf{ch} = (\mathsf{ch}^{(1)}, \mathsf{ch}^{(2)}, \ldots, \mathsf{ch}^{(r)})$, $\mathsf{ch}^{(j)} \in \mathsf{C}$
Parse σ_1 as $\sigma_1 = (\mathsf{resp}^{(1)}, \ldots, \mathsf{resp}^{(r)})$
For $j \in \{1, \ldots, r\}$ **do**
  $b^{(j)} \leftarrow \mathsf{Vf}_{\mathsf{IDS}}(\mathsf{pk}, \mathsf{com}^{(j)}, \mathsf{ch}^{(j)}, \mathsf{resp}^{(j)})$
$b \leftarrow b^{(1)} \wedge b^{(2)} \wedge \cdots \wedge b^{(r)}$
Return b

Fig. 4.3: Fiat-Shamir signature verification

4.2 Security of the Fiat-Shamir Transform

The security of the Fiat-Shamir transform has been investigated for over two decades. The first security proof of the transform was given in the seminal paper of Pointcheval and Stern [49]. They showed that assuming honest-verifier zero knowledge and special soundness of the identification scheme, Construction 5.1 gives EU-CMA secure signatures. Their proof is in the random oracle model and is based on the (now famous) forking lemma. The two main techniques introduced in the forking lemma are rewinding of the adversary and adaptively programming the random oracle. While these have proven to be quite powerful techniques in ROM reductions, they come with a drawback - the reduction is not tight - there is loss of factor the number of adversary's random oracle queries.

Later, Ohta and Okamoto [47] provide a different proof using a modular technique and similar assumptions on the identification scheme. Abdalla *et al.*[1] show that a signature

via the Fiat-Shamir transform is EU-CMA if and only if the identification scheme is IMP-PA, thus minimizing the needed assumption on the identification scheme.

Lately, the main two techniques from the forking lemma cause even more problems in the quantum-accessible random oracle model (QROM) and showing the Fiat-Shamir transform secure in the QROM has proven to be a tedious task (see [2, 23, 55]). The only known way to show security of the Fiat-Shamir transform in the QROM setting [23] is using oblivious commitments. Here the need for rewinding is replaced by introducing an additional trapdoor assumption on the commitments, which in itself is a very strong and problematic assumption.

Very recently, multi-user security of the Fiat-Shamir transform has been investigated in [39]. Although thle authors of [39] attempt to provide a more general framework for multi-user security (previously, tight results have been obtained only for Schnorr like signatures [10]), the assumptions on the IDS still seem too strong to be applicable on many post-quantum schemes. Thus for now, the only general result remains the one from [31], with a loss of factor - the number of users in the scenario.

5

The Fiat-Shamir Transform for 5-pass Identification Schemes

For several intractability assumptions, the most efficient IDS are five pass, i.e. IDS where a transcript consists of five messages. Here, efficiency refers to the size of all communication of sufficient rounds to make the soundness error negligible. This becomes especially relevant when one wants to turn an IDS into a signature scheme as it is closely related to the signature size of the resulting scheme.

As said in the Preliminaries (Chapter 1), the most common 5-pass identification schemes in the literature are those with challenge spaces C_1 and C_2 restricted to q and 2 respectively, that we called $q2$-Identification Schemes. In this chapter, we restrict our attention to such schemes and describe a transformation from passively secure $q2\text{-}IDS$ to unforgeable signatures. The transformation is a direct generalization of the Fiat-Shamir transform for 3-pass schemes (see Chapter 4). The description and security argument follow closely the one from [16], where we first introduced MQDSS.

Note that the Fiat-Shamir transform can be generalized to more general canonical $2n+1$ schemes with non-binary challenge spaces. However, this makes the description and proofs unnecessarily complex, especially because such schemes are not at all common in the literature.

5.1 A Fiat-Shamir transform for $q2$-Identification Schemes

As in the previous chapter, let $\mathsf{IDS}^r = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}^r, \mathcal{V}^r)$ denote the parallel composition of r rounds of the identification scheme $\mathsf{IDS} = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}, \mathcal{V})$.

Construction 5.1 (Fiat-Shamir transform for $q2$-Identification Schemes) *Let $k \in \mathbb{N}$ be the security parameter and let $\mathsf{IDS} = (\mathsf{KGen}_{\mathsf{IDS}}, \mathcal{P}, \mathcal{V})$, where $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2)$, $\mathcal{V} = (\mathsf{ChS}_1, \mathsf{ChS}_2, \mathsf{Vf}_{\mathsf{IDS}})$ be a $q2$-Identification Scheme that achieves soundness with soundness error κ. Select r, the number of (parallel) rounds of IDS, such that $\kappa^r = \mathrm{negl}(k)$, and that the challenge spaces C_1^r and C_2^r of the composition IDS^r, have exponential size in k. Moreover, select two cryptographic hash functions $H_1 : \{0,1\}^* \to \mathsf{C}_1^r$, $H_2 : \{0,1\}^* \to \mathsf{C}_2^r$.*

A $q2$-signature scheme $q2\text{-}\mathsf{Dss}(1^k)$ is a triplet of algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ defined as in Figures 5.1,5.1 and 5.3.

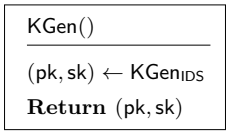


Fig. 5.1: $q2$-signature scheme: Key generation

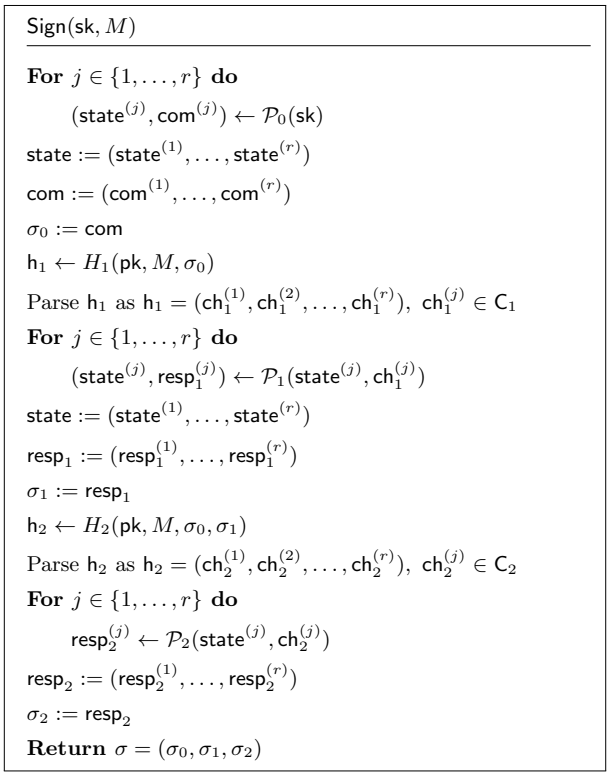


Fig. 5.2: $q2$-signature scheme: Signature generation

The correctness of the scheme follows immediately from the correctness of the IDS.

5.2 Security of $q2$-signature schemes.

The security of the above transform was proven in [16]. The proof is in the random oracle model and builds on techniques introduced by Pointcheval and Stern [49]. Namely, we generalized the well known Forking Lemma and showed that a particular type of rewinding of the adversary together with adaptive programming of the random oracles is useful for showing EU-CMA security. For completeness of this document, we include the complete security reduction in Appendix A.

In summary, the following theorem holds for $q2$-signature schemes:

Theorem 5.2 (EU-CMA security of $q2$-signature schemes [16]). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k)$ a $q2$-IDS that has a key relation R, is KOW secure, is honest-verifier zero-knowledge, and has a $q2$-extractor $\mathcal{E}$. Then $q2\text{-}\mathsf{Dss}(1^k)$, the $q2$-signature scheme derived applying Construction 5.1 is existentially unforgeable under adaptive chosen message attacks.*

$\mathsf{Vf}(\mathsf{pk}, \sigma, M)$

Parse $\sigma = (\sigma_0, \sigma_1, \sigma_2)$
Parse σ_0 as $\sigma_0 = (\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(r)})$
$\mathsf{h}_1 \leftarrow H_1(\mathsf{pk}, M, \sigma_0)$
Parse h_1 as $\mathsf{h}_1 = (\mathsf{ch}_1^{(1)}, \mathsf{ch}_1^{(2)}, \ldots, \mathsf{ch}_1^{(r)}),\ \mathsf{ch}_1^{(j)} \in \mathsf{C}_1$
Parse σ_1 as $\sigma_1 = (\mathsf{resp}_1^{(1)}, \ldots, \mathsf{resp}_1^{(r)})$
$\mathsf{h}_2 \leftarrow H_2(\mathsf{pk}, M, \sigma_0, \sigma_1)$
Parse h_2 as $\mathsf{h}_2 = (\mathsf{ch}_2^{(1)}, \mathsf{ch}_2^{(2)}, \ldots, \mathsf{ch}_2^{(r)}),\ \mathsf{ch}_2^{(j)} \in \mathsf{C}_2$
Parse σ_2 as $\sigma_2 = (\mathsf{resp}_2^{(1)}, \ldots, \mathsf{resp}_2^{(r)})$
For $j \in \{1, \ldots, r\}$ **do**
    $b^{(j)} \leftarrow \mathsf{Vf}_{\mathsf{IDS}}(\mathsf{pk}, \mathsf{com}^{(j)}, \mathsf{ch}_1^{(j)}, \mathsf{resp}_1^{(j)}, \mathsf{ch}_2^{(j)}, \mathsf{resp}_2^{(j)})$
$b \leftarrow b^{(1)} \wedge b^{(2)} \wedge \cdots \wedge b^{(r)}$
Return b

Fig. 5.3: $q2$-signature scheme: Signature verification

Part II

MQDSS Specifications

6

Notations

Throughout this part we will use the standard mathematical notations introduced in Section 1.1. In addition, in Chapter 9 we will use the following notations:

- $a + b$ - sum of a and b
- $a - b$ - difference of a and b
- $a \cdot b$ - product of a and b
- a/b - quotient of a and b
- $\log_2 a$ - logarithm to the base 2 of a
- $a \mod b$ - the non-negative remainder of the integer division of a and b
- $a|b$ - a is a divisor of b
- $\lceil a \rceil$ - ceiling function, returns the smallest integer greater than or equal to a
- $\lfloor a \rfloor$ - floor function, returns the greatest integer larger than or equal to a

- $a \leftarrow b$ - assignment operator, a takes the value of b
- $a \ll b$ - logical left shift, with b being non-negative integer. It is equivalent to $a \cdot 2^b$
- $a \wedge b$ - logical and operator
- $a == b$ - logical equal test, returns true (1) if $a = b$ and false (0) if $a \neq b$
- $a <> b$ - logical non-equal test, returns true (1) if $a \neq b$ and false (0) if $a = b$

- [ ] - empty array of bytes or bits
- $a[i]$ - the i-th element of the array a. The indexing of elements starts from 0.
- $[f(j)|j = 0..n-1]$ - array with elements $f(j)$, when j is iterated from 0 through $n-1$
- $\text{len}(a)$ - returns the length of a (the number of elements in a if we consider it as an array)
- $a||b$ - concatenation of a and b. If we look at $a = [a[0], a[1], \ldots, a[l_a]]$ and $b = [b[0], b[1], \ldots, b[l_2]]$ as arrays, then $a||b$ is the array $[a[0], a[1], \ldots, a[l_a], b[0], b[1], \ldots, b[l_2]]$
- $\text{append}(a, b)$ - appends the element b to the end of the array a
- $\text{subarray}(a, b, c)$ - returns the subarray of a from index b to $c-1$, i.e. returns $[a[b], \ldots, a[c-1]]$
- $\text{trunc}(a, b)$ - truncates the b least significant bits of a, with a being a bit-array and b a non-negative integer. If we look at a in its array representation $a = [a[0], a[1], \ldots, a[l_a]]$, then $\text{trunc}(a, b)$=$[a[0], a[1], \ldots, a[b-1]]$. It is equivalent to $\text{subarray}(a, 0, b)$, for a bit-array a.
- $\text{trim}(a, b)$ - truncates the b most significant bits of a, with a being a bit-array and b a non-negative integer. It is equivalent to $\text{subarray}(a, \text{len}(a) - b, \text{len}(a))$, for a bit-array a.
- $\text{mask}(a, b, c)$ - sets the bits $a[b], \ldots, a[c]$ to 0

- SHAKE256(*seed*, 136) - interface for the digest of SHAKE256 on input *seed* as standardized in FIPS 202, the SHA-3 standard [45].
- SHAKE256absorb(*seed*) - interface for the absorb phase of SHAKE256 for extendable output
- SHAKE256squeeze(*state*)- interface for the squeeze phase of SHAKE256 for extendable output. To obtain extendable output, repeatedly call SHAKE256squeeze(*state*) until needed

7

MQDSS High Level Description

In this chapter, we define the signature scheme MQDSS-q-n in generic terms by describing the required parameters, the functions KGen, Sign and Vf, and the necessary auxiliary functions. In Chapter 8 we will provide concrete parameters and in Chapter 9 we provide a detailed instantiations of the auxiliary functions. A detailed low-level description will be given in Chapter 9.

MQDSS-q-n is a digital signature scheme consisting of three algorithms KGen, Sign and Vf, defined in Sections 7.2, 7.3 and 7.4. The global parameters and auxiliary functions are defined in Section 7.1.

7.1 MQDSS Parameters Description and Auxiliary Functions

Let k be the security parameter.
MQDSS-q-n uses the following additional parameters:

- A positive integer $n \in \mathbb{N}$ - the number of variables and equations of the system $\mathbf{F}$,
- A positive integer $q \in \mathbb{N}$ (a prime or a prime power) - the order of the finite field $\mathbb{F}_q$,
- A positive integer $r \in \mathbb{N}$ - the number of rounds. This parameter is normally $r = \left\lceil k / \log_2 \frac{2q}{q+1} \right\rceil$, but it can also be chosen independently.

MQDSS-q-n uses the following auxiliary functions:

- A pseudorandom generator $\mathrm{PRG}_{\mathsf{sk}} : \{0,1\}^k \to \{0,1\}^{3k}$ used to randomly generate three seeds.
- A pseudorandom generator $\mathrm{PRG}_{\mathbf{s}} : \{0,1\}^k \to \{0,1\}^{n\lceil \log_2 q \rceil}$ used to randomly generate the secret key.
- A pseudorandom generator $\mathrm{PRG}_{\mathbf{rte}} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^{3rn\lceil \log_2 q \rceil}$ used to generate pseudorandom values during the signature generation.
- An extendable output function $\mathrm{XOF}_{\mathbf{F}} : \{0,1\}^k \to \{0,1\}^{F_{len}}$, where for $q = 2$, $F_{len} = n \cdot (\frac{n \cdot (n-1)}{2} + n)$ and for $q > 2$, $F_{len} = n \cdot (\frac{n \cdot (n+1)}{2} + n) \lceil \log_2 q \rceil$. This function is used for generating a multivariate system $\mathbf{F}$ by expanding a seed outputted by $\mathrm{PRG}_{\mathsf{sk}}$.
- Three cryptographic hash functions $\mathcal{H} : \{0,1\}^* \to \{0,1\}^k$, $H_1 : \{0,1\}^* \to {\mathbb{F}_q}^r$, and $H_2 : \{0,1\}^* \to \{0,1\}^r$.
- A string commitment function $Com_0 : {\mathbb{F}_q}^n \times {\mathbb{F}_q}^n \times {\mathbb{F}_q}^n \to \{0,1\}^k$ and
- A string commitment function $Com_1 : {\mathbb{F}_q}^n \times {\mathbb{F}_q}^n \to \{0,1\}^k$,

7.2 MQDSS Key Generation

The MQDSS-q-n key generation algorithm formally samples a $\mathcal{MQ}$ relation. Practically, the algorithm is realized as shown in Figure 7.1.

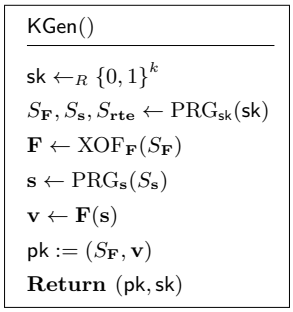


Fig. 7.1: MQDSS-q-n key generation

In more detail, given the security parameter k, the key generation algorithm $\mathsf{KGen}()$ performs the following operations:

- Randomly sample a secret key of k bits $\mathsf{sk} \leftarrow_R \{0,1\}^k$.
- Use the secret key sk as input (seed) to $\mathrm{PRG}_{\mathsf{sk}}$ to derive the following values:
  - $S_{\mathbf{F}}$, a seed of k bits from which the system parameter $\mathbf{F}$ is expanded;
  - $S_{\mathbf{s}}$, a seed of k bits from which the secret input to the $\mathcal{MQ}$ function is generated;
  - $S_{\mathbf{rte}}$, a seed of k bits that is used to sample all vectors $\mathbf{r}_0^{(i)}$, $\mathbf{t}_0^{(i)}$ and $\mathbf{e}_0^{(i)}$, $i \in \{1, \ldots, r\}$. Note that this seed is not yet needed during key generation, but is required during signing.
- Expand the seed $S_{\mathbf{F}}$ using $\mathrm{XOF}_{\mathbf{F}}$ to a F_{len} bits long string, where for $q = 2$, $F_{len} = n \cdot (\frac{n \cdot (n-1)}{2} + n)$ and for $q > 2$, $F_{len} = n \cdot (\frac{n \cdot (n+1)}{2} + n) \lceil \log_2 q \rceil$. Parse the pseudorandom string as an $\mathcal{MQ}$ system $\mathbf{F} \in \mathcal{MQ}(n, n, \mathbb{F}_q)$.
- Use the seed $S_{\mathbf{s}}$ as input to the $\mathrm{PRG}_{\mathbf{s}}$ to obtain $\mathbf{s}$, a string of length $n \lceil \log_2 q \rceil$ bits, that will be used as the secret input to the $\mathcal{MQ}$ function;
- Parse $\mathbf{s}$ as a vector $\mathbf{s} \in \mathbb{F}_q^n$, and evaluate the $\mathcal{MQ}$ system $\mathbf{F}(\mathbf{s})$ to obtain the vector $\mathbf{v} \in \mathbb{F}_q^n$.
- Set $\mathsf{pk} := (S_{\mathbf{F}}, \mathbf{v})$ as the public key.
- Return the public/secret key pair (pk, sk). The public pk is of length $k + n \lceil \log_2 q \rceil$ bits, and the secret sk of length k bits.

The obtained public key pk is of length $k + n \lceil \log_2 q \rceil$ bits, and the secret key sk of length k bits.

7.3 MQDSS Signature Generation

For the MQDSS-q-n signing procedure $\mathsf{Sign}()$, we assume as input a message $M \in \{0,1\}^*$ and a secret key sk. The signing procedure is given in Figure 7.2.

$\mathsf{Sign}(\mathsf{sk}, M)$

$S_{\mathbf{F}}, S_{\mathbf{s}}, S_{\mathbf{rte}} \leftarrow \mathrm{PRG}_{\mathsf{sk}}(\mathsf{sk})$
$\mathbf{F} \leftarrow \mathrm{XOF}_{\mathbf{F}}(S_{\mathbf{F}})$
$\mathbf{s} \leftarrow \mathrm{PRG}_{\mathbf{s}}(S_{\mathbf{s}})$
$\mathsf{pk} := (S_{\mathbf{F}}, \mathbf{F}(\mathbf{s}))$
$R \leftarrow \mathcal{H}(\mathsf{sk}||M)$
$D \leftarrow \mathcal{H}(\mathsf{pk}||R||M)$
$\mathbf{r}_0^{(1)}, \ldots, \mathbf{r}_0^{(r)}, \mathbf{t}_0^{(1)}, \ldots, \mathbf{t}_0^{(r)}, \mathbf{e}_0^{(1)}, \ldots, \mathbf{e}_0^{(r)} \leftarrow \mathrm{PRG}_{\mathbf{rte}}(S_{\mathbf{rte}}, D)$
For $j \in \{1, \ldots, r\}$ **do**
  $\mathbf{r}_1^{(j)} \leftarrow \mathbf{s} - \mathbf{r}_0^{(j)}$
  $c_0^{(j)} \leftarrow Com_0(\mathbf{r}_0^{(j)}, \mathbf{t}_0^{(j)}, \mathbf{e}_0^{(j)})$
  $c_1^{(j)} \leftarrow Com_1(\mathbf{r}_1^{(j)}, \mathbf{G}(\mathbf{t}_0^{(j)}, \mathbf{r}_1^{(j)}) + \mathbf{e}_0^{(j)})$
  $\mathsf{com}^{(j)} := (c_0^{(j)}, c_1^{(j)})$
$\sigma_0 \leftarrow \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}||\ldots||\mathsf{com}^{(r)})$
$\mathsf{ch}_1 \leftarrow H_1(D, \sigma_0)$
Parse ch_1 as $\mathsf{ch}_1 = (\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(r)}), \alpha^{(j)} \in \mathbb{F}_q$
For $j \in \{1, \ldots, r\}$ **do**
  $\mathbf{t}_1^{(j)} \leftarrow \alpha^{(j)}\mathbf{r}_0^{(j)} - \mathbf{t}_0^{(j)}, \quad \mathbf{e}_1^{(j)} \leftarrow \alpha^{(j)}\mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_0^{(j)}$
  $\mathsf{resp}_1^{(j)} := (\mathbf{t}_1^{(j)}, \mathbf{e}_1^{(j)})$
$\sigma_1 \leftarrow (\mathsf{resp}_1^{(1)}||\mathsf{resp}_1^{(2)}||\ldots||\mathsf{resp}_1^{(r)})$
$\mathsf{ch}_2 \leftarrow H_2(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$
Parse ch_2 as $\mathsf{ch}_2 = (b^{(1)}, b^{(2)}, \ldots, b^{(r)}), b^{(j)} \in \{0, 1\}$
For $j \in \{1, \ldots, r\}$ **do**
  $\mathsf{resp}_2^{(j)} \leftarrow \mathbf{r}_{b^{(j)}}^{(j)}$
$\sigma_2 \leftarrow (\mathsf{resp}_2^{(1)}||\mathsf{resp}_2^{(2)}||\ldots||\mathsf{resp}_2^{(r)}||c_{1-b^{(1)}}^{(1)}||c_{1-b^{(2)}}^{(2)}||\ldots||c_{1-b^{(r)}}^{(r)})$
Return $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$

Fig. 7.2: MQDSS-q-n signature generation

In more details, the signer:

- First effectively repeats the $\mathsf{KGen}()$ procedure i.e.,
  - derives $S_{\mathbf{F}}, S_{\mathbf{s}}, S_{\mathbf{rte}}$ from $\mathrm{PRG}_{\mathsf{sk}}(\mathsf{sk})$,
  - exapnds $\mathbf{F} = \mathrm{XOF}_{\mathbf{F}}(S_{\mathbf{F}})$ and $\mathbf{s} = \mathrm{PRG}_{\mathbf{s}}(S_{\mathbf{s}})$ and
  - derives the public key $\mathsf{pk} := (S_{\mathbf{F}}, \mathbf{F}(\mathbf{s}))$.
- Derives a message dependent random value $R = \mathcal{H}(\mathsf{sk} \parallel M)$.
- Using this random value R, the signer computes the randomized message digest $D = \mathcal{H}(R \parallel m)$. The value R must be included in the signature, so that a verifier can derive the same randomized digest.
- The signer then uses the pseudorandom generator $\mathrm{PRG}_{\mathbf{rte}}$ to sample the vectors $\mathbf{r}_0^{(1)}, \ldots, \mathbf{r}_0^{(r)}$, $\mathbf{t}_0^{(1)}, \ldots, \mathbf{t}_0^{(r)}$, $\mathbf{e}_0^{(1)}, \ldots, \mathbf{e}_0^{(r)}$ from $S_{\mathbf{rte}}$ and D.
- For each $j \in \{1, \ldots, r\}$
  - Computes $\mathbf{r}_1^{(j)}$ as the difference $\mathbf{s} - \mathbf{r}_0^{(j)}$,

  - Commits to $(\mathbf{r}_0^{(j)}, \mathbf{t}_0^{(j)}, \mathbf{e}_0^{(j)})$ and to $(\mathbf{r}_1^{(j)}, \mathbf{G}(\mathbf{t}_0^{(j)}, \mathbf{r}_1^{(j)}) + \mathbf{e}_0^{(j)})$ applying the commitment functions Com_0 and Com_1 respectively, to obtain $c_0^{(j)}$ and $c_1^{(j)}$ respectively,
  - Sets $\mathsf{com}^{(j)} := (c_0^{(j)}, c_1^{(j)})$.
- Computes the second part of the signature σ_0 as a digest over the concatenation of all commitments $\sigma_0 \leftarrow \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}||\ldots||\mathsf{com}^{(r)})$,
- Derives the first challenge $\mathsf{ch}_1 = (\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(r)})$ by applying H_1 to (D, σ_0).
- Using the $\alpha^{(j)}$ as individual challenges per round, the signer computes $\mathbf{t}_1^{(j)} \leftarrow \alpha^{(j)}\mathbf{r}_0^{(j)} - \mathbf{t}_0^{(j)}$ and $\mathbf{e}_1^{(j)} \leftarrow \alpha^{(j)}\mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_0^{(j)}$ for all $j \in \{1, \ldots, r\}$,
- Sets the responses $\mathsf{resp}_1^{(j)} := (\mathbf{t}_1^{(j)}, \mathbf{e}_1^{(j)})$ for all $j \in \{1, \ldots, r\}$.
- The concatenation of all responses $\mathsf{resp}_1^{(j)}$ gives the third part of the signature $\sigma_1 \leftarrow (\mathsf{resp}_1^{(1)}||\mathsf{resp}_1^{(2)}||\ldots||\mathsf{resp}_1^{(r)})$.
- The signer computes ch_2 by applying H_2 to the tuple $(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$ and parses it as r binary challenges $b^{(j)} \in \{0, 1\}$.
- For all $j \in \{1, \ldots, r\}$, the signer computes the second responses $\mathsf{resp}_2^{(j)} \leftarrow \mathbf{r}_{b^{(j)}}^{(j)}$.
- Finally, the signer computes the last part of the signature as $\sigma_2 \leftarrow (\mathsf{resp}_2^{(1)}||\mathsf{resp}_2^{(2)}||\ldots||\mathsf{resp}_2^{(r)}||c_{1-b^{(1)}}^{(1)}||c_{1-b^{(2)}}^{(2)}||\ldots||c_{1-b^{(r)}}^{(r)})$, and
- Outputs the signature $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$.

The complete signature is of the following form:

$$\sigma = (R, \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}||\ldots||\mathsf{com}^{(r)}), (\mathsf{resp}_1^{(1)}||\mathsf{resp}_1^{(2)}||\ldots||\mathsf{resp}_1^{(r)}), (\mathsf{resp}_2^{(1)}||\mathsf{resp}_2^{(2)}||\ldots||\mathsf{resp}_2^{(r)}||c_{1-b^{(1)}}^{(1)}||c_{1-b^{(2)}}^{(2)}||\ldots||c_{1-b^{(r)}}^{(r)}))$$

As each element of $\mathbb{F}_q$ requires $\lceil \log_2 q \rceil$ bits, the size of the signature is $(2+r)k+3rn \lceil \log_2 q \rceil$ bits.

7.4 MQDSS Signature Verification

Upon receiving a message M, a signature $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$, and a public key $\mathsf{pk} = (S_{\mathbf{F}}, \mathbf{v})$, the verifier performs the verification routine as listed in Figure 7.3.

In more detail, the main goal of the verification process is to reconstruct the missing commitments, and calculate a value σ_0' that will be verified against the inputted σ_0. The whole procedure is as follows:

- Using the pubic key $\mathsf{pk} = (S_{\mathbf{F}}, \mathbf{v})$ and the value R from the signature σ, compute the system parameter $\mathbf{F} \leftarrow \mathrm{XOF}_{\mathbf{F}}(S_{\mathbf{F}})$ and the randomized message digest $D \leftarrow \mathcal{H}(\mathsf{pk}||R||M)$.
- Since the signature contains σ_0, compute the first challenge ch_1 as $\mathsf{ch}_1 \leftarrow H_1(D, \sigma_0)$ and parse it as $ch_1 = (\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(r)})$, $\alpha^{(j)} \in \mathbb{F}_q$
- Next, compute the challenge $\mathsf{ch}_2 \leftarrow H_2(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$, from the two parts σ_0, σ_1 of the signature and the computed ch_1 in the previous step. Parse it as $\mathsf{ch}_2 = (b^{(1)}, b^{(2)}, \ldots, b^{(r)})$, $b^{(j)} \in \{0, 1\}$.
- Parse the two signature parts σ_1 and σ_2 as $\sigma_1 = (\mathsf{resp}_1^{(1)}||\mathsf{resp}_1^{(2)}||\ldots||\mathsf{resp}_1^{(r)})$ and $\sigma_2 = (\mathsf{resp}_2^{(1)}||\mathsf{resp}_2^{(2)}||\ldots||\mathsf{resp}_2^{(r)}||c_{1-b^{(1)}}^{(1)}||c_{1-b^{(2)}}^{(2)}||\ldots||c_{1-b^{(r)}}^{(r)})$ respectively.

$\mathsf{Vf}(\mathsf{pk}, \sigma, M)$

$\mathbf{F} \leftarrow \mathrm{XOF}_{\mathbf{F}}(S_{\mathbf{F}})$
$D \leftarrow \mathcal{H}(\mathsf{pk}||R||M)$
$\mathsf{ch}_1 \leftarrow H_1(D, \sigma_0)$
Parse ch_1 as $\mathsf{ch}_1 = (\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(r)}), \alpha^{(j)} \in \mathbb{F}_q$
$\mathsf{ch}_2 \leftarrow H_2(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$
Parse ch_2 as $\mathsf{ch}_2 = (b^{(1)}, b^{(2)}, \ldots, b^{(r)}), b^{(j)} \in \{0, 1\}$
Parse σ_1 as $\sigma_1 = (\mathsf{resp}_1^{(1)}||\mathsf{resp}_1^{(2)}|| \ldots ||\mathsf{resp}_1^{(r)})$
Parse σ_2 as $\sigma_2 = (\mathsf{resp}_2^{(1)}||\mathsf{resp}_2^{(2)}|| \ldots ||\mathsf{resp}_2^{(r)}||c_{1-b^{(1)}}^{(1)}||c_{1-b^{(2)}}^{(2)}|| \ldots ||c_{1-b^{(r)}}^{(r)})$
For $j \in \{1, \ldots, r\}$ **do**
  Parse $\mathsf{resp}_1^{(j)}$ as $\mathsf{resp}_1^{(j)} = (\mathbf{t}_1^{(j)}, \mathbf{e}_1^{(j)})$
  If $b^{(j)} == 0$
    $\mathbf{r}_0^{(j)} = \mathsf{resp}_2^{(j)}$
    $c_0^{(j)} \leftarrow Com_0(\mathbf{r}_0^{(j)}, \alpha^{(j)}\mathbf{r}_0^{(j)} - \mathbf{t}_1^{(j)}, \alpha^{(j)}\mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_1^{(j)})$
  else
    $\mathbf{r}_1^{(j)} = \mathsf{resp}_2^{(j)}$
    $c_1^{(j)} \leftarrow Com_1(\mathbf{r}_1^{(j)}, \alpha^{(j)}(\mathbf{v} - \mathbf{F}(\mathbf{r}_1^{(j)})) - \mathbf{G}(\mathbf{t}_1^{(j)}, \mathbf{r}_1^{(j)}) - \mathbf{e}_1^{(j)})$
  $\mathsf{com}^{(j)} := (c_0^{(j)}, c_1^{(j)})$
$\sigma_0' \leftarrow \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}|| \ldots ||\mathsf{com}^{(r)})$
Return $\sigma_0' == \sigma_0$

Fig. 7.3: MQDSS-q-n signature verification

- Since the verifier knows the values $b^{(j)}$ from the previous step, he knows which of the two parts of the commitments $\mathsf{com}^{(j)}$ were included in σ_2, and can now proceed to recovering the other, missing part. This is done for all $j \in \{1, \ldots, r\}$ as follows:
  - Parse $\mathsf{resp}_1^{(j)}$ to obtain $(\mathbf{t}_1^{(j)}, \mathbf{e}_1^{(j)})$, and
  - if $b^{(j)} == 0$ compute $c_0^{(j)}$ as $c_0^{(j)} \leftarrow Com_0(\mathbf{r}_0^{(j)}, \alpha^{(j)}\mathbf{r}_0^{(j)} - \mathbf{t}_1^{(j)}, \alpha^{(j)}\mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_1^{(j)})$, otherwise compute $c_1^{(j)}$ as $c_1^{(j)} \leftarrow Com_1(\mathbf{r}_1^{(j)}, \alpha^{(j)}(\mathbf{v} - \mathbf{F}(\mathbf{r}_1^{(j)})) - \mathbf{G}(\mathbf{t}_1^{(j)}, \mathbf{r}_1^{(j)}) - \mathbf{e}_1^{(j)})$
  - Set $\mathsf{com}^{(j)} := (c_0^{(j)}, c_1^{(j)})$
- Calculate $\sigma_0' \leftarrow \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}|| \ldots ||\mathsf{com}^{(r)})$ from the obtained commitments $\mathsf{com}^{(j)}$
- Return the truth value of $\sigma_0' == \sigma_0$. This means that for verification to succeed, $\sigma_0' = \sigma_0$ should hold.

8

Parameter Sets

8.1 Reference Parameter Sets

Recall (c.f. Section 7.1) that MQDSS-q-n is parametrized by the parameters:

- A positive integer $k \in \mathbb{N}$ - the security parameter,
- A positive integer $n \in \mathbb{N}$ - the number of variables and equations of the system $\mathbf{F}$,
- A positive integer $q \in \mathbb{N}$ (a prime or a prime power) - the order of the finite field $\mathbb{F}_q$,
- A positive integer $r \in \mathbb{N}$ - the number of rounds.

We propose the following two parameter sets as reference parameter sets of MQDSS:

- MQDSS-31-48
  - $k = 256$, $q = 31$, $n = 48$, $r = 269$;
- MQDSS-31-64
  - $k = 384$, $q = 31$, $n = 64$, $r = 403$.

The following Table 8.1 summarizes the basic characteristics of these two parameter sets.

	Security category	k	q	n	r	Public key size (bytes)	Secret key size (bytes)	Signature size (bytes)
MQDSS-31-48	1-2	256	31	48	269	62	32	32882
MQDSS-31-64	3-4	384	31	64	403	88	48	67800

Table 8.1: Basic characteristics of the refernce parameter sets

We have calculated the key sizes and the signature size based on the formulas provided in Section 7.1: The public key is of length $k + n\lceil \log_2 q \rceil$ bits, the secret key sk of length k bits and the signature of length $(2 + r)k + 3rn\lceil \log_2 q \rceil$ bits. The number of rounds r has been calculated as $r = \left\lceil k / \log_2 \frac{2q}{q+1} \right\rceil$.

We also summarize the strength of the reference parameter sets with respect to the best classical and quantum attacks (see Chapter 10). The summary is given in Table 8.2. For more detailed analysis of the algorithms see Chapter 2.

	Best classical attack		Best quantum attack		
	algorithm	Field op.	algorithm	Gates	Depth
MQDSS-31-48	HybridF5	2^{159}	Crossbread	2^{99}	2^{90}
MQDSS-31-64	HybridF5	2^{205}	Crossbread	2^{130}	2^{120}

Table 8.2: Best classical and quantum attacks against the reference parameter sets

As part of the submission package we provide reference (and additional) implementations for the two reference parameter sets MQDSS-31-48 and MQDSS-31-64. The details of the implementations are given in Chapter 9 and Chapter 15.

We emphasize that the chosen reference parameter sets are not the only that are suitable for use, and that should be considered by NIST and the broader community in the evaluation process. In the next section we provide additional parameter sets of comparable performance and security strength but over different fields. We decided to keep the original choice of the field as was initially proposed in [16], and to provide implementations only for parameters over $\mathbb{F}_{31}$. We justify our decision in the next section. To match the security levels identified by NIST, we changed the number of rounds in MQDSS-31-64 compared to [16] from 269 to 403 to match Category 3 and 4, and additionally proposed the lower security set MQDSS-31-48 (not in [16]). Furthermore, we decided not to include a parameter set for the Categories 5 and 6 defined by NIST. Over $\mathbb{F}_{31}$ this would be MQDSS-31-88 or even MQDSS-31-96. As can be seen from Table 8.3 this parameter set (or any for that mater in the category) has signature size in the range of 120KB, which we find a bit out of the practical range of parameters. Nevertheless, if NIST shows interest in such a parameter set, it is simple to extend our implementations and performance analysis to this set as well.

We continue the discussion about the additional parameter sets in the next Section.

8.2 Additional Parameter Sets

In addition to MQDSS-31-48 and MQDSS-31-64 we recommend additional parameter sets of comparable security strength i.e. Categories 1-4, as identified by NIST [46], but over different fields. We also provide parameter sets for the much higher security categories 5 and 6. Their basic characteristics and best attacks are given in Tables 8.3 and 8.4. It is important to note that for the additional recommended parameter sets, we do not provide implementation.

It can be noticed that within a security category, the parameter sets over $\mathbb{F}_{16}$ and $\mathbb{F}_{32}$ are of very similar performance characteristics as the reference parameter sets over $\mathbb{F}_{31}$. However, we decided not to include these in the reference parameter sets.

Our decision is based on two main observations: 1. The field $\mathbb{F}_{31}$ is the most natural choice with respect to implementation of the field arithmetic, since *any* platform already contains instructions for multiplication of natural numbers, but no instructions for $\mathbb{F}_{16}$ or $\mathbb{F}_{32}$. For these fields in general we would have to design specific representations for fast multiplication, or use table lookup intructions. 2. Our optimized implementation using AVX2 instructions is much faster over $\mathbb{F}_{31}$ than $\mathbb{F}_{16}$ or $\mathbb{F}_{32}$. Although, NIST does not require such an implementaion (at least at this stage of the standardisation process), we believe that in practice it is very relevant.

On the other hand, as $\mathbb{F}_{16}$ and $\mathbb{F}_{32}$ are binary fields, these parameter sets are particularly interesting for hardware implementations. Therefore for hardware, we recommend using them, particularly MQDSS-16-56 and MQDSS-16-72 rather than MQDSS-31-48 and MQDSS-31-64.

Moe generally, in the evaluation process we encourage NIST and the community to treat MQDSS-16-56, MQDSS-16-72, MQDSS-32-48 and MQDSS-32-64 with the same level of attention as the reference parameter sets.

Security category	k	q	n	r	Public key size (bytes)	Secret key size (bytes)	Signature size (bytes)
1-2	256	4	88	378	54	32	37108
1-2	256	16	56	281	60	32	32660
1-2	256	32	48	268	62	32	32760
1-2	256	64	40	262	62	32	32028
3-4	384	4	128	567	80	48	81744
3-4	384	16	72	421	84	48	65772
3-4	384	32	64	402	88	48	67632
3-4	384	64	64	393	102	48	82626
5-6	512	4	160	756	104	64	139232
5-6	512	16	96	562	112	64	117024
5-6	512	31	88	537	119	64	123101
5-6	512	32	88	536	119	64	122872
5-6	512	64	88	524	130	64	137416

Table 8.3: Basic characteristics of the additional parameter sets

Security category			Best classical attack		Best quantum attack		
	q	n	algorithm	Field op.	algorithm	Gates	Depth
1-2	4	88	Crossbread	2^{152}	Crossbread	2^{93}	2^{83}
1-2	16	56	Crossbread	2^{163}	Crossbread	2^{98}	2^{89}
1-2	32	48	HybridF5	2^{159}	Crossbread	2^{96}	2^{88}
1-2	64	40	HybridF5	2^{143}	Crossbread	2^{89}	2^{81}
3-4	4	128	Crossbread	2^{226}	Crossbread	2^{129}	2^{119}
3-4	16	72	HybridF5	2^{210}	Crossbread	2^{123}	2^{113}
3-4	32	64	HybridF5	2^{205}	Crossbread	2^{125}	2^{115}
3-4	64	64	HybridF5	2^{217}	Crossbread	2^{136}	2^{127}
5-6	4	160	Crossbread	2^{287}	Crossbread	2^{158}	2^{147}
5-6	16	96	HybridF5	2^{273}	Crossbread	2^{162}	2^{152}
5-6	31	88	HybridF5	2^{273}	Crossbread	2^{179}	2^{168}
5-6	32	88	HybridF5	2^{274}	Crossbread	2^{174}	2^{164}
5-6	64	88	HybridF5	2^{291}	Crossbread	2^{203}	2^{192}

Table 8.4: Best classical and quantum attacks against the additional parameter sets

9

Low Level Description of MQDSS

In Chapter 7, we described the MQDSS scheme in general terms. Here, we complete the specification by giving the byte-level details that should allow an implementer to write a compatible implementation.

This low level description will focus on our reference parameter sets as defined in Section 8.1. Thus, we will assume that $q = 31$, and the underlying field of operation is $\mathbb{F}_{31}$. To provide a slightly more general framework, we will define all functions in terms of parameters $k, n \in \mathbb{N}$, such that $64|k$ and $8|n$. (Such a description will allow application of the following detailed specifications not only to the reference parameter sets, but also to parameter sets over $\mathbb{F}_{31}$ of different security level.)

9.1 Auxiliary Functions

9.1.1 Secret Key Expansion

The secret key of MQDSS, denoted by sk is a $k/8$-byte string. It is used in the key generation process (see Section 7.2) and in the signing process (see Section 7.3) where it is first expanded to three separate values: $S_{\mathbf{F}}$, $S_{\mathbf{s}}$ and $S_{\mathbf{rte}}$. This is done by expanding to $3k/8$ bytes, and interpreting the first $k/8$ bytes as $S_{\mathbf{F}}$, the second $k/8$ as $S_{\mathbf{s}}$ and the last $k/8$ as $S_{\mathbf{rte}}$ (see Algorithm 1).

Algorithm 1 SecretKeyExpansion(sk)

Input: sk
$block \leftarrow \text{SHAKE256}(\mathsf{sk}, 136)$
$S_{\mathbf{F}} \leftarrow \text{subarray}(block, 0, k/8)$
$S_{\mathbf{s}} \leftarrow \text{subarray}(block, k/8, 2k/8)$
$S_{\mathbf{rte}} \leftarrow \text{subarray}(block, 2k/8, 3k/8)$
Output: $S_{\mathbf{F}}$, $S_{\mathbf{s}}$, $S_{\mathbf{rte}}$

9.1.2 Expanding $S_{\mathbf{F}}$, $S_{\mathbf{s}}$ and $S_{\mathbf{rte}}$

The functions $\text{XOF}_{\mathbf{F}}$, $\text{PRG}_{\mathbf{s}}$ and $\text{PRG}_{\mathbf{rte}}$ are instantiated using rejection sampling of the output of the extendable output function SHAKE256 standardized in FIPS 202, the SHA-3 standard [45]. The rejection sampling works as follows: For each output byte of SHAKE256, we ignore the most significant three bits. We discard the resulting value if it is equal to 31 when interpreted as an unsigned integer (i.e. all five bits are set). See Algorithm 2 for details.

Algorithm 2 RejectSample($seed, len$)

Input: $seed$, len
  $array31 \leftarrow [\,]$
  $state \leftarrow$ SHAKE256absorb($seed$)
  while len($array31$) $< len$ **do**
    $block \leftarrow$ SHAKE256squeeze($state$)
    $i \leftarrow 0$
    while $i <$len($block$) $\wedge$ len($array31$) $< len$ **do**
      $cand \leftarrow block[i]$
      if trunc($cand, 5$) <> 11111 **then**
        append($array31$,mask($cand, 5, 7$))
      end if
      $i \leftarrow i + 1$
    end while
  end while
Output: $array31$

Using Algorithm 2 we can easily expand all the necessary values. $\mathbf{F}$ is obtained by direct application of the RejectSample algorithm. The output elements are then interpreted as integers. The elements of $\mathbf{F}$ are in signed integers between -15 and 15, inclusive. We bring the randomly sampled integer to this domain by subtracting 15. Algorithm 3 is our wrapper for this function.

Algorithm 3 $\mathcal{MQ}$system($S_{\mathbf{F}}$)

Input: $S_{\mathbf{F}}$
  $\mathbf{F} \leftarrow$RejectSample($S_{\mathbf{F}}, n(\frac{n(n+1)}{2} + n)$)
  for $0 \leqslant i <$len($\mathbf{F}$) **do**
    $\mathbf{F}[i] \leftarrow \mathbf{F}[i] - 15$
  end for
Output: $\mathbf{F}$

The secret vector $\mathbf{s}$ is derived similarly, with the crucial difference that the secret key elements are not transformed to signed integers. The random elements for the vectors $\mathbf{r}_0$, $\mathbf{t}_0$ and $\mathbf{e}_0$ are derived from the seed $S_{\mathbf{rte}}$ in exactly the same way as the secret vector. Note that we derive all vectors of the same type for all rounds consecutively, i.e. $\mathbf{r}_0^{(1)}$, $\mathbf{r}_0^{(2)}$, $\mathbf{r}_0^{(3)}$, ..., $\mathbf{t}_0^{(1)}$, $\mathbf{t}_0^{(2)}$, ..., $\mathbf{e}_0^{(1)}$, $\mathbf{e}_0^{(2)}$, ... rather than $\mathbf{r}_0^{(1)}$, $\mathbf{t}_0^{(1)}$, $\mathbf{e}_0^{(1)}$, $\mathbf{r}_0^{(2)}$, $\mathbf{t}_0^{(2)}$, $\mathbf{e}_0^{(3)}$, ... (See Algorithm 4 and Algorithm 5).

Algorithm 4 SecretVector($S_{\mathbf{s}}$)

Input: $S_{\mathbf{s}}$
  $\mathbf{s} \leftarrow$RejectSample($S_{\mathbf{F}}, n$)
Output: $\mathbf{s}$

9.1.3 Evaluating F

At the core of the scheme lies evaluation of the $\mathbf{F}$ function, and its bilinear counterpart $\mathbf{G}$. The evaluation of $\mathbf{F}$ can roughly be divided in two parts: the generation of all quadratic terms, and computation of the resulting polynomials for given terms.

For the generation of the quadratic terms, we construct the terms in a variant of graded reverse lexicographic order. We note that for most platforms, this is not the most

Algorithm 5 RTEexpand($S_{\mathbf{rte}}, D$)

Input: $S_{\mathbf{rte}}$
  $array_{\mathbf{rte}} \leftarrow$RejectSample($S_{\mathbf{rte}}||D, 3rn$)
  $array_{\mathbf{r}} \leftarrow$subarray($array_{\mathbf{rte}}, 0, rn$)
  $array_{\mathbf{t}} \leftarrow$subarray($array_{\mathbf{rte}}, rn, 2rn$)
  $array_{\mathbf{e}} \leftarrow$subarray($array_{\mathbf{rte}}, 2rn, 3rn$)
  $\mathbf{r} \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
  $\mathbf{t} \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
  $\mathbf{e} \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
  for $0 \leqslant i < rn; i \leftarrow i + n$ **do**
    $\mathbf{r}[i] \leftarrow subarray(array_{\mathbf{r}}, i, i + n)$
    $\mathbf{t}[i] \leftarrow subarray(array_{\mathbf{t}}, i, i + n)$
    $\mathbf{e}[i] \leftarrow subarray(array_{\mathbf{e}}, i, i + n)$
  end for
Output: $\mathbf{r}$, $\mathbf{t}$, $\mathbf{e}$

efficient way to generate the quadratic terms [16], but it provides a reasonably straightforward method that has decent average performance. Adhering to the same order is crucial for implementations to be compatible, as this determines which elements of the system parameter coincide with which terms.

In order to somewhat accommodate platforms that have combined multiplication and addition instructions, (e.g. the `vpmaddubs` instruction on AVX2), we process pairs of quadratic terms rather than individual coefficients. This format is chosen to still be convenient to handle on platforms that cannot combine multiplications and additions. In particular, platforms with more traditional SIMD instructions can use unpack instructions to de-interleave the vectors.

We describe the process in pseudo-code below, see Algorithm 6. Note that this includes multiplication with elements in $\mathbf{F}$ over $\mathbb{F}_{31}$.

Algorithm 6 EvaluateF($\mathbf{u}, \mathbf{F}$)

Input: $\mathbf{u}, \mathbf{F}$
  $terms \leftarrow [\,]$
  for $0 \leqslant i < n$ **do**
    for $0 \leqslant j < i$ **do**
      append($terms, \mathbf{u}[i] \cdot \mathbf{u}[j] \mod 31$)
    end for
  end for
  $\mathbf{r} \leftarrow [0|j = 0..n - 1]$
  for $0 \leqslant i < n; i \leftarrow i + 2$ **do**
    for $0 \leqslant j < n$ **do**
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \mathbf{u}[i] \cdot \mathbf{F}[i \cdot n + 2 \cdot j] \mod 31$
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \mathbf{u}[i + 1] \cdot \mathbf{F}[i \cdot n + 2 \cdot j + 1] \mod 31$
    end for
  end for
  for $0 \leqslant i < \frac{n \cdot (n+1)}{2}; i \leftarrow i + 2$ **do**
    for $0 \leqslant j < n$ **do**
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \text{terms}[i] \cdot \mathbf{F}[n \cdot m + i \cdot m + 2 \cdot j] \mod 31$
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \text{terms}[i + 1] \cdot \mathbf{F}[n \cdot m + i \cdot m + 2 \cdot j + 1] \mod 31$
    end for
  end for
Output: $\mathbf{r} = \mathbf{F}(\mathbf{u})$

To evaluate the polar form function $\mathbf{G}$, we use almost the same procedure. For completeness, we list it in Algorithm 7. Notably, the differences are limited to a different term generation, and skipping of the square terms (as these cancel out).

Algorithm 7 EvaluateG($\mathbf{u}, \mathbf{v}, \mathbf{F}$)

Input: $\mathbf{u}, \mathbf{v}, \mathbf{F}$
  $terms \leftarrow [\,]$
  for $0 \leqslant i < n$ **do**
    for $0 \leqslant j < i$ **do**
      append($terms, \mathbf{u}[i] \cdot \mathbf{v}[j] + \mathbf{u}[j] \cdot \mathbf{v}[i] \mod 31$)
    end for
  end for
  $\mathbf{r} \leftarrow [0|j = 1..n]$
  for $0 \leqslant i < \frac{n \cdot (n+1)}{2}$; $i \leftarrow i + 2$ **do**
    for $0 \leqslant j < n$ **do**
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \text{terms}[i] \cdot \mathbf{F}[n \cdot m + i \cdot m + 2 \cdot j] \mod 31$
      $\mathbf{r}[j] \leftarrow \mathbf{r}[j] + \text{terms}[i+1] \cdot \mathbf{F}[n \cdot m + i \cdot m + 2 \cdot j + 1] \mod 31$
    end for
  end for
Output: $\mathbf{r} = \mathbf{G}(\mathbf{u}, \mathbf{v})$

9.1.4 Packing and unpacking $\mathbb{F}_{31}$ elements

All field elements included in the signature are stored in packed representation. This means that, when storing a vector of $\mathbb{F}_{31}$ elements, each element is expressed using five bits, representing the element using its smallest non-negative representation as an integer. The first byte of the byte sequence represents the first five bits of the first element, and the three least-significant bits of the next element. The next byte contains the remaining two high bits of the second element, the complete third element, and the least-significant bit of the forth element, etc. Note that for all parameters of MQDSS, we have restricted the value of n to be a multiple of 8. Thus, there is no need to explicitly specify padding, since this will result in byte arrays of exact multiples of eight bits. A vector of elements in $\mathbb{F}_{31}$ is unpacked by applying the inverse of the above operation (see Algorithms 8 and 9).

Algorithm 8 PackArray31($\mathbf{u}$)

Input: $\mathbf{u}$
  $bitstring \leftarrow [\,]$
  for $0 \leqslant i < len(\mathbf{u})$ **do**
    $bitstring \leftarrow bitstring$||trunc($\mathbf{u}$[i],5)
  end for
  $bytearray \leftarrow [\,]$
  for $0 \leqslant i <$len($bitstring$); $i \leftarrow i + 8$ **do**
    append($bytearray$,subarray($bitstring, i, i + 8$))
  end for
Output: $bytearray$

9.1.5 Commitment and hash functions

The commitments Com_0 and Com_1 and the Hash functions $\mathcal{H}$, H_1 and H_2 are instantiated also using SHAKE256. They take as input arrays of $\mathbb{F}_{31}$ elements that need to be in packed

Algorithm 9 UnpackArray31(*bytearray*)

Input: *bytearray*
  bitstring ← [ ]
  for $0 \leqslant i < len(bytearray)$ **do**
    $bitstring \leftarrow bitstring||bytearray[i]$
  end for
  u ← [ ]
  for $0 \leqslant i <$len(*bitstring*); $i \leftarrow i + 5$ **do**
    append(**u**,subarray($bitstring, i, i + 5$)||000)
  end for
Output: **u**

form. The input to the commitment functions Com_0 and Com_1 is a sequence of three, respectively two packed byte arrays. These are simply concatenated bytewise, starting with the vector listed first. The same applies for the hash functions $\mathcal{H}$, H_1 and H_2. Their algorithmic description is given in Algorithms 10-14.

It should come as no surprise that the same rejection sampling method is applied to sample the challenges $\alpha^{(i)} \in \mathbb{F}_{31}$. After absorbing the transcript into the SHAKE state, it is repeatedly squeezed until sufficient elements have been extracted – as before, the least significant 5 bits are considered as an unsigned integer, and is rejected if it is equal to 31.

The binary challenges are obtained by enumerating the bits of the hash output per byte, from least to most significant. (see Algorithm 14.)

Algorithm 10 Com0($\mathbf{r}, \mathbf{t}, \mathbf{e}$)

Input: $\mathbf{r}, \mathbf{t}, \mathbf{e}$
  c_0 ← [ ]
  seed ← (PackArray31(**r**)||PackArray31(**t**)||PackArray31(**e**))
  state ← SHAKE256absorb(*seed*)
  block ← SHAKE256squeeze(*state*)
  c_0 ← subarray($block, 0, k/8$)
Output: c_0

Algorithm 11 Com1($\mathbf{r}, \mathbf{e}$)

Input: $\mathbf{r}, \mathbf{e}$
  c_1 ← [ ]
  seed ← (PackArray31(**r**)||PackArray31(**e**))
  state ← SHAKE256absorb(*seed*)
  block ← SHAKE256squeeze(*state*)
  c_1 ← subarray($block, 0, k/8$)
Output: c_1

Algorithm 12 Hash(*bytearray*)

Input: *bytearray*
  state ← SHAKE256absorb(*bytearray*)
  block ← SHAKE256squeeze(*state*)
  digest ← subarray($block, 0, k/8$)
Output: *digest*

Algorithm 13 Hash1(D, σ_0)

Input: D, σ_0
  $seed \leftarrow D||\sigma_0$
  $\mathsf{ch}_1 \leftarrow$RejectSample$[seed, r]$
Output: ch_1

Algorithm 14 Hash2$(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$

Input: $D, \sigma_0, \mathsf{ch}_1, \sigma_1$
  $seed \leftarrow D||\sigma_0$PackArray31$(\mathsf{ch}_1)||\sigma_1$
  $state \leftarrow$ SHAKE256absorb$(seed)$
  $block \leftarrow$ SHAKE256squeeze$(state)$
  $\mathsf{ch}_2 \leftarrow[\,]$
  for $0 \leqslant i < r$ **do**
    $temp = block[\text{floor}(i/8)]$
    append$(\mathsf{ch}_2, temp[i \mod 8])$
  end for
Output: ch_2

9.2 Putting it all together - Pseudo code of KGen,Sign,Vf

Using the defined auxiliary functions from the previous section, we can provide a low level algorithmic description of the defining algorithms of MQDSS - KGen,Sign,Vf (see Chapter 7).

For the KGen algorithm of MQDSS, we assume the existence of a function rand() that on input $k \in \mathbb{N}$ outputs $k/8$ bytes of strong randomness. It is given in Algorithm 15.

Algorithm 15 KGen(k)

Input: k
  $\mathsf{sk} \leftarrow$rand(k)
  $S_{\mathbf{F}}$, $S_{\mathbf{s}}$, $S_{\mathbf{rte}} \leftarrow$ SecretKeyExpansion(sk)
  $\mathbf{F} \leftarrow \mathcal{MQ}$system$(S_{\mathbf{F}})$
  $\mathbf{s} \leftarrow$SecretVector$(S_{\mathbf{s}})$
  $\mathbf{v} \leftarrow$EvaluateF$(\mathbf{s}, \mathbf{F})$
  $\mathsf{pk} \leftarrow S_{\mathbf{F}}||$PackArray31$(\mathbf{v})$
Output: $(\mathsf{pk}, \mathsf{sk})$

An MQDSS signature is generated with the algorithm Sign (see Algorithm 16). It takes as input a secret key sk and a message to be signed M.

An MQDSS signature is verified using the algorithm Vf (see Algorithm 17). It takes as input a public key sk, a message M, and a signature σ.

Algorithm 16 $\mathsf{Sign}(\mathsf{sk}, M)$

Input: sk, M

$S_{\mathbf{F}}$, $S_{\mathbf{s}}$, $S_{\mathbf{rte}} \leftarrow$ SecretKeyExpansion(sk)
$\mathbf{F} \leftarrow \mathcal{MQ}$system($S_{\mathbf{F}}$)
$\mathbf{s} \leftarrow$SecretVector($S_{\mathbf{s}}$)
$\mathbf{v} \leftarrow$EvaluateF($\mathbf{s}, \mathbf{F}$)
$\mathsf{pk} \leftarrow S_{\mathbf{F}}||$PackArray31($\mathbf{v}$)
$R \leftarrow$Hash($\mathsf{sk}||M$)
$D \leftarrow$Hash($\mathsf{pk}||R||M$)
$\mathbf{r}_0, \mathbf{t}_0, \mathbf{e}_0 \leftarrow$ RTEexpand($S_{\mathbf{rte}}, D$)
$\mathbf{r}_1 \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
$\mathbf{t}_1 \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
$\mathbf{e}_1 \leftarrow [[0|i = 1 \dots n]|i = 1 \dots r]$
$c_0 \leftarrow [[0|i = 1 \dots k/8]|i = 1 \dots r]$
$c_1 \leftarrow [[0|i = 1 \dots k/8]|i = 1 \dots r]$
$\mathsf{com} \leftarrow [\,]$
for $0 \leqslant i < r$ **do**
  $\mathbf{r}_1[i] \leftarrow \mathbf{s} - \mathbf{r}_0[i]$
  $c_0[i] \leftarrow$Com0($\mathbf{r}_0[i], \mathbf{t}_0[i], \mathbf{e}_0[i]$)
  $c_1[i] \leftarrow$Com1($\mathbf{r}_1[i]$,EvaluateG($\mathbf{t}_0[i], \mathbf{r}_1[i], \mathbf{F}$) + $\mathbf{e}_0[i]$)
  $\mathsf{com} \leftarrow \mathsf{com}||$PackArray31($c_0[i]$)||PackArray31($c_1[i]$)
end for
$\sigma_0 \leftarrow$Hash(com)
$\mathsf{ch}_1 \leftarrow$Hash1(D, σ_0)
$\sigma_1 \leftarrow [\,]$
for $0 \leqslant i < r$ **do**
  $\mathbf{t}_1[i] \leftarrow \mathsf{ch}_1[i] \cdot \mathbf{r}_0[i] - \mathbf{t}_0[i]$
  $\mathbf{e}_1[i] \leftarrow \mathsf{ch}_1[i] \cdot$EvaluateF($\mathbf{r}_0[i], \mathbf{F}$)$-\mathbf{e}_0[i]$
  $\sigma_1 \leftarrow \sigma_1||$PackArray31($\mathbf{t}_1[i]$)||PackArray31($\mathbf{e}_1[i]$)
end for
$\mathsf{ch}_2 \leftarrow$Hash2($D, \sigma_0, \mathsf{ch}_1, \sigma_1$)
$\sigma_2 \leftarrow [\,]$
for $0 \leqslant i < r$ **do**
  if $\mathsf{ch}_2[i] == 0$ **then**
    $\sigma_2 \leftarrow \sigma_2||$PackArray31($\mathbf{r}_0[i]$)
  else
    $\sigma_2 \leftarrow \sigma_2||$PackArray31($\mathbf{r}_1[i]$)
  end if
end for
for $0 \leqslant i < r$ **do**
  if $\mathsf{ch}_2[i] == 0$ **then**
    $\sigma_2 \leftarrow \sigma_2||c_1[i]$
  else
    $\sigma_2 \leftarrow \sigma_2||c_0[i]$
  end if
end for

Output: $\sigma = R||\sigma_0||\sigma_1||\sigma_2$

Algorithm 17 $\mathsf{Vf}(\mathsf{pk}, \sigma, M)$

Input: pk, σ, M

$R \leftarrow$subarray$(\sigma, 0, k/8)$
$\sigma_0 \leftarrow$subarray$(\sigma, k/8, 2 \cdot k/8)$
$\sigma_1 \leftarrow$subarray$(\sigma, 2 \cdot k/8, (2 \cdot k + 10 \cdot n \cdot r)/8)$
$\sigma_2 \leftarrow$subarray$(\sigma, (2 \cdot k + 10 \cdot n \cdot r)/8, len(\sigma))$
$S_{\mathbf{F}} \leftarrow$subarray$(\mathsf{pk}, 0, k/8)$
$\mathbf{F} \leftarrow \mathcal{MQ}$system$(S_{\mathbf{F}})$
$D \leftarrow$Hash$(\mathsf{pk}||R||M)$
$\mathsf{ch}_1 \leftarrow$Hash1(D, σ_0)
$\mathsf{ch}_2 \leftarrow$Hash2$(D, \sigma_0, \mathsf{ch}_1, \sigma_1)$
$\mathsf{resp}_1 \leftarrow$UnpackArray31$(\sigma_1)$
$\mathsf{resp}_2 \leftarrow$UnpackArray31(subarray$(\sigma_2, 0, 5 \cdot n \cdot r/8)$)
$c \leftarrow$subarray$(\sigma_2, 5 \cdot n \cdot r/8, len(\sigma_2))$
$\mathsf{com} \leftarrow [\,]$
for $0 \leqslant i < r$ **do**
  $\mathbf{t}_1 \leftarrow \mathsf{resp}_1[2i]$
  $\mathbf{e}_1 \leftarrow \mathsf{resp}_1[2i+1]$
  if $\mathsf{ch}_2[i] == 0$ **then**
    $\mathbf{r}_0 \leftarrow \mathsf{resp}_2[i]$
    $c_0 \leftarrow$Com0$(\mathbf{r}_0, \mathsf{ch}_1[i] \cdot \mathbf{r}_0 - \mathbf{t}_1, \mathsf{ch}_1[i] \cdot$EvaluateF$(\mathbf{r}_0, \mathbf{F}) - \mathbf{e}_1)$
    $c_1 \leftarrow$subarray$(c, i \cdot k/8, (i+1) \cdot k/8)$
  else
    $\mathbf{r}_1 \leftarrow \mathsf{resp}_2[i]$
    $c_1 \leftarrow$Com1$(\mathbf{r}_1, \mathsf{ch}_1[i] \cdot (\mathbf{v} -$EvaluateF$(\mathbf{r}_1, \mathbf{F})) -$EvaluateG$(\mathbf{t}_1, \mathbf{r}_1, \mathbf{F}) - \mathbf{e}_1)$
    $c_0 \leftarrow$subarray$(c, i \cdot k/8, (i+1) \cdot k/8)$
    $\mathsf{com} \leftarrow \mathsf{com}||c_0||c_1$
  end if
end for
$\sigma_0' \leftarrow$Hash(com)

Output: $\sigma_0' == \sigma_0$

10

Security of MQDSS

10.1 EU-CMA security of MQDSS

The security of MQDSS was proven in [16]. The security reduction is in the random oracle model and builds on the results obtained for $q2$ signature schemes (see Appendix A, Section A.1). For completeness we provide the full proof in Appendix A, Section A.2.

Theorem 10.1. MQDSS *is EU-CMA-secure in the random oracle model, if the following conditions are satisfied:*

- *the search version of the $\mathcal{MQ}$ problem is intractable in the average case,*
- *the hash functions $\mathcal{H}$, H_1, and H_2 are modeled as random oracles,*
- *the commitment functions Com_0 and Com_1 are computationally binding, computationally hiding, and have $\mathcal{O}(k)$ bits of output entropy,*
- *the function XOF_F is modeled as random oracle and*
- *the pseudorandom generators $PRG_{\mathbf{sk}}$, $PRG_{\mathbf{s}}$ and $PRG_{\mathbf{rte}}$ have outputs computationally indistinguishable from random for any polynomial time adversary.*

10.2 Attacks Against MQDSS

Having shown the EU-CMA security of MQDSS, the best attacks against the cryptosystem are against the conditions that provide the security. Thus, an adversary could:

- Attack the $\mathcal{MQ}$ problem,
- Attack the computationally binding property of the commitments
- Attack the computationally hiding property of the commitments
- Attack the hash functions
- Attack the pseudo-random generators

Since the commitment functions, the hash functions and the pseudo-random generators are all instantiated using SHAKE256, all the attacks apart from the first boil down to attacking SHAKE256.

One could compromise the security of MQDSS if one breaks the preimage resistance (this will break the hiding property of the commitments), the collision resistance (this will break the binding property of the commitments) or if one finds properties that distinguish the output of SHAKE256 from random. A substantial amount of research has been devoted to the security of SHAKE and the SHA3 standard. The public scrutiny gives confidence

in its security, however the details are out of the scope of this document. We refer the interested reader to [45, 11].

This leaves attacks against the $\mathcal{MQ}$ problem as the point of interest. Since the public key is randomly generated (from a random seed by expanding the seed), the obtained system can be considered as semiregular, i.e. we can be confident that there are no hidden structural weaknesses. This means that the generic algebraic methods are the best algorithms against the $\mathcal{MQ}$ instance in MQDSS and therefore against the system. For details see 2. Based on these conclusions, the security of the proposed parameter sets can be estimated as in Table 8.2. Additional parameters security estimate is given in Table 8.4, and scaled-down parameters estimate in Table 13.3.

11

Design Rationale

In this chapter we discuss all relevant design choices that we made and provide appropriate justification for these choices.

11.1 Parameters

In choosing appropriate parameters for MQDSS, the most important criteria was of course the level of security these parameters provide. In the previous chapters we provided a complete security analysis of MQDSS. We

- proved the security of MQDSS in the random oracle model (cf. Section 10.1),
- analyzed the practical security of the $\mathcal{MQ}$ problem by investigating the state of the art classical and quantum algorithms for solving it (c.f. Section 2.2 and Section 2.3), and
- used known results about the security of the extendable output function SHAKE256, which we used to instantiate the commitments, the pseudo-random generators and hash functions.

Since our security reduction in the ROM is very loose, we found it impractical to use concrete expressions from the reduction in our choice of parameters. Instead, the parameters are based on the best known attacks against the $\mathcal{MQ}$ problem and against SHAKE256. In particular,

- We choose the number of variables and equations in $\mathbf{F}$ to be the same i.e $m = n$, as this gives effectively the hardest instances of the $\mathcal{MQ}$ problem.
- Using the analysis from Section 2.2, we estimate the lower bound of the number of variables n' in order for the resulting $\mathcal{MQ}$ instance to satisfy a particular security level/category (as defined in [46]) in terms of classical field operations of the best classical attacks,
- Using the analysis from Section 2.3, we estimate the lower bound of the number of variables n'' in order for the resulting $\mathcal{MQ}$ instance to satisfy a particular security level/category (as defined in [46]) in terms of quantum circuit size and depth of the best quantum attacks,
- The number of variables n is then chosen as $n = \max\{n', n''\}$.
- We choose the parameter k such that the output of the hash functions $\mathcal{H}, H_1, H_2$ is large enough to satisfy collision resistance security of the level specified by Categories 2,4 and 6.

- Finally, the number of rounds r is chosen such that the parallel composition of r rounds of the SSH 5-pass IDS has soundness error $< 1/2^k$.

11.2 5-pass over 3-pass SSH Identification Scheme

In [41], Sakumoto, Shirai and Hiwatari propose also a 3-pass scheme whose security also provably relies on the $\mathcal{MQ}$ problem and is defined solely over $\mathbb{F}_2$. One could argue that this one is a much more natural choice. Indeed, it is a 3-pass scheme, so one can directly apply the Fiat-Shamir transform that has been scrutinized for decades by the community. In addition it is defined over the Boolean domain, so implementation is particularly easy. However, a careful analysis shows that it has a serious drawback, that make it clearly inferior compared to the 5-pass SSH Identification scheme.

The 3-pass SSH scheme has a soundness error of 2/3 which is greater than $\frac{q+1}{2q}$ (the 5-pass SSH soundness error) for any $q > 2$. Thus for example, in order to satisfy Categories 1-2, the number of rounds would have to be 438, which is much larger than 269 - the number of rounds in MQDSS-31-48 (Security categories 1-2). Now, for Categories 1-2, the number of variables in the $\mathcal{MQ}$ system needs to be at least $n = 160$, which amounts to a signature of size $2k + r(3n + k) = 40360$ bytes (see [16] for derivation of this formula) which is more than 7 KB larger than MQDSS-31-48. For Categories 3-4 the difference is even larger - more than 8KB, since we now need at least $r = 657$ and $n = 224$ which gives a signature of size 76276 bytes.

11.3 Optimizations

In the definition of MQDSS (see Chapter 7) we have used an optimization proposed already in [41]: It is not necessary to include all $2r$ commitments in the transcript. Instead, we include a digest over the concatenation of all commitments $\sigma_0 = \mathcal{H}(\mathsf{com}^{(1)}||\mathsf{com}^{(2)}||\ldots||\mathsf{com}^{(r)})$ and also the commitments $c^{(1)}_{1-b^{(1)}}, c^{(2)}_{1-b^{(2)}}, \ldots, c^{(r)}_{1-b^{(r)}}$ that the verifier can not recompute. This optimization saves $(r-1)k$ bits from the final signature which is more than 8.5KB for MQDSS-31-48 and more than 19KB for MQDSS-31-64. This modification does not cause any problems, since we have shown (c.f.Chapter 10) that it does not disturb the security arguments.

11.4 Other Functions

In order to instantiate the commitment functions, pseudorandom generators and extendable output function, we rely on SHAKE-256, as standardized in FIPS 202, the SHA-3 standard. This gives us a sufficiently large security margin that its preimage and second preimage resistance is not relevant for the overall security level. In general, this means that we simply concatenate the defined inputs as byte arrays, and absorb them into the SHAKE state. Chapter 9 provides some more detail on specific usage.

12

Performance Analysis

12.1 Performance on Intel x64-86

In order to obtain benchmarks, we evaluate our reference implementation on a machine using the Intel x64-86 instruction set. In particular, we use a single core of a 3.5 GHz Intel Core i7-4770K CPU. We follow the standard practice of disabling TurboBoost and hyper-threading. The system has 32 KiB L1 instruction cache, 32 KiB L1 data cache, 256 KiB L2 cache and 8192 KiB L3 cache. Furthermore, it has 32GiB of RAM, running at 1333 MHz. When performing the benchmarks, the system ran on Linux kernel 4.9.0-4-amd64, Debian 9 (Stretch).

We compiled the code using GCC version `6.3.0-18`, with the compiler optimization flag `-O3`. The median resulting cycle counts are listed in the table below.

	keygen	signing	verification
MQDSS-31-48	1 206 730	52 466 398	38 686 506
MQDSS-31-64	2 806 750	169 298 364	123 239 874

12.2 Performance on Intel x64-86 AVX2

Since the evaluation of the MQ function is the most costly part of the computation but also benefits greatly from parallelism, we thought it useful to also provide benchmarks when the scheme is implemented using AVX2 instructions. We used the same system described above to obtain the following measurements, this time including the `-mavx2` compiler flag.

	keygen	signing	verification
MQDSS-31-48	1 069 536	6 369 484	3 951 838
MQDSS-31-64	2 485 394	14 584 882	9 619 442

12.3 Size

As the private key is merely a seed that is used to generate the required secret material, this is 32 respectively 48 bytes for the given parameter sets. The public key contains a public seed, but also $\mathbf{F}(\mathbf{s})$, making it 62 and 88 bytes respectively.

The stack space consumption is largely determined by the size of the signature and the expanded version of $\mathbf{F}$. A straight-forward implementation constructs the transcript in

memory before evaluating the hash function that determines the challenges. More memory-conservative implementations could keep an intermediate hash function state, instead, and stream through the transcript as it is constructed.

The expanded version of **F** requires some active memory. Naively, it benefits from having 57 KiB or 100 KiB (for the different parameter sets, respectively) of active memory available. More memory-constrained implementations could reschedule the different computations in a way that **F** only needs to be parsed once, however, and can thus also make use of a streaming API.

For the given parameter sets, the signature size is respectively 32882 and 67800 bytes (i.e. 32.1 KiB and 66.21 KiB). Since the signature primarily consists of transcripts of rounds of the non-interactive identification protocol, it scales linearly in the number of rounds and in the size of the vectors (see Chapter 5 for more details on this).

13

Security v.s. Performance

MQQDSS depends on four parameters q, k, n, r. The first parameter q determines the underlying field, and changing it has mostly to do with performance, since all the arithmetic operations are performed using different types of instructions which may influence speed for example. In some cases, the choice of q may introduce different dedicated attacks for the particular field, as in the case of $q = 2$, which may have slightly better performance (see Chapter 2 for detailed analysis of the known algorithms against the $\mathcal{MQ}$ problem). For a fixed value of q by increasing or decreasing the parameters k and n we increase or decrease the resistance of the system against known attacks. Note that we have specified earlier $r = \left\lceil k/\log_2 \frac{2q}{q+1} \right\rceil$ but it is possible (if one wants) to independently tune this parameter (for example to increase the performance). We will not consider this possibility in this document, and assume that r is not an independent parameter.

Based on the NIST call document [46], in a similar fashion to the 6 provided security categories, we identify 4 down-scaled categories

- BLOKCIPHER64 (Category 0.1) - the security level of a generic block cipher with 64 bit key.
- HASHFUNCTION128 (Category 0.2) - the security level of a generic hash function with 128 bit output.
- BLOKCIPHER96 (Category 0.3) - the security level of a generic block cipher with 96 bit key.
- HASHFUNCTION192 (Category 0.4) - the security level of a generic hash function with 192 bit output.

Our estimate of the concrete security level these provide in terms of classical and quantum gates, assuming black box treatment of the primitives (i.e. the best attacks are the generic ones) is given in Table 13.1.

	Security category	Classical Gates	Quantum Gates	Quantum circuit depth
0.1	BLOKCIPHER64	2^{74}	2^{50}	2^{46}
0.2	HASHFUNCTION128	2^{77}		
0.3	BLOKCIPHER96	2^{108}	2^{66}	2^{62}
0.4	HASHFUNCTION192	2^{110}		

Table 13.1: Basic characteristics of the scalled down parameter sets

We identify the following parameter sets that satisfy the scaled down security categories 0.1-0.4.

Security category	k	q	n	r	Public key size (bytes)	Secret key size (bytes)	Signature size (bytes)
0.1-0.2	128	4	48	189	28	16	9860
0.1-0.2	128	16	32	141	32	16	9056
0.1-0.2	128	31	24	135	31	16	8267
0.1-0.2	128	32	24	134	31	16	8206
0.1-0.2	128	64	24	131	34	16	9202
0.3-0.4	192	4	64	284	40	24	20496
0.3-0.4	192	16	40	211	44	24	17772
0.3-0.4	192	31	40	202	49	24	20046
0.3-0.4	192	32	40	201	49	24	19947
0.3-0.4	192	64	32	197	48	24	18960

Table 13.2: Basic characteristics of the scalled down parameter sets

Security category	q	n	Best classical attack		Best quantum attack		
			algorithm	Field op.	algorithm	Gates	Depth
0.1-0.2	4	48	Crossbread	2^{79}	Crossbread	2^{57}	2^{48}
0.1-0.2	16	32	Crossbread	2^{82}	Crossbread	2^{59}	2^{51}
0.1-0.2	31	24	Crossbread	2^{77}	Crossbread	2^{59}	2^{50}
0.1-0.2	32	24	Crossbread	2^{78}	Crossbread	2^{53}	2^{45}
0.1-0.2	64	24	Crossbread	2^{90}	Crossbread	2^{60}	2^{52}
0.3-0.4	4	64	Crossbread	2^{106}	Crossbread	2^{71}	2^{62}
0.3-0.4	16	40	Crossbread	2^{106}	Crossbread	2^{72}	2^{63}
0.3-0.4	31	40	Crossbread	2^{128}	Crossbread	2^{86}	2^{76}
0.3-0.4	32	40	Crossbread	2^{129}	Crossbread	2^{83}	2^{73}
0.3-0.4	64	32	Crossbread	2^{116}	Crossbread	2^{73}	2^{64}

Table 13.3: Best classical and quantum attacks against the scalled down parameter sets

Since the signature size of MQDSS is the most critical performance characteristic, it is natural to consider it over other characteristics when estimating the security vs. performance trade-off. For better visual judgement, we have plotted this trade-off for $q = 31$, which is the chosen value of our reference parameter sets (see Section 8.1).

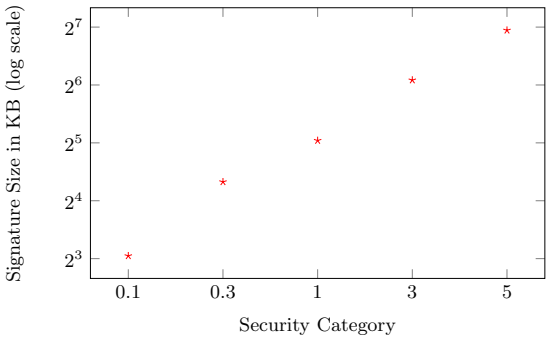


Fig. 13.1: Security category v.s. signature size

14

Strengths and Weaknesses

For any cryptographic design, the final product is a result based on decisions made to satisfy a certain security level, while maintaining desired properties such as performance and usability. This trade-off necessarily introduces weaknesses, but the designers' goal is to preserve enough good features to make the schemes attractive.

MQDSS is not an exception. In this chapter, we summarize and discuss the strengths and weaknesses of our proposal.

Strengths of MQDSS:

- *Small keys.* MQDSS has extremely small keys, comparable to contemporary schemes such as ECDSA that provide only classical security. On the other hand, they are several orders of magnitudes smaller than the keys of other $\mathcal{MQ}$ schemes.
- *Provably secure $\mathcal{MQ}$ signature, with reduction from the $\mathcal{MQ}$ problem.* MQDSS is the first multivariate signature scheme that is provably secure, and whose security relies solely on the $\mathcal{MQ}$ problem. The lack of security proofs throughout the history of $\mathcal{MQ}$ cryptography has made its representatives extremely prone to cryptanalysis and unfortunately the whole area has obtained bad reputation because of this. We believe that MQDSS together with the SSH schemes [41] are a step towards regaining confidence in $\mathcal{MQ}$ cryptography.
- *Flexible parameters.* All four parameters q, n, k, r can be easily tuned to match different security levels and platforms. Even more the number of variables is independent of the number of rounds, so in case of improvement in algebraic attacks against the $\mathcal{MQ}$ problem only the number of variable could be changed.
- *Simple design.* The underlying IDS uses a simple splitting technique based on the bilinearity of the polar form. The rest is a slightly more general Fiat-Shamir transform to turn the interactive protocol into a signature. The design does not utilize complicated algebraic structures (possibly even mathematically poorly understood), there is no dependence on possibly vulnerable distribution samplers, and in general there is very little room for flawed deployment.
- *Suitable for hardware implementation.* Due to the flexible parameters, it is possible to define MQDSS over fields of characteristic 2, such as $\mathbb{F}_{16}$ that are especially suitable for hardware implementation.
- *Naturally parallelizable.* The computations within a round are independent of the other rounds so it is straightforward to perform in parallel all rounds.

- *Inherently constant-time.* The straight-forward way of implementing the scheme is inherently protected against timing attacks. Evaluating the $\mathcal{MQ}$ function can traditionally be done in ways that depend on the input, but this is typically an additional optimization effort. Moreover, our chosen parameter set makes this unattractive on most platforms.

Weaknesses of MQDSS:

- *Large signature size.* Probably the biggest weakness of MQDSS is its signature size. Compared to traditional signature schemes the signature is at least 100 times larger. The same is true for other multivariate schemes. However, traditional $\mathcal{MQ}$ schemes have ad-hoc designs, without proof of security. Even more, in a typical usage scenario of signatures such as PKI, what matters is actually the size of the public key plus the signature. In such a setting MQDSS is still better, beating traditional $\mathcal{MQ}$ schemes by a factor of 2-20 depending on the scheme. On the other hand, provably secure schemes that provide post-quantum security tend to have much larger signatures, and for the schemes we are aware, the signatures are in the same range as MQDSS.
- *Security proof in the ROM, and not in the QROM.* In Section A.1 we showed in the random oracle model that $q2$-signature schemes are EU-CMA secure when the underlying IDS satisfies certain conditions. However, similar to the standard Fiat-Shamir transform, our proof relies on a forking lemma, which introduces two serious problems in the post-quantum setting (i.e. in the quantum accessible random oracle model): rewinding of the adversary, and adaptively programming the random oracle. While it is known how to deal with the latter [54], the former seems to be a serious obstacle [2]. The only known way to fix the Fiat-Shamir transform in the QROM setting [23] is using oblivious commitments, which are a certain kind of trapdoor commitments, effectively avoiding rewinding at the cost of introducing the necessity of a trapdoor function. This makes the solution not applicable in our setting as there are no known trapdoor functions with a reduction from the $\mathcal{MQ}$-problem.
  It is however possible to use a different transform that overcomes the problems of the forking lemma in the QROM. In [17], the authors generalize the Unruh transform [54], and apply it to the 5-pass SSH of Sakumoto *et al.*. The obtained signature scheme is secure in the ROM, but at a huge cost - the signature size becomes $\approx 120KB$ which in our opinion is not in the range of desired practicality.
- *Security proof not tight.* Another weakness of our security proof is that is not at all tight. This is again an inherent weakness introduced by the rewinding technique of the forking lemma. Therefore, in order to produce a tight security reduction for MQDSS one would have to base the proof on different techniques. At the moment, we are not aware of such techniques that we could use.

15

Additional AVX2 Implementation of MQDSS

To demonstrate performance, we have also implemented the scheme using AVX2 vector instructions. As mentioned above, this makes convenient use of the structure of the terms, allowing implementations to benefit from the `vpmaddubs` instruction to combine two multiplications with an addition. In one instruction, this computes two 8 bit SIMD multiplications and a 16 bit SIMD addition. This also underlines the benefit of details such as elements in $\mathbf{F}$ in signed representation, since this allows accumulating more additions in vectorized 16-bit words before performing a reduction.

When arranging reductions, we must strike a careful balance between preventing overflow and not reducing more often than necessary. As we make extensive use of `vpmaddubsw`, which takes both a signed and an unsigned operand to compute the quadratic monomials, we ensure that the input variables for the $\mathcal{MQ}$ function are unsigned values (in particular: $\{0, \ldots, 31\}$). For the coefficients in the system parameter $\mathbf{F}$, we can then freely assume the values are in $\{-15, \ldots, 15\}$, as these are the direct result of a pseudo-random generator.

It turns out to be efficient to immediately reduce the quadratic monomials back to $\{0, \ldots, 31\}$ when they are computed. When we now multiply such a product with an element from the system parameter and add it to the accumulators, the maximum value of each accumulator word will be at most[1] $64 \cdot 31 \cdot 15 = 29760$. As this does not exceed the maximum value of 32768, we only have to perform reductions on each individual accumulator at the very end.

For the smaller parameter set, i.e. $n = 48$, these constraints are less pressing, but the maximum value accumulators remains in the same ballpark. Both $n = 48$ and $n = 64$ benefit from the fact that these parameters are multiples of 16, which results in a very similar optimal implementation strategy and convenient code reuse.

[1] This follows from the fact that we combine 64 such monomials in two `YMM` registers.

References

1. Abdalla, M., An, J.H., Bellare, M., Namprempre, C.: From identification to signatures via the fiat-shamir transform: Minimizing assumptions for security and forward-security. In: Knudsen, L.R. (ed.) Advances in Cryptology — EUROCRYPT 2002: International Conference on the Theory and Applications of Cryptographic Techniques Amsterdam, The Netherlands, April 28 – May 2, 2002 Proceedings. pp. 418–433. Springer Berlin Heidelberg, Berlin, Heidelberg (2002), https://doi.org/10.1007/3-540-46035-7_28
2. Ambainis, A., Rosmanis, A., Unruh, D.: Quantum attacks on classical proof systems: The hardness of quantum rewinding. In: FOCS 2014. pp. 474–483 (2014), http://eprint.iacr.org/2014/296
3. Amy, M., Maslov, D., Mosca, M.: Polynomial-time t-depth optimization of clifford+t circuits via matroid partitioning. IEEE Trans. on CAD of Integrated Circuits and Systems 33(10), 1476–1489 (2014), https://doi.org/10.1109/TCAD.2014.2341953
4. Amy, M., Maslov, D., Mosca, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Trans. on CAD of Integrated Circuits and Systems 32(6), 818–830 (2013), https://doi.org/10.1109/TCAD.2013.2244643
5. Bardet, M., Faugère, J., Salvy, B.: On the complexity of the F5 Gröbner basis algorithm. Journal of Symbolic Computation 70, 49–70 (2015), https://hal.inria.fr/hal-01064519/document
6. Bardet, M., Faugère, J., Salvy, B., Spaenlehauer, P.: On the complexity of solving quadratic boolean systems. Journal of Complexity 29(1), 53–75 (2013), www-polsys.lip6.fr/~jcf/Papers/BFSS12.pdf
7. Bardet, M., Faugère, J.C., Salvy, B., Yang, B.Y.: Asymptotic Behaviour of the Degree of Regularity of Semi-Regular Polynomial Systems. In: Proc. of MEGA 2005, Eighth International Symposium on Effective Methods in Algebraic Geometry (2005)
8. Bardet, M., Faugre, J.C., Salvy, B.: On the complexity of the F5 Gröbner basis algorithm. Journal of Symbolic Computation 70(Supplement C), 49 – 70 (2015), http://www.sciencedirect.com/science/article/pii/S0747717114000935
9. Beauregard, S., Brassard, G., Fernandez, J.M.: Quantum arithmetic on galois fields (2003)
10. Bernstein, D.J.: Multi-user schnorr security, revisited. Cryptology ePrint Archive, Report 2015/996 (2015), https://eprint.iacr.org/2015/996
11. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak reference (2011), http://keccak.noekeon.org/
12. Bettale, L., Faugère, J., Perret, L.: Solving polynomial systems over finite fields: improved analysis of the hybrid approach. In: van der Hoeven, J., van Hoeij, M. (eds.) Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation – ISSAC '12. pp. 67–74. ACM (2012), https://hal.inria.fr/hal-00776070/document
13. Bouillaguet, C., Chen, H.C., Cheng, C.M., Chou, T., Niederhagen, R., Shamir, A., Yang, B.Y.: Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2010. LNCS, vol. 6225, pp. 203–218. Springer (2010), http://dx.doi.org/10.1007/978-3-642-15031-9_14, https://eprint.iacr.org/2010/313
14. Boyer, M., Brassard, G., Hyer, P., Tapp, A.: Tight bounds on quantum searching. Fortschritte der Physik 46(4-5), 493–505 (1998), http://dx.doi.org/10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P
15. Buchberger, B.: Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symb. Comput. 41(3-4), 475–511 (2006)

16. Chen, M.S., Hülsing, A., Rijneveld, J., Samardjiska, S., Schwabe, P.: From 5-pass $\mathcal{MQ}$-based identification to $\mathcal{MQ}$-based signatures. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. LNCS, vol. 10032, pp. 135–165. Springer (2016), http://eprint.iacr.org/2016/708
17. Chen, M.S., Hülsing, A., Rijneveld, J., Samardjiska, S., Schwabe, P.: Sofia: Mq-based signatures in the qrom. Cryptology ePrint Archive, Report 2017/680 (2017), https://eprint.iacr.org/2017/680
18. Cheung, D., Maslov, D., Mathew, J., Pradhan, D.K.: On the design and optimization of a quantum polynomial-time attack on elliptic curve cryptography. In: Kawano, Y., Mosca, M. (eds.) Theory of Quantum Computation, Communication, and Cryptography: Third Workshop, TQC 2008 Tokyo, Japan, January 30 - February 1, 2008. Revised Selected Papers. pp. 96–104. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-89304-2_9
19. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. pp. 1–6. STOC '87, ACM, New York, NY, USA (1987), http://doi.acm.org/10.1145/28395.28396
20. Coppersmith, D.: Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. Mathematics of Computation 62, 333–350 (1994)
21. Courtois, N., Klimov, E., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. LNCS, vol. 1807, pp. 392–407. Springer (2000), www.iacr.org/archive/eurocrypt2000/1807/18070398-new.pdf
22. Courtois, N.T.: Efficient zero-knowledge authentication based on a linear algebra problem MinRank. In: Boyd, C. (ed.) Advances in Cryptology – ASIACRYPT 2001. LNCS, vol. 2248, pp. 402–421. Springer (2001), https://eprint.iacr.org/2001/058
23. Dagdelen, Ö., Fischlin, M., Gagliardoni, T.: The Fiat–Shamir transformation in a quantum world. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT 2013. LNCS, vol. 8270, pp. 62–81. Springer (2013), http://dx.doi.org/10.1007/978-3-642-42045-0_4, https://eprint.iacr.org/2013/245
24. Diem, C.: The XL-algorithm and a conjecture from commutative algebra. In: Lee, P.J. (ed.) Advances in Cryptology – ASIACRYPT 2004. LNCS, vol. 3329, pp. 323–337. Springer (2004), https://www.iacr.org/archive/asiacrypt2004/33290320/33290320.pdf
25. Ding, J., Hu, L., Yang, B.Y., Chen, J.M.: Note on design criteria for rainbow-type multivariates. Cryptology ePrint Archive, Report 2006/307 (2006), https://eprint.iacr.org/2006/307
26. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases (F4). Journal of Pure and Applied Algebra 139, 61–88 (1999), http://www-polsys.lip6.fr/~jcf/Papers/F99a.pdf
27. Faugère, J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation – ISSAC '02. pp. 75–83. ACM (2002), http://www-polsys.lip6.fr/~jcf/Papers/F02a.pdf
28. Faugère, J.C., Levy-dit-Vehel, F., Perret, L.: Cryptanalysis of MinRank. In: Wagner, D. (ed.) Advances in Cryptology – CRYPTO 2008. LNCS, vol. 5157, pp. 280–296. Springer (2008), http://www-polsys.lip6.fr/~jcf/Papers/crypto08.pdf
29. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86: Proceedings. pp. 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg (1987), https://doi.org/10.1007/3-540-47721-7_12
30. Fusco, G., Bach, E.: Phase transition of multivariate polynomial systems. In: Cai, J.Y., Cooper, B.S., Zhu, H. (eds.) International Conference on Theory and Applications of Models of Computation – TAMC 2007. LNCS, vol. 4484, pp. 632–645. Springer (2007), https://minds.wisconsin.edu/bitstream/handle/1793/60544/TR1588.pdf
31. Galbraith, S., Malone-Lee, J., Smart, N.P.: Public key signatures in the multi-user setting. Inf. Process. Lett. 83(5), 263–266 (Sep 2002), http://dx.doi.org/10.1016/S0020-0190(01)00338-6
32. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)
33. Giesbrecht, M., Lobo, A., Saunders, B.D.: Certifying inconsistency of sparse linear systems. In: Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation – ISSAC '98. pp. 113–119 (1998), http://doi.acm.org/10.1145/281508.281591, https://cs.uwaterloo.ca/~mwg/files/incons.pdf
34. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on Computing 17(2), 281–308 (1988), https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Digital%20Signatures/A_Digital_Signature_Scheme_Secure_Against_Adaptive_Chosen-Message_Attack.pdf

35. Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R.: Applying grover's algorithm to aes: Quantum resource estimates. In: Takagi, T. (ed.) Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings. pp. 29–43. Springer International Publishing, Cham (2016), https://doi.org/10.1007/978-3-319-29360-8_3
36. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing – STOC '96. pp. 212–219. ACM (1996), https://arxiv.org/pdf/quant-ph/9605043v3.pdf
37. Joux, A., Vitse, V.: A crossbred algorithm for solving boolean polynomial systems. Cryptology ePrint Archive, Report 2017/372 (2017), http://eprint.iacr.org/2017/372
38. Kepley, S., Steinwandt, R.: Quantum circuits for $\mathcal{O}_{2^n}$-multiplication with subquadratic gate count. Quantum Information Processing 14(7), 2373–2386 (2015), https://doi.org/10.1007/s11128-015-0993-1
39. Kiltz, E., Masny, D., Pan, J.: Optimal security proofs for signatures from identification schemes. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. pp. 33–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-53008-5_2
40. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced Oil and Vinegar signature schemes. In: Stern, J. (ed.) Advances in Cryptology – EUROCRYPT '99. LNCS, vol. 1592, pp. 206–222. Springer (1999), http://www.goubin.fr/papers/OILLONG.PDF
41. KoichiSakumoto, Shirai, T., Hiwatari, H.: Public-key identification schemes based on multivariate quadratic polynomials. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. LNCS, vol. 6841, pp. 706–723. Springer (2011), https://www.iacr.org/archive/crypto2011/68410703/68410703.pdf
42. Lazard, D.: Gröbner-Bases, Gaussian elimination and resolution of systems of algebraic equations. In: van Hulzen, J.A. (ed.) EUROCAL. Lecture Notes in Computer Science, vol. 162, pp. 146–156. Springer (1983)
43. Lokshtanov, D., Paturi, R., Tamaki, S., Williams, R.R., Yu, H.: Beating brute force for systems of polynomial equations over finite fields. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19. pp. 2190–2202 (2017), https://doi.org/10.1137/1.9781611974782.143
44. Montgomery, P.L.: A block lanczos algorithm for finding dependencies over gf(2). In: Guillou, L.C., Quisquater, J.J. (eds.) Advances in Cryptology — EUROCRYPT '95: International Conference on the Theory and Application of Cryptographic Techniques Saint-Malo, France, May 21–25, 1995 Proceedings. pp. 106–120. Springer Berlin Heidelberg, Berlin, Heidelberg (1995), https://doi.org/10.1007/3-540-49264-X_9
45. NIST: FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions (2015), http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
46. NIST: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Cryptology ePrint Archive, Report 2015/996 (2016), http://csrc.nist.gov/groups/ST/postquantum-crypto/documents/call-for-proposals-final-dec-2016.pdf
47. Ohta, K., Okamoto, T.: On concrete security treatment of signatures derived from identification. In: Krawczyk, H. (ed.) Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings. pp. 354–369. Springer Berlin Heidelberg, Berlin, Heidelberg (1998), https://doi.org/10.1007/BFb0055741
48. Patarin, J.: Hidden field equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms. In: Maurer, U. (ed.) Advances in Cryptology – EUROCRYPT '96. LNCS, vol. 1070, pp. 33–48. Springer (1996), http://www.minrank.org/hfe.pdf
49. Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: Maurer, U. (ed.) Advances in Cryptology – EUROCRYPT '96, LNCS, vol. 1070, pp. 387–398. Springer (1996), https://www.di.ens.fr/~pointche/Documents/Papers/1996_eurocrypt.pdf
50. Stern, J.: A new paradigm for public key identification. IEEE Transactions on Information Theory 42(6), 1757–1768 (1996), https://www.di.ens.fr/users/stern/data/St55b.pdf
51. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. 13(4), 354–356 (Aug 1969), http://dx.doi.org/10.1007/BF02165411
52. Thomae, E.: About the Security of Multivariate Quadratic Public Key Schemes. Ph.D. thesis, Ruhr-University Bochum, Germany (2013), https://www.iacr.org/phds/116_EnricoThomae_AboutSecurityMultivariateQuadr.pdf

53. Thomae, E., Wolf, C.: Solving underdetermined systems of multivariate quadratic equations revisited. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) Public Key Cryptography – PKC 2012: 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings. pp. 156–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-30057-8_10
54. Unruh, D.: Non-interactive zero-knowledge proofs in the quantum random oracle model. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. LNCS, vol. 9056, pp. 755–784. Springer (2015), http://dx.doi.org/10.1007/978-3-662-46803-6_25, http://eprint.iacr.org/2014/587
55. Unruh, D.: Post-quantum security of fiat-shamir. Cryptology ePrint Archive, Report 2017/398, to appear in Advances in Cryptology - ASIACRYPT 2017 (2017), https://eprint.iacr.org/2017/398
56. Westerbaan, B., Schwabe, P.: Solving binary $\mathcal{MQ}$ with grover's algorithm. In: Carlet, C., Hasan, A., Saraswat, V. (eds.) Security, Privacy, and Advanced Cryptography Engineering. LNCS, vol. 10076. Springer (2016), document ID: 40eb0e1841618b99ae343ffa073d6c1e, http://cryptojedi.org/papers/#mqgrover
57. Williams, V.V.: Multiplying matrices faster than coppersmith-winograd. In: Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing. pp. 887–898. STOC '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2213977.2214056
58. Yang, B., Chen, J.: Theoretical analysis of XL over small fields. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) Information Security and Privacy. LNCS, vol. 3108, pp. 277–288. Springer (2004), http://dx.doi.org/10.1007/978-3-540-27800-9_24, http://www.iis.sinica.edu.tw/papers/byyang/2386-F.pdf
59. Yang, B.Y., Chen, J.M.: All in the XL family: Theory and practice. In: sik Park, C., Chee, S. (eds.) Information Security and Cryptology – ICISC 2004. pp. 67–86. Springer (2005), http://by.iis.sinica.edu.tw/by-publ/recent/xxl.pdf
60. Yeh, J.Y.C., Cheng, C.M., Yang, B.Y.: Operating Degrees for XL vs. F4/F5 for Generic $\mathcal{MQ}$ with Number of Equations Linear in That of Variables. In: Fischlin, M., Katzenbeisser, S. (eds.) Number Theory and Cryptography: Papers in Honor of Johannes Buchmann on the Occasion of His 60th Birthday. pp. 19–33. Springer (2013), http://www.iis.sinica.edu.tw/papers/byyang/17377-F.pdf

A

Security proofs

A.1 Security of $q2$-signature schemes.

For completeness of this document, we provide in full the security reduction for Construction 5.1 in the random oracle model, and the proof of EU-CMA security of the obtained signature scheme.

To prove this claim, we proceed in several steps. The proof builds on techniques introduced by Pointcheval and Stern [49] (see Section 4.2 for a brief description of the technique). As the reduction is far from being tight, we refrain from doing an exact proof as it does not gain us anything but a complicated statement. We first recall an important tool from [49] called the splitting lemma.

Lemma A.1 (Splitting lemma [49]). *Let $A \subset X \times Y$, such that $\Pr[A(x,y)] \geqslant \epsilon$. Then, there exists $\Omega \subset X$, such that*

$$\Pr[x \in \Omega] \geqslant \epsilon/2,$$
$$\Pr[A(a,y) | a \in \Omega] \geqslant \epsilon/2.$$

We next present a forking lemma for $q2$-signature schemes. The lemma shows that we can obtain four valid signatures which contain four valid transcripts of the underlying IDS, given a successful key-only adversary. Moreover, these four transcripts fulfill a certain requirement on the challenges (here the related parts of the hash function outputs) that we need later.

In the lemma and in the rest of the chapter, we model the functions H_1 and H_2 as independent random oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. Furthermore, for ease of exposition in our proofs, we use a "full" version of a signature, including the outputs h_1 and h_2 of H_1 and H_2, i.e., instead of $\sigma = (\sigma_0, \sigma_1, \sigma_2)$, we assume a signature has the form $\sigma = (\sigma_0, \mathsf{h}_1, \sigma_1, \mathsf{h}_2, \sigma_2)$. (Note that h_1 and h_2 need not be included in the signatures because they can be easily reconstructed from the other values.)

Lemma A.2 (Forking lemma for $q2$-signature schemes). *Let $\mathsf{Dss}(1^k)$ be a $q2$-signature scheme with security parameter $k \in \mathbb{N}$. If there exists a PPT adversary $\mathcal{A}$ that can output a valid signature message pair (M, σ) with non-negligible success probability, given only the public key as input, then, with non-negligible probability, rewinding $\mathcal{A}$ a polynomial number of times with the same random tape but different oracles, outputs 4 valid message signature pairs $(M, \sigma = (\sigma_0, \mathsf{h}_1, \sigma_1, \mathsf{h}_2, \sigma_2))$, $(M, \sigma' = (\sigma_0, \mathsf{h}'_1, \sigma'_1, \mathsf{h}'_2, \sigma'_2))$,*

$(M, \sigma'' = (\sigma_0, \mathsf{h}_1'', \sigma_1'', \mathsf{h}_2'', \sigma_2''))$, $(M, \sigma''' = (\sigma_0, \mathsf{h}_1''', \sigma_1''', \mathsf{h}_2''', \sigma_2'''))$, *such that there exists* $j \in \{1, \ldots, r\}$ *such that:*

$$\begin{aligned}(\mathsf{ch}_1)^{(j)} = (\mathsf{ch}_1')^{(j)} \neq (\mathsf{ch}_1'')^{(j)} = (\mathsf{ch}_1''')^{(j)},\\ (\mathsf{ch}_2)^{(j)} = (\mathsf{ch}_2'')^{(j)} \neq (\mathsf{ch}_2')^{(j)} = (\mathsf{ch}_2''')^{(j)},\end{aligned} \tag{A.1}$$

where $\mathsf{h}_1 = (\mathsf{ch}_1^{(1)}, \mathsf{ch}_1^{(2)}, \ldots, \mathsf{ch}_1^{(r)})$ *and* $\mathsf{h}_2 = (\mathsf{ch}_2^{(1)}, \mathsf{ch}_2^{(2)}, \ldots, \mathsf{ch}_2^{(r)})$ *and similarly for* $\mathsf{h}_1', \mathsf{h}_1'', \mathsf{h}_1'''$ *and* $\mathsf{h}_2', \mathsf{h}_2'', \mathsf{h}_2'''$.

Proof. To prove the Lemma we need to show that we can rewind $\mathcal{A}$ three times (and adaptability program the random oracles) and at the same time, the probability that $\mathcal{A}$ succeeds in forging a (different) signature in all four runs is non-negligible. Moreover, we have to show that the signatures have the additional property claimed in the Lemma, again with non-negligible probability.

Let $\omega \in R_w$ be $\mathcal{A}$'s random tape with R_w the set of allowable random tapes. During the attack $\mathcal{A}$ may ask polynomially many queries (in the security parameter k) $Q_1(k)$ and $Q_2(k)$ to the random oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. Let $q_{1,1}, q_{1,2}, \ldots, q_{1,Q_1}$ and $q_{2,1}, q_{2,2}, \ldots, q_{2,Q_2}$ be the queries to $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. Moreover, let $(r_{1,1}, r_{1,2}, \ldots, r_{1,Q_1}) \in (\mathsf{C}_1^r)^{Q_1}$ and $(r_{2,1}, r_{2,2}, \ldots, r_{2,Q_2}) \in (\mathsf{C}_2^r)^{Q_2}$ the corresponding answers of the oracles.

Denote by F the event that $\mathcal{A}$ outputs a valid message signature pair $(M, \sigma = (\sigma_0, \mathsf{h}_1, \sigma_1, \mathsf{h}_2, \sigma_2))$. Per assumption, this event occurs with non-negligible probability, i.e., $\mathsf{Pr}[\mathsf{F}] = \frac{1}{P(k)}$, for some polynomial $P(k)$. In addition, F implies $\mathsf{h}_1 = \mathcal{O}_1(M, \sigma_0)$ and $\mathsf{h}_2 = \mathcal{O}_2(M, \sigma_0, \mathsf{h}_1, \sigma_1)$. As $\mathsf{h}_1, \mathsf{h}_2$ are chosen uniformly at random from exponentially large sets $\mathsf{C}_1^r, \mathsf{C}_2^r$, the probability that $\mathcal{A}$ did not query $\mathcal{O}_1$ with (M, σ_0) and $\mathcal{O}_2$ with $(M, \sigma_0, \mathsf{h}_1, \sigma_1)$ is negligible. Hence, there exists a polynomial P' such that the event F' that F occurs and $\mathcal{A}$ queried $\mathcal{O}_1$ with (M, σ_0) and $\mathcal{O}_2$ with $(M, \sigma_0, \mathsf{h}_1, \sigma_1)$ has probability $\mathsf{Pr}[\mathsf{F}'] = \dfrac{1}{P'(k)}$.

For the moment consider only the second oracle. From the previous equation, there exists at least one $\beta \leqslant Q_2$ such that

$$\mathsf{Pr}[\mathsf{F}' \wedge q_{2,\beta} = (M, \sigma_0, \mathsf{h}_1, \sigma_1)] \geqslant \frac{1}{Q_2(k)P'(k)}$$

where the probability is taken over the random coins of $\mathcal{A}$ and all answers from $\mathcal{O}_2$, i.e. over the set $\mathcal{B} = \{(\omega, r_{2,1}, r_{2,2}, \ldots, r_{2,Q_2}) | \omega \in R_w \wedge (r_{2,1}, r_{2,2}, \ldots, r_{2,Q_2}) \in (\mathsf{C}_2^r)^{Q_2} \wedge \mathsf{F}' \wedge q_{2,\beta} = (M, \sigma_0, \mathsf{h}_1, \sigma_1)\}$.

(Informally, the following steps just show that the success of an algorithm with non-negligible success probability cannot be conditioned on an event that occurs only with negligible probability (i.e. the outcome of the $q_{2,\beta}$ query landing in some negligible subset).)

The last equation implies that there exists a non-negligible set of "good" random tapes $\Omega_\beta \subseteq R_\omega$ for which $\mathcal{A}$ can provide a valid signature and $q_{2,\beta}$ is the oracle query determining h_2. Applying the splitting lemma, we get that

$$\begin{aligned}\Pr[w \in \Omega_\beta] &\geqslant \frac{1}{2Q_2(k)P'(k)}\\ \Pr[(\omega, r_{2,1}, r_{2,2}, \ldots, r_{2,Q_2}) \in \mathcal{B} | w \in \Omega_\beta] &\geqslant \frac{1}{2Q_2(k)P'(k)}\end{aligned}$$

Applying the same reasoning again we can derive from the later probability being non-negligible that there exists a non-negligible subset $\Omega_{\beta,\omega}$ of the "good" oracle responses

$(r_{2,1}, r_{2,2}, \ldots, r_{2,\beta-1})$ such that $(\omega, r_{2,1}, r_{2,2}, \ldots, r_{2,Q_2}) \in \mathcal{B}$. Applying the splitting lemma again, we get

$$\Pr[(r_{2,1}, \ldots, r_{2,\beta-1}) \in \Omega_{\beta,\omega}] \geqslant \frac{1}{4Q_2(k)P'(k)}$$
$$\Pr[(\omega, r_{2,1}, \ldots, r_{2,Q_2}) \in \mathcal{B} | (r_{2,1}, \ldots, r_{2,\beta-1}) \in \Omega_{\beta,\omega})] \geqslant \frac{1}{4Q_2(k)P'(k)}$$

This means that rewinding $\mathcal{A}$ to the point where it made query $q_{2,\beta}$ and running it with new, random $r'_{2,\beta}, \ldots, r'_{2,Q_2}$ has a non-negligible probability of $\mathcal{A}$ outputting another valid signature. Therefore, we can use $\mathcal{A}$ to find with non-negligible probability two valid message signature pairs $(M, \sigma = (\sigma_0, \mathsf{h}_1, \sigma_1, \mathsf{h}_2, \sigma_2))$, $(M, \sigma' = (\sigma_0, \mathsf{h}'_1, \sigma'_1, \mathsf{h}'_2, \sigma'_2))$, such that $(\sigma_0, \mathsf{h}_1, \sigma_1) = (\sigma_0, \mathsf{h}'_1, \sigma'_1)$. and $\mathsf{h}_2 \neq \mathsf{h}'_2$.

We now rewind the adversary again using exactly the same technique as above but now considering the queries to $\mathcal{O}_1$ and its responses. In the replay we change the responses of $\mathcal{O}_1$ to obtain a third signature that differs from the previously obtained ones in the first associated hash value. In the same manner, it can be shown that with non-negligible probability $\mathcal{A}$ will output a third signature on M, $\sigma'' = (\sigma_0, \mathsf{h}''_1, \sigma''_1, \mathsf{h}''_2, \sigma''_2)$, such that $\mathsf{h}''_1 \neq \mathsf{h}'_1 = \mathsf{h}_1$.

Finally, we rewind the adversary a third time, keeping the responses of $\mathcal{O}_1$ from the last rewind and focusing on $\mathcal{O}_2$ again. Again, with non-negligible probability $\mathcal{A}$ will produce yet another signature on M, $\sigma''' = (\sigma_0, \mathsf{h}'''_1, \sigma'''_1, \mathsf{h}'''_2, \sigma'''_2)$ such that $(\sigma_0, \mathsf{h}''_1, \sigma''_1) = (\sigma_0, \mathsf{h}'''_1, \sigma'''_1)$ and $\mathsf{h}''_2 \neq \mathsf{h}'''_2$.

Summing up , rewinding the adversary three times, we can find four valid signatures $\sigma, \sigma', \sigma'', \sigma'''$ with non-negligible success probability $\frac{1}{P(k)}$ for some polynomial $P(k)$. Let us denote this event by $\mathcal{E}_\sigma$. So we have that

$$\Pr[\mathcal{E}_\sigma] \geqslant \frac{1}{P(k)}.$$

What remains is to show that the obtained signatures satisfy the particular structure from the lemma (Equation A.1) with non-negligible probability.

Let $\mathcal{H}$ be the event that for $(\sigma, \sigma', \sigma''.\sigma''')$ there exists a $j \in \{1, \ldots, r\}$ such that (A.1) is satisfied. For the probability that $\mathcal{A}$ outputs a valid signature with this property, we have that

$$\mathsf{Pr}[\mathcal{E}_\sigma \wedge \mathcal{H}] = \mathsf{Pr}[\mathcal{E}_\sigma] - \mathsf{Pr}[\neg\mathcal{H} \wedge \mathcal{E}_\sigma] \geqslant \frac{1}{P(k)} - \mathsf{Pr}[\neg\mathcal{H} \wedge \mathcal{E}_\sigma]$$

Now, let $\sigma, \sigma', \sigma'', \sigma'''$ be the four valid signatures that $\mathcal{A}$ outputs under the event $\neg\mathcal{H} \wedge \mathcal{E}_\sigma$. This means that (A.1) is not satisfied for $\sigma, \sigma', \sigma'', \sigma'''$ for any $j \in \{1, \ldots, r\}$.

Consider the set $S_{\neg\mathcal{H}}$ of all tuples $(\mathsf{h}_1, \mathsf{h}''_1, \mathsf{h}_2, \mathsf{h}'_2, \mathsf{h}''_2, \mathsf{h}'''_2) \in (\mathsf{C}_1^r)^2 \times (\mathsf{C}_2^r)^4$ where $\mathsf{h}_1 = (\mathsf{ch}_1^{(1)}, \mathsf{ch}_1^{(2)}, \ldots, \mathsf{ch}_1^{(r)}) \in \mathsf{C}_1^r$, (similarly for h'_1), and $\mathsf{h}_2 = (\mathsf{ch}_2^{(1)}, \mathsf{ch}_2^{(2)}, \ldots, \mathsf{ch}_2^{(r)}) \in \mathsf{C}_2^r$, (similarly for $\mathsf{h}'_2, \mathsf{h}''_2, \mathsf{h}'''_2$), and such that for every $j \in \{1, \ldots, r\}$ at least one of the following is true:

$$i.(\mathsf{ch}_1)^{(j)} = (\mathsf{ch}'_1)^{(j)}; \quad ii.(\mathsf{ch}_2)^{(j)} = (\mathsf{ch}'_2)^{(j)}; \quad iii.(\mathsf{ch}''_2)^{(j)} = (\mathsf{ch}'''_2)^{(j)}.$$

It is clear that the hash value tuple $(\mathsf{h}_1, \mathsf{h}''_1, \mathsf{h}_2, \mathsf{h}'_2, \mathsf{h}''_2, \mathsf{h}'''_2)$ in $\mathcal{A}$'s output under the event $\neg\mathcal{H} \wedge \mathcal{E}_\sigma$ must be in $S_{\neg\mathcal{H}}$. Indeed if the hash value tuple does not come from $S_{\neg\mathcal{H}}$, then there exists a $j \in \{1, \ldots, r\}$, such that none of $i., ii., iii.$, holds true, i.e., for this j

$$(\mathsf{ch}_1)^{(j)} \neq (\mathsf{ch}_1')^{(j)} \wedge (\mathsf{ch}_2)^{(j)} \neq (\mathsf{ch}_2')^{(j)} \wedge (\mathsf{ch}_2'')^{(j)} \neq (\mathsf{ch}_2''')^{(j)}.$$

A little thought reveals that the last is equivalent to (A.1), which is a contradiction to the assumption that the tuple comes under the event $\neg\mathcal{H} \wedge \mathcal{E}_\sigma$.

Recall that for $q2$-signatures, C_1 has size q and C_2 size 2. Now, the cardinality of $S_{\neg\mathcal{H}}$ can be calculated to be $|S_{\neg\mathcal{H}}| = (4q(3q+1))^r$, whereas the cardinality of $(\mathsf{C}_1^r)^2 \times (\mathsf{C}_2^r)^4$ is $(16q^2)^r$. This means that

$$\mathsf{Pr}[\neg\mathcal{H} \wedge \mathcal{E}_\sigma] \leqslant \frac{(4q(3q+1))^r}{(16q^2)^r} = \left(\frac{3q+1}{4q}\right)^r,$$

which is negligible in k since according to Construction 5.1, the number of rounds r must be super-logarithmic (in k), to fulfill C_2^r being exponentially large (in k).

Finally,

$$\mathsf{Pr}[\mathcal{E}_\sigma \wedge \mathcal{H}] = \mathsf{Pr}[\mathcal{E}_\sigma] - \mathsf{Pr}[\neg\mathcal{H} \wedge \mathcal{E}_\sigma] \geqslant \frac{1}{P(k)} - \left(\frac{3q+1}{4q}\right)^r = \frac{1}{P(k)} - \mathrm{negl}(k)$$

and hence, the conditions from the lemma are satisfied with non-negligible probability. □

With Lemma A.2 we can already establish unforgeability under key only attacks:

Theorem A.3 (KOA security of $q2$-signature schemes). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k)$ a $q2$-IDS that has a key relation R, is KOW secure, and has a $q2$-extractor. Then $q2\text{-}\mathsf{Dss}(1^k)$, the $q2$-signature scheme derived applying Construction 5.1 is unforgeable under key-only attacks.*

Proof. Let $\mathcal{A}$ be a PPT algorithm that forges a signature in a KOA setting, i.e., given only the public key pk outputs a valid message-signature pair (M, σ) with non-negligible probability ϵ. We show how to construct an algorithm $\mathcal{M}^{\mathcal{A}}$ that given IDS public key and oracle access to $\mathcal{A}$ breaks the KOW security of IDS in essentially the same running time as the given $\mathcal{A}$ and with negligibly different success probability.

On input the IDS public key pk, $\mathcal{M}^{\mathcal{A}}$ runs $\mathcal{A}(\mathsf{pk})$ which outputs a valid message-signature pair (M, σ) for $q2\text{-}\mathsf{Dss}$. Using the technique from Lemma A.2, rewinding $\mathcal{A}$, $\mathcal{M}^{\mathcal{A}}$ obtains four valid signatures that with overwhelming probability contain four valid transcripts that satisfy Equation (A.1). These are exactly the type of transcripts needed for the $q2$-extractor to extract a valid secret key sk'. Since $(\mathsf{pk}, \mathsf{sk}') \in R$, $\mathcal{M}^{\mathcal{A}}$ breaks the KOW security of IDS. □

For $\mathsf{EU\text{-}CMA}$ security, we still have to deal with signature queries. The following lemma shows that a reduction can produce valid responses to the adversarial signature queries if the identification scheme is honest-verifier zero-knowledge.

Lemma A.4. *Let $k \in \mathbb{N}$ the security parameter, $\mathsf{IDS}(1^k)$ a $q2$-IDS that is honest-verifier zero-knowledge. Then any PPT adversary $\mathcal{B}$ against the $\mathsf{EU\text{-}CMA}$-security of $q2\text{-}\mathsf{Dss}(1^k)$, the $q2$-signature scheme derived by applying Construction 5.1, can be turned into a key-only adversary $\mathcal{A}$ against $q2\text{-}\mathsf{Dss}$ with the properties described in Lemma A.2. $\mathcal{A}$ runs in polynomial time and succeeds with essentially the same success probability as $\mathcal{B}$.*

Proof. By construction. We show how to construct an oracle machine $\mathcal{A}^{\mathcal{B},\mathcal{S},\mathcal{O}_1,\mathcal{O}_2}$ that has access to $\mathcal{B}$, an honest-verifier zero-knowledge simulator $\mathcal{S}$, and random oracles $\mathcal{O}_1, \mathcal{O}_2$.

$\mathcal{A}$ produces a valid signature for $q2\text{-}\mathsf{Dss}(1^k)$ given only a public key running in time polynomial in k and achieving essentially the same success probability (up to a negligible difference) as $\mathcal{B}$.

Upon input of public key pk, $\mathcal{A}$ runs $\mathcal{B}^{\mathcal{O}'_1,\mathcal{O}'_2,\mathsf{Sign}}(\mathsf{pk})$ simulating the random oracles (ROs) $\mathcal{O}'_1, \mathcal{O}'_2$, as well as the signing oracle Sign towards $\mathcal{B}$. When $\mathcal{B}$ outputs a forgery (M^*, σ^*), $\mathcal{A}$ just forwards it.

To simulate the ROs, $\mathcal{A}$ keeps two initially empty tables of query-response pairs, one per oracle. Whenever $\mathcal{B}$ queries $\mathcal{O}'_b$, $\mathcal{A}$ first checks if the table for $\mathcal{O}'_b$ already contains a pair for this query. If such a pair exists, $\mathcal{A}$ just returns the stored response. Otherwise, $\mathcal{A}$ forwards the query to its own $\mathcal{O}_b$.

As IDS is honest-verifier zero-knowledge there exists a PPT simulator $\mathcal{S}$ that upon input of a IDS public key generates a valid transcript that is indistinguishable of the transcripts generated by honest protocol executions. Whenever $\mathcal{B}$ queries the signature oracle with message m, $\mathcal{A}$ runs $\mathcal{S}$ r times, to obtain r valid transcripts. $\mathcal{A}$ combines the transcripts to obtain a valid signature $\sigma = (\sigma_0, \mathsf{h}_1, \sigma_1, \mathsf{h}_2, \sigma_2)$. Before outputting σ, $\mathcal{A}$ checks if the table for $\mathcal{O}'_1$ already contains an entry for query (M, σ_0). If so, $\mathcal{A}$ aborts. Otherwise, $\mathcal{A}$ adds the pair $((M, \sigma_0), \mathsf{h}_1)$. Then, $\mathcal{A}$ checks the second table for query $(M, \sigma_0, \mathsf{h}_1, \sigma_1)$. Again, $\mathcal{A}$ aborts if it finds such an entry and adds $((M, \sigma_0, \mathsf{h}_1, \sigma_1), \mathsf{h}_2)$, otherwise.

The probability that $\mathcal{A}$ aborts is negligible in k. When answering signature queries, $\mathcal{A}$ verifies that certain queries were not made before. Both queries contain σ_1 which takes any given value only with negligible probability. On the other hand, the total number of queries that $\mathcal{B}$ makes to all its oracles is polynomially bounded. Hence, the probability that one of the two queries was already made before is negligible. If $\mathcal{A}$ does not abort, it perfectly simulates all oracles towards $\mathcal{B}$. Hence, $\mathcal{B}$ – and thereby $\mathcal{A}$ – succeeds with the same probability as in the real $\mathsf{EU\text{-}CMA}$ game in this case. Hence, $\mathcal{A}$ succeeds with essentially the same probability as $\mathcal{B}$. □

We now got everything we need to prove EU-CMA security. The proof is a straight forward application of Lemma A.2 and Lemma A.4.

Theorem A.5 (EU-CMA security of $q2$-signature schemes). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k)$ a $q2$-IDS that has a key relation R, is KOW secure, is honest-verifier zero-knowledge, and has a $q2$-extractor $\mathcal{E}$. Then $q2\text{-}\mathsf{Dss}(1^k)$, the $q2$-signature scheme derived applying Construction 5.1 is existentially unforgeable under adaptive chosen message attacks.*

Proof. Towards a contradiction, assume that there exists a PPT adversary $\mathcal{A}$ against the $\mathsf{EU\text{-}CMA}$-security of $q2\text{-}\mathsf{Dss}$ succeeding with non-negligible probability. We show how to construct a PPT algorithm $\mathcal{M}^{\mathcal{A}}$ that given the IDS public key and oracle access to $\mathcal{A}$ breaks the KOW security of IDS. Applying Lemma A.4, $\mathcal{M}^{\mathcal{A}}$ can construct a PPT key-only forger $\mathcal{B}$, with essentially the same success probability as $\mathcal{A}$. Given a public key for IDS (which is a valid $q2\text{-}\mathsf{Dss}$ public key) $\mathcal{M}^{\mathcal{A}}$ runs $\mathcal{B}$ as described in Lemma A.2. That way $\mathcal{M}^{\mathcal{A}}$ can use $\mathcal{B}$ to obtain four signatures that per (A.1) lead to four transcripts as required by the $q2$-extractor $\mathcal{E}$. Running $\mathcal{E}$, $\mathcal{M}^{\mathcal{A}}$ can extract a valid secret key sk' that breaks the KOW security of IDS.

$\mathcal{M}^{\mathcal{A}}$ just runs $\mathcal{B}$ and $\mathcal{E}$, two PPT algorithms. Consequently, $\mathcal{M}^{\mathcal{A}}$ runs in polynomial time. Also, $\mathcal{B}$ and $\mathcal{E}$ both have non-negligible success probability implying that $\mathcal{M}^{\mathcal{A}}$ also succeeds with non-negligible probability. □

A.2 Proof of Theorem 10.1 [EU-CMA security of MQDSS]

Before we present the proof, note that as our results from Section A.1 are non-tight we only prove an asymptotic statement. While this does not suffice to make any statement about the security of a specific parameter choice, it provides evidence that the general approach leads to a secure scheme.

To prove this theorem we would like to apply Theorem 5.2 (the same as Theorem A.5). However, Theorem 5.2 was formulated for a slightly more generic construction (see Construction 5.1). The point is that we apply an optimization originally proposed in [50]. So, in our actual proposal (see Chapter 7), the parallel composition of the IDS is slightly different as, instead of the commitments, only the hash of their concatenation is sent (c.f. σ_0 in Figure 7.2). Also, the last message (c.f. σ_2 in Figure 7.2) now contains the remaining commitments. Let's call this optimized version $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$.

Note that since MQDSS is an $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$ signature scheme, we could have focused our attention solely to $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$ schemes already in Chapter 5. However, this would have limited the general applicability of the result, as the above optimization is only applicable to schemes with a certain, less generic, structure such as MQDSS.

As the next Corollary shows, it is easy to verify that the results from Chapter 5 hold for the optimized $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$ scheme as well.

Corollary A.6 (EU-CMA security of $q2$-signature schemes). *Let $k \in \mathbb{N}$, $\mathsf{IDS}(1^k)$ a $q2$-IDS that has a key relation R, is KOW secure, is honest-verifier zero-knowledge, and has a $q2$-extractor $\mathcal{E}$. Then $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$, the optimized $q2$-signature scheme derived by applying Construction 5.1 and the optimization explained above, is existentially unforgeable under adaptive chosen message attacks.*

Proof. Regarding Lemma A.2, note that by removing duplicate information from the signature, we do not affect the ability to extract in any way, and thus the probability of success of the adversary remains exactly the same. Thus Lemma A.2 also holds for $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$.

For Lemma A.4, the arguments are exactly the same with the exception that the probability of abort of $\mathcal{A}$ may now be different, but nevertheless, still negligible. Indeed, the proof of Lemma A.4 uses the fact that the first signature element σ_1 only takes a given value with negligible probability. This follows from the fact that the commitment scheme has big enough output entropy – and thereby also takes a given value with negligible probability. In the case of $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$, this statement follows from the same property of the commitment scheme but also from the randomness of the RO that we used to model the hash function $\mathcal{H}$. Hence, the proof of Lemma A.4 also goes through for $opt\text{-}q2\text{-}\mathsf{Dss}(1^k)$.

Now, the rest of the proof proceeds exactly the same as in Theorem 5.2. □

Based on this corollary we can now prove Theorem 10.1.

Proof (of Theorem 10.1). Towards a contradiction, assume there exists an adversary $\mathcal{A}$ that wins the EU-CMA game against MQDSS with non-negligible success probability. We show that this implies the existence of an oracle machine $\mathcal{M}^{\mathcal{A}}$ that solves the $\mathcal{MQ}$ problem, breaks a property of one of the commitment schemes, or distinguishes the outputs of one of the pseudorandom generators from random. We first define a series of games and argue that the difference in success probability of $\mathcal{A}$ between these games is negligible. We assume that $\mathcal{M}$ runs $\mathcal{A}$ in these games.

Game 0: Is the EU-CMA game for MQDSS.

Game 1: Is Game 0 with the difference that $\mathcal{M}$ replaces the outputs of $\text{PRG}_{\mathbf{rte}}$ by random bit strings.

Game 2: Is Game 1 with the difference that $\mathcal{M}$ replaces the outputs of $\text{PRG}_{\mathbf{sk}}$ and $\text{PRG}_{\mathbf{s}}$ by random bit strings.

Game 3: Is Game 2 with the difference that $\mathcal{M}$ takes as additional input a random equation system $\mathbf{F}$. $\mathcal{M}$ simulates XOF_F towards $\mathcal{A}$, programming XOF_F such that it returns the coefficients representing $\mathbf{F}$ upon input of S_F and uniformly random values on any other input.

Per assumption, $\mathcal{A}$ wins Game 0 with non-negligible success probability. Let's call this ϵ. If the difference in $\mathcal{A}$'s success probability playing Game 0 or Game 1 was non-negligible, we could use $\mathcal{A}$ to distinguish the outputs of $\text{PRG}_{\mathbf{rte}}$ from random. The same argument applies for the difference between Game 1 and Game 2, and $\text{PRG}_{\mathbf{sk}}$ and $\text{PRG}_{\mathbf{s}}$. Finally, the output distribution of XOF_F in Game 3 is the same as in previous games. Hence, there is no difference for $\mathcal{A}$ between Game 2 and Game 3. Accordingly, $\mathcal{A}$'s success probability in these two games is equal.

Now, Game 3 is exactly the EU-CMA game for the optimized $opt\text{-}q2$ signature scheme that is derived from $\mathcal{MQ}$-IDS, the 5-pass IDS from [41].

Next, recall that under the assumption of intractability of the $\mathcal{MQ}$ problem on average and assuming computationally binding and computationally hiding properties of Com_0 and Com_1, $\mathcal{MQ}$-IDS is KOW (c.f. Theorem 3.1), is HVZK (c.f. Theorem 4) and has a $q2$-extractor (c.f. Theorem 7). We can now apply Corollary A.6 on $\mathcal{MQ}$-IDS, and obtain that the $opt\text{-}q2$ signature scheme derived from $\mathcal{MQ}$-IDS is EU-CMA secure. This is a contradiction to the assumption that $\mathcal{A}$ wins Game 3 with non-negligible probability. □

NewHope

Algorithm Specifications and Supporting Documentation

Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas,
Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, Douglas Stebila

Version 1.0 - November 30, 2017

Contents

1 Written specification

1.1 Mathematical background

1.1.1 Basic definitions

Let $\mathbb{Z}$ be the ring of rational integers. We define for an $x \in \mathbb{R}$ the rounding function $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor \in \mathbb{Z}$. Let $\mathbb{Z}_q$, for an integer $q \geq 1$, denote the quotient ring $\mathbb{Z}/q\mathbb{Z}$. We define $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ as the ring of integer polynomials modulo $X^n + 1$. By $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ we mean the ring of integer polynomials modulo $X^n + 1$ where each coefficient is reduced modulo q. In case χ is a probability distribution over $\mathcal{R}$, then $x \xleftarrow{\$} \chi$ means the sampling of $x \in \mathcal{R}$ according to χ.

For a probabilistic algorithm $\mathcal{A}$ we denote by $y \xleftarrow{\$} \mathcal{A}$ that the output of $\mathcal{A}$ is assigned to y and that $\mathcal{A}$ is running with randomly chosen coins. We recall the discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$, which is parametrized by the Gaussian parameter $\sigma \in \mathbb{R}$ and defined by assigning a weight proportional to $\exp(\frac{-x^2}{2\sigma^2})$ to all integers x.

The Euclidean length $\|\mathbf{v}\|$ of a vector $\mathbf{v} = (v_1, \dots v_n) \in \mathbb{R}^n$ is defined as $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}$. A lattice is a discrete subgroup of a finite dimensional Euclidean vector space, *i.e.* a discrete subgroup $\mathcal{L} \subset \mathbb{R}^n$. The minimal distance of a lattice is defined as the Euclidean length of its shortest non-zero vector, namely $\lambda_1(\mathcal{L}) = \min_{\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\mathbf{x}\|$.

1.1.2 Computational problems on lattices

The shortest vector problem (SVP) and the closest vector problem (CVP) are two fundamental problems in lattices and their conjectured intractability is the foundation for a large number of cryptographic applications of lattices.

The (Approximate) Shortest Vector Problem, (SVP), statement from [106]. The shortest vector problem (SVP) asks, given a lattice basis $\mathbf{B}$, to find a shortest nonzero lattice vector, i.e., a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ with $\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$. In the γ-approximate SVP_γ, for $\gamma \geq 1$, the goal is to find a shortest nonzero lattice vector $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{\mathbf{0}\}$ of norm at most $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L}(\mathbf{B}))$.

The SVP asks for *a* shortest nonzero vector in a lattice, but not *the* shortest nonzero vector as several short vectors can exist. The approximate SVP_γ is more difficult for a small factor γ and becomes easier for an increasing γ. An algorithm that solves SVP in polynomial time and with exponential approximation factor $2^{\mathcal{O}(n)}$ is the Lenstra, Lenstra, Lovász (LLL) algorithm [96], which was extended in works like [135, 134, 64] (see [113] for a survey). Algorithms that achieve an exact solution or approximate solutions of SVP within $\text{poly}(n)$ factors either run in $2^{\mathcal{O}(n)}$ and require exponential space [4] or in $2^{\mathcal{O}(n \log n)}$ and require only polynomial space [89]. Based on these observations Micciancio and Regev conclude that "there is no polynomial time algorithm that approximates lattice problems to within polynomial factors" [108].

The (Approximate) Closest Vector Problem, (CVP), statement from [106]. The closest vector problem (CVP) asks, given a lattice basis $\mathbf{B}$ and target vector $\mathbf{t}$, to find the lattice vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that the distance to the target $\|\mathbf{v} - \mathbf{t}\|$ is minimized. In the γ-approximate CVP_γ, for $\gamma \geq 1$, the goal is to find a lattice vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that $\|\mathbf{v} - \mathbf{t}\| \leq \gamma \cdot \text{dist}(\mathbf{t}, \mathcal{L}(\mathbf{B}))$ where $\text{dist}(\mathbf{t}, \Lambda) = \inf\{\|\mathbf{v} - \mathbf{t}\| : \mathbf{v} \in \Lambda\}$ is the distance of $\mathbf{t}$ to Λ.

The CVP is the inhomogeneous version of the SVP and can also be formulated as syndrome decoding problem for full rank lattices [106]. The NP-hardness of SVP was shown by van Emde Boas in [140] for the ℓ_∞ norm. Ajtai then proved that SVP is NP-hard for the ℓ_2 norm using randomized reductions [3] and that the corresponding decision problem is NP-complete. It was also shown in [140] that CVP is NP-hard. However, when building cryptosystems in practice only subclasses of CVP or SVP are used that are not supposed to be NP-hard (see [84, Remark 6.24.]). A comprehensive discussion of the hardness of SVP, CVP, and its variants can be found in [141, Section 2.3] and [107]. In [77] Hanrot et al. provide a survey on the history and state-of-the-art of solvers for SVP and CVP.

1.1.3 Ring-LWE problem

The Learning with Errors (LWE) problem was popularized by Regev [126] who showed that, under a quantum reduction, solving a random LWE instance is as hard as solving certain worst-case instances of certain lattice problems. The LWE problem can be seen as a generalization of the learning parity with noise (LPN) problem [29] and is related to hard decoding problems [108]. In general, to solve the LWE problem, one has to recover a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ when given a sequence of approximate random linear equations on $\mathbf{s}$. Non-quantum reductions from variants of the shortest vector problem to variants of the LWE problem have also been shown [118]. The LWE problem is usually used to build primitives such as CPA or CCA-secure public-key encryption, identity-based encryption (IBE), or fully-homomorphic encryption schemes [128]. It can be defined as a search problem (sLWE) where the task is to recover the secret vector $\mathbf{s}$ or as a decision problem (dLWE) that asks to distinguish LWE samples from uniformly random samples.

The Learning With Errors Problem [126], (sLWE), search version. The learning with errors problem, search version, $\text{sLWE}_{n,m,q,\chi}$, with n unknowns, $m \geq n$ samples, modulo q and with error distribution χ is as follows: for a random secret s uniformly chosen in $\mathbb{Z}_q^n$, and given m samples of the form $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \mod q)$ where $e \xleftarrow{\$} \chi$ and $\mathbf{a}$ is uniform in $\mathbb{Z}_q^n$, recover the secret vector $\mathbf{s}$.

The Learning With Errors Problem [126], (dLWE), decisional version. The learning with errors problem, decisional version, $\text{dLWE}_{n,m,q,\chi}$, with n unknowns, $m \geq n$ samples, modulo q and with error distribution χ is as follows: for a random secret $\mathbf{s}$ uniformly chosen in $\mathbb{Z}_q^n$, and given m samples either all of the form $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \mod q)$ where $e \xleftarrow{\$} \chi$, or from the uniform distribution $(\mathbf{a}, b) \xleftarrow{\$} \mathcal{U}(\mathbb{Z}^n \times \mathbb{Z}_q)$, decide if the samples come from the former or the latter case.

An interesting property of the LWE problem is the equivalence of the (search) sLWE problem and the (decisional) dLWE problem. While it is clear that a solver for the sLWE problem can be used to solve the dLWE problem, it is also possible to solve the sLWE problem if the dLWE problem can be solved.

Theorem 1.1 (Decision to Search Reduction for LWE) *For any integers n and m, any prime $q \leq poly(n)$, and any distribution χ over $\mathbb{Z}_q$, if there exists a PPT algorithm that solves $\text{dLWE}_{n,m,q,\chi}$ with non-negligible probability, then there exists a PPT algorithm that solves $\text{sLWE}_{n,m',q,\chi}$ for some $m' = m \cdot poly(n)$ with non-negligible probability.*

Variants of the LWE problem relying on the ring of integer of a number field (or polynomial rings) were later defined and studied [138, 103]. More specifically, the lattices underlying this problem are module lattices, as in NTRU [83, 137], and its hardness can be related to the worst case hardness of finding short vectors in ideal lattices [138, 103]. The Ring-LWE problem may be defined over the ring of integers of an arbitrary number-field [104, 121]. The general definition is rather intricate involving the so-called co-different ideal $R^\vee$. For simplicity we restrict our definition to the case of cyclotomic number field with a power-of-two conductor.

The Ring Learning With Errors Problem [126], dRLWE, decisional version Let R denote the ring $Z[X]/(X^n + 1)$ for n a power of 2, and R_q the residue ring R/qR. The ring learning with errors problem, decisional version, $\text{dRLWE}_{m,q,\chi}$, with m unknowns, $m \geq 1$ samples, modulo q and with error distribution χ is as follows: for a uniform random secret $\mathbf{s} \xleftarrow{\$} \mathcal{U}(R_q)$, and given m samples either all of the form $(\mathbf{a}, \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \mod q)$ where the coefficients of $\mathbf{e}$ are independently sampled following the distribution χ, or from the uniform distribution $(\mathbf{a}, \mathbf{b}) \xleftarrow{\$} \mathcal{U}(R_q \times R_q)$, decide if the samples come from the former or the latter case.

We will in fact rely on a variant of the above problem, where the secret $\mathbf{s}$ follows the same distribution χ^n as the error $\mathbf{e}$. These variants can be proven to be equivalent to the original problem by putting the system in systematic form, as done in [13].

1.2 Algorithm description

The NewHope cryptosystem is a suite of key encapsulation mechanisms (KEM) denoted as NewHope-CPA-KEM and NewHope-CCA-KEM that are based on the conjectured quantum hardness of the RLWE

problem. Both schemes are based on a variant of the previously proposed NewHope-Simple [8] scheme modeled as semantically secure public-key encryption (PKE) scheme with respect to adaptive chosen plaintext attacks (CPA) that we refer to as NewHope-CPA-PKE. However, in this submission NewHope-CPA-PKE is only used inside of NewHope-CPA-KEM and NewHope-CCA-KEM and not intended to be an independent CPA-secure PKE scheme, in part because it does not accept arbitrary length messages. For our proposed NewHope-CPA-KEM we provide a transformation of NewHope-CPA-PKE into a passively secure KEM. For NewHope-CCA-KEM we show how to realize a semantically secure key encapsulation with respect to adaptive chosen ciphertext attacks (CCA) based on NewHope-CPA-PKE. In this section we only provide a functional description of the algorithms and refer to Section 1.3 for more background on our design decisions. Note that some algorithms, especially the ones dealing with encoding and decoding, are optimized for dimensions $n = 512$ or $n = 1024$ that we are supporting in NewHope. Moreover, the algorithms are designed for $q = 12289$ and $k = 8$ (parameter of the noise distribution). Further information on the supported parameters can be found in Section 1.4.

1.2.1 IND-CPA-secure public key encryption scheme

For our intermediate building block, the passively secure PKE scheme NewHope-CPA-PKE with a fixed message space of 256 bits, we define key generation in Algorithm 1 (NewHope-CPA-PKE.Gen), encryption in Algorithm 2 (NewHope-CPA-PKE.Encrypt), and decryption in Algorithm 3 (NewHope-CPA-PKE.Decrypt). All sub-functions used in NewHope-CPA-PKE are described in this section. Note that we assume in every function implicit access to the global parameters n, q, γ that are determined by the chosen parameter set.

Sampling, randomness, byte arrays and SHAKE. Besides polynomials in $\mathcal{R}_q$ and vectors, the main other data structure we use are byte arrays. As an example, all randomness is sampled as byte arrays. In key generation, $seed \xleftarrow{\$} \{0, \ldots, 255\}^{32}$ denotes the sampling of a byte array with 32 uniform integer elements in the range 0 to 255 from a random number generator. This random number generator shall be unpredictable and should thus be using a physical source of entropy or other means.

As strong hash function we use SHAKE256 as specified in [112]. The SHAKE256(l, d) function takes as input an integer l that specifies the number of output bytes and an input data byte array d. The amount of data to be absorbed is the length of d. As an example, in key generation we use SHAKE256 to compute $v \leftarrow$ SHAKE256$(64, seed)$ where we hash a 32 byte random seed denoted as $seed$ and output a byte array v with 64 elements in the range $\{0, \ldots, 255\}$.

To access byte arrays we use the bracket notation where $v[i]$ for a positive integer i denotes the i-th byte in the array v. To access ranges of bytes we use the notation $x \leftarrow v[i : j]$ for positive integers $i \leq j$ where x is assigned byte i to j of v. By $r \leftarrow \{0, \ldots, 255\}^x$ we declare that r is a byte array of length x. Using a similar notation, by $\mathbf{r} \leftarrow \mathcal{R}_q$ we declare that a variable $\mathbf{r}$ is a polynomial in $\mathcal{R}_q$ where all coefficients are zero. For bit-operations we use the operators $\gg$, $\ll$, $|$, and $\&$ as in the context of the C programming language. Thus $x \gg i$ for positive integers i, x denotes a right-shift by i. The same operator can be applied to a byte of a byte array so that $y[j] \gg i$ for positive integer i, j represents a right shift by i of the j-th byte of the byte array y. A left shift is denoted as $x \ll i$ for positive integers i, y and implicit modular reduction modulo 2^{32} is assumed (equal to writing $(x \ll i) \bmod 2^{32}$). When the left shift operator is applied to the j-th byte of the byte array y as $y[j] \ll i$ for positive integer i, j an implicit reduction modulo 2^8 is assumed (equal to writing $(y[j] \ll i) \bmod 2^8$). The $a \,|\, b$ operator denotes a bit-wise 'or' while the $a \,\&\, b$ operator denotes a bitwise 'and' of two positive integers a, b or of two bytes in a byte array. To convert a byte $a[i]$ in a byte array a to a positive integer z we use $z =$ b2i$(a[i])$. To denote positive integers in hexadecimal representation we use the prefix $0x$ such that 0x01010101 $= 16843009$. To compute the Hamming weight, the sum of all bits that are set to one in binary notation, of a byte or integer b we write HW(b).

Note that NewHope-CPA-PKE.Encrypt does not directly access a random number generator as all pseudo-random data is derived by expansion of a 32-byte user supplied seed $coin \in \{0, \ldots, 255\}^{32}$ that has to be obtained from a true random value generator. This is required to allow the straightforward use of NewHope-CPA-PKE.Encrypt in standard CCA transformations. Decryption is deterministic and does not need random values. For the distribution of the RLWE secret and error we use the centered binomial distribution ψ_k of parameter $k = 8$. In general, one may sample from ψ_k for integer $k > 0$ by computing $\sum_{i=0}^{k-1} b_i - b'_i$, where the $b_i, b'_i \in \{0, 1\}$ are uniform independent bits. The distribution ψ_k is centered (its mean

Algorithm 1 NewHope-CPA-PKE Key Generation

function NewHope-CPA-PKE.Gen()
  $seed \xleftarrow{\$} \{0,\ldots,255\}^{32}$
  $z \leftarrow \mathsf{SHAKE256}(64, seed)$
  $publicseed \leftarrow z[0{:}31]$
  $noiseseed \leftarrow z[32{:}63]$
  $\hat{\mathbf{a}} \leftarrow \mathsf{GenA}(publicseed)$
  $\mathbf{s} \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(noiseseed, 0))$
  $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
  $\mathbf{e} \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(noiseseed, 1))$
  $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
  $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
  return $(pk = \mathsf{EncodePK}(\hat{\mathbf{b}}, publicseed), sk = \mathsf{EncodePolynomial}(\mathbf{s}))$

Algorithm 2 NewHope-CPA-PKE Encryption

function NewHope-CPA-PKE.Encrypt($pk \in \{0,\ldots,255\}^{7\cdot n/4+32}$, $\mu \in \{0,\ldots,255\}^{32}$, $coin \in \{0,\ldots,255\}^{32}$)
  $(\hat{\mathbf{b}}, publicseed) \leftarrow \mathsf{DecodePk}(pk)$
  $\hat{\mathbf{a}} \leftarrow \mathsf{GenA}(publicseed)$
  $\mathbf{s}' \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(coin, 0))$
  $\mathbf{e}' \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(coin, 1))$
  $\mathbf{e}'' \leftarrow \mathsf{Sample}(coin, 2)$
  $\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
  $\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
  $\mathbf{v} \leftarrow \mathsf{Encode}(\mu)$
  $\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{v}$
  $h \leftarrow \mathsf{Compress}(\mathbf{v}')$
  return $c = \mathsf{EncodeC}(\hat{\mathbf{u}}, h)$

Algorithm 3 NewHope-CPA-PKE Decryption

function NewHope-CPA-PKE.Decrypt($c \in \{0,\ldots,255\}^{7\frac{n}{4}+3\frac{n}{8}}$, $sk \in \{0,\ldots,255\}^{7\cdot n/4}$)
  $(\hat{\mathbf{u}}, h) \leftarrow \mathsf{DecodeC}(c)$
  $\hat{\mathbf{s}} \leftarrow \mathsf{DecodePolynomial}(sk)$
  $\mathbf{v}' \leftarrow \mathsf{Decompress}(h)$
  $\mu \leftarrow \mathsf{Decode}(\mathbf{v}' - \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}}))$
  return μ

is 0), has variance $k/2$ and we set $k = 8$ in all instantiations. This gives a standard deviation of $\varsigma = \sqrt{8/2}$. We describe sampling from ψ_8 in Algorithm 4 as the function Sample that takes as input a 32 byte seed *seed* and an integer parameter $0 \leq nonce < 2^8$ for domain separation. This way one seed can be used to sample multiple polynomials. The output is a polynomial $\mathbf{r} \in \mathcal{R}_q$ where all n coefficients are independently distributed according to ψ_8.

Polynomials and the NTT. The main mathematical objects that are manipulated in NewHope are polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ like $\mathbf{s}, \mathbf{e}, \hat{\mathbf{s}}, \hat{\mathbf{a}}, \hat{\mathbf{b}}, \mathbf{s}', \mathbf{e}', \mathbf{e}'', \hat{\mathbf{t}}, \hat{\mathbf{u}}, \hat{\mathbf{e}}, \mathbf{v}, \mathbf{v}'$. For a polynomial $\mathbf{c} \in \mathcal{R}_q$ where $\mathbf{c} = \sum_{i=0}^{n-1} c_i X^i$ we denote by c_i the i-th coefficient of $\mathbf{c}$ for integer $i \in \{0,\ldots,n-1\}$. We use the same notation to access elements of vectors that are not necessary in $\mathcal{R}_q$. Addition or subtraction of polynomials in $\mathcal{R}_q$ (denoted as $+$ or $-$, respectively) is the usual coefficient-wise addition or subtraction,

Algorithm 4 Deterministic sampling of polynomials in $\mathcal{R}_q$ from ψ_8^n

function SAMPLE($seed \in \{0,\ldots,255\}^{32}$, positive integer $nonce$)
  $\mathbf{r} \leftarrow \mathcal{R}_q$
  $extseed \leftarrow \{0,\ldots,255\}^{34}$
  $extseed[0{:}31] \leftarrow seed[0{:}31]$
  $extseed[32] \leftarrow nonce$
  for i **from** 0 **to** $(n/64)-1$ **do**
    $extseed[33] \leftarrow i$
    $buf \leftarrow \mathsf{SHAKE256}(128, extseed)$
    for j **from** 0 **to** 63 **do**
      $a \leftarrow buf[2*j]$
      $b \leftarrow buf[2*j+1]$
      $r_{64*i+j} = \mathsf{HW}(a) + q - \mathsf{HW}(b) \bmod q$
  return $\mathbf{r} \in \mathcal{R}_q$

such that for $\mathbf{a} = \sum_{i=0}^{n-1} a_i X^i \in \mathcal{R}_q$ and $\mathbf{b} = \sum_{i=0}^{n-1} b_i X^i \in \mathcal{R}_q$ we get $\mathbf{a}+\mathbf{b} = \sum_{i=0}^{n-1}(a_i + b_i \mod q)X^i$ and $\mathbf{a}-\mathbf{b} = \sum_{i=0}^{n-1}(a_i - b_i \mod q)X^i$. In general, fast quasi-logarithmic algorithms exist for polynomial multiplication. We explicitly specify how to use the Number Theoretic Transform (NTT); some polynomials are also transmitted in a transformed representation. However, an implementer may choose a different algorithm for polynomial multiplication, like Karatsuba or Schoolbook multiplication, and then transform the result into the NTT domain such that it is compliant with this specification. Moreover, here we just describe the basic definition and refer to [76, 2] and Section 2 for details on the efficient implementation of the NTT.

With the NTT, a polynomial multiplication for elements in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$ can be performed by computing $\mathbf{c} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ for $\mathbf{a},\mathbf{b},\mathbf{c} \in \mathcal{R}_q$. The $\circ$ operator denotes coefficient-wise multiplication of two polynomials $\mathbf{a},\mathbf{b} \in \mathcal{R}_q$ such that $\mathbf{a} \circ \mathbf{b} = \sum_{i=0}^{n-1}(a_i \cdot b_i \mod q)X^i$. The NTT defined in $\mathcal{R}_q$ can be implemented very efficiently if n is a power of two and q is a prime for which it holds that $q \equiv 1 \mod 2n$. This way a primitive n-th root of unity ω and its square root $\gamma = \sqrt{\omega} \bmod q$ exist. By multiplying coefficient-wise by powers of γ before the NTT computation and after the reverse transformation by powers of $\gamma^{-1} \bmod q$, no zero padding is required and an n-point NTT can be used to transform a polynomial with n coefficients.

For a polynomial $\mathbf{g} = \sum_{i=0}^{n-1} g_i X^i \in \mathcal{R}_q$ we define

$$\mathsf{NTT}(\mathbf{g}) = \hat{\mathbf{g}} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$
$$\hat{g}_i = \sum_{j=0}^{n-1} \gamma^j g_j \omega^{ij} \bmod q,$$

where ω is an n-th primitive root of unity and $\gamma = \sqrt{\omega} \bmod q$.

Note that most implementations will use an in-place NTT algorithm which usually requires bit-reversal operations that are not included in the previously given straightforward description of the NTT. As an optimization, we allow implementations to skip these bit-reversals for the forward transformation as all inputs are only random noise. Thus, and slightly counter-intuitive, we define bit-reversal and perform it on polynomials that go into the NTT. With bit-reversal and our straightforward NTT definition, implementers do not need to apply a reversal when using an in-place NTT. Note that in key generation this optimization is transparent to the protocol, but due to the re-encryption implementers have to follow our instructions. For a positive integer v and a power of two n we formally define bit-reversal as $\mathsf{BitRev}(v) = \sum_{i=0}^{\log_2(n)-1}(((v \gg i)\&1) \ll (\log_2(n)-1-i))$. For polynomials $\mathbf{s},\mathbf{z} \in \mathcal{R}_q$ the bit-reversal of a polynomial $\mathbf{s}$ is $\mathbf{z} = \mathsf{PolyBitRev}(\mathbf{s}) = \sum_{i=0}^{n-1} s_i X^{\mathsf{BitRev}(i)}$.

The function NTT^{-1} is the inverse of the function NTT. The computation of NTT^{-1} is essentially the same as the computation of NTT, except that it uses $\omega^{-1} \mod q$, multiplies by powers of $\gamma^{-1} \mod q$ after

the summation, and also multiplies each coefficient by the scalar $n^{-1} \mod q$ so that

$$\mathsf{NTT}^{-1}(\hat{\mathbf{g}}) = \mathbf{g} = \sum_{i=0}^{n-1} g_i X^i, \text{ with}$$
$$g_i = \left(n^{-1} \gamma^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij} \right) \mod q.$$

Note that we define the $x \mod q$ operation for integers x, q to always produce an output in the range $[0, q-1]$. Unless otherwise stated, when we access an element a_i of a polynomial $\mathbf{a} \in \mathcal{R}_q$ we always assume that a_i is reduced modulo q and in the range $[0, q-1]$.

Definition of GenA. The public parameter $\mathbf{a}$ is generated by GenA which takes as input a 32 byte array *seed*. The function is described in Algorithm 5. The resulting polynomial $\mathbf{a}$ (denoted as $\hat{\mathbf{a}}$) is considered to be in the NTT domain. This is possible because the NTT transforms uniform polynomials to uniform polynomials. Inside of GenA we use the SHAKE128 hash function [112] to expand the pseudorandom seed and we define a function that absorbs a byte array into the internal state of SHAKE128 and then we use another function to obtain pseudorandom data by squeezing the internal state. The $state \leftarrow \mathsf{SHAKE128Absorb}(d)$ functions takes as input a byte array d. It outputs a byte array of length 200 that represents the state after absorbing d. To obtain pseudorandom values $buf, state \leftarrow \mathsf{SHAKE128Squeeze}(j, state)$ is used. As input the function takes a positive integer j determining the amount of output blocks of SHAKE128 to be produced and the 200 byte state. It outputs a byte array buf of length $168 \cdot j$ and a byte array *state* of length 200.

Algorithm 5 Deterministic generation of $\hat{\mathbf{a}}$ by expansion of a seed

function GenA($seed \in \{0, \ldots, 255\}^{32}$)
  $\hat{\mathbf{a}} \leftarrow \mathcal{R}_q$
  $extseed \leftarrow \{0, \ldots, 255\}^{33}$
  $extseed[0:31] \leftarrow seed[0{:}31]$
  for i **from** 0 **to** $(n/64) - 1$ **do**
    $ctr \leftarrow 0$
    $extseed[32] \leftarrow i$
    $state \leftarrow \mathsf{SHAKE128Absorb}(extseed)$
    while $ctr < 64$ **do**
      $buf, state \leftarrow \mathsf{SHAKE128Squeeze}(1, state)$
      $j \leftarrow 0$
      for $j < 168$ **and** $ctr < 64$ **do**
        $val \leftarrow \mathsf{b2i}(buf[j]) \,|\, (\mathsf{b2i}(buf[j+1]) \ll 8)$
        if $val < 5 \cdot q$ **then**
          $\hat{a}_{i*64+ctr} \leftarrow val$
          $ctr \leftarrow ctr + 1$
        $j \leftarrow j + 2$
  return $\hat{\mathbf{a}} \in \mathcal{R}_q$

Encoding and decoding of the secret and public key. Note that polynomials are transmitted in the NTT domain and thus for interoperability our definition and parametrization of the NTT has to be used.

To encode a polynomial in $\mathcal{R}_q$ into an array of bytes we use EncodePolynomial as described in Algorithm 6. The function DecodePolynomial as described in Algorithm 7 converts a byte array into an element in $\mathcal{R}_q$. The secret key consists only of one polynomial $\mathbf{s} \in \mathcal{R}_q$ and thus we can directly apply $\mathsf{EncodePolynomial}(\hat{\mathbf{s}})$. The secret key is then either encoded into an array of 869 bytes ($n = 512$) or 1792 bytes ($n = 1024$). The public key is encoded as an array of 928 bytes ($n = 512$) or 1824 bytes ($n = 1024$) by $\mathsf{EncodePK}(\hat{\mathbf{b}}, seed)$ described in Algorithm 8. It takes as input a polynomial $\hat{\mathbf{b}} \in \mathcal{R}_q$ and a byte array *seed* with 32 elements. The $\mathsf{DecodePk}(pk)$ function decodes the public key and is provided in Algorithm 9.

Algorithm 6 Encoding of a polynomial in $\mathcal{R}_q$ to a byte array

function EncodePolynomial($\hat{\mathbf{s}}$)
  $r \leftarrow \{0,\ldots,255\}^{7\cdot n/4}$
  for i **from** 0 **to** $n/4-1$ **do**
    $t0 \leftarrow \hat{s}_{4*i+0} \mod q$
    $t1 \leftarrow \hat{s}_{4*i+1} \mod q$
    $t2 \leftarrow \hat{s}_{4*i+2} \mod q$
    $t3 \leftarrow \hat{s}_{4*i+3} \mod q$
    $r[7*i+0] \leftarrow t0\&\texttt{0xff}$
    $r[7*i+1] \leftarrow (t0 \gg 8)\,|(t1 \ll 6)\&\texttt{0xff}$
    $r[7*i+2] \leftarrow (t1 \gg 2)\&\texttt{0xff}$
    $r[7*i+3] \leftarrow (t1 \gg 10)\,|(t2 \ll 4)\&\texttt{0xff}$
    $r[7*i+4] \leftarrow (t2 \gg 4)\&\texttt{0xff}$
    $r[7*i+5] \leftarrow (t2 \gg 12)\,|(t3 \ll 2)\&\texttt{0xff}$
    $r[7*i+6] \leftarrow (t3 \gg 6)\&\texttt{0xff}$
  return $r \in \{0,\ldots,255\}^{7\cdot n/4}$

Algorithm 7 Decoding of a polynomial represented as a byte array into an element in $\mathcal{R}_q$

function DecodePolynomial($v \in \{0,\ldots,255\}^{7\cdot n/4}$)
  for i **from** 0 **to** $n/4-1$ **do**
    $r \leftarrow \mathcal{R}_q$
    $r_{4*i+0} \leftarrow \mathsf{b2i}(v[7*i+0])\|((\mathsf{b2i}(v[7*i+1])\,\&\,\texttt{0x3f}) \ll 8)$
    $r_{4*i+1} \leftarrow (\mathsf{b2i}(v[7*i+1]) \gg 6)\|(\mathsf{b2i}(v[7*i+2]) \ll 2)\|((\mathsf{b2i}(v[7*i+3])\,\&\,\texttt{0x0f}) \ll 10)$
    $r_{4*i+2} \leftarrow (\mathsf{b2i}(v[7*i+3]) \gg 4)\|(\mathsf{b2i}(v[7*i+4]) \ll 4)\|((\mathsf{b2i}(v[7*i+5])\,\&\,\texttt{0x03}) \ll 12)$
    $r_{4*i+3} \leftarrow (\mathsf{b2i}(v[7*i+5]) \gg 2)\|(\mathsf{b2i}(v[7*i+6]) \ll 6)$
  return $\mathbf{r} \in \mathcal{R}_q$

Algorithm 8 Encoding of the public key

function EncodePK($\hat{\mathbf{b}} \in \mathcal{R}_q, publicseed \in \{0,\ldots,255\}^{32}$)
  $r \leftarrow \{0,\ldots,255\}^{7\cdot n/4+32}$
  $r[0:7\cdot n/4-1] \leftarrow \mathsf{EncodePolynomial}(\hat{\mathbf{b}})$
  $r[7\cdot n/4 : 7\cdot n/4+31] \leftarrow publicseed[0:31]$
  return $r \in \{0,\ldots,255\}^{7\cdot n/4+32}$

Algorithm 9 Decoding of the public key

function DecodePk($pk \in \{0,\ldots,255\}^{7\cdot n/4+32}$)
  $\hat{\mathbf{b}} \leftarrow \mathsf{DecodePolynomial}(pk[0:7\cdot n/4-1])$
  $seed \leftarrow pk[7\cdot n/4 : 7\cdot n/4+31])$
  return $(\hat{\mathbf{b}} \in \mathcal{R}_q, seed \in \{0,\ldots,255\}^{32})$

Encoding and decoding of the ciphertext. The ciphertext encoding is described in Algorithm 13. The ciphertext c is encoded as an array of 1088 bytes ($n = 512$) or 2176 bytes ($n = 1024$) by $\mathsf{EncodeC}(\hat{\mathbf{u}}, h)$ that takes as input a polynomial in $\hat{\mathbf{u}} \in \mathcal{R}_q$ and an array h of $3 \cdot n/8$ bytes that was generated by $\mathsf{Compress}(\mathbf{v}')$ as given in Algorithm 12. The compression and decompression functions simply perform coefficient-wise modulus switching between modulus q and modulus 8 by multiplying by the new modulus and then performing a rounding division by the old modulus. To decode the ciphertext the $\mathsf{DecodeC}$ function is used that outputs $\hat{\mathbf{u}} \in \mathcal{R}_q$ and a byte array h that is then given to $\mathsf{Decompress}$ to obtain $\mathbf{v}' \in \mathcal{R}_q$.

In NewHope-CPA-PKE the 256-bit message μ represented as an array of 32 bytes has to be encoded into an element in $\mathcal{R}_q$ during encryption and decoded from an element in $\mathcal{R}_q$ into a byte array during decryption. To allow robustness against errors each bit of the 256-bit message $\mu \in \{0, \ldots, 255\}^{32}$ is encoded into $\lfloor n/256 \rfloor$ coefficients by Encode (see Algorithm 10). The decoding function Decode (see Algorithm 11) maps from $\lfloor n/256 \rfloor$ coefficients back to the original key bit. For example, for $n = 1024$, take $4 = \lfloor 1024/256 \rfloor$ coefficients (each in the range $\{0, \ldots, q-1\}$, subtract $\lfloor q/2 \rfloor$ from each of them, accumulate their absolute values, and set the key bit to 0 if the sum is larger than q or to 1 otherwise.

Algorithm 10 Message encoding

```
1:  function Encode(μ ∈ {0,...,255}^32)
2:      v ← R_q
3:      for i from 0 to 31 do
4:          for j from 0 to 7 do
5:              mask ← −((msg[i] ≫ j) & 1)
6:              v_{8∗i+j+0} ← mask &(q/2)
7:              v_{8∗i+j+256} ← mask &(q/2)
8:              if n equals 1024 then
9:                  v_{8∗i+j+512} ← mask &(q/2)
10:                 v_{8∗i+j+768} ← mask &(q/2)
11:     return v ∈ R_q
```

Algorithm 11 Message decoding

```
1:  function Decode(v ∈ R_q)
2:      μ ← {0,...,255}^32
3:      for i from 0 to 255 do
4:          t ← |(v_{i+0}  mod q) − (q − 1)/2|
5:          t ← t + |(v_{i+256}  mod q) − (q − 1)/2|
6:          if n equals 1024 then
7:              t ← t + |(v_{i+512}  mod q) − (q − 1)/2|
8:              t ← t + |(v_{i+768}  mod q) − (q − 1)/2|
9:              t ← ((t − q))
10:         else
11:             t ← ((t − q/2))
12:         t ← t ≫ 15
13:         μ[i ≫ 3] ← μ[i ≫ 3] |(t ≪ (i & 7))
14:     return μ ∈ {0,...,255}^32
```

1.2.2 Interconversion to IND-CPA KEM

NewHope-CPA-PKE can be converted to an IND-CPA-secure key encapsulation mechanism by using the public key encryption scheme to convey a secret, K. The PKE's coins and the secret K are computed by

Algorithm 12 Ciphertext compression

function COMPRESS($\mathbf{v}' \in \mathcal{R}_q$)
  $k \leftarrow 0$
  $t \leftarrow \{0, \ldots, 255\}^8$
  $h \leftarrow \{0, \ldots, 255\}^{3 \cdot n/8}$
  for ℓ **from** 0 **to** $n/8 - 1$ **do**
    $i \leftarrow 8 \cdot \ell$
    for j **from** 0 **to** 7 **do**
      $t[j] \leftarrow v'_{i+j} \mod q$
      $t[j] \leftarrow ((\mathsf{b2i}(t[j] \ll 3) + q/2)/q) \,\&\, \mathsf{0x7}$
    $h[k+0] \leftarrow t[0] \,|\, (t[1] \ll 3) \,|\, (t[2] \ll 6)$
    $h[k+1] \leftarrow (t[2] \gg 2) \,|\, (t[3] \ll 1) \,|\, (t[4] \ll 4) \,|\, (t[5] \ll 7)$
    $h[k+2] \leftarrow (t[5] \gg 1) \,|\, (t[6] \ll 2) \,|\, (t[7] \ll 5)$
    $k+ \leftarrow 3$
  return r $in\{0, \ldots, 255\}^{3 \cdot n/8}$

Algorithm 13 Ciphertext encoding

function ENCODEC($\hat{\mathbf{u}} \in \mathcal{R}_q, h \in \{0, \ldots, 255\}^{3 \cdot n/8}$)
  $c[0 : (7 \cdot n/4 - 1)] \leftarrow \mathsf{EncodePolynomial}(\hat{\mathbf{u}})$
  $c[(7 \cdot n/4) : (7 \cdot n/4 + 3 \cdot n/8 - 1)] \leftarrow h$
  return $c \in \{0, \ldots, 255\}^{7 \cdot n/4 + 3 \cdot n/8}$

Algorithm 14 Ciphertext decoding

function DECODEC($c \in \{0, \ldots, 255\}^{7 \cdot n/4 + 3 \cdot n/8}$)
  $\hat{\mathbf{u}} \leftarrow \mathsf{DecodePolynomial}(c[0 : (7 \cdot n/4 - 1)])$
  $h \leftarrow c[(7 \cdot n/4) : (7 \cdot n/4 + 3 \cdot n/8 - 1)]$
  return $(\hat{\mathbf{u}} \in \mathcal{R}_q, h \in \{0, \ldots, 255\}^{3 \cdot n/8})$

Algorithm 15 Ciphertext decompression

function DECOMPRESS($h \in \{0, \ldots, 255\}^{3 \cdot n/8}$)
  $k \leftarrow 0$
  for ℓ **from** 0 **to** $n/8 - 1$ **do**
    $i \leftarrow 8 \cdot \ell$
    $r_{i+0} \leftarrow a[k+0] \,\&\, 7$
    $r_{i+1} \leftarrow (a[k+0] \gg 3) \,\&\, 7$
    $r_{i+2} \leftarrow (a[k+0] \gg 6) \,|\, ((a[1] \ll 2) \,\&\, 4)$
    $r_{i+3} \leftarrow (a[k+1] \gg 1) \,\&\, 7$
    $r_{i+4} \leftarrow (a[k+1] \gg 4) \,\&\, 7$
    $r_{i+5} \leftarrow (a[k+1] \gg 7) \,|\, ((a[2] \ll 1) \,\&\, 6)$
    $r_{i+6} \leftarrow (a[k+2] \gg 2) \,\&\, 7$
    $r_{i+7} \leftarrow (a[k+2] \gg 5)$
    $k \leftarrow k + 3$
    for j **from** 0 **to** 7 **do**
      $r_{i+j} \leftarrow (r_{i+j} * q + 4) \gg 3$
  return $\mathbf{v}' \in \mathcal{R}_q$

hashing random coins, rather than using random coins directly, to protect against attacks involving disclosure of system randomness. The final shared secret is derived from the secret K by hashing. The resulting algorithms for NewHope-CPA-KEM are shown in Algorithms 16, 17, and 18.

Algorithm 16 NewHope-CPA-KEM Key Generation

function NewHope-CPA-KEM.Gen()
  $(pk, sk) \xleftarrow{\$}$ NewHope-CPA-PKE.Gen()
  return (pk, sk)

Algorithm 17 NewHope-CPA-KEM Encapsulation

function NewHope-CPA-KEM.Encaps(pk)
  $coin \xleftarrow{\$} \{0,\ldots,255\}^{32}$
  $K \| coin' \leftarrow \mathsf{SHAKE256}(64, coin) \in \{0,\ldots,255\}^{32+32}$
  $c \leftarrow$ NewHope-CPA-PKE.Encrypt$(pk, K; coin')$
  $ss \leftarrow \mathsf{SHAKE256}(32, K)$
  return (c, ss)

Algorithm 18 NewHope-CPA-KEM Decapsulation

function NewHope-CPA-KEM.Decaps(c, sk)
  $K' \leftarrow$ NewHope-CPA-PKE.Decrypt(c, sk)
  return $ss = \mathsf{SHAKE256}(32, K')$

1.2.3 Transform from IND-CPA PKE to IND-CCA KEM[1]

The Fujisaki–Okamoto transform [59] constructs an IND-CCA2-secure public-key encryption scheme from a one-way-secure public key encryption scheme in the classical random oracle model (with an assumption on the distribution of ciphertexts for each plaintext being sufficiently close to uniform). Targhi and Unruh [139] gave a variant of the Fujisaki–Okamoto transform and showed its IND-CCA2 security against a quantum adversary in the quantum random oracle model under similar assumptions. The results of both FO and TU proceed under the assumption that the public key encryption scheme has perfect correctness, which is not the case for lattice-based schemes. Hofheinz, Hövelmanns, and Kiltz [85] gave a variety of constructions in a modular fashion. We apply their $\mathsf{QFO}^{\not\perp}_m$ transform which constructs an IND-CCA-secure key encapsulation mechanism from an IND-CPA public key encryption scheme and three hash functions; following [32], we make the following modifications, denoting the resulting transform $\mathsf{QFO}^{\not\perp\prime}_m$:

- A single hash function (with longer output) is used to compute K, $coin'$, and d.
- The computation of K, $coin'$, and d also takes the public key pk as input.
- The computation of the shared secret ss also takes the encapsulation c and d as input.

$\mathsf{QFO}^{\not\perp\prime}_m$ transform Let $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a public key encryption scheme with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$, where the randomness space of $\mathsf{Encrypt}$ is $\mathcal{R}^E$. Let $\mathsf{len}_s, \mathsf{len}_K, \mathsf{len}_d, \mathsf{len}_{ss}$ be parameters. Let $G : \{0,\ldots,255\}^* \to \{0,\ldots,255\}^{\mathsf{len}_K} \times \mathcal{R}^E \times \{0,\ldots,255\}^{\mathsf{len}_d}$ and $F : \{0,\ldots,255\}^* \to \{0,\ldots,255\}^{\mathsf{len}_{ss}}$ be hash functions. Define $\mathsf{QKEM}^{\not\perp\prime}_m = \mathsf{QFO}^{\not\perp\prime}_m[\mathsf{PKE}, G, F]$ be the key encapsulation mechanism with $\mathsf{QKEM}^{\not\perp\prime}_m.\mathsf{KeyGen}$, $\mathsf{QKEM}^{\not\perp\prime}_m.\mathsf{Encaps}$ and $\mathsf{QKEM}^{\not\perp\prime}_m.\mathsf{Decaps}$ as shown in Figure 1.

[1] The text in this section is shared with the FrodoKEM submission.

function $\mathsf{QKEM}_m^{\not\perp'}$.KeyGen( )
  $(pk, sk) \xleftarrow{\$} \mathsf{PKE.KeyGen}()$
  $s \xleftarrow{\$} \{0,\ldots,255\}^{\mathsf{len}_s}$
  $\overline{sk} \leftarrow (sk, pk, s)$
  return $(pk, \overline{sk})$

function $\mathsf{QKEM}_m^{\not\perp'}$.Encaps($pk$)
  $\mu \xleftarrow{\$} \mathcal{M}$
  $(K, coin', d) \leftarrow G(pk\|\mu)$
  $c \leftarrow \mathsf{PKE.Encrypt}(pk, \mu; coin')$
  $ss \leftarrow F(K\|c\|d)$
  $\overline{c} \leftarrow (c, d)$
  return $(\overline{c}, ss)$

function $\mathsf{QKEM}_m^{\not\perp'}$.Decaps($(c, d), (sk, pk, s)$)
  $\mu' \leftarrow \mathsf{PKE.Decrypt}(c, sk)$
  $(K', coin'', d') \leftarrow G(pk\|\mu')$
  if $c = \mathsf{PKE.Encrypt}(pk, \mu'; coin'')$ **and** $d = d'$ **then**
    return $ss' \leftarrow F(K'\|c\|d)$
  else
    return $ss' \leftarrow F(s\|c\|d)$

Figure 1: Construction of an IND-CCA-secure key encapsulation mechanism $\mathsf{QKEM}_m^{\not\perp'} = \mathsf{QFO}_m^{\not\perp'}[\mathsf{PKE}, G, F]$ from a public key encryption scheme PKE and hash functions G and F.

1.2.4 IND-CCA-secure key encapsulation mechanism

NewHope-CCA-KEM is derived from NewHope-CPA-PKE by applying the $\mathsf{QFO}_m^{\not\perp'}$ transformation. The hash functions G and F are both taken to be SHAKE256. The length parameters are taken as $\mathsf{len}_s = \mathsf{len}_K = \mathsf{len}_d = \mathsf{len}_{ss} = 32$. The randomness space is $\{0,\ldots,255\}^{32}$. The message space $\mathcal{M}$ is $\{0,\ldots,255\}^{32}$. The ciphertext component c to F is instead computed as SHAKE256(c) for efficiency (a new buffer does not need to be allocated). The KEM public key also caches a hash of the public key to save on computation. Finally, some random values ($seed, rand$ in NewHope-CCA-KEM.Gen and μ in NewHope-CCA-KEM.Encaps) are computed by hashing random coins, rather than using random coins directly, to protect against attacks involving disclosure of system randomness. The resulting algorithms for NewHope-CCA-KEM are shown in Algorithms 19, 20, and 21.

1.2.5 Interconversion to IND-CCA PKE

NewHope-CCA-KEM can be converted to an IND-CCA-secure public key encryption scheme using standard conversion techniques as specified by NIST. In particular, shared secret ss can be used as the encryption key in an appropriate data encapsulation mechanism in the KEM/DEM (key encapsulation mechanism / data encapsulation mechanism) framework [46].

1.3 Design rationale

Currently, we see two main approaches to build practical lattice-based PKEs. One is to base the schemes on NTRU or a related assumption [83, 137]. The other approach we are using in this work is the LWE/RLWE assumption [102]. However, usage of the LWE assumption comes with a cost as keys become rather large. This can be avoided by relying on ideal lattices and the Ring-Learning With Errors (RLWE) assumption. While RLWE certainly features more structure than LWE, no algorithms are known that can exploit this structure and that are thus working more efficiently on RLWE than on LWE. When restricting the design space to ideal lattices due to smaller key sizes, then the seminal work by Lyubashevsky, Peikert and Regev [103, 102] (from now on referred to as LPR10) can be considered as the core basis for follow-up work like [31, 50, 120].

The traditional approach for passively secure LWE-based (and Ring-LWE-based) key encapsulation (KEM) or key exchange is derived straight-forwardly from LWE-based (or Ring-LWE-based) encryption schemes like the ones described in [127, 99, 70]: Alice generates a key pair (sk_A, pk_A), sends pk_A to Bob; Bob chooses a (symmetric) key k, encrypts this key under pk_A and sends it to Alice; Alice decrypts to obtain k. See, for example, [118, Sec. 4.2] for an adaptation of the passively secure lattice-based cryptosystem from [69] to this KEM setting. We will in the following refer to this approach as a Key Transport Mechanism (KTM) or as the *encryption-based* approach for RLWE-based key exchange.

Algorithm 19 NewHope-CCA-KEM Key Generation

function NewHope-CCA-KEM.Gen()
  $(pk, sk) \xleftarrow{\$}$ NewHope-CPA-PKE.Gen()
  $s \xleftarrow{\$} \{0,\ldots,255\}^{32}$
  return $(pk, \overline{sk} = sk\|pk\|\mathsf{SHAKE256}(32, pk)\|s)$

Algorithm 20 NewHope-CCA-KEM Encapsulation

function NewHope-CCA-KEM.Encaps(pk)
  $coin \xleftarrow{\$} \{0,\ldots,255\}^{32}$
  $\mu \leftarrow \mathsf{SHAKE256}(32, coin) \in \{0,\ldots,255\}^{32}$
  $K\|coin'\|d \leftarrow \mathsf{SHAKE256}(96, \mu\|\mathsf{SHAKE256}(32, pk)) \in \{0,\ldots,255\}^{32+32+32}$
  $c \leftarrow$ NewHope-CPA-PKE.Encrypt$(pk, \mu; coin')$
  $ss \leftarrow \mathsf{SHAKE256}(32, K\|\mathsf{SHAKE256}(32, c\|d))$
  return $(\overline{c} = c\|d, ss)$

Algorithm 21 NewHope-CCA-KEM Decapsulation

function NewHope-CCA-KEM.Decaps($\overline{c}, \overline{sk}$)
  $c\|d \leftarrow \overline{c} \in \{0,\ldots,255\}^{32+32}$
  $sk\|pk\|h\|s \leftarrow \overline{sk} \in \{0,\ldots,255\}^{32+32+32+32}$
  $\mu' \leftarrow$ NewHope-CPA-PKE.Decrypt(c, sk)
  $K'\|coin''\|d' \leftarrow \mathsf{SHAKE256}(96, \mu'\|h) \in \{0,\ldots,255\}^{32+32+32}$
  if $c =$ NewHope-CPA-PKE.Encrypt$(pk, \mu'; coin'')$ **and** $d = d'$ **then**
    $fail \leftarrow 0$
  else
    $fail \leftarrow 1$
  $K_0 \leftarrow K'$
  $K_1 \leftarrow s$
  return $ss = \mathsf{SHAKE256}(32, K_{fail}\|\mathsf{SHAKE256}(32, c\|d))$

A slightly different approach is what we will in the following call the *reconciliation-based approach*. Instead of letting Bob choose a secret key, Alice and Bob compute a noisy shared secret value and then use some reconciliation mechanism that allows them to agree on the same shared key (often also referred to as key agreement mechanism in the literature). This idea of a reconciliation mechanism to extract an exact shared value from noisy data is essentially the idea of a fuzzy extractor [52], known, for example, from physically unclonable functions. See, for example, [34, 80].

The reconciliation-based approach for Ring-LWE-based key agreement was listed as a special instance of "Noisy Diffie Hellman" by Gaborit (presenting joint work with Aguilar, Lacharme, Schrek, and Zémor) in his talk at PQCrypto 2010 [61, Slide 6]. It was also described by Lindner and Peikert as "(approximate) key agreement" [99, Sec. 3.1] and was already mentioned vaguely in an invited lecture of Peikert at TCC 2009 [119, Slide 14]. In [48] Ding described an instantiation of a reconciliation-based approach of LWE-based key exchange. The reconciliation mechanism can be used on top of a *matrix form of LWE* (as already used earlier, for example in [68] and [99, Sec. 2.2 and 3.1]) or on top of RLWE [102]. Neither of [61, 99, 119] describe a concrete reconciliation mechanism; the first reconciliation mechanism was the one described by Ding in [48, Sec 1.3] and [50]. Note that later and revised versions of [50] also list Lin [49] and Xie and Lin [51] as authors. Peikert in [120] tweaked Ding's reconciliation mechanism to obtain unbiased keys; the approach by Ding inevitably produces slightly biased key bits.

The main reason for the reconciliation-based approach is a reduced bandwidth requirement. For exam-

ple, [120] advertises "nearly halving the ciphertext size". This estimate comes from the fact that (in a naive version of the encryption-based approach) the second ciphertext polynomial has coefficients in $\{0, \ldots, q-1\}$ whereas the coefficients are in $\{0, 1\}$ (for [50, 49, 51] and [120]) or in $\{0, 1, 2, 3\}$ (for [9]) when using the reconciliation approach.

Our proposal: NewHope. Our submission, NewHope, is based on NewHope-Simple [8] which is a variant of NewHope-Usenix [9]. The main difference is that NewHope-Simple uses the encryption-based approach while NewHope-Usenix is based on the reconciliation-based approach. Alternatively, our submission could also be described as a variant of the scheme by Lyubashevsky, Peikert and Regev [103, 102] to which we apply the modifications from NewHope-Usenix [9] and the ciphertext size reduction technique from [123].

The basic NewHope-CPA-PKE scheme is a semantically secure public-key encryption with respect to adaptive chosen plaintext attacks. This allows us to apply standard transformations to build passively and actively secure KEMs and PKEs. This enables the use of our submission in unauthenticated key-exchange protocol but also in settings where a CCA-secure KEM or PKE is required. As a consequence, in this section we mainly focus on the properties of NewHope-CPA-PKE for which we define key generation in Algorithm 1 encryption in Algorithm 2 and decryption in Algorithm 3.

Parameter choices. We fix $q = 12289$ and $k = 8$ and provide two parameter sets that differ in only one parameter. For our NewHope512 with a bit-security level of 101 we set $n = 512$ and for NewHope1024 with a bit-security level of 233 we choose $n = 1024$. However, for long-term security we recommend NewHope1024. As k is fixed the same binomial sampler can be used for implementations of both parameter sets. Due to the fact that the security level grows with the noise-to-modulus ratio, it makes sense to choose the modulus as small as possible, improving compactness and efficiency together with security. As noise parameter k of the binomial distribution $\psi_k = \sum_{i=1}^{k} b_i - b'_i$ we set $k = 8$ for both parameter sets. This way we achieve a negligible error probability for both parameter sets. We chose the modulus $q = 12289$ as it is the smallest prime for which it holds that $q \equiv 1 \mod 2n$ so that the number-theoretic transform (NTT) can be realized efficiently and that we can transfer polynomials in NTT encoding (see Section 1.2.1). The choice is also appealing as the prime is already used by some implementations of Ring-LWE encryption [132, 47, 100] and BLISS signatures [54, 122]; thus sharing of some code (or hardware modules) between our proposal and an implementation of BLISS would be possible.

Error correction and reconciliation. The reconciliation technique of NewHope-Usenix [9] is generalizing and improving the previous approaches and extracts a single key bit from multiple polynomial coefficients. It relies on non-trivial *lattice-codes* and *lattice-quantizers* [43]. It is efficient, but fairly complex. Due to the complexity of the reconciliation approach in NewHope-Usenix [9] we propose the usage of the encryption-based approach. The difference in bandwidth requirements for NewHope-Usenix and NewHope is much smaller than one might expect. Specifically, the message from Bob to Alice (the ciphertext) in NewHope-Simple requires only 2176 bytes (compared to 2048 bytes in NewHope-Usenix); the message from Alice to Bob (the public key) has the same size (1824 bytes) for both variants. We obtain this result by carefully analyzing and optimizing a technique that is known since at least [118, Sec. 4.2] and has also been used in [123], for lattice-based signatures in [75], in the context of fully homomorphic encryption in [36, Sec. 4.2] and [33, Sec. 5.4] and for lattice-based PRFs in [18] and that was also applied in works like [88]. The idea of this technique is that the low bits of each coefficient of $\mathbf{v}$ mainly carry noise and contribute very little to the successful recovery of the plaintext. One can thus decide to "discard" (i.e, not transmit) those bits and thus shorten the length of $\mathbf{v}$. This can also be seen as switching to a smaller modulus and is therefore also called "modulus switching".

We combine this technique with a simple technique to encode one key bit into 4 coefficients that was first described by Güneysu and Pöppelmann in [123] and minimize the ciphertext size under the constraint that the failure probability of NewHope-Simple does not exceed the failure probability of NewHope.

Noise distribution. We do not use discrete Gaussians as the noise distribution but instead use the centred binomial distribution ψ_k of parameter $k = 8$ for the secret and error term. The reason is that it turns out to be challenging to implement a discrete Gaussian sampler efficiently *and* protected against timing attacks (see [31, 9]). On the other hand, sampling from the centered binomial distribution is easy and does not require high-precision computations or large tables as one may sample from ψ_k by computing $\sum_{i=0}^{k-1} b_i - b'_i$, where the

$b_i, b'_i \in \{0,1\}$ are uniform independent bits. The distribution ψ_k is centered (its mean is 0), has variance $k/2$ and for $k = 8$ this gives a standard deviation of $\varsigma = \sqrt{8/2}$. In Section 4.1.1 a justification of the security of this design decision is given. As explained in Section 4.2, our choice of parameters in NewHope1024 leaves a comfortable margin to the targeted 128 bits of post-quantum security (NIST level 5), which accommodates for the slight loss in security indicated by Theorem 4.1 due to the use of the binomial distribution. Even more important from a practical point of view is that no known attack makes use of the difference in error distribution; what matters for attacks are entropy and standard deviation.

No backdoor. One serious concern in lattice-based cryptography may be the presence of constant polynomials, e.g., the fixed system parameter $\mathbf{a}$ in [31]. As described in Section 1.2.1, our proposal includes pseudorandom generation of this parameter for every key exchange. In the following we discuss the reasons for this decision.

In the worst scenario, the fixed parameter $\mathbf{a}$ could be backdoored. For example, inspired by NTRU trapdoors [83, 137], a dishonest authority may choose mildly small $\mathbf{f}, \mathbf{g}$ such that $\mathbf{f} = \mathbf{g} = 1 \bmod p$ for some prime $p \geq 4 \cdot 8 + 1$ and set $\mathbf{a} = \mathbf{g}\mathbf{f}^{-1} \bmod q$. Then, given $(\mathbf{a}, \mathbf{b} = \mathbf{a}\mathbf{s} + \mathbf{e})$, the attacker can compute $\mathbf{b}\mathbf{f} = \mathbf{a}\mathbf{f}\mathbf{s} + \mathbf{f}\mathbf{e} = \mathbf{g}\mathbf{s} + \mathbf{f}\mathbf{e} \mod q$, and, because $\mathbf{g}, \mathbf{s}, \mathbf{f}, \mathbf{e}$ are small enough, compute $\mathbf{g}\mathbf{s} + \mathbf{f}\mathbf{e}$ in $\mathbb{Z}$. From this he can compute $\mathbf{t} = \mathbf{s} + \mathbf{e} \bmod p$ and, because the coefficients of $\mathbf{s}$ and $\mathbf{e}$ are smaller than 8, their sums are in $[-2 \cdot 8, 2 \cdot 8]$: knowing them modulo $p \geq 4 \cdot 8 + 1$ is knowing them in $\mathbb{Z}$. It now only remains to compute $(\mathbf{b} - \mathbf{t}) \cdot (\mathbf{a} - 1)^{-1} = (\mathbf{a}\mathbf{s} - \mathbf{s}) \cdot (\mathbf{a} - 1)^{-1} = \mathbf{s} \bmod q$ to recover the secret $\mathbf{s}$.

One countermeasure against such backdoors is the "nothing-up-my-sleeve" process, which would, for example, choose $\mathbf{a}$ as the output of a hash function on a common universal string like the digits of π. Yet, even this process may be partially abused [22], and when not strictly required it seems preferable to avoid it.

All-for-the-price-of-one attacks. Even if this common parameter has been honestly generated, it is still rather uncomfortable to have the security of all connections rely on a single instance of a lattice problem. The scenario is an entity that discovers an unforeseen cryptanalytic algorithm, making the required lattice reduction still very costly, but say, not impossible in a year of computation, given its outstanding computational power. By finding *once* a good enough basis of the lattice $\Lambda = \{(a,1)x + (q,0)y | x, y \in \mathcal{R}\}$, this entity could then compromise *all* communications, using for example Babai's decoding algorithm [15].

This idea of massive precomputation that is only dependent on a fixed parameter $\mathbf{a}$ and then afterwards can be used to break all key exchanges is similar in flavor to the 512-bit "Logjam" DLP attack [1]. This attack was only possible in the required time limit because most TLS implementations use fixed primes for Diffie-Hellman. One of the recommended mitigations by the authors of [1] is to avoid fixed primes.

Against all authority. Fortunately, all those pitfalls can be avoided by having the communicating parties generate a fresh $\mathbf{a}$ at each instance of the protocol (as we propose). If in practice it turns out to be too expensive to generate $\mathbf{a}$ for every connection, it is also possible to cache $\mathbf{a}$ on the server side[2] for, say a few hours without significantly weakening the protection against all-for-the-price-of-one attacks. Additionally, the performance impact of generating $\mathbf{a}$ is reduced by sampling $\mathbf{a}$ uniformly directly in NTT format (recalling that the NTT is a one-to-one map), and by transferring only a short 256-bit seed for $\mathbf{a}$.

A subtle question is to choose an appropriate primitive to generate a "random-looking" polynomial $\mathbf{a}$ out of a short seed. For a security reduction, it seems to the authors that there is no way around the (non-programmable) random oracle model (ROM). It is argued in [62] that such a requirement is in practice an overkill, and that any pseudorandom generator (PRG) should also work. And while it is an interesting question how such a reasonable pseudo-random generator would interact with our lattice assumption, the cryptographic notion of a PRG *is not* helpful to argue security. Indeed, it is an easy exercise[3] to build (under the NTRU assumption) a "backdoored" PRG that is, formally, a legitimate PRG, but that makes our scheme insecure. Instead, we prefer to base ourselves on a standard cryptographic hash-function, which is the typical choice of an "instantiation" of the ROM. As a suitable option we see Keccak [27], which has recently been standardized as SHA3 in FIPS-202 [112], and which offers extendable-output functions (XOF) named SHAKE. This avoids costly external iteration of a regular hash function and directly fits our needs. We use SHAKE128 for the generation of $\mathbf{a}$, which offers 128-bits of (post-quantum) security against collisions and preimage attacks. With only a small performance penalty we could have also chosen SHAKE256, but we do not see any

[2] But recall that the secrets $\mathbf{s}, \mathbf{e}, \mathbf{s}', \mathbf{s}', \mathbf{e}''$ have to be sampled fresh for every connection.

[3] Consider a secure PRG p, and parse its output $p(\text{seed})$ as two small polynomial $(\mathbf{f}, \mathbf{g})$: an NTRU secret-key. Define $p'(\text{seed}) = \mathbf{g}\mathbf{f}^{-1} \mod q$: under the decisional NTRU assumption, p' is still a secure PRG. Yet revealing the seed does reveal $(\mathbf{f}, \mathbf{g})$ and provides a backdoor as detailed above.

Table 1: Parameters of NewHope512 and NewHope1024 and derived high-level properties.

Parameter Set	NewHope512	NewHope1024
Dimension n	512	1024
Modulus q	12289	12289
Noise parameter k	8	8
NTT parameter γ	49	7
Decryption error probability	2^{-213}	2^{-216}
Claimed post-quantum bit-security	101	233
NIST Security Strength Category	1	5

Table 2: Sizes of public keys, secret keys, and ciphertexts of our NewHope instantiations in bytes.

Parameter Set	$\|pk\|$	$\|sk\|$	$\|ciphertext\|$
NewHope512-CPA-KEM	928	869	1088
NewHope1024-CPA-KEM	1824	1792	2176
NewHope512-CCA-KEM	928	1888	1120
NewHope1024-CCA-KEM	1824	3680	2208

reason for such a choice, in particular because neither collisions nor preimages lead to an attack against the proposed scheme.

Short-term public parameters. NewHope does not rely on a globally chosen public parameter **a** as the efficiency increase in doing so is not worth the measures that have to be taken to allow trusted generation of this value and the defense against backdoors [22]. Moreover, this approach avoids the rather uncomfortable situation that all connections rely on a single instance of a lattice problem (see Section 1.3) in the flavor of the "Logjam" DLP attack [1].

1.4 Parameters

1.4.1 NewHope512 and NewHope1024

For our NewHope cryptosystem we specify the two parameter sets NewHope512 and NewHope1024 in Table 1. These parameter sets are used to instantiate the NewHope-CPA-KEM or NewHope-CCA-KEM scheme. In case the security level should be specified together with the scheme we use the exemplary notation NewHope1024-CPA-KEM to refer to the NewHope-CPA-KEM scheme instantiated with the NewHope1024 parameter set. In Table 2 we provide public key, secret key, and ciphertext sizes for our two KEMs that support the transmission of a 256-bit message or key. For the justification of the NIST level we refer to Section 5 and for the justification of the post-quantum bit-security we refer to Section 4.2.

The parameters in Table 1 fully define NewHope and all other intermediary parameters can be calculated from there. For convenience, we list intermediary parameters:

- NewHope512: $\gamma = \sqrt{\omega} = 49$; $\omega = 2401$; $\omega^{-1} \bmod q = 11813$; $\gamma^{-1} \bmod q = 1254$; $n^{-1} \bmod q = 12265$
- NewHope1024: $\gamma = \sqrt{\omega} = 7$; $\omega = 49$; $\omega^{-1} \bmod q = 1254$; $\gamma^{-1} \bmod q = 8778$; $n^{-1} \bmod q = 12277$

Note that the parameters of NewHope cannot be freely chosen. The dimension n has to be an integer power of two to support efficient NTT algorithms and to maintain the security properties of RLWE. Degrees that are not power of 2 are also possible, but come with several complications [104, 121], in particular the defining polynomial of the ring can not have the form $X^n + 1$ anymore.

Additionally, n has to be greater or equal than 256 due to our choice of the encoding function that needs to embed a 256-bit message into an n-dimensional polynomial in NewHope-CPA-PKE. The modulus q has to be chosen as integer prime q such that $q \equiv 1 \mod 2n$ to support efficient NTT algorithms. The integer parameter k of the noise distribution has to be chosen such that the probability of decryption errors is negligible. On a high-level, the final security of NewHope depends on (q, n, k) where a larger n and a

larger $\frac{k}{q}$ lead to a higher security level. The choice of γ does not have an impact on the security but is need for correctness (see Section 1.2) and is simply the smallest possible value.

In the unlikely case that a higher security level is required while confidence in the RLWE assumption remains, it is straightforward to choose a NewHopeLudicrous parameter set with dimension $n = 2048$ and $k = 8$. This would basically double execution times and the size of public keys, ciphertexts, secrete keys (maybe). A small increase in security for the NewHope-CPA-KEM is also possible. As the scheme should in practice only be used in an ephemeral setting where decryption errors are less critical it might be possible to slightly increase k (e.g., $k = 16$ as in NewHope-Usenix).

We do not belief that a larger modulus q will result in a performance benefit or better performance/security tradeoff. However, in case q is increased, the parameter k has to be adapted as well. Choosing n not as a power of two would render the scheme insecure. In general, RLWE-based schemes do not requires a prime modulus q for security or performance. However, as NewHope directly uses properties of a negacyclic NTT, parameters have to be chosen so that q is prime and that $q \equiv 1 \mod 2n$. A scheme without restrictions regarding the modulus q would look quite different than NewHope from an implementers perspective.

1.4.2 Toy/challenge parameters

We do no encourage the use of smaller dimensions that $n = 512$ for practical applications. As toy parameter set for cryptanalysis we propose NewHopeToy1 with $n = 256, q = 7681, k = 4$. An even smaller toy parameter NewHopeToy2 set with $n = 128, q = 256, k = 1$ could also be a target for cryptanalysis but would require the reduction of the length of the message supported by NewHope-CPA-PKE to 128-bit.

1.4.3 Cryptographic primitives

NewHope relies on the SHAKE hash function [112] for several purposes:

- NewHope-CPA-PKE uses SHAKE128 to generate the public parameters $\hat{\mathbf{a}}$ from a public seed *seed*. In this instance the assumption is that SHAKE128 acts as a public random function for the given output length. Additionally, SHAKE256 is used to hash and extend the output of the random number generator in key generation.
- NewHope-CPA-KEM uses SHAKE256 to derive intermediate random values and the shared secret. The assumption is that SHAKE256 is a pseudorandom function.
- NewHope-CCA-KEM uses SHAKE256 to derive random keys, intermediate random values, and the shared secret; and to hash the public key and ciphertext. When hashing the public key pk and ciphertext c, the assumption is that SHAKE256 is collision-resistant. When deriving *seed*, *rand*, s, μ, and ss, the assumption is that SHAKE256 is a pseudorandom function.

No padding is used in the derivations above. Multiple inputs are combined by concatenating bitstrings; lengths of the concatenated values are fixed.

1.4.4 Provenance of constants and tables

The following constants are used in NewHope:

- Dimension n: Selected as a power of two to support efficient NTT algorithms and to maintain the security of RLWE.
- Modulus q: Selected as the smallest prime such that $q \equiv 1 \mod 2n$ so that the number-theoretic transform (NTT) can be realized efficiently.
- Noise parameter k: Selected so that the probability of decryption errors is negligible.
- NTT parameter γ: Must be n-th primitive root of unity; we select the smallest such value.
- Domain separation in calls to SHAKE: $\hat{\mathbf{a}}$ is generated pseudorandomly using SHAKE128 from a seed; domain separators are used internally in this generation, and are simply selected as counters.

2 Performance analysis

2.1 Estimated performance on the NIST PQC reference platform

In this section we provide details on our reference implementation written in C and estimate its performance on the NIST PQC reference platform. In Table 3 we list the directory in which the code of our reference implementation located in the submission file. The respective Gen (`crypto_kem_keypair`), Encaps (`crypto_kem_enc`), and Decaps (`crypto_kem_dec`) functions for each instantiation are defined in the file `kem.c` in the respective directory. Note that the code of reference and optimized implementation is identical but we provide two directory structures for completeness. The size of the produced keys and ciphertexts is not dependent on the platform and the numbers can be found in Table 2.

Table 3: Directories of the code of our reference implementation.

Reference Implementation	
NewHope512-CPA-KEM	Reference_Implementation/crypto_kem/newhope512cpa
NewHope512-CCA-KEM	Reference_Implementation/crypto_kem/newhope512cca
NewHope1024-CPA-KEM	Reference_Implementation/crypto_kem/newhope1024cpa
NewHope1024-CCA-KEM	Reference_Implementation/crypto_kem/newhope1024cca

The main emphasis in the C reference implementation is on simplicity and portability. It does not use any floating-point arithmetic and outside of the Keccak (SHAKE256 and SHAKE128) implementation only needs 16-bit and 32-bit integer arithmetic.

NTT Implementation. All polynomial coefficients are represented as unsigned 16-bit integers. Our in-place NTT implementation transforms from bit-reversed to natural order using Gentleman-Sande butterfly operations [67, 42]. One would usually expect that each NTT is preceded by a bit-reversal, but all inputs to NTT are noise polynomials that we can simply consider as being already bit-reversed. This is supported in the description of the algorithm. As explained earlier, the NTT^{-1} operation still involves a bit-reversal. For $n = 1024$ the core of the NTT and NTT^{-1} operation consists of 10 layers of transformations, each consisting of 512 butterfly operations of the form described in Listing 2.

Montgomery arithmetic and lazy reductions. The performance of operations on polynomials is largely determined by the performance of NTT and NTT^{-1}. The main computational bottleneck of those operations are 2304 ($n = 512$) or 5120 ($n = 1024$) butterfly operations, each consisting of one addition, one subtraction and one multiplication by a precomputed constant. Those operations are in $\mathbb{Z}_q$; recall that q is a 14-bit prime. To speed up the modular-arithmetic operations, we store all precomputed constants in Montgomery representation [111] with $R = 2^{18}$, i.e., instead of storing ω^i, we store $2^{18}\omega^i \pmod q$. After a multiplication of a coefficient g by some constant $2^{18}\omega^i$, we can then reduce the result r to $g\omega^i \pmod q$ with the fast Montgomery reduction approach. In fact, we do not always fully reduce modulo q, it is sufficient if the result of the reduction has at most 14 bits. The fast Montgomery reduction routine given in Listing 1a computes such a reduction to a 14-bit integer for any unsigned 32-bit integer in $\{0, \ldots, 2^{32} - q(R-1) - 1\}$. Note that the specific implementation does not work for *any* 32-bit integer; for example, for the input $2^{32} - q(R-1) = 1073491969$ the addition `a=a+u` causes an overflow and the function returns 0 instead of the correct result 4095. In the following we establish that this is not a problem for our software.

Aside from reductions after multiplication, we also need modular reductions after addition, which, for the sake of simplicity and readability, are written as the `C` modulo operator `%`). An alternative and faster approach is to use use "short Barrett reduction" [19] as detailed in Listing 1b. Again, this routine does not fully reduce modulo q, but reduces any 16-bit unsigned integer to an integer of at most 14 bits which is congruent modulo q.

In the context of the NTT and NTT^{-1}, we make sure that inputs have coefficients of at most 14 bits. This allows us to avoid reductions after addition on every second level, because coefficients grow by at most one bit per level and the short Barrett reduction (and the `%` operator) can handle 16-bit inputs. Let us turn our focus to the input of the Montgomery reduction (see Listing 2). Before subtracting `a[j+d]` from `t` we need to add a

multiple of q to avoid unsigned underflow. Coefficients never grow larger than 15 bits and $3 \cdot q = 36867 > 2^{15}$, so adding $3 \cdot q$ is sufficient. An upper bound on the expression `((uint32_t)t + 3*12289 - a[j+d])` is obtained if `t` is $2^{15} - 1$ and `a[j+d]` is zero; we thus obtain $2^{15} + 3 \cdot q = 69634$. All precomputed constants are in $\{0, \ldots, q-1\}$, so the expression `(W * ((uint32_t)t + 3*12289 - a[j+d])`, the input to the Montgomery reduction, is at most $69634 \cdot (q-1) = 855662592$ and thus safely below the maximum input that the Montgomery reduction can handle.

Listing 1 Reduction routines used in the reference implementation.

(a) Montgomery reduction ($R = 2^{18}$).

```
uint16_t mred(uint32_t a) {
  uint32_t u;
  u = (a * 12287);
  u &= ((1 << 18) - 1);
  a += u * 12289;
  return a >> 18;
}
```

(b) Short Barrett reduction.

```
uint16_t bred(uint16_t a) {
  uint32_t u;
  u = ((uint32_t) a * 5) >> 16;
  a -= u * 12289;
  return a;
}
```

Listing 2 The Gentleman-Sande butterfly inside odd levels of our NTT computation. All `a[j]` and `W` are of type `uint16_t`.

```
  W = omega[jTwiddle++];
  t = a[j];
  a[j] = bred(t + a[j+d]);
  a[j+d] = mred(W * ((uint32_t)t + 3*12289 - a[j+d]));
```

Fast random sampling. As a first step before performing any operations on polynomials, both Alice and Bob need to expand the seed to the polynomial **a** using SHAKE256. The implementation we use is based on the "simple" implementation by Van Keer for the Keccak permutation and slightly modified code taken from the "TweetFIPS202" implementation [26] for everything else.

Implementation of GenA. The public parameter **a** is generated from a 32-byte seed through the extendable-output function SHAKE128 [112, Sec. 6.2]. The approach described here slightly differs from the approach described in [9]. Specifically, the coefficients of **a** are generated in $n/64$ independent blocks of 64 coefficients each. To generate block i (of coefficients ranging from a_{64i} to a_{64i+63}) is generated by concatenating the $32 - byte$ seed with a one-byte value of i and feeding the resulting 33-byte extended seed to SHAKE128 is then considered as an array of 16-bit, unsigned, little-endian integers. Each of those integers is used as a coefficient of **a** if it is smaller than $5q$ and rejected otherwise. The first such 16-bit integer is used as the coefficient a_{64i}, the next one as coefficient of a_{64i+1} and so on. Each block needs a total of 64 coefficients, so at least 128 bytes of output from SHAKE128. The probability that a 16-bit value is smaller than $5q$ is 93.75%, so the expected number of bytes of SHAKE128 output per block of **a** is 137. One block of output of SHAKE128 has 168 bytes, so with very large probability only one block of SHAKE128 output is required for each block of **a**.

Performance results. Benchmark results for our reference implementation are reported in Table 4 and were obtained on an Intel Core i7-4770K (Haswell) running at 3491.953 MHz with Turbo Boost and Hyperthreading disabled. We compiled our C reference implementation with gcc-4.9.2 and flags `-O3 -fomit-frame-pointer -march=native`. For all other routines we report the median of 1000 runs.

2.2 Performance on x86 processors using vector extensions

Intel processors since the "Sandy Bridge" generation support Advanced Vector Extensions (AVX) that operate on vectors of 8 single-precision or 4 double-precision floating-point values in parallel. With the introduction of the "Haswell" generation of CPUs, this support was extended also to 256-bit vectors of integers of various sizes (AVX2). It is not surprising that the enormous computational power of these vector instructions has been used before to implement very high-speed crypto (see, for example, [23, 74, 24]) and also our optimized reference implementation targeting Intel Haswell processors uses those instructions to speed up multiple

Table 4: Cycle counts of our NewHope C reference implementation compiled with gcc-4.9.2 on an Intel Core i7-4770K (Haswell) with Turbo Boost and Hyperthreading disabled.

Operation	NH-512-CPA-KEM	NH-512-CCA-KEM	NH-1024-CPA-KEM	NH-1024-CCA-KEM
NTT	21,772	21,772	49,920	49,772
NTT^{-1}	23,384	23,420	53,596	53,408
GenA	16,012	16,052	32,248	32,240
Gen	106,820	117,128	222,922	244,944
Encaps	155,840	180,648	330,828	377,092
Decaps	40,988	206,244	87,080	437,056

Table 5: Cycle counts of an additional NewHope implementation using AVX extensions compiled with gcc-4.9.2 on an Intel Core i7-4770K (Haswell) with Turbo Boost and Hyperthreading disabled.

Operation	NH-512-CPA-KEM	NH-512-CCA-KEM	NH-1024-CPA-KEM	NH-1024-CCA-KEM
NTT	4888	4820	8416	8496
NTT^{-1}	6352	6344	11,708	11,680
GenA	10,804	10,808	21,308	21,480
Gen	56,236	68,080	107,032	129,670
Encaps	85,144	109,836	163,332	210,092
Decaps	19,472	114,176	35,716	220,864

components of the key exchange. We have done an implementation of NewHope targeting such a vectorized architecture.

NTT optimizations. The AVX instruction set has been used before to speed up the computation of lattice-based cryptography, and in particular the number-theoretic transform. Most notably, Güneysu, Oder, Pöppelmann and Schwabe achieve a performance of only 4 480 cycles for a dimension-512 NTT on Intel Sandy Bridge [76]. For arithmetic modulo a 23-bit prime, they represent coefficients as double-precision integers.

We experimented with multiple different approaches to speed up the NTT in AVX. For example, we vectorized the Montgomery arithmetic approach of our C reference implementation and also adapted it to a 32-bit-signed-integer approach. In the end it turned out that floating-point arithmetic beats all of those more sophisticated approaches, so we are now using an approach that is very similar to the approach in [76]. One computation of a dimension-1024 NTT takes $\approx 8\,450$ cycles, unlike the numbers in [76] this does include multiplication by the powers of γ and unlike the numbers in [76], this excludes a bit-reversal.

Fast sampling. For the computation of SHAKE-128 we use the same code as in the C reference implementation. One might expect that architecture-specific optimizations (for example, using AVX instructions) are able to offer significant speedups, but the benchmarks of the eBACS project [25] indicate that on Intel Haswell, the fastest implementation is the "simple" implementation by Van Keer that our C reference implementation is based on. The reasons that vector instructions are not very helpful for speeding up SHAKE (or, more generally, Keccak) are the inherently sequential nature and the 5×5 dimension of the state matrix that makes internal vectorization hard.

Performance results. Benchmark results for our AVX implementation are reported in Table 5 and were obtained on an Intel Core i7-4770K (Haswell) running at 3491.953 MHz with Turbo Boost and Hyperthreading disabled. `-no-pie -O3 -fomit-frame-pointer -msse2avx -mavx2 -march=native`

2.3 Performance estimation on ARM Cortex-M0 and M4

For the performance estimation of NewHope on ARM Cortex-M0 and M4 micrcontrollers we refer to the implementation of NewHope-Usenix by Alkim, Jakubeit, and Schwabe [10]. Their work shows that an implementation of NewHope-Usenix can outperform an implementation of Curve25519 [21] for the

Table 6: Cycle counts of a NewHope-Usenix implementation on a Cortex-M0 and Cortex-M4 microcontroller obtained from [10].

Operation	Cortex-M0	Cortex-M4
NTT	148,517	87,223
NTT^{-1}	167,405	97,789
Generation of **a**	380,855	293,975
Key generation (equiv. to Gen)	1,168,224	964,440
Key gen + shared key (equiv. to Encaps)	1,738,922	1,418,124
Shared key (equiv. to Decaps)	298,877	178,874
ROM usage (bytes)	30,178	22,828

Table 7: Cycle counts of an implementation of a CPA or CCA-secure public-key encryption scheme that is similar to NewHope (both use $n = 1024, q = 12289, k = 8$) on a Cortex-M0 and Cortex-M4 microcontroller obtained from [116].

Operation	Cycle Counts	
	Unmasked	Masked
Key generation	2,669,559	-
CCA2-secure encryption	4,176,684	-
CCA2-secure decryption	4,416,918	25,334,493
CPA-secure encryption	3,910,871	19,315,432
CPA-secure decryption	163,887	550,038
SHAKE128	87,738	201,997
NTT	83,906	-
NTT^{-1}	104,010	-
Uniform sampling (TRNG)	60,014	-
Noise sampling (PRNG)	1,142,448	6,031,463
PRNG (64 bytes)	88,778	202,454

Cortex-M0, like the one presented in [55], by more than a factor of two. Their cycle counts are given in Table 6.

Additionally, we also refer to the work by Oder, Schneider, Pöppelmann, and Güneysu [116] who describe an implementation of CCA2-secure public-key encryption that is similar to NewHope with and without side-channel countermeasures. In Table 7 we provide their results on a Cortex-M4. The instantiated scheme is ring-LWE public-key encryption ($n = 1024$, $q = 12289$, and binomial distribution with parameter $k = 8$) parametrized for negligible decryption errors so that the Fujisaki-Okamoto [59] transformation by Targhi and Unruh can be used [139]. The CCA2-secure encryption takes 4,176,684 cycles, which translates to 25 milliseconds when operating at a clock frequency of 168 MHz. Key generation takes 16 ms at 168 MHz. The application of the CCA2-conversion to the decryption causes a much higher overhead due to the necessary re-encryption. In the unmasked case, it requires 27 times more cycles.

2.4 Performance on MIPS64

Starting as an academic project in Stanford in the 1980s, the MIPS architecture is nowadays, typically utilized in network equipment, laser printers and consumer electronics. It was formerly part of superscalar processors (e.g. the MIPS I R2000 and R3000 of 32-bits, the R4000 of 64 bits (MIPS III) and the R10000 (MIPS IV)) and video game consoles (e.g. PlayStation 1–2 and Nintendo64). Developed by MIPS Technologies, Inc. (now Imagination Technologies[4]), the MIPS architecture is based on the RISC instruction set.

[4] https://www.imgtec.com/mips/

From year 2000, several synthesizable cores has appeared such as the 32-bit 4k, 24k and 64-bit 5k. Afterwards the MIPS32[5] and MIPS64[6] specifications were created. Nowadays, companies such as Loongson Technology, Cavium, Broadcom and Toshiba have licenses for MIPS64. Given the availability of the MIPS64 in the market in a myriad of different network routers, we have selected the MIPS64r3 release for optimizing NewHope1024 and NewHope512. Besides, the size of the available registers makes it ideal for vectorizing the polynomial arithmetic of the algorithm as well as for reducing memory access (this is the case, for instance, of our implementation of the Keccak-f[1600] permutation).

Our target device is a 28 nm cnMIPS III core from a CN7130 SoC, based on the MIPS64r3 architecture, clocked at 1.6 GHz and equipped with 78K instruction cache, 32K data cache and a floating point unit.

The MIPS64r3 architecture. The MIPS64r3 architecture has 32 64-bits registers and standardizes three co-processor encoding regions: CP0 for CPU configuration, cache control, interrupt control and memory management, CP1 is the FPU and CP2 available to other peripherals. The register file is comprised of 64-bit 32 registers where 27 can be used: v0-v1 (value returned by subroutines), a0-a3 (subroutine parameters), t0-t9 (temporary), s0-s8 (subroutine registers), gp (global pointer) and ra (return address). Further, co-processor CP1 provides an additional set of 32 64-bit registers, typically part of the FPU.

NTT optimization. As mentioned before, our choice for a 64-bits architecture instead of the MIPS32 one is largely based on the idea of vectorizing the NTT implementation. In so doing, we apply the following strategy: first, we parallelize the NTT by vectorizing the butterfly operation, second, and directly related to the former idea, we merge the layers of the NTT. Also both the Montgomery and Barrett reductions have been adapted for dealing with 64-bit integers.

Our approach is to parallelize the execution of the NTT by processing more than one coefficient (16 bits) in the architecture registers of 64-bit length. Out of the 32 registers of the MIPS64 architecture, only 27 are available to us for processing the 1,024 or 512 coefficients of the NTT. This way, we could execute one NTT's butterfly operation for n pairs of coefficients in parallel. However, due to the overflow bits of the butterfly operations (addition and multiplication) we can store 2 coefficients in 1 register. In our implementation, we use 16 registers for storing the coefficients, meaning that is actually possible to process $2 \cdot 16 = 32$ coefficients at a time, merging at most $log_2\ 32 = 5$ layers. Nonetheless due to the fact that after the 5^{th} layer the blocks of coefficients must be chosen from a different block of coefficients, that is, not from an adjacent one, we merge 4 layers.

Table 8: Performance figures of the NTT on MIPS64 in number of cycles.

Implementation	no optimization (#cycles)	-O2 (#cycles)	-O3 (#cycles)	opt (#cycles)
NTT1024 (NewHope1024 c32)	439,970	196,989	196,990	-
NTT512 (NewHope512 c32)	197,296	86,651	86,647	-
NTT1024 (vectorized)	-	-	-	85,348
NTT512 (vectorized)	-	-	-	38,755

Polynomial arithmetic vectorization. The same approach we utilized for optimizing the NTT can be applied to implement the polynomial arithmetic operations of NewHope1024 and NewHope512. However, in order to perform coefficients multiplication and pointwise multiplication via vectorization, both 128-bit registers and respective SIMD instructions are required in order to avoid overflows. Since our target architecture lacks both components we have only addressed the vectorization of the coefficient addition whose overflow can be controlled. Further, in order to reduce the impact of our strategy, we reused the coefficient storage method describe in the prior subsection. In this respect, we can perform two coefficient additions using a single addition instruction after an NTT has been performed.

Keccak. The XOF SHAKE used in NewHope relies on the Keccak-f[1600] permutation [27]. Since the main operations against internal state of $5x5$ are performed on 64-bit words, MIPS64 is ideal for doing the permutation directly on 64-bit registers. Besides, loading words into the state is also done using 64-bit words

[5] https://www.imgtec.com/mips/architectures/mips32/
[6] https://www.imgtec.com/mips/architectures/mips64/

and cycle reduction is achieved by (1) maintaining the 25 64-bit words of the state in registers exploiting the large amount of registers available in the MIPS64 architecture during the ρ and χ layers and (2) reducing memory access in the computation of the ρ layer by storing the input values directly from the θ layer.

However, when comparing our implementation with other architectures, the lack of a Bitwise Bit Clear `BIC` instruction in the MIPS64r3 ISA (available for instance in the ARMv7-M architecture[7]) creates a small performance penalty, since every operation of the χ layer requires three instructions (that is, one *and*, one *or* and one *xor* operation).

Performance results. In order to estimate the number of cycles required by each primitive of NewHope1024 and NewHope512 we rely on the performance counters of the coprocessor 0 (CP0). In so doing, we set up the performance counters in the before the start of of the primitive and measure again after the execution has finished.

Table 9: Performance figures of NewHope1024 and NewHope512 (CPA) on MIPS64 in number of cycles.

Implementation	no optimization (#cycles)	`-O2` (#cycles)	`-O3` (#cycles)	`opt` (#cycles)
NewHope1024	8,421,677	3,099,586	2,456,793	1,705,203
NewHope-CPA-KEM Key Generation	2,948,707	1,114,148	857,679	589,613
NewHope-CPA-KEM Encapsulation	4,378,938	1,645,474	1,277,601	882,443
NewHope-CPA-KEM Decapsulation	1,093,115	339,177	322,128	232,543
NewHope512	4,079,865	1,518,736	1,186,890	864,812
NewHope-CPA-KEM Key Generation	1,425,656	544,466	413,041	299,922
NewHope-CPA-KEM Encapsulation	2,123,245	806,941	617,037	448,791
NewHope-CPA-KEM Decapsulation	530,569	167,074	156,203	115,767

Table 10: Performance figures of NewHope1024 and NewHope512 (CCA) on MIPS64 in number of cycles.

Implementation	no optimization (#cycles)	`-O2` (#cycles)	`-O3` (#cycles)	`opt` (#cycles)
NewHope1024	14,466,351	5,524,430	4,290,417	2,871,081
NewHope-CCA-KEM Key Generation	3,329,101	1,298,382	981,655	651,810
NewHope-CCA-KEM Encapsulation	5,078,734	1,481,606	1,517,853	1,021,702
NewHope-CCA-KEM Decapsulation	6,058,512	2,244,514	1,791,563	1,197,556
NewHope512	7,029,073	2,702,808	2,063,742	1,473,698
NewHope-CCA-KEM Key Generation	1,611,787	633,120	470,209	334,100
NewHope-CCA-KEM Encapsulation	2,485,018	979,139	734,888	530,373
NewHope-CCA-KEM Decapsulation	2,931,568	1,089,937	858,111	609,107

Our performance figures suggest that it is possible to achieve a reduction of 196,990 - 85,348 = 111,642 cycles in the NTT computation from NewHope1024 (that is, a speed up of factor 2.3) and a reduction of 86,647 - 38,755 = 47,892 cycles in the NTT computation from NewHope1024 (that is, a speed up of factor 1.8) (See Table 8). With `opt` we refer to the implementations based on the optimization techniques described in this section. Besides, we noticed an overall reduction of factor 1.49 and factor 1.40 in the computation of the whole protocol (NewHope1024 and NewHope512 respectively (CCA-KEM)). These improvements were obtained by comparing our results with a compilation of NewHope1024/ NewHope512 using an aggressive optimization option (`-O3`)[8] (Table 10).

[7] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html`

[8] Using `gcc` version 4.7.0 for cross-compiling on Linux `4.12.0-2-amd64`

3 Known Answer Test values

All KAT values are included in subdirectories of the directory `KAT` of the submission package. Specifically, the KAT values of NewHope512-CPA-KEM are in the subdirectory `KAT/newhope512cpa`, the KAT values of NewHope512-CCA-KEM are in the subdirectory `KAT/newhope512cca`, the KAT values of NewHope1024-CPA-KEM are in the subdirectory `KAT/newhope1024cpa`, and the KAT values of NewHope1024-CCA-KEM are in the subdirectory `KAT/newhope1024cca`. Each of those directories contains the KAT values as generated by the `PQCgenKAT_kem` program provided by NIST. Specifically, those files are:

- `KAT/newhope512cpa/PQCkemKAT_896.req`
- `KAT/newhope512cpa/PQCkemKAT_896.rsp`
- `KAT/newhope512cca/PQCkemKAT_1888.req`
- `KAT/newhope512cca/PQCkemKAT_1888.rsp`
- `KAT/newhope1024cpa/PQCkemKAT_1792.req`
- `KAT/newhope1024cpa/PQCkemKAT_1792.rsp`
- `KAT/newhope1024cca/PQCkemKAT_3680.req`
- `KAT/newhope1024cca/PQCkemKAT_3680.rsp`

4 Justification of security strength

4.1 Provable security reductions

A summary of the provable security reductions underlying the security of NewHope-CCA-KEM is as follows:

1. Using the centred binomial distribution ψ_k instead of a discrete Gaussian distribution provides negligible advantage to an adversary. Section 4.1.1 gives a justification.
2. Using a pseudorandomly generated $\hat{\mathbf{a}}$ in NewHope-CPA-PKE instead of a uniformly random $\hat{\mathbf{a}}$ provides no advantage to an adversary, under the assumption that SHAKE128 is a random oracle.
3. NewHope-CCA-KEM is an IND-CCA-secure KEM under the assumption that NewHope-CPA-PKE is an IND-CPA-secure public key encryption scheme and that G and F (both instantiated as SHAKE256 are random oracles. Theorem 4.2 gives a tight, classical reduction against classical adversaries in the classical random oracle model. Theorem 4.3 gives a non-tight, classical reduction against quantum adversaries in the quantum random oracle model.
4. NewHope-CPA-PKE is an IND-CPA-secure public key encryption scheme under the assumption that the decision ring learning with errors problem is hard. Theorem 4.4 gives a tight, classical reduction against classical or quantum adversaries in the standard model.
5. The decision ring learning with errors problem is hard under the assumption that the search version of the approximate shortest vector problem is hard (in the worst case) on ideal lattices in $\mathcal{R}$, for appropriate parameters. Lyubashevsky et al. [102, Thm. 3.6] give a polynomial-time quantum reduction against classical or quantum adversaries in the standard model. See also [120, Thm. 2.7] for a simplified version of this result.

4.1.1 Binomial noise distribution

The original worst-case to average-case reductions for LWE [127] and Ring-LWE [103] state hardness for *continuous Gaussian* distributions (and therefore also trivially apply to *rounded Gaussians*, which differ from discrete Gaussians). This also extends to discrete Gaussians [35] but such proofs are not necessarily intended for direct implementations. The use of discrete Gaussians (or other distributions with very high-precision sampling) is only crucial for signatures [101] and lattice trapdoors [70], to provide zero-knowledgeness.

The following theorem states that choosing ψ_k as error distribution in NewHope-CPA-KEM (i.e., using the algorithm Sample) does not significantly decrease security compared to a rounded Gaussian distribution with the same standard deviation $\sigma = \sqrt{8/2}$.

Theorem 4.1 *Let ξ be the rounded Gaussian distribution of parameter $\sigma = \sqrt{4}$, that is, the distribution of $\lfloor \sqrt{4} \cdot x \rceil$ where x follows the standard normal distribution. Let $\mathcal{P}$ be the idealized version of* NewHope-CPA-KEM*, where outputs from* Sample *are replaced by samples from ξ. If an (unbounded) algorithm, given as input the public key and ciphertext of* NewHope-CPA-KEM *succeeds in recovering the shared secret ss with probability p, then it would also succeed against $\mathcal{P}$ with probability at least*

$$q \geq p^{9/8} \cdot 2^{-14}.$$

The result also holds for NewHope-CCA-KEM*.*

In [17], Bai et al. identify Rényi divergence as a powerful tool to improve or generalize security reductions in lattice-based cryptography. We review the key properties. The Rényi divergence [129, 17] is parametrized by a real $a > 1$, and defined for two distributions P, Q by:

$$R_a(P\|Q) = \left(\sum_{x \in \mathrm{Supp}(P)} \frac{P(x)^a}{Q(x)^{a-1}} \right)^{\frac{1}{a-1}}.$$

It is multiplicative: if P, P' are independent, and Q, Q' are also independent, then $R_a(P \times P'\|Q \times Q') \leq R_a(P\|Q) \cdot R_a(P'\|Q')$. Finally, Rényi divergence relates the probabilities of the same event E under two different distributions P and Q:

$$Q(E) \geq P(E)^{a/(a-1)} / R_a(P||Q).$$

Proof For our argument, recall that because the final shared key ss is obtained through hashing as $ss \leftarrow \mathsf{SHAKE256}(K)$ before being used, then, in the random oracle model (ROM), any successful attacker must recover K exactly. We call this event E. We also define ξ to be the rounded Gaussian distribution of parameter $\sigma = \sqrt{k/2} = \sqrt{4}$, that is the distribution of $\lfloor \sqrt{4} \cdot x \rceil$ where x follows the standard normal distribution.

A simple script in [9] computes $R_9(\psi_8\|\xi) \approx 1.002$. Yet because $5n$ samples are used per instance of the protocol, we need to consider the divergence $R_9(P\|Q) = R_9(\psi_8, \xi)^{5n}$ where $P = \psi_8^{5n}$ and $Q = \xi^{5n}$. For $n = 512$ we get $R_9(P\|Q) \approx 164$, and for $n = 1024$, we get $R_9(P\|Q) \approx 26889$. In both cases, $R_9(P\|Q) \leq 2^{14}$. ∎

The choice $a = 9$ is rather arbitrary but seemed a good trade-off between the coefficient $1/R_a(\psi_8\|\xi)$ and the exponent $a/(a-1)$. This reduction is provided as a safeguard: switching from Gaussian to binomial distributions can not dramatically decrease the security of the scheme. With practicality in mind, we will simply ignore the loss factor induced by the above reduction, since the best-known attacks against LWE do not exploit the structure of the error distribution, and seem to depend only on the standard deviation of the error (except in extreme cases [14, 90]).

4.1.2 Security of IND-CCA KEM

Theorem 4.2 (IND-CPA PKE $\implies$ IND-CCA KEM in classical ROM) *We define a public key encryption scheme* $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ *with message space $\mathcal{M}$ and which is δ-correct. Let G and F be independent random oracles. Let* $\mathsf{QKEM}_m^{\not\perp'} = \mathsf{QFO}_m^{\not\perp'}[\mathsf{PKE}, G, F]$ *be the KEM obtained by applying the* $\mathsf{QFO}_m^{\not\perp'}$ *transform as in subsubsection 1.2.3. For any classical algorithm $\mathcal{A}$ against the IND-CCA security of* $\mathsf{QKEM}_m^{\not\perp'}$ *that makes q_G and q_F queries to its G and F oracles, there exists a classical algorithm $\mathcal{B}$ against the IND-CPA security of* PKE *such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{QKEM}_m^{\not\perp'}}(\mathcal{A}) \leq \frac{4 \cdot q_{\mathsf{RO}} + 1}{|\mathcal{M}|} + q_{\mathsf{RO}} \cdot \delta + 3 \cdot \mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{B})$$

where $q_{\mathsf{RO}} = q_G + q_F$. Moreover, the running time of $\mathcal{B}$ is about that of $\mathcal{A}$.

Theorem 4.2 follows from Theorems 3.2 and 3.4 of Hofheinz, Hövelmanns, and Kiltz [85], with the following modifications. In the application of HHK's Theorem 3.2, we take $q_V = 0$. Note that Theorems 3.2 and 3.4 of HHK are about the $\mathsf{FO}^{\not\perp}$ transform, which differs from the $\mathsf{QFO}_m^{\not\perp'}$ in the following ways. 1) $\mathsf{QFO}_m^{\not\perp'}$

uses a single hash function (with longer output) to compute K and $coin'$ whereas $\mathsf{FO}^{\not\perp}$ uses two; but this is equivalent in the random oracle model with appropriate output lengths. 2) $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of K and $coin'$ also takes the public key pk as input whereas $\mathsf{FO}^{\not\perp}$ does not; this does not negatively affect any of the theorems, and has the potential to provide multi-target security. 3) $\mathsf{QFO}_m^{\not\perp\prime}$ includes the d value in the ciphertext, whereas $\mathsf{FO}^{\not\perp}$ does not; since d is computed by applying a random oracle G to the secret $\mu \in \mathcal{M}$, taking advantage of d requires querying G on μ, which occurs with the additional $\frac{q}{|\mathcal{M}|}$ probability term added in the theorem.

Theorem 4.3 (IND-CPA PKE $\implies$ IND-CCA KEM in quantum ROM) *We define a public key encryption scheme* $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ *with message space* $\mathcal{M}$ *and which is* δ*-correct. Let* G *and* F *be independent random oracles. Let* $\mathsf{QKEM}_m^{\not\perp\prime} = \mathsf{QFO}_m^{\not\perp\prime}[\mathsf{PKE}, G, F]$ *be the KEM obtained by applying the* $\mathsf{QFO}_m^{\not\perp\prime}$ *transform as in subsubsection 1.2.3. For any quantum algorithm* $\mathcal{A}$ *against the IND-CCA security of* $\mathsf{QKEM}_m^{\not\perp\prime}$ *that makes* q_G *and* q_F *queries to its quantum* G *and* F *oracles, there exists a quantum algorithm* $\mathcal{B}$ *against the IND-CPA security of* PKE *such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{QKEM}_m^{\not\perp\prime}}(\mathcal{A}) \leq 9 \cdot q_{\mathsf{RO}} \cdot \sqrt{q_{\mathsf{RO}}^2 \cdot \delta + q_{\mathsf{RO}} \cdot \sqrt{\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{B}) + \frac{1}{|\mathcal{M}|}}}$$

where $q_{\mathsf{RO}} = q_G + q_F$*. Moreover, the running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$*.*

Theorem 4.3 follows from Lemma 2.3 and Theorems 4.4 and 4.6 of Hofheinz, Hövelmanns, and Kiltz [85], with the following modifications. Note that Theorems 4.4 and 4.6 of HHK are about the $\mathsf{QFO}_m^{\not\perp}$ transform, which differs from the $\mathsf{QFO}_m^{\not\perp\prime}$ in the following ways. 1) $\mathsf{QFO}_m^{\not\perp\prime}$ uses a single hash function (with longer output) to compute K, $coin'$, and d whereas $\mathsf{FO}^{\not\perp}$ uses two; but this is equivalent in the random oracle model with appropriate output lengths. 2) $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of K, $coin'$, and d also takes the public key pk as input whereas $\mathsf{FO}^{\not\perp}$ does not; this does not negatively affect any of the theorems, and has the potential to provide multi-target security. 3) $\mathsf{QFO}_m^{\not\perp\prime}$'s computation of the shared secret ss also takes the encapsulation $\overline{c}$ as input; this does not negatively affect any of the theorems, and provides robustness against ciphertext modification.

4.1.3 Security of IND-CPA PKE

Theorem 4.4 (dRLWE $\implies$ IND-CPA security of NewHope-CPA-PKE) *Let* n *and* q *be integers. Let* χ *be a probability distribution on* $\mathcal{R}_q$*. For any quantum algorithm* $\mathcal{A}$ *against the IND-CPA security of* NewHope-CPA-PKE *(with uniformly random* $\hat{\mathbf{a}}$*), there exists quantum algorithms* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *against the decision ring-LWE problem such that*

$$\mathrm{Adv}^{\mathsf{ind\text{-}cpa}}_{\text{NewHope-CPA-PKE}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{dRLWE}}_{n,q,\chi}(\mathcal{B}_1) + \mathrm{Adv}^{\text{dRLWE}}_{n,q,\chi}(\mathcal{B}_2) \ .$$

Moreover, the running times of $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *are about that of* $\mathcal{A}$*.*

The proof of Theorem 4.4 is essentially the same as that of Lemma 4.1 of [120] or Theorem 1 of [31].

4.2 Cryptanalytic attacks

For our security analysis in this section we mainly rely on the (very pessimistic) concrete security analysis of Ring-LWE based cryptosystems from [9]. Additionally, we also estimate the security level with the approach presented in [7].

4.2.1 Methodology: the core SVP hardness

RLWE as LWE. We analyze the hardness of Ring-LWE as an LWE problem, since, so far, the best known attacks do not make use of the ring structure. Indeed, while some new quantum algorithms against Ideal-SVP recently appeared [56, 39, 28, 44, 45], they do not seem to affect Ring-LWE. Precisely, in [45] two obstacles are discussed. First the approximation factor reached are asymptotically sub-exponential and it is therefore unlikely to affect cryptographic parameters. Secondly, Ring-LWE is proven to be at least as hard as Ideal-SVP,

but the natural approach for a converse reduction seems to require the ring $\mathbb{Z}[X]/(X^n+1)$ to be Euclidean, which is only the case for $n \in \{1, 2, 4\}$ (see [97]).

Attacks against LWE. There are many algorithms to consider in general (see the survey [7]), yet many of those are irrelevant for our parameter set. In particular, because there are only $m = n$ samples available one may rule out BKW types of attacks [90] and linearization attacks [14]. This essentially leaves us with two BKZ [136, 41] attacks, usually referred to as primal and dual attacks that we will briefly recall below.

The algorithm BKZ proceeds by reducing a lattice basis using an SVP oracle in a smaller dimension b. It is known [78] that the number of calls to that oracle remains polynomial, yet concretely evaluating the number of calls is rather painful, and this is subject to new heuristic ideas [41, 40, 12]. We choose to ignore this polynomial factor, and rather evaluate only the *core SVP hardness*, that is the cost of *one call* to an SVP oracle in dimension b, which is clearly a pessimistic estimation from the defender's point of view.

4.2.2 Enumeration versus quantum sieve

Typical implementations of BKZ [65, 41, 38] use an enumeration algorithm as its SVP oracle, yet this algorithm runs in super-exponential time $2^{\Theta(n \log n)}$. On the other hand, the sieve algorithms are known to run in exponential time, but are so far slower in practice for accessible dimensions $b \approx 130$. In recent work Ducas [53] has shown that sieving techniques (in the classical setting) can be used in practice for exact-SVP, being now less than an order of magnitude slower than enumeration already in dimension 60 to 80.

For simplicity and conservatism, we will choose a reasonable lower bound for both enumeration and sieving. Namely, our bounds follow the asymptotic complexity of sieving algorithms, yet ignoring sub-exponential factors, when calculating cost in those attacks. According to the prediction of [41], even with Grover acceleration, the cost of enumeration is also lower-bounded by our estimates for blocksizes $b \geq 250$.

Quantum sieve. A lot of recent work has pushed the efficiency of the original lattice sieve algorithms [114, 109], improving the heuristic complexity from $(4/3)^{b+o(b)} \approx 2^{0.415b}$ down to $\sqrt{3/2}^{b+o(b)} \approx 2^{0.292b}$ using *Locality Sensitive Hashing* (LSH) techniques [93, 20]. The hidden sub-exponential factor is known to be much greater than one in practice, so again, estimating the cost ignoring this factor leaves us with a significant pessimistic margin.

Most of those algorithms have been shown [94, 92] to benefit from Grover's quantum search algorithm, bringing the complexity down to $2^{0.265b}$. It is unclear if further improvements are to be expected, yet, because all those algorithms require classically building lists of size $\sqrt{4/3}^{b+o(b)} \approx 2^{0.2075b}$. It is thus very plausible that the best quantum SVP algorithm would run in time greater than $2^{0.2075b}$.

Discarding enumeration for our analysis. In [41], predictions of the cost of solving SVP classically using sophisticated heuristic enumeration algorithms are given. For example, solving SVP in dimension 100 requires visiting about 2^{39} nodes, and 2^{134} nodes in dimension 250. Note that the cost for enumeration are here given in term of visited nodes in the enumeration tree, and visiting each of those nodes require about 100 cycles according to [41]. For simplicity and conservatism, we will assume that each node requires only one cycle.

Because this enumeration is a backtracking algorithm, it does benefit from the recent quasi-quadratic speedup [110], decreasing the quantum cost to about at least 2^{20} to 2^{67} operations as the dimension increases from 100 to 250. Again, this is quite conservative as the quantum version of this backtracking algorithm is subject to slowdowns polynomial in the depth of the tree.

On the other hand, our best-known attack bound $2^{0.265b}$ gives a cost of 2^{66} in dimension 250, and the best plausible attack bound $2^{0.2075b} \approx 2^{39}$. Because enumeration is super-exponential (both in theory and practice), its cost will be worse than our bounds in dimension larger than 250 and we may safely ignore this algorithm.[9]

We note that a recent technique formalized as discrete pruning [60, 11] seems to outperform the previous pruned enumeration of [41]. Unfortunately, no tools are currently available to fully predict the cost of this new techniques. We hope that future work will clarify these issues. Our current understanding is that this methods visits more nodes of the enumeration tree, but visit them much faster by removing the intricate backtracking steps. By counting the number of visited nodes rather than the count of CPU cycles, our lower bound should therefore also apply to this new discrete pruning technique.

[9] The numbers are taken from the latest full version of [41] available at http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf.

4.2.3 Primal attack

The primal attack consists of constructing a unique-SVP instance from the LWE problem and solving it using BKZ. We examine how large the block dimension b is required to be for BKZ to find the unique solution. Given the matrix LWE instance $(\mathbf{A}, \mathbf{b} = \mathbf{As}+\mathbf{e})$ one builds the lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^{m+n+1} : (\mathbf{A}|-\mathbf{I}_m|-\mathbf{b})\mathbf{x} = \mathbf{0} \bmod q\}$ of dimension $d = m + n + 1$, volume q^m, and with a unique-SVP solution $\mathbf{v} = (\mathbf{s}, \mathbf{e}, 1)$ of norm $\lambda \approx \varsigma\sqrt{n+m}$. Note that the number of used samples m may be chosen between 0 and $2n$ in our case and we numerically optimize this choice.

Success condition. We model the behavior of BKZ using the geometric series assumption (which is known to be optimistic from the attacker's point of view), that finds a basis whose Gram-Schmidt norms are given by $\|\mathbf{b}_i^\star\| = \delta^{d-2i-1} \cdot \mathrm{Vol}(\Lambda)^{1/d}$ where $\delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)}$ [40, 7]. The unique short vector $\mathbf{v}$ will be detected if the projection of $\mathbf{v}$ onto the vector space spanned by the last b Gram-Schmidt vectors is shorter than $\mathbf{b}_{d-b}^\star$. Its projected norm is expected to be $\varsigma\sqrt{b}$, that is the attack is successful if and only if

$$\varsigma\sqrt{b} \leq \delta^{2b-d-1} \cdot q^{m/d}. \tag{1}$$

We note that this analysis introduced in [9] differs and is more conservative than prior works, which were typically based on the hardness of unique-SVP estimates of [63]. The validity of the new analysis has been confirmed by further analysis and experiments in [6].

4.2.4 Dual attack

The dual attack consists of finding a short vector in the dual lattice $\mathbf{w} \in \Lambda' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \mathbf{A}^t\mathbf{x} = \mathbf{y} \bmod q\}$. Assume we have found a vector $(\mathbf{x}, \mathbf{y})$ of length ℓ and compute $z = \mathbf{v}^t \cdot \mathbf{b} = \mathbf{v}^t\mathbf{As} + \mathbf{v}^t\mathbf{e} = \mathbf{w}^t\mathbf{s} + \mathbf{v}^t\mathbf{e} \bmod q$ which is distributed as a Gaussian of standard deviation $\ell\varsigma$ if $(\mathbf{A}, \mathbf{b})$ is indeed an LWE sample (otherwise it is uniform mod q). Those two distributions have maximal variation distance bounded by $\epsilon = 4\exp(-2\pi^2\tau^2)$ where $\tau = \ell\varsigma/q$, that is, given such a vector of length ℓ one has an advantage ϵ against decision-LWE.

The length ℓ of a vector given by the BKZ algorithm is given by $\ell = \|\mathbf{b}_0\|$. Knowing that Λ' has dimension $d = m + n$ and volume q^n we get $\ell = \delta^{d-1}q^{n/d}$. Therefore, obtaining an ϵ-distinguisher requires running BKZ with block dimension b where

$$-2\pi^2\tau^2 \geq \ln(\epsilon/4). \tag{2}$$

Note that small advantages ϵ are not relevant since the agreed key is hashed: an attacker needs an advantage of at least $1/2$ to significantly decrease the search space of the agreed key. He must therefore amplify his success probability by building about $1/\epsilon^2$ many such short vectors. Because the sieve algorithms provide $2^{0.2075b}$ vectors, the attack must be repeated at least R times where

$$R = \max(1, 1/(2^{0.2075b}\epsilon^2)).$$

This makes the conservative assumption that all the vectors provided by the Sieve algorithm are as short as the shortest one.

4.2.5 Security analysis

The cost of the primal attack and dual attacks are given in Table 11. They were obtained by executing our script in `scripts/PQsecurity.py`. According to our analysis, we claim that our proposed parameters for NewHope1024 offer 233 bits of security. Thus we are stronger (and quite likely with a large margin) than a post-quantum security level of 128 bits. In particular, NewHope1024 could even withstand a dimension-halving attack in the line of [66, Sec 8.8.1] based on the Gentry-Szydlo algorithm [71, 98] or the subfield approach of [5]. Note that so far, such attacks are only known for principal ideal lattices or NTRU lattices, and there are serious obstructions to extend them to Ring-LWE, but such precaution seems reasonable until lattice cryptanalysis stabilizes. For our NewHope512 parameter set we claim 101 bits of security.

In addition to our own analysis, we have used a freely available tool to evaluate the concrete security of LWE instances [7]. This approach is less pessimistic than our original security analysis, in particular it takes

Attack	m	b	Known Classical	Known Quantum	Best Plausible
BCNS proposal [31]: $q = 2^{32} - 1$, $n = 1024$, $\varsigma = 3.192$					
Primal	1062	296	86	78	61
Dual	1055	296	86	78	61
NTRUEncrypt [82]: $q = 2^{12}$, $n = 743$, $\varsigma \approx \sqrt{2/3}$					
Primal	613	603	176	159	125
Dual	635	600	175	159	124
JarJar-Usenix [9]: $q = 12289$, $n = 512$, $\varsigma = \sqrt{12}$					
Primal	623	449	131	119	93
Dual	602	448	131	118	92
NewHope-Usenix [9]: $q = 12289$, $n = 1024$, $\varsigma = \sqrt{8}$					
Primal	1100	967	282	256	200
Dual	1099	962	281	255	199
NewHope512: $q = 12289$, $n = 1024$, $\varsigma = \sqrt{4}$					
Primal	540	384	112	101	79
Dual	545	383	112	101	79
NewHope1024: $q = 12289$, $n = 1024$, $\varsigma = \sqrt{4}$					
Primal	999	886	259	235	183
Dual	1048	881	257	233	182

Table 11: Core hardness of NewHope512 and NewHope1024 and selected other proposals from the literature as well as previous instantiations of NewHope. The value b denotes the block dimension of BKZ, and m the number of used samples. Cost is given in $\log_2$ of CPU operations and is the smallest cost for all possible choices of m and b. Note that our estimation is very optimistic about the abilities of the attacker so that our result for the parameter set from [31] *does not* indicate that it can be broken with $\approx 2^{80}$ bit operations, given today's state-of-the-art in cryptanalysis.

account for the number of SVP calls, and estimate the cost of classical sieving to $2^{.292b+16}$. In Table 12 we provide the results for NewHope512 and NewHope1024. These values have been obtained by executing for different values of n and k using the sage module as follows:

```
load("https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py")
k = 8.0; n = 1024; q = 12289; stddev = sqrt(k/2); alpha = alphaf(sigmaf(stddev), q)
_ = estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.sieve)
```

The estimation in Table 12 also leads to the conclusion that NewHope1024, with a security level of 289 bits, reaches well beyond a security level of 128 bits. For NewHope512 a bit-security level of 142 bits is obtained. Most other (R)LWE-based or NTRU-based proposals achieve considerably lower security than NewHope1024. For comparison we also give a lower bound on the security of [31] and do notice a significantly improved security in our proposal. Yet, because of the numerous pessimistic assumption made in our analysis, we do not claim any quantum attacks reaching those bounds. The highest-security parameter set used for RLWE encryption in [72] is very similar to the parameters of JarJar-Usenix. The situation is different for NTRUEncrypt, which has been instantiated with parameters that achieve about 128 bits of security according to our analysis[10]. Specifically, we refer to NTRUEncrypt with $n = 743$ as suggested in [82]. A possible advantage of NTRUEncrypt compared to NewHope is somewhat smaller message sizes, however, this advantage becomes very small when scaling parameters to achieve a similar security margin as NewHope.

[10]For comparison we view the NTRU key-recovery as an homogeneous Ring-LWE instance. We do not take into account the combinatorial vulnerabilities [86] induced by the fact that secrets are ternary. We note that NTRU is a potentially a weaker problem than Ring-LWE: it is in principle subject to a subfield-lattice attack [5], but the parameters proposed for NTRUEncrypt are immune.

	usvp	dec	dual
JarJar-Usenix	161	198	185
NewHope-Usenix	313	410	356
NewHope512	142	171	163
NewHope1024	289	373	334

Table 12: Hardness of NewHope512 and NewHope1024 in the model of [7]. The analysis is based on a cost of $2^{.292b+16.4}$ for each call to the (classical) sieve of [20], an estimate that lies between our lower bound of $2^{.292b}$, and the measured cost in practice [20, 105]. This models also account for the number of calls to the sieve inside BKZ. The 'usvp' attack is similar what we call the 'primal' attack, while the 'dec' attack is a weaker variation trying to solve BDD without embedding it to a unique-SVP problem, by first reducing the primal lattice and then decoding the target vector using Babai decoding. Please refer to [7] for details. The dual attack they consider is also similar to the one described above.

4.2.6 Cost model and margins

Considering that lattice cryptanalysis is not a fully matured research area and that substantial improvement are still appearing, it seems preferable to leave significant margins in our security claims. The state-of-the art is unfortunately not as refined for lattice algorithm as it is can be for a memory-less brute-force attack on AES [73]. An analysis based on the current state of the art, using a model as refined as suggested by the call for proposal seems, in our case, way too intricate, prone to mistakes, and would likely become irrelevant within a few year.

We prefer to perform our analysis in simpler model; yet all the simplifications are done in favor of the attacker, and serve as margins. Those simplifications also contribute to make our analysis and scripts easier to verify. We list them here.

Asymptotic versus concrete. We used theoretical complexity of Sieving omitting sub-exponential factors. For the best asymptotic sieve algorithm [20] with complexity $2^{.292b+o(b)}$, it is typically reported that the fitted practical complexity $f \cdot 2^{cb}$ is quite larger than $2^{.292b}$, including a large constant factor f and a constant c. For example, the initial implementation [20] reports a fit of about $2^{.387b+16}$ clock-cycles on a `x86-64` CPU, in the range of dimensions $60 - 80$.

We prefer not to conclude on a quantified margin considering further works. Indeed, the implementation of [105] reports significant speed-ups using fine tuning and low-level optimizations (up to $\times 50$), but unfortunately no fit is provided. Further improvements are expected by combining those techniques with the very recent SubSieve algorithm of [53]: quantified claims seems premature.

Core SVP hardness versus BKZ. We also ignored the fact that the attacks actually require polynomially many calls to BKZ, and the best concrete predictions are typically based on simulations [41, 7]. Those simulation gets more complicated to perform as we include more techniques such as [12]. One may doubt the reliability of such simulations and fear further improvement of BKZ strategies. Moreover, some amortization strategies of sieving in BKZ sketched in [53] remains to be studied. Our core SVP-hardness approach dismisses those concerns.

CPU cycles versus gates. We also note that the concrete cost measured above is expressed in numbers of `x86-64` CPU cycles, rather than gate count. If one wishes to evaluate the gate count, we warn against naive implementations whose main cost are derived from 'Multiply-and-Add' operations inside inner-product loops. Some recent works [57, 53] show how to avoid most inner-products, resorting primarily to 'xor-popcount' operations.

RAM model versus circuits. The complexity of those algorithms have been analyzed in the quantumly accessible RAM model, but considering the amount of memory they require, it is not clear whether realistic architecture would scale well. Even for classical algorithm, it is not clear that the complexity $2^{.292b+o(b)}$ can be achieved by a circuit. On the contrary, for the simplest version of the sieve algorithm with complexity $2^{0.415b}$, it seems possible to design an efficient circuit with area = time = $2^{.2075b+o(b)}$.[11] It is plausible that

[11] While this has not been studied in details for now, this was pointed out by Paul Kirchner on a public mailing-list `https://groups.google.com/d/msg/cryptanalytic-algorithms/BoSRL0uHIjM/wAkZQlwRAgAJ`.

the LSH techniques can also be applied in the circuit model to some extent, but these techniques will likely be more costly than in the RAM model, if not by exponential factors, at least by a substantial polynomial factor.

Amount of memory. Even without considering the issues with (quantumly) accessing such large amount of memory, mobilizing the required amount of memory can already be considered completely infeasible for the parameters of NewHope1024. Indeed, the fastest algorithm requires $2^{0.265b+o(b)}$ bits of storage, estimated at 2^{233} bits. It is claimed in [20] that the amount of memory may be reduced down to $2^{.2075b+o(b)}$ without affecting the asymptotic running time, but it may affect it significantly in practice. Moreover, even this amount $2^{.2075b+o(b)}$ of memory, concretely lower bounded by 2^{182} bits for NewHope1024 already exceed the numbers of atoms on earth. For NewHope512, the memory lower bound of 2^{79} bits is comparable to the total amount of data storage available world-wide in 2017 (estimated to 295 exabytes $\approx 2^{71}$ in 2007 [81], and growing at a rate of 58% per year).

We note that some variants of sieving require less memory at the cost of being significantly slower both asymptotically and in practice [16, 79].

MAXDEPTH for quantum computation. While the maximal depth of a quantum computation have a direct impact on the security level of primitives like AES, we note that this may not be the case for the lattice attacks considered here. Indeed, while sieving is subject to Grover accelerations [94, 92], these Grover search are applied to a rather small search space, and many of them may be ran in parallel. In that respect, it seems that setting MAXDEPTH to 2^{64} or even 2^{40} may not affect the efficiency of a quantum attack. For simplicity and conservatism, we prefer to not account for such a limitation on the adversary computational resources.

4.2.7 Failure analysis and attack exploiting failure

For our analysis of the failure rate of NewHope512 and NewHope1024 we follow the approach from [9]. The script in `scripts/failure-1024k8.py` gives a failure rate of less than 2^{-216} for NewHope1024. For our NewHope512 instance we obtain a similar failure rate of less than 2^{-213} as provided in the script `scripts/failure-512k8.py`.

Attacks exploiting failure have been studied in [58] against a CPA version of NewHope, and required generating about 4000 decryption requests. The attack consist of using much larger errors than defined by the protocol.

One may fear that an attacker using Grover search could produce a failing ciphertext in time about $2^{-216/2}$ for the CCA versions of our scheme. Yet, this would require the adversary to decide offline whether a ciphertext triggers failure. This is not possible, since triggering failure also involves the randomness of the decryptor's secret. Moreover, the failure rate given above are upper bounds, that we do not expect to be so tight. In conclusion, we do not expect decapsulation failures to induce any weaknesses.

5 Expected security strength

In the light of the analysis of Section 4.2 we estimate the following security levels for the two versions of our scheme, according to the 1 to 5 scale provided in Section 4.A.5 (Security Strength Categories) of the call provided by the NIST:

- NewHope512: Level 1 (equivalent to AES128, *i.e.* 2^{170}/MAXDEPTH quantum gates or 2^{143} classical gates) with a claimed post-quantum bit-security of 101 bits.
- NewHope1024: Level 5 (equivalent to AES256, *i.e.* 2^{298}/MAXDEPTH quantum gates or 2^{272} classical gates) with a claimed post-quantum bit-security of 233 bits.

The above claims are meant for *any* value of MAXDEPTH $\geq 2^{40}$. Indeed, we note that this level are easier to achieve as MAXDEPTH increase (since the security of AES decrease as MAXDEPTH increase, while MAXDEPTH does not affect the security analysis of our scheme).

In more details, the cost given in Table 12 following the methodology of [7] are directly corroborating these security strength, despite optimistic cost of Sieving, and counting CPU cycles rather than gates. The much more conservative lower bounds of Table 11 remain somewhat below the gatecounts associated to these level. Yet, in the lights of the margins discussed in Section 4.2.6, this security strength should still

be comfortably conservative. In particular, in unlike attacks against AES, the fastest attacks against our scheme resort to very large amounts of memory (at least 2^{79} bits for NewHope512, and at least 2^{182} bits for NewHope1024), which makes a direct comparison of gatecounts less relevant.

6 Advantages and limitations

6.1 Summary

From our point of view, NewHope is a fast, efficient, and simple scheme that is a suitable replacement of RSA and ECC. The main advantages of NewHope and our parameter choices are:

- **High performance.** NewHope has been implemented on a wide range of platforms and showed very good performance and features reasonable sized key and ciphertexts. Even for a category 5 scheme with 233 bits of security, performance seems to be similar to currently used elliptic curve based cryptosystems.
- **Simplicity and ease of implementation.** A basic NewHope implementation is very simple and can be done with only few lines of code in a tool like SageMath or other mathematical software. The complexity of the final reference implementation mostly stems from encoding and decoding functions that are unavoidable as well as the particular NTT implementation. Additionally, the difference between parameter sets is kept minimal as only n and γ change between NewHope512 and NewHope1024. Moreover, the NewHope-Usenix code has already been ported into various programming languages[12]. A successful integration of NewHope-Usenix into Google Chrome [95] and OpenSSL/Apache [30] shows the suitability for usage in a hybrid setup.
- **Memory efficiency.** The implicit usage of the NTT allows for memory efficient in place computation. No big temporary data structures are required.
- **Conservative design.** We claim that NewHope1024 has a considerable security margin and is based on a conservative security analysis that leaves room for improvements in cryptanalysis. Moreover, the scheme is designed to be somewhat misuse resistant: for example, the leakage of information from the system random number generator more difficult because we always hash random coins before using them.
- **Implementation security.** While more effort on implementation security is needed, some works already exist that deal with lattice-based cryptography and schemes similar to NewHope.

Some of our design choices lead to certain trade-off and we came to the conclusion we accept some disadvantages in our design due to the benefits we gain in other areas (e.g., speed, performance, simplicity). The disadvantages of NewHope we would like to point out are:

- **Small noise distribution.** The choice of $k = 8$ was made as a tradeoff for both parameter sets, to achieve negligible decryption error rates, and to simplify sampling as we can access the randomness byte-wise. However, some security could be gained by optimizing k for $n = 512$ and $n = 1024$ and for more security in an ephemeral Diffie-Hellman variant where correctness is less important (like the original NewHope-Usenix).
- **Ring-LWE**. The usage of the Ring-LWE problem is the basis for the good performance and simplicity of NewHope and currently no attacks are known that can exploit the addition structure. However, the standard LWE assumption could be considered more conservative and thus a better choice in case the next years lead to progress in the cryptanalysis of RLWE.
- **Limited Parametrization**. It is hard with the current structure of NewHope to construct a scheme that achieves NIST security category 2,3, or 4 as either ring dimension $n = 512$ or $n = 1024$ has to be used.
- **Restrictions due to usage of the NTT.** We use the NTT in our basic CPA-secure scheme for efficiency reasons and we output elements in the NTT domain. Past research shows that the NTT is a very suitable way to implement polynomial multiplication on various platforms, especially for large dimensions n. However, this design choice also somewhat restricts the implementer from choosing a polynomial multiplication algorithm of their choice, like Nussbaumer, Karatsuba, or Schoolbook multiplication, or at least leads to a performance impact of doing so. If a different polynomial

[12] See https://ianix.com/pqcrypto/pqcrypto-deployment.html.

multiplication algorithm is used, it is still required to transform elements into the NTT domain with our exact parameters.

6.2 Compatibility with existing deployments and hybrid schemes

The original IND-CPA-secure key encapsulation mechanism NewHope-Usenix has been demonstrated as suitable for use in existing network protocol deployments and in hybrid schemes by several works.

An experiment conducted by Google [95] used NewHope-Usenix alongside ephemeral elliptic curve Diffie–Hellman (ECDH) key exchange in a hybrid TLS 1.2 ciphersuite in an experimental version of the Chrome browser. In their report at the end of a 4 month experiment, Google engineers reported that they "did not find any unexpected impediment to deploying something like NewHope. There were no reported problems caused by enabling it." They further elaborated that "the median connection latency only increased by a millisecond, the latency for the slowest 5% increased by 20ms and, for the slowest 1%, by 150ms", and speculated the the latency increase was due primarily to increased communication sizes, not computational overhead due to the low computational cost of NewHope-Usenix.

Bos et al. [30] compared the performance of several post-quantum key exchange methods, including NewHope-Usenix, within TLS 1.2 using OpenSSL and Apache, measuring latency and throughput of HTTPS connections using either only post-quantum key exchange or hybrid post-quantum key exchange with ECDH in a local network environment. They found that, despite the larger communication size of NewHope-Usenix, it could support more connections per second and had lower latency than ECDH. Hybrid ECDH + NewHope-Usenix did result in a small decrease in throughput and latency compared to only ECDH connections, but only a 3–5% decrease. For details, see Table 5 of [30].

While the above results were about NewHope-Usenix, NewHope-CCA-KEM should not behave very differently. As noted earlier, the primary difference is that NewHope-CCA-KEM uses key transport to establish its shared secret, rather than reconciliation like in NewHope-Usenix, and that NewHope-CCA-KEM achieves IND-CCA security by using a variant of the Fujisaki–Okamoto transform to reconstruct and check ciphertexts. Communication sizes (ciphertexts) increase by 32 bytes, which should have minimal effect. Computation costs of NewHope-CCA-KEM are higher than NewHope-Usenix, primarily to the re-encryption in the decapsulation operation. NewHope-CCA-KEM's extremely fast performance means the cost of this re-encryption is still quite small.

6.3 Ease of implementation and hardware implementations

Implementations of the RLWE scheme of Lyubashevsky et al. [102] (LPR10) on microcontrollers are given in [100, 124]. Additionally, Infineon has announced the successful implementation of a variant of NewHope on a smart card microcontroller [87].

Several implementations of lattice-based cryptography on reconfigurable hardware have been provided so far. Instantiations of the basic LPR10 scheme on FPGAs are given in [72, 132, 123]. Works that implement NewHope-Usenix on FPGAs are [91] and [115].

6.4 Side-channel resistance

Several works already consider side-channel attacks on lattice-based primitives and the construction of countermeasures. Basic mechanism to protect the NTT and arithmetic of lattice-based schemes can be found in [133]. Works that deal proposed protected implementations of CPA-secure Ring-LWE-based scheme are [131] and [130]. Simple power analysis (SPA) attacks are proposed in [125] and [117]. The first work dealing with side-channel protection of a CCA-secure Ring-LWE-based scheme can be found in [116] (see Table 7 and Section 2.3). They provide a provably first-order secure masking scheme and its non-trivial integration into a CCA2 conversion. An interesting result is that for full protection of the secret key and message in the probing model, a masked noise sampler is required for re-encryption and a first design of corresponding protected binomial sampler is provided. The implementation and measurements were carried out on an ARM Cortex-M4F were experimentally verified by using the common non-specific t-test methodology. The masked CCA2-decryption, with very similar parameters and construction as proposed for NewHope, takes 25,334,493 cycles which is an overhead factor of 5.7 compared to the CCA2-secure

decryption without masking. Thus decryption requires roughly to 152 milliseconds runtime at 168 MHz. The overhead cost for the masking of the CCA2-secure decryption is mainly due to the high cost of the sampling. The sampling in turn heavily depends on the performance of the PRNG, in this case SHAKE128. An insecure approach with an unmasked re-encryption would require around 2 million cycles only. However, such an implementation would not provide sufficient protection against a side-channel adversary in a chosen-ciphertext scenario. Due to the high similarity of [116] and NewHope we expect very similar results for a side-channel secured implementation of our proposal.

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 5–17. ACM, 2015. https://weakdh.org/.

[2] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016. URL: https://eprint.iacr.org/2014/1025.

[3] M. Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In J. S. Vitter, editor, *Proceedings of the 30th ACM Symposium on the Theory of Computing*, pages 10–19. ACM, 1998. doi:10.1145/276698.276705.

[4] M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In J. S. Vitter, P. G. Spirakis, and M. Yannakakis, editors, *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 601–610. ACM, 2001. doi:10.1145/380752.380857.

[5] M. Albrecht, S. Bai, and L. Ducas. A subfield lattice attack on overstretched ntru assumptions. In *Advances in Cryptology – CRYPTO 2016*, volume 9814 of *LNCS*, pages 153–178. Springer, 2016. URL: https://eprint.iacr.org/2016/127.

[6] M. R. Albrecht, F. Göpfert, F. Virdia, and T. Wunderer. Revisiting the expected cost of solving usvp and applications to LWE. In *Advances in Cryptology – ASIACRYPT 2017*. Springer, 2017. To appear.

[7] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015. URL: http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.

[8] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. URL: https://eprint.iacr.org/2016/1157.

[9] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In T. Holz and S. Savage, editors, *Proceedings of the 25th USENIX Security Symposium*, pages 327–343. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[10] E. Alkim, P. Jakubeit, and P. Schwabe. NewHope on ARM Cortex-M. In C. Carlet, M. A. Hasan, and V. Saraswat, editors, *Proceedings of the 6th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, volume 10076 of *LNCS*, pages 332–349. Springer, 2016. doi:10.1007/978-3-319-49445-6_19.

[11] Y. Aono and P. Q. Nguyen. Random sampling revisited: lattice enumeration with discrete pruning. In *Advances in Cryptology – EUROCRYPT 2017*, volume 10211 of *LNCS*, pages 65–102. Springer, 2017. doi:10.1007/978-3-319-56614-6_3.

[12] Y. Aono, Y. Wang, T. Hayashi, and T. Takagi. Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator. In M. Fischlin and J. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9665 of *LNCS*, pages 789–819. Springer, 2016. doi:10.1007/978-3-662-49890-3_30.

[13] B. Applebaum, D. Cash, C. Peikert, and A. Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *LNCS*, pages 595–618. Springer, 2009. doi:10.1007/978-3-642-03356-8_35.

[14] S. Arora and R. Ge. New algorithms for learning in presence of errors. In L. Aceto, M. Henzingeri, and J. Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *LNCS*, pages 403–415. Springer, 2011. https://www.cs.duke.edu/~rongge/LPSN.pdf.

[15] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986. http://www.csie.nuk.edu.tw/~cychen/Lattices/On%20lovasz%20lattice%20reduction%20and%20the%20nearest%20lattice%20point%20problem.pdf.

[16] S. Bai, T. Laarhoven, and D. Stehlé. Tuple lattice sieving. *LMS Journal of Computation and Mathematics*, 19(A):146–162, 2016.

[17] S. Bai, A. Langlois, T. Lepoint, D. Stehlé, and R. Steinfeld. Improved security proofs in lattice-based cryptography: using the Rényi divergence rather than the statistical distance. In T. Iwata and J. H. Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, volume 9452 of *LNCS*, pages 3–24. Springer, 2015. doi:10.1007/978-3-662-48797-6_1.

[18] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom functions and lattices. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, 2012. doi:10.1007/978-3-642-29011-4_42.

[19] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *LNCS*, pages 311–323. Springer, 1987.

[20] A. Becker, L. Ducas, N. Gama, and T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2016.

[21] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Proceedings of the 9th International Conference on Public Key Cryptography (PKC)*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. doi:10.1007/11745853_14.

[22] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, T. Lange, R. Niederhagen, and C. van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive report 2014/571, 2014. URL: https://eprint.iacr.org/2014/571/.

[23] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer strikes back: new DH speed records. In T. Iwata and P. Sarkar, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 8873 of *LNCS*, pages 317–337. Springer, 2014. Full version: http://cryptojedi.org/papers/#kummer.

[24] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015. URL: https://cryptojedi.org/papers/#sphincs.

[25] D. J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems. URL: http://bench.cr.yp.to.

[26] D. J. Bernstein, P. Schwabe, and G. V. Assche. Tweetable FIPS 202, 2015. URL: http://keccak.noekeon.org/tweetfips202.html.

[27] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 313–314. Springer, 2013.

[28] J.-F. Biasse and F. Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 893–902. SIAM, 2016.

[29] A. Blum, M. L. Furst, M. J. Kearns, and R. J. Lipton. Cryptographic primitives based on hard learning problems. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO*, volume 773 of *LNCS*, pages 278–291. Springer, 1993. doi:10.1007/3-540-48329-2_24.

[30] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1006–1018, 2016. doi:10.1145/2976749.2978425.

[31] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, 2015. URL: https://eprint.iacr.org/2014/599.

[32] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, report 2017/634, 2017. URL: https://eprint.iacr.org/2017/634.

[33] J. W. Bos, K. E. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In M. Stam, editor, *Proceedings of the 14th IMA International Conference on Cryptography and Coding*, volume 8308 of *LNCS*, pages 45–64. Springer, 2013. doi:10.1007/978-3-642-45239-0_4.

[34] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls. Efficient helper data key extractor on FPGAs. In *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 181–197, 2008. doi:10.1007/978-3-540-85053-3_12.

[35] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. Classical hardness of learning with errors. In *Proceedings of the 45th ACM Symposium on Theory of Computing (STOC)*, pages 575–584. ACM, 2013. URL: http://arxiv.org/pdf/1306.0281.

[36] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. Cryptology ePrint Archive, report 2011/344, 2011. Preprint of [37]. URL: https://eprint.iacr.org/2011/344.

[37] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[38] D. Cadé, X. Pujol, and D. Stehlé. fplll 4.0.4, 2013. URL: https://github.com/dstehle/fplll.

[39] P. Campbell, M. Groves, and D. Shepherd. Soliloquy: A cautionary tale. In *ETSI 2nd Quantum-Safe Crypto Workshop*, pages 1–9, 2014.

[40] Y. Chen. *Lattice reduction and concrete security of fully homomorphic encryption*. PhD thesis, Université Paris Diderot, 2013. http://www.di.ens.fr/~ychen/research/these.pdf.

[41] Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, 2011. http://www.iacr.org/archive/asiacrypt2011/70730001/70730001.pdf.

[42] E. Chu and A. George. *Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, Boca Raton, FL, USA, 2000.

[43] J. Conway and N. J. A. Sloane. *Sphere packings, lattices and groups*, volume 290 of *Grundlehren der mathematischen Wissenschaften*. Springer Science & Business Media, 3rd edition, 1999.

[44] R. Cramer, L. Ducas, C. Peikert, and O. Regev. Recovering short generators of principal ideals in cyclotomic rings. In *Advances in Cryptology – EUROCRYPT 2016*, volume 9666 of *LNCS*, pages 559–585. Springer, 2016. doi:10.1007/978-3-662-49896-5_20.

[45] R. Cramer, L. Ducas, and B. Wesolowski. Short Stickelberger class relations and application to Ideal-SVP. In *Advances in Cryptology – EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 324–348. Springer, 2017. doi:10.1007/978-3-319-56620-7_12.

[46] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1):167–226, 2003. doi:10.1137/S0097539702403773.

[47] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-LWE encryption. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*, pages 339–344. EDA Consortium, 2015. URL: https://eprint.iacr.org/2014/725.

[48] J. Ding. New cryptographic constructions using generalized learning with errors problem. Cryptology ePrint Archive report 2012/387, 2012. URL: https://eprint.iacr.org/2012/387.

[49] J. Ding and X. Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive report 2012/688, versions 20130303:142425 and 20130303:142813, 2013. URL: https://eprint.iacr.org/2012/688.

[50] J. Ding, X. Xie, and X. Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive report 2012/688, 2012. URL: https://eprint.iacr.org/2012/688.

[51] J. Ding, X. Xie, and X. Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive report 2012/688, version 20140729:180116, 2013. URL: https://eprint.iacr.org/2012/688.

[52] Y. Dodis, L. Reyzin, and A. D. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 523–540. Springer, 2004. doi:10.1007/978-3-540-24676-3_31.

[53] L. Ducas. Shortest vector from lattice sieving: a few dimensions for free. Cryptology ePrint Archive, Report 2017/999, 2017. https://eprint.iacr.org/2017/999.

[54] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal Gaussians. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013. https://eprint.iacr.org/2013/383/.

[55] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2-3):493–514, 2015. URL: https://doi.org/10.1007/s10623-015-0087-1, doi:10.1007/s10623-015-0087-1.

[56] K. Eisenträger, S. Hallgren, A. Kitaev, and F. Song. A quantum algorithm for computing the unit group of an arbitrary degree number field. In *Proceedings of the 46th ACM Symposium on Theory of Computing (STOC)*, pages 293–302. ACM, 2014.

[57] R. Fitzpatrick, C. Bischof, J. Buchmann, Ö. Dagdelen, F. Göpfert, A. Mariano, and B.-Y. Yang. Tuning GaussSieve for speed. In *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 288–305. Springer, 2014. doi:10.1007/978-3-319-16295-9_16.

[58] S. Fluhrer. Cryptanalysis of ring-LWE based key exchange with key share reuse. Cryptology ePrint Archive report 2016/085, 2016. URL: https://eprint.iacr.org/2016/085.

[59] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999. doi:10.1007/3-540-48405-1_34.

[60] M. Fukase and K. Kashiwabara. An accelerated algorithm for solving SVP based on statistical analysis. *Journal of Information Processing*, 23(1):67–80, 2015. URL: https://www.jstage.jst.go.jp/article/ipsjjip/23/1/23_67/_article/-char/ja/.

[61] P. Gaborit. Noisy Diffie-Hellman protocols. Slides of a talk in the recent results session at *Post-Quantum Cryptography - 3rd International Workshop, PQCrypto 2010*, 2010. https://pqc2010.cased.de/rr/03.pdf.

[62] S. D. Galbraith. Space-efficient variants of cryptosystems based on learning with errors, 2013. https://www.math.auckland.ac.nz/~sgal018/compact-LWE.pdf.

[63] N. Gama and P. Nguyen. Predicting lattice reduction. In *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, 2008. doi:10.1007/978-3-540-78967-3_3.

[64] N. Gama and P. Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In C. Dwork, editor, *Proceedings of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 207–216. ACM, 2008. doi:10.1145/1374376.1374408.

[65] N. Gama, P. Q. Nguyen, and O. Regev. Lattice enumeration using extreme pruning. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. doi:10.1007/978-3-642-13190-5_13.

[66] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, 2013. doi:10.1007/978-3-642-38348-9_1.

[67] W. M. Gentleman and G. Sande. Fast Fourier transforms: for fun and profit. In *Fall Joint Computer Conference*, volume 29 of *AFIPS Proceedings*, pages 563–578, 1966. http://cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf.

[68] C. Gentry, S. Halevi, and V. Vaikuntanathan. A simple BGN-type cryptosystem from LWE. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 506–522. Springer, 2010. doi:10.1007/978-3-642-13190-5_26.

[69] C. Gentry, C. Peikert, and V. Vaikuntanathan. How to use a short basis: Trapdoors for hard lattices and new cryptographic constructions, 2008. http://web.eecs.umich.edu/~cpeikert/pubs/trap_lattice.pdf (full version of [70]).

[70] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In C. Dwork, editor, *Proceedings of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 197–206. ACM, 2008. doi:10.1145/1374376.1374407.

[71] C. Gentry and M. Szydlo. Cryptanalysis of the revised NTRU signature scheme. In L. R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 299–320. Springer, 2002. doi:10.1007/3-540-46035-7_20.

[72] N. Göttert, T. Feller, M. Schneider, J. A. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer, 2012. doi:10.1007/978-3-642-33027-8_30.

[73] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, volume 9606 of *LNCS*, pages 29–43. Springer, 2016. doi:10.1007/978-3-319-29360-8_3.

[74] S. Gueron. Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. Cryptology ePrint Archive report 2014/170, 2014. https://eprint.iacr.org/2014/170.

[75] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 530–547. Springer, 2012. doi:10.1007/978-3-642-33027-8_31.

[76] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe. Software speed records for lattice-based signatures. In P. Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013*, volume 7932 of *LNCS*, pages 67–82. Springer, 2013. http://cryptojedi.org/papers/#lattisigns.

[77] G. Hanrot, X. Pujol, and D. Stehlé. Algorithms for the shortest and closest lattice vector problems. In Y. M. Chee, Z. Guo, S. Ling, F. Shao, Y. Tang, H. Wang, and C. Xing, editors, *Coding and Cryptology – Third International Workshop, IWCC 2011*, volume 6639 of *LNCS*, pages 159–190. Springer, 2011. doi:10.1007/978-3-642-20901-7_10.

[78] G. Hanrot, X. Pujol, and D. Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, 2011. URL: https://eprint.iacr.org/2011/198.pdf.

[79] G. Herold and E. Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In *20th International Conference on Public Key Cryptography (PKC)*, volume 10174 of *LNCS*, pages 16–40. Springer, 2017. URL: https://eprint.iacr.org/2017/017.pdf.

[80] A. V. Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A. Sadeghi, I. Verbauwhede, and C. Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for PUF-enabled RFIDs. In A. D. Keromytis, editor, *16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *LNCS*, pages 374–389. Springer, 2012. doi:10.1007/978-3-642-32946-3_27.

[81] M. Hilbert and P. López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[82] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, W. Whyte, and Z. Zhang. Choosing parameters for NTRUEncrypt. Cryptology ePrint Archive report 2015/708, 2015. URL: https://eprint.iacr.org/2015/708.

[83] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: a ring-based public key cryptosystem. In J. P. Buhler, editor, *Algorithmic number theory Symposium (ANTS)*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998. https://www.securityinnovation.com/uploads/Crypto/ANTS97.ps.gz.

[84] J. Hoffstein, J. Pipher, and J. H. Silverman. *An introduction to mathematical cryptography*. Springer Verlag, 2008.

[85] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki–Okamoto transformation. In *Theory of Cryptography – 15th International Conference, TCC 2017*, volume 10677 of *LNCS*, pages 341–371. Springer, 2017. URL: https://eprint.iacr.org/2017/604.

[86] N. Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169. Springer, 2007. doi:10.1007/978-3-540-74143-5_9.

[87] Infineon Technologies. Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip. Press release, May 2017. URL: https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html.

[88] Z. Jin and Y. Zhao. Optimal key consensus in presence of noise. CoRR, report abs/1611.06150, 2016. URL: http://arxiv.org/abs/1611.06150.

[89] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 193–206, 1983. doi:10.1145/800061.808749.

[90] P. Kirchner and P.-A. Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 43–62. Springer, 2015. doi:10.1007/978-3-662-47989-6_3.

[91] P. Kuo, W. Li, Y. Chen, Y. Hsu, B. Peng, C. Cheng, and B. Yang. Post-quantum key exchange on FPGAs. Cryptology ePrint Archive, report 2017/690, 2017. URL: https://eprint.iacr.org/2017/690.

[92] T. Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015. http://www.thijs.com/docs/phd-final.pdf.

[93] T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9216 of *LNCS*, pages 3–22. Springer, 2015. doi:10.1007/978-3-662-47989-6_1.

[94] T. Laarhoven, M. Mosca, and J. van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, 2015. URL: https://eprint.iacr.org/2014/907/.

[95] A. Langley. CECPQ1 results. Blog post on imperialviolet.org, 2016. URL: https://www.imperialviolet.org/2016/11/28/cecpq1.html.

[96] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982. doi:10.1007/bf01457454.

[97] H. W. Lenstra. Euclid's algorithm in cyclotomic fields. *Journal of the London Mathematical Society*, 2(4):457–465, 1975. URL: https://openaccess.leidenuniv.nl/bitstream/handle/1887/2121/346_016.pdf.

[98] H. W. Lenstra and A. Silverberg. Revisiting the Gentry-Szydlo algorithm. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, volume 8616 of *LNCS*, pages 280–296. Springer, 2014. https://eprint.iacr.org/2014/430.

[99] R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In A. Kiayias, editor, *Topics in Cryptology - CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011. https://eprint.iacr.org/2010/613/.

[100] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient Ring-LWE encryption on 8-bit AVR processors. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015*, volume 9293 of *LNCS*, pages 663–682. Springer, 2015. https://eprint.iacr.org/2015/410/.

[101] V. Lyubashevsky. Lattice signatures without trapdoors. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, 2012. https://eprint.iacr.org/2011/537/.

[102] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010. http://www.di.ens.fr/~lyubash/papers/ringLWE.pdf.

[103] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43:1–43:35, 2013. http://www.cims.nyu.edu/~regev/papers/ideal-lwe.pdf.

[104] V. Lyubashevsky, C. Peikert, and O. Regev. A toolkit for ring-LWE cryptography. In *Advances in Cryptology - EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 35–54. Springer, 2013.

[105] A. Mariano, T. Laarhoven, and C. Bischof. A parallel variant of ldsieve for the svp on lattices. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 23–30. IEEE, 2017. URL: https://eprint.iacr.org/2016/890.pdf.

[106] D. Micciancio. CSE206A: Lattices algorithms and applications (spring 2014), 2014. Lecture notes of a course given in UCSD. See http://cseweb.ucsd.edu/classes/sp14/cse206A-a/.

[107] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2002.

[108] D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein, J. Buchmann, and E. Dahmen, editors, *Post-quantum Cryptography*, pages 147–191. Springer, 2009.

[109] D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1468–1480. SIAM, 2010. http://dl.acm.org/citation.cfm?id=1873720.

[110] A. Montanaro. Quantum walk speedup of backtracking algorithms. arXiv preprint arXiv:1509.02374, 2015. http://arxiv.org/pdf/1509.02374v2.

[111] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf.

[112] National Institute of Standards and Technology. FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[113] P. Q. Nguyen and B. Valle. *The LLL Algorithm: Survey and Applications*. Springer, 1st edition, 2009.

[114] P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008. ftp://ftp.di.ens.fr/pub/users/pnguyen/JoMC08.pdf.

[115] T. Oder and T. Güneysu. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In *Progress in Cryptology – LATINCRYPT 2017*, 2017. To appear. URL: http://www.cs.haifa.ac.il/~orrd/LC17/paper51.pdf.

[116] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-secure and masked Ring-LWE implementation. Cryptology ePrint Archive, report 2016/1109, 2016. URL: https://eprint.iacr.org/2016/1109.

[117] A. Park and D. Han. Chosen ciphertext simple power analysis on software 8-bit implementation of ring-LWE encryption. In *2016 IEEE Asian Hardware-Oriented Security and Trust, (AsianHOST)*, pages 1–6. IEEE Computer Society, 2016. doi:10.1109/AsianHOST.2016.7835555.

[118] C. Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, pages 333–342, 2009. doi:10.1145/1536414.1536461.

[119] C. Peikert. Some recent progress in lattice-based cryptography. Slides of an invited tutorial at the 6th IACR Theory of Cryptography Conference – TCC 2009, 2009. URL: http://www.cc.gatech.edu/fac/cpeikert/pubs/slides-tcc09.pdf.

[120] C. Peikert. Lattice cryptography for the Internet. In M. Mosca, editor, *Post-Quantum Cryptography*, volume 8772 of *LNCS*, pages 197–219. Springer, 2014. URL: http://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf.

[121] C. Peikert, O. Regev, and N. Stephens-Davidowitz. Pseudorandomness of ring-LWE for any ring and modulus. In *Proceedings of the 49th ACM Symposium on Theory of Computing (STOC)*, pages 461–473. ACM, 2017. URL: https://eprint.iacr.org/2017/258.

[122] T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *LNCS*, pages 353–370. Springer, 2014. https://eprint.iacr.org/2014/254/.

[123] T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In T. Lange, K. Lauter, and P. Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013. https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf.

[124] T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 346–365. Springer, 2015. doi:10.1007/978-3-319-22174-8_19.

[125] R. Primas, P. Pessl, and S. Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, 2017. doi:10.1007/978-3-319-66787-4_25.

[126] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H. N. Gabow and R. Fagin, editors, *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC)*, pages 84–93. ACM, 2005. doi:10.1145/1060590.1060603.

[127] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34, 2009. http://www.cims.nyu.edu/~regev/papers/qcrypto.pdf.

[128] O. Regev. The learning with errors problem (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity - CCC 2010*, pages 191–204. IEEE Computer Society, 2010. doi:10.1109/CCC.2010.26.

[129] A. Rényi. On measures of entropy and information. In *Fourth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 547–561, 1961. http://projecteuclid.org/euclid.bsmsp/1200512181.

[130] O. Reparaz, R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Additively homomorphic Ring-LWE masking. In *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, volume 9606 of *LNCS*, pages 233–244, 2016. doi:10.1007/978-3-319-29360-8_15.

[131] O. Reparaz, S. S. Roy, F. Vercauteren, and I. Verbauwhede. A masked Ring-LWE implementation. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop*, volume 9293 of *LNCS*, pages 683–702. Springer, 2015. doi:10.1007/978-3-662-48324-4_34.

[132] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE cryptoprocessor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, 2014. https://eprint.iacr.org/2013/866/.

[133] M. O. Saarinen. Arithmetic coding and blinding countermeasures for Ring-LWE. Cryptology ePrint Archive, report 2016/276, 2016. URL: https://eprint.iacr.org/2016/276.

[134] C. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994. doi:10.1007/BF01581144.

[135] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical computer science*, 53(2):201–224, 1987. URL: http://dx.doi.org/10.1016/0304-3975(87)90064-8.

[136] C.-P. Schnorr and M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994. doi:10.1007/BF01581144.

[137] D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 27–47. Springer, 2011. doi:10.1007/978-3-642-20465-4_4.

[138] D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology - ASIACRYPT 2009*, volume 5912, pages 617–635. Springer, 2009. URL: https://eprint.iacr.org/2009/285.

[139] E. E. Targhi and D. Unruh. Post-quantum security of the Fujisaki–Okamoto and OAEP transforms. In M. Hirt and A. D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B*, volume 9986 of *LNCS*, pages 192–216, 2016. doi:10.1007/978-3-662-53644-5_8.

[140] P. van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Universiteit van Amsterdam. Mathematisch Instituut, 1981.

[141] K. Xagawa. *Cryptography with Lattices*. PhD thesis, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2010. http://xagawa.net/pdf/2010Thesis.pdf.

NTRU-HRSS-KEM

Algorithm Specifications And Supporting Documentation

Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe

November 30, 2017

Contents

1 Written specification

1.1 Overview

NTRU-HRSS is a OWCPA-secure public key encryption scheme that was introduced in [15]. It is a direct parameterization of NTRUEncrypt as described in [12]. Its constructional novelty lies entirely in the choice of sample spaces for messages, blinding polynomials, and private keys. These spaces were chosen so that 1) NTRU-HRSS is correct (decryption never fails), 2) it admits a simple and efficient constant time implementation, and 3) it avoids the extraneous parameters common to other instantiations of NTRU.

NTRU-HRSS-KEM is a CCA2-secure KEM that was also introduced in [15]. The construction uses a generic transformation from a OWCPA-secure public key encryption scheme. As a direct KEM transform, it avoids the NAEP padding mechanism used in standard NTRUEncrypt [16]. A similar NTRUEncrypt based KEM was proposed by Martijn Stam in 2005 [21]; the main differences with that work are the underlying choice of parameters for NTRUEncrypt, and the inclusion of an additional hash that is appended to the ciphertext. The additional hash allows for a proof of security in the quantum-accessible random oracle model.

For further justification of design decisions see [15].

1.2 Mathematical definitions

1. $(\mathbb{Z}/n)^\times$ is the multiplicative group of integers modulo n.
2. Φ_n is the polynomial $(x^n - 1)/(x - 1) = x^{n-1} + x^{n-2} + \cdots + 1$.
3. R is the quotient ring $\mathbb{Z}[x]/(x^n - 1)$.
4. S is the quotient ring $\mathbb{Z}[x]/(\Phi_n)$.
5. R/q is the quotient ring $\mathbb{Z}[x]/(q, x^n - 1)$.
6. $S/2$ is the quotient ring $\mathbb{Z}[x]/(2, \Phi_n)$.
7. $S/3$ is the quotient ring $\mathbb{Z}[x]/(3, \Phi_n)$.
8. S/q is the quotient ring $\mathbb{Z}[x]/(q, \Phi_n)$.
9. $\mathsf{Mod\_Rq}(a)$ is the canonical R/q-representative of the polynomial a. The canonical R/q-representative of a is the unique polynomial b of degree at most $n-1$ with coefficients in $\{-q/2, -q/2+1, \ldots q/2-1\}$ such that a and b are equivalent as elements of R/q.
10. $\mathsf{Mod\_S2}(a)$ is the canonical $S/2$-representative of the polynomial a. The canonical $S/2$-representative of a polynomial a is the unique polynomial b of degree at most $n-2$ with coefficients in $\{0, 1\}$ such that a and b are equivalent as elements of $S/2$.
11. $\mathsf{Mod\_S3}(a)$ is the canonical $S/3$-representative of the polynomial a. The canonical $S/3$-representative of a is the unique polynomial b of degree at most $n - 2$ with coefficients in $\{-1, 0, 1\}$ such that a and b are equivalent as elements of $S/3$.
12. $\mathsf{Mod\_Sq}(a)$ is the canonical S/q-representative of the polynomial a. The canonical S/q-representative of a is the unique polynomial b of degree at most $n-2$ with coefficients in $\{-q/2, -q/2+1, \ldots q/2-1\}$ such that a and b are equivalent as elements of S/q.

Implementation notes:

1. It may be more natural, or more efficient, to represent elements of $S/3$ with coefficients in $\{0, 1, 2\}$ and/or elements of R/q with coefficients in $\{0, 1, \ldots, q - 1\}$. The algorithms below are written to be independent of the coefficient range for canonical representatives. However, the degree requirements are strict.

1.3 Parameters

1.3.1 n

An odd prime integer that satisfies the following two conditions:

- the order of 2 in $(\mathbb{Z}/n)^\times$ is $n-1$,
- the order of 3 in $(\mathbb{Z}/n)^\times$ is $n-1$.

Recommended value: 701

1.3.2 k

The Sample_T uniformity parameter. Algorithm 1.8.1 uses $2 \cdot \mathsf{k}$ pseudorandom bits to sample a single value in $\{-1, 0, 1\}$.

Recommended value: 2

1.3.3 seed_bits

The length of the seed used by the pseudorandom generator in Algorithm 1.8.1.

Recommended value: 256

1.3.4 coin_bits

The length of random bitstring used in encapsulation.

Recommended value: 256

1.3.5 shared_key_bits

The number of bits of key material produced by the KEM.

Recommended value: 256

1.4 Derived constants

The constants in this section are all functions of n. Descriptions of algorithms in later sections use these constants without explicit reference to n. Recommended values assume $n = 701$.

1.4.1 logq

The smallest positive integer such that the key encapsulation mechanism is correct with $q = 2^{\mathsf{logq}}$. See [15] for the proof of correctness.

Formula: $\lceil 7/2 + \log_2(n) \rceil$

Recommended value: 13

1.4.2 q

The smallest power of two guaranteeing correctness of the key encapsulation mechanism.

Formula: 2^{logq}

Recommended value: 8192

1.4.3 s3_packed_bits

Bit-length of the output of S3_to_bits (Section 1.9.3).

Formula: $8 \cdot \lceil (n-1)/5 \rceil$

Recommended value: 1120

1.4.4 owcpa_public_key_bits

Bit-length of the output of Rq_to_bits (Section 1.9.1).

Formula: $(n-1) \cdot \mathsf{logq}$

Recommended value: 9100

1.4.5 owcpa_private_key_bits

Bit-length of two elements of $S/3$ encoded using S3_to_bits (Section 1.9.3).

Formula: $2 \cdot \mathsf{s3\_packed\_bits}$

Recommended value: 2240

1.4.6 owcpa_ciphertext_bits

Bit-length of one elements of R/q encoded using Rq_to_bits (Section 1.9.1).

Formula: $(n-1) \cdot \mathsf{logq}$

Recommended value: 9100

1.4.7 cca_public_key_bits

Formula: owcpa_public_key_bits

Recommended value: 9100

1.4.8 cca_private_key_bits

Formula: owcpa_private_key_bits + owcpa_public_key_bits

Recommended value: 10220

1.4.9 cca_ciphertext_bits

Formula: owcpa_ciphertext_bits + s3_packed_bits

Recommended value: 10220

1.5 Summary of recommended parameters and derived constants

n	701
k	2
seed_bits	256
coin_bits	256
shared_key_bits	256
logq	13
q	8192
s3_packed_bits	1120
owcpa_public_key_bits	9100
owcpa_private_key_bits	2240
owcpa_ciphertext_bits	9100
cca_public_key_bits	9100
cca_private_key_bits	10220
cca_ciphertext_bits	10220

Table 1: Recommended parameters and derived constants

1.6 Externally defined algorithms

1.6.1 SHAKE128

Input:

- A bitstring M of arbitrary length.
- The output length parameter d.

Output:

- A bitsring of length d.

Description:

1. Output KECCAK[256]($M||1111, d$), as defined in [20].

1.7 Arithmetic Algorithms

Algorithms for integer addition, integer multiplication, polynomial addition, polynomial multiplication, and modular reduction (Mod_Rq, Mod_S2, Mod_S3, and Mod_Sq) are omitted.

1.7.1 S3_to_R

Input:

- A polynomial a.

Output:

- A polynomial b of degree at most $n-1$ that satisfies:
  - $b \equiv 0 \bmod (x-1)$,
  - $b \equiv a \bmod (p, \Phi_n)$.

Description:

1. Set $v_0 = \mathsf{S3\_inverse}(x - 1)$ [1.7.6]
2. Set $v_1 = v_0 \cdot a$
3. Set $v_2 = \mathsf{S3\_to\_Zx}(v_1)$ [1.7.2]
4. Set $b = (x - 1) \cdot v_2$
5. Output b

Implementation notes:

1. The result b is of degree at most $n - 1$ and has coefficients in $\{-1, 0, 1\}$.
2. The explicit value of $\mathsf{S3\_inverse}(x - 1)$ is

$$v_0 = \mathsf{Mod\_S3}\left((n-1) + \sum_{i=1}^{n-1} n \cdot (1 + i - n) \cdot x^i\right).$$

3. Pseudocode for a fast implementation is given in [15].

1.7.2 S3_to_Zx

Input:

– A polynomial a.

Output:

– The unique polynomial b of degree at most $n - 2$ with coefficients in $\{-1, 0, 1\}$ such that a and b are equivalent as elements of $S/3$.

Description:

1. Output $\mathsf{Mod\_S3}(a)$.

Implementation notes:

1. The set of canonical $S/3$-representatives is defined so that S3_to_Zx is trivial. An implementation that uses a different set of $S/3$-representatives may replace a call to S3_to_Zx with a contextually equivalent routine. For example, when a call to S3_to_Zx is followed by an operation in R/q the implementation may combine the lift to $\mathbb{Z}[x]$ and the reduction modulo q into a single operation.

1.7.3 Sq_to_Zx

Input:

– A polynomial a.

Output:

– The unique polynomial b of degree at most $n - 2$ with coefficients in $\{-q/2, -q/2 + 1, \ldots, q/2 - 1\}$ such that a and b are equivalent as elements of S/q.

Description:

1. Output $\mathsf{Mod\_Sq}(a)$.

1.7.4 Rq_to_Zx

Input:

– A polynomial a.

Output:

– The unique polynomial b of degree at most $n-1$ with coefficients in $\{-q/2, -q/2+1, \ldots, q/2-1\}$ such that a and b are equivalent as elements of R/q.

Description:

1. Output $\mathsf{Mod\_Rq}(a)$.

1.7.5 S2_inverse

Input:

– A polynomial a.

Output:

– An $S/2$-representative b that satisfies $\mathsf{Mod\_S2}(a \cdot b) = 1$.

Implementation notes:

1. Inverting an element of $S/2$ in constant time is nontrivial. Pseudocode for one method is provided in [15].

1.7.6 S3_inverse

Input:

– A polynomial a.

Output:

– An $S/3$-representative b that satisfies $\mathsf{Mod\_S3}(a \cdot b) = 1$.

Implementation notes:

1. Inverting an element of $S/3$ in constant time is nontrivial. Pseudocode for one method is provided in [15].

1.7.7 Sq_inverse

Input:

– A polynomial a.

Output:

– An S/q-representative b that satisfies $\mathsf{Mod\_Sq}(a \cdot b) = 1$.

Description:

1. Set $v_0 = \mathsf{S2\_inverse}(a)$ [1.7.5]
2. Set $t = 2$
3. While $t < q$
4. Set $v_0 = \mathsf{Mod\_Sq}(v_0 \cdot (2 - a \cdot v_0))$
5. Set $t = t \cdot t$

6. Endwhile
7. Output Mod_Sq(v_0)

Implementation notes:

1. When Sq_inverse is called from Generate_Public_Key [1.10.3] it is safe to replace the calls to Mod_Sq in Lines 4 and 7 with calls to Mod_Rq.

1.8 Sampling Algorithms

1.8.1 Sample_T

Input:

- A bitstring *seed* of length seed_bits.
- A bitstring *domain* of length 64.

Output:

- An $S/3$-representative.

Description:

1. Set $v = 0$ (The zero polynomial)
2. Set $i = 0$
3. Set $\ell = 2 \cdot k \cdot (n-1)$
4. Set $b_1 b_2 \ldots b_\ell = \mathsf{SHAKE128}(domain \,\|seed, \ell)$
5. While $i < n - 1$
6. Set $v_i = \sum_{j=1}^{k} b_{2ki+j} - b_{2ki+k+j}$
7. Set $i = i + 1$
8. Endwhile
9. Output Mod_S3(v)

Implementation notes:

1. In this document we use four domain strings `domm`, `domr`, `domf`, `domg`. In our implementation `domm` and `domr` are the zero string, `domf` =`0x0100000000000000` and `domg` =`0x0200000000000000`. This will likely be changed in a future version.

1.8.2 Sample_Tplus

Input:

- A bitstring *seed* of length seed_bits.
- A bitstring *domain* of length 64.

Output:

- The canonical $S/3$-representative of a polynomial $v = \sum_{i=0}^{n-2} v_i x^i$ where $v_i \in \{-1, 0, 1\}$ for all i and $\sum_{i=1}^{n-2} v_i v_{i-1} \geq 0$.

Description:

1. Set v = Sample_T($seed, domain$) [1.8.1]
2. Set $t = \sum_{i=1}^{n-2} v_i \cdot v_{i-1}$
3. Set $s = -1$ if $t < 0$, otherwise set $s = 1$

4. Set $i = 0$
5. While $i < n - 2$
6. Set $v_i = s \cdot v_i$
7. Set $i = i + 2$
8. Endwhile
9. Output $\mathsf{Mod\_S3}(v)$

Implementation notes:

1. The value t in Line 2 satisfies $-n + 1 < t < n - 1$.

1.9 Encoding Algorithms

The algorithms in this section are specified at the bit level. When converting to octets a bitstring is padded with zeros until it is of length $8 \cdot \ell$ for some ℓ. The encoding is then order preserving for indices at distance at least 8, but order reversing within octets. Hence $b_1, \ldots, b_7, b_8, b_9, \ldots, b_{15}, b_{16}$ is encoded as $b_8 b_7 \ldots b_1, b_{16} b_{15} \ldots b_9$.

1.9.1 Rq_to_bits

Input:

– A polynomial a that satisfies $a(1) \equiv 0 \pmod{q}$.

Output:

– A bitstring $b_1 b_2 \ldots b_\ell$ of length $\ell = (n-1) \cdot \mathsf{logq}$.

Description:

1. $v = \mathsf{Mod\_Rq}(a)$ (Ensure $v = \sum_{i=0}^{n-1} v_i x^i$)
2. $i = 0$
3. while $i \leq n - 2$
4. Set $b_{i\mathsf{logq}+1} b_{i\mathsf{logq}+2} \ldots b_{i\mathsf{logq}+8}$ so that $\sum_{j=1}^{\mathsf{logq}} b_{i\mathsf{logq}+j} 2^{\mathsf{logq}-j} \equiv v_i \pmod{q}$.
5. $i = i + 1$
6. endwhile
7. Output $b_1 b_2 \ldots b_\ell$.

Implementation notes:

1. The coefficient v_{n-1} is not encoded. The condition $a(1) \equiv 0 \pmod{q}$ ensures that v_{n-1} can be recovered from the first $n - 1$ coefficients.

1.9.2 Rq_from_bits

Input:

– A bitstring $b_1 b_2 \ldots b_\ell$ of length $\ell = (n-1) \cdot \mathsf{logq}$.

Output:

– An R/q-representative.

Description:

1. $v = 0$ (The zero polynomial)
2. $i = 0$

3. while $i \leq n-2$
4. Set $c = \sum_{j=1}^{\mathsf{logq}} b_{i\mathsf{logq}+j} \cdot 2^{\mathsf{logq}-j}$.
5. Set $v_i = c$
6. Set $v_{n-1} = v_{n-1} - c$
7. Set $i = i+1$
8. endwhile
9. Output $\mathsf{Mod\_Rq}(v)$

1.9.3 S3_to_bits

Input:

– A polynomial a.

Output:

– A bitstring $b_1 b_2 \ldots b_\ell$ of length $\ell = \mathsf{s3\_packed\_bits}$. [1.4.3]

Description:

1. $v = \mathsf{Mod\_S3}(a)$ (Ensure $v = \sum_{i=0}^{n-2} v_i x^i$)
2. $i = 0$
3. while $i < \lfloor (n-1)/5 \rfloor$
4. Set $c_1, c_2, \ldots, c_5 \in \{0,1,2\}$ with $c_j \equiv v_{5\cdot i+j} \pmod 3$ for $1 \leq j \leq 5$.
5. Set $b_{8\cdot i+1} b_{8\cdot i+2} \ldots b_{8\cdot i+8}$ so that $\sum_{j=1}^{8} b_{8\cdot i+j} 2^{8-j} = \sum_{j=1}^{5} c_j 3^{5-j}$.
6. $i = i+1$
7. endwhile
8. Output $b_1 b_2 \ldots b_\ell$.

Implementation notes:

1. In Line 4 we have $c_j = 0$ if $j > n-2$

1.9.4 S3_from_bits

Input:

– A bitstring $a_1 a_2 \ldots a_\ell$ of length $\ell = \mathsf{s3\_packed\_bits}$. [1.4.3]

Output:

– An $S/3$-representative.

Description:

1. $b = 0$ (The zero polynomial)
2. $i = 0$
3. while $i < \lfloor (n-1)/5 \rfloor$
4. Set $c_1, c_2, \ldots, c_5 \in \{0,1,2\}$ so that $\sum_{j=1}^{8} b_{8\cdot i+j} 2^{8-j} = \sum_{j=1}^{5} c_j 3^{5-j}$.
5. Set $b_{5\cdot i+1} = c_1$
6. Set $b_{5\cdot i+2} = c_2$
7. Set $b_{5\cdot i+3} = c_3$
8. Set $b_{5\cdot i+4} = c_4$
9. Set $b_{5\cdot i+5} = c_5$
10. $i = i+1$
11. endwhile
12. Output $\mathsf{Mod\_S3}(b)$.

1.10 Key Encapsulation Mechanism

The algorithms Generate_Key [1.10.1], Encapsulate [1.10.4], and Decapsulate [1.10.5] are a key encapsulation mechanism that is CCA-secure in the quantum random oracle model. The other functions in this section are used by these algorithms but are not part of the public API.

1.10.1 Generate_Key

Input:

- The system parameters.

Output:

- A bitstring *packed_public_key* of length cca_public_key_bits . [1.4.7]
- A bitstring *packed_private_key* of length cca_private_key_bits . [1.4.8]

Description:

1. Let *seed* be a string of seed_bits uniform random bits.
2. Set $f, f_p =$ Generate_Private_Key(*seed*) [1.10.2]
3. Set $h =$ Generate_Public_Key(*seed*, f) [1.10.3]
4. Set *packed_public_key* = Rq_to_bits(h) [1.9.1]
5. Set *packed_private_key* = S3_to_bits(f) ∥ S3_to_bits(f_p) [1.9.3]

1.10.2 Generate_Private_Key

Input:

- The system parameters.
- A bitstring *seed* of length seed_bits.

Output:

- $S/3$-representatives f and f_p that satisfy Mod_S3($f \cdot f_p$) $= 1$.

Description:

1. Set $f =$ Sample_Tplus(*seed*, `domf`) [1.8.2]
2. Set $f_p =$ S3_inverse(f) [1.7.6]
3. Output f and f_p

1.10.3 Generate_Public_Key

Input:

- The system parameters.
- A bitstring *seed* of length seed_bits.
- An $S/3$-representative f.

Output:

- An R/q-representative h that satisfies the following three conditions:
  1. $h(1) \equiv 0 \pmod q$,
  2. Mod_Rq($h \cdot f$) $= 3 \cdot (x - 1) \cdot g$ for some g with coefficients in $\{-1, 0, 1\}$,
  3. Mod_S3($h \cdot f$) $= 0$.

Description:

1. Set $v_0 = \mathsf{Sample\_Tplus}(seed, \mathtt{domg})$ [1.8.2]
2. Set $g = \mathsf{S3\_to\_Zx}(v_0)$ [1.7.2]
3. Set $v_1 = \mathsf{Sq\_inverse}(f)$ [1.7.7]
4. Set $f_q = \mathsf{Sq\_to\_Zx}(v_1)$ [1.7.3]
5. Set $v_2 = 3 \cdot (x - 1) \cdot g \cdot f_q$
6. Set $h = \mathsf{Mod\_Rq}(v_2)$
7. Output h

Implementation notes:

1. The lifts to $\mathbb{Z}[x]$ are trivial when canonical representatives are used. Implementations that use different sets of representatives may map g and f_q directly to the appropriate R/q-representatives without going through $\mathbb{Z}[x]$.

1.10.4 Encapsulate

Input:

- The system parameters.
- A bitstring *packed_public_key* of length cca_public_key_bits . [1.4.7]

Output:

- A bitstring *shared_key* of length shared_key_bits.
- A bitstring *packed_cca_ct* of length cca_ciphertext_bits . [1.4.9]

Description:

1. Let *seed* be a string of seed_bits uniform random bits.
2. Set $m = \mathsf{Sample\_T}(seed, \mathtt{domm})$ [1.8.1]
3. Set $packed\_m = \mathsf{S3\_to\_bits}(m)$ [1.9.3]
4. Set $hashes = \mathsf{SHAKE128}(packed\_m, \mathsf{coin\_bits} + \mathsf{shared\_key\_bits} + \mathsf{s3\_packed\_bits})$ [1.6.1]
5. Parse *hashes* as *coins* ‖ *shared_key* ‖ *qrom_hash* with
   - *coins* of length coin_bits,
   - *shared_key* of length shared_key_bits, and
   - *qrom_hash* of length s3_packed_bits . [1.4.3]
6. Set $packed\_owcpa\_ct = \mathsf{NTRU\_OWF\_Public}(packed\_m, packed\_public\_key, coins)$. [1.10.6]
7. Set $packed\_cca\_ct = packed\_owcpa\_ct \parallel qrom\_hash$

1.10.5 Decapsulate

Input:

- The system parameters.
- A bitstring *packed_key_pair* of length cca_private_key_bits . [1.4.8]
- A bitstring *packed_cca_ct* of length cca_ciphertext_bits . [1.4.9]

Output:

- A bitstring *shared_key* of length shared_key_bits. [1.3.5]

Description:

1. Parse *packed_key_pair* as *packed_private_key* ‖ *packed_public_key* with
   - *packed_private_key* of length owcpa_private_key_bits and [1.4.5]
   - *packed_public_key* of length owcpa_public_key_bits . [1.4.4]
2. Parse *packed_cca_ct* as *packed_owcpa_ct* ‖ *qrom_hash* with
   - *packed_owcpa_ct* of length owcpa_ciphertext_bits and [1.4.6]
   - *qrom_hash* of length s3_packed_bits . [1.4.3]
3. Set *packed_m* = NTRU_OWF_Private(*packed_private_key*, *packed_owcpa_ct*) [1.10.7]
4. Set *hashes* = SHAKE128(*packed_m*, coin_bits + shared_key_bits + s3_packed_bits) [1.6.1]
5. Parse *hashes* as *coins* ‖ *shared_key* ‖ *re_qrom_hash* with
   - *coins* of length coin_bits,
   - *shared_key* of length shared_key_bits, and
   - *re_qrom_hash* of length s3_packed_bits . [1.4.3]
6. Let *re_packed_owcpa_ct* = NTRU_OWF_Public(n, *packed_m*, *packed_public_key*, *coins*) [1.10.6]
7. If *re_packed_owcpa_ct* ‖ *re_qrom_hash* is bitwise equal to *packed_owcpa_ct* ‖ *qrom_hash*
8. Output *shared_key*.
9. Else
10. Output the zero string of length shared_key_bits.
11. Endif

1.10.6 NTRU_OWF_Public

Input:

- The system parameters.
- A bitstring *packed_m* of length s3_packed_bits . [1.4.3]
- A bitstring *packed_public_key* of length owcpa_public_key_bits . [1.4.4]
- A bitstring *coins* of length coin_bits.

Output:

- A bitstring *packed_owcpa_ct* of length owcpa_ciphertext_bits . [1.4.6]

Description:

1. Set h = Rq_from_bits(*packed_public_key*) [1.9.2]
2. Set v_0 = Sample_T(*coins*, `domr`) [1.8.1]
3. Set r = S3_to_Zx(v_0) [1.7.2]
4. Set v_1 = S3_from_bits(*packed_m*) [1.9.4]
5. Set m = S3_to_R(v_1) [1.7.1]
6. Set e = Mod_Rq($r \cdot h + m$)
7. Set *packed_owcpa_ct* = Rq_to_bits(e) [1.9.1]

1.10.7 NTRU_OWF_Private

Input:

- A bitstring *packed_private_key* of length owcpa_private_key_bits.
- A bitstring *packed_owcpa_ct* of length owcpa_ciphertext_bits.

Output:

- A bitstring *packed_m* of length s3_packed_bits.

Description:

1. Parse *packed_private_key* as *packed_f* ‖ *packed_fp* with
   - *packed_f* of length s3_packed_bits , and [1.4.3]
   - *packed_fp* of length s3_packed_bits . [1.4.3]
2. Parse *packed_owcpa_ct* as *packed_owcpa_ct* ‖ *qrom_hash* with
   - *packed_owcpa_ct* of length owcpa_ciphertext_bits , and [1.4.6]
   - *qrom_hash* of length s3_packed_bits . [1.4.3]
3. Set e = Rq_from_bits(*packed_owcpa_ct*) [1.9.2]
4. Set v_0 = S3_from_bits(*packed_f*) [1.9.4]
5. Set f = S3_to_Zx(v_0) [1.7.2]
6. Set f_p = S3_from_bits(*packed_fp*) [1.9.4]
7. Set v_1 = Mod_Rq($e \cdot f$)
8. Set v_2 = Mod_S3($v_1 \cdot f_p$)
9. Set *packed_m* = S3_to_bits(v_2) [1.9.3]
10. Output *packed_m*

2 Performance analysis

The results in this section are with respect to the parameters listed in Table 1.5.

2.1 Description of platform

In order to obtain benchmarks, we evaluate our reference implementation on a machine using the Intel x64-86 instruction set. In particular, we use a single core of a 3.5 GHz Intel Core i7-4770K CPU. We follow the standard practice of disabling TurboBoost and hyper-threading. The system has 32 KiB L1 instruction cache, 32 KiB L1 data cache, 256 KiB L2 cache and 8192 KiB L3 cache. Furthermore, it has 32GiB of RAM, running at 1333 MHz. When performing the benchmarks, the system ran on Linux kernel 4.9.0-4-amd64, Debian 9 (Stretch). We compiled the code using GCC version `6.3.0-18`, with the compiler optimization flag `-O3`.

We used the same platform described above to evaluate our AVX2 implementation. For the AVX2 implementation, we included the additional compiler flags `'-march=native'` and `'-mtune=native'`.

2.2 Time

The median resulting cycle counts are listed in the table below.

	key generation	encapsulation	decapsulation
reference C	18 151 998	1 208 946	3 578 538
optimized AVX2	294 874	38 456	68 458

2.3 Space

The public key consists of 1138 bytes, and the secret key takes 1418 bytes. The transmitted ciphertext consists of 1278 bytes. Of the ciphertext size, 140 bytes are a direct result of the transformation from the underlying OW-CPA secure scheme to the CCA2 secure KEM.

Our reference implementation uses almost 11 KiB of stack space and our AVX2 software uses just over 43 KiB, but this was not a target of optimization and should not be considered to be a lower bound.

2.4 How parameters affect performance

As the main arithmetic operations are (sub-)quadratic, we would expect that doubling n would lead to at most a factor of 4 overhead in time. Indeed, preliminary tests with our reference implementation suggest that $n = 1373$ would be less than a factor of 4 times slower. Likewise we would expect memory (and communication cost) to roughly double. Given that these parameters span an large range of relevant security levels (See Table 5), it is fair to say that parameters have only a modest impact on performance.

2.5 Optimizations

We refer to [15] for a detailed discussion of the optimizations used in our AVX2 implementation.

3 Known Answer Test values

All KAT values are included in subdirectories of the directory `KAT/ntruhrss701` of the submission package. The KAT values were generated by the `PQCgenKAT_kem` program provided by NIST. The complete list of KAT files is:

- `KAT/ntruhrss701/PQCkemKAT_1418.req`,
- `KAT/ntruhrss701/PQCkemKAT_1418.rsp`.

4 Expected security strength

4.1 Security definitions

NTRU-HRSS-KEM meets the standard IND-CCA2 security definition for a key encapsulation mechanism. Parameters have been chosen so that decryption failure is impossible, and a key can be reused at least 2^{64} times without compromising security. This follows from Dent's proof of security for the transform in [8, Table 5.], and from the presumed one-wayness of the underlying encryption scheme, NTRU-HRSS.

4.2 Rationale

Based on the analysis in Section 5.1, we expect that violating the one-wayness of NTRU-HRSS would require computational resources greater than those required to perform a key search on AES-128.

Our security claim is based primarily on Table 4, which costs the best known classical attack at 2^{136} operations and 2^{136} space. These operations mask large factors that put the true cost of the attack well above the 2^{145} bit operations required to attack AES-128, even in a RAM model.

Our security claim is also based on Table 5, which costs the best known quantum attack at 2^{123} Grover iterations. This attack is in the quantum RAM model and requires quantum-accessible classical memory of size 2^{123}. Again, large factors are ignored and it is unlikely that this attack maintains its advantage over the classical variant when it is instantiated in a quantum circuit model.

5 Analysis with respect to known attacks

5.1 Lattice attacks

Some background on lattices is assumed. In this section we denote the degree of Φ_n by $n' = n - 1$.

Each element of S can be uniquely represented by a polynomial of the form $v = \sum_{i=0}^{n'-1} v_i x^i$. The correspondence between these representatives and their coefficient vectors, $v \mapsto (v_0, \ldots, v_{n'-1}) \in \mathbb{R}^{n'}$, allows one to view S as a euclidean lattice. Lattice attacks on NTRU begin from the observation that for any $h \in S$ the set

$$L_h := \left\{(a, b) \in S^2 \ : \ b \equiv a \cdot h \pmod{q}\right\}$$

is, likewise, a lattice in $\mathbb{R}^{2 \cdot n'}$. A basis for L_h is given by the rows of the $2n' \times 2n'$ matrix

$$[L_h] = \begin{pmatrix} I_{n'} & H \\ 0 & q \cdot I_{n'} \end{pmatrix}, \tag{1}$$

where the i-th row of H for $0 \leq i \leq n' - 1$ is the coefficient vector of $\mathsf{Mod\_Sq}(x^i \cdot h)$. If h is an NTRU public key then, by construction[1], the secret key corresponds to a vector (f, g) in L_h. Since (f, g) is known to have small norm, one might hope to recover it using lattice reduction.

Attacks involving L_h have gone through several reformulations. The earliest such attack, due to Hoffstein, Pipher, and Silverman, was framed as an exact key recovery problem [11]. Coppersmith and Shamir later observed that any short vector in L_h, not just (f, g), could be used to invert the NTRU one way function [7]. It was this reformulation, as an approximation problem, that initiated the first serious efforts to understand the difficulty of finding short vectors in L_h [12, 18].

Two additional observations, which appear as early as May's work on the cryptanalysis of NTRU-107 [18], bring us to the modern attack strategy. The first is that the ratio of successive minima $\lambda_2(L_h)/\lambda_1(L_h)$ is a parameter of interest in assessing the difficulty of finding a short vector in L_h. That is to say that the NTRU problem reduces to unique SVP. Surprisingly, this brings us back to an exact search for (f, g), or one of a few related vectors, but unique SVP is known to be easier than approximate SVP in practice [9]. The second observation is that, when lattice reduction is expensive, it may be fruitful to guess a subset of the coefficients of (f, g). May's dimension reduction technique [18] and May and Silverman's pattern method [19] trade success probability in guessing coefficients of (f, g) against the cost of solving unique SVP in a sublattice of L_h.

The most effective lattice attacks in the literature treat NTRU key recovery as a unique shortest vector problem. Depending on the assumed non-asymptotic cost of lattice reduction, these attacks make use of guessing (or ignoring) coefficients to reduce the lattice dimension. Recently introduced sieve algorithms have single exponential cost that is small enough to call into question the effectiveness of guessing coefficients, however it is not clear that combinatorial techniques are irrelevant in realistic models of computation.

In Section 5.3.1 we estimate the cost of solving unique SVP in L_h in the Core-SVP cost model from the NewHope paper [3]. We do not aim to provide a complete description of the cost model, or of unique SVP methods in general. Background and references on solving unique SVP, as well as substantial discussion of the Core-SVP cost model, can be found in Albrecht–Göpfert–Virdia–Wunderer [1]. While Albrecht *et al.* focus on LWE, their analysis of LWE with short secrets applies directly to NTRU. In fact, the attack described in [1, Section 5.1] is identical[2] to May's "dimension reduction" attack applied

[1] The output of $\mathsf{Generate\_Public\_Key}$ is the R/q-representative of $3 \cdot (x - 1) \cdot g/f$, but h here should be thought of as the S/q-representative of g/f. Since $3 \cdot (x - 1)$ is a unit in S/q this change is purely syntactic.

[2] The attack in [1, Section 5.1] uses a lattice generated by the rows of

$$\begin{pmatrix} \nu I_{n'} & -A^T & 0 \\ 0 & q \cdot I_m & 0 \\ 0 & c & 1 \end{pmatrix}$$

where (A, c) is an instance of the LWE problem. By replacing $-A^T$ by H and c by an NTRU ciphertext we get a message recovery attack on NTRU. By replacing $-A^T$ by H and omitting the row containing c we get May's attack. Taking $m < n'$ corresponds to dimension reduction. The use of the parameter ν is a standard lattice reduction trick that also appears in [7] and [18].

to $[L_h]$, despite being developed independently and in a different context.

In Section 5.4, we consider Howgrave-Graham's hybrid attack [13] in the Core-SVP cost model. The hybrid attack is a unique SVP attack that uses a meet-in-the-middle strategy for guessing coefficients.

Both analyses require the following fact, which is easily derived by series multisection of $(1+x)^{2\cdot\mathsf{k}}$.

Fact 1. *Let Y be a random variable distributed according to the centered binomial distribution of parameter k. Then*

$$\begin{aligned}\Pr\left[Y \equiv 0 \pmod 3\right] &= \tfrac{1}{3}\left(1 + \tfrac{2}{2^{2\cdot\mathsf{k}}}\right),\\ \Pr\left[Y \equiv \pm 1 \pmod 3\right] &= \tfrac{1}{3}\left(1 - \tfrac{1}{2^{2\cdot\mathsf{k}}}\right).\end{aligned}$$

For the recommended parameter, $k = 2$, this gives a distribution on $(-1, 0, 1)$ of $(\frac{5}{16}, \frac{6}{16}, \frac{5}{16})$. The expected euclidean length of m coefficients is therefore $\sqrt{m \cdot 10/16}$, and the entropy per coefficient is ≈ 1.579 bits.

5.2 Cost of SVP Algorithms.

In the following sections we use four different cost estimates for SVP-b. The cost of the List-Decoding Sieve from [4] is summarized in Table 2. The cost of the List-Decoding Sieve when Grover search is used to answer nearest-neighbor queries is summarized in Table 3.

We estimate the cost of solving SVP by enumeration in dimension b using a quasilinear fit to the experimental data of Chen and Nguyen [6]. Following [2] we use the trend line:

$$enum(b) = 0.18728 \cdot b\log_2(b) - 1.0192 \cdot b + 16.10. \tag{2}$$

The cost of using an enumeration algorithm for SVP$-b$ is then estimated as $2^{enum(b)}$.

Finally we consider a hypothetical square-root speedup in the cost of enumeration on a quantum computer, for a cost of $2^{enum(b)/2}$. While this is a purely hypothetical improvement, it is no less hypothetical than a quantum variant of the List-Decoding Sieve which maintains its asymptotic cost in a quantum circuit model. As we shall see, even this dramatic speedup in enumeration is unlikely to be competitive with the *classical* List-Decoding sieve, especially if a restriction is imposed on the depth of quantum computations.

Pre-quantum cost of List-Decoding Sieve		
Metric	Time	Space
Balanced	$2^{0.292\cdot b}$	$2^{0.292\cdot b}$
Min. Space	$2^{0.368\cdot b}$	$2^{0.208\cdot b}$

Table 2: Cost of the List-Decoding Sieve as a function of the dimension, with all subexponential factors suppressed [4].

Post-quantum cost of List-Decoding Sieve		
Metric	Grover Iterations	Space
Balanced	$2^{0.265\cdot b}$	$2^{0.265\cdot b}$
Min. Space	$2^{0.2975\cdot b}$	$2^{0.208\cdot b}$

Table 3: Cost of the List-Decoding Sieve, as a function of the dimension, when Grover search is used to answer nearest neighbor queries [17]. Again subexponential factors are suppressed, but we have units of Grover iterations rather than time. This is because it is not clear that the algorithm can be instantiated in a quantum circuit model without re-evaluating its asymptotic cost.

5.3 Core-SVP Cost Estimates.

The Core-SVP estimate was introduced in the security evaluation of NewHope [3]. A thorough description can be found in [1]. Success criteria for the primal and hybrid attacks are given in the following sections.

5.3.1 Primal Attack.

The primal attack has two parameters: b, the blocksize used for lattice reduction, and m, the dimension reduction parameter[3]. The primal attack attempts to solve unique SVP in a sublattice $\Lambda_{primal} \subseteq L_h$ of rank $d = n' + m$ and volume q^m. A BKZ-b reduced basis $V = \{v_1, \ldots, v_d\}$ for Λ_{primal} is computed. Following this, a single call to an SVP-b routine is made on $\{v'_{d-b+1}, v'_{d-b+2}, \ldots v'_d\}$, where v'_i is v_i projected orthogonally to the first $d-b$ vectors of V. The parameters b and m are chosen so that a short vector in this projected sublattice is likely to be mapped to a short vector in Λ_{primal} by Babai's nearest plane algorithm.

The success condition is with respect to the length of the first vector in the last block of the reduced basis, v_{d-b+1}. The Gram-Schmidt vectors of the reduced basis, $v_1^*, v_2^*, \ldots, v_d^*$, are expected to satisfy $||v_{i+1}|| \leq \delta^{d-2i} \cdot q^{m/d}$ where $\delta = ((\pi \cdot b)^{1/b} \cdot b/(2\pi \cdot e))^{1/(2(b-1))}$. The assumption that this is the case is known as the Geometric Series Assumption.

In [3], it was suggested that an attacker could expect to recover (f, g) from its projection orthogonal to the first $d-b$ vectors of V if

$$\sqrt{b/d} \cdot ||(f, g)|| \leq ||v^*_{d-b+1}|| \approx \delta^{2b-d} \cdot q^{m/d}. \tag{3}$$

Further evidence for this claim was given in [1].

From Fact 1 we have $\sqrt{b/d} \cdot ||(f, g)|| \approx \sqrt{b} \cdot \sqrt{10/16}$. The Core-SVP cost of the attack is found by minimizing the cost of one call to an algorithm for SVP-b over all choices of m and b for which (3) is satisfied.

In Tables 4 and 5 we give optimal parameters for the primal attack in the Core-SVP model. The main entries of interest are for $n' = 700$, corresponding to our recommended parameter set. We also list the cost for $n' = 940$ and $n' = 1372$, as these give some indication for how security scales with n and may be useful in comparisons with other proposals.

Primal Attack with List-Decoding Sieve					
n'	m	b	Metric	Operations	Vectors
700	626	465	Balanced	2^{136}	2^{136}
			Min. Space	2^{171}	2^{96}
940	824	616	Balanced	2^{180}	2^{180}
			Min. Space	2^{226}	2^{127}
1372	1150	969	Balanced	2^{283}	2^{283}
			Min. Space	2^{357}	2^{201}

Table 4: Optimal parameters for the primal attack when the cost of SVP$-b$ is as given in Table 2.

[3] In the LWE context m is the number of LWE samples used by the attacker.

Primal Attack with List-Decoding Sieve and Grover Search						
n'	m	b	Metric	Grover Iterations	Iteration Depth	Vectors
700	626	465	Balanced	2^{123}	2^{26}	2^{123}
			Min. Space	2^{138}	2^{41}	2^{96}
940	824	616	Balanced	2^{163}	2^{35}	2^{163}
			Min. Space	2^{183}	2^{55}	2^{127}
1372	1150	969	Balanced	2^{257}	2^{55}	2^{257}
			Min. Space	2^{288}	2^{87}	2^{201}

Table 5: Optimal parameters for the primal attack when the cost of SVP$-b$ is as given in Table 3.

Cost of quantum queries The quantum cost of the list decoding sieve depends, crucially, on the use of the quantum RAM model of computation. Determining its cost in a circuit model is an open problem of considerable interest. Each Grover iteration accesses a block of memory of size roughly equal to the square of the figure given in the "Iteration Depth" column. If quantum RAM requires active error correction, then the advantage over the Table 4 cost would be lost.

Size of vectors Even assuming just b bits per vector, the Table 4 cost of attacking our recommended parameter set would inflate to 2^{145} bits. Merely populating this memory would already be as expensive than a key search on AES-128.

Effect of MAXDEPTH The List Decoding Sieve allows for a large degree of parallelization and can be tuned to avoid reasonable MAXDEPTH bounds in a classical RAM model. To a lesser extent, this is also true when Grover search is used to perform individual near-neighbor searches. We have listed the "Iteration Depth" of these searches, i.e. the number of Grover iterations required to perform each near-neighbor query, in Table 5.

Depending on the circuit cost of one Grover iteration, it is plausible that a small MAXDEPTH could be saturated. For example, the time optimal parameterization in dimension 465 has iteration depth 2^{26}. This would saturate a 2^{40} limit on MAXDEPTH if the circuit for a Grover iteration had depth 2^{14} quantum gates. For comparison, the circuit for one *round* of AES-128 given in [10] has depth $2^{13.4}$ logical quantum gates. That said, limiting MAXDEPTH to 2^{64} would likely have no impact on the Core-SVP security estimate for our $n = 701$ parameter set.

5.4 Hybrid attack

The hybrid attack targets a sublattice $\Lambda_{hybrid} \subseteq L_h$ of rank $d = n' + m$. A BKZ-b reduced basis $V = \{v_1, \ldots, v_{d-s}\}$ for a rank $d - s$ sublattice of Λ_{hybrid} is computed. Suppose w is a short vector in L_h. The attacker attempts to guess s coefficients of the projection of w orthogonal to V. If the guess is correct, then it can be lifted to a short vector in Λ_{hybrid} using Babai's nearest plane algorithm. The success condition is with respect to the length of the *last* Gram-Schmidt vector in the last block of the reduced basis, v^*_{d-s}. Heuristically, one can expect the attack to work when

$$||v^*_{d-s}|| = \delta^{2s-d+2} \cdot q^{m/d} \geq 2 \cdot ||w||_\infty = 2.$$

We have found that the hybrid attack is not competitive with the primal attack when both attacks use the List-Decoding Sieve with cost given by Table 3. However there are still some interesting trade-offs to consider. Recall that the costs in Table 3 depend on the use of the quantum RAM model. It is not clear whether quantum RAM is less expensive than general purpose quantum circuitry. This leads us to consider parallel hybrid attacks that use less quantum circuitry than would be required to run the List-Decoding Sieve on a quantum computer.

In Table 6 we see that the hybrid attack can outperform the classical List-Decoding Sieve if Grover iterations are inexpensive. However this is only true when MAXDEPTH $= \infty$. Reasonable limits on MAXDEPTH eliminate the advantage of the hybrid attack.

Hybrid Attack with List-Decoding Sieve							
n'	MAXDEPTH	m	b	s	Grover Iterations	Iteration Depth	Processors
700	∞	626	459	169	2^{133}	2^{133}	2^{0}
	2^{128}	621	461	167	2^{134}	2^{128}	2^{6}
	2^{96}	641	482	150	2^{140}	2^{96}	2^{44}
	2^{64}	660	502	134	2^{146}	2^{64}	2^{82}
940	∞	822	614	227	2^{179}	2^{179}	2^{0}
1372	∞	1164	980	363	2^{286}	2^{286}	2^{0}

Table 6: Cost of the hybrid attack using the List-Decoding Sieve with cost given by Table 2. The coefficient guessing stage of the attack is done with Grover search. The classical time and space required for the list-decoding sieve is matched to the number of Grover iterations. The MAXDEPTH limit only affects the coefficient guessing stage.

In Table 7 we consider the cost of the hybrid attack when enumeration is used to solve SVP$-b$. This is an attractive option as it requires only polynomial space. However, as the table indicates, reasonable limits on MAXDEPTH force one to use massive amounts of parallelism. Even with MAXDEPTH $= 2^{128}$ it is clearly better to use a space optimized (classical) List-Decoding Sieve.

Hybrid Attack with Enumeration and Grover Search							
n'	MAXDEPTH	m	b	s	Grover Iterations	Iteration Depth	Processors
700	∞	508	345	264	2^{208}	2^{208}	2^{0}
	2^{128}	546	383	232	2^{241}	2^{128}	2^{110}
	2^{96}	558	396	221	2^{252}	2^{96}	2^{156}
	2^{64}	574	411	208	2^{265}	2^{64}	2^{200}
940	∞	654	446	373	2^{294}	2^{294}	2^{0}
1372	∞	860	653	626	2^{494}	2^{494}	2^{0}

Table 7: Optimal parameters for the hybrid attack when SVP$-b$ is solved by enumeration of cost $2^{enum(b)}$. The formula for $enum(b)$ is Equation (2). The coefficient guessing stage of the attack is done with Grover search. The MAXDEPTH limit only affects the coefficient guessing stage.

Finally in Table 8 we consider the effect of a hypothetical square-root speedup in the cost of enumeration. We assume that this speedup is due to a quantum algorithm, so it is limited by MAXDEPTH. Even with this massive speedup, it seems that reasonable limits on MAXDEPTH force a high degree of parallelism. The entry with MAXDEPTH $= 2^{96}$ is interesting, but it seems likely that the attack in Table 4 is better.

Hybrid Attack with $\sqrt{\text{Enumeration}}$ and Grover Search							
n'	MAXDEPTH	m	b	s	Grover Iterations	Iteration Depth	Processors
700	∞	602	440	184	2^{145}	2^{145}	2^{0}
	2^{128}	602	440	184	2^{163}	2^{128}	2^{34}
	2^{96}	602	440	184	2^{195}	2^{96}	2^{98}
	2^{64}	602	440	184	2^{227}	2^{64}	2^{162}
940	∞	781	572	263	2^{207}	2^{207}	2^{0}
1372	∞	1060	866	453	2^{357}	2^{357}	2^{0}

Table 8: Optimal parameters for the hybrid attack when SVP$-b$ is solved by enumeration of cost $2^{enum(b)/2}$. The formula for $enum(b)$ is Equation (2). The coefficient guessing stage of the attack is done with Grover search. The MAXDEPTH limit only affects all parts of the computation.

5.5 Attacks on symmetric primitives

The only symmetric primitive we use is SHAKE128. This also meets the Category 1 security level. We note that the KEM can, in principle, be used to exchange close to $n \cdot \log 3$ bits of key material. A more secure symmetric primitive can be substituted without changing any other details of the construction.

6 Advantages and limitations

We focus on comparisons with other lattice based systems.

6.1 Compared with Standard NTRU.

Some advantages and disadvantages of NTRU-HRSS compared with Standard NTRU (as defined in [16]) are as follows.

Advantages.

- **No decryption failures.** NTRU EES parameter sets have small but non-zero decryption failure probability.
- **No padding mechanisms.** By using a direct construction of a KEM, we have avoided the need for a padding mechanism like NAEP [14].
- **No fixed weight distributions.** NTRU EES parameter sets use fixed weight coefficient vectors to ensure that information about secret keys (resp. messages) is not revealed through $h(1)$ (resp $c(1)$). This is more difficult to implement in constant time than the combination of Sample_T and S3_to_R used in NTRU-HRSS.
- **No rejection sampling.** NTRU EES uses fixed length strings of uniform random bits to sample uniform random trits and uniform values in $\{0, 1, \ldots, n-1\}$. In order for these processes to succeed with all but negligible probability, many bits must be sampled.
- **Secret keys are always invertible.** The restrictions on n listed in Section 1.3.1 ensure that f is always invertible modulo 2. NTRU EES parameters are chosen so that the probability of generating a non-invertible f is small but not necessarily zero.

Disadvantages.

- **Large modulus.** NTRU-HRSS needs a comparatively large modulus to eliminate decryption failure. This decreases security and increases communication cost. The $n = 701$ parameter set for NTRU-HRSS would gain an estimated 20 bits of security by using modulus $q = 2048$ instead of $q = 8192$. Ciphertext length would also drop by 175 bytes.
- **Inverses mod p.** NTRU EES takes $f \equiv 1 \pmod p$ and thereby avoids multiplying by $f^{-1} \pmod p$ during decryption. Doing the same in NTRU-HRSS would require an even larger modulus.

6.2 Compared with Streamlined NTRUPrime.

Some advantages and disadvantages of NTRU-HRSS compared with Streamlined NTRUPrime (as defined in [5]) are as follows.

6.2.1 Advantages.

- **No fixed weight distributions.** Streamlined NTRUPrime uses relies on fixed weight distributions for its proof of correctness.
- **Private keys are always invertible.** It is possible to pick an f that is not invertible modulo 3 in Streamlined NTRUPrime.
- **Power of 2 modulus.** Streamlined NTRUPrime requires a prime modulus. Some arithmetic operations are faster when q is a power of 2.

6.2.2 Disadvantages.

- **Cyclotomic ring.** NTRUPrime was designed to avoid "worrisome structure" of cyclotomic rings. While algebraic structure does not figure into the cost of the best known attacks on NTRU-HRSS, it is conceivable that better algebraic attacks exist. It is also conceivable that such attacks would not apply to the rings used by NTRUPrime.
- **Probabilistic encryption.** Streamlined NTRUPrime is constructed as a deterministic public key encryption scheme.
- **No "LWR"-style rounding.** Streamlined NTRUPrime ciphertexts can be compressed.

6.3 Compared with LWE systems.

Advantages.

- **No decryption failures.** Most practical LWE schemes opt for a small decryption failure rate rather than for a narrow coefficient distribution or a large modulus.

Disadvantages.

- **Larger dimension for equal security.** In order to eliminate decryption failure, NTRU-HRSS uses trinary secret keys and messages. This results in a lower level of security than could be had with the same dimension and modulus but larger noise.

References

[1] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving usvp and applications to LWE. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 297–322, 2017. 18, 20

[2] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. IACR Cryptology ePrint Archive report 2015/046, 2015. https://eprint.iacr.org/2015/046. 19

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In Thorsten Holz and Stefan Savage, editors, *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. https://cryptojedi.org/papers/#newhope. 18, 20

[4] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *"27th Annual ACM-SIAM Symposium on Discrete Algorithms"*. ACM-SIAM, 2016. 19

[5] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime. In Jan Camenisch and Carlisle Adams, editors, *Selected Areas in Cryptography – SAC 2017*, LNCS, to appear. Springer, 2017. http://ntruprime.cr.yp.to/papers.html. 24

[6] Yuanmi Chen. *Lattice reduction and concrete security of fully homomorphic encryption*. PhD thesis, l'Université Paris Diderot, 2013. 19

[7] Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *LNCS*, pages 52–61. Springer, 1997. http://dx.doi.org/10.1007/3-540-69053-0_5. 18

[8] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding*, volume 2898 of *LNCS*, pages 133–151. Springer, 2003. http://www.cogentcryptography.com/papers/designer.pdf. 17

[9] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008: 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, LNCS, pages 31–51. Springer, 2008. https://www.iacr.org/archive/eurocrypt2008/49650031/49650031.pdf. 18

[10] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. *Applying Grover's Algorithm to AES: Quantum Resource Estimates*, pages 29–43. Springer International Publishing, Cham, 2016. 21

[11] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A new high speed public key cryptosystem, 1996. draft from at CRYPTO '96 rump session. http://web.securityinnovation.com/hubfs/files/ntru-orig.pdf. 18

[12] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory – ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998. http://dx.doi.org/10.1007/BFb0054868. 4, 18

[13] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169. Springer, 2007. http://www.iacr.org/archive/crypto2007/46220150/46220150.pdf. 19

[14] Nick Howgrave-Graham, Joseph H. Silverman, Ari Singer, and William Whyte. NAEP: Provable security in the presence of decryption failures. Cryptology ePrint Archive, Report 2003/172, 2003. https://eprint.iacr.org/2003/172. 23

[15] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, LNCS. Springer, 2017. http://cryptojedi.org/papers/#ntrukem. 4, 5, 8, 9, 17

[16] IEEE. IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices. IEEE Std 1363.1-2008, 2009. http://dx.doi.org/10.1109/IEEESTD.2009.4800404. 4, 23

[17] Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015. http://www.thijs.com/docs/phd-final.pdf. 19

[18] Alexander May. Cryptanalysis of NTRU, 1999. https://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/paper/cryptanalysisofntru.ps. 18

[19] Alexander May and Joseph H. Silverman. Dimension reduction methods for convolution modular lattices. In Joseph H. Silverman, editor, *Cryptography and Lattices: International Conference – CaLC 2001*, volume 2146 of *LNCS*, pages 110–125. Springer, 2001. http://dx.doi.org/10.1007/3-540-44670-2_10. 18

[20] NIST. FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 7

[21] Martijn Stam. A key encapsulation mechanism for NTRU. In Nigel P. Smart, editor, *Cryptography and Coding*, volume 3796 of *LNCS*, pages 410–427. Springer, 2005. 4

NTRU Prime

20171130

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Daniel J. Bernstein, University of Illinois at Chicago
- Chitchanok Chuengsatiansup, INRIA and ENS de Lyon
- Tanja Lange, Technische Universiteit Eindhoven
- Christine van Vredendaal, Technische Universiteit Eindhoven

E-mail address (preferred): `authorcontact-ntruprime@box.cr.yp.to`

Telephone (if absolutely necessary): +1-312-996-3422

Postal address (if absolutely necessary): Daniel J. Bernstein, Department of Computer Science, University of Illinois at Chicago, 851 S. Morgan (M/C 152), Room 1120 SEO, Chicago, IL 60607–7053.

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: ×. See also printed version of "Statement by Each Submitter".

Document generated with the help of `pqskeleton` version 20171123.

Contents

1 Introduction

A 2015 algorithm breaks dimension-N SVP (under plausible assumptions) in time $2^{(c+o(1))N}$ as $N \to \infty$ with $c \approx 0.292$. See [9]. For comparison, the best algorithm known just five years earlier had a much worse $c \approx 0.415$, and the best algorithm known just ten years before that took time $2^{\Theta(N \log N)}$.

Gentry's original FHE system at STOC 2009, with standard "cyclotomic" choices of rings, is now known (again under plausible assumptions) to be broken in polynomial time by a quantum algorithm. See [12]. Peikert claimed in 2015 that the weakness in Gentry's system was specific to Gentry's short generators and inapplicable to Ideal-SVP:

> Although cyclotomics have a lot of structure, nobody has yet found a way to exploit it in attacking Ideal-SVP/BDD ... For commonly used rings, principal ideals are an extremely small fraction of all ideals. ... The weakness here is not so much due to the structure of cyclotomics, but rather to the extra structure of principal ideals that have short generators.

However, the attack was then combined with further features of cyclotomics to break Ideal-SVP (again under plausible assumptions) with approximation factor $2^{N^{1/2+o(1)}}$, a terrifying advance compared to the previous $2^{N^{1+o(1)}}$. See [24].

As these attack examples illustrate, the security of lattice-based cryptography is not well understood. There are serious risks of further advances in

- SVP algorithms,
- algorithms that exploit the "approximation factors" used in cryptography,
- algorithms that exploit the structure of cryptographic problems such as LWE,
- algorithms that exploit the multiplicative structure of *efficient* cryptographic problems such as Ring-LWE,
- algorithms that exploit the structure of these problems for the *specific* rings chosen by users, and
- algorithms to break cryptosystems without breaking these problems.

The point of this submission is that the attack surface in lattice-based cryptography can be significantly reduced with only a minor loss of efficiency. In fact, despite the extra security criteria imposed below, the two cryptosystems in this submission are two of the smallest and fastest lattice-based cryptosystems.

2 General algorithm specification (part of 2.B.1)

This submission provides two key-encapsulation mechanisms: "Streamlined NTRU Prime" and "NTRU LPRime".

2.1 Streamlined NTRU Prime parameter space

Streamlined NTRU Prime has parameters (p, q, w) subject to the following restrictions: p is a prime number; q is a prime number; w is a positive integer; $2p \geq 3w$; $q \geq 16w + 1$; $x^p - x - 1$ is irreducible in the polynomial ring $(\mathbb{Z}/q)[x]$.

We abbreviate the ring $\mathbb{Z}[x]/(x^p - x - 1)$, the ring $(\mathbb{Z}/3)[x]/(x^p - x - 1)$, and the field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ as $\mathcal{R}$, $\mathcal{R}/3$, and $\mathcal{R}/q$ respectively. We refer to an element of $\mathcal{R}$ as **small** if all of its coefficients are in $\{-1, 0, 1\}$, and **weight w** if exactly w of its coefficients are nonzero.

Streamlined NTRU Prime also has the following parameters: an encoding of public keys as strings; an encoding of rounded ring elements (see below) as strings; and a hash function mapping each small polynomial to two fixed-length output strings, a "confirmation" and a "session key".

2.2 Streamlined NTRU Prime key generation

The receiver generates a public key as follows:

- Generate a uniform random small element $g \in \mathcal{R}$. Repeat this step until g is invertible in $\mathcal{R}/3$. (There are various standard ways to test invertibility: for example, one can check divisibility of g by the irreducible factors of $x^p - x - 1$ modulo 3, or one can deduce invertibility as a side effect of various algorithms to compute $1/g$ in $\mathcal{R}/3$.)
- Generate a uniform random small weight-w element $f \in \mathcal{R}$. (Note that f is nonzero and hence invertible in $\mathcal{R}/q$, since $w \geq 1$.)
- Compute $h = g/(3f)$ in $\mathcal{R}/q$. (By assumption q is a prime larger than 3, so 3 is invertible in $\mathcal{R}/q$, so $3f$ is invertible in $\mathcal{R}/q$.)
- Encode h as a string $\overline{h}$, using the aforementioned encoding of public keys as strings. The public key is $\overline{h}$.
- Save the following secrets: f in $\mathcal{R}$; and $1/g$ in $\mathcal{R}/3$.

2.3 Streamlined NTRU Prime encapsulation

The sender generates a ciphertext as follows:

- Decode the public key $\overline{h}$, obtaining $h \in \mathcal{R}/q$.
- Generate a uniform random small weight-w element $r \in \mathcal{R}$.
- Compute $hr \in \mathcal{R}/q$.
- Round each coefficient of hr, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3, producing $c \in \mathcal{R}$. (If $q \in 1+3\mathbb{Z}$ then each coefficient of c is in $\{-(q-1)/2, \ldots, -6, -3, 0, 3, 6, \ldots, (q-1)/2\}$. If $q \in 2+3\mathbb{Z}$ then each coefficient of c is in $\{-(q+1)/2, \ldots, -6, -3, 0, 3, 6, \ldots, (q+1)/2\}$. Rounding adds an element from $\{-1, 0, 1\}$ to each coefficient.)
- Encode c as a string $\underline{c}$, using the aforementioned encoding of rounded ring elements as strings.
- Hash r, obtaining a confirmation C and a session key K. The ciphertext is the concatenation $C\,\underline{c}$.

2.4 Streamlined NTRU Prime decapsulation

The receiver decapsulates a ciphertext $C\,\underline{c}$ as follows:

- Decode $\underline{c}$, obtaining $c \in \mathcal{R}$.
- Multiply by $3f$ in $\mathcal{R}/q$.
- View each coefficient of $3fc$ in $\mathcal{R}/q$ as an integer between $-(q-1)/2$ and $(q-1)/2$, and then reduce modulo 3, obtaining a polynomial e in $\mathcal{R}/3$.
- Multiply by $1/g$ in $\mathcal{R}/3$.
- Lift e/g in $\mathcal{R}/3$ to a small polynomial $r' \in \mathcal{R}$.
- Compute c', C', K' from r' as in encapsulation.
- If r' is small, r' has weight w, $c' = c$, and $C' = C$, then output K'. Otherwise output False.

If $C\,\underline{c}$ is a legitimate ciphertext then c is obtained by rounding the coefficients of hr to the nearest multiples of 3; i.e., $c = m + hr$ in $\mathcal{R}/q$, where m is small. All coefficients of the polynomial $3fm + gr$ in $\mathcal{R}$ are in $[-8w, 8w]$ by Theorem 2 below, and thus in $[-(q-1)/2, (q-1)/2]$ since $q \geq 16w+1$. Viewing each coefficient of $3fc = 3fm+gr$ as an integer in $[-(q-1)/2, (q-1)/2]$ thus produces exactly $3fm+gr \in \mathcal{R}$, and reducing modulo 3 produces $gr \in \mathcal{R}/3$; i.e., $e = gr$ in $\mathcal{R}/3$, so $e/g = r$ in $\mathcal{R}/3$. Lifting now produces exactly r since r is small; i.e., $r' = r$. Hence $(c', C', K') = (c, C, K)$. Finally, $r' = r$ is small, r' has weight w, $c' = c$, and $C' = C$, so decapsulation outputs $K' = K$, the same session key produced by encapsulation.

Theorem 1 *Fix integers $p \geq 3$ and $w \geq 1$. Let $r, g \in \mathbb{Z}[x]$ be polynomials of degree at most $p-1$ with all coefficients in $\{-1, 0, 1\}$. Assume that r has at most w nonzero coefficients. Then $gr \bmod x^p - x - 1$ has each coefficient in the interval $[-2w, 2w]$.*

Theorem 2 *Fix integers $p \geq 3$ and $w \geq 1$. Let $m, r, f, g \in \mathbb{Z}[x]$ be polynomials of degree at most $p-1$ with all coefficients in $\{-1, 0, 1\}$. Assume that f and r each have at most w nonzero coefficients. Then $3fm + gr \bmod x^p - x - 1$ has each coefficient in the interval $[-8w, 8w]$.*

2.5 NTRU LPRime parameter space

NTRU LPRime has parameters (p, q, w, δ, I) subject to the following restrictions: p is a prime number; q is a prime number; w, δ, I are positive integers; $2p \geq 3w$; I is a multiple of 8; $p \geq I$; $q \geq 16w + 2\delta + 3$; $x^p - x - 1$ is irreducible in the polynomial ring $\mathcal{R}/q$.

As before, we abbreviate the ring $\mathbb{Z}[x]/(x^p - x - 1)$, the ring $(\mathbb{Z}/3)[x]/(x^p - x - 1)$, and the field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ as $\mathcal{R}$, $\mathcal{R}/3$, and $\mathcal{R}/q$ respectively. We refer to an element of $\mathcal{R}$ as **small** if all of its coefficients are in $\{-1, 0, 1\}$, and **weight w** if exactly w of its coefficients are nonzero.

NTRU LPRime also has the following parameters: an encoding of rounded ring elements (see below) as strings; a hash function mapping each I-bit string to three fixed-length output strings, a "cipher key" and a "confirmation" and a "session key"; a function Small from the set of cipher keys to the set of small weight-w elements in $\mathcal{R}$; a function Generator from a set of "seed" strings to $\mathcal{R}/q$; a function Top from $(\mathbb{Z}/q)^I$ to a fixed-length set of strings; and a function Right from the same set of strings to $(\mathbb{Z}/q)^I$ such that each coordinate of the difference $\text{Right}(\text{Top}(C)) - C$ is in $\{0, 1, \ldots, \delta\}$ for each $C \in (\mathbb{Z}/q)^I$.

2.6 NTRU LPRime key generation

The receiver generates a public key as follows:

- Generate a uniform random seed S.
- Compute $G = \text{Generator}(S) \in \mathcal{R}/q$.
- Generate a uniform random small weight-w element $a \in \mathcal{R}$.
- Compute $aG \in \mathcal{R}/q$.
- Round each coefficient of aG, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3, producing $A \in \mathcal{R}$.
- Encode A as a string $\underline{A}$. The public key is the concatenation $S\underline{A}$.
- Save the secret a.

2.7 NTRU LPRime encapsulation

The sender generates a ciphertext as follows:

- Decode the public key $S\underline{A}$, obtaining a seed S and a polynomial $A \in \mathcal{R}$.
- Compute $G = \text{Generator}(S) \in \mathcal{R}/q$.
- Generate a uniform random I-bit string $r = (r_0, r_1, \ldots, r_{I-1})$.
- Hash r, obtaining a cipher key k, a confirmation H, and a session key K.
- Compute $b = \text{Small}(k) \in \mathcal{R}$.
- Compute bG in $\mathcal{R}/q$.
- Compute bA in $\mathcal{R}/q$. (Only the bottom I coefficients of bA, the coefficients $(bA)_0, (bA)_1, \ldots, (bA)_{I-1}$ of $x^0, x^1, \ldots, x^{I-1}$ respectively, will be used; other coefficients do not need to be computed.)
- Round each coefficient of bG, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3, producing $B \in \mathcal{R}$.
- Encode B as a string $\underline{B}$.
- Compute $C = (C_0, C_1, \ldots, C_{I-1}) \in (\mathbb{Z}/q)^I$ as follows: $C_j = (bA)_j + r_j(q-1)/2$.
- Compute $\tilde{C} = \text{Top}(C)$.
- The ciphertext is the concatenation $H\underline{B}\tilde{C}$. The session key is K.

2.8 NTRU LPRime decapsulation

The receiver decapsulates a ciphertext $H\underline{B}\tilde{C}$ as follows:

- Decode $\underline{B}$, obtaining $B \in \mathcal{R}$.
- Compute $T = \text{Right}(\tilde{C}) \in (\mathbb{Z}/q)^I$.
- Compute aB in $\mathcal{R}/q$. (Only the bottom I coefficients of aB will be used.)
- Compute $r'_0, r'_1, \ldots, r'_{I-1} \in \{0, 1\}$ as follows. View $T_j - (aB)_j + 4w + 1 \in \mathbb{Z}/q$ as an integer between $-(q-1)/2$ and $(q-1)/2$. Then r'_j is the sign bit of this integer: 1 if the integer is negative, otherwise 0.
- Compute a ciphertext c' and session key K' from $r' = (r'_0, r'_1, \ldots, r'_{I-1})$ as in encapsulation.
- If the ciphertext c' is $H\underline{B}\tilde{C}$, then output K'. Otherwise output False.

The public key A is obtained by rounding the coefficients of aG to the nearest multiples of 3; i.e., $A = aG + d$ in $\mathcal{R}/q$, where d is small.

If $H\underline{B}\tilde{C}$ is a legitimate ciphertext then $\underline{B}$ is an encoding of B which is obtained by rounding the coefficients of bG to the nearest multiples of 3; i.e., $B = bG + e$ in $\mathcal{R}/q$, where e is small, and $\tilde{C} = \mathrm{Top}(C)$ with $C_j = (bA)_j + r_j(q-1)/2$.

By construction the functions Top and Right are such that each coordinate of $\mathrm{Right}(\mathrm{Top}(C)) - C$ is in $\{0, 1, \ldots, \delta\}$ for each $C \in (\mathbb{Z}/q)^I$, i.e,. $\mathrm{Right}(\mathrm{Top}(C))_j - C_j \in \{0, 1, \ldots, \delta\}$.

Then

$$\begin{aligned}
T_j - (aB)_j + 4w + 1 &= \mathrm{Right}(\mathrm{Top}(C))_j - (a(bG+e))_j + 4w + 1 \\
&= \mathrm{Right}(\mathrm{Top}(C))_j - C_j + C_j - ((abG)_j + (ae)_j) + 4w + 1 \\
&= \mathrm{Right}(\mathrm{Top}(C))_j - C_j + (bA)_j + r_j(q-1)/2 - ((abG)_j + (ae)_j) + 4w + 1 \\
&= \mathrm{Right}(\mathrm{Top}(C))_j - C_j + (baG)_j + (bd)_j + r_j(q-1)/2 - ((abG)_j + (ae)_j) + 4w + 1 \\
&= \mathrm{Right}(\mathrm{Top}(C))_j - C_j + (bd)_j - (ae)_j + 4w + 1 + r_j(q-1)/2 \in \mathbb{Z}/q.
\end{aligned}$$

All coefficients of the polynomials bd and ae are in $[-2w, 2w]$ by Theorem 1, thus

$$1 \leq \mathrm{Right}(\mathrm{Top}(C))_j - C_j + (bd)_j - (ae)_j + 4w + 1 \leq 8w + \delta + 1.$$

Viewing each coefficient of $T_j - (aB)_j + 4w + 1$ as an integer in $[-(q-1)/2, (q-1)/2]$ thus produces an integer in $[1, 8w+\delta+1]$ if and only if $r_j = 0$ and an integer in $[-(q-1)/2, -(q-1)/2 + 8w + \delta]$ if and only if $r_j = 1$ because $8w + \delta + 1 \leq (q-1)/2$ by construction.

This means that $r'_j = r_j$, thus $r' = r$ and $c' = c$, so decapsulation outputs $K' = K$, the same session key produced by encapsulation.

3 List of parameter sets (part of 2.B.1)

3.1 Parameter set `kem/sntrup4591761`

Streamlined NTRU Prime with $p = 761$, $q = 4591$, $w = 286$, and the following functions.

Encoding of public keys as strings: View the input polynomial in little-endian form as a sequence of coefficients of $x^0, x^1, \ldots, x^{764}$. The coefficients of $x^{761}, \ldots, x^{764}$ are always 0.

View each coefficient in $\mathbb{Z}/4591$ as an element of $\{-2295, \ldots, 2295\}$. Add 2295 to obtain an element of $\{0, \ldots, 4590\}$.

Write each batch of 5 elements c_0, c_1, c_2, c_3, c_4 in radix $6144 = 3 \cdot 2^{11}$ as the integer $c_0 + 6144c_1 + 6144^2c_2 + 6144^3c_3 + 6144^4c_4$. This integer is below $6144^5 < 2^{63}$. Write this integer as 8 bytes in little-endian form.

This produces $8(765/5) = 1224$ bytes. The last 6 bytes are always 0 and are suppressed, so a public key is encoded as 1218 bytes.

Encoding of rounded ring elements as strings: View the input polynomial in little-endian form as a sequence of coefficients of $x^0, x^1, \ldots, x^{761}$. The coefficient of x^{761} is always 0.

View each coefficient in $\mathbb{Z}/4591$ as an element of $\{-2295, -2292, \ldots, 2292, 2295\}$; recall that ciphertext coefficients are always multiples of 3. Add 2295 to obtain an element of $\{0, 3, \ldots, 4587, 4590\}$. Divide by 3 to obtain an element of $\{0, 1, \ldots, 1530\}$.

Write each batch of 3 elements c_0, c_1, c_2 in radix $1536 = 3 \cdot 2^9$ as the integer $c_0 + 1536c_1 + 1536^2c_2$. This integer is below $1536^3 < 2^{32}$. Write this integer as 4 bytes in little-endian form.

This produces $4(762/3) = 1016$ bytes. The last byte is always 0 and is suppressed, so a rounded ring element is encoded as 1015 bytes.

Hash function: View the input polynomial r in little-endian form as a sequence of coefficients of $x^0, x^1, \ldots, x^{763}$. The coefficients of $x^{761}, x^{762}, x^{763}$ are always 0.

Add 1 to each coefficient, obtaining an element of $\{0, 1, 2\}$. Write each batch of 4 elements in radix 4, obtaining a byte. Overall this produces $764/4 = 191$ bytes.

Hash the resulting byte string with SHA-512, obtaining a 256-bit confirmation followed by a 256-bit session key.

3.2 Parameter set `kem/ntrulpr4591761`

NTRU LPRime with $p = 761$, $q = 4591$, $w = 250$, $\delta = 292$, $I = 256$, and the following functions.

Encoding of rounded ring elements as strings: Same as in `sntrup4591761`.

Hash function: View the 256-bit string r in little-endian form as a 32-byte string, i.e. the first byte of r is $r_0 + 2r_1 + \cdots + 128r_7$, the next byte is $r_8 + 2r_9 + \cdots + 128r_{15}$, etc.

Hash r with SHA-512, obtaining a 32-byte cipher key k followed by a 32-byte intermediate key k'. Hash k' with SHA-512, obtaining a 32-byte confirmation followed by a 32-byte session key.

Mapping to $\mathcal{R}$: For each 32-byte string k, $\mathrm{Small}(K) \in \mathcal{R}$ is defined as follows:

- Use AES-256-CTR with key k, starting from counter 0, to generate $4p$ bytes of output.
- View each 4 bytes of output in little-endian form, obtaining p elements of $\{0, 1, \ldots, 2^{32} - 1\}$.
- Clear the bottom bit of each of the first w integers; now each of those integers is 0 modulo 2.

- Set the bottom bit, and clear the next bit, of each of the remaining $p - w$ integers; now each of those integers is 1 modulo 4.
- Sort the integers.
- Reduce each integer modulo 4, and subtract 1, obtaining p elements of $\{-1, 0, 1\}$, of which exactly w are nonzero.
- View these elements as a polynomial in little-endian form, namely Small(K).

Mapping to $\mathcal{R}/q$: The set of seeds is the set of 32-byte strings. For each 32-byte string K, Generator(K) $\in \mathcal{R}/q$ is defined as follows:

- Use AES-256-CTR with key K, starting from counter 0, to generate $4p$ bytes of output.
- View each 4 bytes of output in little-endian form, obtaining p elements of $\{0, 1, \ldots, 2^{32} - 1\}$.
- Reduce each of these elements modulo q, obtaining p elements of $\{0, 1, \ldots, q - 1\}$.
- Obtain p elements of $\{-(q-1)/2, \ldots, (q-1)/2\}$ by subtractng $(q-1)/2$ from each integer.
- View these elements as a polynomial in little-endian form, namely Generator(K).

Top bits: For each $C \in (\mathbb{Z}/q)^{256}$, Top($C$) is a 128-byte string defined as follows:

- View each C_j as an integer between -2295 and 2295.
- Compute $T_j = \lfloor (114(C_j + 2156) + 16384)/32768 \rfloor \in \{0, 1, \ldots, 15\}$ for each j.
- Define $\text{Top}(C) = (T_0 + 16T_1, T_2 + 16T_3, \ldots, T_{254} + 16T_{255})$.

For each 128-byte string T, Right(T) $\in (\mathbb{Z}/q)^{256}$ is defined as follows:

- Extract $T_0, T_1, \ldots, T_{255} \in \{0, 1, \ldots, 15\}$ from T in little-endian form.
- Compute $R_j = 287T_j - 2007$ for each j.
- Define $\text{Right}(T) = (R_0, R_1, \ldots, R_{255})$.

One can check each integer $c \in \{-2295, \ldots, 2295\}$ to see that $(287t - 2007) - c \in \{0, 1, \ldots, 292\}$ where $t = \lfloor (114(c + 2156) + 16384)/32768 \rfloor$.

4 Design rationale (part of 2.B.1)

There are many different ideal-lattice-based public-key encryption schemes in the literature, including many versions of NTRU; many Ring-LWE-based cryptosystems; and now Streamlined NTRU Prime and NTRU LPRime. These are actually many different points in a high-dimensional space of possible cryptosystems. We give a unified description of the advantages and disadvantages of what we see as the most important options in each dimension, in particular explaining the choices that we made in Streamlined NTRU Prime and NTRU LPRime. Beware that there are many interactions between options. For example, using Gaussian errors is incompatible with eliminating decryption failures, because there is always a small probability of large samples combining with large values. Using *truncated* Gaussian errors is compatible with eliminating decryption failures, but requires a much larger modulus q. Neither of these options is compatible with the simple tight KEM that we use.

4.1 The ring

The choice of cryptosystem includes a choice of a monic degree-p polynomial $P \in \mathbb{Z}[x]$ and a choice of a positive integer q. As in Section 2, we abbreviate the ring $\mathbb{Z}[x]/P$ as $\mathcal{R}$, and the ring $(\mathbb{Z}/q)[x]/P$ as $\mathcal{R}/q$.

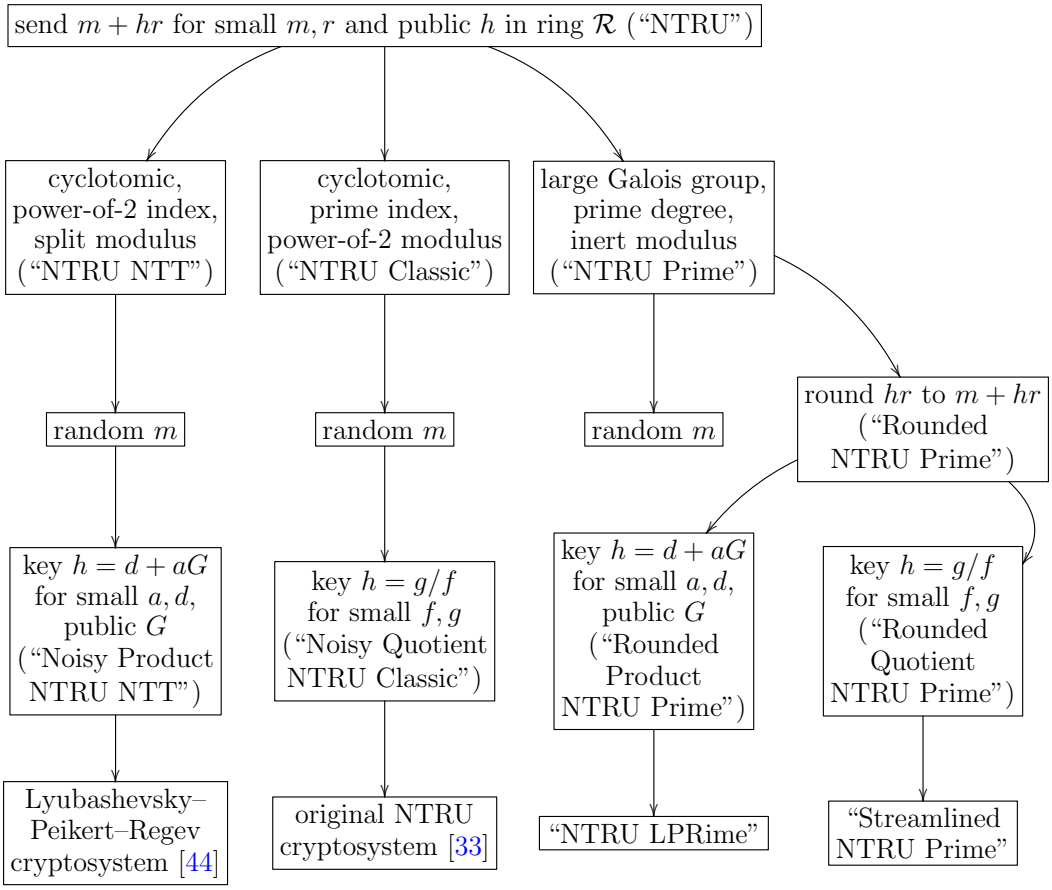
Common choices of $\mathcal{R}/q$ are as follows:

- "NTRU Classic": Rings of the form $(\mathbb{Z}/q)[x]/(x^p - 1)$, where p is a prime and q is a power of 2, are used in the original NTRU cryptosystem [33].
- "NTRU NTT": Rings of the form $(\mathbb{Z}/q)[x]/(x^p + 1)$, where p is a power of 2 and $q \in 1 + 2p\mathbb{Z}$ is a prime, are used in typical "Ring-LWE-based" cryptosystems such as [3].
- "NTRU Prime": Fields of the form $(\mathbb{Z}/q)[x]/(x^p - x - 1)$, where p is prime, are used in this submission.

NTRU Prime uses a prime-degree number field with a large Galois group and an inert modulus, minimizing the number of ring homomorphisms available to the attacker. As an analogy, conservative prime-field discrete-logarithm systems also minimize the number of ring homomorphisms available to the attacker.

We expect the future situation, like the current situation, to be a mix of the following three scenarios:

- Some lattice-based systems are broken whether or not they have unnecessary homomorphisms. As an analogy, some discrete-logarithm systems are broken whether or not they have unnecessary homomorphisms.



- Some lattice-based systems are unbroken whether or not they have unnecessary homomorphisms. As an analogy, some discrete-logarithm systems are unbroken whether or not they have unnecessary homomorphisms.
- Some lattice-based systems are broken *only if* they have unnecessary homomorphisms. As an analogy, some discrete-logarithm systems are broken only if they have unnecessary homomorphisms. Eliminating unnecessary homomorphisms rescues these systems, and removes the need to worry about what attackers can do with these homomorphisms.

The current situation is that homomorphisms eliminated by NTRU Prime are used in the following attack papers: [18], [28], [23], [20], [24], and [8]. See our "NTRU Prime" paper for further details.

4.2 The public key

The receiver's public key, which we call h, is an element of $\mathcal{R}/q$.

4.3 Inputs and ciphertexts

In the original NTRU system, ciphertexts are elements of the form $m + hr \in \mathcal{R}/q$. Here $h \in \mathcal{R}/q$ is the public key as above, and m, r are small elements of $\mathcal{R}$ chosen by the sender.

Subsequent systems labeled as "NTRU" have generally extended ciphertexts to include additional information, for various reasons explained below; but these cryptosystems all share the same core design element, sending $m + hr \in \mathcal{R}/q$ where m, r are small secrets and h is public. We suggest systematically using the name "NTRU" to refer to this design element, and more specific names (e.g., "NTRU Classic" vs. "NTRU Prime") to refer to other design elements.

We refer to (m, r) as "input" rather than "plaintext" because in any modern public-key cryptosystem the input is randomized and is separated from the sender's plaintext by symmetric primitives such as hash functions. See Section 4.5.

In the original NTRU specification [33], m was allowed to be any element of $\mathcal{R}$ having all coefficients in a standard range. The range was $\{-1, 0, 1\}$ for all of the suggested parameters, with q not a multiple of 3, and we focus on this case for simplicity (although we note that some other lattice-based cryptosystems have taken the smaller range $\{0, 1\}$, or sometimes larger ranges).

Current NTRU Classic specifications such as [32] prohibit m that have an unusually small number of 0's or 1's or -1's. For random m, this prohibition applies with probability $<2^{-10}$, and in case of failure the sender can try encoding the plaintext as a new m, but this is problematic for applications with hard real-time requirements. The reason for this prohibition is that NTRU Classic gives the attacker an "evaluate at 1" homomorphism from $\mathcal{R}/q$ to $\mathbb{Z}/q$, leaking $m(1)$. The attacker scans many ciphertexts to find an occasional ciphertext where the value $m(1)$ is particularly far from 0; this value constrains the search space for the corresponding m by enough bits to raise security concerns. In NTRU Prime, $\mathcal{R}/q$ is a field, so this type of leak cannot occur.

Streamlined NTRU Prime actually uses a different type of ciphertext, which we call a "rounded ciphertext". The sender chooses a small r as input and computes $hr \in \mathcal{R}/q$. The sender obtains the ciphertext by rounding each coefficient of hr, viewed as an integer between $-(q-1)/2$ and $(q-1)/2$, to the nearest multiple of 3. This ciphertext can be viewed as an example of the original ciphertext $m + hr$, but with m chosen so that each coefficient of $m + hr$ is in a restricted subset of $\mathbb{Z}/q$.

With the original ciphertexts, each coefficient of $m + hr$ leaves 3 possibilities for the corresponding coefficients of hr and m. With rounded ciphertexts, each coefficient of $m + hr$ also leaves 3 possibilities for the corresponding coefficients of hr and m, except that the

boundary cases $-(q-1)/2$ and $(q-1)/2$ (assuming $q \in 1+3\mathbb{Z}$) leave only 2 possibilities. In a pool of 2^{64} rounded ciphertexts, the attacker might find one ciphertext that has 15 of these boundary cases out of 761 coefficients; these occasional exceptions have very little impact on known attacks. It would be possible to randomize the choice of multiples of 3 near the boundaries, but we prefer the simplicity of having the ciphertext determined entirely by r. It would also be possible to prohibit ciphertexts at the boundaries, but as above we prefer to avoid restarting the encryption process.

More generally, we say "Rounded NTRU" for any NTRU system in which m is chosen deterministically by rounding hr to a standard subset of $\mathbb{Z}/q$, and "Noisy NTRU" for the original version in which m is chosen randomly. Rounded NTRU has two advantages over Noisy NTRU. First, it reduces the space required to transmit $m+hr$. Second, the fact that m is determined by r simplifies protection against chosen-ciphertext attacks; see Section 4.5.

[49, Section 4] used an intermediate non-deterministic possibility to provide some space reduction for a public-key cryptosystem: first choose m randomly, and then round $m+hr$, obtaining $m'+hr$. The idea of rounded hr as a *deterministic* substitute for noisy $m+hr$ was introduced in [7] in the context of a symmetric-key construction, was used in [5] to construct another public-key encryption system, and was further studied in [13] and [4]. All of the public-key cryptosystems in these papers have ciphertexts longer than Noisy NTRU, but applying the same idea to Noisy NTRU produces Rounded NTRU, which has shorter ciphertexts.

4.4 Key generation and decryption

In the original NTRU cryptosystem, the public key h has the form $3g/f$ in $\mathcal{R}/q$, where f and g are secret. Decryption computes $fc = fm+3gr$, reduces modulo 3 to obtain fm, and multiplies by $1/f$ to obtain m.

Streamlined NTRU Prime changes the position of the 3, taking h as $g/(3f)$ rather than $3g/f$. Decryption computes $3fc = 3fm+gr$, reduces modulo 3 to obtain gr, and multiplies by $1/g$ to obtain r. This change lets us compute (m,r) by first computing r and then multiplying by h, whereas otherwise we would first compute m and then multiply by $1/h$. One advantage is that we skip computing $1/h$; another advantage is that we need less space for storing a key pair. This $1/h$ issue does not arise for NTRU variants that compute r as a hash of m, but those variants are incompatible with rounded ciphertexts, as discussed in Section 4.5.

More generally, we say "Quotient NTRU" for NTRU with h computed as a ratio of two secret small polynomials. An alternative is what we call "Product NTRU", namely NTRU with h of the form $d+aG$, where a and d are secret small polynomials. Here $G \in \mathcal{R}/q$ is public, like h, but unlike h it does not need a hidden multiplicative structure: it can be, for example, a standard chosen randomly by a trusted authority, or output of a long hash function applied to a standard randomly chosen seed, or (as proposed in [3]) output of a long hash function applied to a per-receiver seed supplied along with h as part of the public key.

Product NTRU does not allow the same decryption procedure as Quotient NTRU. The first

Product NTRU system, introduced by Lyubashevsky, Peikert, and Regev in [44] (originally in talk slides in 2010), sends $e + rG$ as additional ciphertext along with $m + hr + M$, where, as before, m and r are small polynomials, e is another small polynomial, and M is a polynomial consisting of solely 0 or $\lfloor q/2 \rfloor$ in each position. The receiver computes $(m + hr + M) - a(e + rG) = M + m + dr - ae$, and rounds to 0 or $\lfloor q/2 \rfloor$ in each position, obtaining M. Note that $m + dr - ae$ is small, since all of m, d, r, a, e are small.

The ciphertext size here, two elements of $\mathcal{R}/q$, can be improved in various ways. One can replace hr with fewer coefficients, for example by summing batches of two or three coefficients [53], before adding M and m. Rounded Product NTRU rounds $hr+M$ to obtain $m+hr+M$, rounds rG to obtain $e+rG$, and (to similarly reduce key size) rounds aG to obtain $d+aG$. Decryption continues to work even if $m + hr + M$ is compressed to two bits per coefficient.

A disadvantage of Product NTRU is that r is used twice, exposing approximations to both rG and hr. This complicates security analysis compared to simply exposing an approximation to hr. State-of-the-art attacks against Ring-LWE, which reveals approximations to any number of random public multiples of r, are significantly faster for many multiples than for one multiple. Perhaps this indicates a broader weakness, in which each extra multiple hurts security.

Quotient NTRU has an analogous disadvantage: if one moves far enough in the parameter space [39] then state-of-the-art attacks distinguish g/f from random more efficiently than they distinguish $m + hr$ from random. Perhaps this indicates a broader weakness. On the other hand, if one moves far enough in another direction in the parameter space [61], then g/f has a security proof.

We find both of these issues worrisome: it is not at all clear which of Product NTRU and Quotient NTRU is a safer option.[1] We see no way to simultaneously avoid both types of complications. We have opted to present details of Streamlined NTRU Prime, an example of Quotient NTRU Prime; and of NTRU LPRime, an example of Product NTRU Prime.

If exposing approximations to two multiples of r damages the security of Product NTRU, perhaps exposing fewer bits does less damage. The compression techniques mentioned above, such as replacing $m + hr + M$ with fewer coefficients and releasing only a few top bits of each coefficient, naturally expose fewer bits than uncompressed ciphertexts. NTRU LPRime releases a few top bits of each of the bottom coefficients of $m + hr + M$, enough coefficients to communicate a hard-to-guess input M.

The Quotient NTRU literature, except for the earliest papers, takes f of the form $1 + 3F$, where F is small. This eliminates the multiplication by the inverse of f modulo 3. In Streamlined NTRU Prime we have chosen to skip this speedup for two reasons. First, in the long run we expect cryptography to be implemented in hardware, where a multiplication

[1] Peikert claimed in [50], modulo terminology, that Product NTRU is "at least as hard" to break as Quotient NTRU (and "likely strictly harder"). This claim ignores the possibility of attacks against the reuse of r in Product NTRU. There are no theorems justifying Peikert's claim, and we are not aware of an argument that eliminating this reuse is less important than eliminating the g/f structure. For comparison, switching from NTRU NTT and NTRU Classic to NTRU Prime eliminates structure used in some state-of-the-art attacks without providing new structure used in other attacks.

in $\mathcal{R}/3$ is far less expensive than a multiplication in $\mathcal{R}/q$. Second, this speedup requires noticeably larger keys and ciphertexts for the same security level, and this is important for many applications, while very few applications will notice the CPU time for Streamlined NTRU Prime.

4.5 Padding, KEMs, and the choice of q

In Streamlined NTRU Prime and NTRU LPRime we use the modern "KEM+DEM" approach introduced by Shoup; see [58]. This approach is much nicer for implementors than previous approaches to public-key encryption. For readers unfamiliar with this approach, we briefly review the analogous options for RSA encryption.

RSA maps an input m to a ciphertext $m^e \bmod n$, where (n, e) is the receiver's public key. When RSA was first introduced, its input m was described as the sender's plaintext. This was broken in reasonable attack models, leading to the development of various schemes to build m as some combination of fixed padding, random padding, and a short plaintext; typically this short plaintext is used as a shared secret key. This turned out to be quite difficult to get right, both in theory (see, e.g., [59]) and in practice (see, e.g., [46]), although it does seem possible to protect against arbitrary chosen-ciphertext attacks by building m in a sufficiently convoluted way.

The "KEM+DEM" approach, specifically Shoup's "RSA-KEM" in [58] (also called "Simple RSA"), is much easier:

- Choose a uniform random integer m modulo n. This step does not even look at the plaintext.
- To obtain a shared secret key, simply apply a cryptographic hash function to m.
- Encrypt and authenticate the sender's plaintext using this shared key.

Any attempt to modify m, or the plaintext, will be caught by the authenticator.

"KEM" means "key encapsulation mechanism": $m^e \bmod n$ is an "encapsulation" of the shared secret key $H(m)$. "DEM" means "data encapsulation mechanism", referring to the encryption and authentication using this shared secret key. Authenticated ciphers are normally designed to be secure for many messages, so $H(m)$ can be reused to protect further messages from the sender to the receiver, or from the receiver back to the sender. It is also easy to combine KEMs, for example combining a pre-quantum KEM with a post-quantum KEM, by simply hashing the shared secrets together.

When NTRU was introduced, its input (m, r) was described as a sender plaintext m combined with a random r. This is obviously not secure against chosen-ciphertext attacks. Subsequent NTRU papers introduced various mechanisms to build (m, r) as increasingly convoluted combinations of fixed padding, random padding, and a short plaintext.

It is easy to guess that KEMs simplify NTRU, the same way that KEMs simplify RSA; we are certainly not the first to suggest this. However, all the NTRU-based KEMs we have found in the literature (e.g., [60] and [55]) construct the NTRU input (m, r) by hashing a shorter input and verifying this hash during decapsulation; typically r is produced as a hash of m. These KEMs implicitly assume that m and r can be chosen independently, whereas rounded ciphertexts (see Section 4.3) have r as the sole input. It is also not clear that generic-hash chosen-ciphertext attacks against these KEMs are as difficult as inverting the NTRU map from input to ciphertext: the security theorems are quite loose.

We instead follow a simple generic KEM construction introduced in the earlier paper [25, Section 6] by Dent, backed by a tight security reduction [25, Theorem 8] saying that generic-hash chosen-ciphertext attacks are as difficult as inverting the underlying function:

- Like RSA-KEM, this construction hashes the input, in our case r, to obtain the session key.
- Decapsulation verifies that the ciphertext is the correct ciphertext for this input, preventing per-input ciphertext malleability.
- The KEM uses additional hash output for key confirmation, making clear that a ciphertext cannot be generated except by someone who knows the corresponding input.

Key confirmation might be overkill from a security perspective, since a random session key will also produce an authentication failure; but key confirmation allows the KEM to be audited without regard to the authentication mechanism, and adds only 3% to our ciphertext size.

Dent's security analysis assumes that decryption works for all inputs. We achieve this in Streamlined NTRU Prime by requiring $q \geq 16w + 1$. Recall that decryption sees $3fm + gr$ in $\mathcal{R}/q$ and tries to deduce $3fm + gr$ in $\mathcal{R}$; the condition $q \geq 16w + 1$ guarantees that this works, since each coefficient of $3fm + gr$ in $\mathcal{R}$ is between $-(q-1)/2$ and $(q-1)/2$ by Theorem 2. Taking different shapes of m, r, f, g, or changing the polynomial $P = x^p - x - 1$, would change the bound $16w + 1$; for example, replacing g by $1 + 3G$ would change $16w + 1$ into $24w + 3$.

Similarly, NTRU LPRime takes $q \geq 16w + 2\delta + 3$ to avoid decryption failures. Sending along merely top bits of $m + hr + M$ means that there is an additional error, producing a slightly worse bound than in the Streamlined NTRU Prime case. Another difference in details is that decryption reconstructs only M, not m; NTRU LPRime chooses r deterministically[2] as a hash of M.

In lattice-based cryptography it is standard to take somewhat smaller values of q. The idea is that coefficients in $3fm + gr$ are produced as sums of many $+1$ and -1 terms, and these

[2]This requires another layer of security analysis beyond Dent's security analysis. The core question is whether it is hard to recover a random M from ciphertext and public key, when r is chosen randomly. The next question, the extra layer, is whether it is hard to recover a random M from ciphertext and public key, when r is chosen as a hash of M. The third question, addressed by Dent's security analysis, is whether the KEM is hard to break.

terms *usually* cancel, rather than conspiring to produce the maximum conceivable coefficient. However, this idea led to attacks that exploited occasional decryption failures; see [35] and, for an analogous attack on code-based cryptography using QC-MDPC codes, [30]. It is common today to choose q so that decryption failures will occur with, e.g., probability 2^{-80}; but this does not meet Dent's assumption that decryption always works. This nonzero failure rate appears to account for most of the complications in the literature on NTRU-based KEMs. We prefer to guarantee that decryption works, making the security analysis simpler and more robust.

4.6 The shape of small polynomials

As noted in Section 4.3, the coefficients of m are chosen from the limited range $\{-1, 0, 1\}$. The NTRU literature [33, 37, 31, 32] generally puts the same limit on the coefficients of r, g, and f, except that if f is chosen with the shape $1 + 3F$ (see Section 4.4) then the literature puts this limit on the coefficients of F. Sometimes these "ternary polynomials" are further restricted to "binary polynomials", excluding coefficient -1.

The NTRU literature further restricts the Hamming weight of r, g, and f. Specifically, a cryptosystem parameter is introduced to specify the number of 1's and -1's. For example, there is a parameter t (typically called "d" in NTRU papers) so that r has exactly t coefficients equal to 1, exactly t coefficients equal to -1, and the remaining $p - 2t$ coefficients equal to 0. These restrictions allow decryption for smaller values of q (see Section 4.5), saving space and time. Beware, however, that if t is *too* small then there are attacks; see our security analysis in Section 6.

In Streamlined NTRU Prime we keep the requirement that r have Hamming weight $w = 2t$, and keep the requirement that these w nonzero coefficients are all in $\{-1, 1\}$, but we drop the requirement of an equal split between -1 and 1. This allows somewhat more choices of r. The same comments apply to f. Similarly, we require g to have all coefficients in $\{-1, 0, 1\}$ but the distribution is otherwise unconstrained. We also require that f and g be invertible in $\mathcal{R}/q$, which simply means nonzero given that $P(x)$ is irreducible for NTRU Prime, and that g be invertible in $\mathcal{R}/3$.

These changes would affect the conventional NTRU decryption procedure: they expand the *typical* size of coefficients of fm and gr, forcing larger choices of q to avoid *noticeable* decryption failures. But we instead choose q to avoid *all* decryption failures (see Section 4.5), and these changes do not expand our *bound* on the size of the coefficients of fm and gr.

In NTRU LPRime we similarly choose small weight-w polynomials with coefficients in $\{-1, 0, 1\}$ without restricting the distribution of -1 and 1 beyond the weight.

Elsewhere in the literature on lattice-based cryptography one can find larger coefficients: consider, e.g., the quinary polynomials in [27], and the even wider range in [3]. In [61], the coefficients of f and g are sampled from a very wide discrete Gaussian distribution, allowing a proof regarding the distribution of g/f. However, this appears to produce *worse* security for any given key size. Specifically, there are no known attack strategies blocked

by a Gaussian distribution, while the very wide distribution forces q to be very large to enable decryption (see Section 4.5), producing a much larger key size (and ciphertext size) for the same security level. Furthermore, wide Gaussian distributions are practically always implemented with variable-time algorithms, creating security problems, as illustrated by the successful cache-timing attacks in [17] and [51].

5 Detailed performance analysis (2.B.2)

5.1 Description of platform

The following measurements were collected using `supercop-20170904` running on a computer named `titan0`. The CPU on `titan0` is an Intel Xeon E3-1275 v3 (Haswell) running at 3.5 GHz. Turbo Boost is disabled. `titan0` has 32GB of RAM and runs Ubuntu 14.04. Benchmarks used `./do-part`, which ran on one core of the CPU. The compiler list was reduced to just `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv`.

NIST says that the "NIST PQC Reference Platform" is "an Intel x64 running Windows or Linux and supporting the GCC compiler." `titan0` is an Intel x64 running Linux and supporting the GCC compiler. Beware, however, that different Intel CPUs have different cycle counts.

5.2 Time

In the first measurement run (many timings), the median encapsulation time for `sntrup4591761` was 59456 cycles, and the median decapsulation time was 97684 cycles. Timings were practically identical in the second measurement run (59476, 97624) and the third measurement run (59508, 97692).

Key-generation time was slower, over 6 million cycles. With more effort one can eliminate most of these cycles,[3] but our current key-generation cost is already negligible. Specifically:

- The standard design goal of IND-CCA2 security means that it is safe to generate a key once and use the key any number of times. The situation in several recent lattice-based KEMs (for example, BCNS [15], New Hope [3], and Frodo [14]) is completely different: they are not designed to resist, and do not resist, chosen-ciphertext attacks, so they generate a new key for every ciphertext, so their key-generation time is important.
- Forward secrecy does *not* require constant generation of new keys. A typical quad-core 3GHz server generating a new short-term key every minute is using under 1/100000 of its CPU time on key generation with our current software.

[3] For example, "fast gcd" techniques incorporate subquadratic-time multiplication methods such as Karatsuba's method, and are compatible with constant-time computations.

- A user who (for some reason) wants to generate many keys more quickly than this can use Montgomery's trick to batch the inversions. Montgomery's trick replaces (e.g.) 1000 inversions with 2997 multiplications and just 1 inversion. This reduces the cost of generating each key below 300000 cycles.

Our software is analogous to the original Curve25519 software [10], which emphasized encryption/decryption speed and did not bother speeding up occasional key-generation computations.

`ntrulpr4591761` is estimated to be somewhat slower than `sntrup4591761`, although it is faster than `sntrup4591761` for key generation.

5.3 Space

Public keys for `sntrup4591761` occupy 1218 bytes. Ciphertexts occupy only 1047 bytes. Secret keys occupy 1600 bytes.

Public keys for `ntrulpr4591761` occupy 1047 bytes. Ciphertexts occupy 1175 bytes. Secret keys occupy 1238 bytes.

5.4 How parameters affect performance

Encapsulation and decapsulation involve a few multiplications in the ring $\mathcal{R}/q$. The asymptotic cost of multiplication, as p and q grow, is essentially linear in $p \log_2 q$, the number of bits in a ring element. Other operations scale at least as well as this.

6 Analysis of known attacks (2.B.5)

We start with existing *pre-quantum* NTRU attack strategies, adapt those strategies to the context of Streamlined NTRU Prime, and quantify their effectiveness. In particular, we account for the impact of changing $x^p - 1$ to $x^p - x - 1$, and using small f rather than $f = 1+3F$ with small F. For comparability we assume here that the weight w in Streamlined NTRU Prime is taken as $2t$, where t is the number of 1's and the number of -1's in the original NTRU cryptosystem.

We consider NTRU LPRime in Section 6.8. We consider post-quantum security in Section 6.6.

6.1 Warning: underestimates are dangerous

Underestimating attack cost can *damage* security, for reasons explained in [11, full version, Appendix B.1.2], so we prefer to use accurate cost estimates. However, accurately evaluating

the cost of lattice attacks is generally quite difficult. The literature very often explicitly resorts to underestimates. Comprehensively fixing this problem is beyond the scope of this submission, but we have started work in this direction, as illustrated by Section 6.7. At the same time it is clear that the best attack algorithms known today are much better than the best attack algorithms known a few years ago, so it is unreasonable to expect that the algorithms have stabilized. We plan to periodically issue updated security estimates to reflect ongoing work.

6.2 Meet-in-the-middle attack

Odlyzko's meet-in-the-middle attack [36, 34] on NTRU works by splitting the space of possible keys $\mathcal{F}$ into two parts such that $\mathcal{F} = \mathcal{F}_1 \oplus \mathcal{F}_2$. Then in each loop of the algorithm partial keys are drawn from $\mathcal{F}_1$ and $\mathcal{F}_2$ until a collision function (defined in terms of the public key h) indicates that $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$ have been found such that $f = f_1 + f_2$ is the private key.

The number of choices for f is $\binom{p}{t}\binom{p-t}{t}$ in original NTRU and $\binom{p}{2t}2^{2t}$ in Streamlined NTRU Prime. A first estimate is that the number of loops in the algorithm is the square root of the number of choices of f. However, this estimate does not account for equivalent keys. In NTRU Classic, a key (f, g) is equivalent to all of the rotated keys $(x^i f, x^i g)$ and to the negations $(-x^i f, -x^i g)$, and the algorithm succeeds if it finds any of these rotated keys. The $2p$ rotations and negations are almost always distinct, producing a speedup factor very close to $\sqrt{2p}$.

The structure of the NTRU Prime ring is less friendly to this attack. Say f has degree $p-c$; typically c is around $p/2t$, since there are $2t$ terms in f. Multiplying f by $x, x^2, \ldots, x^{c-1}$ produces elements of $\mathcal{F}$, but multiplying f by x^c replaces x^{p-c} with x^p mod $x^p - x - 1 = x + 1$, changing its weight and thus leaving $\mathcal{F}$. It is possible but rare for subsequent multiplications by x to reenter $\mathcal{F}$. Similarly, one expects only about $p/2t$ divisions by x to stay within $\mathcal{F}$, for a total of only about p/t equivalent keys, or $2p/t$ when negations are taken into account. We have confirmed these estimates with experiments.

One could modify the attack to use a larger set $\mathcal{F}$, but this seems to lose more than it gains. Furthermore, similar wraparounds for g compromise the effectiveness of the collision function. To summarize, the extra term in $x^p - x - 1$ seems to increase the attack cost by a factor around $\sqrt{t}$, compared to NTRU Classic; i.e., the rotation speedup is only around $\sqrt{2p/t}$ rather than $\sqrt{2p}$.

On the other hand, some keys f allow considerably more rotations. We have decided to assume a speedup factor of $\sqrt{2(p-t)}$, since we designed some pathological polynomials f with that many (not consecutive) rotations in the set. For random r the speedup is much smaller. This means that the number of loops before this attack is expected to find f is bounded by

$$L = \sqrt{\binom{p}{2t}2^{2t}} \Big/ \sqrt{2(p-t)}. \tag{1}$$

In each loop, t vectors of size p are added and their coefficients are reduced modulo q. We thus estimate the attack cost as Lpt. The storage requirement of the attack is approximately $L \log_2 L$. We can reduce this storage by applying collision search to the meet-in-the-middle attack (see [48, 62]). In this case we can reduce the storage capacity by a factor s at the expense of increasing the running time by a factor $\sqrt{s}$.

6.3 Streamlined NTRU Prime lattice

As with NTRU Classic, we can embed the problem of recovering the private keys f, g into a lattice problem. Saying $3h = g/f$ in $\mathcal{R}/q$ is the same as saying $3hf + qk = g$ in $\mathcal{R}$ for some polynomial k; in other words, there is a vector (k, f) of length $2p$ such that

$$\begin{pmatrix} k & f \end{pmatrix} \begin{pmatrix} qI & 0 \\ H & I \end{pmatrix} = \begin{pmatrix} k & f \end{pmatrix} B = \begin{pmatrix} g & f \end{pmatrix},$$

where H is a matrix with the i'th vector corresponding to $x^i \cdot 3h \bmod x^p - x - 1$ and I is the $p \times p$ identity matrix. We will call B the *Streamlined NTRU Prime public lattice basis*. This lattice has determinant q^p. The vector (g, f) has norm at most $\sqrt{2p}$. The Gaussian heuristic states that the length of the shortest vector in a random lattice is approximately $\det(B)^{1/(2p)}\sqrt{\pi ep} = \sqrt{\pi epq}$, which is much larger than $\sqrt{2p}$, so we expect (g, f) to be the shortest nonzero vector in the lattice.

Finding the secret keys is thus equivalent to solving the Shortest Vector Problem (SVP) for the Streamlined NTRU Prime public lattice basis. The fastest currently known method to solve SVP in the NTRU public lattice is the hybrid attack, which we discuss below.

A similar lattice can be constructed to instead try to find the input pair (m, r). However, there is no reason to expect the attack against (m, r) to be easier than the attack against (g, f): r has the same range as f, and m has essentially the same range as g. Recall that Streamlined NTRU Prime does not have the NTRU Classic problem of leaking $m(1)$. There are occasional boundary constraints on m (see Section 4.3), and there is also an $\mathcal{R}/3$ invertibility constraint on g, but these effects are minor.

6.4 Hybrid security

The best known attack against the NTRU lattice is the hybrid lattice-basis-reduction-and-meet-in-the-middle attack described in [34]. The attack works in two phases: the reduction phase and the meet-in-the-middle phase.

Applying lattice-basis-reduction techniques will mostly reduce the middle vectors of the basis [56]. Therefore the strategy of the reduction phase is to apply lattice-basis reduction, for example BKZ 2.0 [21], to a submatrix B' of the public basis B. We then get a reduced basis $T = UBY$:

$$\left(\begin{array}{c|c|c} I_u & 0 & 0 \\ \hline 0 & U' & 0 \\ \hline 0 & 0 & I_{u'} \end{array}\right) \cdot \left(\begin{array}{c|c|c} qI_u & 0 & 0 \\ \hline * & B' & 0 \\ \hline * & * & I_{u'} \end{array}\right) \cdot \left(\begin{array}{c|c|c} I_u & 0 & 0 \\ \hline 0 & Y' & 0 \\ \hline 0 & 0 & I_{u'} \end{array}\right) = \left(\begin{array}{c|c|c} qI_u & 0 & 0 \\ \hline * & T' & 0 \\ \hline * & * & I_{u'} \end{array}\right)$$

Here Y is orthonormal and T' is again in lower triangular form.

In the meet-in-the-middle phase we can use a meet-in-the-middle algorithm to guess options for the last u' coordinates of the key by guessing halves of the key and looking for collisions. If the lattice basis was reduced sufficiently in the first phase, a collision resulting in the private key will be found by applying a rounding algorithm to the half-key guesses. More details on how to do this can be found in [34].

To estimate the security against this attack we adapt the analysis of [32] to the set of keys that we use in Streamlined NTRU Prime. Let u be the dimension of I_u and u' be the dimension of $I_{u'}$. For a sufficiently reduced basis the meet-in-the-middle phase will require on average

$$-\frac{1}{2}\left(\log_2(2(p-t)) + \sum_{0 \le a \le \min\{2t,u'\}} 2^a \binom{u'}{a} v(a) \log_2(v(a))\right) \tag{2}$$

work, where the $\log_2(2(p-t))$ term accounts for equivalent keys and

$$v(a) = \frac{2^{2t-a}\binom{p-u'}{2t-a}}{2^{2t}\binom{p}{2t}} = \frac{2^{-a}\binom{p-u'}{2t-a}}{\binom{p}{2t}}. \tag{3}$$

The quality of a basis after lattice reduction can be measured by the Hermite factor $\delta = ||\mathbf{b_1}||/\det(B)^{1/p}$. Here $||\mathbf{b_1}||$ is the length of the shortest vector among the rows of B. To be able to recover the key in the meet-in-the-middle phase, the $(2p-u-u') \times (2p-u-u')$ matrix T' has to be sufficiently reduced. For given u and u' this is the case if the lattice reduction reaches the required value of δ. This Hermite factor has to satisfy

$$\log_2(\delta) \le \frac{(p-u)\log_2(q)}{(2p-(u+u'))^2} - \frac{1}{2p-(u'+u)}. \tag{4}$$

We use the BKZ 2.0 simulator of [21] to determine the best BKZ 2.0 parameters, specifically the "block size" β and the number of "rounds" n, needed to reach a root Hermite factor δ. To get a concrete security estimate of the work required to perform BKZ-2.0 with parameters β and n we use the conservative formula determined by [32] from the experiments of [22]:

$$\text{Estimate}(\beta, p, n) = 0.000784314\beta^2 + 0.366078\beta - 6.125 + \log_2(p \cdot n) + 7. \tag{5}$$

This estimate and the underlying experiments rely on "enumeration"; see Section 6.7 for a comparison to "sieving". This analysis also assumes that the probabily of two halves of the key colliding is 1. We will also conservatively assume this, but a more realistic estimate can be found in [63]. Using these estimates we can determine the optimal u and u' to attack a parameter set and thereby estimate its security.

Lastly we note that this analysis is easily adaptable to generalizing the coefficients to be in the set $\{-d, -(d-1), \ldots, d-1, d\}$ by replacing base 2 in the exponentiations in Equations 1, 2 and 3 with $2d$. In this case however the range of t, by a generalization of Theorem 2, decreases to $q > 16(d^3 + d^2)t$.

6.5 Algebraic attacks

The attack strategy of Ding [26], Arora–Ge [6], and Albrecht–Cid–Faugère–Fitzpatrick–Perret [2] takes subexponential time to break dimension-n LWE with noise width $o(\sqrt{n})$, and polynomial time to break LWE with constant noise width. However, these attacks require many LWE samples, whereas typical cryptosystems in the NTRU family provide far less data to the attacker. When these attacks are adapted to cryptosystems that provide only (say) $2n$ samples, they end up taking more than $2^{0.5n}$ time, even when the noise is limited to $\{0, 1\}$. See generally [2, Theorem 7] and [43, Case Study 1].

6.6 Quantum attacks

Grover's algorithm, amplitude amplification, and quantum walks produce better exponents for some of the subroutines used above. Preliminary estimates indicate that the overall impact on Streamlined NTRU Prime security levels is much less than the impact upon AES-256 security levels. Further analysis is required.

6.7 Memory, parallelization, and sieving algorithms

The security estimates above rely on enumeration algorithms [52, 29, 38, 32]. For very large dimensions, the performance of enumeration algorithms is slightly super-exponential and is known to be suboptimal. The provable sieving algorithms of Pujol and Stehlé [54] solve dimension-β SVP in time $2^{2.465\ldots\beta+o(\beta)}$ and space $2^{1.233\ldots\beta+o(\beta)}$, and more recent SVP algorithms [1] take time $2^{\beta+o(\beta)}$. More importantly, under heuristic assumptions, sieving is much faster. The most recent work on lattice sieving (see [9, 42]) has pushed the heuristic complexity down to $2^{0.292\ldots\beta+o(\beta)}$.

Simply comparing 0.292β to enumeration exponents suggests that sieving could be faster than enumeration for sizes of β of relevance to cryptography. However, this comparison ignores two critical caveats regarding the performance of sieving. First, a closer look at polynomial factors indicates that the $o(\beta)$ here is positive. Consider, e.g., [9, Figure 3], which reports a best fit of $2^{0.387\beta-15}$ for its fastest sieving experiments. The comparison in [47] takes this caveat into account and concludes that the sieving cutoff is "far out of reach".

Second, sieving uses much more storage as β grows: at least $2^{0.208\ldots\beta+o(\beta)}$ bits of storage, again with positive $o(\beta)$. It is not known how to reduce the storage without large increases in the number of operations. Furthermore, sieving is bottlenecked by random access to

storage, and this random access also becomes slower as the amount of storage increases. The slowdown is approximately the square root of the storage in realistic cost models; see, e.g., [16].

Enumeration fits into very little memory even for large β. Kuo, Schneider, Dagdelen, Reichelt, Buchmann, Cheng, and Yang [41] showed that enumeration parallelizes effectively within and across GPUs. An attacker who can afford enough hardware for sieving for large β can instead use the same amount of hardware for enumeration, obtaining an almost linear parallelization speedup.

We do not mean to suggest that the operation-count ratio should be multiplied by the sieving storage (accounting for this enumeration speedup) *and* further by the square root of the storage (accounting for the cost of random access inside sieving): this would ignore the possibility of a speedup from parallelizing sieving. "Mesh" sorting algorithms such as the Schnorr–Shamir algorithm [57] sort n small items in time just $O(\sqrt{n})$, which is optimal in realistic models of parallel computation; these algorithms can be used as subroutines inside sieving, reducing the asymptotic cost penalty to just $2^{0.104...\beta+o(\beta)}$. However, this is still much less effective parallelization than [41].

This cost penalty for sieving is ignored in measurements such as [45] and [9, Figure 3], and in the resulting comparisons such as [47]. These measurements are limited to sieving sizes that fit into DRAM on a single computer, and do not account for the important increase in memory cost as β increases. Another way to see the same issue would be to scale sieving *down* to a small enough size to fit into GPU multiprocessors; this would demonstrate a sieving speedup for smaller β, for fundamentally the same reason that there will be a sieving slowdown for larger β.

In the absence of any realistic analyses of sieving cost for large β, we have decided to omit sieving from our security estimates. There is very little reason to believe that sieving can beat enumeration inside any attack that fits within our security target.

6.8 Attacks against NTRU LPRime

NTRU LPRime is similar to Streamlined NTRU Prime from an attack perspective. In particular, a lattice attack that finds small (m, r), given a random h and given $c = m + hr$, breaks both Streamlined NTRU Prime and NTRU LPRime.

Above we focused on the similar problem of finding small (g, f) given h and given $0 = g - 3hf$. Having to consider this second problem is a complication avoided by NTRU LPRime, and if this problem is easier then NTRU LPRime could be more secure than Streamlined NTRU Prime.

On the other hand, NTRU LPRime has its own complication, namely that it also releases an approximation to a second multiple of r. If this makes the r-recovery problem easier then NTRU LPRime could be less secure than Streamlined NTRU Prime. As noted above, we find both of these complications worrisome; further security analysis is required.

A simpler issue is that, for the same sizes p and q, NTRU LPRime places slightly smaller limits on the weight w than Streamlined NTRU Prime does. A smaller weight reduces the quantitative security level against various attack strategies discussed above.

7 Expected strength (2.B.4) in general

7.1 Security definitions

Our security goal is IND-CCA2. See Section 8 for quantitative estimates of the security of specific parameter sets.

Our general strategy for handling multi-target attacks is to aim for a very high single-target security level, and then rely on the fact that T-target attacks gain at most a factor T. We have not introduced complications aimed at making multi-target attacks even more difficult.

Our current software allows multiple encodings of ciphertexts and keys, for example allowing 4591 as a synonym for 0. NIST has stated a preference for implementations that enforce unique encodings, and we plan to adjust our software accordingly.

Lattice-based encryption also has various symmetries, analogous to well-known ECC symmetries. For example, if c (plus confirmation) is a Streamlined NTRU Prime ciphertext under public key h, then $-c$ (plus the same confirmation) is a Streamlined NTRU Prime ciphertext for the same session key under public key $-h$.

7.2 Rationale

See Section 6 for an analysis of known attacks.

Algorithm 1 searches for (p, q, t, λ), where λ is Section 6's estimate of the *pre-quantum* security level for parameters (p, q, t) with $w = 2t$. The subroutine nextprime(i) returns the first prime number $>i$. The subroutine viableqs(p, q_b) returns all primes q larger than p and smaller than q_b for which it holds that $x^p - x - 1$ is irreducible in $(\mathbb{Z}/q)[x]$. The subroutine mitmcosts uses the estimates from Equation (1) to determine the bitsecurity level of the parameters against a straightforward meet-in-the-middle attack. To find u, u', β, n we set u to the hybridbkzcost of the previous iteration (initially 0) and do a binary search for u' such that the two phases of the hybrid attack are of equal cost. For each u' we determine the Hermite factor required with Equation (4), use the BKZ-2.0 simulator to determine the optimal β and n to reach the required Hermite factor and use Equations (5) and (2) to determine the hybridbkzcost and hybridmitmcost.

Note that this algorithm outputs the largest value of t such that there are no decryption failures according to Theorem 2 and that no more than $2/3$ of the coefficients of f are set. Experiments show that decreasing t to t_1 linearly decreases the security level by approximately $t - t_1$.

Algorithm 1: Determine parameter sets for security level above ℓ.

Input: Upper bound q_b for q, range $[p_1, p_2]$ for p, lower bound ℓ for security level
Result: Viable parameters p, q and t with security level λ.
$p \leftarrow p_1 - 1$ (the prime we are currently investigating)
while $p \leq p_2$ **do**
  $p \leftarrow \text{nextprime}(p)$
  $Q \leftarrow \text{viableqs}(p, q_b)$
  for $q \in Q$ **do**
    $t \leftarrow \min\{\lfloor (q-1)/32 \rfloor, \lfloor p/3 \rfloor\}$
    $\lambda_1 \leftarrow \text{mitmcosts}(p, t)$
    if $\lambda_1 \geq \ell$ **then**
      Find u, u', β, n such that BKZ-2.0 costs are approximately equal to meet-in-the-middle costs in the hybrid attack.
      $\lambda_2 \leftarrow \max\{\text{hybridbkzcost}, \text{hybridmitmcost}\}$
      return $p, q, t, \min\{\lambda_1, \lambda_2\}$

See the NTRU Prime paper for a table of Streamlined NTRU Prime parameter sets with $465 < p < 970$ and $q < 20000$. Our recommended parameters $(p, q, w) = (761, 4591, 286)$ with estimated pre-quantum security 2^{248} provide an excellent tradeoff between size and security level.

The analysis of NTRU LPRime parameter sets works the same way and gives 2^{225} for our recommended parameters $(p, q, w) = (761, 4591, 250)$.

8 Expected strength (2.B.4) for each parameter set

8.1 Parameter set `kem/sntrup4591761`

Category 5. 2^{248} is marginally smaller than 2^{256}, but we expect that further analysis along the lines of [63], together with analysis of memory and communication costs, will show that this parameter set is much more expensive to break than AES-256.

8.2 Parameter set `kem/ntrulpr4591761`

Category 5. 2^{225} is noticeably smaller than 2^{256}, due to the smaller polynomial weight compared to `sntrup4591761`. However, note that the analysis by Wunderer [63] showed an underestimate of security by a factor of at least 2^{30} for a previous set of parameters analyzed by the methodology above; memory and communication costs are likely to be on an even larger scale. Hence, we expect that further analysis will show that this parameter set is harder to break than AES-256. See also Section 6.8.

9 Advantages and limitations (2.B.6)

There are several proposals of lattice-based cryptosystems that appear to provide high security with keys and ciphertexts fitting into just a few kilobytes. This proposal is designed to have the smallest attack surface, minimizing the number of avenues available to cryptanalysts. Some recent attacks against lattice-based cryptosystems rely on homomorphisms eliminated by this proposal.

At the same time this proposal provides unusually small sizes and excellent speed. One of the reasons for this performance is that this proposal provides the flexibility to target any desired lattice dimension rather precisely, without the "jumps" that appear in most proposals. Future advances in understanding the exact security level of lattice-based cryptography will allow this proposal to be tuned accordingly.

Beware, however, that there are other recent attacks against lattice-based cryptography, including impressive advances against SVP. As noted before, the security of lattice-based cryptography is not well understood. This is a general limitation of lattice-based cryptography. The same limitation is shared by many—but not all—post-quantum proposals.

References

[1] Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in 2^n time using discrete Gaussian sampling: Extended abstract. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 733–742. ACM, 2015. http://arxiv.org/abs/1412.7994.

[2] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *ACM Comm. Computer Algebra*, 49(2):62, 2015. https://eprint.iacr.org/2014/1018.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016. https://eprint.iacr.org/2015/1092.

[4] Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from LWE to LWR. *IACR Cryptology ePrint Archive*, 2016:589, 2016. https://eprint.iacr.org/2016/589.

[5] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited – new reduction, properties and applications. In Canetti and Garay [19], pages 57–74. https://eprint.iacr.org/2013/098.

[6] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011. https://users.cs.duke.edu/~rongge/LPSN.pdf.

[7] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737. Springer, 2012. https://eprint.iacr.org/2011/401.

[8] Jens Bauch, Daniel J. Bernstein, Henry de Valence, Tanja Lange, and Christine van Vredendaal. Short generators without quantum computers: The case of multiquadratics. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 27–59, 2017. https://multiquad.cr.yp.to.

[9] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Krauthgamer [40], pages 10–24. https://eprint.iacr.org/2015/1128.

[10] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. https://cr.yp.to/papers.html#curve25519.

[11] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2013. https://cr.yp.to/papers.html#nonuniform.

[12] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In Krauthgamer [40], pages 893–902. http://fangsong.info/files/pubs/BS_SODA16.pdf.

[13] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal

Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2016. https://eprint.iacr.org/2015/769.

[14] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In *ACM Conference on Computer and Communications Security*, pages 1006–1018. ACM, 2016. https://eprint.iacr.org/2016/659.

[15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society, 2015. https://eprint.iacr.org/2014/599.

[16] Richard P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, 1981. http://maths-people.anu.edu.au/~brent/pd/rpb055.pdf.

[17] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2016. https://eprint.iacr.org/2016/300.

[18] Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: a cautionary tale, 2014. http://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.

[19] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*. Springer, 2013.

[20] Hao Chen, Kristin Lauter, and Katherine E. Stange. Vulnerable Galois RLWE families and improved attacks. *IACR Cryptology ePrint Archive*, 2016. https://eprint.iacr.org/2016/193.

[21] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[22] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates (full version), 2011. http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf.

[23] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 559–585. Springer, 2016. https://eprint.iacr.org/2015/313.

[24] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to Ideal-SVP. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 324–348, 2017. https://eprint.iacr.org/2016/885.

[25] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2003. https://eprint.iacr.org/2002/174.

[26] Jintai Ding. Solving LWE problem with bounded errors in polynomial time. *IACR Cryptology ePrint Archive*, 2010:558, 2010. https://eprint.iacr.org/2010/558.

[27] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Canetti and Garay [19], pages 40–56. https://eprint.iacr.org/2013/383.

[28] Kirsten Eisenträger, Sean Hallgren, and Kristin E. Lauter. Weak instances of PLWE. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 2014. https://eprint.iacr.org/2014/784.

[29] Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777278-8/S0025-5718-1985-0777278-8.pdf.

[30] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on mdpc with cca security using decoding errors. Cryptology ePrint Archive, Report 2016/858, 2016. https://eprint.iacr.org/2016/858.

[31] Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, volume 5536 of *Lecture Notes in Computer Science*, pages 437–455, 2009.

[32] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing parameters for NTRUEncrypt. *IACR Cryptology ePrint Archive*, 2015. https://eprint.iacr.org/2015/708.

[33] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

[34] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 150–169. Springer, 2007. https://www.iacr.org/archive/crypto2007/46220150/46220150.pdf.

[35] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of NTRU encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 226–246. Springer, 2003. http://www.di.ens.fr/~pointche/Documents/Papers/2003_crypto.pdf.

[36] Nick Howgrave-Graham, Joseph H Silverman, and William Whyte. A meet-in-the-middle attack on an NTRU private key. Technical report, NTRU Cryptosystems, June 2003. Report, 2003. https://www.securityinnovation.com/uploads/Crypto/NTRUTech004v2.pdf.

[37] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3, 2005. https://eprint.iacr.org/2005/045.

[38] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206, New York, NY, USA, 1983. ACM.

[39] Paul Kirchner and Pierre-Alain Fouque. Comparison between subfield and straightforward attacks on NTRU. Cryptology ePrint Archive, Report 2016/717, 2016. https://eprint.iacr.org/2016/717.

[40] Robert Krauthgamer, editor. *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. SIAM, 2016.

[41] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes A. Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme enumeration on GPU and in clouds: How many dollars you need to break SVP challenges. In Bart Preneel and

Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2011. http://www.iis.sinica.edu.tw/papers/byyang/12158-F.pdf.

[42] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015. https://eprint.iacr.org/2014/744.pdf.

[43] Vadim Lyubashevsky. Future directions in lattice cryptography (talk slides), 2016. http://troll.iis.sinica.edu.tw/pkc16/slides/Invited_Talk_II--Directions_in_Practical_Lattice_Cryptography.pptx.

[44] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43, 2013. https://eprint.iacr.org/2012/230.

[45] Artur Mariano, Christian H. Bischof, and Thijs Laarhoven. Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*, pages 590–599. IEEE Computer Society, 2015. https://eprint.iacr.org/2015/041.

[46] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 733–748. USENIX Association, 2014. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer.

[47] Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 276–294. SIAM, 2015. https://eprint.iacr.org/2014/569.

[48] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999. http://people.scs.carleton.ca/~paulv/papers/JoC97.pdf.

[49] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 333–342. ACM, 2009. https://eprint.iacr.org/2008/481.

[50] Chris Peikert. "A useful fact about Ring-LWE that should be known better: it is *at least as hard* to break as NTRU, and likely strictly harder. 1/" (tweet), 2017. http://archive.is/B9KEW.

[51] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In *CCS*, pages 1843–1855. ACM, 2017. https://eprint.iacr.org/2017/490.

[52] Michael Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *SIGSAM Bull.*, 15(1):37–44, February 1981.

[53] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2013. https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf.

[54] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. *IACR Cryptology ePrint Archive*, 2009. https://eprint.iacr.org/2009/605.

[55] Halvor Sakshaug. Security analysis of the NTRUEncrypt public key encryption scheme, 2007. brage.bibsys.no/xmlui/bitstream/handle/11250/258846/426901_FULLTEXT01.pdf.

[56] Claus-Peter Schnorr. Lattice reduction by random sampling and birthday methods. In Helmut Alt and Michel Habib, editors, *STACS*, volume 2607 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2003. http://www.math.uni-frankfurt.de/~dmst/research/papers/schnorr.random_sampling.2003.ps.

[57] Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 255–263. ACM, 1986.

[58] Victor Shoup. A proposal for an ISO standard for public key encryption. *IACR Cryptology ePrint Archive*, 2001. https://eprint.iacr.org/2001/112.

[59] Victor Shoup. OAEP reconsidered. *J. Cryptology*, 15(4):223–249, 2002. https://eprint.iacr.org/2000/060.

[60] Martijn Stam. A key encapsulation mechanism for NTRU. In Nigel P. Smart, editor, *Cryptography and Coding, 10th IMA International Conference, Cirencester, UK, December 19-21, 2005, Proceedings*, volume 3796 of *Lecture Notes in Computer Science*, pages 410–427. Springer, 2005.

[61] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2011. https://www.iacr.org/archive/eurocrypt2011/66320027/66320027.pdf.

[62] Christine van Vredendaal. Reduced memory meet-in-the-middle attack against the NTRU private key. *LMS Journal of Computation and Mathematics*, 19(A):43–57, 001 2016. https://eprint.iacr.org/2016/177.

[63] Thomas Wunderer. Revisiting the hybrid attack: Improved analysis and refined security estimates, 2016. https://eprint.iacr.org/2016/733.

A Statements

These statements "must be mailed to Dustin Moody, Information Technology Laboratory, Attention: Post-Quantum Cryptographic Algorithm Submissions, 100 Bureau Drive – Stop 8930, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, or can be given to NIST at the first PQC Standardization Conference (see Section 5.C)."

First blank in submitter statement: full name. Second blank: full postal address. Third, fourth, and fifth blanks: name of cryptosystem. Sixth and seventh blanks: describe and enumerate or state "none" if applicable.

First blank in patent statement: full name. Second blank: full postal address. Third blank: enumerate. Fourth blank: name of cryptosystem.

First blank in implementor statement: full name. Second blank: full postal address. Third blank: full name of the owner.

A.1 Statement by Each Submitter

I, _____, of _____, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that (check one):

- *I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ OR (check one or both of the following):*
  - *to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ may be covered by the following U.S. and/or foreign patents:*

    _____
  - *I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations:*

    _____

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

A.2 Statement by Patent (and Patent Application) Owner(s)

If there are any patents (or patent applications) identified by the submitter, including those held by the submitter, the following statement must be signed by each and every owner, or each owner's authorized representative, of each patent and patent application identified.

I, _____, of _____, am the owner or authorized representative of the owner (print full name, if different than the signer) of the following patent(s) and/or patent application(s): _____

and do hereby commit and agree to grant to any interested party on a worldwide basis, if the cryptosystem known as _____ is selected for standardization, in consideration of its evaluation and selection by NIST, a non-exclusive license for the purpose of implementing the standard (check one):

- *without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination, OR*
- *under reasonable terms and conditions that are demonstrably free of any unfair discrimination.*

I further do hereby commit and agree to license such party on the same basis with respect to any other patent application or patent hereafter granted to me, or owned or controlled by me, that is or may be necessary for the purpose of implementing the standard.

I further do hereby commit and agree that I will include, in any documents transferring ownership of each patent and patent application, provisions to ensure that the commitments and assurances made by me are binding on the transferee and any future transferee.

I further do hereby commit and agree that these commitments and assurances are intended by me to be binding on successors-in-interest of each patent and patent application, regardless of whether such provisions are included in the relevant transfer documents.

I further do hereby grant to the U.S. Government, during the public review and the evaluation process, and during the lifetime of the standard, a nonexclusive, nontransferrable, irrevocable, paid-up worldwide license solely for the purpose of modifying my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability) for incorporation into the standard.

Signed:

Title:

Date:

Place:

A.3 Statement by Reference/Optimized Implementations' Owner(s)

The following must also be included:

I, _____, _____, am the owner or authorized representative of the owner _____ of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

The Picnic Signature Algorithm

Specification

Contact: Greg Zaverucha (gregz@microsoft.com)

October 31, 2017
Version 1.0

Contents

1 Introduction

This document specifies the Picnic public-key digital signature algorithm. It also describes cryptographic primitives used to construct Picnic, and methods for serializing signatures and public keys.

Picnic is designed to provide security against attacks by quantum computers, in addition to attacks by classical computers. The building blocks are a zero-knowledge proof system (with post-quantum security), and symmetric key primitives like hash functions and block ciphers, with well-understood post-quantum security. In particular, Picnic *does not* require number-theoretic, or structured hardness assumptions.

1.1 Overview of the Picnic Signature Algorithm

This section gives a very brief overview of the Picnic design. For a detailed description and a complete list of references to related work see [CDG+17], and the additional documentation submitted to the NIST Post-Quantum Standardization process. A reference implementation is available at `https://github.com/Microsoft/Picnic`.

The public key in Picnic is the pair (C, p) where $C = E_{sk}(p)$, and where E is a block cipher, sk a secret key and p is a plaintext block. The block cipher E is LowMC [ARS+16, ARS+15]. To create a signature, the signer creates a non-interactive proof of knowledge of sk, and binds the proof with the message to be signed. LowMC was chosen because the resulting signature size is smaller than alternative choices.

The proof of knowledge is a specialized version of ZKBoo [GMO16], called ZKB++. Informally, the prover simulates a multiparty computation protocol (MPC protocol) that allows players to jointly compute $E_{sk}(p)$, when each player has a share of sk. For Picnic, the number of players is always three. The idea is to have the prover commit to the simulated state and transcripts of all players, then have the verifier "corrupt" a random subset of the simulated players by seeing their complete state. The verifier then checks that the computation was done correctly from the perspective of the corrupted players, and if so, he has some assurance that the output is correct. The MPC protocol ensures that corrupting any two of the three players does not reveal information about the secret. Iterating this process multiple times in parallel gives the verifier high assurance that the prover knows the secret.

To make the proof non-interactive there are two options. The Fiat-Shamir transform (FS) yields a signature scheme that is secure in the random oracle model (ROM), whereas the Unruh transform (UR), yields a signature scheme that is secure in the quantum ROM (QROM). The UR signatures are larger, however.

1.2 Contributors

The Picnic signature algorithm was designed by the following team.
Melissa Chase, Microsoft
David Derler, Graz University of Technology
Steven Goldfeder, Princeton
Claudio Orlandi, Aarhus University
Sebastian Ramacher, Graz University of Technology
Christian Rechberger, Graz University of Technology & DTU
Daniel Slamanig, AIT Austrian Institute of Technology
Greg Zaverucha, Microsoft

2 Notation

This section describes the notation used in this document. In addition to the notation in Table 1, the notation `vec[0..2]` denotes a vector of three elements: `vec[0]`, `vec[1]`, `vec[2]`. When `vec` is used without an index it refers to the entire vector. All indexing is zero-based.

S	The expected security strength in bits (against classical attacks).
n	The LowMC blocksize, in bits.
k	The LowMC key size, in bits. This is also the signing key.
s	The LowMC number of s-boxes.
r	The LowMC number of rounds.
KDF	A key derivation function (defined in 3.3).
H	A hash function.
T	Number of parallel repetitions required for soundness of the proof of knowledge.
ℓ_H	The output length of H, in bytes.
$\oplus$	the binary exclusive or (XOR) of equal-length bitstrings.

Table 1: Notation used in this document.

3 Cryptographic Components

This section describes the cryptographic components that are used in the Picnic algorithm.

3.1 LowMC

Signing and verification compute the LowMC circuit, as part of then non-interactive MPC protocol simulation. The signing and verification algorithms specified here include sufficient detail to implement LowMC. However, implementations need some constants that are part of the LowMC definition. These parameters are different for each of the three LowMC parameter sets in Table 2.

`Kmatrix` an array of $n \times k$ binary matrices, one for initial whitening, and one for each LowMC round ($r + 1$ in total)

`Lmatrix` an array of $n \times n$ binary matrices, one for each LowMC round (r in total)

`roundconstant` an array of n-bit vectors, one for each LowMC round

We use the LowMC constants from the LowMC reference implementation [Tie17], without modification. These are included in the Picnic reference implementation, in the header file `lowmc_constants.c`.

3.2 Hash functions

The hash functions in this specification are all based on the SHAKE128 or SHAKE256 SHA-3 functions [NIS15] that have variable output length. In this document when we write H, this the SHAKE function given in Table 2 with the fixed output length also specified in Table 2.

There are multiple hashing operations when computing signatures, once to compute commitments, once to compute the challenge, (optionally) when computing a second type of commitment, and when using a seed value in multiple places. We prepend a fixed byte to the input of H in order to differentiate hash outputs in different uses. When computing commitments we use H_0 defined as $H_0(x) = H(\texttt{0x00}||x)$ and when computing the challenge we use H_1 defined as $H_1(x) = H(\texttt{0x01}||x)$. The UR parameter sets also use H when computing the function G (defined in Section 4.5.6), and here we use H_2 defined as $H_2(x) = H(\texttt{0x02}||x)$. Before each use of a seed value, used in multiple places, we hash it before use with H_2 in one instance and H_4, H_5 in the others, which prepend the bytes `0x02`, `0x04` and `0x05` respectively.

3.3 Key Derivation Functions

When creating and verifying signatures we must expand a short random value (128 to 512 bits) called the seed, into a longer one (about 1KB). This is done with a

extendable-output function (XOF), based on SHA3, called SHAKE [NIS15]. This choice allows a single function family (SHA3) for both hashing and key derivation, as SHAKE with a fixed output length is also a secure hash function. At security level 1 we use SHAKE128 and security levels 3 and 5 we use SHAKE256. In this specification all calls to the KDF specify the complete input as a bitstring, i.e., additional values such as the context, label and output length, must be encoded as described here, and passed to the XOF as a single input.

3.4 Views

Signing and verification must compute the views of three players in the MPC protocol simulation. An individual view object has three components

`view.iShare` The input key share of this player, k bits long.

`view.transcript` The transcript of all communication during the protocol. The length of this depends on the number of AND gates in the LowMC instance being used. In particular, the number of AND gates is $3rs$, so the length of the transcript is the number of bytes required to store $3rs$ bits.

`view.oShare` The output share of this player, k bits long.

Views must be serialized as the simple concatenation of the above three values when serialized to compute commitments. In the `UR` variants we also compute additional commitments with the function G. The input to G includes the input share only if the view is index 2 (corresponding to the third party) followed by the transcript, and not the output share.

4 The Picnic Signature Algorithm

This section describes the parameter sets for Picnic, and the three main operations key generation, signing and verifying.

4.1 Parameters

Table 2 gives parameters for three security levels L1, L3 and L5, as described in [oST16], corresponding to the security of AES-128, AES-192 and AES-256 (respectively). For each of the three security levels there are two possible signature algorithms, one based on the Fiat-Shamir transform (FS), and one based on the Unruh

(UR) transform. For discussion of the differences between these two variants, see [CDG+17].

All parameters are chosen such that they are expected to provide S bits of security against classical attacks, and at least $S/2$ bits of security against quantum attacks.

Parameter Set	S	n	k	s	r	Hash/KDF	Digest length	T
`picnic-L1-FS` `picnic-L1-UR`	128	128	128	10	20	SHAKE128	256	219
`picnic-L3-FS` `picnic-L3-UR`	192	192	192	10	30	SHAKE256	384	329
`picnic-L5-FS` `picnic-L5-UR`	256	256	256	10	38	SHAKE256	512	438

Table 2: Parameters by security level.

Parameter Set	Public key	Private key	Signature
`picnic-L1-FS`	32	16	34000
`picnic-L1-UR`	32	16	53929
`picnic-L3-FS`	48	24	76740
`picnic-L3-UR`	48	24	121813
`picnic-L5-FS`	64	32	132824
`picnic-L5-UR`	64	32	209474

Table 3: Key and signature sizes (in bytes) by security level. For the FS variants, the signature length varies based on the challenge, here we state the largest possible signature. On average the signature will be slightly less than this.

4.2 Key Generation

This section describes how to generate a signing key pair. The public key is denoted $pk = (C, p)$ and the secret key is denoted sk. The input to key generation is a security level (one of $S = 128$, 192 or 256). Note that for a key pair of security level S it is technically possible to use it with both signature algorithms defined at this level, e.g., a key pair created with the 128-bit parameter set may be used with both `picnic-L1-FS` and `picnic-L1-UR`. It is not recommended to use a key pair with multiple signature algorithms.

1. Choose a random n-bit string p, and a random k-bit string sk.

2. Using LowMC with the parameters given in Table 2, compute the encryption of p with sk, $C = E(sk, p)$.

3. Output: The pair (sk, pk). The secret key is sk, and the public key pk is (C, p).

4.3 Signing Operation

The functions `matrix_mul`, `mpc_sbox`, `mpc_xor`, `mpc_and` and `H3` used to specify sign are specified in later sections (Sections 4.5.4, 4.5.1, 4.5.3, 4.5.2 and 4.5.5 resp.). The description of signature generation is independent of the security level, but changes for the signature algorithms using the Unruh transform: `picnic-L1-UR`, `picnic-L3-UR` and `picnic-L5-UR`. The description below is with respect to a fixed security parameter, and the flag UR indicates whether the Unruh transform is used.

Input: Signer's key pair (sk, pk), a message to be signed the byte array M, such that $1 \leq |M| \leq 2^{55}$.

Output: Signature on M as a byte array.

1. Initialize a list of triples of views `views[0..T-1][0..2]`, a list of commitments `C[0..T-1][0..2]` (byte arrays, each of length ℓ_H), and a list of seeds `seeds[0..T-1][0..2]`. If UR is set, initialize a list of commitments `G[0..T-1][0..2]` (byte arrays of variable length, not exceeding the length of a view, including the input share. See Step 3d below.).

2. Populate `seeds` with $3T$ random seeds, each of length S bits. It is recommended that these be derived deterministically, by calling the KDF in Table 2, with input

$$sk \| M \| pk \| S$$

where S is encoded as a 16-bit little endian integer. The number of bytes requested is $3T(S/8)$ (three seeds for each of T iterations, each of size $S/8$ bytes).

The test vectors associated with this document will use this method to simplify testing. However, the specific method of generating `seed` does not affect interoperability, and implementations may differ (e.g., by choosing the seeds uniformly at random, using an alternative derivation method, or including alternative inputs to derivation). For implementations seeking to randomize the signature function, it is recommended to use the derivation described here, but to append a $2S$ bit random value to the KDF input.

3. For each parallel iteration t from 0 to $T-1$:

   (a) Create three random tapes, denoted `rand[0..2]`, using the KDF specified in Table 2, and the input seeds from Step 2. The seed is hashed with H_2 then the digest and the output length are concatenated and input to the KDF. The output length is encoded as a 16-bit little-endian integer. Tape `rand[0]` and `rand[1]` have length $k + 3rs$ bits, and tape `rand[2]` has length $3rs$ bits. We use the notation `rand[i].nextBit()` to read the next bit of the tape.

   (b) Compute three shares of sk, denoted `x[0..2]`, each of length k bits:

      i. `x[0]` = first k bits of tape `rand[1]`

      ii. `x[1]` = first k bits of tape `rand[2]`

      iii. `x[2]` = $sk \oplus$ `x[0]` $\oplus$ `x[1]`

   (c) Simulate the MPC protocol to compute the LowMC encrypt circuit, recording the views of the three players. Let `state[0..2]`, be a triple of n-bit vectors.

      i. Compute the initial key shares, and whitening:
      `key = matrix_mul(x, Kmatrix[0])`

      ii. XOR the round key with p, the plaintext portion of the public key (C, p). For `i` from 0 to 2:
      `state = mpc_xor_constant(key, p)`

      iii. For each LowMC round `i` from 1 to r

         A. Compute the round `i` key shares:
         `key = matrix_mul(x, Kmatrix[i])`
         The function `matrix_mul` is defined in Section 4.5.4.

         B. Apply substitution layer (s-boxes) to `state`:
         `state = mpc_sbox(state, rand, views[t])`
         The function `mpc_sbox` is defined in Section 4.5.1.

         C. Apply affine layer to `state`:
         `state = matrix_mul(state, Lmatrix[i-1])`

         D. Update the state with the XOR of the round constant and the state:
         `state = mpc_xor_constant(state, roundconstant[i-1])`
         The function `mpc_xor_constant` is defined in Section 4.5.3.

E. Update the state with the XOR of the round key and the state:
`state = mpc_xor(state, key)`

iv. Store the output shares in the views, for `i` from 0 to 2:
`views[t][i].oShare = state[i]`

(d) Form commitments `C[t][0..2]`. For `i` from 0 to 2:
`C[t][i] =` H_0`( `H_4`(seed[i]), view[i])`
If the flag *UR* is set, for `i` from 0 to 2, compute:
`G[t][i]=` G`(`H_4`(seed[i]), view[i])`
Note that G is length-preserving, and when $e_t = 0$, the length of `G[t][i]` is longer by n bits, since the view includes the input share in addition to the transcript.

4. Compute the challenge e, by hashing the output shares, commitments, the signer's public key pk and the message M.

```
e =  H3(
     view[0][0].oShare, view[0][1].oShare, view[0][2].oShare,
     ...
     view[t-1][0].oShare, view[t-1][1].oShare, view[t-1][2].oShare,
     C[0][0], C[0][1], C[0][2],
     ...
     C[t-1][0], C[t-1][1], C[t-1][2],
     [G[0][0], G[0][1], G[0][2],
     ...
     G[t-1][0], G[t-1][1], G[t-1][2],]
     pk, M)
```

The function `H3` is defined in Section 4.5.5, it is a hash function with output in $\{0,1,2\}^t$. The commitments `G[i][j]` must be included when the flag *UR* is set, and omitted otherwise. We write e as $(e_0, \ldots, e_{t-1})$ where $e_i \in \{0,1,2\}$.

5. For each round t from 0 to $T-1$, assemble the proof. For the challenge $e_t \in \{0,1,2\}$, compute $i = e_t + 2 \pmod 3$ and set
b_t `= C[t][`i`], [G[t][i]]`
Note that `G[t][i]` is only present if *UR* is set. Then,
if $e_t = 0$, set z_t to
`view[t][1].transcript, seed[t][0], seed[t][1]`
else if $e_t = 1$, set z_t to
`view[t][2].transcript, seed[t][1], seed[t][2], view[t][2].iShare`

else if $e_t = 2$, set z_t to
`view[t][0].transcript, seed[t][2], seed[t][0], view[t][2].iShare`

6. Serialize $(e, b_0, \ldots, b_{t-1}, z_0, \ldots, z_{t-1})$ as described in Section 5.1 and output it as the signature.

4.4 Verification Operation

This section describes the Verify operation, to verify a signature created by the Sign operation in Section 4.3. The functions `matrix_mul`, `mpc_sbox_verify`, `mpc_xor`, `mpc_and` and `H3` used to specify verify are specified in later sections (Sections 4.5.4, 4.5.1, 4.5.3, 4.5.2 and 4.5.5 resp.). As with signing, the steps below work for all security levels, and the flag *UR* is set for parameter sets using the Unruh transform.

Input: Signer's public key *pk*, a message as a byte array M, such that $1 \leq |M| \leq 2^{55}$, a signature σ (also a byte array).

Output: `valid` if σ is a signature of M with respect to *pk* or `invalid` if not.

1. Deserialize the signature σ to $(e, b_0, \ldots, b_{t-1}, z_0, \ldots, z_{t-1})$ as described in Section 5.2. If deserialization fails, reject the signature and output `invalid`. Write e as $(e_0, \ldots, e_{t-1})$ where $e_i \in \{0, 1, 2\}$.
2. Initialize lists to contain the three commitments `C[0..t-1][0..2]`, output shares `outputs[0..t-1][0..2]`, and extra commitments `G[0..t-1][0..2]` (if *UR* is set only), for each parallel iteration. These will be inputs to `H3`, verification will re-compute some of these values, and use some provided as part of the signature.
3. For each parallel iteration t from 0 to $T - 1$:
   (a) Initialize two views `view[0]` and `view[1]`, random tapes `rand[0]` and `rand[1]`, and key shares `x[0]` and `x[1]`.
   (b) For this step there are three cases, one for each challenge value, as in Step 5 of the Sign operation.
   If $e_t = 0$:
      i. Use the provided `seed[t][0]` to recompute the random tape `rand[0]`.
      ii. Use the provided `seed[t][2]` to recompute the random tape `rand[1]`.

iii. Set `view[0].iShare` and `x[0]` to the first k bytes of `rand[0]`.
iv. Set `view[1].iShare` and `x[1]` to the first k bytes of `rand[1]`.

If $e_t = 1$:

i. Use the provided `seed[t][1]` to recompute the random tape `rand[0]`.
ii. Use the provided `seed[t][2]` to recompute the random tape `rand[1]`.
iii. Set `view[0].iShare` and `x[0]` to the first k bytes of `rand[0]`.
iv. Set `view[1].iShare` and `x[1]` to the input share in z_t.

If $e_t = 2$:

i. Use the provided `seed[t][2]` to recompute the random tape `rand[0]`.
ii. Use the provided `seed[t][0]` to recompute the random tape `rand[1]`.
iii. Set `view[0].iShare` and `x[0]` to the input share in z_t.
iv. Set `view[1].iShare` and `x[1]` to the first k bytes of `rand[1]`.

(c) Simulate the MPC protocol to compute the LowMC encrypt circuit. This is similar to signing since the circuit is the same, but because we are only simulating two of the parties instead of all three, the MPC subroutines are slightly different.

i. Compute initial round keys `key[0]` and `key[1]`:
```
key = matrix_mul(x, Kmatrix[0])
```
The function `matrix_mul` is defined in Section 4.5.4.

ii. Initialize shares of the state `state[0]` and `state[1]` with p, the plaintext portion of the public key (C, p), and the key.
```
state = mpc_xor_constant_verify(key, p, e_t )
```

iii. For each LowMC round `i` from 1 to r

A. Compute the round i key shares
```
key = matrix_mul(x, Kmatrix[i])
```

B. Apply substitution layer (s-boxes) to `state`:
```
state = mpc_sbox_verify(state, rand, views[t])
```

C. Apply affine layer to `state`:
```
state = matrix_mul(state, Lmatrix[i-1])
```

D. Update the state with the XOR of the round constant and the state:

`state = mpc_xor_constant_verify(state, roundconstant[i-1],` e_t`)`

E. Update the state with the XOR of the round key and the state:
`state = mpc_xor(state, key)`

iv. Store the output shares in the views:
`view[0].oShare = state[0]`
`view[1].oShare = state[1]`

v. Update the list of commitments. Two commitments are recomputed based on the recomputed views, and the third is provided in the proof.
`C[t][`e_t`] =` H_0`( ` H_4`(seed[0]), view[0])`
`C[t][`$e_t + 1 \bmod 3$`] =` H_0`( ` H_4`(seed[1]), view[1])`
`C[t][`$e_t + 2 \bmod 3$`] =` c
where c is the commitment provided as part of the proof, the first element in b_t. If UR is set, additionally update `G` as follows:
`G[t][`e_t`] =` G`(`H_4`(seed[0]), view[0])`
`G[t][`$e_t + 1 \bmod 3$`] =` G`( ` H_4`(seed[1]), view[1])`
`G[t][`$e_t + 2 \bmod 3$`] =` c'
where c' is the commitment provided as part of the proof, the second element in b_t.

vi. Update the list of output shares
`outputs[t][`e_t`] = view[0].oShare`
`outputs[t][`$e_t + 1$`] = view[1].oShare`
`outputs[t][`$e_t + 2$`] = view[0].oShare` $\oplus$ `view[1].oShare` $\oplus$ C
where C is the ciphertext component of the public key (C, p), and the addition is done modulo 3 (as above).

(d) Recompute the challenge

e' = `H3(`
`outputs[0][0], outputs[0][1], outputs[0][2],`
`...`
`outputs[T-1][0], outputs[T-1][1], outputs[T-1][2],`
`C[0][0], C[0][1], C[0][2],`
`...`
`C[T-1][0], C[T-1][1], C[T-1][2],`
`[G[0][0], G[0][1], G[0][2],`
`...`
`G[T-1][0], G[T-1][1], G[T-1][2],]`
pk, M`)`

The commitments `G[i][j]` must be included when the flag *UR* is set, and omitted otherwise.

(e) If e and e' are equal, output `valid` and otherwise output `invalid`.

4.5 Supporting Functions

The Sign (§4.3) and Verify (§4.4) operations use similar functions to simulate the MPC protocol used in the proof of knowledge. This section describes these functions.

4.5.1 LowMC S-Box Layer: `mpc_sbox`, `mpc_sbox_verify`

This section describes how the internal LowMC state is updated in the s-box layer. The number of s-boxes is fixed per parameter set, see Table 2. The input is the three shares of the state, random tapes and views. The tapes and the views are input because the operations in the s-box layer use ANDs and so this function must update the transcript of the MPC protocol. This function also depends on the parameter r, defined in Table 2. The function `mpc_sbox` is used when signing, and verification uses `mpc_sbox_verify`, which has the same definition, but calls to `mpc_and` are replaced with calls to `mpc_and_verify`.

In the following pseudocode, indexing is *bitwise* and zero-based. The temporary variables are triples of bits `a[0..2]`, `b[0..2]` and `c[0..2]`, of each of the three input shares (`ab`, `bc` and `ca` have the same type).

Input: Shares of LowMC state `state`, random tapes `rand`, and `views` as defined in Section 4.3. The input `views` a triple of views, corresponding to one parallel round.
Output: The input variable `state` is modified in place
Pseudocode:

```
for i from 0 to (3*r - 1)
    for j from 0 to 2
        a[j] = state[j][n - 1 - i - 2]
        b[j] = state[j][n - 1 - i - 1]
        c[j] = state[j][n - 1 - i]

    ab = mpc_AND(a, b, rand, views)
    bc = mpc_AND(b, c, rand, views)
    ca = mpc_AND(c, a, rand, views)

    for j from 0 to 2
```

```
state[j][n - 1 - i - 2] = a[j] XOR bc[j]
state[j][n - 1 - i - 1] = a[j] XOR b[j] XOR ca[j]
state[j][n - 1 - i] = a[j] XOR b[j] XOR c[j] XOR ab[j]
```

4.5.2 MPC AND Operations: `mpc_and`, `mpc_and_verify`

These functions take secret shares of bits `a`, `b` and compute the binary AND `c = a AND b`, updating the transcript of the MPC protocol. The randomness is read from the pre-computed random tapes, also provided as input. For signing, `mpc_and` takes three inputs, and for verification, a simpler two-input version, `mpc_and_verify` is used. Note that in verification, one of the players' output shares is provided as input.

`mpc_and`
Input: random tapes `rand`, the triple of views for this parallel round `views`, and secret-shared inputs `a[0..2]`, `b[0..2]`
Output: secret shares `c[0..2] = a` AND `b`, updates to the transcripts in `views`
Pseudocode:

```
r[0] = rand[0].nextBit()
r[1] = rand[1].nextBit()
r[2] = rand[2].nextBit()

for i from 0 to 2
    c[i] = (a[i] AND b[(i + 1) % 3]) XOR
           (a[(i + 1) % 3] AND b[i]) XOR
           (a[i] AND b[i]) XOR
           r[i] XOR r[(i + 1) % 3]
    views[i].transcript.append(c[i])
return c
```

`mpc_and_verify`
Input: random tapes `rand`, the pair of views for this parallel round `views`, and secret-shared inputs `a[0..1]`, `b[0..1]`
Output: secret shares `c[0..1] = a` AND `b`, updates to the transcripts in `views`
Pseudocode:

```
r[0] = rand[0].nextBit()
r[1] = rand[1].nextBit()
```

```
c[0] = (a[0] AND b[1]) XOR (a[1] AND b[0]) XOR
       (a[0] AND b[0]) XOR r[0] XOR r[1]
views[0].transcript.append(c[0])

c[1] = views[1].transcript.nextBit()

return c
```

4.5.3 MPC XOR Operations: `mpc_xor`, `mpc_xor_constant`

This function takes secret-shared input bits a, b and computes the secret shares of $c = a \oplus b$. Unlike the AND operation, which requires communication between players, the XOR operation is done locally in the MPC protocol, and does not need to update the views.

Input: m bit vectors of length L: `a[0..m - 1][0..L - 1]` and `b[0..m - 1][0..L - 1]`
Output: XOR of the two inputs `c[0..2][0..L - 1]`
Pseudocode:

```
for i = 0 to m - 1
  c[i] = a[i] XOR b[i] // XOR of L-bit strings
```

Note that (i) m is always 3 during the Sign operation, and 2 during verify, and (ii) implementations may work on multiple bits simultaneously using the processor's XOR instruction on word size operands.

XOR with a constant When one of the operands is a public constant instead of a secret share vector, the constant is XORed with only one of the secret shares. When signing, in `mpc_xor_constant`, the first share is always XORed with the constant. When verifying, in `mpc_xor_constant_verify`, if the challenge $e_t = 0$ then we XOR the first secret share with the constant, and when $e_t = 2$ we XOR the second secret share with the constant. (This is because the state corresponding to the first player is in a different position depending on the challenge.)

4.5.4 Binary Vector-Matrix Multiplication: `matrix_mul`

This function computes a vector-matrix product, with elements in GF(2). For signing, three vectors x, y, and z in $\mathrm{GF}(2)^k$ are input along with a single matrix $M \in \mathrm{GF}(2)^{k \times k}$, and three vectors xM, yM and zM in $\mathrm{GF}(2)^k$ are output. For signature

verification, only x and y are input, and xM and yM are output. The pseudocode below is modified for verification by omitting lines depending on z.

The function parity(v) is the usual parity function: on input a vector v, of length k, it returns 1 if the number of 1 bits in v is odd, and zero otherwise. It can be implemented as $v_0 \oplus v_1 \oplus ... \oplus v_{k-1}$.

Let `x[i]` denote the i-th bit of x, and `M[i][j]` denote the bit in the i-th row and j-th column of M.

Input: three k-bit vectors x, y, z, a k-bit by k-bit matrix M
Output: three k-bit vectors $a = xM$, $b = yM$ and $c = zM$
Pseudocode:

```
tempA, tempB, tempC are k-bit vectors
for i = 0 to k - 1
    for j = 0 to k - 1
        tempA[j] = x[j] AND M[i][j]
        tempB[j] = y[j] AND M[i][j]
        tempC[j] = z[j] AND M[i][j]
   a[k - 1 - i] = parity(tempA)
   b[k - 1 - i] = parity(tempB)
   c[k - 1 - i] = parity(tempC)
Output (a,b,c)
```

Notes

1. If inputs and outputs may overlap (e.g., when computing $x = xM$) a temporary variable is required for the output.

2. There are many ways to compute this function, implementations may use an alternative algorithm for better efficiency. For example, see [Alb17].

4.5.5 Computing the Challenge: `H3`

The function `H3` hashes an arbitrary length bitstring to a length T output in $\{0, 1, 2\}$ (i.e., `H3` : $\{0,1\}^* \rightarrow \{0,1,2\}^t$). The hash function H is called on the input, then iterated as required, to compute an output of length T.

In the pseudocode below, the hash function H is given in Table 2, along with the value for the parameter T. Recall that H_1 is defined as $H_1(x) = H(0x01||x)$.

Input: bitstring b
Output: vector e, of integers in $\{0, 1, 2\}$
Pseudocode:

1. Compute $h = H_1(b)$, write h in binary as $(h_0, h_1, ..., h_S)$.
2. Iterate over pairs of bits $(h_0, h_1), (h_2, h_3), \ldots$. If the pair is

   $(0, 0)$, append 0 to e,

   $(0, 1)$, append 1 to e,

   $(1, 0)$, append 2 to e,

   $(1, 1)$, do nothing.

   If e has length T, return.
3. If all pairs are consumed and e still has fewer than T elements, set $h = H(h)$ and return to Step 2.

4.5.6 Function G

The function G has two inputs: a *seed* of length S bits, and a view, v, of varying length. The output has length ℓ_G, computed as the sum of the length of the seed and the length of the view. Recall that not all views are equal length, ℓ_G differs depending on which player computed the view. G is implemented with the KDF from Table 2, namely, with SHAKE and the following input:

$$H_5(seed) \| v \| \ell_G$$

The integer ℓ_G is encoded as a 16-bit little endian integer.

5 Serialization

In this section we specify how to serialize and deserialize Picnic keys and signatures.

5.1 Serialization of Signatures

This section specifies how to serialize signatures created in Section 4.3.

This is a binary, fixed-length encoding, designed to minimize the space required by the signature. The components of the signature (views, seeds, commitments, etc.)

are all of fixed length for a given parameter set. The Fiat-Shamir parameter sets have signatures that vary in size, depending on the challenge; note that in Step 5, an additional input share is output when the challenge is 1 or 2. The serialization does not include an identifier indicating the parameter set, as not all applications require it.

Input: The signature $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 4.3, Step 5.

Output: A byte array B, encoding the signature.

1. Write the challenge to B, using $2T$ bits, padding with zero bits on the right to the nearest byte.
2. For each t from 0 to $T-1$, append (b_t, z_t) as follows. For values that do not use an even number of bytes, pad with zero bits to the next byte.
   (a) Append b_t, a commitment of length ℓ_H, and if the *UR* flag is set, also append the second commitment (denoted `G[t][i]` in Step 3d of signing).
   (b) Append z_i (in the order presented in Step 5 of Sign)
      i. Append the transcript.
      ii. Append the two seed values in z_t,
      iii. If e_t is 1 or 2, append the input share.
3. Output B.

5.2 Deserialization of Signatures

This section describes how to deserialize a byte array created by Section 5.1 to a signature for use in verification. The deserialization process reads the input bytes linearly. Since the signature length can vary depending on the challenge (encoded first in the byte array), it is recommended that implementations first compute the expected length from e, and reject the signature before parsing further, if B does not have the expected number of remaining bytes.

Input: A byte array B, encoding the signature.

Output: The signature $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 4.3, Step 5, or `null` if deserialization fails.

1. Read the first $(2T + 7)/8$ bytes from B. If the read fails, return `null`. Ensure that each pair of bits in the first $2T$ bits are in $\{0, 1, 2\}$ and return `null` if not. If padding bits are required for this value of T (see §5.1), ensure that all padding bits are zero, and return `null` if not. Assign these bytes to e. We use the notation $e = (e_0, \ldots, e_{T-1})$ to denote the individual pairs of bits.
2. For each t from 0 to $T - 1$, read (b_t, z_t) from B as follows. If any of the reads are not possible because B is too short, abort and return `null`.
   (a) Create b_t by reading a commitment of length ℓ_H from B. If *UR* is set, also read a second commitment from B, of length $3rs + n$ bits when $e_t == 0$ and $3rs$ bits otherwise.
   (b) Read z_t, as follows:
      i. Read the transcript from B, assign it to the first component of z_t. Recall that the length of the transcript is $3rs$ bits (where r and s are specified in Table 2).
      ii. Read the first seed value of length S bits from B, append it to z_t.
      iii. Read the second seed value of length S bits from B, append it to z_t.
      iv. If e_t is 1 or 2, read an input share of length S bits from B and append it to z_t.
3. Output $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$.

5.3 Serialization of Picnic Keys

A Picnic public key (C, p) should be serialized as the bits of C, followed by the bits of p. Both are first converted to byte arrays, are both S bits long, and S is guaranteed to be a multiple of eight. For a given parameter set, public keys can therefore be unambiguously parsed. Note that the length of a serialized public key uniquely identifies the security level, but not the exact parameter set, e.g., public keys for both `Picnic-L1-FS` and `Picnic-L1-UR` have the same length. Applications that handle multiple parameter sets are responsible for encoding the parameter set along with the public key.

Serializing the private key is done by serializing the S bits of sk, as a byte array. As with public keys, the length of the private key identifies the security level, but not the parameter set. Applications working with private keys for multiple parameter sets must also serialize the parameter set.

6 Additional Considerations

6.1 Signing Large Messages

Note that the sign operation makes two passes over M, once to generate the per-signature randomness, and once when computing the challenge. In applications where this cost is prohibitive, it is recommended to first hash M, and pass $H(M)$ to the signature algorithm specified here. The function H must be collision resistant, and the performance of picnic signatures is only weakly affected by the output length. Implementations that must pre-hash M should use SHAKE-256 with 512-bit digests, SHA3-512, or SHA-512.

A signing key used with pre-hashing must not be used without it, and vice-versa.

6.2 Test Vectors

The reference implementation[1] and the submission package for the NIST Post-Quantum Standardization process contain test vectors that implementations may use to verify conformance with this specification. The test vectors contain serialized versions of Picnic key pairs, messages and the corresponding Picnic signature. The intermediate values list the individual components of the signature, that should be produced after deserialization.

Note that key generation tests the correctness of an implementation's LowMC implementation, and in particular, that all of the constants required by LowMC are correct. In order to test the output of signing against a known value, implementations must use the de-randomized implementation specified here (§4.3, Step 2), where the per-signature ephemeral random values are derived from the signer's secret key and the message to be signed (as opposed to being randomly generated).

References

[Alb17] Martin Albrecht. m4ri: Further reading. m4ri Wiki, 2017. Accessed June 2017.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[1] Available online at https://github.com/Microsoft/Picnic.

[ARS+16] Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. Cryptology ePrint Archive, Report 2016/687, 2016.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. Cryptology ePrint Archive, Report 2017/279 and ACM CCS 2017, 2017.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, 2016.

[NIS15] NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology (NIST), FIPS PUB 202, U.S. Department of Commerce, 2015.

[oST16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December 2016. https://beta.csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[Tie17] Tyge Tiessen. LowMC reference implementation, September 2017. Available at https://github.com/LowMC/lowmc, HEAD was 3994bc857661ac33134b36163b131a215f0fe9c3 when constants were generated.

Submission to NIST's post-quantum project: lattice-based digital signature scheme qTESLA

Name of the cryptosystem: qTESLA

Principal and auxiliary submitters:

Nina Bindel, (Principal submitter)	Technische Universität Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Germany, Email: nbindel@cdc.informatik.tu-darmstadt.de, Phone: 004961511620667 Signature: 
Sedat Akleylek,	Ondokuz Mayis University, Turkey
Erdem Alkim,	Ege University, Turkey
Paulo S. L. M. Barreto,	University of Washington Tacoma, USA
Johannes Buchmann,	Technische Universität Darmstadt, Germany
Edward Eaton,	ISARA Corporation, Canada
Gus Gutoski,	ISARA Corporation, Canada
Juliane Krämer,	Technische Universität Darmstadt, Germany
Patrick Longa,	Microsoft Research, USA
Harun Polat,	Technische Universität Darmstadt, Germany
Jefferson E. Ricardini,	University of São Paulo, Brazil
Gustavo Zanon,	University of São Paulo, Brazil

Inventors of the cryptosystem:

All submitters by name based on a previous scheme by Shi Bai and Steven Galbraith and extensive previous works as explained in the body of this document.

Owners of the cryptosystem:

None (dedicate to the public domain)

Contents

1 Introduction

This document presents a detailed specification of qTESLA, a post-quantum signature scheme based on the hardness of the decisional ring learning with errors (R-LWE) problem. In contrast to other alternatives, qTESLA is a conservative yet efficient signature scheme that has been instantiated according to the provided security reduction. That is, qTESLA instantiations are *provably* secure in the (quantum) random oracle model. To this end, the scheme comes accompanied by a *non-tight* reduction in the random oracle model, and a *tight* reduction in the quantum random oracle model from R-LWE.

Concretely, qTESLA is designed to target *three* security levels:

- qTESLA-128: NIST's security category 1.
- qTESLA-192: NIST's security category 3.
- qTESLA-256: NIST's security category 5.

Despite the aforementioned security assurances in its parameter selection, qTESLA still achieves good performance with a competitive memory footprint. Furthermore, design decisions have been made towards enabling simple, easy-to-protect implementations.

In the remainder of this section, we describe previous works related to the proposed signature scheme qTESLA. In Section 2, we give the specification details of the scheme, including a basic and a formal algorithmic description, the functions that are required for its implementation, and the proposed parameter sets. In Section 3, we analyze the performance of our implementations. Section 4 includes the details of our known answer values. Then, we discuss the (provable) security of our proposal in Section 5, including an analysis of the concrete security level and the security against implementation attacks. Section 6 ends this document with a summary of the advantages and limitations of qTESLA.

1.1 Related work

The signature scheme proposed in this submission is the result of a long line of research. The first work in this line is the signature scheme proposed by Bai and Galbraith [14] which is based on the Fiat-Shamir construction of Lyubashevsky [50]. The scheme by Bai and Galbraith is constructed over standard lattices and comes with a (non-tight) security reduction from the learning with errors (LWE) and the short integer solution problem (SIS) in the random oracle model. Dagdelen *et al.* presented improvements and the first implementation of the Bai-Galbraith scheme [27]. The scheme was subsequently studied under the name TESLA by Alkim, Bindel, Buchmann, Dagdelen, Eaton, Gutoski, Krämer, and Pawlega [9], who provided an alternate security reduction from the LWE problem in the quantum random oracle model.

A variant of TESLA over ideal lattices was derived under the name ring-TESLA [1] by Akleylek, Bindel, Buchmann, Krämer, and Marson. Since then, subsequent works [16,41] have been presented. Most notably, a version of the scheme ring-TESLA called TESLA# [16] by Barreto, Longa, Naehrig, Ricardini, and Zanon included several implementation improvements. Moreover, there exist several works [19, 20, 36] concerned with the analysis of ring-TESLA with respect to implementation attacks, i.e., fault and side-channel attacks.

The signature scheme presented in the following assembles the advantages acquired in the prior works resulting in the quantum-secure signature scheme qTESLA.

Acknowledgments

We are grateful to Michael Naehrig for his valuable feedback.

SA and EA are partially supported by TÜBITAK under grant no. EEEAG-116E279.
NB is partially supported by the German Research Foundation (DFG) as part of project P1 within the CRC 1119 CROSSING.
JR is supported by the joint São Paulo Research Foundation (FAPESP)/Intel Research grant 2015/50520-6 "Efficient Post-Quantum Cryptography for Building Advanced Security Applications".

2 Specification

Next, we give an informal description of the basic scheme that is used to specify qTESLA. A formal specification of qTESLA's key generation, signing and verification algorithms then follows in Section 2.2. The correctness of the scheme is discussed in Section 2.3. We describe the implementation of the functions required by qTESLA in Section 2.4, and explain all the system parameters and the proposed parameter sets in Section 2.5.

2.1 Basic signature scheme

Informal descriptions of the algorithms that give rise to the signature scheme qTESLA are shown in Algorithms 1, 2 and 3. Below, we first define two basic terms that are required by the algorithms, namely, *B-short* and *well-rounded*.

An integer polynomial y is *B-short* if each coefficient is at most B in absolute value. We call an integer polynomial w *well-rounded* if w is $(\lfloor q/2 \rfloor - L_E)$-short and $[w]_L$ is $(2^d - L_E)$-short, where $[\cdot]_L$ is the value represented by the d least significant bits of w. Similarly,

Algorithm 1 Informal description of the key generation

Require: -
Ensure: Secret key $sk = (s, e, a)$, public key $pk = (a, t)$

1: $a \leftarrow \mathcal{R}_q$ invertible ring element
2: Choose $s, e \in \mathcal{R}$ with entries from $\mathcal{D}_\sigma$.
3: If the h largest entries of e sum to L_E then sample new e and retry at step 2.
4: If the h largest entries of s sum to L_S then sample new s and retry at step 2.
5: $t = as + e \in \mathcal{R}_q$.
6: Return secret key $sk = (s, e)$ and public key $pk = (a, t)$.

Algorithm 2 Informal description of the signature generation

Require: Message m, secret key $sk = (s, e, a)$,
Ensure: Signature (z, c).

1: Choose y uniformly at random among B-short polynomials in $\mathcal{R}_q$.
2: $c \leftarrow H(\lfloor ay \rceil_M, m)$.
3: $z \leftarrow y + sc$.
4: If z is not $(B - L_S)$-short then retry at step 1.
5: If $ay - ec$ is not well-rounded then retry at step 1.
6: Return signature (z, c).

Algorithm 3 Informal description of the verification

Require: Message m, public key $pk = (a, t)$, purported signature (z, c)
Ensure: "Accept" or "reject".

1: If z is not $(B - L_S)$-short then return reject.
2: $w \leftarrow az - tc \mod q$
3: If $H(\lfloor w \rceil_M, m) \neq c$ then return reject.
4: Return accept.

$\lfloor \cdot \rceil_M$ is the value represented by the corresponding most significant bits. For simplicity we assume that the hash oracle $H(\cdot)$ maps from $\{0,1\}^*$ to $\mathbb{H}$, where $\mathbb{H}$ denotes the set of polynomials $c \in \mathcal{R}$ with coefficients in $\{-1, 0, 1\}$ with exactly h nonzero entries, i.e., we ignore the encoding function F introduced in Section 2.2.

As can be seen, the description in Algorithm 2 implies that the signature scheme is non-deterministic, i.e., that different randomness is required for each signing operation, even if the message is the same. Specifically, this feature is fixed by the random generation of the polynomial y in Step 1 of Algorithm 2.

In Section 2.2, we discuss how the scheme can be converted to deterministic. Deterministic

signatures have the advantage that different randomness is used for different messages with very high probability and that sampling can be implemented more easily since access to a source of high-quality randomness is not needed. We discuss the (dis-)advantages of deterministic vs. probabilistic signatures in more detail in Section 5.4.

2.2 Formal description of qTESLA

Below, we define all the necessary functions, sets, and system parameters in qTESLA.

The description of the scheme depends on the following system parameters: λ, κ, n, q, σ, L_E, L_S, B, d, and h. Let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$, $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, $\mathcal{R}_{q,[I]} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in [-I, I]\}$, and $\mathbb{H}_{n,h} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in \{-1, 0, 1\},\ \sum_{i=0}^{n-1} |f_i| = h\}$. Let $\mathcal{R}$ be a ring then we denote the inverse elements in this ring by $\mathcal{R}^\times$. Let $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}$. Then we define the reduction $(f \mod q)$ of f modulo q to be $(f \mod q) = \sum_{i=0}^{n-1} (f_i \mod q) x^i \in \mathcal{R}_q$. Let $d \in \mathbb{N}$ and $c \in \mathbb{Z}$. We denote by $[c]_L$ the unique integer in $(-2^{d-1}, 2^{d-1}] \subset \mathbb{Z}$ such that $c = [c]_L$ modulo 2^d. Let $[\cdot]_M$ be the function $[\cdot]_M : \mathbb{Z} \to \mathbb{Z}, c \mapsto (c - [c]_L)/2^d$. Furthermore, let $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}_q$, then $[f]_L = \sum_{i=0}^{n-1} [f_i]_L\, x^i$ and $[f]_M = \sum_{i=0}^{n-1} [f_i]_M\, x^i$. Let $f \in \mathcal{R}_q$ be a polynomial with coefficients being ordered (without losing any generality) as $|f_1| \geq |f_2| \geq ... \geq |f_n|$. Then we define $\max_i(f) = f_i$.

The centered discrete Gaussian distribution for $x \in \mathbb{Z}$ with standard deviation σ is defined to be $\mathcal{D}_\sigma = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$, where $\sigma > 0$, $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$, and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{x=1}^{\infty} \rho_\sigma(x)$. We write $c \leftarrow_\sigma \mathbb{Z}$ to denote sampling a value c with distribution $\mathcal{D}_\sigma$. For a polynomial $c \in \mathcal{R}$, we write $c \leftarrow_\sigma \mathcal{R}$ to denote sampling each coefficient of c with distribution $\mathcal{D}_\sigma$. For a finite set S, we denote sampling the element s uniformly from S with $s \leftarrow_\$ S$.

We define the following functions (refer to the specified sections for explicit details about their implementation):

- The generation of the polynomial a as $\mathrm{GenA} : \{0,1\}^\kappa \to \mathcal{R}_q^\times$ (cf. Section 2.4.3),
- an encoding function to encode hash values to polynomials $\mathrm{Enc} : \{0,1\}^\kappa \to \mathbb{H}_{n,h}$ (cf. Section 2.4.4),
- the two pseudo random functions $\mathrm{PRF}_1 : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$ and $\mathrm{PRF}_2 : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}_{q,[B]}$ (cf. Section 2.4.5), and
- a hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$ (cf. Section 2.4.5).

The details of qTESLA's key generation, signing and signature verification are given in Algorithms 6, 7, and 8, respectively. The two subroutines *checkE* and *checkS* that are called during key generation are depicted in Algorithms 4 and 5, respectively.

Algorithm 4 Subroutine *checkE* to ensure correctness of the scheme; *checkE* ensures that $\|ec\|_\infty \leq L_E$

Require: $e \in \mathcal{R}$
Ensure: $\{0,1\}$ ▷ false, true

1: **if** $\sum_{i=1}^{h} \max_i(e) > L_E$ **then**
2: **return** 0
3: **end if**
4: **return** 1

Algorithm 5 Subroutine *checkS* to simplify the security reduction; *checkS* ensures that $\|sc\|_\infty \leq L_S$

Require: $s \in \mathcal{R}$
Ensure: $\{0,1\}$ ▷ false, true

1: **if** $\sum_{i=1}^{h} \max_i(s) > L_S$ **then**
2: **return** 0
3: **end if**
4: **return** 1

Algorithm 6 qTESLA's key generation

Require: -
Ensure: $sk = (s, e, \text{seed}_y, \text{seed}_a)$, $pk = (\text{seed}_a, t)$

1: $\text{seed}_a, \text{seed}_y \leftarrow_\$ \{0,1\}^\kappa$
2: $a \leftarrow \text{GenA}(\text{seed}_a)$
3: $s \leftarrow_\sigma \mathcal{R}$
4: **if** $checkS(s) = 0$ **then**
5: Restart at step 3
6: **end if**
7: $e \leftarrow_\sigma \mathcal{R}$
8: **if** $checkE(e) = 0$ **then**
9: Restart at step 7
10: **end if**
11: $t = as + e \mod q$
12: $sk \leftarrow (s, e, \text{seed}_y, \text{seed}_a)$
13: $pk \leftarrow (\text{seed}_a, t)$
14: **return** sk, pk

Remark 1. *We note that the description of our scheme can be easily generalized to use more than one sample of the ring learning with errors problem. In particular, that would mean that the public key consist of* $seed_{a_1}, ..., seed_{a_k}$ *(corresponding to* $a_1, ..., a_k$*) and* $t_1, ..., t_k$*, and that the secret key consist of the polynomials* $s, e_1, ..., e_k, seed_y$*. Our analysis of the expected security also holds for a generalization with* $k > 1$*. However, the description and implementation of the scheme are substantially simpler for* $k = 1$*.*

2.3 Correctness of the scheme

According to Algorithms 6 and 7, the following holds for an honestly generated signature (c', z) with $c = \text{Enc}(c')$ and elements from the key generation a, t, s, e:

Algorithm 7 qTESLA's signature generation

Require: m, $sk = (s, e, \text{seed}_y, \text{seed}_a)$
Ensure: c', z

1: $a \leftarrow \text{GenA}(\text{seed}_a)$
2: counter $\leftarrow 0$
3: rand $\leftarrow \text{PRF}_1(\text{seed}_y, m)$
4: $y \leftarrow \text{PRF}_2(\text{rand}, \text{counter})$
5: $v = ay \mod q$
6: $c' \leftarrow H(\lfloor v \rceil_M, m)$
7: $c \leftarrow \text{Enc}(c')$
8: $z \leftarrow y + sc$
9: **if** $z \notin \mathcal{R}_{q,[B-L_S]}$ **then**
10: counter $++$
11: Restart at step 4
12: **end if**
13: $w \leftarrow v - ec \mod q$
14: **if** $\|\lfloor w \rceil_L\|_\infty > 2^d - L_E \vee \|w\|_\infty > \lfloor q/2 \rfloor - L_E$ **then**
15: counter $++$
16: Restart at step 4
17: **end if**
18: **return** (c', z)

Algorithm 8 qTESLA's signature verification

Require: $m, (c', z), pk = (\text{seed}_a, t)$
Ensure: $\{0, 1\}$ ▷ reject, accept

1: $c \leftarrow \text{Enc}(c')$
2: $a \leftarrow \text{GenA}(\text{seed}_a)$
3: $w \leftarrow az - tc \mod q$
4: **if** $z \in \mathcal{R}_{q,[B-L_S]} \wedge c = H(\lfloor w \rceil_M, m)$ **then**
5: **return** 1
6: **end if**
7: **return** 0

$z \in \mathcal{R}_{q,[B-U]}$, $\|sc\|_\infty \leq L_S$, $\|ec\|_\infty \leq L_E$, $\|\lfloor ay - ec \rceil_L\|_\infty \leq 2^d - L_E$, and $\|ay - ec\|_\infty \leq \lfloor q/2 \rfloor - L_E$. In order for the verification algorithm to accept a signature it has to hold that: (i) $z \in \mathcal{R}_{q,[B-U]}$, which holds trivially, and (ii) $\lfloor ay \rceil_M = \lfloor az - tc \rceil_M$, which we argue next.

We know that

$$\begin{aligned} [az - tc]_M &= [ay + asc - asc - ec]_M && (1)\\ &= [ay - ec]_M && (2)\\ &= \frac{ay - ec - [ay - ec]_L}{2^d}. && (3) \end{aligned}$$

We know that $\|[ay - ec]_L\|_\infty < 2^d - L_E$ and $\|ay - ec\|_\infty \leq \lfloor q/2 \rfloor - L_E$. Hence, $\|ay - ec - [ay - ec]_L\|_\infty < q/2$, and thus, no wrap-around occurs. Furthermore, since $\|ec\|_\infty \leq L_E$ and $\|[ay - ec]_L\|_\infty \leq 2^d - L_E$, we know that $-ec - [ay - ec]_L = [-ec - (ay - ec)]_L$ and hence,

$$\frac{ay - ec - [ay - ec]_L}{2^d} = \frac{ay - [ay]_L}{2^d} = [ay]_M\,. \quad (4)$$

2.4 Implementation details of the required functions

2.4.1 Gaussian sampling

One of the advantages of qTESLA is that Gaussian sampling is only required during key generation to sample s and e (see Alg. 6). Nevertheless, certain applications might require an efficient and secure implementation of key generation and that, in particular, be protected against timing and cache attacks. In the following, we adopt the Gaussian sampler proposed in [16], which is an improvement upon the sampler proposed by Ducas *et al.* [29, Section 6].

The basic idea of the Gaussian sampler by Ducas *et al.* [29, Algorithms 10–12] is to start from a distribution that approximates the desired Gaussian distribution. From there, a high-quality Gaussian is obtained by rejection sampling guided by Bernoulli distributions $\mathcal{B}_\rho$ with parameters ρ related to the standard deviation σ of the desired Gaussian distribution. Ducas *et al.* implement those Bernoulli distributions by decomposing them into ℓ certain base distributions $(\mathcal{B}_{\rho_0}, \mathcal{B}_{\rho_1}, \ldots, \mathcal{B}_{\rho_{\ell-1}})$ where the ρ constants are precomputed to the desired accuracy, and then sampling from those base distributions to that accuracy. Even though this Bernoulli decomposition is reportedly quite efficient, its running time highly depends on the private bits. Besides that, each $\mathcal{B}_{c_\rho}$ must be sampled to the same precision as the target distribution, which is why the total amount of entropy needed to obtain one Gaussian sample is much higher than theoretically necessary, roughly $O(\ell\lambda)$ bits rather than $O(\lambda)$ for security level λ.

However, because qTESLA only needs a basic Gaussian sampler for key generation, it is possible to obtain a much simpler construction [16]. In particular, only one Bernoulli distribution $\mathcal{B}_\rho$ is needed, instead of ℓ base distributions $(\mathcal{B}_{\rho_0}, \mathcal{B}_\rho, \ldots, \mathcal{B}_{\rho_{\ell-1}})$. Thus, the bias

is simply computed by $\rho = \exp(-t/2\sigma^2)$ using well-known exponentiation techniques. The value ρ is an approximation of a real number in the interval $[0, 1]$ to the desired precision. For more details, refer to [16] and [29, Section 6].

2.4.2 Deterministic random bit generation

qTESLA requires the deterministic generation of random bits to produce seeds from random pre-seed values. Specifically, the key generation algorithm requires the generation of seeds seed_a and seed_y in Step 1 (Alg. 6). This is done with the SHA-3 derived extendable output function cSHAKE. The format to call this function is given by cSHAKE$(X, L, \text{“ ”}, S)$ for an input bit string X and a domain separator S [45] (note that the function-name bit string is left empty). The function returns a bit string of L bits as output.

2.4.3 Generation of a: GenA

In qTESLA, a polynomial a is freshly generated per secret/public keypair using a seed seed_a. This seed is then stored as part of the public key so that the signing and verification operations can regenerate a.

The approach above permits to save bandwidth since we only need κ bits to store seed_a instead of the $n\lceil\log(q)\rceil$ bits that are required to represent the full polynomial. Moreover, the use of a fresh a per keypair makes more difficult the introduction of backdoors and reduces drastically the scope of all-for-the-price-of-one attacks [10, 16].

The procedure to generate a is as follows. First, a pre-seed is obtained from the system RNG. This pre-seed is then hashed using cSHAKE to obtain seed_a, as described in Section 2.4.2. Finally, to generate a via the expansion of seed_a, we use cSHAKE [45] such that the output size is enough to fill out all the coefficients of the polynomial. Moreover, the output of cSHAKE is filtered to make sure that a belongs to the correct ring. Note that, as a precaution, we avoid exposing directly the output of the system RNG through seed_a, and use a hashed value instead.

2.4.4 Encoding function

The encoding function Enc takes the output of the hash function H and maps it to a vector with entries in $\{-1, 0, 1\}$ of length n and weight h (representing a polynomial of degree $n - 1$). In the signature generation we need to map the hash input $([v]_M, m)$ to a polynomial $c \in \mathbb{H}_{n,h} \subset \mathcal{R}_q$ (cf. line 6 and 7 of Algorithm 7). We break this up into $\text{Enc}(H([v]_M, m)) = \text{Enc}(c') = c$ to obtain smaller signatures $(c', z) \in \{0, 1\}^\kappa \times \mathcal{R}_q$.

We implement the encoding function Enc as in [1] and as depicted in Algorithm 9. The elements $r_1, ..., r_h$ are chosen randomly by a PRF, given $c' \leftarrow H([v]_M, m)$ as input. The value c_{pos} is the (pos)-th element of the vector $c \in \mathbb{H}_{n,h}$, which is initialized as a zero vector. This algorithm is an extension of an algorithm originally proposed in [32, Section 4.4] which in turn relies on [29].

Algorithm 9 Encoding function Enc

Require: $c' \in \{0,1\}^\kappa$
Ensure: $c \in \mathbb{H}_{n,h}$

1: $r_1, ..., r_{h-1}, r_h \leftarrow \text{PRF}(c')$
2: **for** $i = 1, ..., h$: **do**
3: $\quad pos \leftarrow (r_i \ll 8) \vee (r_{i+1})$
4: $\quad$ **if** $r_{i+2} \mod 2 = 1$ **then**
5: $\quad\quad c_{pos} \leftarrow -1$
6: $\quad$ **else**
7: $\quad\quad c_{pos} \leftarrow 1$
8: $\quad$ **end if**
9: **end for**
10: **return** c

2.4.5 Hash and pseudo-random functions

qTESLA's signing procedure requires the hash function H as well as the pseudo-random functions PRF_1 and PRF_2. We adopt SHA-3 [33] for function H, and cSHAKE [45] for functions PRF_1 and PRF_2.

PRF_1 takes as input the seed seed_y and the message m and maps it to a byte array, i.e., $\text{PRF}_1 : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^\kappa$ (cf. line 3 of Algorithm 7). To do this we use the output of cSHAKE.

PRF_2 takes as input the values rand and counter and maps them to a ring element, i.e., $\text{PRF}_2 : \{0,1\}^\kappa \times \mathbb{Z} \rightarrow \mathcal{R}_{q,[B]}$ (cf. line 4 of Algorithm 7). To do this we use the output of cSHAKE and split it into n chunks representing the coefficients of the polynomial y in $\mathcal{R}_{q,[B]}$.

It is worth noting that we take the hash output size κ to be larger or equal to the security level λ. This is consistent with the use of the hash in a Fiat-Shamir style signature scheme such as qTESLA. In the Fiat-Shamir paradigm for signatures, preimage resistance is relevant while collision resistance is much less, given that we take the hash size to be enough to resist preimage attacks[1].

[1] We chose the hash size aiming for security of Category 5, according to NIST's categories of security

2.5 System parameters and parameter selection

In this section, we describe qTESLA's system parameters and our choice of parameter sets. We summarize all bounds and our concrete parameter sets in Table 1. We explain how we estimate the bit security of our signature scheme in Section 5.2.

Herein, we propose three parameter sets that we classify according to NIST's categories of security as follows:

qTESLA-128:	NIST's security category 1,
qTESLA-192:	NIST's security category 3,
qTESLA-256:	NIST's security category 5.

Our parameters are chosen according to the security reduction provided in Theorem 6, Section 5.1. This implies the following: suppose that parameters are constructed for a certain security level. By virtue of our security reduction these parameters correspond to an instance of the R-LWE problem. Since our parameters are chosen according to the provided security reduction, this reduction provably guarantees that our scheme has the selected security level as long as the corresponding R-LWE instance is intractable. In other words, hardness statements for R-LWE instances have a provable consequence for the security levels of our scheme.

Since the presented reduction is tight, the tightness gap of our reduction is equal to 1 for our choice of parameters and, hence, the concrete bit security of our signature scheme is essentially the same as the bit hardness of the underlying R-LWE instance. We make our sage script used to choose parameters available. It is called `parameterchoice.sage` and can be found in the submission folder "Script_to_choose_parameters".

Let λ be the security parameter, i.e., the targeted bit security of the instantiation is λ. Let $n \in \mathbb{Z}_{>0}$ be the dimension, i.e., $n-1$ is the polynomial degree. To use efficient polynomial multiplication, i.e., the number theoretic transform (NTT) in the ring $\mathcal{R}_q$, we restrict ourselves to a polynomial degree of a power of two, i.e, $n = 2^l$ for $l \in \mathbb{N}$. Let σ be the standard deviation of the centered discrete Gaussian distribution that is used to sample the coefficients of the secret and error polynomials. To use the fast Gaussian sampler as described in Section 2.4.1, we choose $\sigma = \frac{\xi}{\sqrt{2\ln 2}}$ for some $\xi \in \mathbb{Z}_{>0}$. The parameter κ defines the output (resp., input) length of random functions described in Section 2.4.5. The parameter h defines the encoding function described in Section 2.4.4. More concretely, it defines the number of non-zero elements of the output of the encoding function.

The values L_E and L_S are used to bound the coefficients in the error and secret polynomials

for preimage resistance. In a scenario that excludes Groover'a algorithm a hash function with an output length of λ is expected to have preimage resistance of 2^λ. When considering the quadratic acceleration of Groover's algorithm, the preimage resistance is only $\approx 2^{\lambda/2}$. In such a case, the hash output length should be 2λ for an aspired security level of λ.

Table 1: Description and bounds of the parameters according to the tight security reduction in the quantum random oracle model with $q_h = 2^{128}$ and $q_s = 2^{64}$; we choose $M = 0.3$; we write parameters used in the implementation in **bold**

Param.	Description	Requirement	qTesla-128	qTesla-192	qTesla-256
λ	security parameter	-	128	192	256
$\mathbf{n}$	dimension ($n-1$ is the poly. degree)	power-of-two	**1 024**	**2 048**	**2 048**
σ, ξ	standard deviation of centered discrete Gaussian distribution	$\sigma = \frac{\xi}{\sqrt{2\ln 2}}$		**8.5**, 10	
$\mathbf{q}$	modulus	$q = 1 \mod 2n$, $q^n \geq \lvert\Delta\mathbb{S}\rvert \cdot \lvert\Delta\mathbb{L}\rvert \cdot \lvert\Delta\mathbb{H}\rvert$, $q^n \geq 2^{4\lambda+n(d+1)} 3 q_s^3 (q_s+q_h)^2$	**8 058 881** $\leq 2^{23}$	**12 681 217** $\leq 2^{24}$	**27 627 521** $\leq 2^{25}$
$\mathbf{h}$	# of non-zero entries of output elements of Enc	$2^h \cdot \binom{n}{h} \geq 2^{2\lambda}$	**36**	**50**	**72**
κ	output length hash function H and input length GenA, PRF_1, PRF_2, Enc	$\kappa \geq \lambda$		256	
$\mathbf{L_E}$, η_E $\mathbf{L_S}$, η_S	bound in *checkE* bound in *checkS*	$\eta_E \cdot h \cdot \sigma$ $\eta_S \cdot h \cdot \sigma$	**798**, 2.48 **758**, 2.61	**1 117**, 2.68 **1 138**, 2.63	**1 534**, 2.48 **1 516**, 2.51
$\mathbf{B}$	determines the interval the randomness is chosen in during sign	$B \geq \frac{\sqrt[n]{M}+2L_S-1}{2(1-\sqrt[n]{M})}$, near to power-of-two	$\mathbf{2^{20}-1}$	$\mathbf{2^{21}-1}$	$\mathbf{2^{22}-1}$
$\mathbf{d}$	number of rounded bits	$\left(1-\frac{2\cdot L_E+1}{2^d}\right)^n \geq 0.3$, $d > \log_2(B)$	**21**	**22**	**23**
$\lvert\Delta\mathbb{H}\rvert$ $\lvert\Delta\mathbb{S}\rvert$ $\lvert\Delta\mathbb{L}\rvert$	see definition below in the text	$\sum_{j=0}^{h}\sum_{i=0}^{h-j}\binom{n'}{2i}2^{2i}\binom{n'-2i}{j}2^j$ $(4(B-L_S)+1)^n$ $(2^{d+1}+1)$	$\approx 2^{447}$ $\approx 2^{22526}$ $2^{22}+1$	$\approx 2^{675}$ $\approx 2^{47102}$ $2^{23}+1$	$\approx 2^{898}$ $\approx 2^{49150}$ $2^{24}+1$
δ_w δ_z	acc. prob. of w in line 19 during sign acc. prob. z in line 19 during sign	experimentally experimentally	0.50 0.50	0.33 0.25	0.33 1.00
δ_{keygen}	acc. prob. of key pairs	experimentally		1.00	
sig size pk size sk size	theoretical size signature [byte] theoretical size public key [byte] theoretical size secret key [byte]	$\kappa + n(\lceil\log_2(B-L_S)\rceil+1)$ $n(\lceil\log_2(q)\rceil)+\kappa$ $2n(\lceil\log_2(t\cdot\sigma+1)\rceil)+2\kappa$ with $t = 13.4$, 16.4, or 18.9	2 720 2 976 1 856	5 664 6 176 4 160	5 920 6 432 4 128

during *checkE* and *checkS*, respectively. However, since the rejection probability of key pairs during the key generation is close to zero for our parameter sets (as determined experimentally) the key space is not restricted noticeably. Both bounds, L_E and L_S, impact the rejection probability during the signature generation, as follows. Larger the values of L_E and L_S will increase the acceptance probability during the key generation. But they will also decrease acceptance probability in the signature generation line 14 and line 9, respectively. We determine the best trade-off between those two acceptance probabilities experimentally. We start choosing $L_E = \eta_E \cdot h \cdot \sigma$ (resp., $L_S = \eta_S \cdot h \cdot \sigma$) with $\eta_E = \eta_S = 2.8$

and try different values for $\eta_E, \eta_S \in [2.0, 3.0]$. Let $M = 0.3$ be a value of our choosing that determines (together with L_S and B) the acceptance probability of the rejection sampling in line 9 Algorithm 7. The parameter B defines the interval of the random polynomial y (cf. line 4 of Algorithm 7) and it is determined by M and the parameter L_S as follows:

$$\left(\frac{2B - 2L_S + 1}{2B + 1}\right)^n \geq M \Leftrightarrow B \geq \frac{\sqrt[n]{M} + 2L_S - 1}{2(1 - \sqrt[n]{M})}.$$

We select the rounding value d to be larger than $\log_2(B)$ and such that the acceptance probability of the check $\|[w]_L\|_\infty > 2^d - L_E$ in Line 14 of Algorithm 7 is upper bounded by 0.7 when using the sage script to choose parameters. Changing the value L_E as described above, impacts the rejection probability of w as well. We determine the acceptance probability δ_z of z and δ_w of w during sign and the acceptance probability of key pairs δ_{keygen} experimentally and summarize the result in Table 1.

The parameter q is chosen to fulfill several bounds and assumptions that are motivated by the security reduction or efficient implementation requirements. To simplify our statement in the security reduction we ensure that $q^n \geq |\Delta\mathbb{S}| \cdot |\Delta\mathbb{L}| \cdot |\Delta\mathbb{H}|$ with the following definition of sets: $\mathbb{S}$ is the set of polynomials $z \in \mathcal{R}_{q,[B-L_S]}$ and $\Delta\mathbb{S} = \{z - z' \ : \ z, z' \in \mathbb{S}\}$, $\mathbb{H}$ is the set of polynomials $c \in \mathcal{R}_{q,[1]}$ with exactly h nonzero coefficients and $\Delta\mathbb{H} = \{c - c' \ : \ c, c' \in \mathbb{H}\}$, and $\Delta\mathbb{L} = \{x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M \in \mathcal{R}_{q,[2^d-1]}\}$. To choose parameters according to the security reduction the following equation (cf. Theorem 6) has to hold:

$$\frac{2^{3\lambda+n(d+1)} \cdot 3 \cdot q_s^3(q_s + q_h)^2}{q^n} \leq 2^{-\lambda} \Leftrightarrow q \geq \left(2^{4\lambda+n(d+1)} \cdot 3 \cdot q_s^3(q_s + q_h)^2\right)^{1/n}.$$

To be able to use fast polynomial multiplication we choose q to be a prime integer such that $q \mod 2n = 1$.

As stated in the NIST call for proposals (Section 4.A.4), we choose the number of classical queries to the sign oracle to be $q_s = 2^{64}$ for all our parameter sets. Moreover, we choose the number of queries of a hash function to be $q_h = 2^{128}$.

Key and signature sizes Given all parameters as explained above, we determine the key and signature sizes as follows. The theoretical length of the signature in bits is given by $\kappa + n \cdot (\lceil \log_2(B - L_S) \rceil + 1)$ and the public key is represented by $n \cdot (\lceil \log_2(q) \rceil) + \kappa$ bits. To determine the size of the secret key we note that for $t > 0$ it holds that $Pr_{x \leftarrow_\sigma \mathbb{Z}}[|x| > t\sigma] \leq 2e^{-t^2/2}$. For example for $t = 13.4$, $t = 16.4$, and $t = 18.9$ the probability $Pr_{x \leftarrow_\sigma \mathbb{Z}}[|x| > t\sigma]$ is less or equal 2^{-128}, 2^{-192}, and 2^{-256}, respectively. Therefore, the theoretical size of the secret key is given by $n \cdot (\lceil \log_2(14\sigma + 1) \rceil) + n \cdot (\lceil \log_2(t \cdot \sigma + 1) \rceil) + 2\kappa$ bits with $t = 13.4$, $t = 16.4$, and $t = 18.9$ for qTesla-128, qTesla-192, and qTesla-256, respectively.

Table 2: Different key and signature sizes of our proposed parameter sets; we abbreviate theoretical sizes with TS and sizes as used in the implementations with IS; sizes are given in bytes.

Parameter set	TS/IS	public key	secret key	signature
qTesla-128	TS	2 976	1 856	2 720
	IS	4 128	2 112	3 104
qTesla-192	TS	6 176	4 160	5 664
	IS	8 224	8 256	6 176
qTesla-256	TS	6 432	4 128	5 920
	IS	8 224	8 256	6 176

We determined the key and signature sizes in our reference implementation as smallest suitable data type which can hold $max((\lceil \log_2(14\sigma + 1) \rceil), (\lceil \log_2(t \cdot \sigma + 1) \rceil))$, which is byte for qTesla-128, and 16 bit integer for qTesla-192, and qTesla-256. Table 2 shows key and signature sizes according to the theoretical sizes and sizes as in the implementations for our three proposed parameter sets in comparison.

3 Performance analysis

The submission package includes a simple yet efficient reference implementation written exclusively in C.

To evaluate the performance of the provided implementation, we ran our benchmarking suite on a machine powered by a 2.40 GHz Intel Core i5-6300U (Skylake) processor, running Ubuntu 16.04.3 LTS. As is standard practice, TurboBoost was disabled during the tests. For compilation we used clang version 3.8.0 with the command `clang -O3`. See Table 3 for the results.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-128`	3 402	2 495	520	3 015
`qTESLA-192`	5 875	9 686	1 065	10 751
`qTESLA-256`	12 433	26 063	1 310	38 496

Table 3: Performance (in thousands of cycles) of qTESLA on a 2.40 GHz Intel Core i5-6300U (Skylake) processor. Cycle counts are rounded to the nearest 10^3 cycles.

The results in Table 3 correspond to a relatively simple implementation of qTESLA. Nevertheless, they demonstrate that the scheme is practical for most applications. We expect

significant improvements in the future with a fully optimized implementation.

4 Known answer values

The submission includes KAT values with tuples that contain message size (`mlen`), message (`msg`), public key (`pk`), secret key (`sk`), signature size (`smlen`) and signature (`sm`) values for all the proposed security levels. The KAT files can be found in the media folder: `\KAT\PQCsignKAT_qTesla-128.rsp`, `\KAT\PQCsignKAT_qTesla-192.rsp`, and `\KAT\PQCsignKAT_qTesla-256.rsp` for qTESLA-128, qTESLA-192 and qTESLA-256, respectively.

5 Expected security strength

It this section we discuss the expected security strength of and possible attacks against qTESLA. This includes two statements about the theoretical security and the parameter choices depending on them. To this end we first define the hardness assumptions qTESLA is based on.

We define the ring short integer solution problem (R-SIS) similar to [30].
Definition 2 (Ring short integer solution problem $R - SIS_{n,k,q,\beta}$)**.** *Given $a_1, ..., a_k \leftarrow_{\$} \mathcal{R}_q$. Then the ring short integer solution problem $R - SIS_{n,k,q,\beta}$ is to find solutions $u_1, ..., u_k, u_{k+1} \in \mathcal{R}_q$, $u_i \neq 0$ for at least one i, such that $(a_1, ..., a_k, 1) \cdot (u_1, ..., u_{k+1})^T = a_1u_1 + ... + a_ku_k + u_{k+1} = 0 \mod q$ and $\|u_1\|, ..., \|u_{k+1}\| \leq \beta$.*

We define the learning with errors distribution and the ring learning with errors problem (LWE) in the following.
Definition 3 (Learning with Errors Distribution)**.** *Let $n, q > 0$ be integers, $s \in \mathcal{R}$, and χ be a distribution over $\mathcal{R}$. We define by $\mathcal{D}_{s,\chi}$ the LWE distribution which outputs $(a, \langle a, s\rangle + e) \in \mathcal{R}_q \times \mathcal{R}_q$, where $a \leftarrow_{\$} \mathcal{R}_q$ and $e \leftarrow \chi$.*

Since our signature scheme is based on the decisional learning with errors problem, we omit the definition of the search version and state only the decisional learning with errors problem.
Definition 4 (Ring Learning with Errors Problem $R - LWE_{n,m,q,\chi}$)**.** *Let $n, q > 0$ be integers and χ be a distribution over $\mathcal{R}$. Moreover, let $s \in \mathcal{R}$ and $\mathcal{D}_{s,\chi}$ be the learning with errors distribution. Given m tuples $(a_1, t_1), ..., (a_m, t_m)$, the decisional ring learning with errors problem $R - LWE_{n,m,q,\chi}$ is to distinguish whether $(a_i, t_i) \leftarrow \mathcal{U}(\mathcal{R}_q \times \mathcal{R}_q)$ or $(a_i, t_i) \leftarrow \mathcal{D}_{s,\chi}$ for all i.*

5.1 Provable security in the (quantum) random oracle model

The security of our scheme qTESLA is supported by two statements reducing the hardness of lattice-based assumptions to the security of our proposed signature scheme in the (quantum) random oracle model. In this subsection we give the two statements but we do not give formal security proofs since they are very close to the original results as explained below.

The first reduction (cf. Theorem 5) follows the approach proposed by Bai and Galbraith [14] closely and gives a non-tight reduction from R-LWE and R-SIS to the existentially unforgeability under chosen-message attack (EUF-CMA) of qTESLA in the random oracle model.

Theorem 5. *Let* $2^n \cdot \binom{n}{h} \geq 2^\lambda$, $(2R+1)^2 \geq q^n 2^\kappa$, *and* $q > 4B$. *If there exists an adversary* A *that forges a signature of the signature scheme qTESLA described in Section 2.2 in time* t_Σ *and with success probability* ϵ_Σ, *then there exists a reduction* R *that solves either*

- *the* $R-LWE_{n,m,q,\sigma}$ *with* $m = 1$ *problem in time* $t_{LWE} \approx t_\Sigma$ *with* $\epsilon_{LWE} \geq \epsilon_\Sigma/2$, *or*
- *the* $R-SIS_{n,k,q,\beta}$ *problem with* $\beta = \max\{k2^{d-1}, 2(B-U)\} + 2hR$ *in time* $t_{SIS} \approx 2t_\Sigma$ *with* $\epsilon_{SIS} \geq \frac{1}{2}(\epsilon_\Sigma - \frac{1}{2^\kappa}) \left(\frac{(\epsilon_\Sigma - \frac{1}{2^\kappa})}{q_h} - \frac{1}{2^\kappa} \right) + \epsilon_\Sigma/2$ *with our choice of parameters.*

The second security reduction (cf. Theorem 6) gives a tight reduction in the *quantum* random oracle model from R-LWE to EUF-CMA of qTESLA. In our opinion the second theorem is much stronger since it shows security against adversaries that have quantum access to a quantum random oracle and we will therefore always refer to Theorem 6 when we talk about the security of the scheme. We emphasize that Theorem 6 gives a reduction from the decisional ring learning with errors problem where in Theorem 5 also the decisional ring SIS problem is used. Currently, Theorem 6 holds assuming a conjecture as stated and explained below.

Theorem 6. *Let the parameters be as in Table 1. Furthermore, assume that Conjecture 7 holds. If there exists an adversary* A *that forges a signature of the signature scheme qTESLA described in Section 2.2 in time* t_Σ *and with success probability* ϵ_Σ, *then there exists a reduction* R *that solves the* $R-LWE_{n,m,q,\sigma}$ *problem with* $m = 1$ *in time* $t_{LWE} \approx t_\Sigma$ *with* $\epsilon_\Sigma \leq \frac{2^{3\lambda+(d+1)} \cdot 3 \cdot q_s^3 (q_s+q_h)^2}{q} + \frac{2q_h+5}{2^\lambda} + \epsilon_{LWE}$ *with our choice of parameters.*

The proof follows the approach proposed in [9] except for the computation of the two probabilities $\mathrm{coll}(a, e)$ and $\mathrm{nwr}(a, e)$ that we explain in the following. For simplicity we assume that the randomness is sampled uniformly random in $\mathcal{R}_{q,[B]}$ as in Algorithm 2. We define $\Delta\mathbb{L}$ to be the set $\{x - x' : x, x' \in \mathcal{R}$ and $[x]_M = [x']_M \in \mathcal{R}_{q,[2^d-1]}\}$. Furthermore, we call a polynomial w *well-rounded* if w is in $\mathcal{R}_{q,[\lfloor q/2 \rfloor - L]}$ and $[w] \in \mathcal{R}_{q,[(2^d-L)]}$. We define

the following quantities for keys (a, t), (s, e)

$$\mathrm{nwr}(a, e) \stackrel{\mathrm{def}}{=} \Pr_{(y,c)\in\mathbb{Y}\times\mathbb{H}}[ay - ec \text{ not well-rounded }] \tag{5}$$

$$\mathrm{coll}(a, e) \stackrel{\mathrm{def}}{=} \max_{(w)\in\mathbb{W}} \left\{ \Pr_{(y,c)\in\mathbb{Y}\times\mathbb{H}}[[ay - ec]_M = w] \right\}. \tag{6}$$

Informally speaking $\mathrm{nwr}(a, e)$ refers to the probability over random (y, c) that $ay - ec$ is not well-rounded. This quantity varies as a function of a, e. In contrast to [9], we cannot upper bound this in general in the ring setting. Hence, we first assume that $\mathrm{nwr}(a, e) < \frac{2}{3}$ and afterwards check experimentally that this holds true. As our acceptance probability of w in line 19 of Algorithm 7 (signature generation) is at least 0.34 for all parameter sets (cf. δ_w in Table 1), the bound $\mathrm{nwr}(a, e) < \frac{2}{3}$ holds.

Secondly, we need to bound the probability $\mathrm{coll}(a, e)$. In [9, Lemma 4] the corresponding probability $\mathrm{coll}(A, E)$ for standard lattices is upper bounded. Unfortunately, we were not able to transfer the proof to the ring setting for the following reason. In the proof of [9, Lemma 4], it is used that if the randomness y is not equal to 0 the vector Ay is uniformly random distributed over $\mathbb{Z}_q$ and hence also $Ay - Ec$ is uniformly random distributed over $\mathbb{Z}_q$. This does not necessarily hold if the *polynomial* y is chosen uniformly in $\mathcal{R}_{q,[B]}$. Moreover, in Equation (99) in [9], ψ denotes the probability that a random vector $x \in \mathbb{Z}_q^m$ is in $\Delta\mathbb{L}$:

$$\psi \stackrel{\mathrm{def}}{=} \Pr_{x\in\mathbb{Z}_q^m}[x \in \Delta\mathbb{L}] \leq \left(\frac{2^{d+1}}{q}\right)^m. \tag{7}$$

The quantity ψ is a function of the TESLA parameters q, m, d. It is negligibly small.

We cannot prove a similar statement for the signature scheme qTESLA over ideals. Instead, we need to *conjecture* the following.

Conjecture 7. *Let I be a non-zero ideal in $\mathcal{R}_q$ and let $r \in \mathcal{R}_q$ be a fixed choice of ring elements. Then it holds that the probability over a uniformly distributed element $x \leftarrow_\$ I$ that $x + r \in \Delta\mathbb{L}$ is negligibly small.*

The intuition behind our conjecture is as follows. Let ψ_I denote the probability that a random element from the ideal I lands in $\Delta\mathbb{L}$. We know that ψ_I is small when the ideal $I = \mathcal{R}_q$, i.e., a negligibly small fraction of elements from $\mathcal{R}_q$ are in $\Delta\mathbb{L}$. Furthermore, the set $\Delta\mathbb{L}$ appears to have no relationship with the ideal structure of the ring, so it seems reasonable to view each ideal as a "random" subset of $\mathcal{R}_q$ in the following sense: No larger or smaller portion of elements in the ideal I is in $\Delta\mathbb{L}$ than that portion of elements of $\mathcal{R}_q$ that is in $\Delta\mathbb{L}$.

Hence, the corresponding statement described above and needed in [9, Lemma 4] translates for qTESLA to the following. If $y \neq 0$ then ay is a uniformly random element of some non-

zero ideal I. The polynomial c is fixed and the polynomial e is independent of the polynomial a, and y. Hence, by our conjecture (with $x = ay$ and $r = ec$) it holds that the probability of Equation (107) in [9] is negligibly small. Thus, assuming that our conjecture holds true, [9, Lemma 4] and hence the security reduction in [9] holds for qTESLA as well.

5.2 Bit security of our proposed parameter sets

In the following we describe how we estimate the concrete security of our proposed parameters. To this end, we first describe how the security of our scheme depends on the hardness of R-LWE and afterwards we describe how we derive the bit hardness of the underlying R-LWE instance. We classify our three parameter sets according to NIST's categories of security in Section 2.5.

5.2.1 Correspondence between security and hardness

The security reduction given in Section 5.1, Theorem 6 provides a reduction from the hardness of the decisional ring learning with errors problem and bounds *explicitly* the forging probability with the success probability of the reduction. More formally, let ϵ_Σ and t_Σ denote the success probability and the run time of a forger against our signature scheme and let ϵ_{LWE} and t_{LWE} denote analogous quantities for the reduction presented in the proof of Theorem 6. We say that R-LWE is η-*bit hard* if $t_{LWE}/\epsilon_{LWE} \geq 2^\eta$; and we say that the signature scheme is λ-*bit secure* if $t_\Sigma/\epsilon_\Sigma \geq 2^\lambda$.

Since we choose parameters such that $\epsilon_{LWE} \approx \epsilon_\Sigma$ and $t_\Sigma \approx t_{LWE}$, the bit hardness of the R-LWE instance is the same as the bit security of our signature scheme.

5.2.2 Estimation of the hardness of R-LWE

Since the introduction of the learning with errors problem over rings [52], it is an open question whether the R-LWE problem is as hard as the LWE problem. Several results exist that exploit the ideal structure of some ideal lattices [23, 26, 35, 37]. However, up to now, these results are not known to be applicable to R-LWE. In particular, the found weaknesses do not apply to our instances. Consequently, we estimate the hardness of R-LWE using state-of-the-art attacks against LWE.

Albrecht, Player, and Scott [8] presented the *LWE-Estimator*, a software to estimate the hardness of LWE given the matrix dimension n, the modulus q, the relative error rate $\alpha = \frac{\sqrt{2\pi}\sigma}{q}$, and the number of given LWE samples. The LWE-Estimator estimates the hardness against the fastest LWE solvers currently known, i.e., it outputs an upper (conservative) bound on the number of operations an attack needs to break a given LWE instance.

In particular, the following attacks are considered in the *LWE-Estimator*: The meet-in-the-middle exhaustive search, the coded Blum-Kalai-Wassermann algorithm [42], the dual lattice recently published [3], the enumeration approach by Linder and Peikert [49], the primal attack described in [6,15], and the Arora-Ge algorithm [11] using Gröbner bases [4]. Moreover, the latest analysis to compute the block sizes used in the lattice basis reduction BKZ published recently by Albrecht *et al.* [2] are implemented.

Furthermore, quantum speed-ups for the sieving algorithm used in BKZ [47, 48] are considered. Another recent quantum attack, called quantum hybrid attack, by Göpfert, van Vredendaal, and Wunderer [40] is not considered in our analysis (and the *LWE-Estimator*). The hybrid attack is most efficient on the learning with errors problem with very small secret and error, e.g., binary or ternary. Since the coefficients of the secret and error of qTESLA are chosen Gaussian distributed, the attack is not efficiently applicable on our instances.

The *LWE-Estimator* is the result of many different contributions and contributors. It is open source and hence easily checked and maintained by the community. Hence, we find the *LWE-Estimator* to be a suitable tool to estimate the hardness of our chosen LWE instances. We integrated the LWE-Estimator with commit-id `9302d42` on 2017-09-27 in our sage script.

In the following we describe very briefly the most efficient LWE solvers for our instances, i.e., the decoding attack and the embedding approach, following closely the description of [18]. The Blum-Kalai-Wasserman algorithm [5, 46] is omitted since it requires exponentially many samples.

The embedding attack. The standard embedding attack solves LWE via reduction to the unique shortest vector problem (uSVP). During the reduction an $m+1$-dimensional lattice that contains the error vector e is created. Since e is very short for typical LWE instances, this results in a uSVP instance that is usually solved by applying basis reduction.

Let $(A, c = As + e \mod q)$ and t be the distance $\mathrm{dist}(c, L(A)) = \|c - x\|$ where $x \in L(A)$, such that $\|c-x\|$ is minimized. Then the lattice $L(A)$ can be embedded in the lattice $L(A')$, with $A' = \begin{pmatrix} A & c \\ 0 & t \end{pmatrix}$. If $t < \frac{\lambda_1(L(A))}{2\gamma}$, the higher-dimensional lattice $L(A')$ has a unique shortest vector $c' = (-e, t) \in Z_q^{m+1}$ with length $\|c'\| = \sqrt{m\alpha^2q^2/(2\pi) + |t|^2}$ [27,51]. In the LWE-Estimator $t = 1$ is used. Therefore, e can be extracted from c', As is known, and s can be solved for. Based on Albrecht *et al.* [7], Göpfert shows [39, Section 3.1.3] that the standard embedding attack succeeds with non-negligible probability if $\delta_0 \leq \left(\frac{q^{1-\frac{n}{m}}\sqrt{\frac{1}{e}}}{\tau\alpha q} \right)^{\frac{1}{m}}$,

where m is the number of LWE samples. The value τ is experimentally determined to be $\tau \leq 0.4$ for a success probability of $\epsilon = 0.1$ [7].

The efficiency of the embedding attack highly depends on the number of samples. In case of LWE instances with limited number of samples, the lattice $\Lambda_q^{\perp}(A_o) = \{v \in \mathbb{Z}^{m+n+1} | A_o \cdot v = 0 \bmod q\}$ with $A_o = [A|I|b]$ can be used as the embedding lattice.

The decoding attack. The decoding attack treats an LWE instance as an instance of the bounded distance decoding problem (BDD). The attack can be divided into two phases: Basis reduction and finding closest vector to target vector. In the first phase, basis reduction algorithms like BKZ [55] are applied. Afterwards, in the second phase, the nearest plane algorithm [13] (or variants) are applied to find the closest vector to As and thereby eliminate the error vector e of the LWE instance. Now, the secret can be accessed, as the closest vector equals an LWE instance's As.

5.3 Resistance to implementation attacks

Recently, the scheme ring-TESLA [1] was analyzed with respect to cache side channels with the software tool CacheAudit [20]. It was the first time that a post-quantum scheme was analyzed with program analysis. The authors found potential cache side channels, proposed countermeasures, and showed the effectiveness of their mitigations with CacheAudit. Since the implementation of ring-TESLA is similar to our implementation of qTESLA, we implemented all countermeasures proposed in [20] to secure our scheme against bit leakage via cache side channels.

The implementation of ring-TESLA was also analyzed regarding fault attacks [19, 36] and it was found that ring-TESLA is vulnerable to fewer fault attacks then, e.g., the signature scheme BLISS [29]. Due to the similarities of the implementations of ring-TESLA and qTESLA, the results from [19] are transferable to qTESLA. Another possible fault attack is described in Section 5.4.

5.4 Deterministic vs. probabilistic signature scheme

The following discussion is about how to generate the randomness y in Algorithm 7, line 4-6, and how different approaches prevent or enable different attacks.

In the current description in Algorithm 7, signatures are generated deterministically, i.e., for the same message always the same signature is generated. To this end an additional secret seed_y is part of the secret key. The value seed_y is used to generate a randomness rand and afterwards, rand is used to generate the polynomial y. The advantage of this approach

is that a different randomness is used for different messages with very high probability. Hence, attacks that exploit a fixed randomness, such as done for Sony's playstation 3 [22], are prevented. Another advantage is that no access to a source of high-quality randomness is needed.

Our approach, however, might open a vulnerability to a fault attack proposed in [53] and briefly described in the following: Assume a signature (z, c) is generated for message m. Afterwards, a signature for the same message m is asked again. However, during the generation of the second signature a fault is injected on the hash value c yielding the value c_{faulted}, hence the second signature is $(z_{\text{faulted}}, c_{\text{faulted}})$. Computing $z - z_{\text{faulted}} = sc - sc_{\text{faulted}} = s(c - c_{\text{faulted}})$, gives the s since $c - c_{\text{faulted}}$ is known to the attacker. The authors of [53] argue that the attack is rather realistic and that it is applicable to all deterministic *Schnorr-like* signatures. To prevent the fault attack but to still get new randomness for every message one could use *weak* randomness as input for the PRF. For example, instead of using the same seed_y from the secret key, $\text{seed}_y \leftarrow_{\$} \{0,1\}^{\kappa}$ could be sampled freshly every time. This would yield again a probabilistic signature scheme. Hence, we decided to stick to our proposal. Furthermore, in [53] the attack is only described against ECDSA and EdDSA signatures. Due to the rejection sampling and other correctness checks during the signature generation, this fault attack might not be as successful on our signature scheme as it is on ECDSA and EdDSA signatures.

6 Advantages and limitations

In this section we summarize the advantages and limitations of our proposed signature scheme qTESLA. Within that we compare our scheme with other post-quantum and classical signatures.

Security of our signature scheme. Our signature scheme is provably EUF-CMA secure: a security reduction from the hardness of the decisional ring learning with errors problem to EUF-CMA security of our scheme is given. Our security reduction (cf. Theorem 6) is given in the quantum random oracle model, i.e., a quantum adversary is allowed to ask the random oracle in super position. Our security reduction is based on a variant of our scheme over standard lattices [9]. To port the reduction given in [9], we use a heuristic argument as explained in Section 5.1. Our security reduction is explicit, i.e., we can explicitly give the relation between the success probabilities of solving the R-LWE problem and to forge signatures of qTESLA. Our security reduction is tight which is a desirable property because when choosing the scheme's parameters according to security reductions, tight reductions lead to smaller parameters and hence better performance.

Choice of parameters. Parameters can be chosen either heuristically or according to existing security reductions. The heuristic approach identifies the security level of an instantiation of a scheme by a certain parameter set with the hardness level of the instance of the underlying lattice problem that corresponds to these parameters regardless of the tightness gap of the provided security reduction. The parameter choice according to a reduction can be considered as a more convincing security argument since it provably guarantees that our scheme has the selected security level as long as the corresponding R-LWE instance is intractable. Our three parameter sets are chosen regarding our given quantum security reduction.

The security of our proposed parameter sets are estimated against known state-of-the-art classical and quantum algorithms to solve the learning with errors problem. Furthermore, our parameters are chosen with a comfortable gap between the targeted and the estimated bit security they provide such that they might be secure against improved or unknown LWE solvers as well. Moreover, our choice of parameters is easy comprehensible: All relations between the parameters are explained and we make our sage script used to choose parameters available[2]. Hence, if more parameter sets are needed they can be chosen easily.

Ease of Implementation. qTESLA has a very compact structure consisting of a few, ease-to-implement functions. Moreover, in contrast to popular R-LWE based schemes, qTESLA does not enforce the use of the number theoretic transform (NTT), i.e., its use is optional and the scheme remains fully compatible with an implementation that uses a straightforward schoolbook polynomial multiplication. This design decision enables the possibility of even simpler implementations. Another advantage of qTESLA is that Gaussian sampling is only required during key generation. Even if the fast Gaussian sampler included in this document is not used, most applications will not be impacted by the use of a slower Gaussian sampler.

Implementation attacks. We protect the signature generation against cache side channels by implementing the countermeasures proposed in [20]. Furthermore, the predecessor of our proposed scheme was already analyzed with respect to fault attacks [19, 36].

Applicability of our scheme. Our proposal is a good candidate to be integrated to hybrid signature schemes easing the transition from classical to post-quantum cryptography. The key sizes of all three parameter sets are small enough to be used in hybrid signature schemes [21]. Following [21] it should be appropriate to be used in X.509 standard version 3 [25], to be used in TLSv1.2 [28] for most browsers and libraries tested in [21], and to

[2]It is called `parameterchoice.sage` and can be found in the submission folder "Script_to_choose_parameters".

be used in the Cryptographic Message Syntax (CMS) [43] that is the main cryptographic component of S/MIME [54].

Comparison with selected state-of-the-art signature schemes. In the following we give a comparison of the key and signature sizes with selected classical and post-quantum signature schemes. We do not compare qTESLA with other post-quantum signatures regarding the running time because cycle counts, in particular for lattice-based signature schemes, are usually given for optimized implementations that utilize fast AVX2 arithmetic. Such optimizations, however, are not requested by NIST. A comparison of cycle counts obtained from different platforms might be misleading.

Table 4 summarizes the key and signature sizes of selected signature schemes. Moreover, it also states the underlying computational assumptions although not all construction do rely *provably* on the corresponding hardness assumption. Furthermore, only few of the parameters in the table are chosen according to provided security reductions and the bit security of the parameters are not always estimated against classical and quantum adversaries. We distinguish the different was to choose parameters in the table.

As can be seen in Table 4, qTESLA is among the post-quantum schemes with the smallest signature size if parameters are chosen with regard to quantum algorithms. In particular, the signature size of qTESLA is several magnitudes smaller than hash-based and multivariate signatures. Only the lattice-based scheme BLISS has noticeably smaller signatures. The parameters proposed for BLISS, however, are not chosen with state-of-the-art methods, not according to the provided security reduction, and the bit security is not estimated against quantum adversaries.

In comparison with the classical signature schemes RSA and ECDSA for the same security level, qTESLA has larger signature sizes. However, qTESLA is comparable with RSA-3072 in view of secret key size.

Table 4: Overview of selected state-of-the-art post-quantum and classical signature schemes; signature and key sizes are given in byte [B]; we write "–" if no corresponding data is available

Software/ Scheme	Comp. Assum.	Bit Security	Key Size [B]	Sig. Size [B]
Selected lattice-based signatures schemes				
qTESLA qTesla-128[a] (this document)	R-LWE	128[b]	pk: 2 976 sk: 1 856	2 720
qTESLA qTesla-192[a] (this document)	R-LWE	192[b]	pk: 6 176 sk: 4 160	5 664
qTESLA qTesla-256[a] (this document)	R-LWE	256[b]	pk: 6 432 sk: 4 128	5 920
Dilithium -high [30]	module SIS module LWE	125[b]	pk: 1 472 sk: –	2 700
GPV-poly[a] [34, 38]	R-SIS	96[c]	pk: 55 705 sk: 26 316	32 972
BLISS-B-IV [31, 57]	R-SIS, NTRU	182[c]	pk: 896 sk: 384	812
Selected other post-quantum signature schemes				
gravity-SPHINCS [12]	Hash collisions, 2nd preimage	128[b]	pk: 32 sk: 64	22 304
SPHINCS-256 [17]	Hash collisions, 2nd preimage	128[b]	pk: 1 056 sk: 1 088	41 000
MQDSS-31-64 [24]	Multivariate Quadratic system	128[b]	pk: 72 sk: 64	40 952
Selected classic signature schemes				
RSA-3072 [56]	Integer Factorization	128[d]	pk: 384 sk: 1 728	384
ECDSA (P-256) [44]	Elliptic Curve Discrete Logarithm	128[d]	pk: 64 sk: 96	64

[a] Parameters are chosen according to given security reduction in the quantum random oracle model.
[b] Bit security analyzed against classical and quantum adversaries.
[c] Bit security analyzed against classical adversaries.
[d] Broken against quantum computers (bit security analyzed against classical adversaries).

References

[1] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa*, volume 9646 of *LNCS*, pages 44–60. Springer, 2016.

[2] Martin Albrecht, Florian Göpfert, Fernando Vidria, and Thomas Wunderer. Revisiting the Expected Cost of Solving uSVP and Applications to LWE. In *ASIACRYPT 2017 - Advances in Cryptology, to appear*. Springer, 2017.

[3] Martin R. Albrecht. On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *LNCS*, pages 103–129, 2017.

[4] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *ACM Comm. Computer Algebra*, 49(2):62, 2015.

[5] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Designs, Codes and Cryptography*, 74(2):325–354, 2015.

[6] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-svp. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013*, volume 8565 of *LNCS*, pages 293–310. Springer, 2013.

[7] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In Hyang-Sook Lee and Dong-Guk Han, editors, *ICISC 13: 16th International Conference on Information Security and Cryptology*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310, Seoul, Korea, November 27–29, 2014. Springer, Heidelberg, Germany.

[8] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[9] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting TESLA in the quantum random oracle model. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum*

Cryptography - 8th International Workshop, PQCrypto 2017, volume 10346 of *LNCS*, pages 143–162. Springer, 2017.

[10] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 327–343. USENIX Association, 2016.

[11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *LNCS*, pages 403–415. Springer, 2011.

[12] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. Cryptology ePrint Archive, Report 2017/933, 2017. https://eprint.iacr.org/2017/933.

[13] László Babai. On lovász' lattice reduction and the nearest lattice point problem. In K. Mehlhorn, editor, *STACS 1985*. Springer, 1985.

[14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47, San Francisco, CA, USA, February 25–28, 2014. Springer, Heidelberg, Germany.

[15] Shi Bai and Steven D. Galbraith. Lattice decoding attacks on binary LWE. In Willy Susilo and Yi Mu, editors, *ACISP 14: 19th Australasian Conference on Information Security and Privacy*, volume 8544 of *Lecture Notes in Computer Science*, pages 322–337, Wollongong, NSW, Australia, July 7–9, 2014. Springer, Heidelberg, Germany.

[16] Paulo S. L. M. Barreto, Patrick Longa, Michael Naehrig, Jefferson E. Ricardini, and Gustavo Zanon. Sharper ring-lwe signatures. Cryptology ePrint Archive, Report 2016/1026, 2016. http://eprint.iacr.org/2016/1026.

[17] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[18] Nina Bindel, Johannes Buchmann, Florian Göpfert, and Markus Schmidt. Estimation of the hardness of the learning with errors problem with a restricted number of samples. Cryptology ePrint Archive, Report 2017/140, 2017. https://eprint.iacr.org/2017/140.

[19] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis*

and Tolerance in Cryptography, FDTC 2016, pages 63–77. IEEE Computer Society, 2016.

[20] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In *Proceedings of the 10th International Symposium on Foundations & Practice of Security (FPS)*, 2017. To appear.

[21] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. Transitioning to a quantum-resistant public key infrastructure. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, volume 10346 of *LNCS*, pages 384–405. Springer, 2017.

[22] bushing, marcan, and sven. Console hacking 2010 – ps3 epic fail. 27th Chaos Communication Congress, 2010. https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.

[23] Peter Campbell, Michael Groves, and Dan Shepherd. SOLILOQUY: A cautionary tale. ETSI 2nd Quantum-Safe Crypto Workshop, 2014. http://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.

[24] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass *MQ* -based identification to *MQ* -based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security*, volume 10032 of *LNCS*, pages 135–165, 2016.

[25] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[26] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9666 of *LNCS*, pages 559–585. Springer, 2016.

[27] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. High-speed signatures from standard lattices. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 84–103. Springer, 2015.

[28] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[29] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.

[30] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Report 2017/633, 2017. http://eprint.iacr.org/2017/633.

[31] Léo Ducas. Accelerating bliss: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874, 2014. http://eprint.iacr.org/2014/874/.

[32] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Report 2013/383, 2013. https://eprint.iacr.org/2013/383.

[33] M. J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. National Institute of Standards and Technology (NIST), Gaithersburg (MD), USA, 8 2015.

[34] Rachid El Bansarkhani and Jan Sturm. An efficient lattice-based multisignature scheme with applications to bitcoins. In Sara Foresti and Giuseppe Persiano, editors, *CANS 2016*, pages 140–155, Cham, 2016. Springer International Publishing.

[35] Yara Elias, Kristin E. Lauter, Ekin Ozman, and Katherine E. Stange. Provably weak instances of ring-lwe. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference*, volume 9215 of *LNCS*, pages 63–92. Springer, 2015.

[36] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference*, volume 10532 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2017.

[37] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[38] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the fortieth annual ACM symposium on Theory of computing (STOC 2008)*, pages 197–206. ACM, 2008.

[39] Florian Göpfert. *Securely Instantiating Cryptographic Schemes Based on the Learning with Errors Assumption*. PhD thesis, Darmstadt University of Technology, Germany, 2016.

[40] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, volume 10346 of *LNCS*, pages 184–202. Springer, 2017.

[41] Shay Gueron and Fabian Schlieker. Optimized implementation of ring-TESLA. GitHub at https://github.com/fschlieker/ring-TESLA, 2016.

[42] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-bkw: Solving LWE using lattice codes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 23–42. Springer, 2015.

[43] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (INTERNET STANDARD), September 2009.

[44] James Howe, Thomas Pöppelmann, Máire O'neill, Elizabeth O'sullivan, and Tim Güneysu. Practical lattice-based digital signature schemes. *ACM Trans. Embed. Comput. Syst.*, 14, 2015.

[45] John Kelsey. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. *NIST Special Publication*, 800:185, 2016.

[46] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for lwe with applications to cryptography and lattices. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 43–62. Springer, 2015.

[47] Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2016.

[48] Thijs Laarhoven, Michele Mosca, and Joop Pol. Solving the Shortest Vector Problem in Lattices Faster Using Quantum Search. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *Lecture Notes in Computer Science*, pages 83–101. Springer Berlin Heidelberg, 2013.

[49] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011.

[50] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[51] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, pages 577–594. Springer, 2009.

[52] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[53] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017. http://eprint.iacr.org/2017/1014.

[54] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), January 2010.

[55] Claus P. Schnorr and Taras Shevchenko. Solving subset sum problems of densioty close to 1 by randomized BKZ-reduction. Cryptology ePrint Archive, Report 2012/620, 2012. http://eprint.iacr.org/2012/620.

[56] Mikael Sjöberg. *Post-quantum algorithms for digital signing in Public Key Infrastructures*. PhD thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2017.

[57] Thomas Wunderer. Revisiting the hybrid attack: Improved analysis and refined security estimates. Cryptology ePrint Archive, Report 2016/733, 2016. https://eprint.iacr.org/2016/733.

Name of Proposal:

Rainbow

Principal Submitter:

Jintai Ding

email: jintai.ding@gmail.com
phone: 513 556 - 4024
organization: University of Cincinnati
postal address: 4314 French Hall, OH 45221 Cincinnati, USA

Auxiliary Submitters: Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang

Inventors: c.f. Submitters

Owners: c.f. Submitters

Jintai Ding (Signature)

Additional Point of Contact:
Bo-Yin Yang
email: by@crypto.tw
phone: 886-2-2788-3799
Fax: 886-2-2782-4814
organization: Academia Sinica
postal address: 128 Academia Road, Section 2
Nankang, Taipei 11529, Taiwan

Rainbow - Algorithm Specification and Documentation

Type: Signature scheme

Family: Multivariate Cryptography, SingleField schemes

1 Algorithm Specification

In this section we present the Rainbow signature scheme as proposed in [7].

1.1 Parameters

- finite field $\mathbb{F} = \mathbb{F}_q$ with q elements
- integers $0 < v_1 < \cdots < v_u < v_{u+1} = n$
- index sets $V_i = \{1, \ldots, v_i\},\ O_i = \{v_i + 1, \ldots, v_{i+1}\}$ $(i = 1, \ldots, u)$
  Note that each $k \in \{v_1 + 1, \ldots, n\}$ is contained in exactly one of the sets O_i.
- we have $|V_i| = v_i$ and set $o_i := |O_i|$ $(i = 1, \ldots, u)$
- number of equations: $m = n - v_1$
- number of variables: n

1.2 Key Generation

Private Key. The *private key* consists of

- two invertible affine maps $\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m$ and $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$
- the quadratic *central* map $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^n$, consisting of m multivariate polynomials $f^{(v_1+1)}, \ldots, f^{(n)}$.
  Remember that, according to the definition of the sets O_i (see Section 1.1), there exists, for any $k \in \{v_1 + 1, \ldots, n\}$ exactly one $\ell \in \{1, \ldots, u\}$ such that $k \in O_\ell$. The polynomials $f(k)$ $(k = v_1 + 1, \ldots, n)$ are of the form

$$\begin{aligned} f^{(k)}(x_1, \ldots, x_n) &= \sum_{i,j \in V_\ell, i \leq j} \alpha_{ij}^{(k)} x_i x_j \\ &+ \sum_{i \in V_\ell, j \in O_\ell} \beta_{ij}^{(k)} x_i x_j + \sum_{i \in V_\ell \cup O_\ell} \gamma_i^{(k)} x_i + \delta^{(k)}, \quad (1) \end{aligned}$$

  where $\ell \in \{1, \ldots, u\}$ is the only integer such that $k \in O_\ell$ (see above). The coefficients $\alpha_{ij}^{(k)}$, $\beta_{ij}^{(k)}$, $\gamma_i^{(k)}$ and $\delta^{(k)}$ are randomly chosen $\mathbb{F}$ elements.

The size of the private key is

$$\underbrace{m \cdot (m+1)}_{\text{affine map } \mathcal{S}} + \underbrace{n \cdot (n+1)}_{\text{affine map } \mathcal{T}} + \underbrace{\sum_{i=1}^{u} \left(\frac{v_i \cdot (v_i + 1)}{2} + v_i \cdot o_i + v_i + o_i + 1 \right)}_{\text{central map } \mathcal{F}}$$

field elements.

Public Key. The *public key* is the composed map

$$\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$$

and therefore consists of m quadratic polynomials in the ring $\mathbb{F}[x_1, \ldots, x_n]$.
The size of the public key is

$$m \cdot \frac{(n+1) \cdot (n+2)}{2}$$

field elements.

1.3 Signature Generation

Given a document d to be signed, one uses a hash function $\mathcal{H} : \{0,1\} \to \mathbb{F}^m$ to compute the hash value $\mathbf{h} = \mathcal{H}(d) \in \mathbb{F}^m$. A signature $\mathbf{z} \in \mathbb{F}^n$ of the document d is then computed as follows.

1. Compute $\mathbf{x} = \mathcal{S}^{-1}(\mathbf{h}) \in \mathbb{F}^m$.
2. Compute a pre-image $\mathbf{y} \in \mathbb{F}^n$ of $\mathbf{x}$ under the central map $\mathcal{F}$. This is done as shown in Algorithm 1.
3. Compute the signature $\mathbf{z} \in \mathbb{F}^n$ by $\mathbf{z} = \mathcal{T}^{-1}(\mathbf{y})$.

Algorithm 1 Inversion of the Rainbow central map

Input: Rainbow central map $\mathcal{F} = (f^{(v_1+1)}, \ldots, f^{(n)})$, vector $\mathbf{x} \in \mathbb{F}^m$.
Output: vector $\mathbf{y} \in \mathbb{F}^n$ with $\mathcal{F}(\mathbf{y}) = \mathbf{x}$.

Choose random values for the variables $y_1, \ldots, y_{v_1}$ and substitute these values into the polynomials $f^{(i)}$ $(i = v_1 + 1, \ldots, n)$.
for $\ell = 1$ to u **do**
    Perform Gaussian Elimination on the polynomials $f^{(i)}$ $(i \in O_\ell)$ to get the values of the variables y_i $(i \in O_\ell)$.
    Substitute the values of y_i $(i \in O_\ell)$ into the polynomials $f^{(i)}$ $(i = v_{\ell+1} + 1, \ldots, n)$.
end for

1.4 Signature Verification

Given a document d and a signature $\mathbf{z} \in \mathbb{F}^n$, the authenticity of the signature is checked as follows.

1. Use the hash function $\mathcal{H}$ to compute the hash value $\mathbf{h} = \mathcal{H}(d) \in \mathbb{F}^m$.
2. Compute $\mathbf{h}' = \mathcal{P}(\mathbf{z}) \in \mathbb{F}^m$.

If $\mathbf{h}' = \mathbf{h}$ holds, the signature $\mathbf{z}$ is accepted, otherwise it is rejected.

The Rainbow signature scheme can be defined for any number of layers u. For $u = 1$ we obtain the well known UOV signature scheme. However, choosing $u = 2$ leads to a better scheme wth more efficient computations and smaller key sizes at the same level of security. Choosing $u > 2$ gives only a very small benefit in terms of performance, but needs larger keys to reach the same security level. Therefore, for ease of implementation, 2 is a common choice for the number of Rainbow layers. For ease of implementation and performance issues, it is further common to choose the size of the two Rainbow layers (i.e. the values of o_1 and o_2) to be equal. In our parameter recommendations, we follow these two

guidelines [1].

The following algorithms `RainbowKeyGen`, `RainbowSign` and `RainbowVer` illustrate the key generation, signature generation and signature verification processes of Rainbow in algorithmic form.

Algorithm 2 `RainbowKeyGen`: Key Generation of Rainbow

Input: Rainbow parameters (q, v_1, o_1, o_2)
Output: Rainbow key pair (sk, pk)

$m \leftarrow o_1 + o_2$
$n \leftarrow m + v_1$
repeat
    $M_S \leftarrow \texttt{Matrix}(q, m, m)$
until IsInvertible(M_S) == **TRUE**
$c_S \leftarrow_R \mathbb{F}^m$
$\mathcal{S} \leftarrow \texttt{Aff}(M_S, c_S)$
$InvS \leftarrow M_S^{-1}$
repeat
    $M_T \leftarrow \texttt{Matrix}(q, n, n)$
until IsInvertible(M_T) == **TRUE**
$c_T \leftarrow_R \mathbb{F}^n$
$\mathcal{T} \leftarrow \texttt{Aff}(M_T, c_T)$
$InvT \leftarrow M_T^{-1}$
$\mathcal{F} \leftarrow \texttt{Rainbowmap}(q, v_1, o_1, o_2)$
$\mathcal{P} \leftarrow \mathcal{S} \circ \mathcal{F} \circ \mathcal{T}$
sk $\leftarrow (InvS, c_S, \mathcal{F}, InvT, c_T)$
pk $\leftarrow \mathcal{P}$
return (sk, pk)

The possible input values of Algorithm `RainbowKeyGen` are specified in Section 1.8. $\texttt{Matrix}(q, m, n)$ returns an $m \times n$ matrix with coefficients chosen uniformly at random in $\mathbb{F}_q$, while $\texttt{Aff}(M, c)$ returns the affine map $M \cdot x + c$. $\texttt{Rainbowmap}(q, v_1, o_1, o_2)$ returns a Rainbow central map according to the parameters (q, v_1, o_1, o_2) (see equation (1)). The coefficients $\alpha_{ij}^{(k)}$, $\beta_{ij}^{(k)}$, $\gamma_i^{(k)}$ and $\delta^{(k)}$ are hereby chosen uniformly at random from $\mathbb{F}_q$.

Altogether, we need in `RainbowKeyGen` the following number of randomly chosen

[1] In the parameter set IIIb we slightly differ from this rule by choosing $o_2 > o_1$. The reason for this is that, in this case, an unbalanced design of the Rainbow layers provides higher security against quantum attacks.

$\mathbb{F}_q$-elements:

$$\underbrace{m \cdot (m+1)}_{\text{affine map } \mathcal{S}} + \underbrace{n \cdot (n+1)}_{\text{affine map } \mathcal{T}} + \underbrace{o_1 \cdot \left(\frac{v_1 \cdot (v_1+1)}{2} + v_1 \cdot o_1 + v_1 + o_1 + 1 \right)}_{\text{central polynomials of the first layer}}$$
$$+ \underbrace{o_2 \cdot \left(\frac{(v_1+o_1) \cdot (v_1+o_1+1)}{2} + (v_1+o_1) \cdot o_2 + v_1 + o_1 + o_2 + 1 \right)}_{\text{central polynomials of the second layer}}$$

Algorithm 3 `RainbowSign`: Signature generation process of Rainbow

Input: Rainbow private key $(InvS, c_S, \mathcal{F}, InvT, c_T)$, document d
Output: signature $\mathbf{z} \in \mathbb{F}^n$ such that $\mathcal{P}(\mathbf{z}) = \mathcal{H}(d)$

1: $\mathbf{h} \leftarrow \mathcal{H}(d)$
2: $\mathbf{x} \leftarrow InvS \cdot (\mathbf{h} - c_S)$
3: $\mathbf{y} \leftarrow \texttt{InvF}(\mathcal{F}, \mathbf{x})$
4: $\mathbf{z} \leftarrow InvT \cdot (\mathbf{y} - c_T)$
5: **return** $\mathbf{z}$

Algorithm 4 `InvF`: Inversion of the Rainbow central map

Input: Rainbow central map $\mathcal{F} = (f^{(v_1+1)}, \ldots, f^{(n)})$, vector $\mathbf{x} \in \mathbb{F}^m$.
Output: vector $\mathbf{y} \in \mathbb{F}^n$ with $\mathcal{F}(\mathbf{y}) = \mathbf{x}$.

1: **repeat**
2: $\quad y_1, \ldots, y_{v_1} \leftarrow_R \mathbb{F}$
3: $\quad \hat{f}^{(v_1+1)}, \ldots, \hat{f}^{(n)} \leftarrow f^{(v_1+1)}(y_1, \ldots, y_{v_1}), \ldots, f^{(n)}(y_1, \ldots, y_{v_1})$.
4: $\quad t, (y_{v_1+1}, \ldots, y_{v_2}) \leftarrow \texttt{Gauss}(\hat{f}^{(v_1+1)} = x_{v_1+1}, \ldots, \hat{f}^{(v_2)} = x_{v_2})$
5: $\quad$ **if** $t ==$ **TRUE then**
6: $\quad\quad \hat{f}^{(v_2+1)}, \ldots, \hat{f}^{(n)} \leftarrow \hat{f}^{(v_2+1)}(y_{v_1+1}, \ldots, y_{v_2}), \ldots, \hat{f}^{(n)}(y_{v_1+1}, \ldots, y_{v_2})$
7: $\quad\quad t, (y_{v_2+1}, \ldots, y_n) \leftarrow \texttt{Gauss}(\hat{f}^{(v_2+1)} = x_{v_2+1}, \ldots, \hat{f}^{(n)} = x_n)$
8: $\quad$ **end if**
9: **until** $t ==$ **TRUE**
10: **return** $\mathbf{y} = (y_1, \ldots, y_n)$

In Algorithm `InvF`, the function `Gauss` returns a binary value $t \in \{$**TRUE**, **FALSE**$\}$ indicating whether the given linear system is solvable, and if so a random solution of the system. In `InvF` we make use of at least v_1 random field elements (depending on how often we have to perform the loop to find a solution).

1.5 Changes needed to achieve EUF-CMA Security

The standard Rainbow signature scheme as described above provides only universal unforgeability. In order to obtain EUF-CMA security, we apply a transformation similar to that in [15]. The main difference is the use of a random

Algorithm 5 `RainbowVer`: Signature verification of Rainbow

Input: document d, signature $\mathbf{z} \in \mathbb{F}^n$
Output: **TRUE** or **FALSE**
1: $\mathbf{h} \leftarrow \mathcal{H}(d)$
2: $\mathbf{h}' \leftarrow \mathcal{P}(\mathbf{z})$
3: **if** h'==h **then**
4: **return TRUE**
5: **else**
6: **return FALSE**
7: **end if**

binary vector r called salt. Instead of generating a signature for $\mathbf{h} = \mathcal{H}(d)$ as in Algorithm `RainbowSign`, we generate a signature for $\mathcal{H}(\mathcal{H}(d)||r)$. The modified signature has the form $\sigma = (\mathbf{z}, r)$, where $\mathbf{z}$ is a standard Rainbow signature. By doing so, we ensure that an attacker is not able to forge any (hash value/ signature) pair. In particular, we apply the following changes to the algorithms `RainbowKeyGen`, `RainbowSign` and `Rainbowver`.

- In the algorithm `RainbowKeyGen`*, we choose an integer ℓ as the length of a random salt; ℓ is appended both to the private and the public key.
- In the algorithm `RainbowSign`*, we choose first randomly the values of the vinegar variables $\in F^{v_1}$; after that, we choose a random salt $r \in \{0,1\}^\ell$ and perfom the standard Rainbow signature generation process for $\mathbf{h} = \mathcal{H}(\mathcal{H}(d)||r)$ to obtain a signature $\sigma = (\mathbf{z}||r)$. If the linear system in step 2 of the signature generation process has no solutions, we choose a new value for the salt r and try again.
- The verification algorithm `RainbowVer`* returns **TRUE** if $\mathcal{P}(\mathbf{z}) = \mathcal{H}(\mathcal{H}(d)||r)$, and **FALSE** otherwise

Algorithms `RainbowKeyGen`*, `RainbowSign`* and `RainbowVer`* show the modified key generation, signing and verification algorithms.

Algorithm 6 `KeyGen`*: Modified Key Generation Algorithm for Rainbow

Input: Rainbow parameters (q, v_1, o_1, o_2), length ℓ of salt
Output: Rainbow key pair (sk, pk)
1: $pk, sk \leftarrow$ `RainbowKeyGen`(q, v_1, o_1, o_2)
2: $sk \leftarrow sk, \ell$
3: $pk \leftarrow pk, \ell$
4: **return** (sk, pk)

Algorithm 7 `RainbowSign`*: Modified signature generation process for Rainbow

Input: document d, Rainbow private key $(InvS, c_S, \mathcal{F}, InvT, c_T)$, length ℓ of the salt

Output: signature $\sigma = (\mathbf{z}, r) \in \mathbb{F}^n \times \{0,1\}^\ell$ such that $\mathcal{P}(\mathbf{z}) = \mathcal{H}(\mathcal{H}(d)||r)$

1: **repeat**
2: $y_1, \ldots, y_{v_1} \leftarrow_R \mathbb{F}$
3: $\hat{f}^{(v_1+1)}, \ldots, \hat{f}^{(n)} \leftarrow f^{(v_1+1)}(y_1, \ldots, y_{v_1}), \ldots, f^{(n)}(y_1, \ldots, y_{v_1})$
4: $(\hat{F}, c_F) \leftarrow \texttt{Aff}^{-1}(\hat{f}^{(v_1+1)}, \ldots, \hat{f}^{(n)})$
5: **until** IsInvertible$(\hat{F})$ == **TRUE**
6: $InvF = \hat{F}^{-1}$
7: **repeat**
8: $r \leftarrow \{0,1\}^\ell$
9: $\mathbf{h} \leftarrow \mathcal{H}(\mathcal{H}(d)||r)$
10: $\mathbf{x} \leftarrow InvS \cdot (\mathbf{h} - c_S)$
11: $(y_{v_1+1}, \ldots, y_{v_2}) \leftarrow InvF \cdot ((x_{v_1+1}, \ldots, x_{v_2}) - c_F)$
12: $\hat{f}^{(v_2+1)}, \ldots, \hat{f}^{(n)} \leftarrow \hat{f}^{(v_2+1)}(y_{v_1+1}, \ldots, y_{v_2}), \ldots, \hat{f}^{(n)}(y_{v_1+1}, \ldots, y_{v_2})$
13: $t, (y_{v_2+1}, \ldots, y_n) \leftarrow \text{Gauss}(\hat{f}^{(v_2+1)} = x_{v_2+1}, \ldots, \hat{f}^{(n)} = x_n)$
14: **until** t == **TRUE**
15: $\mathbf{z} = InvT \cdot (\mathbf{y} - c_T)$
16: $\sigma \leftarrow (\mathbf{z}, r)$
17: **return** σ

In Algorithm `RainbowSign`*, the function `Aff`$^{-1}$ takes as input an affine map $\mathcal{G} = M \cdot x + c$ and returns M and c.
Note that, in line 9 of the algorithm, we do not compute $\mathcal{H}(d||r)$, but $\mathcal{H}(\mathcal{H}(d)||r)$. In case we have to perform this step several times for a long message d, this improves the efficiency of our scheme significantly.

Algorithm 8 `RainbowVer`*: Modified signature verification process for Rainbow

Input: document d, signature $\sigma = (\mathbf{z}, r) \in \mathbb{F}^n \times \{0,1\}^\ell$

Output: boolean value **TRUE** or **FALSE**

1: $\mathbf{h} \leftarrow \mathcal{H}(\mathcal{H}(d)||r)$
2: $\mathbf{h}' \leftarrow \mathcal{P}(\mathbf{z})$
3: **if** h' == **h then**
4: **return TRUE**
5: **else**
6: **return FALSE**
7: **end if**

Similar to [15] we find that every attacker, who can break the EUF-CMA security of the modified scheme, can also break the standard Rainbow signature scheme.

In order to get a secure scheme, we have to ensure that no salt is used for more than one signature. Under the assumption of up to 2^{64} signatures being generated with the system [13], we choose the length of the salt r to be 128 bit (independent of the security level).

1.6 Note on the hash function

In our implementation we use SHA-2 as the underlying hash function. The SHA-2 hash function family contains the four hash functions SHA224, SHA256, SHA384 and SHA512 with output lengths of 224, 256, 384 and 512 bits respectively. In the Rainbow instances aimed at NIST security categories I and II (see Section 1.8), we use SHA256 as the underlying hash function. In the Rainbow instances for the security categories III/IV and V/VI, we use SHA384 and SHA512 respectively.

In case a slightly longer (non standard) hash output is needed for our scheme (Rainbow schemes over GF(31) and GF(256)), we proceed as follows. In order to obtain a hash value of length $256 < k < 384$ bit for a document d, we set

$$\mathcal{H}(d) = \text{SHA256}(d) \;\; || \;\; \text{SHA256}(\text{SHA256}(d))|_{1,\ldots,k-256}.$$

(analogously for hash values of length $384 < k < 512$ bits and $k > 512$ bits)
By doing so, we ensure that a collision attack against the hash function $\mathcal{H}$ is at least as hard as a collision attack against SHA256 (rsp. SHA384, SHA512).

1.7 Note on the generation of random field elements

During the key and signature generation of Rainbow, we make use of a large number of random field elements. These are obtained by calling a cryptographic random number generator such as that from the OpenSSL library. The random number generator used in our implementations is the `AES CTR_DRBG` function.In debug mode, our software can either generate the random bits and store them in a file, or read the required random bits from a file (for Known Answer Tests). In the case of Rainbow over GF(31), we use a sort of "rejection sampling". We first generate a random byte, corresponding to an integer $v \in \{0, \ldots, 255\}$. We reject v, if $v < 8$. Otherwise, the random GF(31) element v' is given by $v' = v \bmod 31 \in \{0, \ldots, 31\}$.

1.8 Parameter Choice

We propose the following nine parameter sets for Rainbow

Ia $(\mathbb{F}, v_1, o_1, o_2) = (\text{GF}(16), 32, 32, 32)$ (64 equations, 96 variables)

Ib $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(31), 36, 28, 28)$ (56 equations, 92 variables)

Ic $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 40, 24, 24)$ (48 equations, 88 variables)

IIIb $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(31), 64, 32, 48)$ (80 equations, 144 variables)

IIIc $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 68, 36, 36)$ (72 equations, 140 variables)

IVa $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(16), 56, 48, 48)$ (96 equations, 152 variables)

Vc $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 92, 48, 48)$ (96 equations, 188 variables)

VIa $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(16), 76, 64, 64)$ (128 equations, 204 variables)

VIb $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(31), 84, 56, 56)$ (112 equations, 196 variables)

The proposed parameter sets are denoted as follows: The roman number indicates the NIST security category which the given Rainbow instance aims at (see Section 5.2). The letter indicates the finite field used in the scheme (a for GF(16), b for GF(31) and c for GF(256)). Note that, in some cases, a given parameter set fulfills the requirements of more than one of the security categories. For example, the Rainbow instance Ib (using GF(31) as underlying field) was designed to meet the criteria of NIST security categories I and II.

Additionally, we give here four further parameter sets providing less security and demonstrating certain issues how the parameter choice affects the security of the scheme.

0a $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 40, 16, 16)$
For the parameter set 0a, the number of equations (given by $o_1 + o_2$) contained in the public key is not large enough to prevent direct attacks against the scheme. In fact, the complexity of this attack against the given Rainbow instance is only about 2^{109} classical gate equivalents.

0b $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 24, 24, 24)$
For the parameter set 0b, the value of v_1 is too small to prevent the Rainbow-Band-Separation attack. In fact, the complexity of this attack against the given Rainbow instance is only about 2^{119} classical gate equivalents.

0c $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 40, 36, 12)$
While, for the parameter set 0c, both the number of equations and variables are the same as for parameter set Ic, it provides significantly less security. The reason for this is the small value of o_2, which makes the scheme vulnerable to the HighRank attack. The security of this Rainbow instance is therefore only about 2^{120} classical gate equivalents.

0d $(\mathbb{F}, v_1, o_1, o_2) = (\mathrm{GF}(256), 40, 8, 40)$
While, for the parameter set 0d, both the number of equations and variables are the same as for parameter set Ic, it provides significantly less

security. The reason for this is the high value of o_2, which makes the scheme vulnerable to the UOV attack. The security of this Rainbow instance is therefore only about 2^{84} classical gate equivalents.

1.8.1 Data Conversion between bitstrings and GF(31) elements

When using Rainbow with GF(31) as the underlying finite field, we have to convert sequences of GF(31) elements into bitstrings and vice versa. During key generation, the public and private keys are generated as sequences of GF(31) elements and have to be converted into bitstrings for storage. In the signature generation process, we have to convert the private key and the hash value of the document (bitstrings) into sequences of GF(31) elements, generate the signature as a sequence of GF(31) elements and store it as a bitstring. During signature varification, we have to convert the public key, the hash value and the signature into sequences of GF(31) elements before running Algorithm 8. In order to perform this conversion, we proceed as follows.

To convert a sequence of GF(31) elements into a bitstring, we store 3 GF(31) elements in two bytes. In case the number of GF(31) elements is not divisible by three, we store each of the last k mod 3 GF(31) elements into 8 bits. Therefore, the length of the bitstring needed to store a key or signature of k GF(31) elements can be computed as

$$\text{length}_{\text{bitstring}} = (k \text{ div } 3) \cdot 16 + (k \text{ mod } 3) \cdot 8 \text{ bits.}$$

To compute the length of a hash value fitting into k GF(31) elements, we compute

$$\text{length}_{\text{hash value}} = (k \text{ div } 5) \cdot 24 + (k \text{ mod } 5) \cdot 4 \text{ bits.}$$

This uses the fact that 5 GF(31) elements can be effficiently used to store 3 bytes, while every additional GF(31) element can cover four bits.

2 Key Storage

2.1 Representation of Finte Field Elements

2.1.1 GF(31)

Elements of GF(31) are stored as integers in the range of 0 to 30. Any number out of this range, for example 31, is considered as a format error.
In order to reduce the key size, we apply the "packing operations" described in Section 1.8.1.

2.1.2 GF(16)

Elements of GF(2) are stored as one bit 0 or 1. Elements of GF(4) are stored in two bits as linear polynomials over GF(2). The constant term of the polynomial is hereby stored in the least significant bit. Elements of GF(16) are stored in 4 bits as linear polynomials over GF(4). The constant term of the polynomial is hereby stored in the 2 least significant bits. Two adjacent GF(16) elements are packed into one byte.

2.1.3 GF(256)

Elements of GF(256) are stored in one byte as linear polynomials over GF(16). The constant term of the polynomial is hereby stored in the 4 least significant bits.

2.2 Public Key

The public key $\mathcal{P}$ of Rainbow is a system of m multivariate quadratic poynomials in n variables (we write $\mathcal{P} := \mathcal{MQ}(m, n)$). The monomials are ordered in the graded-reverse-lexicographic order.

$$\begin{aligned}
y_1 &= q_{1,1,1}x_1x_1 + q_{2,1,1}x_2x_1 + q_{2,2,1}x_2x_2 + q_{3,1,1}x_3x_1 + q_{3,2,1}x_3x_2 + \cdots \\
&+ l_{1,1}x_1 + l_{2,1}x_2 + \cdots + l_{n,1}x_n + c_1 \\
y_2 &= q_{1,1,2}x_1x_1 + q_{2,1,2}x_2x_1 + q_{2,2,2}x_2x_2 + q_{3,1,2}x_3x_1 + q_{3,2,2}x_3x_2 + \cdots \\
&+ l_{1,2}x_1 + l_{2,2}x_2 + \cdots + l_{n,2}x_n + c_2 \\
&\vdots \\
y_m &= q_{1,1,m}x_1x_1 + q_{2,1,m}x_2x_1 + q_{2,2,m}x_2x_2 + q_{3,1,m}x_3x_1 + q_{3,2,m}x_3x_2 + \cdots \\
&+ l_{1,m}x_1 + l_{2,m}x_2 + \cdots + l_{n,m}x_n + c_m \qquad (2)
\end{aligned}$$

Hereby, $q_{i,j,k}$ is the coefficient of the quadratic monomial x_ix_j of the polynomial y_k, $l_{i,k}$ the coefficient of the linear monomial x_i in y_k and c_k the constant coefficient of y_k $(1 \leq j \leq i \leq n,\ 1 \leq k \leq m)$.
In the key file we store the coefficients of the linear terms first, followed by those of the quadratic and constant terms.

If we consider the indices of the coefficients $l_{i,k}$, $q_{i,j,k}$ and c_k as 2-, 3- and 1-digit numbers respectively, we order the coefficient with smaller indices in front. The coefficient sequence of the $\mathcal{MQ}(m,n)$ system (2) , is therefore stored (for the underlying fields GF(16) and GF(256)) in the form

$$[l_{1,1}, l_{1,2}, \ldots, l_{1,m}, l_{2,1}, \ldots, l_{n,m}, q_{1,1,1}, q_{1,1,2}, \ldots, q_{1,1,m}, q_{2,1,1}, \ldots, q_{n,n,m}, c_1, \ldots, c_m].$$

2.2.1 The case of GF(31)

For the underlying field GF(31), we store the coefficients of the public key in a slightly different order to improve the performance of the verification process. While we still store the coefficients of the linear terms in front of those of the quadratic and constant terms, the coefficients of the linear and quadratic blocks are stored in "two-column" manner. By doing so, the coefficient sequence of the $\mathcal{MQ}(m,n)$ system (2) has the form

$$\begin{aligned}[\quad & l_{1,1}, l_{2,1}, l_{1,2}, l_{2,2}, \ldots, l_{1,m}, l_{2,m}, l_{3,1}, l_{4,1} \ldots, l_{n-1,m}, l_{n,m}, \\ & q_{1,1,1}, q_{2,1,1}, q_{1,1,2}, q_{2,1,2}, \ldots, q_{1,1,m}, q_{2,1,m}, q_{3,1,1}, q_{4,1,1}, \ldots, q_{n-1,n,m}, q_{n,n,m}, \\ & c_1, c_2, \ldots, c_m].\end{aligned}$$

The reason for storing the public key in this form is explained in Section 3.2.1. Note that, in order to be stored in this way, the number of variables in the public key must be even. However, for our parameter sets (see Section 1.8), this is the case.
After having generated the coefficient sequence, we apply the "packing operations" of Section 1.8.1 to reduce the size of the public key.

2.3 Secret Key

The secret key comprises the three components $\mathcal{T}, \mathcal{S}$, and $\mathcal{F}$. These components are stored in the order $\mathcal{T}, \mathcal{S}$, and $\mathcal{F}$.

2.3.1 The affine maps $\mathcal{T}$ and $\mathcal{S}$

Suppose the affine map $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$ is given by

$$\mathcal{T}(\mathbf{x}) = \begin{bmatrix} t_{11} & t_{12} & \ldots & t_{1n} \\ & \vdots & \ddots & \\ t_{n1} & t_{n2} & \ldots & t_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} .$$

We store the matrix in column-major form. Hence, the affine map $\mathcal{T}$ is stored as a sequence

$$[t_{11}, t_{21}, \ldots, t_{n1}, t_{12}, \ldots, t_{nn}, c_1, , \ldots, c_n] .$$

The affine map $\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m$ is stored in the same manner.

2.3.2 The central map $\mathcal{F}$

The central map $\mathcal{F}$ consists of two layers of quadratic equations. Recall that $\mathcal{F} = (f^{(v_1+1)}(\mathbf{x}), \ldots, f^{(n)}(\mathbf{x}))$ and

$$f^{(k)}(\mathbf{x}) = \sum_{i,j \in V_\ell, i \le j} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \in V_\ell, j \in O_\ell} \beta_{ij}^{(k)} x_i x_j + \sum_{i \in V_\ell \cup O_\ell} \gamma_i^{(k)} x_i + \delta^{(k)},$$

where $\ell \in \{1, 2\}$ is again the only integer such that $k \in O_\ell$.

For the first layer we have $V_1 := \{1, \ldots, v_1\}$ and $O_1 := \{v_1 + 1, \ldots, v_1 + o_1\}$, for the second layer $V_2 := \{1, \ldots, v_2 = v_1 + o_1\}$ and $O_2 := \{v_2 + 1, \ldots, n = v_2 + o_2\}$. The two layers of the central map $\mathcal{F}$ are stored separately.

While storing the first layer of $\mathcal{F}$, the coefficients of the equations $f^{(v_1+1)}, \ldots, f^{(v_2)}$ are further divided into 3 parts denoted as "vv", "vo" , and "o-linear". They are stored in the secret key in the order "o-linear", followed by "vo" , and followed by "vv".

vv : The "vv" part is an $\mathcal{MQ}(o_1, v_1)$ system, whose components are of the form

$$\sum_{i,j \in V_1, i \le j} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \in V_1} \gamma_i^{(k)} x_i + \delta^{(k)} \quad \text{for } k \in O_1 \ .$$

It is stored in the same manner as the $\mathcal{MQ}(m, n)$ system of the public key (see Section 2.3). Note that the "vv" part contains, additionally to the coefficients of the quadratic $v \times v$ terms, the coefficients of the terms linear in the Vinegar variables and all the constant terms of the map $\mathcal{F}$.

vo : The "vo" part contains the remaining quadratic terms which are

$$\sum_{i \in V_1} \sum_{j \in O_1} \beta_{ij}^{(k)} x_i x_j = \begin{bmatrix} x_{v_1+1}, \ldots, x_{v_1+o_1} \end{bmatrix} \begin{bmatrix} \beta_{11}^{(k)} & \cdots & \beta_{v_1 1}^{(k)} \\ \vdots & \ddots & \\ \beta_{1 o_1}^{(k)} & \cdots & \beta_{v_1 o_1}^{(k)} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{v_1} \end{bmatrix} \quad \text{for } k \in O_1 \ .$$

The "vo" part is stored in the form of o_1 column-major $o_1 \times v_1$ matrices, yielding the sequence

$$[\beta_{11}^{(v_1+1)}, \ldots, \beta_{1 o_1}^{(v_1+1)}, \beta_{21}^{(v_1+1)}, \ldots, \beta_{v_1 o_1}^{(v_1+1)}, \beta_{11}^{(v_1+2)}, \ldots, \beta_{v_1 o_1}^{(v_1+o_1)}].$$

o-linear : This part contains the coefficients of the remaining linear terms in the oil variables $x_{v_1+1}, \ldots, x_{v_2}$ of the first layer. These terms are given by

$$\begin{bmatrix} f_{v_1+1}(\mathbf{x}) \\ \vdots \\ f_{v_2}(\mathbf{x}) \end{bmatrix} = \ldots + \begin{bmatrix} \gamma_{v_1+1}^{(v_1+1)} & \cdots & \gamma_{v_2}^{(v_1+1)} \\ \vdots & \ddots & \\ \gamma_{v_1+1}^{(v_2)} & \cdots & \gamma_{v_2}^{(v_2)} \end{bmatrix} \begin{bmatrix} x_{v_1+1} \\ \vdots \\ x_{v_2} \end{bmatrix} \ .$$

The "o-linear" part is stored in the form of a row-major matrix, yielding the sequence

$$[\gamma_{v_1+1}^{(v_1+1)}, \ldots, \gamma_{v_2}^{(v_1+1)}, \gamma_{v_1+1}^{(v_1+2)}, \ldots, \gamma_{v_2}^{(v_2)}].$$

The coefficients of the second Rainbow layer are stored in the same way.

2.3.3 The case of GF(31)

For the underlying field GF(31), the coefficient sequence of the "vv" terms is stored in the same "two-column" format as the public key (see Section 2.2.1). Furthermore, all the column-major matrices in $\mathcal{T}$ and $\mathcal{S}$ as well as the "vo" part of $\mathcal{F}$ are also stored in "two-column" form. For example, the coefficients of the map $\mathcal{T}$ are stored as the sequence

$$[t_{11}, t_{1,2}, t_{2,1}, t_{2,2}, t_{3,1}, \ldots, t_{n,2}, t_{1,3}, \ldots, t_{n,n-1}, t_{n,n}, c_1, \ldots, c_n].$$

Note again that we use here the fact that n is an even number. The row-major matrix in "o-linear" is stored as described in the previous section.
After having generated the coefficient sequence of the private key, we apply the "packing operations" of Section 1.8.1 to reduce the key size.

3 Implementation Details

3.1 Arithmetic over Finite Fields

For GF(31), the straightforward use of arithmetic operations of the AVX2 instruction set is possible. We use `(V)PMULHRSW` (packed multiply high rounded signed word) to take remainders by 31 which is faster than using shifts.

3.1.1 The case of GF(16)

For multiplications over GF(16), our general strategy is the use of `VPSHUFB/TBL` for multiplication tables. While multiplying a bunch $\mathbf{a}$ of GF(16) elements stored in an SIMD register with a scalar $b \in$ GF(16), we load the table of results of multiplication with b and follow with one `(V)PSHUFB` for the result $\mathbf{a} \cdot b$.

Time-Constancy issues: Addressing table entries is a side-channel leakage which reveals the value of b to a cache-time attack [4].
When time-constancy is needed, the straightforward method is again to use `VPSHUFB`. However, we do not use multpilication tables as above, but logarithm and exponentiation tables, and store the result in log-form if warranted. That is, we compute $a \cdot b = g^{(\log_g a + \log_g b)}$, and due to the characteristic of `(V)PSHUFB`, setting $\log_g 0 = -42$ is sufficient to make this operation time-constant even

when multiplying three elements.[2] We shall see a different method below when working on an constant-time evaluation of a multivariate quadratic system over GF(16) (in the following sections, we denote this task shortly by "Evaluation of $\mathcal{MQ}$").

3.1.2 The case of GF(256)

Multiplications over GF(256) can be implemented using 2 table lookup instructions in the mainstream Intel SIMD instruction set. One `(V)PSHUFB` is used for the lower 4 bits, the other one for the top 4 bits.

Time-Constancy issues: For time-constant multiplications, we adopt the *tower field* representation of GF(256) which considers an element in GF(256) as a degree-1 polynomial over GF(16). The sequence of tower fields from which we build GF(256) is the following:

$$\begin{aligned} \text{GF}(4) &:= \text{GF}(2)[e_1]/(e_1^2 + e_1 + 1), \\ \text{GF}(16) &:= \text{GF}(4)[e_2]/(e_2^2 + e_2 + e_1), \\ \text{GF}(256) &:= \text{GF}(16)[e_3]/(e_3^2 + e_3 + e_2 e_1) \ . \end{aligned}$$

Using this representation, we can build constant-time multiplications over GF(256) from the techniques of GF(16). A time-constant GF(256) multiplication costs about 3 GF(16) multplications for multiplying 2 degree-1 polynomials over GF(16) with the Karatsuba method and one extra table lookup instruction for reducing the degree-2 term.

3.2 The Public Map and Evaluation of $\mathcal{MQ}$

The public map of Rainbow is a straightforward evaluation of an $\mathcal{MQ}$ system.

3.2.1 Evaluation of $\mathcal{MQ}$ over GF(31)

The matrix-like coefficients of $\mathcal{P}$ are stored as 8-bit values because we heavily rely on the AVX2 instruction `VPMADDUBSW`. In one instruction, `VPMADDUBSW` computes two 8-bit SIMD multiplications and a 16-bit SIMD addition.
All these operations are time-constant.
In order to perform these operations efficiently, we store the coefficients of the public key as described in Section 2.2.1.
Because `VPMADDUBSW` takes both a signed and an unsigned operand, one of the matrix and the monomial vector must be stored as signed bytes and one as unsigned bytes. Since $64 \cdot 31 \cdot 15 = 29760 < 2^{15}$, we can handle two `YMM` registers full of monomials before performing reductions on each individual accumulator. During computation, field elements are expressed as signed 16-bit values. If $m = 64$, we require 1024 bits of storage for each vector, precisely fitting in four 256-bit SIMD (`YMM`) registers.

[2] Here, g is a generator of the multiplicative group GF(16)*.

To efficiently compute all polynomials for a given set of monomials, we keep all required data in registers and try to avoid register spilling throughout the computation, as much as possible.

3.2.2 Evaluation of $\mathcal{MQ}$ over GF(16) and GF(256)

For pure public-key operations, the multiplications over GF(16) can be done by simply (1) loading the multiplication tables (`multab`) by the value of the multiplier and (2) performing a `VPSHUFB` for 32 results simultaneously. The multiplications over GF(256) can be performed with the same technique via 2 `VPSHUFB` instructions, using the fact that one lookup covers 4 bits. Another trick is to multiply a vector of GF(16) elements by two GF(16) elements with one `VPSHUFB` since `VPSHUFB` can actually be seen as 2 independent `PSHUFB` instructions.

3.2.3 Constant-Time Evaluation of $\mathcal{MQ}$ over GF(16) and GF(256)

While time-constancy issues are not important for the public key operations, we have to consider this issue during the evaluation of the "vv" terms of the central map $\mathcal{F}$.

In order to achieve time-constancy, we have to avoid loading `multab` according to a secret index for preventing cache-time attacks. To do this, we "generate" the desired `multab` instead of "loading" it by a secret value. More precisely, when evaluating $\mathcal{MQ}$ with a vector $\mathbf{w} = (w_1, w_2, \ldots, w_n) \in \mathrm{GF}(16)^n$, we can achieve a time-constant evaluation if we already have the `multab` of $\mathbf{w}$, which is $(w_1 \cdot \texttt{0x0}, \ldots, w_1 \cdot \texttt{0xf}), \ldots, (w_n \cdot \texttt{0x0}, \ldots, w_n \cdot \texttt{0xf})$, in the registers. [3] In other words, instead of performing memory access indexed by a secret value, we perform a sequential memory access indexed by the index of variables to prevent revealing side-channel information.

We show the generation of `multab` for elements $\mathbf{w} \in \mathrm{GF}(16)$ in Figure 1. A further matrix-transposition-like operation is needed to generate the desired `multab`. The reason for this is that the initial byte from each register forms our first new table, corresponding to w_1, the second byte from each register is the table of multiplication by w_2, etc. Computing one of these tables costs 16 calls of `PSHUFB` and we can generate 16 or 32 tables simultaneously using the SIMD environment. The amortized cost for generating one `multab` is therefore 1 `PSHUFB` plus some data movements.

As a result, the constant-time evaluation of $\mathcal{MQ}$ over GF(16) or GF(256) is only slightly slower than the non-constant time version.

[3] Note here and in the following. If we have a natural basis $(b_0 = 1, b_1, \ldots)$ of a binary field GF(q), we represent b_j by 2^j for convenience. So b_1 is 2, $1 + b_1$ is 3, ..., $1 + b_1 + b_2 + b_3$ is `0xF` for elements of GF(16), and analogously for larger fields; using the same method, the AES field representation of GF(2^8) is called `0x11B` because it uses $x^8 + x^4 + x^3 + x + 1$ as irreducible polynomial.

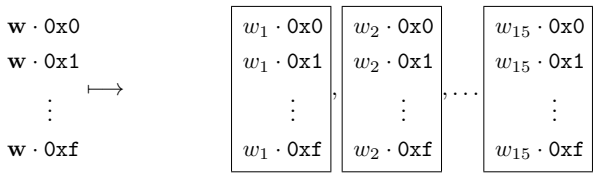


Figure 1: Generating `multab` for $\mathbf{w} = (w_1, w_2, \ldots w_{16})$. After $\mathbf{w} \cdot$ `0x0`, $\mathbf{w} \cdot$ `0x1`, ..., $\mathbf{w} \cdot$ `0xf` are calculated, each row stores the results of multiplications and the columns are the `multab` corresponding to $w_1, w_2, \ldots, w_{15}$. The `multab` of w_1, w_2, ...,w_{15} can be generated by collecting data in columns.

3.3 Gaussian Elimination in Constant Time

We use constant-time Gaussian elimination in the signing process of Rainbow. Constant-time Gaussian elimination was originally presented in [1] for GF(2) matrices and we extend the method to other finite fields. The problem of eliminations is that the pivot may be zero and one has to swap rows with zero pivots with other rows, which reveals side-channel information. To test pivots against zero and switch rows in constant time, we can use the current pivot as a predicate for conditional moves and switch with every possible row which can possibly contain non-zero leading terms. This constant-time Gaussian elimination is slower than a straigtforward Gaussian elimination (see Table 1), but is still an $O(n^3)$ operation.

Table 1: Benchmarks on solving linear systems with Gauss elimination on Intel XEON E3-1245 v3 @ 3.40GHz, in CPU cycles.

system	plain elimination	constant version
GF(16), 32×32	6,610	9,539
GF(31), 28×28	7,889	10,227
GF(256), 20×20	4,702	9,901

4 Performance Analysis

4.1 Key and Signature Sizes

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	public key size (kB)	private key size (kB)	hash size (bit)	signature size (bit) [1]
Ia	(GF(16),32,32,32)	148.5	97.9	256	512
Ib	(GF(31),36,28,28)	148.3	103.7	268	624
Ic	(GF(256),40,24,24)	187.7	140.0	384	832
IIIb	(GF(31),64,32,48)	512.1	371.4	384	896
IIIc	(GF(256),68,36,36)	703.9	525.2	576	1,248
IVa	(GF(16),56,48,48)	552.2	367.3	384	736
Vc	(GF(256), 92,48,48)	1,683.3	1,244.4	768	1,632
VIa	(GF(16), 76,64,64)	1,319.7	871.2	512	944
VIb	GF(31), 84,56,56)	1,321.0	922.4	536	1,176

[1] 128 bit salt included

Table 2: Key and Signature Sizes for Rainbow

4.2 Performance on the NIST Reference Platform

Processor: Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz (Skylake)
Clock Speed: 3.30GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
No use of special processor instructions

parameter set		key gen.	sign. gen.	sign. verif.
Ia	cycles	1,302M	601k	350k
	time (ms)	394	0.182	0.106
	memory	3.3MB	3.0MB	2.6MB
Ib	cycles	4,578M	2,044k	1,944k
	time (ms)	1,387	0.619	0.589
	memory	3.6MB	3.3MB	2.9MB
Ic	cycles	4,089M	1,521k	939k
	time (ms)	1,239	0.461	0.285
	memory	3.3MB	3.0MB	2.8MB
IIIb	cycles	26,172M	5,471k	4,908k
	time (ms)	7,931	1.658	1.487
	memory	5.7MB	3.6MB	3.9MB
IIIc	cycles	31,612M	4,047k	2,974k
	time (ms)	9,579	1.226	0.901
	memory	4.6MB	2.9MB	3.1MB
IVa	cycles	11,176M	1,823k	1,241k
	time (ms)	3,387	0.552	0.376
	memory	4.3MB	3.0MB	2.8MB
Vc	cycles	116,046M	8,688k	6,174k
	time (ms)	35,165	2.633	1.871
	memory	7.0MB	3.7MB	3.9MB
VIa	cycles	45,064M	3,916k	2,897k
	time (ms)	13,655	1.187	0.878
	memory	6.1MB	3.8MB	3.8MB
VIb	cycles	164,689M	16,755k	11,224k
	time (ms)	49,906	5.077	3.401
	memory	10.3MB	4.4MB	6.0MB

Table 3: Performance of Rainbow on the NIST Refernce Platform (Linux/Skylake)

4.3 Performance on Other Platforms

Processor: Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz (Skylake)
Clock Speed: 3.30GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
Use of AVX2 vector instructions

parameter set		key gen.	sign. gen.	sign. verif.
Ia	cycles	1,081M	75.5 k	25.5 k
	time (ms)	328	0.023	0.008
	memory	3.0MB	3.0MB	2.8MB
Ib	cycles	141M	426k	496k
	time (ms)	42.83	0.129	0.15
	memory	3.6MB	3.2MB	2.9MB
Ic	cycles	183M	111k	57.5k
	time (ms)	55.4	0.034	0.017
	memory	3.3MB	3.0MB	2.8MB
IIIb	cycles	813M	1,469k	1,791k
	time (ms)	246	0.445	0.543
	memory	5.9MB	4.1MB	4.1MB
IIIc	cycles	1,430M	326k	275k
	time (ms)	433	0.099	0.083
	memory	4.6MB	3.5MB	3.3MB
IVa	cycles	8,673M	899k	181k
	time (ms)	2,628	0.272	0.055
	memory	4.1MB	3.3MB	3.2MB
Vc	cycles	4,633M	616k	472k
	time (ms)	1,404	0.187	0.143
	memory	7.0MB	4.2MB	4.5MB
VIa	cycles	6,689M	575k	367k
	time (ms)	2,027	0.174	0.111
	memory	6.1MB	3.8MB	3.8MB
VIb	cycles	3,518M	3,655k	4690k
	time (ms)	1,066	1.108	1.421
	memory	10.0MB	5.3MB	6.0MB

Table 4: Performance of Rainbow on Linux/Skylake (AVX2)

Processor: Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz (Broadwell)
Clock Speed: 2.1GHz
Memory: 64GB (4x16) ECC DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
Operating System: Linux 4.8.5, GCC compiler version 6.4
Use of AVX2 vector instructions

parameter set		key gen.	sign. gen.	sign. verif.
Ia	cycles	1,147M	80.1k	27.0k
	time (ms)	546	0.038	0.013
	memory	3.0MB	2.9MB	2.7MB
Ib	cycles	150M	452k	526k
	time (ms)	71.4	0.216	0.250
	memory	3.7MB	3.2MB	2.8MB
Ic	cycles	194M	118k	61k
	time (ms)	92	0.056	0.029
	memory	3.1MB	3.2MB	2.7MB
IIIb	cycles	899M	1,282k	1,790k
	time (ms)	428	0.714	0.852
	memory	5.8MB	3.9MB	3.9MB
IIIc	cycles	1,535M	338k	274k
	time (ms)	736	0.161	0.130
	memory	4.6MB	3.4MB	3.3MB
IVa	cycles	9,161M	954k	212k
	time (ms)	3,464	0.454	0.101
	memory	4.2MB	3.2MB	3.1MB
Vc	cycles	5,019M	662k	472k
	time (ms)	2,390	0.315	0.225
	memory	6.9MB	4.1MB	4.2MB
VIa	cycles	45,182M	3,292k	2,899k
	time (ms)	13,882	0.998	0.878
	memory	6.7M	3.1M	3.5M
VIb	cycles	3,651M	3,697k	4,647k
	time (ms)	1,738	1.761	2.213
	memory	10.0MB	5.2MB	6.0MB

Table 5: Performance of Rainbow on Linux/Broadwell (AVX2)

4.4 Note on the measurements

Turboboost is disabled on our platforms. The main compilation flags are `gcc -O2 -std=c99 -Wall -Wextra (-mavx2)`. The used memory is measured during an actual run using `/usr/bin/time -f "%M"` (average of 10 runs). For key generation we take the average of 10 runs; for signature generation and verifi-

cation the average of 500 runs. As expected, Skylake is superior to Broadwell (which is almost the same as Haswell) in almost all cases.

4.5 Note on Rainbow schemes over GF(31)

For "b" (GF(31)) parameters, both the public map and the private map are on the order of a few cycles per byte of key, which represents an $\sim 1.5\times$ slowdown from GF(31) MQ evaluation in the literature. The main reason for this is that all keys are packed as mentioned in the implementation section. If we verify or (more likely) sign multiple times using the same key, this time goes down.

4.6 Trends as the number n of variables increases

Signing: The secret map involves Gaussian Elimination and time-constant MQ evaluation. Both are $O(n^3)$ operations.

Verification: The public map involves straightforward MQ evaluations and in the case of "b" (GF(31)) parameters unpacking. These are $O(n^3)$ operations (note: the public key size is also n^3).

Key Generation: Key generation is done via the standard method (interpolation) which is of order $O(n^5)$, The size of the resulting public key is $O(n^3)$.

These theoretical estimations match very well the above experimental data (when looking at Rainbow instances over the same base field).

5 Expected Security Strength

The following table gives an overview over the 6 NIST security categories proposed in [13]. The three values for the number of quantum gates correspond to values of the parameter `MAXDEPTH` of 2^{40}, 2^{64} and 2^{96}.

category	$\log_2$ classical gates	$\log_2$ quantum gates
I	143	130 / 106 / 74
II	146	
III	207	193 / 169 / 137
IV	210	
V	272	258 / 234 / 202
VI	274	

Table 6: NIST security categories

All known attacks against Rainbow are basically classical attacks, some of which can be sped up by Grover's algorithm. Due to the long serial computation of Grover's algorithm and the large memory consumption of the attacks, we feel safe in choosing a value of `MAXDEPTH` between 2^{64} and 2^{96}.

5.1 General Remarks

The Rainbow signature scheme as described in Section 1.5 of this proposal fulfills the requirements of the EUF-CMA security model (existential unforeability under chosen message attacks). The parameters of the scheme (in particular the length of the random salt) are chosen in a way that up to 2^{64} messages can be signed with one key pair. The scheme can generate signatures for messages of arbitrary length (as long as the underlying hash function can process them).

5.2 Practical Security

In this section we analyze the security offered by the parameter sets proposed in Section 1.8.

Since there is no proof for the practical security of Rainbow, we choose the parameters of the scheme in such a way that the complexities of the known attacks against the scheme (see Section 6) are beyond the required levels of security.

The formulas in Section 6 give complexity estimates for the attacks in terms of field multiplications. To translate these complexities into gate counts as proposed in the NIST specification, we assume the following.

- one field multiplication in the field GF(q) takes about $\log_2(q)^2$ bit multiplications (AND gates) and the same number of additions (XOR gates).

- for each field multiplication performed in the process of the attack, we also need an addition of field elements. Each of these additions costs $\log_2(q)$ bit additions (XOR).

Therefore, the number of gates required by an attack can be computed as

$$\#\text{gates} = \#\text{field multiplications} \cdot (2 \cdot \log_2(q)^2 + \log_2(q)).$$

The following tables show the security provided by the proposed Rainbow instances against

- direct attacks (Section 6.2)
- the MinRank attack (Section 6.3)
- the HighRank attack (Section 6.4)
- the UOV attack (Section 6.5) and
- the Rainbow Band Separation (RBS) attack (Section 6.6).

While the direct attack is a signature forgery attack, which has to be performed for each message separately, the MinRank, HighRank, UOV and RBS attack are key recovery attacks. After having recovered the Rainbow private key using one of these attacks, an adversary can generate signatures in the same way as a legitimate user.

For each parameter set and each attack, the first entry in the cell shows (the base 2 logarithm of) the number of classical gates, while the second entry (if available) shows (the base 2 logarithm of) the number of logical quantum gates needed to perform the attack. In each row, the value printed in bold shows the complexity of the best attack against the given Rainbow instance.

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	$\log_2$(#gates) direct	MinRank	HighRank	UOV	RBS
Ia	(GF(16),32,32,32)	164.5	161.3	150.3	149.2	**145.0**
		146.5	95.3	**86.3**	87.2	145.0
Ib	(GF(31),36,28,28)	160.4	212.9	161.5	198.4	**148.1**
		129.3	121.2	**92.1**	111.7	145.6
Ic	(GF(256),40,24,24)	151.6	358.5	215.9	337.4	**148.2**
		130.8	194.5	**119.9**	181.4	148.2

A collision attack against the hash function underlying the Rainbow instances Ia, Ib and Ic is at least as hard as a collision attack against SHA256 (see Section 1.6). Therefore, the three Rainbow instances Ia, Ib and Ic meet the requirements of security category I. The Rainbow instances Ib and Ic also meet the requirements of security category II.

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	$\log_2$(#gates) direct	MinRank	HighRank	UOV	RBS
IIIb	(GF(31),64,32,48)	**214.4**	354.0	262.5	260.9	216.9
		179.4	193.0	**143.6**	144.5	214.5
IIIc	(GF(256),68,36,36)	215.2	585.1	313.9	563.8	**217.4**
		183.5	309.1	**169.9**	295.8	217.4

A collision attack against the hash functions underlying the Rainbow instances IIIb and IIIc is at least as hard as a collision attack against SHA384. Therefore, the Rainbow instances IIIb and IIIc meet the requirements of the security categories III and IV.

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	$\log_2$(#gates) direct	MinRank	HighRank	UOV	RBS
IVa	(GF(16),56,48,48)	233.4	259.9	216.3	247.5	**215.5**

A collision attack against the hash function underlying the Rainbow instance IVa is at least as hard as a collision attack against SHA384. Therefore, the Rainbow instance IVa meets the requirements of security category IV.

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	$\log_2$(#gates) direct	MinRank	HighRank	UOV	RBS
Vc	(GF(256),92,48,48)	**275.4**	778.8	411.2	747.4	278.6
		235.5	406.8	**219.2**	393.4	278.6

A collision attack against the hash functions underlying the Rainbow instance Vc is at least as hard as a collision attack against SHA512. Therefore, the Rainbow instance Vc meets the requirements of the security categories V and VI.

parameter set	parameters $(\mathbb{F}, v_1, o_1, o_2)$	$\log_2$(#gates) direct	MinRank	HighRank	UOV	RBS
VIa	(GF(16),76,64,64)	302.6	341.6	281.6	329.2	**278.1**
VIb	(GF(31),84,56,56)	289.9	454.9	303.5	440.2	**279.8**

A collision attack against the hash functions underlying the Rainbow instances VIa and VIb is at least as hard as a collision attack against SHA512. Therefore, the Rainbow instances VIa and VIb meet the requirements of the security category VI.

5.2.1 Overview

The following table gives an overview of the security provided by our Rainbow instances. For each of the 6 NIST security categories [13] it lists the proposed

Rainbow instances meeting the corresponding requirements.

security category	GF(16)	GF(31)	GF(256)
I	Ia	Ib	Ic
II	-	Ib	Ic
III	-	IIIb	IIIc
IV	IVa	IIIb	IIIc
V	-	-	Vc
VI	VIa	VIb	VIc

Table 7: Proposed Rainbow Instances and their Security Categories

As can be seen from the table, we have, for each of the 6 NIST security categories, one Rainbow instance over the field GF(256). Over the smaller fields GF(16) and GF(31), some of the security categories (especially the quantum ones I, III and V) are not occupied. The reason for this are the large parameters needed to prevent in particular the quantum HighRank attack, which make these schemes inefficient.

5.3 Side Channel Resistance

In our implementation of the Rainbow signature scheme (see Section 3) all key dependent operations are performed in a time-constant manner. Therefore, our implementation is immune against timing attacks.

6 Analysis of Known Attacks

Known attacks against the Rainbow signature scheme include

- collision attacks against the hash function (Section 6.1)
- direct attacks (Section 6.2)
- the MinRank attack (Section 6.3)
- the HighRank attack (Section 6.4)
- The Rainbow-Band-Separation (RBS) atttack (Section 6.6)
- The UOV attack (Section 6.5)

In the presence of quantum computers, one also has to consider brute force attacks accelerated by Grover's algorithm (see Section 6.7).
While direct and brute force attacks are signature forgery attacks, which have to be performed for every message separately, rank attacks as well as the RBS and UOV attack are key recovery attacks. After having recovered the Rainbow private key using one of these attacks, the attacker can generate signatures in the same way as a legitimate user.

6.1 Collision attacks against the hash function

Since the Rainbow signature scheme follows the Hash then Sign approach, it can be attacked by finding collisions of the used hash function. We do not consider specific attacks against hash functions here, but consider the used hash function $\mathcal{H}$ as a perfect random function $\mathcal{H} : \{0,1\}^\star \to \mathbb{F}^m$.
Therefore, in order to prevent a (classical) collision attack against the hash function used in the Rainbow scheme, the number m of equations in the public system of Rainbow must be chosen such that

$$m \cdot \log_2 q \geq \text{seclev},$$

where q is the cardinality of the finite field and seclev is the required level of security. In other words, in order to prevent collision attacks against the used hash function, the number m of equations in the public key (and central map) of Rainbow must be chosen to be at least

$$m \geq \frac{2 \cdot \text{seclev}}{\log_2 q}.$$

By this choice of m, we ensure that a (classical) collision attack against the hash function used in the Rainbow scheme requires at least 2^{seclev} evaluations of the hash function $\mathcal{H}$.

6.2 Direct Attacks

The most straightforward attack against multivariate schemes such as Rainbow is the direct algebraic attack, in which the public equation $\mathcal{P}(\mathbf{z}) = \mathbf{h}$ is considered as an instance of the MQ-Problem. Since the public system of Rainbow is an underdetermined system with $n \approx 1.5 \cdot m$, the most efficient way to solve this equation is to fix $n - m$ variables to create a determined system before applying an algorithm such as XL or a Gröbner Basis technique such as F_4 or F_5 [10]. It can be expected that the resulting determined system has exactly one solution. In some cases one obtains even better results when guessing additional variables before solving the system (hybrid approach) [2]. The complexity of solving such a system of m quadratic equations in m variables using an XL Wiedemann approach can be estimated as

$$\text{Complexity}_{\text{direct; classical}} = \min_k \left(q^k \cdot 3 \cdot \binom{m-k+d_{\text{reg}}}{d_{\text{reg}}}^2 \cdot \binom{m-k}{2} \right)$$

field multiplications, where d_{reg} is the so called degree of regularity of the system. As it was shown by experiments, the public systems of Rainbow behave very similar to random systems. We therefore can estimate the degree of regularity as the smallest integer d for which the coefficient of t^d in

$$\frac{(1-t^2)^m}{(1-t)^{m-k}}$$

is non-positive.

In the presence of quantum computers, the additional guessing step of the hybrid approach might be sped up by Grover's algorithm. By doing so, we can estimate the complexity of a quantum direct attack by

$$\text{Complexity}_{\text{direct; quantum}} = \min_k \left(q^{k/2} \cdot 3 \cdot \binom{m-k+d_{\text{reg}}}{d_{\text{reg}}}^2 \cdot \binom{m-k}{2} \right)$$

field multiplications. Here, the value of d_{reg} can be estimated as above.

6.3 The MinRank Attack

In the **MinRank** attack [3] the attacker tries to find a linear combination of the public polynomials of minimal rank. In the case of Rainbow, such a linear combination of rank v_2 corresponds to a linear combination of the central polynomials of the first layer. By finding o_1 of these low rank linear combinations, it is therefore possible to identify the central polynomials of the first layer and to recover an equivalent Rainbow private key. As shown by Billet et al. [3], this step can be performed by

$$\text{Complexity}_{\text{MinRank; classical}} = q^{v_1+1} \cdot m \cdot \left(\frac{n^3}{3} - \frac{m^2}{6} \right) \tag{3}$$

field multiplications.

By the use of Grover's algorithm in the searching step, we can reduce this complexity to

$$\text{Complexity}_{\text{MinRank; quantum}} = q^{\frac{v_1+1}{2}} \cdot m \cdot \left(\frac{n^3}{3} - \frac{m^2}{6}\right) \tag{4}$$

field multiplications.

There exists an alternative formulation of the MinRank attack, the so called Minors Modelling. In this formulation, the MinRank problem is solved by solving a system of nonlinear polynomial equations (given by the $v_2 + 1$ minors of the matrix representing the required linear combination). The complexity of this attack can be estimated as

$$\text{Complexity}_{\text{MinRank; Minors}} = \binom{n + v_2 + 1}{v_2 + 1}^{\omega},$$

where $2 < \omega \leq 3$ is the linear algebra constant of solving a system of linear equations.
However, in the case of Rainbow, this complexity is higher than that of the MinRank attack using linear algebra techniques (see equation (3)). Furthermore, since we deal with a highly overdetermined system here, the MinRank attack using Minors Modelling can not be sped up by quantum techniques.

When analyzing the security of our Rainbow instances (see Section 5.2), we therefore use equations (3) and (4) to estimate the complexity of the MinRank attack.

6.4 The HighRank attack

The goal of the **HighRank** attack [5] is to identify the (linear representation of the) variables appearing the lowest number of times in the central polynomials (these correspond to the Oil-variables of the last Rainbow layer, i.e. the variables x_i with $i \in O_u$). The complexity of this attack can be estimated as

$$\text{Complexity}_{\text{HighRank; classical}} = q^{o_u} \cdot \frac{n^3}{6}.$$

In the presence of quantum computers, we can speed up the searching step using Grover's algorithm. Such we get

$$\text{Complexity}_{\text{HighRank; quantum}} = q^{o_u/2} \cdot \frac{n^3}{6}.$$

field multiplications.

6.5 UOV - Attack

Since Rainbow can be viewed as an extension of the well known Oil and Vinegar signature scheme [11], it can be attacked using the UOV attack of Kipnis and Shamir [12].
One considers Rainbow as an UOV instance with $v = v_1 + o_1$ and $o = o_2$. The goal of this attack is to find the pre-image of the so called Oil subspace $\mathcal{O}$ under the affine transfomation $\mathcal{T}$, where $\mathcal{O} = \{\mathbf{x} \in \mathbb{F}^n : x_1 = \cdots = x_v = 0\}$. Finding this space allows to separate the oil from the vinegar variables and recovering the private key.
The complexity of this attack can be estimated as

$$\text{Complexity}_{\text{UOV}-\text{Attack; classical}} = q^{n-2o_2-1} \cdot o_2^4$$

field multiplications.
Using Grover's algorithm, this complexity might be reduced to

$$\text{Complexity}_{\text{UOV}-\text{Attack; quantum}} = q^{\frac{n-2o_2-1}{2}} \cdot o_2^4$$

field multiplications.

6.6 Rainbow-Band-Separation Attack

The Rainbow-Band-Separation attack [8] aims at finding linear transformations $\mathcal{S}$ and $\mathcal{T}$ transforming the public polynomials into polynomials of the Rainbow form (i.e. Oil $\times$ Oil terms must be zero). To do this, the attacker has to solve several nonlinear multivariate systems. The complexity of this step is determined by the complexity of solving the first (and largest) of these systems, which consists of $n + m - 1$ quadratic equations in n variables. Since this is an overdetermined system (more equations than variables), we usually do not achieve a speed up by guessing variables before applying an algorithm like XL. However, in order to be complete, we consider the hybrid approach in our complexity estimate. Such we get

$$\text{Complexity}_{\text{RBS; classical}} = \min_k \cdot q^k \cdot 3 \cdot \binom{n + d_{\text{reg}} - k}{d_{\text{reg}}}^2 \cdot \binom{n-k}{2}$$

field multiplications. Again, the multivariate quadratic systems generated by this attack behave much like random systems. We can therefore estimate the value of d_{reg} as the smallest integer d, for which the coefficient of t^d in

$$\frac{(1-t^2)^{m+n-1}}{(1-t)^{n-k}}$$

is non-positive.

By using Grover's algorithm, we can speed up the guessing step of the hybrid approach. By doing so, we get

$$\text{Complexity}_{\text{RBS; quantum}} = \min_k \cdot q^{k/2} \cdot 3 \cdot \binom{n + d_{\text{reg}} - k}{d_{\text{reg}}}^2 \cdot \binom{n-k}{2}$$

field multiplications. The value of d_{reg} can be estimated as above.
However, as the optimal number k of variables to be guessed during the attack is very small (in most cases it is 0), the impact of quantum speed up on the complexity of the Rainbow-Band-Separation attack is quite limited.

6.7 Quantum Brute-Force-Attacks

In the presence of quantum computers, a brute force attack against the scheme can be sped up drastically using Grover's algorithm. For example, in [16] it was shown that a binary system of m equations in m variables can be solved using

$$2^{m/2} \cdot 2 \cdot m^3$$

bit operations. In general, we expect due to Grover's algorithm a quadratic speed up of a brute force attack. To reach a security level of seclev bits, we therefore need at least

$$m \geq \frac{2 \cdot \text{seclev}}{\log_2 q}$$

equations.
However, this condition is already needed to prevent collision attacks against the hash function. Therefore, we do not consider quantum brute force attacks in the parameter choice of our Rainbow instances.

7 Advantages and Limitations

The main advantages of the Rainbow signature scheme are

- **Efficiency**. The signature generation process of Rainbow consists of simple linear algebra operations such as matrix vector multiplication and solving linear systems over small finite fields. Therefore, the Rainbow scheme can be implemented very efficiently and is one of the fastest available signature schemes [9].
- **Short signatures**. The signatures produced by the Rainbow signature scheme are of size only a few hundred bits and therefore much shorter than those of RSA and other post-quantum signature schemes (see Section 4.1).
- **Modest computational requirements**. Since Rainbow only requires simple linear algebra operations over a small finite field, it can be efficiently implemented on low cost devices, without the need of a cryptographic coprocessor [6].
- **Security**. Though there does not exist a formal security proof which connects the security of Rainbow to a hard mathematical problem such as MQ, we are quite confident in the security of our scheme. Rainbow is based on the well known UOV signature scheme, against which, since its invention in 1999, no attack has been found. Rainbow itself was proposed in 2005, and the last attack requiring a parameter change was found in 2008 (ten years ago). Since then, despite of rigorous cryptanalysis, no attack against Rainbow has been developed. We furthermore note here that, in contrast to some other post-quantum schemes, the theoretical complexities of the known attacks against Rainbow match very well the experimental data. So, all in all, we are quite confident in the security of the Rainbow signature scheme.
- **Simplicity.** The design of the Rainbow schemes is extremely simple. Therefore, it requires only minimum knowledge in algebra to understand and implement the scheme. This simplicity also implies that there are not many structures of the scheme which could be utilized to attack the scheme. Therefore it is very unlikely that there are additional structures that can be used to attack the scheme which have not been discovered during more than 12 years of rigorous cryptanalysis.

On the other hand, the main disadvantage of Rainbow is the **large size of the public and private keys**. The (public and private) key sizes of Rainbow are, for security levels beyond 128 bit, in the range of 100 kB-1 MB and therefore much larger than those of classical schemes such as RSA and ECC and some other post-quantum schemes. However, due to increasing memory capabilities even of medium devices (e.g. smartphones), we do not think that this will be a major problem. Furthermore, we would like to point out that there exists a

technique to reduce the public key size of Rainbow by up to 65 % [14]. However such techniques in general come with the cost of a slower key generation process, and more important, these techniques often make the security analysis harder. This is why we do not want to apply these techniques for now. Nevertheless, in the future, we may apply these techniques, in particular, for special applications.

References

[1] D.J. Bernstein, T. Chou, P. Schwabe: McBits: Fast constant-time code based cryptography. CHES 2013, LNCS vol. 8086, pp. 250 - 272. Springer, 2013.

[2] L. Bettale, J.-C. Faugére, L. Perret: Hybrid approach for solving multivariate systems over finite fields. Journal of Mathematical Cryptology, 3: 177-197, 2009.

[3] O. Billet, H. Gilbert. Cryptanalysis of Rainbow: SCN 2006, LNCS vol. 4116, pp. 336 - 347. Springer, 2006.

[4] J. Bonneau, I. Mironov: Cache-Collision Timing Attacks Against AES. CHES 2006, LNCS vol. 4249, pp. 201 - 215. Springer, 2006.

[5] D. Coppersmith, J. Stern, S. Vaudenay: Attacks on the birational signature scheme. CRYPTO 1994, LNCS vol. 773, pp. 435 - 443. Springer, 1994.

[6] P. Czypek, S. Heyse, E. Thomae: Efficient implementations of MQPKS on constrained devices. CHES 2012, LNCS vol. 7428, pp. 374-389. Springer, 2012.

[7] J. Ding, D. Schmidt: Rainbow, a new multivariable polynomial signature scheme. ACNS 2005, LNCS vol. 3531, pp. 164 - 175. Springer, 2005.

[8] J. Ding, B.-Y. Yang, C.-H. O. Chen, M.-S. Che, C.-M. Cheng: New differential-algebraic attacks and reparametrization of Rainbow. ACNS 2008, LNCS vol. 5037, pp. 242 - 257. Springer, 2008.

[9] eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to

[10] J.-C. Faugére: A new effcient algorithm for computing Gröbner Bases (F4). Journal of Pure and Applied Algebra, 139:61 - 88, 1999.

[11] A. Kipnis, J. Patarin, L. Goubin: Unbalanced Oil and Vinegar schemes. EUROCRYPT 1999, LNCS vol. 1592, pp. 206 - 222. Springer, 1999.

[12] A. Kipnis, A. Shamir: Cryptanalysis of the Oil and Vinegar signature scheme. CRYPTO 1998, LNCS vol. 1462, pp. 257 - 266. Springer, 1998.

[13] NIST: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. Available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`

[14] A. Petzoldt, S. Bulygin, J. Buchmann: CyclicRainbow - a Multivariate Signature Scheme with a Partially Cyclic Public Key. INDOCRYPT 2010, LNCS vol. 6498, pp. 33 - 48. Springer, 2010.

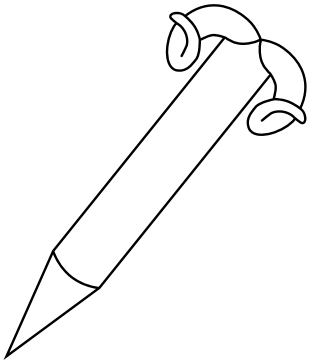
[15] K. Sakumoto, T. Shirai, H. Hiwatari: On Provable Security of UOV and HFE Signature Schemes against Chosen-Message Attack. PQCrypto 2011, LNCS vol. 7071, pp 68 - 82. Springer, 2011.

[16] P. Schwabe, B. Westerbaan: Solving Binary *MQ* with Grover's Algorithm. SPACE 2016, LNCS vol. 10076, pp. 303 - 322. Springer 2016.

Ramstake

KEM Proposal for NIST PQC Project

November 30, 2017



cryptosystem name	Ramstake
principal submitter	Alan Szepieniec imec-COSIC KU Leuven alan.szepieniec@esat.kuleuven.be tel. +3216321953 Kasteelpark Arenberg 10 bus 2452 3001 Heverlee Belgium
auxiliary submitters	-
inventors / developers	same as principal submitter; relevant prior work is credited as appropriate
owner	same as principal submitter
backup contact info	alan.szepieniec@gmail.com
signature	

Contents

1. Introduction

The long-term security of confidential communication channels relies on their capacity to resist attacks by quantum computers. To this end, NIST envisions a transition away from public key cryptosystems that are known to fail in this scenario, and towards the so-called *post-quantum* cryptosystems. One of the functionalities in need of a post-quantum solution that is essential for securing online communication is *ephemeral key exchange*. This protocol enables two parties to agree on a

shared secret key at a cost so insignificant as to allow immediate erasure of all secret key material after execution, as an additional security measure. In the case where the order of the messages need not be interchangeable, this functionality is beautifully captured by the *key encapsulation mechanism* (KEM) formalism of Cramer and Shoup [6]. The same formalism has the added benefit of capturing the syntax and security of the first part of IND-CCA-secure arbitrary-length hybrid encryption schemes, enabling a separation of the public key layer from the symmetric key layer.

The desirable properties of a post-quantum KEM are obvious upon consideration. It should be fast and it should generate short messages, not require too much memory and be implementable on a small area or in a few lines of code. It should inspire confidence by relying on long-standing hard problems or possibly even advertising a proof of security. However, this design document is predicated on the greater importance of a property not included in the previous enumeration: *simplicity*. The requirement for advanced degrees in mathematics on the part of the implementers presents a giant obstacle to mass adoption, whereas no such obstacle exists for mathematically straightforward schemes. More importantly, complexity has the potential to hide flaws and insecurities as they can only be exposed by experts in the field. In contrast, a public key scheme that is accessible to a larger audience is open to scrutiny from that same larger audience, and should therefore engender a greater confidence than a scheme that only a few experts were not able to break.

This document presents Ramstake, a post-quantum key encapsulation mechanism that excels in this category of simplicity. Aside from the well-established tools of hash functions, pseudorandom number generators, and error-correcting codes, Ramstake requires only high school mathematics. Though not optimized for message size and speed, Ramstake is still competitive in these categories with messages of less than one hundred kilobytes generated in a handful of milliseconds on a regular desktop computer at the highest security level. For security, Ramstake relies on a relatively new and under-studied hard problem, which requires several years of attention attention from the larger cryptographic community before it inspires confidence. The flipside of this drawback is the advantage associated with problem diversity: Ramstake is likely to remain immune to attacks that affect other branches of post-quantum cryptography.

Innovation. In a nutshell, this hard problem requires finding *sparse* solutions to linear equations modulo a large Mersenne prime, *i.e.* a prime of the form $p = 2^\pi - 1$. The binary expansions of the solution (x_1, x_2) consist overwhelmingly of zeros. Specifically, these integers can be described as

$$x_i = \sum_{j=1}^{w} 2^{e_j} \quad . \tag{1}$$

We refer to the integer's *Hamming weight* w as the number of ones; their positions e_j are generally chosen uniformly at random from $\{0, \ldots \pi - 1\}$. Ramstake's analogue of the discrete logarithm problem requires finding x_1 and x_2 of this form from G and $H = x_1 G + x_2 \bmod p$. This is an affine variant of the Low Hamming Weight Ratio problem of the Aggarwal *et al.* Mersenne prime cryptosystem [1], whose task is to obtain f and g of this form (1) from $H = fg^{-1} \bmod p$.

Where the Aggarwal *et al* cryptosystem builds on the indistinguishability of low Hamming weight ratios, Ramstake builds on a noisy Diffie-Hellman protocol [2, 3] instead. Alice and Bob agree on a random integer G between 0 and p. Alice chooses sparse integers x_1 and x_2 and sends $H = x_1G + x_2 \bmod p$ to Bob. Bob chooses sparse integers y_1 and y_2 and sends $F = y_1G + y_2 \bmod p$ to Alice. Alice computes $S_a = x_1F \bmod p$ and Bob computes $S_b = y_1G \bmod p$ and both integers approximate $S = x_1y_1G \bmod p$ in the following sense: since p is a Mersenne prime, reduction modulo p does not increase the integer's Hamming weight and as a result the differences $S_a - S = x_1y_2 \bmod p$ and $S_b - S = y_1x_2 \bmod p$ have a sparse binary expansion. Therefore, if x_1, x_2, y_1, y_2 have a sufficiently low Hamming weight, the binary expansions of S_a and S_b agree in most places. Alice and Bob have thus established a shared noisy secret stream of data, or since it will be used as a one-time pad, a *shared noisy one-time pad* (SNOTP, "snow-tipi").

From SNOTP to KEM. There are various constructions in the literature for obtaining KEMs from SNOTPs, each different in its own subtle way. The next couple of paragraphs give a high-level description of a generic transformation targeting IND-CCA security, which is inspired by the "encryption-based approach" of NewHope-Simple [4]. This construction makes abstraction of the underlying sparse integer mathematics.

The encapsulation algorithm is a deterministic algorithm taking a fixed-length random seed s as an explicit argument. If more randomness is needed than is contained in this seed, it is generated from a cryptographically secure pseudorandom number generator (CSPRNG). The algorithm outputs a ciphertext c and a symmetric key k.

The encapsulation algorithm uses an error-correcting code such as Reed-Solomon or BCH to encode the seed s into a larger bitstring. Then the ciphertext c consists of three parts: 1) a contribution to the noisy Diffie-Hellman protocol; 2) the encoding of the seed but one-time-padded with the encapsulator's view of the SNOTP; and 3) the hash of the seed. The decapsulation algorithm computes its own view of the SNOTP using the Diffie-Hellman contribution and undoes the one-time pad to obtain the encoding up to some errors. Under certain conditions, the error-correcting code is capable of retrieving the original seed s from this noisy codeword. At this point, the decapsulation algorithm runs the encapsulation algorithm with the exact same arguments, thus guaranteeing that the produced symmetric key k is the same for both parties. Robust IND-CCA security comes from the fact that the decapsulator can compare bit by bit the received ciphertext against the one that was recreated from the transmitted seed, in addition to verifying the seed's hash against the one that was part of the ciphertext.

2. Specification

2.1. Parameters

The generic description of the scheme refers the following parameters without reference to their value. Concrete values are given in Section 2.4.

- p — the Mersenne prime modulus, satisfies $p = 2^{\pi} - 1$;
- π — the number of bits in the binary expansion of p;
- w — the Hamming weight, which determine the number of ones in the binary expansion of secret sparse integers;
- ν — the number of codewords to encode the transmitted seed into;
- n — the length of a single codeword (in number of bytes);
- κ — the targeted security level (in $\log_2$ of classical operations);
- λ — the length of seed values (in number of bits);
- χ — the length of the symmetric key (in number of bits).

2.2. Tools

2.2.1. Error-Correcting Codes

Ramstake relies on Reed-Solomon codes over $\mathrm{GF}(2^8)$ with designed distance $\delta = 224$ and dimension $k = 32$. Codewords are $n = 255$ field elements long and if there are 111 or fewer errors they can be corrected. With this choice of finite field, one field element coincides with one byte. The following subroutines are used abstractly:

- encode takes a string of $8k = 256$ bits and outputs a sequence of $8n$ bits that represents the Reed-Solomon encoding of the input.
- decode takes a string of $8n$ bits representing a noisy codeword and tries to decode it. If the codeword is decodable, this routine returns the error symbol $\perp$.

This abstract interface suffices for the description of the KEM. Moreover, any concrete instantiation can be exchanged for any other instantiation that adheres to the same interface, or that modifies the interface slightly to retain compatibility.

2.2.2. CSPRNG

Both key generation and encapsulation require a seed expander. All randomness can be generated up front; there is no need to record state and update it as pseudorandomness is generated. We use $\mathsf{xof}(s, \ell)$ to denote the invocation of the CSPRNG to generate a string of ℓ pseudorandom bytes from the seed s.

This abstract interface suffices for the description of the KEM. In the implementations, xof is instantiated with `SHAKE256`. Like in the case of the Reed-Solomon codec, any concrete instantiation can be exchanged for any other instantiation that adheres to the same interface.

2.3. Description

2.3.1. Serialization of Integers

All big integers represent elements in $\{0, \ldots, p-1\}$ and are therefore fully defined by π bits. Denote by $\mathsf{serialize}(a)$ the array if $\lceil \frac{\pi}{8} \rceil$ bytes satisfying

$$a = \sum_{i=0}^{\lceil \frac{\pi}{8} \rceil - 1} \mathsf{serialize}(a)[i] \times 256^i \ . \tag{2}$$

This serialization puts the least significant byte first and pads the array with zeros to meet the given length if the integer is not large enough. It is essentially Little-Endian padded to length $\lceil \frac{\pi}{8} \rceil$, and corresponds with the GMP function `mpz_export`($\cdot$, NULL, -1, 1, 1, 0, a) regardless of whether the integer a is large enough.

2.3.2. Data Structures

Ramstake uses five data structures: a random seed, a secret key, a public key, a ciphertext, and a symmetric key. Random seeds are bitstrings of length λ, whereas symmetric keys are bitstrings of length χ. The other three data structures are more involved.

Secret key. A secret key consists of the following items:

- `seed` — a random seed which fully determines the rest of the secret key in addition to the public key;
- a, b — sparse integers, represented by π bits each.

Public key. A public key consists of the following items:

- `g_seed` – a random seed which is used to generate the random integer G;
- C — integer between 0 and p which represents a noisy Diffie-Hellman contribution. This value satisfies $C = aG + b \bmod p$.

Ciphertext. A ciphertext consists of the following items:

- D — integer between 0 and p which represents a noisy Diffie-Hellman contribution; this value satisfies $D = a'G + b' \bmod p$ where a', b' are secret sparse integers sampled by the encapsulator;
- `seedenc` — string of $8n\nu$ bits; this value is the bitwise xor of the binary expansion of the first $n\nu$ bytes of $\mathsf{serialize}(S)$ and the sequence of ν times $\mathsf{encode}(s)$, where s is the random seed that is the argument to the encapsulation algorithm, and where S is the encapsulator's view of the SNOTP: $S = a'(aG + b) \bmod p$.
- h — hash of the seed s; the purpose of this value is twofold: 1) to speed up decapsulation by enabling the decoder to recognize correct decodings, and 2) to anticipate a proof technique in which the simulator answers decapsulation queries by finding this value's inverse.

These objects are serialized by appending the serializations of their member items in the order presented above. No length information is necessary as the size of each object is a function of the parameters. We overload *serialize* to denote that operation.

In this notation, the symmetric key $k \in \{0,1\}^\chi$ satisfies $k = \mathsf{H}(\mathsf{serialize}(pk) \| coins)$, where pk is the public key and where *coins* is the byte string of random coins used by the encapsulator. Ramstake instantiates H with SHA3-256 with output truncated to χ bits, but any other secure hash function suffices.

2.3.3. Algorithms

A KEM consists of three algorithms, KeyGen, Encaps, and Decaps. Pseudocode for Ramstake's three algorithms is presented in Algorithms 3, 4, and 5. All three functionalities obtain a pseudorandom integer G from a short seed; this subprocedure is called generate_g and is shown in Algorithm 1. Algorithms KeyGen and Encaps rely on a common subroutine called sample_sparse_integer which deterministically samples a sparse integer given enough random bytes and a target Hamming weight, and which is described in Algorithm 2.

algorithm generate_g
input: $\mathtt{seed} \in \{0,1\}^\lambda$ — random seed
output: $g \in \{0, \ldots, p-1\}$ — pseudorandom integer

1: $\mathtt{r} \leftarrow \mathsf{xof}(\mathtt{seed}, \lfloor \frac{\pi}{8} \rfloor + 2)$
2: $g \leftarrow 0$
3: **for** i **from** 0 **to** $\lfloor \frac{\pi}{8} \rfloor + 1\}$ **do:**
4: $\quad g \leftarrow 256 \times g + \mathtt{r}[i]$
5: **end**
6: **return** $g \bmod p$

Algorithm 1: Procedure to sample a random integer from $\{0, \ldots, p-1\}$.

algorithm sample_sparse_integer
input: $\mathtt{r} \in \{0, \ldots, 255\}^{4 \times \mathtt{weight}}$ — enough random bytes
$\mathtt{weight} \in \{0, \ldots, \pi\}$ — number of one bits
output: $a \in \{0, \ldots, p-1\}$ — a sparse integer

1: $a \leftarrow 0$
2: **for** i **from** 0 **to** $\mathtt{weight} - 1$ **do:**
3: $\quad u \leftarrow (\mathtt{r}[4i] \times 256^3 + \mathtt{r}[4i+1] \times 256^2 + \mathtt{r}[4i+2] \times 256 + \mathtt{r}[4i+1]) \bmod \pi$
4: $\quad a \leftarrow a + 2^u$
5: **end**
6: **return** a

Algorithm 2: Procedure to sample a sparse integer from a CSPRNG.

algorithm KeyGen
input: $\mathtt{seed} \in \{0,1\}^\lambda$ — random seed
output: sk — secret key
pk – public key

▷ expand randomness
1: $\mathtt{r} \leftarrow \mathsf{xof}(\mathtt{seed}, 4 \times w + 4 \times w + \lambda/8)$

▷ grab seed for G and generate G
2: $\mathtt{seed\_g} \leftarrow \mathtt{r}[0 : (\lambda/8)]$
3: $G \leftarrow \mathsf{generate\_g}(\mathtt{seed\_g})$

▷ get sparse integers a and b
4: $a \leftarrow \mathsf{sample\_sparse\_integer}(\mathtt{r}[(\lambda/8) : (\lambda/8 + 4 \times w)], w)$
5: $b \leftarrow \mathsf{sample\_sparse\_integer}(\mathtt{r}[(\lambda/8 + 4 \times w) : (\lambda/8 + 4 \times w + 4 \times w)], w)$

▷ compute Diffie-Hellman contribution
6: $C \leftarrow aG + b \bmod p$

7: **return** $sk = (s, a, b)$, $pk = (\mathtt{g\_seed}, C)$

Algorithm 3: Generate a secret and public key pair.

algorithm Encaps
input: $\mathtt{seed} \in \{0,1\}^\lambda$ — random seed
pk — public key
output: $ctxt$ — ciphertext
$k \in \{0,1\}^\chi$ – symmetric key

▷ extract randomness and generate G from seed
1: $\mathtt{r} \leftarrow \mathsf{xof}(\mathtt{seed}, 4 \times w + 4 \times w)$
2: $G \leftarrow \mathsf{generate\_g}(pk.\mathtt{seed\_g})$

▷ sample sparse integers
3: $a' \leftarrow \mathsf{sample\_sparse\_integer}(\mathtt{r}[0 : (4 \times w)], w)$
4: $b' \leftarrow \mathsf{sample\_sparse\_integer}(\mathtt{r}[(4 \times w) : (4 \times w + 4 \times w)], w)$

▷ compute Diffie-Hellman contribution and SNOTP
5: $D \leftarrow a'G + b' \bmod p$
6: $S \leftarrow a'\, pk.C \bmod p$

▷ encode random seed and apply SNOTP
7: $\mathtt{seedenc} \leftarrow \mathsf{serialize}(S)[0 : (n\nu)]$
8: **for** i **from** 0 **to** $\nu - 1$ **do:**
9: $\mathtt{seedenc}[(in) : ((i+1)n)] \leftarrow \mathtt{seedenc}[(in) : ((i+1)n)] \oplus \mathsf{encode}(\mathtt{seed})$
10: **end**

▷ compute symmetric key
11: $k \leftarrow \mathsf{H}(\mathsf{serialize}(pk) \| \mathtt{r})$

▷ complete ciphertext; and return ciphertext and symmetric key
12: $h \leftarrow \mathsf{H}(\mathtt{seed})$
13: **return** $ctxt = (D, \mathtt{seedenc}, h), k$

Algorithm 4: Encapsulate: generate a ciphertext and a symmetric key.

algorithm Decaps
input: $ctxt = (D, \texttt{seedenc}, h)$ — ciphertext
$sk = (\texttt{seed}, a, b)$ — secret key
output: k — symmetric key on success; otherwise $\perp$

▷ recreate public key from secret key seed
1: $\texttt{seed\_g} \leftarrow \mathsf{xof}(sk.\texttt{seed}, \lambda/8)$
2: $G \leftarrow \mathsf{generate\_g}(\texttt{seed\_g})$
3: $C \leftarrow \texttt{sk}.a\, G + \texttt{sk}.b \bmod p$
▷ obtain SNOTP and decode `seedenc`
4: $S' \leftarrow \texttt{sk}.a\, \texttt{ctxt}.D \bmod p$
5: $\texttt{str} \leftarrow \mathsf{serialize}(S')[0 : (n\nu)] \oplus ctxt.\texttt{seedenc}$
6: **for** i **from** 0 **to** $\nu - 1$ **do:**
7: $s \leftarrow \mathsf{decode}(\texttt{str}[(in) : ((i+1)n)])$
8: **if** $s \neq \perp$ **and** $\mathsf{H}(s) = ctxt.h$ **then:**
9: **break**
10: **end**
11: **end**
12: **if** $s = \perp$ **then:**
13: **return** $\perp$
14: **end**
▷ recreate and test ciphertext
$ctxt', k \leftarrow \mathsf{Enc}(s, \texttt{pk} = (\texttt{g\_seed}, C))$
15: **if** $ctxt \neq ctxt'$ **do:**
16: **return** $\perp$
17: **end**
18: **return** k

Algorithm 5: Decapsulate: generate symmetric key and test validity of the given ciphertext.

2.4. Parameter Sets

This document proposes two sets of parameters, called "Ramstake RS 216091", "Ramstake RS 756839". These parameter sets target security levels 128 and 256 in terms of log_2 of required number of operations to mount a successful attack on a classical computer. Both attacks considered in Section 4.3 are fully Groverizable, thus enabling the quantum adversary to divide these target security levels by two. All parameter sets use SHA3-256, SHAKE256, and Reed-Solomon error correction over $\mathbb{F}_{2^8}$ with code length $n = 255$ and design distance $\delta = 224$.

Table 1: Ramstake parameter sets, resulting public key and ciphertext size in kilobytes, and targeted security notion and NIST security level.

π	216091	756839
w	64	128
ν	4	6
λ	256	256
χ	256	256
$\lvert pk \rvert$	26.41 kB	92.42 kB
$\lvert ctxt \rvert$	27.41 kB	93.91 kB
security	IND-CCA	IND-CCA
NIST level	1	5

3. Performance

3.1. Failure Probability

There is a nonzero probability of decapsulation failure even without malicious activity. This event occurs when the two views of the SNOTP are too different, requiring the correction of too many errors. It is possible to find an exact expression for this probability. However, the following argument opts for a more pragmatic approach.

The Reed-Solomon code used has design distance $\delta = 224$, meaning that it can correct up to $t = \lfloor \frac{\delta-1}{2} \rfloor = 111$ byte errors. Decapsulation fails when all ν codewords contain more than 111 errors. By treating the number of errors e in each codeword as independent normally distributed variables, one can obtain a reasonable estimate of the failure probability.

The Sage script `Scripts/parameters.sage`, which is included in the submission package, computes the mean (μ) and standard deviation (σ) of these distributions empirically. For many different random G and appropriately sparse a, b, a', b', the number of different bytes between $\mathsf{serialize}(aa'G + ba' \bmod p)[0:255]$ and $\mathsf{serialize}(aa'G + b'a \bmod p)[0:255]$ is computed. From many such trials it computes μ and σ and a recommended number of codewords ν such that the failure probability drops below 2^{-64}. (Indeed, this script is where the values for ν in the parameter sets of Table 2.4 come from.) The statistics are shown in Table 2.

It is possible to push the failure probability even lower by increasing ν. However, this increase results in a larger ciphertext.

Table 2: Mean μ and standard deviation σ of number of errors in a codeword, along with recommended number of codewords ν for a failure probability less than 2^{-64}.

	216091	756839
μ	72.56	81.38
σ	7.89	7.93
ν	4	6
$\left(1 - \Phi(\frac{e-\mu}{\sigma})\right)^{\nu}$	$\leq 2^{-64}$	$\leq 2^{-64}$

3.2. Complexity

3.2.1. Asymptotic

The loops in the pseudocode of Algorithms 1—5 run through a number of iterations determined by the parameters ν, w, π. Of these parameters, ν is independent of the security parameter κ. The relations between w, π and the security parameter κ are more complex. First π must be large enough to spread out roughly $2w^2$ burst-errors so as to guarantee a low enough byte-error-rate and hence non-failure. Second, the slice-and-dice attack of Section 4.3 must be taken into account as well. These parameters are constrained for non-failure by

$$\frac{2w^2}{\pi} \leq c \ , \tag{3}$$

for some constant c roughly around 0.04. For security, the constraint is

$$2w \geq \kappa \ . \tag{4}$$

These equations thus require $\pi \sim \kappa^2$. The size of the public key and ciphertext grows linearly with this number.

While KeyGen, Encaps and Decaps contain only a small fixed number of big field operations, the modulus of this field is p and the field elements involved therefore have an expansion of up to π bits. Nevertheless, there are two available optimizations to ameliorate this cost. (However, none of the provided implementations employ them.)

- Mersenne form. Reduction modulo p does not require costly division as it does for generic moduli. Instead, shifting and adding does the trick. Let $a = a_o \times p + a_r$ with $a_r < p$. Then $a_r + a_o = a \bmod p$.

- Sparsity. In every big field operation, at least one term or factor is sparse. As a result, the sums can be computed through w localized bitflips with carry. The products can be computed through w shifts and as many full additions.

Consequently, the cost of integer arithmetic is linear π and in w. Therefore, the complexity of all three algorithms is $O(\kappa^3)$.

3.2.2. Pratice

The file `perform.c`, which is included in the submission package, runs a number of trials and collects timing and cycle count information. Table 3 presents the information collected from the optimized implementations during 10 000 trials on a Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz machine with 6144 kB of cache on each of its four cores, with 7741 MB of RAM, and running CentOS linux.

Table 3: Implementation statistics — time and cycle count.

	time (ms)	cycles
Ramstake RS 216091		
KeyGen	2.8	9445009
Encaps	5.4	17700978
Decaps	11.1	36706919
Total	19.3	63852906
Ramstake RS 756839		
KeyGen	13.0	43148424
Encaps	24.1	79342014
Decaps	46.9	154721609
Total	84.1	277212047

It is not surprising that Decaps takes the longest, because it runs Encaps as a subprocedure. The striking difference between Encaps and KeyGen is due to the encoding procedure of the error correcting code. Dealing with this error-correcting code is even more costly in Decaps where the errors are corrected.

3.2.3. Memory and Pseudorandomness

It is difficult to estimate the memory requirements of the error-correcting code algebra as well of the big integer arithmetic for two reasons. 1) The current implementation outsources this operation to another library. 2) because this content is highly dynamic: how much memory is needed depends on the value of the mathematical object being represented. By contrast, the memory requirements of the three main functionalities' outputs is easily determined.

The secret key consists of one $\lambda/8$ byte seed and two integers of (after serialization) $\lceil \pi/8 \rceil$ bytes each, although the integers can be generated anew from the seed. The public key contains one seed of $\lambda/8$ bytes and one integer of $\lceil \pi/8 \rceil$ bytes. The ciphertext consists of one integer of $\lceil \pi/8 \rceil$ bytes, a stream of $n\nu$ bytes representing the one-time-padded repetition code, and a hash of $\chi/8$ bytes. Table 4 summarizes these sizes and presents concrete values for the given parameter sets.

All pseudorandomness is generated (*i.e.* extracted from a short seed) in the first line of those functions that need it. So this is $8w + \lambda/8$ for KeyGen, and $8w$ for Encaps. The Decaps function does not require pseudorandomness but it must get the $\lambda/8$-byte seed for G from the secret key seed via the same CSPRNG. Since Decaps invokes Encaps as a subprocedure, it inherits those requirements for extracting and storing pseudorandomness also.

Table 4: Size (in bytes) of output objects.

	secret key	public key	ciphertext
formula	$\lambda/8 + 2\lceil \pi/8 \rceil$	$\lambda/8 + \lceil \pi/8 \rceil$	$\lceil \pi/8 \rceil + n\nu + \chi/8$
Ramstake 216019	54056	27044	28064
Ramstake 756839	189242	94637	96111

4. Security

4.1. Hard Problems

Ramstake relies on the hardness of at least two problems related to finding sparse solutions to affine equations modulo a pseudo-Mersenne prime p. The formal problem statement of the first is as follows.

Low Hamming Combination (LHC) Problem.
Given: Two coefficients $A, B \in \mathbb{F}_p$ in a large Mersenne prime field $\mathbb{F}_p$.
Task: Find two elements $x_1, x_2 \in \mathbb{F}_p$ with binary expansions of Hamming weight at most w_1 and w_2 respectively, such that $B = Ax_1 + x_2 \bmod p$.

The problem was implicitly introduced by Aggarwal *et al.* [1] in the form of an assumption, which states that the distribution $(A, Ax_1 + x_2)$ is indistinguishable from (A, C) when C is drawn uniformly at random and x_1, x_2 uniformly at random subject to having the required Hamming weight. The same paper also introduces the Low Hamming Ratio Search (LHRS) Problem, which asks to find a pair of low Hamming weight integers x_1, x_2 satisfying $x_2/x_1 = H$. The LHRS Problem is equivalent to the subset of the LHC Problem where $B = 0$. (To see this, set $H = -A$. $\square$)

The LHC problem is only the analogue of the discrete logarithm problem in Diffie-Hellman key agreement. The adversary does not need to compute discrete logarithms; he merely needs to break the Diffie-Hellman problem, which comes in search and decisional variants. The analogues of these problems for sparse integers is formally stated below.

Low Hamming Diffie-Hellman Search (LHDHS) Problem.
Given: Three integers (G, H, F) where $H = x_1G + x_2 \bmod p$ and $F = y_1G + y_2 \bmod p$ for some integers x_1, y_1 of Hamming weight w_1 and x_2, y_2 of Hamming weight w_2.
Task: Find an integer S whose Hamming distance with $x_1F \bmod p$ is at most t, and whose Hamming distance with $y_1H \bmod p$ is also at most t.

Low Hamming Diffie-Hellman Decision (LHDHD) Problem.
Given: Four integers (G, H, F, S) where $H = x_1G + x_2 \bmod p$ and $F = y_1G + y_2 \bmod p$ for some integers x_1, y_1 of Hamming weight w_1 and x_2, y_2 of Hamming weight w_2.
Task: Decide whether or not the Hamming distances between S and $x_1F \bmod p$, and between S and $y_1H \bmod p$, are at most t.

Security requires these problems to be hard, meaning that all polynomial-time quantum adversaries decide the LHDHD Problem with a success probability negligibly far from that of a random guess. The assumed hardness of LHDHD implies

that LHDHS is hard as well, which in turn implies that LHC is hard also. It is unclear how to solve LHDHD in a way that avoids implicitly solving LHC.

It is clear that breaking LHDHS is enough to break the scheme, as that allows the attacker to unpad the seed encoding and recover the seed from there. It is not clear whether security also relies on the LHDHD problem but we include that problem for the sake of completeness, because many Diffie-Hellman type cryptosystems rely on the proper analogue of the Decisional Diffie-Hellman problem.

4.2. SNOTP-to-KEM Construction

There is a gap between the Low Hamming Diffie-Hellman Decision Problem and the IND-CCA (or even IND-CPA) security of Ramstake, originating from the SNOTP-to-KEM construction. I am working on a proof of security but it is unavailable at this point. The following obstacles make such a proof highly non-trivial.

- Failure events in the noisy Diffie-Hellman protocol affect security, especially in the chosen ciphertext model.
- The search problems may be solved in more than one way.
- Circular encryption: the one-time pad is not independent of the message it hides.
- The hash functions should be modeled as quantum-accessible random oracles. However, many classical proof techniques fail in the quantum random oracle model.

It is conceivable that a security proof can only be made to work conditioned on some changes being made to the construction, for instance by changing the inputs to the hash functions. Nevertheless, I do not expect the proof to recommend big changes, thus leaving the construction's big picture intact:

- generate noisy Diffie-Hellman protocol contributions from a short random seed;
- use the noisy Diffie-Hellman key to one-time-pad the error-correcting encoding of the seed;
- undo the noisy one-time pad and decode the codeword;
- invoke the encapsulation algorithm with identical arguments and test if the generated ciphertext matches the received one exactly.

4.3. Attacks

4.3.1. Slice and Dice

Beunardeau *et al.* present an attack exploiting the sparsity of the solutions to the LHRS Problem [5], but it applies equally to the LHC Problem. The attack proceeds as follows.

For each trial, partition the range $R = \{0, \ldots, \pi - 1\}$ into a number of subranges. This number should not be too large, at most a couple hundred. Do this once for x_1 and once for x_2. This yields

$$R_1^{(0)} \sqcup \cdots \sqcup R_1^{(k-1)} = R_2^{(0)} \sqcup \cdots \sqcup R_2^{(\ell-1)} = R \ . \tag{5}$$

Set each such subrange to active or inactive at random. Ensure that the total cardinality of all inactive subranges is at least π.

Each subrange corresponds to a variable $r_i^{(j)}$ whose binary expansion matches that of x_i but restricted to that subrange. Formulaically, this means

$$x_i = \sum_{j=0}^{k-1} 2^{\mathsf{min}(R_i^{(j)})} r_i^{(j)} \quad \text{and} \quad 0 \leq r_i^{(j)} < 2^{\#R_i^{(j)}} \quad . \tag{6}$$

At this point, trim the sums in the left side of Eqn. 6 by dropping the terms that correspond to inactive subranges and replace x_1 and x_2 by their corresponding trimmed sums in the equation $B = Ax_1 + x_2 \bmod p$. Use LLL to find a short solution vector.

A single trial is successful if LLL succeeds in finding the solution that corresponds to the sparse solution. This happens if the guess at inactive subranges is correct, namely if their respective variables are indeed zero (because then their omission does not change the value of the sum).

For the sake of generality, assume x_1 has Hamming weight w_1 and x_2 has Hamming weight w_2. The optimal attacker activates a proportion $\frac{w_1}{w_1+w_2}$ of the range associated to x_1, and a proportion $\frac{w_2}{w_1+w_2}$ of the range associated to x_2. Then the probability of all 1-bits being located inside the active subranges is given by

$$P = \left(\frac{w_1}{w_1 + w_2}\right)^{w_2} \times \left(\frac{w_2}{w_1 + w_2}\right)^{w_1} \quad . \tag{7}$$

The formula is a lot simpler when $w_1 = w_2 = w$, and in this case security mandates that

$$2w \geq \kappa \quad . \tag{8}$$

This algorithm is fully Groverizable. Therefore, the security level halves when considering quantum adversaries with unlimited circuit depth.

4.3.2. Spray and Pray

Spray and pray is essentially a smart brute force search. Choose a random assignment for x_1 with Hamming weight w_1, compute x_2 from the given information and test if its Hamming weight is at most w_2. Assuming the solution is unique, the success probability of a single trial is one over the size of the search space, or $1/\binom{\pi}{w}$. So κ bits of security requires

$$\mathsf{log}_2 \binom{\pi}{w} \geq \kappa \quad . \tag{9}$$

For the parameter sets 216091 and 756839, the left-hand-side of Eqn. 9 is over 838 and 1783, respectively. While the algorithm is fully Groverizable, dividing these numbers by two in order to account for quantum adversaries still results in wildly infeasible complexity.

4.3.3. Stupid Brute Force

Instead of guessing one variable and computing the other from that guess, stupid brute force guesses both at once. A single such guess succeeds with probability $1/\binom{\pi}{w}^2$, *i.e.*, much less likely than the intelligent brute force of the spray-and-pray strategy described above.

Another stupid brute force attack attempts to guess the input of the CSPRNG. By design, these seeds are all 256 bits in length, making for a classical complexity of 2^{256} and 2^{128} quantumly (again assuming unlimited depth).

4.3.4. Lattice Reduction

Aggarwal *et al.* already consider lattice attacks on their cryptosystem and in particular on the LHRS Problem. They observe that it is possible to generate basis vectors for a lattice in which the sought after solution is a short vector. However, that same lattice will contain even shorter vectors that do not correspond to a sparse solution to the original problem. It might be possible to eliminate these parasitical solutions by running lattice reduction with respect to the infinity norm instead of the Euclidean norm, but it is not clear how to do this.

4.3.5. Algebraic System Solving

It is possible in theory to formulate the sparsity constraint algebraically, by constructing polynomials over $\mathbb{F}_p$ that evaluate to zero in all points that satisfy the constraint. At this point a Gröbner basis algorithm can be used to compute a solution. However, the degree of this constraint polynomial is infeasibly large, roughly $\binom{\pi}{w}$. Constructing it requires more work than exhaustively enumerating all potential solutions and testing to see if the linear equations are satisfied.

Another option is to treat the coefficients of the binary expansion of the solutions, as variables in and of themselves. This strategy requires adding polynomials to require that each coefficient lie in $\{0, 1\}$, and that at most w of them are different from zero. The result is a nonlinear system of roughly $4\pi + 2\binom{\pi}{w+1}$ equations in 2π variables with some polynomials having degree $\binom{\pi}{w+1}$. For any practical parameter set, it is infeasible to fully represent this system of equations, let alone to solve it.

4.3.6. Error Triggering

An attacker who can query the decapsulation oracle can obtain feedback on whether the decapsulator was able to decode the transmitted codeword. With enough failures, the attacker can infer the decapsulator's view of the SNOTP. Once the attacker is in possession of this value, he can proceed to decapsulate any ciphertext.

However, in order to exploit this channel of information, the attacker must generate ciphertexts that fail during decapsulation. If his query ciphertext is not the exact output of the encapsulation algorithm upon invocation with the transmitted seed, then the manipulation will trigger a decapsulation failure regardless of whether decoding was successful. In other words, in order to obtain meaningful information about failure events, the attacker must restrict himself to querying only legitimate

outputs of Encaps. Worse still, he has no way of knowing beforehand whether or not a ciphertext is more or less likely to cause failure before the first failure response. Since the failure probability is less than 2^{-64}, the attacker has to make on the order 2^{64} honest queries to get this first failure response.

5. Advantages and Limitations

Advantage: Simplicity. Simplicity is the key selling point of Ramstake. Simple schemes are easier to implement, easier to debug, and easier to analyze. While simpler schemes are sometimes also easier to break, a scheme's resilience to attacks should not rely on its complexity.

Advantage: Problem Diversity. Ramstake relies on different hard problems compared other branches of post-quantum cryptography. Consequently, breakthroughs in cryptanalysis or hard problem solving that break or severely harm other schemes may leave Ramstake intact.

Limitation: New Hard Problem. The hard problem on which Ramstake relies is new and understudied. As a result, it does not offer much assurance of security compared to schemes that have existed (and remained unbroken) for much longer.

Limitation: No Proof. Ramstake claims to offer IND-CCA security even though there is no security reduction to the underlying hard problem. It is therefore conceivable that an attack might break the scheme even without solving the hard problem. Nevertheless, simply because something has not been proven secure yet does not mean it is insecure.

Limitation: Bandwidth and Speed. Lattice-based KEMs are likely to be faster and to require less bandwidth. Nevertheless, Ramstake is competitive in comparison to the very first lattice-based and code-based cryptosystems, and it is conceivable that sparse integer cryptosystems will undergo a similar evolution. However, potential future improvements should not be considered for standardization at this point.

Acknowledgments

The author is thankful to Fré Vercauteren, Reza Reyhanitabar and Ward Beullens for useful discussions and insights. Also, the feedback from NIST after the September deadline was highly useful and highly appreciated. The author is being supported by a Ph.D. Fellowship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (VLAIO, formerly IWT).

References

[1] Aggarwal, D., Joux, A., Prakash, A., Santha, M.: A new public-key cryptosystem via mersenne numbers. IACR Cryptology ePrint Archive 2017, 481 (2017), http://eprint.iacr.org/2017/481

[2] Aguilar, C., Gaborit, P., Lacharme, P., Schrek, J., Zémor, G.: Noisy diffie-hellman protocols (2010), https://pqc2010.cased.de/rr/03.pdf, PQCrypto 2010 The Third International Workshop on Post-Quantum Cryptography (recent results session)

[3] Aguilar, C., Gaborit, P., Lacharme, P., Schrek, J., Zémor, G.: Noisy diffie-hellman protocols or code-based key exchanged and encryption without masking (2010), https://rump2010.cr.yp.to/fae8cd8265978675893352329786cea2.pdf, CRYPTO 2010 (rump session)

[4] Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Newhope without reconciliation. IACR Cryptology ePrint Archive 2016, 1157 (2016), http://eprint.iacr.org/2016/1157

[5] Beunardeau, M., Connolly, A., Géraud, R., Naccache, D.: On the hardness of the mersenne low hamming ratio assumption. IACR Cryptology ePrint Archive 2017, 522 (2017), http://eprint.iacr.org/2017/522

[6] Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. IACR Cryptology ePrint Archive 2001, 108 (2001), http://eprint.iacr.org/2001/108

A. IP Statement

A.1. Statement by Submitter

I, Alan Szepieniec, of Kasteelpark Arenberg 10 / 3001 Heverlee / Belgium , do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as Ramstake, is my own original work, ~~or if submitted jointly with others, is the original work of the joint submitters~~.

I further declare that (check one):

✓ I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as Ramstake; OR (check one or both of the following):

☐ to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as Ramstake, may be covered by the following U.S. and/or foreign patents: "none";

☐ I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations: "none".

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive

financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3 in the Call For Proposals for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3 of the Call For Proposals, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed: Alan Szepieniec
Title: ir.
Date:
Place:

A.2. Statement By Implementation Owner

I, Alan Szepieniec, Kasteelpark Arenberg 10 / 3001 Heverlee / Belgium, am the owner ~~or authorized representative of the owner (print full name, if different than the signer)~~ of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed: Alan Szepieniec
Title: ir.
Date:
Place:

SABER: Mod-LWR based KEM

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Jan-Pieter D'Anvers, KU Leuven, imec-COSIC
- Angshuman Karmakar, KU Leuven, imec-COSIC
- Sujoy Sinha Roy, KU Leuven, imec-COSIC
- Frederik Vercauteren, KU Leuven, imec-COSIC

E-mail address: `frederik.vercauteren@gmail.com`

Telephone: +32-16-37-6080

Postal address:
Prof. Dr. Ir. Frederik Vercauteren
COSIC - Electrical Engineering
Katholieke Universiteit Leuven
Kasteelpark Arenberg 10
B-3001 Heverlee
Belgium

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: ×. See also printed version of "Statement by Each Submitter".

Document based on `pqskeleton` version 20170923 by Daniel Bernstein.

Contents

1 Introduction

Lattice based cryptography is one of the most promising cryptographic families that is believed to offer resistance to quantum computers. We introduce Saber, a family of cryptographic primitives that rely on the hardness of the Module Learning With Rounding problem (Mod-LWR). We first describe Saber.PKE, an IND-CPA secure encryption scheme, and transform it into Saber.KEM, an IND-CCA secure key encapsulation mechanism, using a version of the Fujisaki-Okamoto transform. The design goals of Saber were simplicity, efficiency and flexibility resulting in the following choices: all integer moduli are powers of 2 avoiding modular reduction and rejection sampling entirely; the use of LWR halves the amount of randomness required compared to LWE-based schemes and reduces bandwidth; the module structure provides flexibility by reusing one core component for multiple security levels.

2 General algorithm specification (part of 2.B.1)

2.1 Notation

We denote with $\mathbb{Z}_q$ the ring of integers modulo an integer q with representants in $[0, q)$ and for an integer z, we denote with $z \bmod q$ the reduction of z in $[0, q)$. R_q is the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$ with n a fixed power of 2 (we only need $n = 256$). For any ring R, $R^{l\times k}$ denotes the ring of $l \times k$-matrices over R. For $p|q$, the $\bmod p$ operator is extended to (matrices over) R_q by applying it coefficient-wise. Single polynomials are written without markup, vectors are bold lower case and matrices are denoted with bold upper case. $\mathcal{U}$ denotes the uniform distribution and β_μ is a centered binomial distribution with parameter μ and corresponding standard deviation $\sigma = \sqrt{\mu/2}$. If χ is a probability distribution over a set S, then $x \leftarrow \chi$ denotes sampling $x \in S$ according to χ. If χ is defined on $\mathbb{Z}_q$, $\boldsymbol{X} \leftarrow \chi(R_q^{l\times k})$ denotes sampling the matrix $\boldsymbol{X} \in R_q^{l\times k}$, where all coefficients of the entries in $\boldsymbol{X}$ are sampled from χ. The randomness that is used to generate the distribution can be specified as follows: $\boldsymbol{X} \leftarrow \chi(R_q^{l\times k}; r)$, which means that the coefficients of the entries in matrix $\boldsymbol{X} \in R_q^{l\times k}$ are sampled deterministically from the distribution χ using seed r.

The bitwise shift operations $\ll$ and $\gg$ have the usual meaning when applied to an integer and are extended to polynomials and matrices by applying them coefficient-wise. We use the part selection function $\texttt{bits}(x, i, j)$ with $x \in \mathbb{Z}_{2^k}$ and $k \leq j \leq i$ to access j consecutive bits of a positive integer x *ending* at the i-th index, producing an integer in $\mathbb{Z}_{2^j}$; i.e., written in C code the function returns $(x \gg (i - j))\&(2^j - 1)$. The part selection function is extended to polynomials and matrices by applying it coefficient-wise, where input polynomials are in R_{2^k} and output polynomials in R_{2^j}. Finally let $\lfloor\rceil$ denote rounding to the nearest integer, which is extended to polynomials and matrices coefficient-wise.

2.2 Parameter space

The parameters for Saber are:

- q,p,t: The moduli involved in the scheme are chosen to be powers of 2, in particular $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$ and $t = 2^{\epsilon_t}$ with $\epsilon_q > \epsilon_p > (\epsilon_t + 1)$, so we have $2t \mid p \mid q$. A higher choice for parameters p and t, will result in lower security, but higher correctness. A python script that calculates optimal values for p and t is part of the submission.
- μ: The coefficients of the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ are sampled according to a centered binomial distribution $\beta_\mu(R_q^{l\times 1})$ with parameter μ, where $\mu < p$. A higher value for μ will result in a higher security, but a lower correctness of the scheme.
- n, l: The degree n and the number l of polynomials in the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ determine the dimension of the underlying lattice problem as $l \cdot n$. Increasing the dimension of the lattice problem increases the security, but reduces the correctness.
- $\mathcal{F}, \mathcal{G}, \mathcal{H}$: The hash functions that are used in the protocol. Functions $\mathcal{F}$ and $\mathcal{H}$ are implemented using SHA3-256, while $\mathcal{G}$ is implemented using SHA3-512.
- `gen`: The extendable output function that is used in the protocol, which is implemented using SHAKE-128.

2.3 Constants

The algorithm uses two constants: a constant polynomial $h \in R_q$ with all coefficients set equal to $(2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_t - 2})$ and a constant vector $\boldsymbol{h} \in R_q^{l\times 1}$ consisting of polynomials all coefficients of which are set to the constant $2^{\epsilon_q - \epsilon_p - 1}$. These constants are used to replace rounding operations by a simple bit select.

2.4 Saber Public Key Encryption

Saber.PKE is the public key encryption scheme consisting of the triplet of algorithms (Saber.PKE.KeyGen, Saber.PKE.Enc, Saber.PKE.Dec) as described in Algorithms 1, 2 and 3 respectively. The more detailed technical specifications are given in Section 10.

2.4.1 Saber.PKE Key Generation

The Saber.PKE key generation is specified by the following algorithm.

Algorithm 1: $\texttt{Saber.PKE.KeyGen}()$

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b} = \texttt{bits}(\boldsymbol{A}\boldsymbol{s} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}))$

2.4.2 Saber.PKE Encryption

The Saber.PKE Encryption is specified by the following algorithm, with optional argument r.

Algorithm 2: $\texttt{Saber.PKE.Enc}(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$

$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
if *r is not specified* **then**
  $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b'} = \texttt{bits}(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
$v' = \boldsymbol{b}^T\texttt{bits}(\boldsymbol{s'}, \epsilon_p, \epsilon_p) \in R_p$
$c_m = \texttt{bits}(v' + 2^{\epsilon_p - 1}m, \epsilon_p, \epsilon_t + 1) \in R_{2t}$
return $c := (c_m, \boldsymbol{b'})$

2.4.3 Saber.PKE Decryption

The Saber.PKE Decryption is specified by the following algorithm.

Algorithm 3: $\texttt{Saber.PKE.Dec}(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$

$v = \boldsymbol{b'}^T\texttt{bits}(\boldsymbol{s}, \epsilon_p, \epsilon_p) \in R_p$
$m' = \texttt{bits}(v - 2^{\epsilon_p - \epsilon_t - 1}c_m + h, \epsilon_p, 1) \in R_2$
return m'

2.5 Saber Key-Encapsulation Mechanism

Saber.KEM is the key-encapsulation mechanism consisting of the triplet of algorithms (Saber.KEM.KeyGen, Saber.KEM.Enc, Saber.KEM.Dec) as described in Algorithms 4, 5 and 6 respectively. The more detailed technical specifications are given in Section 10.

2.5.1 Saber.KEM Key Generation

The Saber key generation is specified by the following algorithm.

Algorithm 4: `Saber.KEM.KeyGen()`

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b} = \mathtt{bits}(\boldsymbol{A}\boldsymbol{s} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
$pkh = \mathcal{F}(seed_{\boldsymbol{A}}, \boldsymbol{b})$
$z = \mathcal{U}(\{0,1\}^{256})$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}, z, pkh))$

2.5.2 Saber.KEM Key Encapsulation

The Saber key encapsulation is specified by the following algorithm and makes use of Saber.PKE.Enc as specified in Algorithm 2.

Algorithm 5: `Saber.KEM.Encaps`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$

$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c = \mathtt{Saber.PKE.Enc}(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
return (c, K)

2.5.3 Saber.KEM Key Decapsulation

The Saber key decapsulation is specified by the following algorithm and makes use of Saber.PKE.Dec as specified in Algorithm 3.

Algorithm 6: `Saber.KEM.Decaps`$(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$

$m' = \mathtt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
$(\hat{K}', r') = \mathcal{G}(pkh, m')$
$c' = \mathtt{Saber.PKE.Enc}(pk, m'; r')$
if $c = c'$ **then**
  return $K = \mathcal{H}(\hat{K}', c)$
else
  return $K = \mathcal{H}(z, c)$

3 List of parameter sets (part of 2.B.1)

3.1 Saber.PKE parameter sets

For Saber.PKE, we define the following parameters sets with corresponding security levels in Table 1. The secret key can be compressed by only storing the $log_2(\mu)$ LSB for each coefficient in the entries of $\boldsymbol{s}$. The values for a compressed secret key can be found in brackets.

Sec Cat	fail prob	attack	Classical	Quantum	pk (B)	sk (B)	ciphertext (B)
LightSaber-PKE: $k = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^2$, $\mu = 10$							
1	2^{-120}	primal dual	126 126	115 115	672	832(256)	736
Saber-PKE: $k = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^3$, $\mu = 8$							
3	2^{-136}	primal dual	199 198	181 180	992	1248(288)	1088
FireSaber-PKE: $k = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^5$, $\mu = 6$							
5	2^{-165}	primal dual	270 270	246 245	1312	1664(384)	1472

Table 1: Security and correctness of Saber.PKE.

3.2 Saber.KEM parameter sets

For Saber.KEM, we define the following parameters sets with corresponding security levels in Table 2. The secret key can be compressed by only storing the $log_2(\mu)$ LSB for each coefficient in the entries of $\boldsymbol{s}$. The values for a compressed secret key can be found in brackets. Note that only the secret key size (sk) differs from the Saber.PKE table.

Sec Cat	fail prob	attack	Classical	Quantum	pk (B)	sk (B)	ciphertext (B)
LightSaber-KEM: $k = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^2$, $\mu = 10$							
1	2^{-120}	primal dual	126 126	115 115	672	1568(992)	736
Saber-KEM: $k = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^3$, $\mu = 8$							
3	2^{-136}	primal dual	199 198	181 180	992	2304(1344)	1088
FireSaber-KEM: $k = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^5$, $\mu = 6$							
5	2^{-165}	primal dual	270 270	246 245	1312	3040(1760)	1472

Table 2: Security and correctness of Saber.KEM.

4 Design rationale (part of 2.B.1)

Our design combines several existing techniques resulting in a very simple implementation, that reduces both the amount of randomness and the bandwidth required.

- Learning with Rounding (LWR) [6]: schemes based on (variants of) LWE require sampling from noise distributions which needs randomness. Furthermore, the noise is included in public keys and ciphertexts resulting in higher bandwidth (which can be mitigated by the use of compression techniques akin to LWR). In LWR based schemes, the noise is deterministically obtained by scaling down from a modulus q to modulus p, which does not need randomness and naturally reduces bandwidth for keys and ciphertexts.
- Modules [15, 8]: the module versions of the problems allow to interpolate between the original pure LWE/LWR problems and their ring versions, lowering computational complexity and bandwidth in the process. As in 'Kyber' [8], we use modules to protect against attacks on the ring structure of Ring-LWE/LWR and to provide flexibility. By increasing the rank of the module, it is easy to move to higher security levels without any need to change the underlying arithmetic.
- Reconciliation: we use a simple reconciliation scheme [2] to reduce the failure rate significantly and therefore also the parameters.
- Choice of moduli: all integer moduli in the scheme are powers of 2. This has several advantages: there is no need for explicit modular reduction; sampling uniformly modulo a power of 2 is trivial and thus avoids rejection sampling or other complicated sampling routines, which is important for constant time implementations; we immediately have that the moduli $p \mid q$ in LWR, which implies that the scaling operation maps the uniform distribution modulo q to the uniform distribution modulo p. The main disadvantage of using such moduli is that it excludes the use of the number theoretic transform (NTT) to speed up polynomial multiplication. We remark however that using a compression technique as in 'Kyber' requires one to move back to the polynomial representation (the 'time domain'), so if low bandwidth is a design goal, a scheme that works purely in the NTT-domain ('frequency domain') is simply not possible.

5 Detailed performance analysis (2.B.2)

We evaluated the performance of the software implementation on a Dell laptop with an Intel(R) Core(TM) i7-6600U CPU 2.60GHz processor, Ubuntu operating system, and gcc compiler 7.0. We disabled hyperthreading and TurboBoost. The performance results for the various parameter sets of Saber.KEM can be found in Table 3

Our key encapsulation mechanism uses three hash functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$. For hash functions $\mathcal{F}$ and $\mathcal{H}$, SHA3-256 is used, while $\mathcal{G}$ is implemented using SHA3-512.

Table 3: Performance of Saber.KEM. Cycles for key generation, encapsulation, and decapsulation are represented by **K**, **E**, and **D** respectively in the 4th column. Sizes of secret key (sk), public key (pk) and ciphertext (c) are reported in the last column.

Scheme	Problem	Security	Cycles	Bytes
LightSaber-KEM	Module-LWR	115	**K:** 105,881 **E:** 155,131 **D:** 179,415	**sk:** 1,568 **pk:** 672 **c:** 736
Saber-KEM	Module-LWR	180	**K:** 216,597 **E:** 267,841 **D:** 318,785	**sk:** 2,304 **pk:** 992 **c:** 1,088
FireSaber-KEM	Module-LWR	245	**K:** 360,539 **E:** 400,817 **D:** 472,366	**sk:** 3,040 **pk:** 1,312 **c:** 1,472

6 Expected strength (2.B.4) in general

6.1 Security

The IND-CPA security of Saber.PKE can then be reduced to the decisional Mod-LWR problem as shown by the following theorem:

Theorem 6.1. *For any adversary A, there exist three adversaries B_0, B_1 and B_2 such that $Adv^{ind\text{-}cpa}_{Saber.PKE}(A) \leqslant Adv^{prf}_{\mathbf{gen}()}(B_0) + Adv^{mod\text{-}lwr}_{l,l,\nu,q,p}(B_1) + Adv^{mod\text{-}lwr}_{l+1,l,\nu,q,q/\zeta}(B_2)$, where $\zeta = \min(\frac{q}{p}, \frac{p}{2t})$.*

The correctness of Saber.PKE can be calculated using the python scripts included in the submission, following theorem 6.2:

Theorem 6.2. *Let $\boldsymbol{A}$ be a matrix in $R_q^{l\times l}$ and $\boldsymbol{s}, \boldsymbol{s'}$ two vectors in $R_q^{l\times 1}$ sampled as in Saber.PKE. Define $\boldsymbol{e}$ and $\boldsymbol{e'}$ as the rounding errors introduced by scaling and rounding $\boldsymbol{As}$ and $\boldsymbol{A}^T\boldsymbol{s'}$, i.e. $\texttt{bits}(\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}, \epsilon_q, \epsilon_p) = \frac{p}{q}\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{e}$ and $\texttt{bits}(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{h}, \epsilon_q, \epsilon_p) = \frac{p}{q}\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}$. Let $e_r \in R_q$ be a polynomial with uniformly distributed coefficients with range $[-p/4t, p/4t]$. If we set*

$$\delta = Pr[||(\boldsymbol{s'}^T\boldsymbol{e} - \boldsymbol{e'}^T\boldsymbol{s} + e_r) \mod p||_\infty > p/4]$$

then after executing the Saber.PKE protocol, both communicating parties agree on a n-bit key with probability $1 - \delta$.

This IND-CPA secure encryption scheme is the basis for the IND-CCA secure KEM Saber.KEM=(`Encaps, Decaps`), which is obtained by using an appropriate transformation. Recently, several post-quantum versions [11, 18, 16, 12] of the Fujisaki-Okamoto transform with corresponding security reductions have been developed. At this point, the FO$^{\not\perp}$ transformation in [11] with post-quantum reduction from Jiang et al. [12] gives the tightest reduction for schemes with non-perfect correctness. However, other transformation could be used to turn Saber.PKE into a CCA secure KEM.

6.1.1 Security in the Random Oracle Model

By modeling the hash functions $\mathcal{G}$ and $\mathcal{H}$ as random oracles, a lower bound on the CCA security can be proven. We use the security bound of Hofheinz et al. [11], which considers a KEM variant of the Fujisaki-Okamoto transform that can also handle a small failure probability δ of the encryption scheme. This failure probability should be cryptographically negligibly small for the security to hold. Using Theorem 3.2 and Theorem 3.4 from [11], we get the following theorems for the security and correctness of our KEM in the random oracle model:

Theorem 6.3. *For a IND-CCA adversary B, making at most $q_{\mathcal{H}}$ and $q_{\mathcal{G}}$ queries to respectively the random oracle $\mathcal{G}$ and $\mathcal{H}$, and q_D queries to the decryption oracle, there exists an IND-CPA adversary A such that:*

$$Adv^{ind\text{-}cca}_{Saber.KEM}(B) \leqslant 3Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + q_{\mathcal{G}}\delta + \frac{2q_{\mathcal{G}} + q_{\mathcal{H}} + 1}{2^{256}}.$$

6.1.2 Security in the Quantum Random Oracle Model

Jiang et al. [12] provide a security reduction against a quantum adversary in the quantum random oracle model from IND-CCA security to OW-CPA security. IND-CPA with a sufficiently large message space M implies OW-CPA [11, 7], as is given by following lemma:

Theorem 6.4. *For an OW-CPA adversary B, there exists an IND-CPA adversary A such that:*

$$Adv^{ow\text{-}cpa}_{Saber.PKE}(B) \leqslant Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + 1/|M|$$

Therefore, we can reduce the IND-CCA security of Saber.KEM to the IND.CPA security of the underlying public key encryption:

Theorem 6.5. *For any IND-CCA quantum adversary B, making at most $q_{\mathcal{H}}$ and $q_{\mathcal{G}}$ queries to respectively the random quantum oracle $\mathcal{G}$ and $\mathcal{H}$, and q_D many (classical) queries to the decryption oracle, there exists an adversary A such that:*

$$Adv^{ind\text{-}cca}_{Saber.KEM}(B) \leqslant 2q_{\mathcal{H}}\frac{1}{\sqrt{2^{256}}} + 4q_{\mathcal{G}}\sqrt{\delta} + 2(q_{\mathcal{G}} + q_{\mathcal{H}})\sqrt{Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + 1/|M|}$$

In all attack scenarios we assume that the depth of quantum computation is limited to 2^{64} quantum gates.

6.2 Multi target protection

As described in [8], hashing the public key into $\hat{K}$ has two beneficial effects: it makes sure that K depends on the input of both parties, and it offers multi-target protection. In this scenario, the adversary uses Grover's algorithm to precompute an m that has a relatively high failure probability. Hashing pk into $\hat{K}$ ensures that an attacker is not able to use precomputed 'weak' values of m.

7 Expected strength (2.B.4) for each parameter set

The expected strengths of Saber.PKE and Saber.KEM for each parameter set are included in Table 1 and Table 2.

8 Analysis of known attacks (2.B.5)

The vectors $\boldsymbol{b}, \boldsymbol{b}'$ and $\boldsymbol{c}$ are constructed using the secret $\boldsymbol{s}$ or $\boldsymbol{s}'$, and are publicly known. The security of the secrets needs to be guaranteed through the underlying Mod-LWR problem by choosing appropriate parameters, both for the main Mod-LWR construction with $\boldsymbol{b}$ and $\boldsymbol{b}'$, as for the reconciliation construction which generates $\boldsymbol{c}$.

Our security analysis is similar to the one in 'a New Hope' [3]. The hardness of Mod-LWR is analyzed as an LWE problem, since there are no known attacks that make use of the Module or LWR structure. A set of l LWR samples given by with $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l})$ and $\boldsymbol{s} \leftarrow \beta_\mu(R_q^{l\times 1})$, can be rewritten as an LWE problem in the following way:

$$\begin{aligned}&\left(\boldsymbol{A}, \left\lfloor \frac{p}{q}(\boldsymbol{As} \mod q)\right\rceil \mod p\right)\\ =&\left(\boldsymbol{A}, \frac{p}{q}(\boldsymbol{As} \mod q) + \boldsymbol{e} \mod p\right)\end{aligned}.$$

We can lift this to a problem modulo q by multiplying by $\frac{q}{p}$:

$$\frac{q}{p}\boldsymbol{b} = \boldsymbol{As} + \frac{q}{p}\boldsymbol{e} \mod q\,,$$

where $q/p\,\boldsymbol{e}$ is the random variable containing the error introduced by the rounding operation, of which the coefficients are discrete and nearly uniformly distributed in $(-q/2p, q/2p]$.

BKW type of attacks [13] and linearization attacks [4] are not feasible, since the number of samples is at most double the dimension of the lattice. Moreover, the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ are dense enough to avoid the sparse secret attack described by Albrecht [1]. These attacks only remain infeasible if the generated secret vectors are timely refreshed. As a result, we end up with two main type of attacks: the primal and the dual attack, that make use of BKZ lattice reduction [9, 17].

BKZ uses an SVP oracle in lower dimension b to perform the lattice reduction. The running time of this oracle is exponential in the dimension of the lattice, and the oracle is executed a polynomial number of times. In this report, the security of Saber is based on only one execution of the SVP-oracle, which is a very conservative underestimation of the real security. Laarhoven [14] estimated the complexity for the state-of-the-art SVP solver in high dimensions as $2^{0.292b}$, which can be lowered to $2^{0.265b}$ using Grover's search algorithm.

8.1 Weighted Primal Attack

The primal attack constructs a lattice that has a unique shortest vector that contains the noise $\boldsymbol{e}$ and the secret $\boldsymbol{s}$. BKZ, with a certain block dimension b, can be used to find this unique solution. An LWE sample $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ can be transformed to the following lattice: $\Lambda = \{\boldsymbol{v} \in \mathbb{Z}^{m+n+1} : (\boldsymbol{A}|\boldsymbol{I}_m|-\boldsymbol{b})\boldsymbol{v} = 0 \mod q\}$, with dimension $d = m+n+1$ and volume q^m. The unique shortest vector in this lattice is $\boldsymbol{v} = (\boldsymbol{s}, \boldsymbol{e}, 1)$, and it has a norm of $\lambda \approx \sqrt{n\sigma_s^2 + m\sigma_e^2}$. Using heuristic models, the primal attack succeeds if [3]:

$$\begin{aligned}\sqrt{n\sigma_s^2 + m\sigma_e^2} <& \delta^{2b-d-1}\mathrm{Vol}(\Lambda)^{\frac{1}{d}} \\ \text{where: } \delta =& ((\pi b)^{\frac{1}{d}} \frac{b}{2\pi e})^{\frac{1}{2(b-1)}}\end{aligned}$$

However, the vector $\boldsymbol{v} = (\boldsymbol{s}, \boldsymbol{e}, 1)$ is unbalanced since $||\boldsymbol{s}_i||$ is not necessarily equal to $||\boldsymbol{e}_i||$. In our case, $||\boldsymbol{s}_i|| < ||\boldsymbol{e}_i||$, which can be exploited by the lattice rescaling method described by Bai et al. [5], and further analysed in [10]. The expected norm of each entry of $\boldsymbol{s}$ is σ_s, while the approximate expected norm of components of e_i is $\sigma_e = \sqrt{q^2/12p^2}$. We can therefore construct the weighted lattice:

$$\begin{aligned}\Lambda' =& \{(\boldsymbol{x}, \boldsymbol{y}', z) \in \mathbb{Z}^n \times (\alpha^{-1}\mathbb{Z})^m \times \mathbb{Z} : (\boldsymbol{x}, \alpha\boldsymbol{y}', z) \in \Lambda\} \\ \text{where: } \alpha =& \frac{\sigma_e}{\sigma_s}\end{aligned}$$

with dimension $(n + m + 1)$ and volume $(q/\alpha)^m$. Analogous to [3], the primal attack is successful if the projected norm of the unique shortest vector on the last b Gram-Schmidt vectors is shorter than the $(d-b)^{\text{th}}$ Gram-Schmidt vector, or:

$$\sigma_s\sqrt{b} \leqslant \delta^{2b-d-1}\left(\frac{q}{\alpha}\right)^{\frac{m}{d}}.$$

8.2 Weighted Dual Attack

The dual attack tries to distinguish between an LWE sample $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ and a uniformly random sample by finding a short vector $(\boldsymbol{v}, \boldsymbol{w})$ in the lattice $\Lambda = \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{y} \mod q\}$. This short vector is used to compute a distinguisher $z = \boldsymbol{vb}$. If $\boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}$, we can write $z = \boldsymbol{vAs} + \boldsymbol{ve} = \boldsymbol{ws} + \boldsymbol{ve}$, which is small and approximately Gaussian distributed. If $\boldsymbol{b}$ is generated uniformly, z will also be uniform in q.

Since in our case, $||\boldsymbol{s}_i|| < ||\boldsymbol{e}_i||$, we observe that the $\boldsymbol{ws}$ term will be smaller than the $\boldsymbol{ve}$ term. The weighted attack optimizes the shortest vector so that these terms have a similar variance, by considering the weighted lattice $\Lambda' = \{(\boldsymbol{x}, \boldsymbol{y}') \in \mathbb{Z}^m \times (\alpha^{-1}\mathbb{Z})^n : (\boldsymbol{x}, \alpha\boldsymbol{y}') \in \Lambda \mod q\}$. This lattice has a dimension of $n+m$ and a volume of $(q/\alpha)^n$, so the BKZ algorithm will output a shortest vector $\boldsymbol{u} = (\boldsymbol{v}, \boldsymbol{w}') = (\boldsymbol{v}, \alpha\boldsymbol{w})$ of size $||\boldsymbol{u}|| \approx \delta^{m+n}(q/\alpha)^{\frac{n}{m+n}}$. Calculating z with the shortest vector in the LWE case gives $z = \boldsymbol{w}'(\alpha\boldsymbol{s}) + \boldsymbol{ve}$, of which the Gaussian distribution standard deviation will thus be equal to $\sigma_z = ||\boldsymbol{u}||\sigma_e = ||\boldsymbol{u}||\sqrt{q^2/12p^2}$.

Following the strategy of [3], we can now calculate the cost of the dual attack. The statistical distance between a uniformly distributed z and a Gaussian distributed z is bounded by $\epsilon = 4\exp(-2\pi^2\tau^2)$, where $\tau = \sigma_z/q$. Since the key is hashed, an advantage of ϵ is not sufficient and must be repeated at least $R = \max(1, 1/(2^{0.2075b}\epsilon^2))$ times. The cost of the dual attack is thus equal to:

$$\text{Cost}_{\text{dual}} = \text{Cost}_{\text{BKZ}} R = b2^{cb} R, .$$

9 Advantages and limitations (2.B.6)

Advantages:

- No modular reduction: since all moduli are powers of 2 we do not require explicit modular reduction. Furthermore, sampling a uniform number modulo a power of 2 is trivial in that it does not require any rejection sampling or more complicated sampling routines. This is especially important when considering constant time implementations.
- Modular structure and flexibility: the core component consists of arithmetic in the fixed polynomial ring $\mathbb{Z}_{2^{13}}[X]/(X^{256}+1)$ for all security levels. To change security, one simply uses a module of higher rank.
- Less randomness required: due to the use of Mod-LWR, our algorithm requires less randomness since no error sampling is required as in (Mod-)LWE.
- Low bandwidth: again due to the use of Mod-LWR, the bandwidth required is lower than similar systems based on (Mod-)LWE.

Limitations:

- The use of two-power moduli precludes NTT-like multiplication algorithms, so we have to resort to Toom-Cook and Karatsuba.
- The functionality is limited to an encryption scheme and a KEM. No signature scheme is provided.

10 Technical Specifications (2.B.1)

This section provides technical specifications for implementing Saber. For more details, the reader may read the C source code present in the reference implementation package.

10.1 Data Types and Conversions

10.1.1 Bit Strings and Byte Strings

A bit is an element of the set $\{0, 1\}$ and a bit string is an ordered sequence of bits. In a bit string the rightmost or the first bit is the least significant bit and the leftmost or the last bit is the most significant bit. A byte is a bit string of length 8 and a byte string is an ordered array of bytes. Following the same convention, the rightmost or the first byte is the least significant byte and the leftmost or the last byte is the most significant byte.

For example, consider the byte string of length three: $3d$ $2c$ $1b$. The most significant byte is $3d$ and the least significant byte is $1b$. This byte string corresponds to the bit string 0011 1101 0010 1100 0001 1011. The least significant bit of the byte string is 1 and the most significant bit is 0.

10.1.2 Concatenation of Bit Strings

Concatenation of two bit strings b_0 to b_1 is denoted by $b_1 \parallel b_0$ where b_0 is present in the least significant part and b_1 is present in the most significant part. The length of the concatenated bit string is the sum of the lengths of b_0 and b_1.

Similarly concatenation of n bit strings b_0 to b_{n-1} is denoted by $b_{n-1} \parallel b_{n-2} \parallel \ldots \parallel b_1 \parallel b_0$ where b_0 is present in the least significant part and b_{n-1} is present in the most significant part. Naturally the length of the concatenated bit string is the sum of the lengths of b_0 to b_{n-1}.

10.1.3 Concatenation of Byte Strings

Concatenation of two byte strings B_0 to B_1 is denoted by $B_1 \parallel B_0$ where B_0 is present in the least significant part and B_1 is present in the most significant part. The length of the concatenated byte string is the sum of the lengths of B_0 and B_1.

Similarly concatenation of n byte strings B_0 to B_{n-1} is denoted by $B_{n-1} \parallel B_{n-2} \parallel \ldots \parallel B_1 \parallel B_0$ where B_0 is present in the least significant part and B_{n-1} is present in the most significant part. Naturally the length of the concatenated byte string is the sum of the lengths of B_0 to B_{n-1}.

10.1.4 Polynomials

All polynomials in $R_q = \mathbb{Z}_q[x]/(x^n+1)$ have 256 coefficients and the coefficients are of size 13 bits since $n = 256$ and $q = 2^{13}$. All polynomials in $R_p = \mathbb{Z}_p[x]/(x^n+1)$ have 256 coefficients and the coefficients are of size 10 bits since $n = 256$ and $p = 2^{10}$. The i-th coefficient of a

polynomial object, say pol, is accessed by $pol[i]$. In the following example

$$pol = c_{255}x^{255} + \ldots + c_1 x + c_0 \tag{1}$$

the constant coefficient c_0 is accessed by $pol[0]$ and the highest-degree (i.e. x^{255}) coefficient c_{255} is accessed by $pol[255]$.

10.1.5 Vectors

A vector in $R_q^{l\times 1}$ is an ordered collection of l polynomials from R_q. The i-th element of a vector object, say $\boldsymbol{v} \in R_q^{l\times 1}$, is accessed by $\boldsymbol{v}[i]$, where $(0 \leq i \leq l-1)$.

10.1.6 Matrices

A matrix in $R_q^{l\times m}$ is a collection of $l \times m$ polynomials in row-major order. The polynomial present in the i-th row and j-th column a matrix object, say $\boldsymbol{M}$, is accessed by $\boldsymbol{M}[i,j]$. Here $(0 \leq i \leq l-1)$ and $(0 \leq j \leq m-1)$.

10.1.7 Data conversion algorithms

The data conversion algorithms are defined as follows.

- $\mathsf{BS2POL}_q$: This function takes a byte string of length $13\times 256/8$ and transforms it into a polynomial in R_q. The algorithm is shown in Alg. 7.

Algorithm 7: Algorithm $\mathsf{BS2POL}_q$

Input: BS: byte string of length $13 \times 256/8$
Output: pol_q: polynomial in R_q

Interpret BS as a bit string of length 13×256.
Split it into bit strings each of length 13 and obtain $(bs_{255} \parallel \ldots \parallel bs_0) = BS$.
for $(i = 0,\ i < 256,\ i = i + 1)$ **do**
    $pol_q[i] \leftarrow bs_i$
return pol

- $\mathsf{POL}_q\mathsf{2BS}$: This function takes a polynomial from R_q and transforms it into a byte string of length $13 \times 256/8$. The algorithm is shown in Alg. 8.
- $\mathsf{BS2POLVEC}_q$: This function takes a byte string of length $l\times 13\times 256/8$ and transforms it into a vector in $R_q^{l\times 1}$. The algorithm is shown in Alg. 9.
- $\mathsf{POLVEC}_q\mathsf{2BS}$: This function takes a vector from $R_q^{l\times 1}$ and transforms it into a byte string of length $l \times 13 \times 256/8$. The algorithm is shown in Alg. 10.

Algorithm 8: Algorithm $\mathsf{POL}_q\mathsf{2BS}$

Input: pol_q: polynomial in R_q
Output: BS: byte string of length $13 \times 256/8$

1 Interpret the coefficients of pol_q as bit strings, each of length 13.
2 Concatenate the coefficients and obtain the bit string $bs = (pol_q[255] \parallel \ldots \parallel pol_q[0])$ of length 13×256.
3 Interpret the bit string bs as the byte string BS of length $13 \times 256/8$.
4 **return** BS

Algorithm 9: Algorithm $\mathsf{BS2POLVEC}_q$

Input: BS: byte string of length $l \times 13 \times 256/8$
Output: $\boldsymbol{v}$: vector into $R_q^{l \times 1}$

1 Split BS into l byte strings of length $13 \times 256/8$ and obtain $(BS_{l-1} \parallel \ldots \parallel BS_0) = BS$
2 **for** $(i = 0,\ i < l,\ i = i + 1)$ **do**
3 $\quad \boldsymbol{v}[i] = \mathsf{BS2POL}_q(BS_i)$
4 **return** $\boldsymbol{v}$

Algorithm 10: Algorithm $\mathsf{POLVEC}_q\mathsf{2BS}$

Input: $\boldsymbol{v}$: vector in $R_q^{l \times 1}$
Output: BS: byte string of length $l \times 13 \times 256/8$

1 Instantiate the byte strings BS_0 to BS_{l-1} each of length $13 \times 256/8$.
2 **for** $(i = 0,\ i < l,\ i = i + 1)$ **do**
3 $\quad BS_i = \mathsf{POLVEC}_q\mathsf{2BS}(\boldsymbol{v}[i])$
4 Concatenate these byte strings and get the byte string $BS = (BS_{l-1} \parallel \ldots \parallel BS_0)$.
5 **return** BS

- $\mathsf{BS2POL}_p$: This function takes a byte string of length $10 \times 256/8$ and transforms it into a polynomial in R_p. The algorithm is shown in Alg. 11.

Algorithm 11: Algorithm $\mathsf{BS2POL}_p$

Input: BS: byte string of length $10 \times 256/8$
Output: pol_p: polynomial in R_p

1 Interpret BS as a bit string of length 10×256.
2 Split it into bit strings each of length 10 and obtain $(bs_{255} \parallel \ldots \parallel bs_0) = BS$.
3 **for** $(i = 0,\ i < 256,\ i = i + 1)$ **do**
4 $\quad pol_p[i] \leftarrow bs_i$
5 **return** pol

- $\mathsf{POL}_p\mathsf{2BS}$: This function takes a polynomial from R_p and transforms it into a byte string of length $10 \times 256/8$. The algorithm is shown in Alg. 12.

Algorithm 12: Algorithm $\mathsf{POL}_p\mathsf{BS}$

Input: pol_p: polynomial in R_p
Output: BS: byte string of length $10 \times 256/8$

Interpret the coefficients of pol_p as bit strings, each of length 10.
Concatenate the coefficients and obtain the bit string $bs = (pol_p[255] \parallel \ldots \parallel pol_p[0])$ of length 10×256.
Interpret the bit string bs as the byte string BS of length $10 \times 256/8$.
return BS

- $\mathsf{BS2POLVEC}_p$: This function takes a byte string of length $l \times 10 \times 256/8$ and transforms it into a vector $\boldsymbol{v} \in R_p^{l \times 1}$. The algorithm is shown in Alg. 13.

Algorithm 13: Algorithm $\mathsf{BS2POLVEC}_p$

Input: BS: byte string of length $l \times 10 \times 256/8$
Output: $\boldsymbol{v}$: vector into $R_p^{l \times 1}$

Split BS into byte strings of size $10 \times 256/8$ and obtain $(BS_{l-1} \parallel \ldots \parallel BS_0) = BS$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
    $\boldsymbol{v}[i] = \mathsf{BS2POL}_p(BS_i)$
return $\boldsymbol{v}$

- $\mathsf{POLVEC}_p\mathsf{2BS}$: This function takes a vector from $R_p^{l \times 1}$ and transforms it into a byte string of length $l \times 10 \times 256/8$. The algorithm is shown in Alg. 14.

Algorithm 14: Algorithm $\mathsf{POLVEC}_p\mathsf{2BS}$

Input: $\boldsymbol{v}$: vector in $R_p^{l \times 1}$
Output: BS: byte string of length $l \times 10 \times 256/8$

Instantiate the byte strings BS_0 to BS_{l-1} each of length $10 \times 256/8$.
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
    $BS_i = \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{v}[i])$
Concatenate these byte strings and get the byte string $BS = (BS_{l-1} \parallel \ldots \parallel BS_0)$.
return BS

- $\mathsf{MSG2POL}_p$: This function takes a 32 byte message and transforms it into a polynomial in R_p. The algorithm is shown in Alg. 15

Algorithm 15: Algorithm $\mathsf{MSG2POL}_p$

Input: m: 32-byte message.
Output: pol_p: polynomial in R_p.

Constant $const = \log_2(p) - 1$
Split m into bit strings each of length 1 and obtain $(m_{255} \parallel \ldots \parallel m_0) = m$.
for $(i = 0,\ i < 256,\ i = i + 1)$ **do**
    $pol_p[i] = (m_i \ll const)$
return pol_p

10.2 Supporting Functions

10.2.1 SHAKE-128

SHAKE-128, standardized in FIPS-202, is used as the extendable-output function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and generates the output byte string of length 'output_length' in the byte array *output_byte_string* as described below.

$$\mathsf{SHAKE\text{-}128}(output\_byte\_string, \text{output\_length}, input\_byte\_string, \text{input\_length}) \tag{2}$$

10.2.2 SHA3-256

SHA3-256, standardized in FIPS-202, is used as a hash function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and generates the output byte string of length 32 in the byte array *output_byte_string* as described below.

$$\mathsf{SHA3\text{-}256}(output\_byte\_string, input\_byte\_string, \text{input\_length}) \tag{3}$$

10.2.3 SHA3-512

SHA3-512, standardized in FIPS-202, is used as a hash function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and generates the output byte string of length 64 in the byte array *output_byte_string* as described below.

$$\mathsf{SHA3\text{-}512}(output\_byte\_string, input\_byte\_string, \text{input\_length}) \tag{4}$$

10.2.4 HammingWeight

This function returns the Hamming weight of the input bit string. For example,

$$w = \mathsf{HammingWeight}(a) \tag{5}$$

returns the Hamming weight of the input bit string a to the integer w. Naturally, HammingWeight always returns non-negative integers.

10.2.5 Randombytes

This function outputs a random byte string of a specified length. The following example shows how to use randombytes to generate a random byte string *seed* of length SABER_SEEDBYTES.

$$\mathsf{randombytes}(\mathit{seed}, \text{SABER\_SEEDBYTES})$$

10.2.6 PolyMul

This function performs polynomial multiplications in R_p and R_q. For two polynomials a and b in R_p, their product $c \in R_p$ is computed using PolyMul as follows.

$$c = \mathsf{PolyMul}(a, b, p)$$

Similarly, for two polynomials a' and b' in R_q, their product $c' \in R_q$ is computed using PolyMul as follows.

$$c' = \mathsf{PolyMul}(a', b', q)$$

10.2.7 MatrixVectorMul

This function performs multiplication of a matrix, say $\boldsymbol{M} \in R_q^{l \times l}$, and a vector $\boldsymbol{v} \in R_q^{l \times 1}$ and returns the product vector $\boldsymbol{mv} = \boldsymbol{M} * \boldsymbol{v} \in R_q^{l \times 1}$. The algorithm is described in Alg. 16. The function is used in the following way.

$$\boldsymbol{mv} = \mathsf{MatrixVectorMul}(\boldsymbol{M}, \boldsymbol{v}, q)$$

10.2.8 VectorMul

This function takes a vector $\boldsymbol{v}_a \in R_p^{l \times 1}$ and a vector $\boldsymbol{v}_b \in R_p^{l \times 1}$ and computes the product of $\boldsymbol{v}_a^T$ and $\boldsymbol{v}_b$, which is a polynomial $c \in R_p$. Here $\boldsymbol{v}_a^T$ stands for the transpose of $\boldsymbol{v}_a$. The algorithm is described in Alg. 17. The function is used in the following way.

Algorithm 16: Algorithm MatrixVectorMul

Input: $\boldsymbol{M}$: matrix in $R_q^{l\times l}$,
$\boldsymbol{v}$: vector in $R_q^{l\times 1}$,
q: coefficient modulus
Output: $\boldsymbol{mv}$: vector in $R_q^{l\times 1}$

Instantiate polynomial object c
for *($i = 0$, $i < l$, $i = i + 1$)* **do**
  $c = 0$
  for *($j = 0$, $j < l$, $j = j + 1$)* **do**
    $c = c + \mathsf{PolyMul}(\boldsymbol{M}[i, j], \boldsymbol{v}[j], q)$
  $\boldsymbol{mv}[i] = c$
return $\boldsymbol{mv}$

$$c = \mathsf{VectorMul}(\boldsymbol{v}_a, \boldsymbol{v}_b, p)$$

Algorithm 17: Algorithm VectorMul

Input: $\boldsymbol{v}_a$: vector in $R_p^{l\times 1}$,
$\boldsymbol{v}_b$: vector in $R_p^{l\times 1}$,
p: coefficient modulus
Output: c: polynomial in R_p

$c \leftarrow 0$
for *($i = 0$, $i < l$, $i = i + 1$)* **do**
  $c = c + \mathsf{PolyMul}(\boldsymbol{v}_a[i], \boldsymbol{v}_b[i], p)$
return c

10.2.9 Verify

This function compares two byte strings of the same length and outputs a binary bit. The output bit is '1' if the byte strings are equal; otherwise it is '0'. The following example shows how to use Verify to compare the byte strings BS_0 and BS_1 of length 'input_length'.

$$c = \mathsf{Verify}(BS_0, BS_1, input\_length) \tag{6}$$

If $BS_0 = BS_1$ then $c = 0$; otherwise $c = 1$.

10.2.10 Round

This function takes a polynomial in R_q and rounds it into a polynomial in R_p. The steps are shown in Alg. 18. The following example shows the use of Round to transform a polynomial

$pol_q \in R_q$ into a polynomial $pol_p \in R_p$.

$$pol_p = \mathsf{Round}(pol_q)$$

Algorithm 18: Algorithm Round for rounding a polynomial $\in \mathbb{R}_q$

Input: pol_q: polynomial $\in \mathbb{R}_q$
Output: pol_p: polynomial $\in \mathbb{R}_p$

```
1 const = ϵ_q − ϵ_p
2 for (i = 0, i < 256, i = i + 1) do
3     pol_p[i] = (pol_q[i] ≫ const) (mod p)
4 return pol_p
```

10.2.11 Floor

This function takes an input a and returns the largest integer less than or equal to a. The following example shows how the use of this function.

$$c = \mathsf{floor}(a)$$

For e.g., $c = 1$ when $a = \frac{3}{2}$ and $c = -2$ when $a = \frac{-3}{2}$.

10.2.12 ReconDataGen

This function takes a polynomial from R_p and generates the reconciliation vector. The steps performed in ReconDataGen are shown in Alg. 19. The following example shows how to use ReconDataGen to compute the reconciliation byte string rec from the input polynomial $pol_p \in R_p$.

$$rec = \mathsf{ReconDataGen}(pol_p)$$

10.2.13 Recon

This function takes a polynomial in R_p and associated reconciliation byte string as inputs and generates a bit string of length 256. The internal steps are shown in Alg. 20. The following example shows the use of Recon to generate the 256-bit bit string K from the reconciliation byte string rec and the polynomial $pol_p \in R_p$.

$$K = \mathsf{Recon}(rec, pol_p)$$

Algorithm 19: Algorithm ReconDataGen for generation of reconciliation data from a polynomial $\in \mathbb{R}_p$

Input: pol_p: polynomial $\in \mathbb{R}_p$
Output: *rec*: byte string of length (RECON_SIZE+1) $\times$ 256/8 bytes.

1 $const = \epsilon_p - \texttt{RECON\_SIZE} - 1$
2 Instantiate a vector object $\boldsymbol{v}$
3 **for** $(i = 0,\ i < 256,\ i = i + 1)$ **do**
4 $\quad \boldsymbol{v}[i] = pol_p[i] \gg const$
5 Interpret the elements of $\boldsymbol{v}$ as bit strings, each of length RECON_SIZE+1 bits.
6 Concatenate the elements and obtain the bit string $bs = (\boldsymbol{v}[255] \parallel \ldots \parallel \boldsymbol{v}[0])$ of length (RECON_SIZE + 1) $\times$ 256.
7 Interpret the bit string bs as the byte string *rec* of length (RECON_SIZE + 1) $\times$ 256/8.
8 **return** *rec*

Algorithm 20: Algorithm Recon

Input: *rec*: vector of 256 elements where each element is of RECON_SIZE+1 bits
pol_p: polynomial $\in \mathbb{R}_p$
Output: K: bit string of length 256

1 $const_0 = \epsilon_p - \texttt{RECON\_SIZE} - 1$
2 $const_1 = 2^{\epsilon_p - 2} - 2^{\epsilon_p - 2 - \texttt{RECON\_SIZE}}$
3 **for** $(i = 0,\ i < 256,\ i = i + 1)$ **do**
4 $\quad temp = rec[i] \ll const_0$
5 $\quad temp = pol_p[i] - temp + const_1$
6 $\quad K_i = \mathsf{floor}(\frac{temp}{2^{\log_2(p)-1}})$ `// outputs a bit`
7 **return** $K = (K_{255} \parallel \ldots \parallel K_0)$

10.2.14 GenMatrix

This function generates a matrix in $R_q^{l \times l}$ from a random byte string (called seed) of length SABER_SEEDBYTES. The steps are described in the algorithm GenMatrix in Alg. 21. The use of GenMatrix to generate the matrix $\boldsymbol{A} \in R_q^{l \times l}$ from the seed $\text{seed}_{\boldsymbol{A}}$ is as follows.

$$\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$$

10.2.15 GenSecret

This function takes a random byte string (called seed) of length SABER_SEEDBYTES as input and outputs a secret which is a vector in $R_q^{l \times 1}$ with coefficients sampled from a centered binomial distribution β_μ. The steps are described in the algorithm GenSecret in Alg. 22 The use of GenSecret to generate a secret $\boldsymbol{s} \in R_q^{l \times 1}$ from a random seed $seed_{\boldsymbol{s}}$ is shown as follows.

Algorithm 21: Algorithm GenMatrix for generation of matrix $\boldsymbol{A} \in R_q^{l \times l}$

Input: $seed_{\boldsymbol{A}}$: random seed of length SABER_SEEDBYTES
Output: $\boldsymbol{A}$: matrix in $R_q^{l \times l}$

Instantiate byte string object buf of length $l^2 \times n \times \epsilon_q/8$
SHAKE-128($buf, l^2 \times n \times \epsilon_q/8, seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
Split buf into $l^2 \times n$ equal byte strings of bit length ϵ_q and obtain $(buf_{l^2n-1} \parallel \ldots \parallel buf_0) = buf$
$k = 0$
for *($i_1 = 0$, $i_1 < l$, $i_1 = i_1 + 1$)* **do**
    for *($i_2 = 0$, $i_2 < l$, $i_2 = i_2 + 1$)* **do**
        for *($j = 0$, $j < n$, j=j+1)* **do**
            $\boldsymbol{A}[i_1, i_2][j] = buf_k$
            $k = k + 1$
return $\boldsymbol{A} \in R_q^{l \times l}$

$$\boldsymbol{s} = \mathsf{GenSecret}(seed_{\boldsymbol{s}})$$

Algorithm 22: Algorithm GenSecret for generation of secret $\boldsymbol{s} \in R_q^{l \times 1}$

Input: $seed_{\boldsymbol{s}}$: random seed of length SABER_SEEDBYTES
Output: $\boldsymbol{s}$: vector in R_q^2

Instantiate a byte string object buf of length $l \times n \times \mu/8$
SHAKE-128(buf, $l \times n \times \mu/8$, $seed_{\boldsymbol{s}}$, SABER_SEEDBYTES)
Split buf into $2 \times l \times n$ bit strings of length $\mu/2$ bits and obtain $(buf_{2ln-1} \parallel \ldots \parallel buf_0) = buf$
$k = 0$
for *($i = 0$, $i < l$, $i = i + 1$)* **do**
    for *($j = 0$, $j < n$, $j = j + 1$)* **do**
        $\boldsymbol{s}[i][j] = \mathtt{HammingWeight}(buf_k) - \mathtt{HammingWeight}(buf_{k+1})$
        $k = k + 2$
return $\boldsymbol{s} \in R_q^2$

10.3 IND-CPA encryption

The IND-CPA encryption consists of 3 components,

- Saber.PKE.KeyGen, returns public key and the secret key to be used in the encryption.
- Saber.PKE.Enc, returns the ciphertext obtained by encrypting the message.

- Saber.PKE.Dec, returns a message obtained by decrypting the ciphrtext.

10.3.1 Saber.PKE.KeyGen

This function generates IND-CPA public and secret key pair as byte strings of length `SABER_INDCPA_PUBKEYBYTES` and `SABER_INDCPA_SECRETKEYBYTES` respectively. The details of Saber.PKE.KeyGen are provided in Alg. 23.

Algorithm 23: Algorithm Saber.PKE.KeyGen for IND-CPA public and secret key pair generation

Output: $PublicKey_{cpa}$: byte string of public key,
$SecretKey_{cpa}$: byte string of secret key

randombytes($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
SHAKE-128($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES, $seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
randombytes($seed_{\boldsymbol{s}}$, SABER_NOISE_SEEDBYTES)
$\boldsymbol{A}$ = GenMatrix($seed_{\boldsymbol{A}}$)
$\boldsymbol{s}$ = GenSecret($seed_{\boldsymbol{s}}$)
$\boldsymbol{v}$ = MatrixVectorMul($\boldsymbol{A}^T, \boldsymbol{s}$) // Here $\boldsymbol{A}^T$ is transpose of $\boldsymbol{A}$
for ($i = 0$, $i < l$, $i = i + 1$) **do**
    $\boldsymbol{v}_p[i]$ = Round($\boldsymbol{v}[i]$)
$SecretKey_{cpa}$ = POLVEC$_q$2BS($\boldsymbol{s}$)
pk = POLVEC$_p$2BS($\boldsymbol{v}_p$)
$PublicKey_{cpa} = seed_{\boldsymbol{A}} \parallel pk$
return ($PublicKey_{cpa}$, $SecretKey_{cpa}$)

10.3.2 Saber.PKE.Enc

This function receives a 256-bit message m, a random seed $seed_{enc}$ of length `SABER_SEEDBYTES` and the public key $PublicKey_{cpa}$ as the inputs and computes the corresponding ciphertext $CipherText_{cpa}$. The steps are described in Alg. 24.

10.3.3 Saber.PKE.Dec

This function receives Saber.PKE.Enc generated $CipherText_{cpa}$ and Saber.PKE.KeyGen generated $SecretKey_{cpa}$ as inputs and computes the decrypted message m. The steps are shown in Alg. 25.

10.4 IND-CCA KEM

The IND-CCA KEM consists of 3 algorithms.

Algorithm 24: Algorithm Saber.PKE.Enc for INC-CPA encryption

Input: m: message bit string of length 256,
$seed_{\boldsymbol{s'}}$: random byte string of length SABER_SEEDBYTES,
$PublicKey_{cpa}$: public key generated using Saber.PKE.KeyGen
Output: $CipherText_{cpa}$: byte string of ciphertext

Extract pk and $seed_{\boldsymbol{A}}$ from $PublicKey_{cpa} = (pk \parallel seed_{\boldsymbol{A}})$
$\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$
$\boldsymbol{s'} = \mathsf{GenSecret}(seed_{\boldsymbol{s'}})$
$\boldsymbol{v} = \mathsf{MatrixVectorMul}(\boldsymbol{A}, \boldsymbol{s'})$
Instantiate vector object $\boldsymbol{v}_p \in R_p^{l \times 1}$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
    $\boldsymbol{v}_p[i] = \mathsf{Round}(\boldsymbol{v}[i])$
$ct = \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{v}_p)$
$\boldsymbol{v'} = \mathsf{BS2POLVEC}_p(pk)$
$pol_p = \mathsf{VectorMul}(\boldsymbol{v'}, \boldsymbol{s'}, p)$
$m_p = \mathsf{MSG2POL}(m)$
$m_p = m_p + pol_p \mod p$
$rec = \mathsf{ReconDataGen}(m_p)$
$CipherText_{cpa} = (rec \parallel ct)$
return $CipherText_{cpa}$

Algorithm 25: Algorithm Saber.PKE.Dec for IND-CPA decryption

Input: $CipherText_{cpa}$: byte string of ciphertext generated using Saber.PKE.Enc,
$SecretKey_{cpa}$: byte string of secret key generated using Saber.PKE.KeyGen
Output: m: decrypted message bit string of length 256

$\boldsymbol{s} = \mathsf{BS2POLVEC}_q(SecretKey_{cpa})$
Extract $(rec \parallel ct) = CipherText$
$\boldsymbol{b} = \mathsf{BS2POLVEC}_p(ct)$
$v' = \mathsf{VectorMul}(\boldsymbol{b}, \boldsymbol{s}, p)$
$m' = \mathsf{Recon}(rec, v')$
$m = \mathsf{POL2MSG}(m')$
return (m)

- Saber.KEM.KeyGen, returns public key and the secret key to be used in the key encapsulation.
- Saber.KEM.Encaps, this function takes the public key and generates a session key and the ciphertext of the seed of the session key.
- Saber.KEM.Decaps, this function receives the ciphertext and the secret key and returns the session key corresponding to the ciphertext.

10.4.1 Saber.KEM.KeyGen

This function returns the public key and the secret key in two separate byte arrays of size SABER_PUBLICKEYBYTES and SABER_SECRETKEYBYTES respectively. The function is described in Alg. 26.

Algorithm 26: Algorithm Saber.KEM.KeyGen for generating public and private key pair.

Output: $PublicKey_{cca}$: public key for encapsulation,
$SecretKey_{cca}$: secret key for decapsulation

1 $(PublicKey_{cpa}, SecretKey_{cpa})$ = Saber.PKE.KeyGen()
2 SHA3-256($hash\_pk$, $PublicKey_{cpa}$, SABER_INDCPA_PUBKEYBYTES)
3 randombytes(z, SABER_KEYBYTES)
4 $SecretKey_{cca} = (z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa})$
5 $PublicKey_{cca} = PublicKey_{cpa}$
6 **return** $(PublicKey_{cca}, SecretKey_{cca})$

10.4.2 Saber.KEM.Encaps

This function generates a session key and the ciphertext corresponding the key. The algorithm is described in Alg 27.

Algorithm 27: Algorithm Saber.KEM.Encaps for generating session key and ciphertext.

Input: $PublicKey_{cca}$: public key generated by Saber.KEM.KeyGen
Output: $SessionKey_{cca}$: session key,
$CipherText_{cca}$: cipher text corresponding to the session key

1 randombytes(m, SABER_KEYBYTES)
2 SHA3-256(m, m, SABER_KEYBYTES)
3 SHA3-256($hash\_pk$, $PublicKey_{cca}$, SABER_INDCPA_PUBKEYBYTES)
4 $buf = (hash\_pk \parallel m)$
5 SHA3-512(kr, buf, 2×SABER_KEYBYTES)
6 Split kr in two equal chunks of length SABER_KEYBYTES and obtain $(r \parallel k) = kr$
7 $CipherText_{cca}$ = Saber.PKE.Enc($m, r, PublicKey_{cca}$)
8 SHA3-256(r', $CipherText_{cca}$, SABER_BYTES_CCA_DEC)
9 $kr' = (r' \parallel k)$
10 SHA3-256($SessionKey_{cca}$, kr', 2×SABER_KEYBYTES)
11 **return** $(SessionKey_{cca}, CipherText_{cca})$

10.4.3 Saber.KEM.Decaps

This function returns a secret key by decapsulating the received ciphertext. The algorithm is described in Alg 28.

Algorithm 28: Algorithm Saber.KEM.Decaps for recovering session key from ciphertext

Input: $CipherText_{cca}$: cipher text generated by Saber.KEM.Encaps,
$SecretKey_{cca}$: public key generated by Saber.KEM.KeyGen
Output: $SessionKey_{cca}$: session key

Extract $(z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa}) = SecretKey_{cca}$
$m =$ Saber.PKE.Dec($CipherText_{cca}$, $SecretKey_{cpa}$)
$buf \leftarrow hash\_pk \parallel m$
SHA3-512(kr, buf, 2×SABER_KEYBYTES)
Split kr in two equal chunks of length SABER_KEYBYTES and obtain $(r \parallel k)$
$CipherText'_{cca} =$ Saber.PKE.Enc($m, r, PublicKey_{cpa}$)
$c =$ Verify($CipherText'_{cca}$, $CipherText_{cca}$, SABER_BYTES_CCA_DEC)
SHA3-256(r', $CipherText'_{cca}$, SABER_BYTES_CCA_DEC)
if $c = 0$ **then**
  $temp = (r' \parallel k)$
else
  $temp = (z \parallel k)$
SHA3-256($SessionKey_{cca}$, $temp$, 2×SABER_KEYBYTES)
return $SessionKey_{cca}$

10.5 Implementation constants

The values of the implementation constants used in the algorithms are provided in Table 4.

Table 4: Implementation constants

Constants	LightSaber	Saber	FireSaber
SABER_SEEDBYTES	32	32	32
RECON_SIZE	2	3	5
SABER_INDCPA_PUBKEYBYTES	672	992	1312
SABER_INDCPA_SECRETKEYBYTES	832	1248	1664
SABER_NOISE_SEEDBYTES	32	32	32
SABER_PUBLICKEYBYTES	672	992	1312
SABER_SECRETKEYBYTES	1568	2304	3040
SABER_KEYBYTES	32	32	32
SABER_HASHBYTES	32	32	32
SABER_BYTES_CCA_DEC	736	1088	1472

References

[1] Martin R. Albrecht. *On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL*, pages 103–129. Springer International Publishing, Cham, 2017.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation, 2016. http://cryptojedi.org/papers/#newhopesimple.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. Document ID: 0462d84a3d34b12b75e8f5e4ca032869, http://cryptojedi.org/papers/#newhope.

[4] Sanjeev Arora and Rong Ge. *New Algorithms for Learning in Presence of Errors*, pages 403–415. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[5] Shi Bai and Steven D. Galbraith. *Lattice Decoding Attacks on Binary LWE*, pages 322–337. Springer International Publishing, Cham, 2014.

[6] Abhishek Banerjee, Chris Peikert, and Alon Rosen. *Pseudorandom Functions and Lattices*, pages 719–737. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[7] James Birkett and Alexander W. Dent. Relations among notions of plaintext awareness. In *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, pages 47–64, 2008.

[8] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[9] Yuanmi Chen and Phong Q. Nguyen. *BKZ 2.0: Better Lattice Security Estimates*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[10] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. Lizard: Cut off the tail! practical post-quantum public-key encryption from lwe and lwr. Cryptology ePrint Archive, Report 2016/1126, 2016. http://eprint.iacr.org/2016/1126.

[11] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. Cryptology ePrint Archive, Report 2017/604, 2017. http://eprint.iacr.org/2017/604.

[12] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. Post-quantum ind-cca-secure kem without additional hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[13] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 43–62, 2015.

[14] Thijs Laarhoven. Search problems in cryptography. PhD thesis, Eindhoven University of Technology, 2015. http://www.thijs.com/docs/phd-final.pdf.

[15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, Jun 2015.

[16] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. Cryptology ePrint Archive, Report 2017/1005, 2017. https://eprint.iacr.org/2017/1005.

[17] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1):181–199, Aug 1994.

[18] Ehsan Ebrahimi Targhi and Dominique Unruh. *Post-Quantum Security of the Fujisaki-Okamoto and OAEP Transforms*, pages 192–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

SPHINCS$^+$

Submission to the NIST post-quantum project

Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe

November 30, 2017

Contents

1. Introduction

Hash-based signature schemes were developed as one-time signature schemes in the late 1970s by Lamport [12] and extended to more signatures by Merkle [13]. The security of these schemes is easy to analyze and relies solely on the properties of the used hash function. However, Merkle's tree-based signature scheme required fixing at key-generation time the number of signatures to be made, keeping this number small for performance. Most importantly, the system required users to remember a state: some information to remember how many signatures were already made with the key.

In the 40 years since Lamport's scheme, many ideas improved the performance, practicality, and theoretical foundations of hash-based signatures, culminating in XMSS [6], which is in the last phase of being standardized by the CFRG as the first post-quantum signature scheme. A strong point of these systems is that they need very few security assumptions – the hash function even need not be collision resistant. The only downside of XMSS is that it is stateful, which makes it not fit the standard definition of signature schemes as, e.g., stated in the NIST call for submissions.

SPHINCS [4] was designed by Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, and Wilcox-O'Hearn as a *stateless* hash-based signature scheme and was the first signature scheme to propose parameters to resist quantum cryptanalysis. SPHINCS uses many components from XMSS but works with larger keys and signatures to eliminate the state.

This document is about the SPHINCS$^+$ construction. At a high level, SPHINCS$^+$ works like SPHINCS. The basic idea is to authenticate a huge number of few-time signature (FTS) key pairs using a so-called hypertree. FTS schemes are signature schemes that allow a key pair to produce a small number of signatures, e.g., in the order of ten for our parameter sets.

For each new message, a (pseudo)random FTS key pair is chosen to sign the message. The signature consists then of the FTS signature and the authentication information for that FTS key pair. The authentication information is roughly a hypertree signature, i.e. a signature using a certification tree of Merkle tree signatures.

More specifically, a hypertree is a tree of hash-based many-time signatures (MTS). These many-time signatures allow a key pair to sign a fixed number N of messages – for SPHINCS$^+$ N is a power of 2, for example 256. The MTS key pairs themselves are organized in an N-ary tree with d layers. On the top layer $d-1$ there is a single MTS key pair which is used to sign the public keys of N MTS key pairs that form layer $d-2$. Each of these N MTS key pairs is used to sign another N MTS public keys forming layer $d-3$. This goes on down to the N^{d-1} key pairs on the bottom layer which are used to sign N FTS public keys, each, leading to a total number of N^d authenticated FTS key pairs. The authentication information for an FTS key pair consists of the d MTS signatures that build a path from the FTS key pair to the top MTS tree.

An MTS signature is just a classical Merkle-tree signature in the case of SPHINCS$^+$. It consists of a one-time signature (OTS) on the given message plus the authentication path in the binary hash-tree, authenticating the N OTS key pairs of one MTS key pair.

The public key of SPHINCS$^+$ is essentially the public key of the top level MTS which is just the root node of its binary hash tree and hence, a single hash value. However, actual SPHINCS$^+$ public keys additionally contain a public seed value of the same length as the root node. This is due to technical reasons explained in the detailed specification below.

The SPHINCS$^+$ secret key is just a single secret seed value. From this, all the OTS and

FTS secret keys are generated in a pseudorandom manner. The OTS and FTS secret keys together fully determine the whole virtual structure of an SPHINCS$^+$ key pair.

1.1. SPHINCS$^+$ vs SPHINCS

SPHINCS$^+$ builds on SPHINCS by introducing several improvements:

- Multi-target attack protection: We apply the mitigation techniques from [9] using keyed hash functions. Each hash function call is keyed with a different key and applies different bitmasks. Keys and bitmasks are pseudorandomly generated from an address specifying the context of the call, and a public seed. For this we introduce the notion of tweakable hash functions which in addition to the input value take a public seed and an address.
- Tree-less WOTS$^+$ public key compression: The last nodes of the WOTS$^+$ chains are not compressed using an L-tree but using a single tweakable hash function call. This call again receives an address and a public seed to key this call and to generate a bitmask as long as the input.
- FORS: A HORST key pair does not consist anymore of a single monolithic tree. Instead it consists of k trees of height a The leaves of these trees are the hashes of the 2^a secret key elements. The public key is the tweakable hash of the concatenation of all the root nodes as for the WOTS$^+$ public key.

  A FORS key pair can be used to sign $k2^a$bit message digests. The digest is first split into k strings m_i of length 2^a bits each. Next, every m_i is interpreted as an integer in $[0, 2^a - 1]$. Here m_i selects the m_i-th secret key element of the i-th tree for the signature. The signature also contains the authentication paths for all the selected secret key elements, which means one path of length a per tree. Verification uses the signature to reconstruct the root nodes and compresses them using the tweakable hash.
- Verifiable index selection: The message digest is now computed as follows. First, we deterministically generate randomness

  $$\mathbf{R} = \mathbf{PRF}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M).$$

  Where `OptRand` is a 256 bit value, per default 0 but can be filled with random bits e.g. taken from a TRNG to avoid deterministic signing (this might be desirable to counter side channel attacks). Then we compute message digest and index as

  $$(\mathrm{md}||\mathrm{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}, M)$$

  where $\mathbf{PK} = (\mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root})$ contains the top root node and the public seed. Hence, we can omit the index in the SPHINCS signature as it would be redundant. This allows to tighten HORST security.

1.2. Organization

In this document we give a formal specification of the SPHINCS$^+$ construction. We follow a bottom-up approach to specify SPHINCS$^+$. We start with basic notation. Afterwards we define WOTS$^+$, the OTS used in SPHINCS$^+$. Next, we specify XMSS, the MTS used in

SPHINCS$^+$, and how it is used to do HT signatures. Then, we define FORS, the FTS used, to finally specify SPHINCS$^+$. Afterwards we discuss different instantiations and explain the design rationale. Then we present a security analysis, give performance values and conclude with a discussion of advantages and limitations.

2. Notation

In the following we start defining basic mathematical operations on integers and bit strings. From that we work our way to more specific basic methods used later in the specification.

2.1. Data Types

Bytes and byte strings are the fundamental data types. A byte is a sequence of eight bits. The set of bytes is denoted as $\mathbb{B}$. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string of length 3. An array of byte strings is an ordered, indexed set starting with index 0 in which all byte strings have identical length. We assume big-endian representation for any data types or structures.

2.2. Functions

We define the following functions:

$\lceil x \rceil$(or ceil(x)) : for x a real number returns the smallest integer greater than or equal to x.

$\lfloor x \rfloor$(or floor(x)) : for x a real number returns the largest integer less than or equal to x.

$\log(x)$: for x a non-negative real number returns the logarithm to base 2 of x. In pseudocode this function is written as `lg`.

$\text{Trunc}_\ell(x)$: truncates the bit-string x to the first ℓ bits.

2.3. Operators

When a and b are integers, mathematical operators are defined as follows:

$\hat{}$: a^b denotes the result of a raised to the power of b.

$\cdot$: $a \cdot b$ denotes the product of a and b. This operator is sometimes omitted in the absence of ambiguity, as in usual mathematical notation.

/ : a/b denotes the quotient of a by non-zero b.

% : $a \ \% \ b$ denotes the non-negative remainder of the integer division of a by b.

$+$: $a + b$ denotes the sum of a and b.

$-$: $a - b$ denotes the difference of a and b.

++ : a++ denotes incrementing a by 1, i.e., $a = a + 1$.

<< : $a << b$ denotes a logical left shift of a by b positions, for b being non-negative, i.e., $a \cdot 2^b$.

>> : $a >> b$ denotes a logical right shift of a by b positions, for b being non-negative, i.e. $\text{floor}(a/2^b)$.

The standard order of operations is used when evaluating arithmetic expressions.

Arrays are used in the common way, where the i-th element of an array A is denoted $A[i]$. Byte strings are treated as arrays of bytes where necessary: If X is a byte string, then $X[i]$ denotes its i-th byte, where $X[0]$ is the leftmost, highest order byte.

If A and B are byte strings of equal length, then:

A AND B denotes the bitwise logical conjunction operation.

A XOR B (or $A \oplus B$) denotes the bitwise logical exclusive disjunction operation.

When B is a byte and i is an integer, then $B >> i$ denotes the logical right-shift by i positions.

If X is an x-byte string and Y a y-byte string, then $X||Y$ denotes the concatenation of X and Y, with $X||Y = X[0] \dots X[x-1]Y[0] \dots Y[y-1]$.

2.4. Integer to Byte Conversion (Function `toByte`)

For x and y non-negative integers, we define $Z = \texttt{toByte}(x, y)$ to be the y-byte string containing the binary representation of x in big-endian byte-order.

2.5. Strings of Base-w Numbers (Function `base_w`)

A byte string can be considered as a string of base w numbers, i.e. integers in the set $\{0, \dots, w-1\}$. The correspondence is defined by the function $\texttt{base\_w}(X, w, \texttt{out\_len})$ as follows. Let X be a `len_X`- byte string, and w is an element of the set $\{4, 16, 256\}$, then $\texttt{base\_w}(X, w, \texttt{out\_len})$ outputs an array of `out_len` integers between 0 and $w-1$ (Figure 1). The length `out_len` is REQUIRED to be less than or equal to $8 * \texttt{len\_X}/\log(w)$.

```
# Input: len_X-byte string X, int w, output length out_len
# Output: out_len int array basew

base_w(X, w, out_len) {
  int in = 0;
  int out = 0;
  unsigned int total = 0;
  int bits = 0;
  int consumed;

  for ( consumed = 0; consumed < out_len; consumed++ ) {
      if ( bits == 0 ) {
          total = X[in];
          in++;
          bits += 8;
      }
      bits -= lg(w);
```

```
        basew[out] = (total >> bits) AND (w - 1);
        out++;
    }
    return basew;
}
```

Algorithm 1: `base_w` – Computing the base-w representation

Figure 1: For example, if X is the (big-endian) byte string 0x1234, then `base_w`$(X, 16, 4)$ returns the array $a = \{1, 2, 3, 4\}$.

```
                X (represented as bits)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             X[0]          |          X[1]

          X (represented as base 16 numbers)
+-----------+-----------+-----------+-----------+
|     1     |     2     |     3     |     4     |
+-----------+-----------+-----------+-----------+

                  base_w(X, 16, 4)
+-----------+-----------+-----------+-----------+
|     1     |     2     |     3     |     4     |
+-----------+-----------+-----------+-----------+
    a[0]        a[1]        a[2]        a[3]

                  base_w(X, 16, 3)
+-----------+-----------+-----------+
|     1     |     2     |     3     |
+-----------+-----------+-----------+
    a[0]        a[1]        a[2]

                  base_w(X, 16, 2)
+-----------+-----------+
|     1     |     2     |
+-----------+-----------+
    a[0]        a[1]
```

2.6. Member Functions (Functions `set`, `get`)

To simplify algorithm descriptions, we assume the existence of member functions. If a complex data structure like a public key PK contains a variable X then `PK.getX()` returns the value of X for this public key. Accordingly, `PK.setX(Y)` sets variable X in PK to the value held by Y. Since camelCase is used for member function names, a value z may be referred to as Z in

the function name, e.g. `getZ`.

2.7. Cryptographic (Hash) Function Families

SPHINCS$^+$ makes use of several different function families with cryptographic properties. Every SPHINCS$^+$ instantiation MUST describe how to implement each of the following functions. For the main instantiations given in this document, this will be done using a single (hash) function, i.e., SHA2-256 or SHAKE-128. Specific instantiations are given in Section 7.

SPHINCS$^+$ applies the multi-target mitigation technique from [9], independently keying and randomizing each hash function call in the original SPHINCS. The implementation of this randomization and keying differs for different instantiations as different function families (e.g., SHA2 or SHAKE) have different properties. Hence, we introduce *tweakable* hash functions as a layer of abstraction. All algorithms in this specification use *tweakable* hash functions in place of traditional hash functions. Later, in Section 7, we describe how to implement the tweakable hash functions.

In addition to several tweakable hash functions, SPHINCS$^+$ makes use of two PRFs and a keyed hash function. Input and output length are given in terms of the security parameter n and the message digest length m, both to be defined more precisely in the coming sections.

2.7.1. Tweakable Hash Functions (Functions `T_l`, `F`, `H`)

A *tweakable* hash function takes a public seed $\mathbf{PK}.\mathtt{seed}$ and context information in form of an address $\mathbf{ADRS}$ in addition to the message input. This allows to make the hash function calls for each key pair and position in the virtual tree structure of SPHINCS$^+$ independent from each other. The addressing scheme will be described in Section 2.7.3.

The schemes described in this specification build upon several instantiations of tweakable hash functions of the form

$$\begin{aligned} &\mathbf{T}_\ell : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n} \to \mathbb{B}^n, \\ &\mathrm{md} \leftarrow \mathbf{T}_\ell(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M) \end{aligned}$$

mapping and ℓn-byte message M to an n-byte hash value md using an n-byte seed $\mathbf{PK}.\mathtt{seed}$ and a 32-byte address $\mathbf{ADRS}$. The function $\mathbf{T}_\ell$ is denoted by `T_l` in pseudocode.

There are two special cases which we rename for consistency with previous descriptions of hash-based signature schemes:

$$\begin{aligned} &\mathbf{F} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n \to \mathbb{B}^n, \\ &\mathbf{F} \stackrel{\mathrm{def}}{=} \mathbf{T}_1 \\ &\mathbf{H} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n} \to \mathbb{B}^n \\ &\mathbf{H} \stackrel{\mathrm{def}}{=} \mathbf{T}_2 \end{aligned}$$

2.7.2. PRF and Message Digest (Functions `PRF`, `PRF_msg`, `H_msg`)

SPHINCS$^+$ makes use of a pseudorandom function $\mathbf{PRF}$ for pseudorandom key generation:

$$\mathbf{PRF} : \mathbb{B}^n \times \mathbb{B}^{32} \to \mathbb{B}^n.$$

In addition, SPHINCS$^+$ uses a pseudorandom function $\mathbf{PRF_{msg}}$ to generate randomness for the message compression:

$$\mathbf{PRF_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^n.$$

To compress the message to be signed, SPHINCS$^+$ uses an additional keyed hash function $\mathbf{H_{msg}}$ that can process arbitrary length messages:

$$\mathbf{H_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^m.$$

2.7.3. Hash Function Address Scheme (Structure of ADRS)

An address **ADRS** is a 32-byte value that follows a defined structure. In addition, it comes with `set` methods to manipulate the address. We explain the generation of addresses in the following sections where they are used. Essentially, all functions have to keep track of the current context, updating the addresses after each hash call.

There are five different types of addresses for the different use cases. One type is used for the hashes in WOTS$^+$ schemes, one is used for compression of the WOTS$^+$ public key, the third is used for hashes within the main Merkle tree construction, another is used for the hashes in the Merkle tree in FORS, and the last is used for the compression of the tree roots of FORS. These types largely share a common format. We describe them in more detail, below.

The structure of an address complies with word borders, with a word being 32 bits long in this context. Only the tree address (i.e. the index of a specific subtree in the main tree) is too long to fit a single word: for this, we reserve three words. An address is structured as follows. It always starts with a layer address of one word in the most significant bits, followed by a tree address of three words. These addresses describe the position of a tree within the hypertree. The layer address describes the height of a tree within the hypertree starting from height zero for trees on the bottom layer. The tree address describes the position of a tree within a layer of a multi-tree starting with index zero for the leftmost tree. The next word defines the type of the address. It is set to 0 for a WOTS$^+$ hash address, to 1 for the compression of the WOTS$^+$ public key, to 2 for a hash tree address, to 3 for a FORS address, and to 4 for the compression of FORS tree roots.

We first describe the WOTS$^+$ address (Figure 2). In this case, the type word is followed by the key pair address that encodes the index of the WOTS$^+$ key pair within the specified tree. The next word encodes the chain address (i.e. the index of the chain within WOTS$^+$), followed by a word that encodes the address of the hash function call within the chain. Note that the address of the bottom of the chain is also used to generate the secret keys based on **SK**.`seed`.

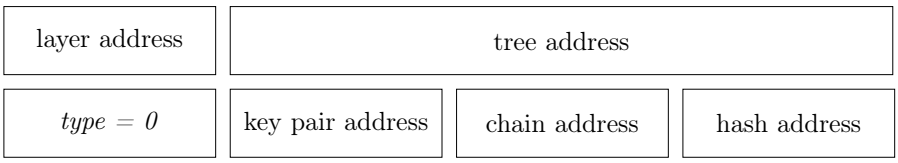


Figure 2: WOTS$^+$ hash address.

The second type (Figure 3) is used to compress the WOTS$^+$ public keys. The type word is set to 1. Similar to the address used within WOTS$^+$, the next word encodes the key pair

address. The remaining two words are not needed, and thus remain zero. We zero pad the address to the constant length of 32 bytes.

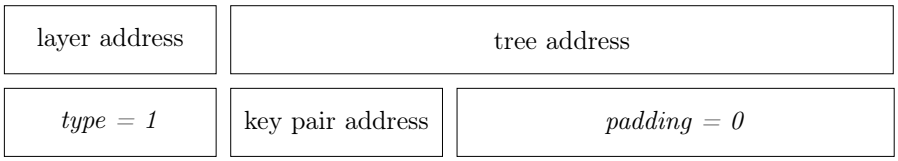


Figure 3: WOTS$^+$ public key compression address.

The third type (Figure 4) addresses the hash functions in the main tree. In this case the type word is set to 2, followed by a zero padding of one word. The next word encodes the height of the tree node that is being computed, followed by a word that encodes the index of this node at that height.

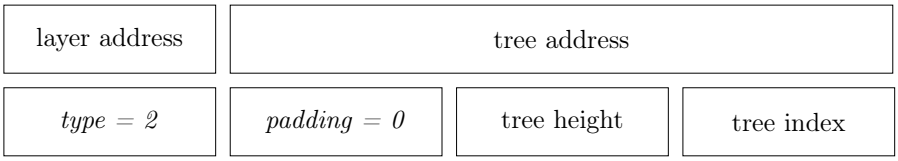


Figure 4: hash tree address.

The next type (Figure 5) is of a similar format, and is used to describe the hash functions in the FORS tree. The type word is set to 3. The key pair address is used to signify which FORS key pair is used, identical to the key pair address in the WOTS$^+$ hash addresses. Its value is the same as that of the WOTS$^+$ key pair that is used to authenticate it, i.e. its index as a leaf in the specified tree. The tree height and tree index fields are used to address the hashes within the FORS tree. This is done like for the above-mentioned hashes in the main tree, with the additional consideration that the tree indices are counted continuously across the different FORS trees. The addresses at tree height 0 are used to generate the leaf nodes from **SK**.`seed`.

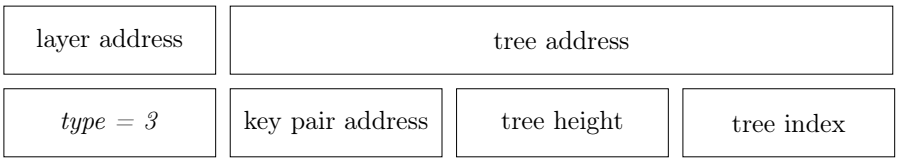


Figure 5: FORS tree address.

The final type (Figure 6) is used to compress the tree roots of the FORS trees. The type word is set to 4. Like the WOTS$^+$ public key compression address, it contains only the address of the FORS key pair, but is padded to the full length.

All fields within these addresses encode unsigned integers. When describing the generation of addresses we use `set` methods that take positive integers and set the bits of a field to the binary representation of that integer, in big-endian notation. Throughout this document, we

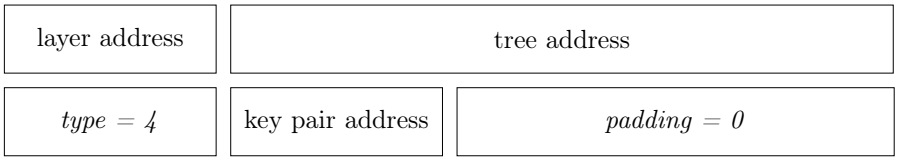


Figure 6: FORS tree roots compression address.

adhere to the convention of assuming that changing the type word of an address (indicated by the use of the `setType()` method) initializes the subsequent three words to zero.

In order to make keeping track of the types easier throughout the pseudo-code in the rest of this document, we refer to them respectively using the constants `WOTS_HASH`, `WOTS_PK`, `TREE`, `FORS_TREE` and `FORS_ROOTS`.

3. WOTS$^+$ One-Time Signatures

This section describes the WOTS$^+$ scheme, in a version similar to [7]. WOTS$^+$ is a OTS scheme; while a private key can be used to sign any message, each private key MUST NOT be used to sign more than a single message. In particular, if a private key is used to sign two different messages, the scheme becomes insecure.

The description given here is tailored to the use inside of SPHINCS$^+$. It assumes that the scheme is used as a subroutine inside a higher order scheme and is not sufficient for a standalone implementation of WOTS$^+$. The section starts with an explanation of parameters. Afterwards, the so-called chaining function, which forms the main building block of the WOTS$^+$ scheme, is explained. A description of the algorithms for key generation and signing follows. Finally, we give an algorithm to compute a WOTS$^+$ public key from a WOTS$^+$ signature. This will be used as a subroutine in SPHINCS$^+$ signature verification.

3.1. WOTS$^+$ Parameters

WOTS$^+$ uses the parameters n and w; they both take positive integer values. These parameters are summarized as follows:

- n: the security parameter; it is the message length as well as the length of a private key, public key, or signature element in bytes.
- w: the Winternitz parameter; it is an element of the set $\{4, 16, 256\}$.

These parameters are used to compute values `len`, $\texttt{len}_1$ and $\texttt{len}_2$:

- `len`: the number of n-byte-string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{n}{\log(w)} \right\rceil, \; \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

The security parameter n is the same as the security parameter n for SPHINCS$^+$. The value of n determines the in- and output length of the tweakable hash function used for WOTS$^+$.

The value of n also determines the length of messages that can be processed by the WOTS$^+$ signing algorithm. The parameter w can be chosen from the set $\{4, 16, 256\}$. A larger value of w results in shorter signatures but slower operations; it has no effect on security. Choices of w are limited to the values 4, 16, and 256 since these values yield optimal trade-offs and easy implementation. WOTS$^+$ parameters are implicitly included in algorithm inputs as needed.

3.2. WOTS$^+$ Chaining Function (Function `chain`)

The chaining function (Algorithm 2) computes an iteration of **F** on an n-byte input using a WOTS$^+$ hash address **ADRS** and a public seed **PK**.`seed`. The address **ADRS** MUST have the first seven 32-bit words set to encode the address of this chain. In each iteration, the address is updated to encode the current position in the chain before **ADRS** is used to process the input by **F**.

In the following, **ADRS** is a 32-byte WOTS$^+$ hash address as specified in Section 2.7.3 and **PK**.`seed` is a n-byte string. The chaining function takes as input an n-byte string X;, a start index i, a number of steps s, as well as **ADRS** and **PK**.`seed`. The chaining function returns as output the value obtained by iterating **F** for s times on input X.

```
#Input: Input string X, start index i, number of steps s, public seed PK.seed,
    address ADRS
#Output: value of F iterated s times on X

chain(X, i, s, PK.seed, ADRS) {
  if ( s == 0 ) {
    return X;
  }
  if ( (i + s) > (w - 1) ) {
    return NULL;
  }
  byte[n] tmp = chain(X, i, s - 1, PK.seed, ADRS);

  ADRS.setHashAddress(i + s - 1);
  tmp = F(PK.seed, ADRS, tmp);
  return tmp;
}
```

Algorithm 2: `chain` – Chaining function used in WOTS$^+$.

3.3. WOTS$^+$ Private Key (Function `wots_SKgen`)

The WOTS$^+$ private key, denoted by sk (s for secret), is a length `len` array of n-byte strings. This private key MUST NOT be used to sign more than one message. This private key is only implicitly used. Therefore, the following is just to support a better understanding of the following algorithms. Each n-byte string in the WOTS$^+$ private key is derived from a secret seed **SK**.`seed` which is part of the SPHINCS$^+$ secret key and a WOTS$^+$ address **ADRS** using **PRF**. The same secret seed is used to generate all secret key values within SPHINCS$^+$. The address used to generate the i-th n-byte string of sk MUST encode the position of the i-th hash chain of this WOTS$^+$ instance within the SPHINCS$^+$ structure.

The following pseudocode (Algorithm 3) describes an algorithm to generate a WOTS$^+$ private key.

```
#Input: secret seed SK.seed, address ADRS
```

```
#Output: WOTS+ private key sk

wots_SKgen(SK.seed, ADRS) {
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    sk[i] = PRF(SK.seed, ADRS);
  }
  return sk;
}
```

Algorithm 3: `wots_SKgen` – Generating a WOTS$^+$ private key.

3.4. WOTS$^+$ Public Key Generation (Function `wots_PKgen`)

A WOTS$^+$ key pair defines a virtual structure that consists of len hash chains of length w. Each of the len stings of n-bytes in the private key defines the start node for one hash chain. The public key is the tweakable hash of the end nodes of these hash chains. To compute the hash chains, the chaining function (Algorithm 2) is used. A WOTS$^+$ hash address **ADRS** and a seed **PK**.seed have to be provided by the calling algorithm as well as a secret seed **SK**.seed. The address **ADRS** MUST encode the address of the WOTS$^+$ key pair within the SPHINCS$^+$ structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.seed used here is public information also available to a verifier. The following pseudocode (Algorithm 4) describes an algorithm for generating the public key pk.

```
#Input: secret seed SK.seed, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk

wots_PKgen(SK.seed, PK.seed, ADRS) {
  wotspkADRS = ADRS; // copy address to create OTS public key address
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    sk = PRF(SK.seed, ADRS);
    tmp[i] = chain(sk[i], 0, w - 1, PK.seed, ADRS);
  }
  wotspkADRS.setType(WOTS_PK);
  wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_len(PK.seed, wotspkADRS, tmp);
  return pk;
}
```

Algorithm 4: `wots_PKgen` – Generating a WOTS$^+$ public key.

3.5. WOTS$^+$ Signature Generation (Function `wots_sign`)

A WOTS$^+$ signature is a length `len` array of n-byte strings. The WOTS$^+$ signature is generated by mapping a message M to `len` integers between 0 and $w-1$. To this end, the message is transformed into $\texttt{len}_1$ base-w numbers using the `base_w` function defined in Section 2.5. Next, a checksum over M is computed and appended to the transformed message as $\texttt{len}_2$ base-w numbers using the `base_w` function. Note that the checksum may reach a maximum integer value of $\texttt{len}_1 \cdot (w-1) \cdot 2^8$ and therefore depends on the parameters n and w. For the parameter sets given in Section 7, a 32-bit unsigned integer is sufficient to hold the checksum. If other parameter sets are used, the size of the variable holding the integer value of the

checksum MUST be sufficiently large. Each of the base-w integers is used to select a node from a different hash chain. The signature is formed by concatenating the selected nodes. A WOTS$^+$ hash address **ADRS**, a public seed **PK**.seed, and a secret seed **SK**.seed have to be provided by the calling algorithm. The address will encode the address of the WOTS$^+$ key pair within a greater structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.seed used here is public information also available to a verifier while the secret seed **SK**.seed is private information. The pseudocode for generating a WOTS$^+$ signature **sig** is shown below (Algorithm 5).

```
#Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
#Output: WOTS+ signature sig

wots_sign(M, SK.seed, PK.seed, ADRS) {
  csum = 0;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    sk = PRF(SK.seed, ADRS);
    sig[i] = chain(sk, 0, msg[i], PK.seed, ADRS);
  }
  return sig;
}
```

Algorithm 5: `wots_sign` – Generating a WOTS+ signature on a message M.

The data format for a signature is given in Figure 7.

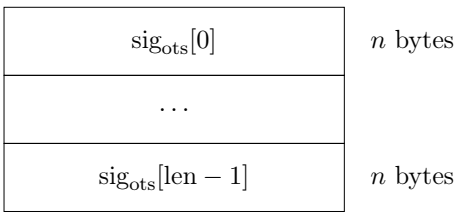


Figure 7: WOTS$^+$ Signature data format.

3.6. WOTS$^+$ Compute Public Key from Signature (Function `wots_pkFromSig`)

SPHINCS$^+$ uses implicit signature verification for WOTS$^+$. In order to verify a WOTS$^+$ signature **sig** on a message M, the verifier computes a WOTS$^+$ public key value from the

signature. This can be done by "completing" the chain computations starting from the signature values, using the base-w values of the message hash and its checksum. This step, called `wots_pkFromSig`, is described below in Algorithm 6. The result of `wots_pkFromSig` then has to be verified. In a standalone version, this would be done by simple comparison. When used in SPHINCS$^+$ the output value is verified by using it to compute a SPHINCS$^+$ public key.

A WOTS$^+$ hash address **ADRS** and a public seed **PK**.`seed` have to be provided by the calling algorithm. The address will encode the address of the WOTS$^+$ key pair within the SPHINCS$^+$ structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.`seed` used here is public information also available to a verifier.

```
#Input: Message M, WOTS+ signature sig, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk_sig derived from sig

wots_pkFromSig(sig, M, PK.seed, ADRS) {
  csum = 0;
  wotspkADRS = ADRS;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    tmp[i] = chain(sig[i], msg[i], w - 1 - msg[i], PK.seed, ADRS);
  }

  wotspkADRS.setType(WOTS_PK);
  wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk_sig = T_len(PK.seed, wotspkADRS, tmp);
  return pk_sig;
}
```

Algorithm 6: `wots_pkFromSig` – Computing a WOTS+ public key from a message and its signature.

4. The SPHINCS$^+$ Hypertree

In this section, we explain how the SPHINCS$^+$ hypertree is built. We first explain how WOTS$^+$ gets combined with a binary hash tree, leading to a fixed input-length version of the eXtended Merkle Signature Scheme (XMSS). Afterwards, we explain how to go to a hypertree from there. The hypertree might be viewed as a fixed input-length version of multi-tree XMSS (XMSS^{MT}).

4.1. (Fixed Input-Length) XMSS

XMSS is a method for signing a potentially large but fixed number of messages. It is based on the Merkle signature scheme. It authenticates $2^{h'}$ WOTS$^+$ public keys using a binary tree of height h'. Hence, an XMSS key pair for height h' can be used to sign $2^{h'}$ different messages. Each node in the binary tree is an n-byte value which is the tweakable hash of the concatenation of its two child nodes. The leaves are the WOTS$^+$ public keys. The XMSS public key is the root node of the tree. In SPHINCS$^+$, the XMSS secret key is the single secret seed that is used to generate all WOTS$^+$ secret keys.

An XMSS signature in the context of SPHINCS$^+$ consists of the WOTS$^+$ signature on the message and the so-called authentication path. The latter is a vector of tree nodes that allow a verifier to compute a value for the root of the tree starting from a WOTS$^+$ signature. A verifier computes the root value and verifies its correctness. A standalone XMSS signature also contains the index of the used WOTS$^+$ key pair. In the context of SPHINCS$^+$ this is not necessary as the SPHINCS$^+$ signature allows to compute the index for each XMSS signature contained.

4.1.1. XMSS Parameters

XMSS has the following parameters:

h' : the height (number of levels - 1) of the tree.

n : the length in bytes of messages as well as of each node.

w : the Winternitz parameter as defined for WOTS$^+$ in the previous Section.

There are $2^{h'}$ leaves in the tree. XMSS signatures are denoted by $\mathbf{SIG}_{\text{XMSS}}$ (`SIG_XMSS` in pseudocode). WOTS$^+$ signatures are denoted by sig.

XMSS parameters are implicitly included in algorithm inputs as needed.

4.1.2. XMSS Private Key

In the context of SPHINCS$^+$, an XMSS private key is the single secret seed $\mathbf{SK}$.`seed` contained in the SPHINCS$^+$ secret key. It is used to generate the WOTS$^+$ secret keys within the structure of an XMSS key pair as described in Section 3.

4.1.3. TreeHash (Function `treehash`)

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine `treehash` (Algorithm 7) accepts a secret seed $\mathbf{SK}$.`seed`, a public seed $\mathbf{PK}$.`seed`, an unsigned integer s (the start index), an unsigned integer z (the target node height), and an address **ADRS** that encodes the address of the containing tree. For the height of a node within a tree, counting starts with the leaves at height zero. The `treehash` algorithm returns the root node of a tree of height z with the leftmost leaf being the WOTS$^+$ pk at index s. It is REQUIRED that $s \,\%\, 2^z = 0$, i.e. that the leaf at index s is a leftmost leaf of a sub-tree of height z. Otherwise the algorithm fails as it would compute non-existent nodes. The `treehash` algorithm described here uses a stack holding up to $(z-1)$ nodes, with the usual stack functions push() and pop(). We furthermore assume that the height of a node (an unsigned integer) is stored alongside a node's value (an n-byte string) on the stack.

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack

treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       ADRS.setType(WOTS_HASH);
       ADRS.setKeyPairAddress(s + i);
       node = wots_PKgen(SK.seed, PK.seed, ADRS);
       ADRS.setType(TREE);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

Algorithm 7: `treehash` – The TreeHash algorithm.

4.1.4. XMSS Public Key Generation (Function `xmss_PKgen`)

The XMSS public key is computed as described in `xmss_PKgen` (Algorithm 10). In the context of SPHINCS$^+$ the XMSS public key PK is the root of the binary hash tree. The root is computed using `treehash`. The public key generation takes a secret seed **SK**.`seed`, a public seed **PK**.`seed`, and an address **ADRS**. The latter encodes the position of this XMSS instance within the SPHINCS$^+$ structure.

```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: XMSS public key PK

xmss_PKgen(SK.seed, PK.seed, ADRS) {
     pk = treehash(SK.seed, 0, h', PK.seed, ADRS)
     return pk;
}
```

Algorithm 8: `xmss_PKgen` – Generating an XMSS public key.

4.1.5. XMSS Signature

An XMSS signature is a $((\texttt{len} + h') * n)$-byte string consisting of

- a WOTS$^+$ signature **sig** taking $\texttt{len} \cdot n$ bytes,
- the authentication path **AUTH** for the leaf associated with the used WOTS$^+$ key pair taking $h' \cdot n$ bytes.

The authentication path is an array of h' n-byte strings. It contains the siblings of the nodes in on the path from the used leaf to the root. It does not contain the nodes on the path

itself. These nodes in **AUTH** are needed by a verifier to compute a root node for the tree from a WOTS$^+$ public key. A node N is addressed by its position in the tree. $N(x, y)$ denotes the yth node on level x with $y = 0$ being the leftmost node on a level. The leaves are on level 0, the root is on level h'. An authentication path contains exactly one node on every layer $0 \leq x \leq (h' - 1)$. For the ith WOTS$^+$ key pair, counting from zero, the jth authentication path node is

$$\mathbf{AUTH}[j] = N\left(j, \lfloor \frac{i}{2^j} \rfloor \oplus 1\right)$$

The computation of the authentication path is discussed in Section 4.1.6.
The data format for a signature is given in Figure 8.

sig (`len` $\cdot\, n$ bytes)
AUTH [0] (n bytes)
...
AUTH [h-1] (n bytes)

Figure 8: XMSS Signature

4.1.6. XMSS Signature Generation (Function `xmss_sign`)

To compute the XMSS signature of a message M in the context of SPHINCS$^+$, the secret seed **SK**.`seed`, the public seed **PK**.`seed`, the index idx of the WOTS$^+$ key pair to be used, and the address **ADRS** of the XMSS instance are needed. First, a WOTS$^+$ signature of the message digest is computed using the WOTS$^+$ instance at index idx. Next, the authentication path is computed.

The node values of the authentication path MAY be computed in any way. The least memory-intensive method is to compute all nodes using the `treehash` algorithm (Algorithm 7). This is described here. Note that the details of how this step is implemented are not relevant to interoperability; it is not necessary to know any of these details in order to perform the signature verification operation.

```
# Input: n-byte message M, secret seed SK.seed, index idx, public seed PK.seed,
    address ADRS
# Output: XMSS signature SIG_XMSS = (sig || AUTH)

xmss_sign(M, SK.seed, idx, PK.seed, ADRS)
     // build authentication path
     for ( j = 0; j < h'; j++ ) {
       k = floor(idx / (2^j)) XOR 1;
       AUTH[j] = treehash(SK.seed, k * 2^j, j, PK.seed, ADRS);
     }
```

```
    ADRS.setType(WOTS_HASH);
    ADRS.setKeyPairAddress(idx);
    sig = wots_sign(M, SK.seed, PK.seed, ADRS);
    SIG_XMSS = sig || AUTH;
    return SIG_XMSS;
}
```

Algorithm 9: `xmss_sign` – Generating an XMSS signature.

4.1.7. XMSS Compute Public Key from Signature (Function `xmss_pkFromSig`)

SPHINCS$^+$ makes use of implicit signature verification of XMSS signatures. An XMSS signature is used to compute a candidate XMSS public key, i.e., the root of the tree. This is used in further computations (signature of the tree above) and implicitly verified by the outcome of that computation. Hence, this specification does not contain an `xmss_verify` method but the method `xmss_pkFromSig`.

The method `xmss_pkFromSig` takes an n-byte message M, an XMSS signature $\mathbf{SIG}_{\text{XMSS}}$, a signature index idx, a public seed **PK**.seed, and an address **ADRS**. The latter encodes the position of the current XMSS instance within the virtual structure of the SPHINCS$^+$ key pair. First, `wots_pkFromSig` is used to compute a candidate WOTS$^+$ public key. This in turn is used together with the authentication path to compute a root node which is then returned. The algorithm `xmss_pkFromSig` is given as Algorithm 10.

```
# Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte message M,
    public seed PK.seed, address ADRS
# Output: n-byte root value node[0]

xmss_pkFromSig(idx, SIG_XMSS, M, PK.seed, ADRS){

    // compute WOTS+ pk from WOTS+ sig
    ADRS.setType(WOTS_HASH);
    ADRS.setKeyPairAddress(idx);
    sig = SIG_XMSS.getWOTSSig();
    AUTH = SIG_XMSS.getXMSSAUTH();
    node[0] = wots_pkFromSig(sig, M, PK.seed, ADRS);

    // compute root from WOTS+ pk and AUTH
    ADRS.setType(TREE);
    ADRS.setTreeIndex(idx);
    for ( k = 0; k < h'; k++ ) {
      ADRS.setTreeHeight(k+1);
      if ( (floor(idx / (2^k)) % 2) == 0 ) {
        ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
        node[1] = H(PK.seed, ADRS, (node[0] || AUTH[k]));
      } else {
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
        node[1] = H(PK.seed, ADRS, (AUTH[k] || node[0]));
      }
      node[0] = node[1];
    }
    return node[0];
}
```

Algorithm 10: `xmss_pkFromSig` – Computing an XMSS public key from an XMSS signature.

4.2. HT: The Hypertee

The SPHINCS$^+$ hypertree HT is a variant of XMSS^{MT}. It is essentially a certification tree of XMSS instances. A HT is a tree of several layers of XMSS trees. The trees on top and intermediate layers are used to sign the public keys, i.e., the root nodes, of the XMSS trees on the respective next layer below. Trees on the lowest layer are used to sign the actual messages, which are FORS public keys in SPHINCS$^+$. All XMSS trees in HT have equal height.

Consider a HT of total height h that has d layers of XMSS trees of height $h' = h/d$. Then layer $d-1$ contains one XMSS tree, layer $d-2$ contains $2^{h'}$ XMSS trees, and so on. Finally, layer 0 contains $2^{h-h'}$ XMSS trees.

4.2.1. HT Parameters

In addition to all XMSS parameters, a HT requires the hypertree height h and the number of tree layers d, specified as an integer value that divides h without remainder. The same tree height $h' = h/d$ and the same Winternitz parameter w are used for all tree layers.

4.2.2. HT Key Generation (Function `ht_PKgen`)

The HT private key is the secret seed **SK**.`seed` which is used to generate all the WOTS$^+$ private keys within the virtual structure spanned by the HT.

The HT public key is the public key (root node) of the single XMSS tree on the top layer. Its computation is explained below. The public key generation takes as input a private and a public seed.

```
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT

ht_PKgen(SK.seed, PK.seed){
     ADRS = toByte(0, 32);
     ADRS.setLayerAddress(d-1);
     ADRS.setTreeAddress(0);
     root = xmss_PKgen(SK.seed, PK.seed, ADRS);
     return root;
}
```

Algorithm 11: `ht_PKgen` – Generating an HT public key.

4.2.3. HT Signature

A HT signature $\mathbf{SIG_{HT}}$ is a byte string of length $(h + d * \texttt{len}) * n$. It consists of d XMSS signatures (of $(h/d + \texttt{len}) * n$ bytes each).

The data format for a signature is given in Figure 9

4.2.4. HT Signature Generation (Function `ht_sign`)

To compute a HT signature $\mathbf{SIG_{HT}}$ of a message M using, `ht_sign` (Algorithm 12) described below uses `xmss_sign` as defined in Section 4.1.6. The algorithm `ht_sign` takes as input a message M, a private seed **SK**.`seed`, a public seed **PK**.`seed`, and an index idx. The index identifies the leaf of the hypertree to be used to sign the message. The HT signature then

XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer 0) $((h/d + \texttt{len}) \cdot n$ bytes)
XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer 1) $((h/d + \texttt{len}) \cdot n$ bytes)
...
XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer $d-1$) $((h/d + \texttt{len}) \cdot n$ bytes)

Figure 9: HT signature

consists of a stack of XMSS signatures using the XMSS trees on the path from the leaf with index idx to the top tree. Note that idx is passed as two separate arguments, split into an index to address the specific tree and the leaf index within that tree. This allows for a somewhat higher hypertree, as one can use a 64-bit integer for tree_idx to support parameters that conform to $h < 64 + h/d$. This matches the parameters in this specification If other parameter sets are used that allow greater h, the data type of tree_idx MUST be adapted accordingly.

Algorithm `ht_sign` uses `xmss_pkFromSig` to compute the root node of an XMSS instance after that instance was used for signing. An alternative is to use `xmss_PKgen`. However, `xmss_PKgen` rebuilds the whole tree while `xmss_pkFromSig` only does one call to `wots_pkFromSig` and $(h' - 1)$ calls to **H**. The algorithm `ht_sign` as described below is just one way to generate a HT signature. Other methods MAY be used as long as they generate the same output.

```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index
    idx_tree, leaf index idx_leaf
# Output: HT signature SIG_HT

ht_sign(M, SK.seed, PK.seed, idx_tree, idx_leaf) {
     // init
     ADRS = toByte(0, 32);

     // sign
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     SIG_tmp = xmss_sign(M, SK.seed, idx_leaf, PK.seed, ADRS);
     SIG_HT = SIG_HT || SIG_tmp;
     root = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx_tree;
        idx_tree = (h - j * (h / d)) most significant bits of idx_tree;
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idx_tree);
        SIG_tmp = xmss_sign(root, SK.seed, idx_leaf, PK.seed, ADRS);
        SIG_HT = SIG_HT || SIG_tmp;
        if ( j < d - 1 ) {
           root = xmss_pkFromSig(idx_leaf, SIG_tmp, root, PK.seed, ADRS);
        }
     }
     return SIG_HT;
```

```
}
```

Algorithm 12: `ht_sign` – Generating an HT signature

4.2.5. HT Signature Verification (Function `ht_verify`)

HT signature verification (Algorithm 13) can be summarized as d calls to `xmss_pkFromSig` and one comparison with a given value. HT signature verification takes a message M, a signature $\mathbf{SIG_{HT}}$, a public seed $\mathbf{PK}$.seed, an index idx (split into a tree index and a leaf index, as above), and a HT public key $\mathbf{PK}_{\mathrm{HT}}$.

```
# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree,
    leaf index idx_leaf, HT public key PK_HT.
# Output: Boolean

ht_verify(M, SIG_HT, PK.seed, idx_tree, idx_leaf, PK_HT){
     // init
     ADRS = toByte(0, 32);

     // verify
     SIG_tmp = SIG_HT.getXMSSSignature(0);
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     node = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx;
        idx_tree = (h - j * h / d) most significant bits of idx;
        SIG_tmp = SIG_HT.getXMSSSignature(j);
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idx_tree);
        node = xmss_pkFromSig(idx_leaf, SIG_tmp, node, PK.seed, ADRS);
     }
     if ( node == PK_HT ) {
       return true;
     } else {
       return false;
     }
}
```

Algorithm 13: `ht_verify` – Verifying a HT signature $\mathbf{SIG_{HT}}$ on a message M using a HT public key $\mathbf{PK}_{\mathrm{HT}}$

5. FORS: Forest Of Random Subsets

The SPHINCS$^+$ hypertree HT is not used to sign the actual messages but the public keys of FORS instances which in turn are used to sign message digests. FORS (pronounced [fɔːrs]), short for forest of random subsets, is a few-time signature scheme (FTS). FORS is an improvement of HORST [4] which in turn is a variant of HORS [17]. For security it is essential that the input to FORS is the output of a hash function. In the following we describe FORS as acting on bit strings.

FORS uses parameters k and $t = 2^a$ (example parameters are $t = 2^{15}, k = 10$). FORS signs strings of length ka bits. Here, we deviate from defining sizes in bytes as the message

length in bits might not be a multiple of eight. The private key consists of kt random n-byte strings grouped into k sets, each containing t n-byte strings. The private key values are pseudorandomly generated from the main private seed **SK**.`seed` in the SPHINCS$^+$ private key. In SPHINCS$^+$, the FORS private key values are only temporarily generated as an intermediate result when computing the public key or a signature.

The FORS public key is a single n-byte hash value. It is computed as the tweakable hash of the root nodes of k binary hash trees. Each of these binary hash trees has height a and is used to authenticate the t private key values of one of the k sets. Accordingly, the leaves of a tree are the (tweakable) hashes of the values in its private key set.

A signature on a string M consists of k private key values – one per set of private key elements – and the associated authentication paths. To compute the signature, md is split into k a-bit strings. Next, each of these bit strings is interpreted as an integer between 0 and $t-1$. Each of these integers is used to select one private key value from a set. I.e., if the first integer is i, the ith private key element of the first set gets selected and so on. The signature consists of the selected private key elements and the associated authentication paths.

SPHINCS$^+$ uses implicit verification for FORS, only using a method to compute a candidate public key from a signature. This is done by computing root nodes of the k trees using the indices computed from the input string as well as the private key values and authentication paths form the signature. The tweakable hash of these roots is then returned as candidate public key.

We now describe the parameters and algorithms for FORS.

5.1. FORS Parameters

FORS uses the parameters n, k, and t; they all take positive integer values. These parameters are summarized as follows:

- n: the security parameter; it is the length of a private key, public key, or signature element in bytes.
- k: the number of private key sets, trees and indices computed from the input string.
- t: the number of elements per private key set, number of leaves per hash tree and upper bound on the index values. The parameter t MUST be a power of 2. If $t = 2^a$, then the trees have height a and the input string is split into bit strings of length a.

Inputs to FORS are bit strings of length $k \log t$.

5.2. FORS Private Key (Function `fors_SKgen`)

In the context of SPHINCS$^+$, a FORS private key is the single private seed **SK**.`seed` contained in the SPHINCS$^+$ private key. It is used to generate the kt n-byte private key values using **PRF** with an address. While these values are logically grouped into a two-dimensional array, for implementations it makes sense to assume they are in a one-dimensional array of length kt. The jth element of the ith set is then stored at $\mathsf{sk}[ik + j]$. To generate one of these elements, a FORS address **ADRS** is used, that encodes the position of the FORS key pair within SPHINCS$^+$ and has tree height set to 0 and leaf index set to $ik + j$:

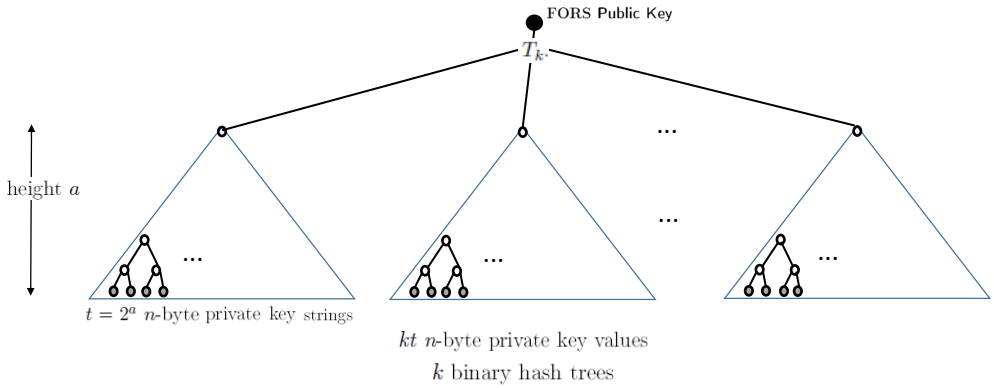


Figure 10: FORS trees and PK

```
#Input: secret seed SK.seed, address ADRS, secret key index idx = ik+j
#Output: FORS private key sk

fors_SKgen(SK.seed, ADRS, idx) {
  ADRS.setTreeHeight(0);
  ADRS.setTreeIndex(idx);
  sk = PRF(SK.seed, ADRS);

  return sk;
}
```

Algorithm 14: `fors_SKgen` – Computing a FORS private key value.

5.3. FORS TreeHash (Function `fors_treehash`)

Before coming to the FORS public key, we have to discuss computation of the trees. For the computation of the n-byte nodes in the FORS hash trees, the subroutine `fors_treehash` is used. It is essentially the same algorithm as `treehash` (Algorithm 7) in Section 4.1. The two differences are how the leaf nodes are computed and how addresses are handled. However, as the addresses are similar, an implementation can implement both algorithms in the same routine easily.

Algorithm `fors_treehash` accepts a secret seed **SK**.`seed`, a public seed **PK**.`seed`, an unsigned integer s (the start index), an unsigned integer z (the target node height), and an address **ADRS** that encodes the address of the FORS key pair. As for `treehash`, the `fors_treehash` algorithm returns the root node of a tree of height z with the leftmost leaf being the hash of the private key element at index s. Here, s is ranging over the whole kt private key elements. It is REQUIRED that $s \,\%\, 2^z = 0$, i.e. that the leaf at index s is a leftmost leaf of a sub-tree of height z. Otherwise the algorithm fails as it would compute non-existent nodes.

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack
```

```
fors_treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       ADRS.setTreeHeight(0);
       ADRS.setTreeIndex(s + i);
       sk = PRF(SK.seed, ADRS);
       node = F(PK.seed, ADRS, sk);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

Algorithm 15: The `fors_treehash` algorithm.

5.4. FORS Public Key (Function `fors_PKgen`)

In the context of SPHINCS$^+$, the FORS public key is never generated alone. It is only generated together with a signature. We include `fors_PKgen` for completeness, a better understanding, and testing. Algorithm `fors_PKgen` takes a private seed **SK**.`seed`, a public seed **PK**.`seed`, and a FORS address **ADRS**. The latter encodes the position of the FORS instance within SPHINCS$^+$. It outputs a FORS public key.

```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK

fors_PKgen(SK.seed, PK.seed, ADRS) {
     forspkADRS = ADRS; // copy address to create FTS public key address

     for(i = 0; i < k; i++){
         root[i] = fors_treehash(SK.seed, i*k, a, PK.seed, ADRS);
     }
     forspkADRS.setType(FORS_ROOTS);
     forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
     pk = T_k(PK.seed, forspkADRS, root);
     return pk;
}
```

Algorithm 16: `fors_PKgen` – Generate a FORS public key.

5.5. FORS Signature Generation (Function `fors_sign`)

A FORS signature is a length $k(\log t + 1)$ array of n-byte strings. It contains k private key values, n-bytes each, and their associated authentication paths, $\log t$ n-byte values each.

The algorithm `fors_sign` takes a $(k \log t)$-bit string M, a private seed **SK**.`seed`, a public seed **PK**.`seed`, and an address **ADRS**. The latter encodes the position of the FORS instance within SPHINCS$^+$. It outputs a FORS signature $\mathbf{SIG}_{\text{FORS}}$.

```
#Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
#Output: FORS signature SIG_FORS

fors_sign(M, SK.seed, PK.seed, ADRS) {
  // compute signature elements
  for(i = 0; i < k; i++){
    // get next index
    unsigned int idx = bits i*t to (i+1)*t - 1 of M;

    // pick private key element
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    SIG_FORS = SIG_FORS || PRF(SK.seed, ADRS);

    // compute auth path
    for ( j = 0; j < a; j++ ) {
      s = floor(idx / (2^j)) XOR 1;
      AUTH[j] = fors_treehash(SK.seed, i * k + s * 2^j, j, PK.seed, ADRS);
    }
    SIG_FORS = SIG_FORS || AUTH;
  }
  return SIG_FORS;
}
```

Algorithm 17: `fors_sign` – Generating a FORS signature on string M.

The data format for a signature is given in Figure 11.

Private key value (tree 0) (n bytes)
AUTH (tree 0) ($\log t \cdot n$ bytes)
...
Private key value (tree $k-1$) (n bytes)
AUTH (tree $k-1$) ($\log t \cdot n$ bytes)

Figure 11: HT signature

5.6. FORS Compute Public Key from Signature (Function `fors_pkFromSig`)

SPHINCS$^+$ makes use of implicit signature verification of FORS signatures. A FORS signature is used to compute a candidate FORS public key. This public key is used in further computations (message for the signature of the XMSS tree above) and implicitly verified by the outcome of that computation. Hence, this specification does not contain a `fors_verify` method but the method `fors_pkFromSig`.

The method `fors_pkFromSig` takes a $k \log t$-bit string M, a FORS signature $\mathbf{SIG}_{\text{FORS}}$, a public seed **PK**.`seed`, and an address **ADRS**. The latter encodes the position of the FORS

instance within the virtual structure of the SPHINCS$^+$ key pair. First, the roots of the k binary hash trees are computed using `fors_treehash`. Afterwards the roots are hashed using the tweakable hash function $\mathbf{T}_k$. The algorithm `fors_pkFromSig` is given as Algorithm 18. The method `fors_pkFromSig` makes use of functions $\mathbf{SIG}_{\text{FORS}}$`.getSK(i)` and $\mathbf{SIG}_{\text{FORS}}$`.getAUTH(i)`. The former returns the ith secret key element stored in the signature, the latter returns the ith authentication path stored in the signature.

```
# Input: FORS signature SIG_FORS, (k lg t)-bit string M, public seed PK.seed,
    address ADRS
# Output: FORS public key

fors_pkFromSig(SIG_FORS, M, PK.seed, ADRS){

  // compute roots
  for(i = 0; i < k; i++){
    // get next index
    unsigned int idx = bits i*t to (i+1)*t - 1 of M;

    // compute leaf
    sk = SIG_FORS.getSK(i);
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    node[0] = F(PK.seed, ADRS, sk);

    // compute root from leaf and AUTH
    auth = SIG_FORS.getAUTH(i);
    ADRS.setTreeIndex(idx);
    for ( j = 0; j < a; j++ ) {
      ADRS.setTreeHeight(j+1);
      if ( (floor(idx / (2^j)) % 2) == 0 ) {
        ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
        node[1] = H(PK.seed, ADRS, (node[0] || auth[j]));
      } else {
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
        node[1] = H(PK.seed, ADRS, (auth[j] || node[0]));
      }
      node[0] = node[1];
    }
    root[i] = node[0];
  }

  forspkADRS = ADRS; // copy address to create FTS public key address
  forspkADRS.setType(FORS_ROOTS);
  forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_k(PK.seed, forspkADRS, root);
  return pk;
}
```

Algorithm 18: `fors_pkFromSig` – Compute a FORS public key from a FORS signature.

6. SPHINCS$^+$

We now have all ingredients to describe our main construction SPHINCS$^+$. Essentially, SPHINCS$^+$ is an orchestration of the methods and schemes described before. It only adds randomized message compression and verifiable index generation.

6.1. SPHINCS$^+$ Parameters

SPHINCS$^+$ has the following parameters:

n : the security parameter in bytes.

w : the Winternitz parameter as defined in Section 3.1.

h : the height of the hypertree as defined in Section 4.2.1.

d : the number of layers in the hypertree as defined in Section 4.2.1.

k : the number of trees in FORS as defined in Section 5.1.

t : the number of leaves of a FORS tree as defined in Section 5.1.

All the restrictions stated in the previous sections apply. Recall that we use $a = \log t$. Moreover, from these values the values m and `len` are computed as

- m: the message digest length in bytes. It is computed as

$$m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor.$$

  While only $h + k \log t$ bits would be needed, using the longer m as defined above simplifies implementations significantly.

- `len`: the number of n-byte string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{n}{\log(w)} \right\rceil, \quad \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

In the following, we assume that all algorithms have access to these parameters.

6.2. SPHINCS$^+$ Key Generation (Function `spx_keygen`)

The SPHINCS$^+$ private key contains two elements. First, the n-byte secret seed **SK**.`seed` which is used to generate all the WOTS$^+$ and FORS private key elements. Second, an n-byte PRF key **SK**.`prf` which is used to deterministically generate a randomization value for the randomized message hash.

The SPHINCS$^+$ public key also contains two elements. First, the HT public key, i.e. the root of the tree on the top layer. Second, an n-byte public seed value **PK**.`seed` which is sampled uniformly at random.

As `spx_sign` does not get the public key, but needs access to **PK**.`seed` (and possibly to **PK**.`root` for fault attack mitigation), the SPHINCS$^+$ secret key contains a copy of the public key.

The description of algorithm `spx_keygen` assumes the existence of a function `sec_rand` which on input i returns i-bytes of cryptographically strong randomness.

```
# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)

spx_keygen( ){
     SK.seed = sec_rand(n);
     SK.prf = sec_rand(n);
     PK.seed = sec_rand(n);
     PK.root = ht_PKgen(SK.seed, PK.seed);
     return ( (SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root) );
}
```

Algorithm 19: `spx_keygen` – Generate a SPHINCS$^+$ key pair.

The format of a SPHINCS$^+$ private and public key is given in Figure 12.

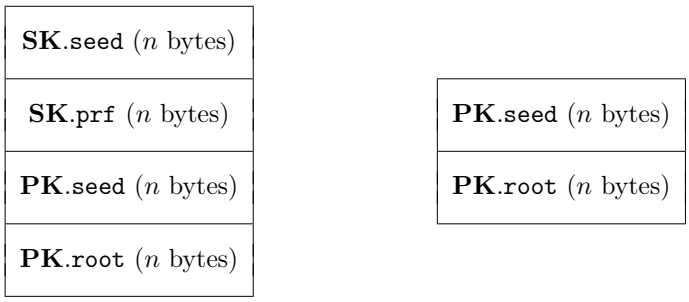


Figure 12: Left: SPHINCS$^+$ secret key. Right: SPHINCS$^+$ public key.

6.3. SPHINCS$^+$ Signature

A SPHINCS$^+$ signature $\mathbf{SIG_{HT}}$ is a byte string of length $(1+k(a+1)+h+d\texttt{len})n$. It consists of an n-byte randomization string R, a FORS signature $\mathbf{SIG_{FORS}}$ consisting of $k(a+1)$ n-byte strings, and a HT signature $\mathbf{SIG_{HT}}$ of $(h+d\texttt{len})n$ bytes.

The data format for a signature is given in Figure 9

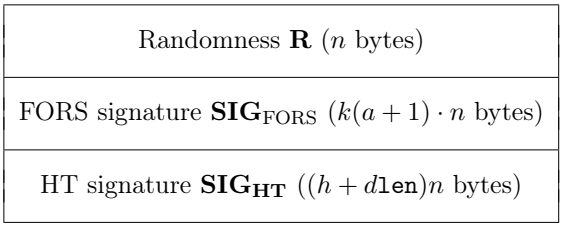


Figure 13: SPHINCS$^+$ signature

6.4. SPHINCS$^+$ Signature Generation (Function `spx_sign`)

Generating a SPHINCS$^+$ signature consists of four steps. First, a random value $\mathbf{R}$ is pseudorandomly generated. Next, this is used to compute a m byte message digest which is split

into a $\lfloor (k \log t + 7)/8 \rfloor$-byte partial message digest `tmp_md`, a $\lfloor (h - h/d + 7)/8 \rfloor$-byte tree index `tmp_idx_tree`, and a $\lfloor (h/d + 7)/8 \rfloor$-byte leaf index `tmp_idx_leaf`. Next, the actual values md, `idx_tree`, and `idx_leaf` are computed by extracting the necessary number of bits. The partial message digest md is then signed with the `idx_leaf`-th FORS key pair of the `idx_tree`-th XMSS tree on the lowest HT layer. The public key of the FORS key pair is then signed using HT. As described in Section 4.2.3, the index is never actually used as a whole, but immediately split into a tree index and a leaf index, for ease of implementation.

When computing **R**, the PRF takes a n-byte string `opt` which is initialized with zero but can be overwritten with randomness if the global variable RANDOMIZE is set. This option is given as otherwise SPHINCS$^+$ signatures would be always deterministic. This might be problematic in some settings. See Section 9 and Section 11 for more details.

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG

spx_sign(M, SK){
     // init
     ADRS = toByte(0, 32);

     // generate randomizer
     opt = toByte(0, 32);
     if(RANDOMIZE){
       opt = rand(n);
     }
     R = PRF_msg(SK.prf, opt, M);
     SIG = SIG || R;

     // compute message digest and index
     digest = H_msg(R, PK.seed, PK.root, M);
     tmp_md = first floor((ka +7)/ 8) bytes of digest;
     tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
     tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;

     md = first ka bits of tmp_md;
     idx_tree = first h - h/d bits of tmp_idx_tree;
     idx_leaf = first h/d bits of tmp_idx_leaf;

     // FORS sign
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     ADRS.setType(FORS_TREE);
     ADRS.setKeyPairAddress(idx_leaf);

     SIG_FORS = fors_sign(md, SK.seed, PK.seed, ADRS);
     SIG = SIG || SIG_FORS;

     // get FORS public key
     PK_FORS = fors_pkFromSig(SIG_FORS, M, PK.seed, ADRS);

     // sign FORS public key with HT
     ADRS.setType(TREE);
     SIG_HT = ht_sign(PK_FORS, SK.seed, PK.seed, idx_tree, idx_leaf);
     SIG = SIG || SIG_HT;

     return SIG;
```

```
}
```

Algorithm 20: `spx_sign` – Generating a SPHINCS$^+$ signature

6.5. SPHINCS$^+$ Signature Verification (Function `spx_verify`)

SPHINCS$^+$ signature verification (Algorithm 21) can be summarized as recomputing message digest and index, computing a candidate FORS public key, and verifying the HT signature on that public key. Note that the HT signature verification will fail if the FORS public key is not matching the real one (with overwhelming probability). SPHINCS$^+$ signature verification takes a message M, a signature **SIG**, and a SPHINCS$^+$ public key **PK**.

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean

spx_verify(M, SIG, PK){
     // init
     ADRS = toByte(0, 32);
     R = SIG.getR();
     SIG_FORS = SIG.getSIG_FORS();
     SIG_HT = SIG.getSIG_HT();


     // compute message digest and index
     digest = H_msg(R, PK.seed, PK.root, M);
     tmp_md = first floor((ka +7)/ 8) bytes of digest;
     tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
     tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;

     md = first ka bits of tmp_md;
     idx_tree = first h - h/d bits of tmp_idx_tree;
     idx_leaf = first h/d bits of tmp_idx_leaf;

     // compute FORS public key
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     ADRS.setType(FORS_TREE);
     ADRS.setKeyPairAddress(idx_leaf);

     PK_FORS = fors_pkFromSig(SIG_FORS, md, PK.seed, ADRS);

     // verify HT signature
     ADRS.setType(TREE);
     return ht_verify(PK_FORS, SIG_HT, PK.seed, idx_tree, idx_leaf, PK.root);
}
```

Algorithm 21: `spx_verify` – Verify a SPHINCS$^+$ signature **SIG** on a message M using a SPHINCS$^+$ public key **PK**

7. Instantiations

This section discusses instantiations for SPHINCS$^+$. SPHINCS$^+$ can be viewed as a signature template. It is a way to build a signature scheme by instantiating the cryptographic function

families used. We consider different ways to implement the cryptographic function families as different signature systems. Orthogonal to instantiating the cryptographic function families are parameter sets. Parameter sets assign specific values to the SPHINCS$^+$ parameters described in Section 7.1 below.

In this section, we first define the requirements on parameters and discuss existing trade-offs between security, sizes, and speed controlled by the different parameters. Then we propose 6 different parameter sets that match NIST security levels I, III, and V (2 parameter sets per security level). Afterwards we propose three different instantiations for the cryptographic function families of SPHINCS$^+$. These instantiation are indeed three different signature schemes. We propose SPHINCS$^+$-SHAKE256, SPHINCS$^+$-SHA-256, and SPHINCS$^+$-Haraka. The former two use the cryptographic hash functions defined in FIPS PUB 202, respectively FIPS PUB 180, to instantiate the cryptographic function families. The latter uses a new cryptographic (hash) function called Haraka, proposed in [11].

7.1. SPHINCS$^+$ Parameter Sets

SPHINCS$^+$ is described by the following parameters already described in the previous sections. All parameters take positive integer values.

n : the security parameter in bytes.

w : the Winternitz parameter.

h : the height of the hypertree.

d : the number of layers in the hypertree.

k : the number of trees in FORS.

t : the number of leaves of a FORS tree.

Recall that we use $a = \log t$. Moreover, from these values the values m and `len` are computed as

- m: the message digest length in bytes. It is computed as $m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$.
- `len`: the number of n-byte string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{n}{\log(w)} \right\rceil, \quad \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

We now repeat the roles of, requirements on, and properties of these parameters. Afterwards, we give several formulas that show their exact influence on performance and security.

The security parameter n is also the output length of all cryptographic function families besides $\mathbf{H_{msg}}$. Therefore, it largely determines which security level a parameter set reaches. It is also the size of virtually any node within the SPHINCS$^+$ structure and thereby also the size of all elements in a signature, i.e., the signature size is a multiple of n.

The Winternitz parameter w determines the number and length of the hash chains per WOTS$^+$ instance. A greater value for w linearly increases the length of the hash chains

but logarithmically reduces their number. The number of hash chains exactly corresponds to the number of n-byte values in a WOTS$^+$ signature. Thereby it largely influences the size of a SPHINCS$^+$ signature. The product of the number and the length of hash chains directly correlates with signing speed as essentially all time in HT signature generation is spent computing WOTS$^+$ public keys. Therefore, greater w means shorter signatures but slower signing. However, note the exponential gap. The bigger w gets, the more expensive is the signature size reduction. The Winternitz parameter does not influence SPHINCS$^+$ security.

The height of the hypertree h determines the number of FORS instances. Hence, it determines the probability that a FORS key pair is used several times, given the number of signatures made with a SPHINCS$^+$ key pair. Hence, the height has a direct impact on security: A taller hypertree gives more security. On the other hand, a taller tree leads to larger signatures.

The number of layers d is a pure performance trade-off parameter and does not influence security. It determines the number of layers of XMSS trees in the hypertree. Hence, d must divide h without remainder. The parameter d thereby defines the height of the XMSS trees used. The greater d, the smaller the subtrees, the faster signing. However, d also controls the number of layers and thereby the number of WOTS$^+$ signatures within a HT and thereby a SPHINCS$^+$ signature.

The parameters k and t determine the performance and security of FORS. The number of leaves of a tree in FORS t must be a power of two while k can be chosen freely. A smaller t generally leads to smaller and faster signatures. However, for a given security level a smaller t requires a greater k which increases signature size and slows down signing. Hence, it is important to balance these two parameters. This is best done using the formulas below.

The message digest length m is the output length of $\mathbf{H_{msg}}$ in bytes. It is $\lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$ bytes.

The number $\texttt{len}$ of chains in a WOTS$^+$ key pair determines the WOTS$^+$ signature size.

7.1.1. Influence of Parameters on Security and Performance

In the following we provide formulas to compute speed, size and security for a given SPHINCS$^+$ parameter set. This supports parameter selection. We also provide a SAGE script in Appendix A.

Key Generation. Generating the SPHINCS$^+$ private key and $\mathbf{PK}.\texttt{seed}$ requires three calls to a secure random number generator. Next we have to generate the top tree. For the leaves we need to do $2^{h/d}$ WOTS$^+$ key generations ($\texttt{len}$ calls to $\mathbf{PRF}$ for generating the $\texttt{sk}$ and $w\texttt{len}$ calls to $\mathbf{F}$ for the $\texttt{pk}$) and we have to compress the WOTS$^+$ public key (one call to $T_{\texttt{len}}$). Computing the root of the top tree requires $(2^{h/d} - 1)$ calls to $\mathbf{H}$.

Signing. For randomization and message compression we need one call to $\mathbf{PRF}$, $\mathbf{PRF_{msg}}$ and $\mathbf{H_{msg}}$. The FORS signature requires kt calls to $\mathbf{PRF}$ and $\mathbf{F}$. Further, we have to compute the root of k binary trees of height $\log t$ which adds $k(t-1)$ calls to $\mathbf{H}$. Finally, we need one call to T_k. Next, we compute one HT signature which consists of d trees similar to the key generation. Hence, we have to do $d(2^{h/d})$ times $\texttt{len}$ calls to $\mathbf{PRF}$ and $w\texttt{len}$ calls to $\mathbf{F}$ as well as $d(2^{h/d})$ calls to $\mathbf{T}_{\texttt{len}}$. For computing the root of each tree we get additionally $d(2^{h/d} - 1)$ calls to $\mathbf{H}$.

Table 1: Overview of the number of function calls we require for each operation. We omit the single calls to $\mathbf{H_{msg}}, \mathbf{PRF_{msg}}$, and $\mathbf{T}_k$ for signing and single calls to $\mathbf{H_{msg}}$ and $\mathbf{T}_k$ for verification as they are negligible when estimating speed.

	$\mathbf{F}$	$\mathbf{H}$	$\mathbf{PRF}$	$T_{\texttt{len}}$
Key Generation	$2^{h/d}w\texttt{len}$	$2^{h/d}-1$	$2^{h/d}\texttt{len}$	$2^{h/d}$
Signing	$kt + d(2^{h/d})w\texttt{len}$	$k(t-1) + d(2^{h/d}-1)$	$kt + d(2^{h/d})\texttt{len} + 1$	$d2^{h/d}$
Verification	$kt + dw\texttt{len}$	$k(t-1) + h$	kt	d

Table 2: Key and signature sizes

	SK	PK	Sig
Size	$4n$	$2n$	$(h + k(\log t + 1) + d \cdot len + 1)n$

Verification. First we need to compute the message hash using $\mathbf{H_{msg}}$. We need to do one FORS verification which requires kt calls to $\mathbf{PRF}$ and $\mathbf{F}$, $k(t-1)$ calls to $\mathbf{H}$ and one call to T_k for hashing the roots. Next, we have to verify d XMSS signatures which takes $< w\texttt{len}$ calls to $\mathbf{F}$ and one call to $T_{\texttt{len}}$ each for WOTS$^+$ signature verification. It also needs dh/d calls to $\mathbf{H}$ for the d root computations.

The size of the SPHINCS$^+$ private and public keys along with the signature can be deduced from Section Section 6 as shown in Table 2.

The classical security level, or bit security of SPHINCS$^+$ against generic attacks can be computed as

$$b = -\log\left(\frac{1}{2^{8n}} + \sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right).$$

The quantum security level, or bit security of SPHINCS$^+$ against generic attacks can be computed as

$$b = -\frac{1}{2}\log\left(\frac{1}{2^{8n}} + \sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right).$$

Here, we are neglecting the small constant factors inside the logarithm. For details see Section 9.

7.1.2. Proposed Parameter Sets and Security Levels

As explained in the previous subsection, even for a fixed security level the design of SPHINCS$^+$ supports many different tradeoffs between signature size and speed. In Table 3 we list 6 parameter sets that—together with the cycle counts given in Table 4— illustrates how these tradeoffs can be used to obtain concrete parameter sets optimizing for signature size and concrete parameter sets optimizing for speed. Specifically, we propose parameter sets achieving security levels 1, 3, and 5; for each of these security levels propose one size-optimized (ending

Table 3: Example parameter sets for SPHINCS$^+$ targeting different security levels and different tradeoffs between size and speed. The column labeled "bitsec" gives the bit security computed as described in Section 9; the column labeled "sec level" gives the security level according to the levels specified in Section 4.A.5 of the Call for Proposals. As explained later, **for Haraka the security level is limited to 2:** i.e., it is 1 for $n = 16$, and 2 for $n = 24$ or $n = 32$.

	n	h	d	$\log(t)$	k	w	bitsec	sec level	sig bytes
SPHINCS$^+$-128s	16	64	8	15	10	16	133	**1**	8 080
SPHINCS$^+$-128f	16	60	20	9	30	16	128	**1**	16 976
SPHINCS$^+$-192s	24	64	8	16	14	16	196	**3**	17 064
SPHINCS$^+$-192f	24	66	22	8	33	16	194	**3**	35 664
SPHINCS$^+$-256s	32	64	8	14	22	16	255	**5**	29 792
SPHINCS$^+$-256f	32	68	17	10	30	16	254	**5**	49 216

on 's' for "small") and one speed-optimized (ending on 'f' for "fast") parameter set. The parameter sets were obtained with the help of a Sage script that we list in Appendix A. In the first line of that script, set the "target bit security" to a desired value (in our case, close to 128 for security level 1, close to 192 for security level 3, and close to 256 for security level 5). The output of the script will be a long list of possible parameters achieving this security level together with the signature size and an estimate of the performance, using the formulas from Section 7.1.1 above.

Note that we did *not* obtain our proposed parameter sets simply by searching this output for the smallest or the fastest option. The reason is that, for example, optimizing for size without caring about speed at all results in signatures of a size of ≈ 15 KB for a bit security of 256, but computing one signature takes more than 20 minutes on our benchmark platform. Such a tradeoff might be interesting for very few select applications, but we cannot think of many applications that would accept such a large time for signing. Instead, the proposed parameter sets are what we consider "non-extreme"; i.e., with a signing time of at most a few seconds in our non-optimized implementation.

The choice of these parameters is orthogonal to the choice of hash function. In Section 7.2 we describe 3 different instantiations of the underlying hash function. Together with the 6 parameter sets listed in Table 3 we obtain 18 different instantiations of SPHINCS$^+$.

7.2. Instantiations of Hash Functions

In this section we define three different signature schemes which are obtained by instantiating the cryptographic function families of SPHINCS$^+$ with SHA-256, SHAKE256, and Haraka. To instantiate the tweakable hash functions, all proposals first use **PRF** to generate pseudorandom *bitmasks* which are then XORed with the input message. The masked messages are denoted as $M^{\oplus}$.

7.2.1. SPHINCS$^+$-SHAKE256

For SPHINCS$^+$-SHAKE256 we define

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}, 8n),\\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1, M_2) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}||M_2^{\oplus}, 8n),\\
\mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M) &= \text{SHAKE256}(\mathbf{R}||\mathbf{PK}.\mathtt{seed}||\mathbf{PK}.\mathtt{root}||M, 8m),\\
\mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \text{SHAKE256}(\mathbf{SEED}||\mathbf{ADRS}, 8n),\\
\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M) &= \text{SHAKE256}(\mathbf{SK}.\mathtt{prf}||\mathtt{OptRand}||M, 8n).
\end{aligned} \tag{1}$$

Generating the Masks. SHAKE256 can be used as an XOF which allows us to generate the bitmasks for arbitrary length messages directly. For a message M with l bits we compute

$$M^{\oplus} = M \oplus \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}, l).$$

7.2.2. SPHINCS$^+$-SHA-256

In a similar way we define the functions for SPHINCS$^+$-SHA-256 as

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1) &= \text{SHA-256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}),\\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1, M_2) &= \text{SHA-256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}||M_2^{\oplus}),\\
\mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M) &= \text{MGF1-SHA-256}(\text{SHA-256}(\mathbf{R}||\mathbf{PK}.\mathtt{seed}||\mathbf{PK}.\mathtt{root}||M), m),\\
\mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \text{SHA-256}(\mathbf{SEED}||\mathbf{ADRS}),\\
\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{seed}, \mathtt{OptRand}, M) &= \text{HMAC-SHA-256}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}||M).
\end{aligned} \tag{2}$$

which uses MGF1 as defined in RFC 2437 and HMAC as defined in FIPS-198-1. Note that MGF1 takes the as last input the output length in bytes.

Generating the Masks. SHA-256 can be turned into a XOF using MGF1 which allows us to generate the bitmasks for arbitrary length messages directly. For a message M with l bytes we compute

$$M^{\oplus} = M \oplus \text{MGF1-SHA-256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}, l).$$

Shorter Outputs. If a parameter set requires an output length $n < 32$-bytes for $\mathbf{F}$, $\mathbf{H}$, $\mathbf{PRF}$, and $\mathbf{PRF_{msg}}$ we take the first n bytes of the output and discard the remaining.

7.2.3. SPHINCS$^+$-Haraka

Our third instantiation is based on the Haraka short-input hash function. Haraka is not a NIST-approved hash function, and since it is new it needs further analysis. We specify SPHINCS$^+$-Haraka as third signature scheme to demonstrate the possible speed-up by using a dedicated short-input hash function.

As the Haraka family only supports input sizes of 256 and 512 bits we extend it with a sponge-based construction based on the 512-bit permutation π. The sponge has a rate of 256-bit respectively a capacity of 256-bit and the number of rounds used in π is 5. The padding scheme is the same as defined in FIPS PUB 202 for SHAKE256.

We denote this sponge as HarakaS(M, d), where M is the padded message and d is the length of the message digest in bits. A 256-bit message block M_i is absorbed into the state S by

$$\text{Absorb}(M, S) : S = \pi(S \oplus (M || \texttt{toByte}(0, 32))). \tag{3}$$

The d-bit hash output h is computed by squeezing blocks of r bits

$$\begin{aligned} \text{Squeeze}(S) : h &= h || \text{Trunc}_{256}(S) \\ S &= \pi(S). \end{aligned} \tag{4}$$

For a more efficient construction we *tweak* the round constants of Haraka using $\mathbf{PK}.\texttt{seed}$.[1] As $\mathbf{PK}.\texttt{seed}$ is the same for all hash function calls for a given key pair we expand $\mathbf{PK}.\texttt{seed}$ using HarakaS and use the result for the round constants in all instantiations of Haraka used in SPHINCS$^+$. In total there are 40 128-bit round constants defined by

$$RC_0, \ldots, RC_{39} = \text{HarakaS}(\mathbf{PK}.\texttt{seed}, 5120). \tag{5}$$

This only has to be done once for each key pair for all subsequent calls to Haraka hence the costs for this are amortized. We denote Haraka with the round constants tweaked by $\mathbf{PK}.\texttt{seed}$ as $\text{Haraka}_{\mathbf{PK}.\texttt{seed}}$. We can now define all functions we need for SPHINCS$^+$-Haraka as

$$\begin{aligned} \mathbf{F}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1) &= \text{Haraka512}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS} || M_1^{\oplus}), \\ \mathbf{H}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1, M_2) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS} || M_1^{\oplus} || M_2^{\oplus}, 8n), \\ \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\texttt{seed}, \mathbf{PK}.\texttt{root}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{R} || \mathbf{PK}.\texttt{root} || M, 8m), \\ \mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \text{Haraka256}_{\mathbf{SEED}}(\mathbf{ADRS}), \\ \mathbf{PRF_{msg}}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{SK}.\texttt{prf} || \texttt{OptRand} || M, 8n). \end{aligned} \tag{6}$$

For $\mathbf{F}$ we pad $M_1^{\oplus}$ with zero if $n < 32$. Note that $\mathbf{H}$ and $\mathbf{H_{msg}}$ will always have a different $\mathbf{ADRS}$ and we therefore do not need any further domain separation.

Generating the Masks. The mask for the message used in $\mathbf{F}$ is generated by computing

$$M_1^{\oplus} = M_1 \oplus \text{Haraka256}(\mathbf{ADRS}) \tag{7}$$

respectively for $\mathbf{H}$

$$M_1^{\oplus} || M_2^{\oplus} = (M_1 || M_2) \oplus \text{Haraka512}(\mathbf{ADRS} || \texttt{toByte}(0, 32)). \tag{8}$$

For all other purposes the masks are generated using HarakaS. For a message M with l bytes we compute

$$M^{\oplus} = M \oplus \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}, l).$$

Shorter Outputs. If a parameter set requires an output length $n <$ 32-bytes for $\mathbf{F}$ and $\mathbf{PRF}$, we take the first n bytes of the output and discard the remaining.

Security Restrictions. Note that our instantiation using Haraka employs the sponge construction with a capacity of 256-bits. Hence, in contrast to SPHINCS$^+$-SHA-256 and SPHINCS$^+$-SHAKE256, SPHINCS$^+$-Haraka reaches security level 2 for 32- and 24-byte outputs and security level 1 for 16-byte outputs.

[1] This is similar to the ideas used for the MDx-MAC construction [16].

8. Design rationale

The design rationale behind SPHINCS$^+$ is to follow the original SPHINCS construction and apply several results from more recent research. The idea behind SPHINCS was as follows. One can build a stateless hash-based signature scheme using a massive binary certification tree and selecting a leaf at random for each message to be signed. The problem with this approach is that the tree has to be extremely high, i.e., a height of about twice the security level would be necessary. This leads to totally unpractical signature sizes. Using a hypertree instead of a binary certification tree allows to trade speed for signature size. However, this is still not sufficient to get practical sizes and speed.

The main new idea in SPHINCS was to not use the leaves directly to sign messages but to use the leaves to certify FTS key pairs. This allowed to massively reduce the total tree height (by a factor about 4). This is due to the fact that the security of an FTS instance degrades with every signature a key pair is used for. Hence, the height of the tree does not have to be such that collisions do only occur with negligible probability anymore. Instead, it has to be ensured that the product of the probability of a γ-times collision on a leaf and the forging probability of an adversary after seeing γ FTS signatures (with the same key pair) is negligible.

From this, it is mainly a question of balancing parameters to find a practical scheme. For the full original reasoning see [4].

In the following we give a more detailed reasoning regarding the changes made to SPHINCS in SPHINCS$^+$, and changes that were discussed by the SPHINCS$^+$ team but got discarded.

8.1. Changes Made

We changed several details of SPHINCS leading to SPHINCS$^+$. The reasoning behind those changes is discussed in the following.

8.1.1. Multi-Target Attack Protection

SPHINCS was designed to be collision-resilient i.e., to not be vulnerable to collision attacks against the used hash function. This had two reasons. First, it allowed to choose a smaller output length at the same security level which led to smaller signatures. Second, collision resistance is a far stronger assumption than the used (second-)preimage resistance and pseudorandomness assumptions.

However, the use of (second-)preimage resistance introduced a new issue as pointed out in [9]: Multi-target attacks. Preimage resistance properties are targeted properties. An adversary is asked to invert the function on a given target value, or to find a second-preimage for a given target value. If it suffices to break the given property for one out of many targets, the adversarial effort is reduced by a factor of the number of targets. To prevent this e apply the mitigation techniques from [9] using keyed hash functions. Each hash function call is keyed with a different key and applies different bitmasks. Keys are derived from, and bitmasks are pseudorandomly generated from a public seed and an address specifying the context of the call. For this we introduce the notion of tweakable hash functions which take in addition to the input value a public seed and an address.

This pseudorandom generation of bitmasks comes at the cost of introducing a random oracle assumption for the PRF used to generate the bitmasks. However, this only applies to

the pseudorandom generation of the bitmasks. I.e., if all bitmasks would be stored in the public key, the scheme would have a standard model security proof (even if these bitmasks where generated using exactly the same way but without giving away the seed). Hence, the security reduction in [9] is in the quantum-accessible random oracle model.

One difference to [9] is that in all instantiations of SPHINCS$^+$, keys are not pseudorandomly generated. Instead, the concatenation of public seed and address is used to practically key the functions. Given how the tweakable hash functions are instantiated, this means that we assume that there do not exist any (exponentially large) subsets of the domain on which second-preimage finding is easy. This assumption holds for any hash function based on the sponge or Merkle-Dåmgard construction, assuming the block or compression function behaves like a random function.

8.1.2. Tree-less WOTS$^+$ Public Key Compression

SPHINCS$^+$ compresses the end nodes of the WOTS$^+$ hash chains with a single call to a tweakable hash function, while SPHINCS used a so called L-tree. The reason to use L-trees in SPHINCS was that this required only two n-byte bitmasks per layer, i.e., $2\lceil \log \texttt{len} \rceil$ bitmasks. A single call to a tweakable hash requires `len` n-byte bitmasks. As the bitmasks were stored in the public key, this meant smaller public keys. Now, that bitmasks are pseudorandomly generated anyway and hence are not stored in the public key anymore, this argument does not apply. On the opposite, tree based compression is slower than using a single call to a tweakable hash with longer input.

8.1.3. FORS

FORS was used to replace HORST. HORST, as its predecessor HORS, had the problem that weak messages existed as recently independently pointed out in [1]. More specifically, in HORST the message is also split into k indexes as for FORS. However, these indexes all selected values from the same single set of secret key values. Hence, if the same index appeared multiple times in a signature, still only a single secret value would be required. In extreme cases this means that for the signature of a message only a single secret value has to be know. FORS prevents this using separate secret value sets per index obtained from the message. Even if a message maps k-times to the same index, the signature now contains k different secret values.

For the same parameters k and t this would mean an increase in signature size and worse speed as now k trees of height $\log t$ have to be computed instead of one and for each signature value an authentication path of length $(\log t) - 1$ is needed. However, due to the strengthened security, we can choose different values for k and t. This in the end leads to smaller signatures than for HORST.

We also considered a method similar to Octopus [2]. The idea is that authentication paths in HORST largely overlap. Hence, it becomes possible to reduce the signature size removing any redundancy in the authentication paths. This comes at the cost of a rather involved method to collect the right nodes as well as variable size signatures. In practice this means that one still has to prepare for the worst case. This worst case indeed still has smaller signatures than HORST. We decided against this option as the FORS signature size matches that of Octopus' worst case signature size. At the same time, FORS gives more flexibility in the choice of k and t, and comes with a far simpler signature and verification method that Octopus.

8.1.4. Verifiable Index Selection

In SPHINCS the index of the HORST instance to be used was pseudorandomly selected. This had the drawback that the index appeared random to a verifier and it was impossible to verify that the index was indeed generated that way. This allowed an adversary a multi-target attack on HORST (similarly for FORS in SPHINCS$^+$). An adversary could first map a message to an index set and then check if the necessary secret values were already uncovered for some HORST key pair. Then it would just select the index of that HORST key pair as index and succeed in forging a signature.

To prevent this attack, we decided to make index generation verifiable. More specifically, we generate the index together with the message digest:

We compute message digest and index as

$$(\text{md}||\text{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}, M)$$

where $\mathbf{PK} = (\mathbf{PK}.\texttt{seed}||\mathbf{PK}.\texttt{root})$ contains the top root node and the public seed.

This way, an adversary can no longer freely choose an index. Indeed, selecting a message immediately also fixes the index. This method has another advantage in addition to avoiding the multi-target attack against FORS/HORST. We can omit the index in the SPHINCS signature as it would be redundant.

8.1.5. Making Deterministic Signing Optional

The pseudorandom generation of randomizer $\mathbf{R}$ now allows to use additional randomness. It takes a n-byte value `OptRand`. Per default `OptRand` is set to 0 but it can be filled with random bits e.g. taken from a TRNG. The randomizer is then computed as

$$\mathbf{R} = \mathbf{PRF}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M).$$

That way, deterministic signing becomes optional. Deterministic signing can be a problem for devices which are susceptible to side-channel attacks as it allows to collect several traces for the exactly same computation by just asking for a signature on the same message multiple times.

We could of course also have replaced $\mathbf{R}$ by a truly random value on default. This would have caused the scheme to become susceptible to bad randomness. The new method prevents this. If `OptRand` is a high entropy string, $\mathbf{R}$ has as much entropy as that string. If `OptRand` is left as zero or has only little entropy, $\mathbf{R}$ is just a pseudorandom value as in SPHINCS.

8.2. Discarded Changes

In Section 8.1.3, we already explained that we discarded the use of an Octopus-like method as we found a better alternative.

One more idea which we discarded on the way was a signature - secret key size trade-off. To further shrink the SPHINCS$^+$ signature size, the top z layers of the hypertree can be merged together into a a single tree of height zh'. That way an SPHINCS$^+$ signature includes $z-1$ less WOTS$^+$ signatures. This decreases the signature size by $n \cdot len(z-1)$ bytes, but typically comes at the cost of speed as now a tree of height zh' has to be computed for each signature generation. This can be prevented by storing the nodes at height ih', where $0 < i < z$, as part of the secret key. These nodes (auxiliary data) can be used to build the authentication

paths to the root of the merged tree without actually computing the whole tree. Indeed, authentication path computation in this case gets faster than computing the authentication paths for z tree layers in the original hypertree. The size of the auxiliary data is $n\sum_{i=1}^{z-1} 2^{ih'}$. While this already grows extremely fast, the real problem turned out to be key generation time. As the full tree still has to be computed once during key generation, key generation time increases. Key generation would now take $2^{zh'}$ WOTS$^+$ key generations.

Initial experiments suggested that key generation time easily moves into the order of minutes already for $z = 2$ while the benefit in signature size is 1KB or 2KB for $w = 256$ and $w = 16$ respectively. In addition, this optimization significantly complicates implementations as the top tree has to be handled differently than the remaining trees. Hence, this idea was discarded.

9. Security Evaluation (including estimated security strength and known attacks)

The security of SPHINCS$^+$ is based on standard properties of the used function families and the assumption that the PRF used within the instantiations of the tweakable hash functions (to generate the bitmasks) can be modeled as a random oracle. We want to emphasize again that this assumption about the random oracle is limited to the pseudorandom generation of bitmasks.

In this section we give a security reduction for SPHINCS$^+$ underpinning the above claim. The security reduction essentially combines the original SPHINCS security reduction from [4], the XMSS-T security reduction from [9], and a new security analysis for multi-instance FORS.

In our technical specification of SPHINCS$^+$ we used the abstraction of tweakable hash functions to allow for different ways of keying a function and generating bitmasks. In the security reduction we will remove this abstraction and assume that each call to the hash function used to instantiate the tweakable hash is keyed with a different value and inputs are XORed with a bitmask before being processed. Moreover, we assume that the bitmasks are generated using a third PRF called $\mathbf{PRF_{BM}}$. The PRF $\mathbf{PRF_{BM}}$ is the single function assumed to behave like a random oracle. Finally, we make a statistical assumption on the hash function F. Informally we require that every element in the image of F has at least two preimages, i.e.,

$$(\forall k \in \{0,1\}^n)(\forall y \in \mathrm{IMG}(\mathrm{F}_k))(\exists x, x' \in \{0,1\}^n) : x \neq x' \wedge \mathrm{F}_k(x) = f_k(x'). \tag{9}$$

Informally, we will prove the following Theorem where F, H, and T are the cryptographic hash functions used to instantiate $\mathbf{F}$ and $\mathbf{H}$, respectively.

Theorem 9.1 *For security parameter $n \in \mathbb{N}$, parameters w, h, d, m, t, k as described above, SPHINCS$^+$ is existentially unforgeable under post-quantum adaptive chosen message attacks if*

- F*,* H*, and* T *are post-quantum distinct-function multi-target second-preimage resistant function families,*
- F *fulfills the requirement of Eqn. 9,*
- $\mathbf{PRF}, \mathbf{PRF_{msg}}$ *are post-quantum pseudorandom function families,*

- $\mathbf{PRF_{BM}}$ *is modeled as a quantum-accessible random oracle, and*
- $\mathbf{H_{msg}}$ *is a post-quantum interleaved target subset resilient hash function family.*

More specifically, the insecurity function $\mathrm{InSec}^{\text{PQ-EU-CMA}}\left(SPHINCS^{+};\xi,2^{h}\right)$ *describing the maximum success probability over all adversaries running in time* $\leq \xi$ *against the* PQ-EU-CMA *security of SPHINCS*$^{+}$ *is bounded by*

$$\begin{aligned}\mathrm{InSec}^{\text{PQ-EU-CMA}}\left(SPHINCS^{+};\xi\right) \leq 2(&\mathrm{InSec}^{\text{PQ-PRF}}\left(\mathbf{PRF};\xi\right)+\mathrm{InSec}^{\text{PQ-PRF}}\left(\mathbf{PRF_{msg}};\xi\right)\\ &+\mathrm{InSec}^{pq\text{-}itsr}\left(\mathbf{H_{msg}};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{F};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{H};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{T};\xi\right))\end{aligned} \tag{10}$$

9.1. Preliminaries

Before we start with the proof, we have to provide two definitions. In general, we refer the reader to [9] for formal definitions of the above properties with two exceptions. First, we use a variant of post-quantum multi-function multi-target second-preimage resistance called post-quantum *distinct*-function multi-target second-preimage resistance. The distinction here is that the targets are given for distinct but predefined functions from the family while for the multi-function notion, the functions are sampled together with the target, uniformly at random.

Second, we define a variant of subset-resilience which captures the use of FORS in SPHINCS^{+} which we call (post-quantum) interleaved target subset resilience. The idea is that from a theoretical point of view, one can think of the 2^{h} FORS instances as a single huge HORS-style signature scheme. The secret key consists of 2^{h} key-sets which in turn consist of k key-subsets of t secret n-byte values, each. The message digest function $\mathbf{H_{msg}}$ maps a message to a key-set (by outputting the index) and a set of indexes such that each index is used to select one secret value per key-subset of the selected key-set.

Formally, the security of this multi-instance FORS boils down to the inability of an adversary

- to learn actual secret values which were not disclosed before,
- to replace secret values by values of its choosing, and
- to find a message which is mapped to a key-set and a set of indexes such that the adversary has already seen the secret values indicated by the indexes for that key-set.

The former two points will be shown to follow from the properties of F, H, and T as well as those of **PRF**. The latter point is exactly what (post-quantum) interleaved target subset resilience captures.

We define those properties in the following.

Post-quantum distinct-function, multi-target second-preimage resistance (PQ-DM-SPR). In the following let $\lambda \in \mathbb{N}$ be the security parameter, $\alpha = \mathrm{poly}(\lambda), \kappa = \mathrm{poly}(\lambda)$, and $\mathcal{H}_{\lambda} = \{\mathrm{H}_K : \{0,1\}^{\alpha} \to \{0,1\}^{\lambda}\}_{K\in\{0,1\}^{\kappa}}$ be a family of functions. We define the success probability of any (quantum) adversary $\mathcal{A}$ against PQ-MM-SPR. This definition is parameterized by the number of targets

$$\mathrm{Succ}^{\text{PQ-DM-SPR}}_{\mathcal{H}_\lambda,p}(\mathcal{A}) = \mathsf{Pr}\left[(\forall\{K_i\}_1^q \subset (\{0,1\}^\kappa)^q), M_i \xleftarrow{\$} \{0,1\}^\alpha, 0 < i \le p; \right.$$
$$(j, M') \xleftarrow{\$} \mathcal{A}((K_1, M_1), \ldots, (K_p, M_p)) :$$
$$\left. M' \neq M_j \wedge \mathrm{H}_{K_j}(M_j) = \mathrm{H}_{K_j}(M')\right] . \tag{11}$$

(Post-quantum) interleaved target subset resilience. In the following let $\lambda \in \mathbb{N}$ be the security parameter, $\alpha = \mathrm{poly}(\lambda), \kappa = \mathrm{poly}(\lambda)$, and $\mathcal{H}_\lambda = \{\mathrm{H}_K : \{0,1\}^\alpha \to \{0,1\}^\lambda\}_{K\in\{0,1\}^\kappa}$ be a family of functions. Further consider the mapping function $\mathrm{MAP}_{h,k,t} : \{0,1\}^\lambda \to \{0,1\}^h \times [0, t-1]^k$ which for parameters h, k, t maps an λ-bit string to a set of k indexes $((I, 1, J_1), \ldots, (I, k, J_k))$ where I is chosen from $[0, 2^h - 1]$ and each J_i is chosen from $[0, t-1]$. Note that the same I is used for all tuples (I, i, J_i).

We define the success probability of any (quantum) adversary $\mathcal{A}$ against PQ-MM-SPR of $\mathcal{H}_\lambda$. Let $\mathrm{G} = \mathrm{MAP}_{h,k,t} \circ \mathcal{H}_\lambda$. This definition uses an oracle $\mathcal{O}(\cdot)$ which upon input of a α-bit message M_i samples a key $K_i \xleftarrow{\$} \{0,1\}^\kappa$ and returns K_i and $\mathrm{G}(K_i, M_i)$. The adversary may query this oracle with messages of its choosing. The adversary would like to find another G input whose output is covered by the G outputs produced by the oracle, without the input being one of the inputs used by the oracle. Note that the adversary knows the description of G and can evaluate it on randomizer-message pairs of its choosing. However, these queries do not count into the set of values which need to cover the adversary's output.

$$\mathrm{Succ}^{\text{pq-itsr}}_{\mathcal{H},q}(\mathcal{A}) = \mathsf{Pr}\Big[(K, M) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(1^\lambda) \quad \text{s.t.} \quad \mathrm{G}(K, M) \subseteq \bigcup_{j=1}^{q} \mathrm{G}(K_j, M_j)$$
$$\wedge\, (K, M) \notin \{(K_j, M_j)\}_1^q\Big]$$

where q denotes the number of oracle queries of $\mathcal{A}$ and the pairs $\{(K_j, M_j)\}_1^q$ represent the responses of oracle $\mathcal{O}$.

Note that this is actually a strengthening of (post-quantum) target subset resilience in the multi-target setting. In the multi-target version of target subset resilience, $\mathcal{A}$ was able to freely choose the common index I for its output. In interleaved target subset resilience, I is determined by G and input M.

9.2. Security Reduction

The security reduction is essentially an application of techniques used especially in [9]. Hence, we will only roughly sketch it here.

We want to bound the success probability of an adversary $\mathcal{A}$ against the PQ-EU-CMA security of SPHINCS$^+$. We start with GAME.0 which is the original PQ-EU-CMA game. Now consider a second game GAME.1 where all outputs of **PRF** are replaced by truly random values. The difference in success probability of any forger $\mathcal{A}$ must be bound by $\mathrm{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$ otherwise we could use $\mathcal{A}$ to break the pseudorandomness of **PRF** with a success probability greater $\mathrm{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$ which would contradict the definition of $\mathrm{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$.

Next, consider a game GAME.2 which is the same as GAME.1 but all outputs of $\mathbf{PRF}_{\mathbf{msg}}$ are replaced by truly random values. Following the same reasoning as above, the difference

in success probability of any adversary $\mathcal{A}$ playing in the two games must be bounded by $\mathrm{InSec}^{\text{PQ-PRF}}(\mathbf{PRF_{msg}}; \xi)$.

Next, we consider GAME.3 where we consider the game lost if $\mathcal{A}$ outputs a valid forgery $(M, \mathbf{SIG})$ where the FORS signature part of **SIG** differs from the signature which would be obtained by signing M with the secret key of the challenger. The difference of any $\mathcal{A}$ in winning the two games must be bounded by $\mathrm{InSec}^{\text{PQ-DM-SPR}}(\mathrm{F}; \xi) + \mathrm{InSec}^{\text{PQ-DM-SPR}}(\mathrm{H}; \xi) + \mathrm{InSec}^{\text{PQ-DM-SPR}}(\mathrm{T}; \xi)$. Otherwise, we could use $\mathcal{A}$ to break the post-quantum distinct-function, multi-target second-preimage resistance of F, H, or T. A detailed proof of this follows exactly along the lines of the security reduction for XMSS-T in [9]. Given distinct challenges for each call to F, H or T for the key-set defined by $\mathbf{PK}.\mathtt{seed}$ and the address space, we program $\mathbf{PRF_{BM}}$ to output bitmasks which are the XOR of the input to the according tweakable hash function and the given challenge. That way we program the actual input to the hash function to be the challenge value. This allows us to extract a second preimage if a collision happens between the forgery and the honestly generated signature. A pigeon hole argument can be used to show that such a collision must exist in this case.

Next, we consider GAME.4 which differs from GAME.3 in that we are considering the game lost if an adversary outputs a valid forgery $(M, \mathbf{SIG})$ where the FORS signature part of **SIG** contains a secret value which is the same as that of an honestly generated signature of M but was not contained in any of the signatures obtained by $\mathcal{A}$ via the singing oracle. The difference of any (unbounded) $\mathcal{A}$ in the two games is bounded by $1/2$ times the success probability of $\mathcal{A}$ in GAME.3. The reason is that the secret values which were not disclosed to $\mathcal{A}$ before still contain 1 bit of entropy, even for an unbounded $\mathcal{A}$.

Finally, we have to bound the success probability of $\mathcal{A}$ in GAME.4. But GAME.4 can be viewed as the (post-quantum) interleaved target subset resilience game. Because, if $\mathcal{A}$ returns a valid signature and succeeds in the GAME, the FORS signature must be valid and consist only of values that have been observed by $\mathcal{A}$ in previous signatures. Hence, the success probability of $\mathcal{A}$ in GAME.4 is bounded by $\mathrm{InSec}^{\text{pq-itsr}}(\mathbf{H_{msg}}; \xi)$ per definition.

Putting things together we obtain the claimed bound. □

9.3. Security Level / Security Against Generic Attacks

As shown in Theorem 9.1, the security of SPHINCS$^+$ relies on the properties of the functions used to instantiate all the cryptographic function families (and the way they are used to instantiate the function families). In the following we assume that there do not exist any structural attacks against the used functions SHA-256, SHAKE256, and Haraka. In later sections we justify this assumption for each of the function familes.

For now, we only consider generic attacks. We now consider generic classical and quantum attacks against distinct-function multi-target second-preimage resistance, pseudorandomness (of function families), and interleaved target subset resilience. Runtime of adversaries is counted in terms of calls to the cryptographic function families.

9.3.1. Distinct-Function Multi-Target Second-Preimage Resistance

To evaluate the complexity of generic attacks against hash function properties the hash functions are commonly modeled as (family of) random functions. Note, that for random functions there is no difference between distinct-function multi-target second-preimage resistance and multi-function multi-target second-preimage resistance. Every key just selects a new ran-

dom function, independent of the key being random or not. In [9] it was shown that the success probability of any classical q_{hash}-query adversary against multi-function multi-target second-preimage resistance of a random function with range $\{0,1\}^{8n}$ (and hence also against distinct-function multi-target second-preimage resistance) is exactly $\frac{q_{\text{hash}}+1}{2^{8n}}$. For q_{hash}-query quantum adversaries the success probability is $\Theta(\frac{(q_{\text{hash}}+1)^2}{2^{8n}})$. Note that these bounds are independent of the number of targets.

9.3.2. Pseudorandomness of Function Families

The best generic attack against the pseudorandomness of a function family is commonly believed to be exhaustive key search. Hence, for a function family with key space $\{0,1\}^{8n}$ the success probability of a classical adversary that evaluates the function family on q_{key} keys is again bounded by $\frac{q_{\text{key}}+1}{2^{8n}}$. For q_{key}-query quantum adversaries the success probability of exhaustive search in an unstructured space with $\{0,1\}^{8n}$ elements is $\Theta(\frac{(q_{\text{key}}+1)^2}{2^{8n}})$ as implicitly shown in [9] (just consider this as preimage search of a random function).

9.3.3. Interleaved Target Subset Resilience

To evaluate the attack complexity of generic attacks against interleaved target subset resilience we again assume that the used hash function family is a family of random functions.

Recall that there are parameters h, k, t where $t = 2^a$. These parameters define the following process of choosing sets: generate independent uniform random integers $I, J_1, \ldots, J_k$, where I is chosen from $[0, 2^h - 1]$ and each J_i is chosen from $[0, t-1]$; then define $S = \{(I, 1, J_1), (I, 2, J_2), \ldots, (I, k, J_k)\}$. (In the context of SPHINCS$^+$, S is a set of positions of FORS private key values revealed in a signature: I selects the FORS instance, and J_i selects the position of the value revealed from the ith set inside this FORS instance.)

The core combinatorial question here is the probability that $S_0 \subset S_1 \cup \cdots \cup S_q$, where each S_i is generated independently by the above process. (In the context of SPHINCS$^+$, this is the probability that a new message digest selects FORS positions that are covered by the positions already revealed in q signatures.) Write S_α as $\{(I_\alpha, 1, J_{\alpha,1}), (I_\alpha, 2, J_{\alpha,2}), \ldots, (I_\alpha, k, J_{\alpha,k})\}$.

For each α, the event $I_\alpha = I_0$ occurs with probability $1/2^h$, and these events are independent. Consequently, for each $\gamma \in \{0, 1, \ldots, q\}$, the number of indices $\alpha \in \{1, 2, \ldots, q\}$ such that $I_\alpha = I_0$ is γ with probability $\binom{q}{\gamma}(1 - 1/2^h)^{q-\gamma}/2^{h\gamma}$.

Define DarkSide_γ as the conditional probability that $(I_0, i, J_{0,i}) \in S_1 \cup \cdots \cup S_q$, given that the above number is γ. In other words, $1 - \text{DarkSide}_\gamma$ is the conditional probability that $(I_0, i, J_{0,i}) \notin \{(I_1, i, J_{1,i}), (I_2, i, J_{2,i}), \ldots, (I_q, i, J_{q,i})\}$. There are exactly γ choices of $\alpha \in \{1, 2, \ldots, q\}$ for which $I_\alpha = I_0$, and each of these has probability $1 - 1/t$ of $J_{\alpha,i}$ missing $J_{0,i}$. These probabilities are independent, so $1 - \text{DarkSide}_\gamma = (1 - 1/t)^\gamma$.

The conditional probability that $S_0 \subset S_1 \cup \cdots \cup S_q$, again given that the above number is γ, is the kth power of the DarkSide_γ quantity defined above. Hence the total probability ϵ that $S_0 \subset S_1 \cup \cdots \cup S_q$ is

$$\sum_\gamma \text{DarkSide}_\gamma^k \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}} = \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}}.$$

For example, if $t = 2^{14}$, $k = 22$, $h = 64$, and $q = 2^{64}$, then $\epsilon \approx 2^{-256.01}$ (with most of the sum coming from γ between 7 and 13). The set S_0 thus has probability $2^{-256.01}$ of being

covered by 2^{64} sets $S_1, \ldots, S_q$. (In the SPHINCS$^+$ context, a message digest chosen by the attacker has probability $2^{-256.01}$ of selecting positions covered by 2^{64} previous signatures.)

Hence, for any classical adversary which makes q_{hash} queries to function family $\mathcal{H}_n$ the success probability is

$$(q_{\text{hash}}+1)\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}.$$

As this for random $\mathcal{H}_n$ is search in unstructured data, the best a quantum adversary can do is Grover search. This leads to a success probability of

$$\mathcal{O}\left((q_{\text{hash}}+1)^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right).$$

For computations, note that the $\mathcal{O}$ is small, and that $(1-1/t)^{\gamma}$ is well approximated by $1-\gamma/t$.

9.3.4. Security Level of a Given Parameter Set

If we take the above success probabilities for generic attacks and plug them into Theorem 9.1 we get a bound on the success probability of SPHINCS$^+$ against generic attacks of classical and quantum adversaries. Let q denote the number of adversarial signature queries. For classical adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$\begin{aligned}
\text{InSec}^{\text{EU-CMA}}\left(\text{SPHINCS}^+; q_{\text{hash}}\right) &\leq 2\Big(\frac{q_{\text{hash}}+1}{2^{8n}}+\frac{q_{\text{hash}}+1}{2^{8n}}\\
&\quad+\text{InSec}^{\text{pq-itsr}}\left(\mathbf{H_{msg}}; q_{\text{hash}}\right)+\frac{q_{\text{hash}}+1}{2^{8n}}+\frac{q_{\text{hash}}+1}{2^{8n}}+\frac{q_{\text{hash}}+1}{2^{8n}}\Big)\\
&= 10\frac{q_{\text{hash}}+1}{2^{8n}}+2(q_{\text{hash}}+1)\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\\
&= \mathcal{O}\left(\frac{q_{\text{hash}}}{2^{8n}}+(q_{\text{hash}})\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right). \quad (12)
\end{aligned}$$

Similarly, for quantum adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$\begin{aligned}
\text{InSec}^{\text{PQ-EU-CMA}}\left(\text{SPHINCS}^+; q_{\text{hash}}\right) &\leq 2\Big(\frac{(q_{\text{hash}}+1)^2}{2^{8n}}+\frac{(q_{\text{hash}}+1)^2}{2^{8n}}\\
&\quad+\text{InSec}^{\text{pq-itsr}}\left(\mathbf{H_{msg}}; q_{\text{hash}}\right)+\frac{(q_{\text{hash}}+1)^2}{2^{8n}}+\frac{(q_{\text{hash}}+1)^2}{2^{8n}}+\frac{(q_{\text{hash}}+1)^2}{2^{8n}}\Big)\\
&= 10\frac{(q_{\text{hash}}+1)^2}{2^{8n}}+\mathcal{O}\left(2(q_{\text{hash}}+1)^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right)\\
&= \mathcal{O}\left(\frac{(q_{\text{hash}})^2}{2^{8n}}+2(q_{\text{hash}})^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right). \quad (13)
\end{aligned}$$

To compute the security level also known as bit security one sets this bound on the success probability to equal 1 and solves for q_{hash}.

9.4. Implementation Security and Side-Channel Protection

Timing attacks. Typical implementations of SPHINCS$^+$ are naturally free of any secret-dependent branches or secretly indexed loads or stores. SPHINCS$^+$ implementations are thus free of the two most notorious sources of timing variation. An exception is potentially SPHINCS$^+$-Haraka, because Haraka is based on AES, which is well known to exhibit timing vulnerabilities in software implementations [3, 15, 5, 14]. Clearly, SPHINCS$^+$-Haraka should only be used in environments that support AES in hardware (like almost all modern 64-bit Intel and AMD and many ARMv8a processors). On *some* processors also certain arithmetic instructions do not run in constant time; examples are division instructions on Intel processors and the `UMULL` multiplication instruction on ARM Cortex-M3 proceesors. Again, typical implementations of SPHINCS$^+$ naturally do not use these instructions with secret data as input – secret data is only processed by symmetric cryptographic primitives that are *designed* to not make use of such potentially dangerous arithmetic.

Differential and fault attacks. We expect that any implementation of SPHINCS$^+$ without dedicated protection against differential power or electromagnetic radiation (EM) attacks or against fault-injection attacks will be vulnerable to such attacks. Deployment scenarios of SPHINCS$^+$ in which an attacker is assumed to have the power to mount such attacks require specially protected implementations. For protection against differential attacks this will typically require masking of the symmetric primitives; for protection against fault-injection attacks countermeasures on the hardware level. One additional line of defense against such advanced implementation attacks is included in the specification of SPHINCS$^+$, namely the option to randomize the signing procedure via the value `OptRand` (see Subsection 8.1.5).

9.5. Security of SPHINCS$^+$-SHAKE256

NIST has standardized several applications of the Keccak permutation, such as the SHA3-256 hash function and the SHAKE256 extendable-output function, after a multi-year Cryptographic Hash Algorithm Competition involving extensive public input. All of these standardized Keccak applications have a healthy security margin against all attacks known.

Discussions of the theory of cryptographic hash functions typically identify a few important properties such as collision resistance, preimage resistance, and second-preimage resistance; and sometimes include a few natural variants of the attack model such as multi-target attacks and quantum attacks. It is important to understand that cryptanalysts engage in a much broader search for any sort of behavior that is feasible to detect and arguably "non-random". NIST's call for SHA-3 submissions highlighted preimage resistance etc. but then stated the following:

> Hash algorithms will be evaluated against attacks or observations that may threaten existing or proposed applications, or demonstrate some fundamental flaw in the design, such as exhibiting nonrandom behavior and failing statistical tests.

It is, for example, non-controversial to use Keccak with a partly secret input as a PRF: any attack against such a PRF would be a tremendous advance in SHA-3 cryptanalysis, even though the security of such a PRF is not implied by properties such as preimage resistance. Similarly, a faster-than-generic attack against the interleaved-target-subset-resilience property, being able to find an input with various patterns of output bits, would be a tremendous advance.

The particular function SHAKE256 used in SPHINCS$^+$-SHAKE256 has an internal "capacity" of 512 bits. There are various attack strategies that search for 512-bit internal collisions, but this is not a problem even at the highest security category that we aim for. There is also progress towards showing the hardness of generic quantum attacks against the sponge construction. Of course, second-preimage resistance is limited by the n-byte output length that we use.

9.6. Security of SPHINCS$^+$-SHA-256

NIST's SHA-2 family has been standardized for many more years than SHA-3. The standardization and popularity of SHA-2 mean that these functions are attractive targets for cryptanalysts, but this has not produced any attacks of concern: each of the members of this family has a comfortable security margin against all known attacks.

The broad cryptanalytic goal of finding non-random behavior (see above) is not a new feature of SHA-3. For example, the security analysis of the popular HMAC-SHA-256 message-authentication code is based on the security analysis of NMAC-SHA-256, which in turn is based on a pseudorandomness assumption for SHA-256.

The particular function SHA-256 used in SPHINCS$^+$-SHA-256 has a "chaining value" of only 256 bits, making it slightly weaker in some metrics than SHAKE256 with 256-bit output. However, it is still suitable for all of our target security categories.

9.7. Security of SPHINCS$^+$-Haraka

Both Haraka-256 and Haraka-512 provide a (second)-preimage resistance of 256-bit in the pre-quantum setting and the best known quantum attack is Grover's search on 256-bit. However, the sponge construction we use for HarakaS has a capacity of 256-bit which allows at most security level 2. The best attack breaking any of the security properties required for SPHINCS$^+$ is a preimage attack which corresponds to a collision search on 256-bit for the sponge construction we use. Instances with larger output size are limited by this and provide a less efficient trade-off between security and efficiency.

Another aspect is that we pseudo-randomly generate round constants derived from a seed. An attacker cannot influence the values of the constants for one instance, but can search for instances having weak constants. As shown by Jean [10], a weak choice of round constants can lead to more efficient preimage attacks. In general, a bad choice of round constants does not break the symmetry of a single round. In the case of Haraka, which combines several calls of two rounds of AES-128 per round to create bigger blocks, the round constants have to break the symmetry within two rounds of AES, but also between the different calls of the two rounds. Let us first focus on Haraka-256.

To break the symmetry within one round of AES, we require that the value of the round constant is not the same for each column. For round constants generated via an extendable-output function from a random 256-bit seed, we consider this event to happen with a probability of 2^{-96}. Moreover, that the symmetry of two rounds of AES is not broken by round-constants happens with 2^{-192}. In other words, since one instance of Haraka-256 uses 10 times 2-round AES, only for a fraction of $10 \cdot 2^{-192}$ instances/keys, we expect that the symmetry within one call of 2 rounds of AES is not broken. Even if this happens, all other 2 round AES calls used in Haraka-256 have with a high probability constants that break the symmetry of 2 rounds of AES for all other calls. Hence, we do not expect any negative consequences for the security.

Haraka-256 processes two 2-round AES-calls in parallel per round. So, we also do not want to have the same round constants in these calls. This condition happens with probability $5 \cdot 2^{-256}$. Furthermore, the probability that two rounds have the same round constants is $10 \cdot 2^{-512}$. Similar observations are also valid for Haraka-512. Hence, we conclude that it is very unlikely, that a pseudo-random generation of the round constants per instance leads to weak round constants.

10. Performance

In order to obtain benchmarks, we evaluate our reference implementation on a machine using the Intel x86-64 instruction set. In particular, we use a single core of a 3.5 GHz Intel Core i7-4770K CPU. We follow the standard practice of disabling TurboBoost and hyper-threading. The system has 32 KiB L1 instruction cache, 32 KiB L1 data cache, 256 KiB L2 cache and 8192 KiB L3 cache. Furthermore, it has 32GiB of RAM, running at 1333 MHz. When performing the benchmarks, the system ran on Linux kernel 4.9.0-4-amd64, Debian 9 (Stretch). We compiled the code using GCC version `6.3.0-18`, with the compiler optimization flag `-O3`.

10.1. Runtime

For the defined parameter sets, the resulting cycle counts are listed in Table 4.

For Haraka, it is especially relevant to also examine platforms that have the AES-NI instruction set available. We used the same system as described above, this time including the `march=native` compiler flag. Performance results are listed in Table 5.

10.2. Space

In Table 6, we list the key and signature sizes (in bytes) for the defined parameter sets. In terms of memory consumption, we remark that the reference implementation tends towards low stack usage. This shows for example in procedures such as computing authentication paths and tree roots, which is done using the treehash algorithm (which requires stack usage linear in the tree height, rather than the naive exponential approach of first computing the entire tree and then cherry-picking the relevant nodes).

11. Advantages and Limitations

The advantages and limitations of SPHINCS$^+$ can be summarized in one sentence: On the one hand, SPHINCS$^+$ is probably the most conservative design of a post-quantum signature scheme, on the other hand, it is rather inefficient in terms of signature size and speed. In the following we discuss disadvantages and advantages in some more detail.

Disadvantage: Signature size and speed. The clear drawback of SPHINCS$^+$ is signing speed and signature size. SPHINCS$^+$ is clearly not competing to be the smallest or fastest signature scheme. However, as shown in Section 7.1.1 there exists a magnitude of possible trade-offs allowing to tweak SPHINCS$^+$ as long as one can tolerate at least one of the two, i.e., somewhat slow signing *or* somewhat large signatures.

Advantage: "Minimal Security Assumptions". In contrast to other post-quantum crypto schemes (including signatures as well as public-key encryption schemes), SPHINCS$^+$ does not

	key generation	signature generation	verification
SPHINCS$^+$-SHAKE256-128s	617 619 732	8 610 599 004	10 222 936
SPHINCS$^+$-SHAKE256-128f	19 348 784	580 904 788	24 826 884
SPHINCS$^+$-SHAKE256-192s	907 587 276	17 586 416 344	15 036 680
SPHINCS$^+$-SHAKE256-192f	28 200 752	757 001 640	40 338 224
SPHINCS$^+$-SHAKE256-256s	1 210 939 356	13 842 403 104	20 889 204
SPHINCS$^+$-SHAKE256-256f	75 031 996	1 664 510 764	41 469 276
SPHINCS$^+$-SHA-256-128s	307 425 484	4 606 958 168	5 514 124
SPHINCS$^+$-SHA-256-128f	9 625 644	302 359 220	12 901 012
SPHINCS$^+$-SHA-256-192s	576 727 832	12 239 247 980	10 740 192
SPHINCS$^+$-SHA-256-192f	17 902 436	487 388 724	26 456 352
SPHINCS$^+$-SHA-256-256s	1 095 050 628	12 893 347 756	19 141 296
SPHINCS$^+$-SHA-256-256f	68 819 608	1 558 148 364	38 316 192
SPHINCS$^+$-Haraka-128s	917 405 356	16 992 635 344	19 360 272
SPHINCS$^+$-Haraka-128f	28 814 020	1 056 761 824	45 964 624
SPHINCS$^+$-Haraka-192s	1 244 530 184	38 062 259 596	27 243 200
SPHINCS$^+$-Haraka-192f	42 782 840	1 276 694 620	69 760 728
SPHINCS$^+$-Haraka-256s	1 817 324 180	28 860 355 888	42 380 420
SPHINCS$^+$-Haraka-256f	113 876 252	3 172 247 452	76 203 004

Table 4: Runtime benchmarks for SPHINCS$^+$

	key generation	signature generation	verification
SPHINCS$^+$-Haraka-128s	49 744 640	894 895 320	1 008 528
SPHINCS$^+$-Haraka-128f	1 571 136	56 112 652	2 416 584
SPHINCS$^+$-Haraka-192s	77 210 192	2215 515 952	1 605 220
SPHINCS$^+$-Haraka-192f	2 438 644	71 288 464	3 815 048
SPHINCS$^+$-Haraka-256s	99 650 528	1461 553 268	2 160 040
SPHINCS$^+$-Haraka-256f	6 255 152	164 592 828	3 975 960

Table 5: Runtime benchmarks for SPHINCS$^+$-Haraka on AES-NI

	public key size	secret key size	signature size
SPHINCS$^+$-128s	32	64	8 080
SPHINCS$^+$-128f	32	64	16 976
SPHINCS$^+$-192s	48	96	17 064
SPHINCS$^+$-192f	48	96	35 664
SPHINCS$^+$-256s	64	128	29 792
SPHINCS$^+$-256f	64	128	49 216

Table 6: Key and signature sizes in bytes

introduce a new intractability assumption. The security of SPHINCS$^+$ is solely based on assumptions about the used hash function. A secure hash function is required by *any efficient signature scheme* that supports arbitrary input lengths.

Moreover, a collision attack against the hash function does not suffice to break the security of SPHINCS$^+$. We consider this an important feature given the successful collision attacks on MD5 and SHA1. Especially given that even for MD5 second-preimage resistance has not been broken, yet.

Finally, the cryptographic community has a good understanding of (exact) hash-function security, especially after the recent SHA3 competition. This is in contrast to the relatively new problems used in other areas of post-quantum cryptography. Even though some of those problems are known already for a long time, estimating the hardness of solving specific problem instances is far less understood.

Advantage: State-of-the-art attacks are easily analyzed. The most efficient attacks known against SPHINCS$^+$ are easy to state and analyze, such as searching for a hash input that has a particular pattern of output bits. The analogous quantum attacks are also easy to state and analyze, such as using Grover's algorithm to accelerate the same search. This allows precise quantification of the security levels provided by SPHINCS$^+$.

Advantage: Small key sizes. Another advantage of SPHINCS$^+$ is the small size of the keys, in particular the public-key size. In many applications public keys are transmitted frequently; almost as frequently as signatures. This is typically the case for certificates (or certificate chains) as used, for example, in TLS.

Advantage: Overlap with XMSS. One more feature of SPHINCS$^+$ is the large overlap with the stateful hash-based signature scheme XMSS. Especially the verification code of XMSS is almost entirely contained within the SPHINCS$^+$ verification code. Hence, in scenarios like virtual private networks where clients authenticate towards a gateway using signatures it is easy to combine these two. While every client that actually can support to handle a state can use XMSS, every other client can use SPHINCS$^+$. Only the gateway has to support verification of both, XMSS and SPHINCS$^+$ signatures. This becomes especially interesting as SPHINCS$^+$ is not particularly well suited for resource-constrained devices (although it was shown that it is in principle possible to implement SPHINCS$^+$ on such devices [8]). However, most resource-constrained devices can deal with a state and XMSS is far better suited for these devices.

Advantage: Reuse of established building blocks. SPHINCS$^+$ uses the basic hash function as building block many times. Any speedup to implementations of SHA-256, SHAKE256 or Haraka directly benefits the SPHINCS$^+$ speed. In particular hardware support for hash functions in the CPU, cryptographic coprocessors, or via instruction-set extensions instantly leads to faster SPHINCS$^+$ signatures (or to smaller SPHINCS$^+$ signatures via tuning w).

References

[1] Jean-Philippe Aumasson and Guillaume Endignoux. Clarifying the subset-resilience problem. Cryptology ePrint Archive, Report 2017/909, 2017. https://eprint.iacr.org/2017/909. 40

[2] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. Cryptology ePrint Archive, Report 2017/933, 2017. https://eprint.iacr.org/2017/933. 40

[3] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. Document ID: cd9faae9bd5308c440df50fc26a517b4, https://cr.yp.to/papers.html#cachetiming. 48

[4] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer Berlin Heidelberg, 2015. 4, 23, 39, 42

[5] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer-Verlag Berlin Heidelberg, 2006. http://www.jbonneau.com/AES_timing_full.pdf. 48

[6] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011. 4

[7] Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013. 12

[8] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016*, volume 9614 of *LNCS*, pages 446–470, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 52

[9] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016*, volume 9614 of *LNCS*, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. 5, 9, 39, 40, 42, 43, 44, 45, 46

[10] Jérémy Jean. Cryptanalysis of Haraka. *IACR Trans. Symmetric Cryptol.*, 2016(1):1–12, 2016. 49

[11] Stefan Kölbl, Martin Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 – efficient short-input hashing for post-quantum applications. volume 2016, pages 1–29, 2017. 33

[12] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979. 4

[13] Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1990. 4

[14] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag Berlin Heidelberg, 2007. 48

[15] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. http://eprint.iacr.org/2005/271/. 48

[16] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995. 38

[17] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Batten and Jennifer Seberry, editors, *Information Security and Privacy 2002*, volume 2384 of *LNCS*, pages 1–47. Springer, 2002. 23

A. Parameter-evaluation Sage script

```
#tsec,hashbytes = 125,16
#tsec,hashbytes = 189,24
tsec,hashbytes = 253,32

F = RealField(100)

def ld(r):
  return F(log(r)/log2)

def pow(p,e):
  return F(p)**e

def qhitprob(qs,r):
  p = F(1/leaves)
  return binomial(qs,r)*(pow(p,r))*(pow(1-p,qs-r))

def la(m,w):
  return ceil(m / log(w,2))

def lb(m,w):
  return floor( log(la(m,w)*(w-1), 2) / log(w,2)) + 1

def lc(m,w):
  return la(m,w) + lb(m,w)


for h in range(60,74,2):
  leaves = 2**h
  for b in range(4,17):
    for k in range(5,40):
      sigma=0
      for r in range(1,300):
        r = F(r)
        p = min(1,F((r/F(2**b)))**k)
        sigma += qhitprob(2^64,long(r))*p
      if(sigma<2**-tsec):
        for d in range(4,h):
          if(h % d == 0 and h <= 64+(h/d)):
            for w in [16,256]:
              wots = lc(8*hashbytes,w)
              sigsize = ((b+1)*k+h+wots*d+1)*hashbytes
              if(sigsize < 50000):
                print h,
                print d,
                print b,
                print k,
                print w,
                print int(ld(sigma)),
                print sigsize,
                # Speed estimate based on (rough) hash count
                print (k*2**(b+1) + d*(2**(h/d)*(wots*w+1)))
```