Practical post-quantum cryptography

Joost Rijneveld

Cover design with Marlies Rijneveld
Printed by GVO drukkers & vormgevers

ISBN: 978-94-6332-568-4

This work has been supported by the European Commission
through the H2020 program under contract ICT-645622 (PQCRYPTO)

PUBLIC DOMAIN No rights reserved

Practical post-quantum cryptography

Proefschrift
ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op

woensdag 20 november 2019
om 14:30 uur precies

door

Leendert Cornelis Joost Rijneveld

geboren op 11 december 1992
te Culemborg

Promotor:

prof. dr. Lejla Batina

Copromotor:

dr. Peter Schwabe

Manuscriptcommissie:

prof. dr. Joan Daemen (voorzitter)

dr. Diego Aranha
Aarhus Universitet, Denemarken

prof. dr.-ing. Tim Güneysu
Ruhr-Universität Bochum, Duitsland

prof. dr. ir. Ingrid Verbauwhede
Katholieke Universiteit Leuven, België

prof. dr. Bo-Yin Yang
Academia Sinica, Taiwan

If you're gonna try and walk on water
make sure you wear your comfortable shoes

Arctic Monkeys, *Piledriver Waltz*

Thanks

Throughout the years I have often claimed that a large part of being able to do a PhD involves being surrounded by a supportive group of people. I stand by that claim, and, looking back, I feel immensely fortunate and grateful for everyone that shaped these past years. There is no way I could have done this without you, and I want to take this opportunity to make that explicit.

First and foremost, I am very thankful to have been Peter's student. I distinctly recall going into a meeting to discuss a research project I felt I had neglected, and walking out with not only a master's thesis subject, but also a PhD position. Over the years countless such discussions followed, always offering new perspectives, guidance, and motivation. I am grateful for your informal, pragmatic attitude, and admire the way you combine academia and engineering. Thanks — for everything.

While there is only one name on the cover of this thesis, it would not have existed without innumerable contributions by just as many coauthors. I thank Andy, Dan, Ebo, Erik, Fang, Jan, Joeri, John, Ko, Matthias, Ming-Shing, Peter, Pol, Ruben, Simona, and Stefan, as well as the numerous contributors to our standardization efforts. I want to thank Andy in particular, who has been like a second supervisor to me. I cannot overstate how comforting it is to be able to rely on your thorough understanding and expertise where proofs, practice, and paper-writing meet.

I thank Matthias, Mijco, Peter, Simona, and Wesley for proofreading various parts of this thesis, and apologize for still sneaking a few typos past your red pens.

I have yet to meet a more welcoming group of colleagues than the Digital Security department, and I fondly remember our coffee-break discussions. I will not attempt to list the many regulars, but want to specifically thank Fabian and Ronny for their never-ending punctuality and perseverance, even in the face of vacations and conferences. For atmopshere, anecdotes, advice, and teaching me to teach, I thank Bart. For Fridays, I thank Dan, Guillaume, Jon, Ko, Matthias, and Sander; may your stack of Brouwers never reach the ceiling, so that you must keep trying. Regardless of how many times we ended up getting moved, I thank Benoît, Joost (for once we do not have to disambiguate), Ko, Niels, and Pedro for 'our' office, and Barış and Rachel for making us part of so many special moments.

One of the undeniable perks of PhD life is the travel, of which I was fortunate enough to enjoy plenty. It would not have been nearly half as fun without the usual suspects from Eindhoven, and for the holidays, discussions, timely skepticism, and that one beer at Lux, I thank Christine, Gustavo, Leon, and Lorenz.

For hosting me at Apple over the summer and giving me a glimpse at what goes on behind closed doors, I thank Steve and Yannick.

Time outside of work weighs just as heavily, and well into my first year I was happy to call K49 home. One year later, it still felt like it was. For the comfortable refuge at the end of a long day, I thank Erwin, Koen, Luuk, and Wesley.

For obscene amounts of snacks and red wine; for spending months on a season; for real conversations; for pubquizzes, dinner, board games, and quite literally crawling through the mud, I thank Carmen, Koen, Luuk, and Nienke.

For a constant tether, wherever we may be, I thank #RU. It is truly a miracle we got any work done at all. I look forward to many more weekends of board games, movies, cake, and hot tubs, and will happily claim a spot in the castle. Really, every PhD student should have a #promovendi. In particular I thank Judith for sage life advice, and Annelies and Bas for opening up their home on countless occasions.

For all the encouragement, beta, and sweaty palms, I thank Annelies, Bart, Bas, Daan, Daniëlle, Gerdriaan, Hannah, Jesse, Judith, Kimberley, Laurens, Margot, Marlon, Pieter-Paul, Pol, Ramon, Rik, Thom, and Wouter. *Aller!*

For support and perspective throughout these final months, even when time and timing was most inconvenient, I thank Leonie.

For all the beers, parties, games, and misery, I thank Erik, Feicko, Feiko, Guido, Jaime, Jim, Kars, Koen, Lennard, Lucien, Olav, Rick, Roel, Roger, Roy, Teun, Timo, and, for everything, Wesley. I count myself lucky to have known many of you for so long, and look forward to many more years together as we embrace civilian life.

But most of all I thank my parents, Janny and Mijco, and my sister, Marlies. For following along every step of the way, for trying to really understand, for bearing with me as I traveled, and for your relentless and unconditional love and support.

Thank you all.

Joost Rijneveld
Cupertino, August 2019

Contents

Thanks 7

Contents 9

1 Introduction 13
  1.1 Post-quantum cryptography . . . 14
  1.2 Organization of this thesis . . . 15
  1.3 Contributions . . . 16
    1.3.1 Software availability and data management . . . 21

2 Preliminaries 23
  2.1 Definitions and notation . . . 23
    2.1.1 The security parameter . . . 24
    2.1.2 Symmetric primitives . . . 25
    2.1.3 Digital signature schemes . . . 28
    2.1.4 Key-encapsulation mechanisms . . . 29
    2.1.5 Algorithms and protocols . . . 32
  2.2 NIST's Post-Quantum Cryptography Standardization project . . . 32
  2.3 Cryptographic engineering . . . 34
    2.3.1 Programming abstractions . . . 34
    2.3.2 Side-channel resistance . . . 37
  2.4 Platforms and architectures . . . 39
    2.4.1 Intel x86, x86-64 and AVX2 . . . 40
    2.4.2 ARMv7 on the Cortex-M series . . . 40
    2.4.3 Java Card . . . 42

3 Hash-based signatures 43
  3.1 One-time signature schemes 44
    3.1.1 Lamport's one-time signature scheme 45
    3.1.2 Winternitz' improvement 48
  3.2 Merkle trees 52
    3.2.1 Many-time digital signature schemes 52
    3.2.2 From one to many 53
    3.2.3 Treehash 55
    3.2.4 Secret seeds 60
    3.2.5 Tree traversal 61
  3.3 XMSS and XMSS^{MT} 62
    3.3.1 Collision resilience 62
    3.3.2 The hypertree 63
    3.3.3 Multi-target attacks 64
    3.3.4 Comparing XMSS and XMSS-T concretely 67
    3.3.5 RFC 8391 70
  3.4 XMSS^{MT} on the Java Card 72
    3.4.1 Java Card platform and limitations 73
    3.4.2 Implementation 75
    3.4.3 Java Card API recommendations and considerations 80
  3.5 SPHINCS 85
    3.5.1 Eliminate the state 85
    3.5.2 HORST 87
    3.5.3 High-performance hash functions 90
  3.6 ARMed SPHINCS 91
    3.6.1 The Cortex-M3 92
    3.6.2 Implementing SPHINCS-256 on the Cortex-M3 93
    3.6.3 Performance 96
    3.6.4 Comparing to XMSS^{MT} 99
  3.7 SPHINCS$^+$ 101
    3.7.1 Tweakable hash functions 102
    3.7.2 FORS 104
    3.7.3 Instances 106
    3.7.4 Performance 111

4 MQ-based signatures 113
4.1 Identification schemes 114
4.1.1 The Fiat-Shamir transform 117
4.2 The MQ problem 119
4.3 The [SSH11] 5-pass identification scheme 120
4.4 Fiat-Shamir for 5-pass identification schemes 123
4.4.1 The [EDV+12] proof 124
4.4.2 A Fiat-Shamir transform for most $(2n + 1)$-pass IDS 128
4.5 MQDSS 130
4.5.1 The 5-pass scheme over $\mathbb{F}_{31}$ 131
4.6 MQDSS-31-64 136
4.6.1 Parameter selection 136
4.6.2 Implementation details 138
4.6.3 Performance 141
4.6.4 The NIST submission 142
4.7 $\mathcal{MQ}$-based signatures in the QROM 145
4.7.1 Unruh's transform 147
4.8 SOFIA 148
4.8.1 Tweaks and optimizations 154
4.9 SOFIA-4-128 156
4.9.1 Parameter selection 156
4.9.2 Implementation details 159
4.9.3 Performance 162

Appendices to Chapter 4 165
4.A The 3-pass scheme over $\mathbb{F}_2$ 165
4.A.1 Parameter selection 168
4.A.2 Implementation details 169
4.A.3 Performance 169

5 Lattice-based KEMs 171
5.1 NTRU-HRSS 172
5.1.1 Parameters 173
5.1.2 CPA-secure NTRU encryption 174
5.1.3 Fujisaki-Okamoto and an IND-CCA2-secure KEM 176
5.1.4 The NIST submission 177

5.2 High-speed key encapsulation 180
  5.2.1 Polynomial multiplication 180
  5.2.2 Inverting polynomials 184
  5.2.3 Performance and comparison 188
5.3 Polynomials in $\mathbb{Z}_{2^m}[x]$ 189
  5.3.1 Kindi, NTRUEncrypt, RLizard, and Saber 192
  5.3.2 ARM Cortex-M4 196
5.4 Multiplication in $\mathbb{Z}_{2^m}[x]$ 199
  5.4.1 Revisiting Karatsuba and Toom-Cook 199
  5.4.2 Small schoolbook multiplications 201
5.5 Measuring multiplication performance 203
  5.5.1 Isolated multiplications 204
  5.5.2 Encapsulation and decapsulation 208
  5.5.3 Profiling optimized implementations 212

Outlook 215

Bibliography 217

Symbols and acronyms 251

Summary 255

Samenvatting 257

About the author 259

Chapter 1

Introduction

Ever since mankind has been communicating, there seems to have been a desire to do so *in secret.* Dating back to the ancient Egyptians and their Greek counterparts across the Mediterranean, the Romans, the Arabs, late-medieval Italian leaders and linguists, a French alchemist, and all the way to the beheading of Mary Queen of Scots, history is riddled with tales and traces of hidden messages and codes [Kah96; Sin99; Sal05]. If information is power, so is its obfuscation. And with the rise of communication comes an ever-increasing interest in its security.

Historically, cryptography —the art of 'hidden writing'— has been a cloak-and-dagger field for spies, conspirators, diplomats and military strategists; its complement, cryptanalysis, the domain of linguists. It was ad-hoc, it was pen-and-paper, and it was tedious. The previous century changed this dramatically. With the construction of automatic encryption machines (most famously the Enigma), cryptography was quickly industrialized. Cryptanalysis followed in lockstep: the first programmable electronic computer was Colossus [Cop06], a massive apparatus designed specifically to counter the war-time cipher machines.

It took several more decades for cryptography to outgrow its air of militarism and politics, but, still deeply intertwined with the surging development in automated computing, it is now so commonplace that it would be hard to imagine a functioning digital society without it. Yet, befitting of its heritage in obscurity and gloom, real-world use of cryptography goes largely unnoticed.

It seems almost impossible to write an introduction to any thesis in cryptography without citing the groundbreaking work by Diffie and Hellman [DH76]. In 1976, their introduction of public-key cryptography opened up a wide range of applications beyond communicating hidden messages. Most notably, in particular in the context of this thesis, Diffie and Hellman describe two new primitives: publicly verifiable digital signatures and key exchange over an insecure channel. Subsequent work by Merkle [Mer90] and Rivest, Shamir, and Adleman [RSA78] provides further practical constructions. The former will be discussed extensively in Chapter 3; the latter makes a brief reappearance below.

Digital signatures provide a way to ensure authenticity and integrity of messages. Given a message (e.g., a letter, statement, or contract), a signer uses their *secret key* to produce a signature to pair it with. Anyone in possession of the signer's corresponding *public key* is then able to verify the origin and correctness of the message. This complements secrecy of messages, which is achieved by first establishing a shared key through a key exchange, and then encrypting messages using this established key. Together, these fundamental primitives form the basis of essentially all of the secure communication on the internet of today.[1]

Cryptography predating the work by Diffie and Hellman is sometimes referred to as *classic* or *traditional*; everything after *"New Directions"* is typically *modern*. In this thesis, the term 'classical' will refer to the cryptography of today. With the computer on the brink of a potential revolution of quantum computing, cryptography, too, is gearing up for another large overhaul.

1.1 Post-quantum cryptography

For years, physicists have been predicting the imminent construction of a large-scale universal quantum computer: a computer that makes use of properties from quantum mechanics to perform computations far beyond the reach of classical computers. Although sometimes portrayed as a catch-all solution to solve the world's most pressing problems, so far, quantum algorithms are highly specialized. Unfortunately, one such specialized algorithm [Sho97] solves exactly the problems that classical cryptography requires to be hard to solve. In particular, Shor's algorithm can be used to efficiently split large composite numbers into their prime factors, as well as solve the related discrete-logarithm problem. These problems underly the Diffie-Hellman key exchange, RSA, and all of the elliptic-curve cryptography currently in use. Together, these schemes encompass essentially all of the public-key cryptography deployed right now.

A quantum computer that is able to run Shor's algorithm for cryptographically relevant inputs is yet to be built, and it is unclear when (and even if) this will happen. Still, the cryptographic community watches in tense anticipation; developing and deploying new cryptographic schemes and protocols is no easy task either.

---

[1] This thesis almost exclusively focuses on public-key cryptography. Symmetric-key cryptography, while similarly crucial and fundamental, is largely out of scope.

Cryptography that remains secure in the presence of an adversary with access to a quantum computer is called *post-quantum cryptography*.[2] The crucial difference between post-quantum cryptography and traditional cryptography lies in the problems on which it is based[3] — problems for which no efficient quantum algorithm is known. The problems proposed so far can roughly be categorized in five classes, each with their own characteristic strengths and weaknesses. These relate to hash functions, multivariate quadratic equations, lattices, error-correcting codes and supersingular isogenies. For now, we merely make a passing mention of these categories; the former three form the basis of the remainder of this thesis.

Over the last decade, research in the field of post-quantum cryptography has steadily progressed. While it used to be experimental, large and slow (indeed, the logo of the EU PQCRYPTO project is an aptly chosen tortoise), post-quantum cryptography is rapidly becoming more practical. This is both a consequence of and a stimulus for standardization efforts. Most notably, in 2016, the American National Institute for Standards and Technology (NIST) started a multi-year project towards standardizing post-quantum cryptography. With hundreds of participants from both academia and industry and a clock eerily ticking away towards uncertainty, practical post-quantum cryptography is now more relevant than ever.

1.2 Organization of this thesis

Following this introduction, Chapter 2 provides some general context for the material presented in this thesis. We begin by establishing a number of common definitions and notation, and follow with general discussion on cryptographic engineering, software implementations, the relevant development platforms, and NIST's Post-Quantum Cryptography Standardization project.

The main matter of this thesis consists of three technical chapters. Chapters 3, 4, and 5 each address one of the hash-based, $\mathcal{MQ}$-based and lattice-based classes — the first two consider digital signatures, and the latter discusses key exchange. Each of these are self-contained and can be read separately.

---

[2] Also known as *quantum-safe cryptography*, post-quantum cryptography does not require a quantum computer. It should run efficiently on a classical computer, but an adversary should be unable to attack the scheme using both classical and quantum computation. This is quite different from *quantum cryptography*, where also the honest participants are required to have specific, expensive equipment.

[3] In modern cryptography, breaking a cryptosystem is typically shown to be equivalent to solving a hard mathematical problem. This relation forms the basis of 'security reductions', proving the security of a scheme in terms of the difficulty of solving specific instances of the related problem.

1.3 Contributions

All the work presented in this thesis is the result of collaboration with a wide range of coauthors. While it is not always easy to credit contributions to individual authors, the following subsections provide an overview of the work each chapter was based on, explicitly highlighting my own contributions. As is common for publications in subfields of mathematics, authors are ordered alphabetically.

Chapter 3: Hash-based signatures

In Chapter 3, we examine hash-based digital signature schemes. The chapter contains a thorough discussion of historical constructions, leading up to the recent XMSS, SPHINCS, and SPHINCS$^+$ schemes. We discuss scheme design, and describe several implementations, in particular on embedded platforms. This chapter finds its basis in four academic papers and two technical documents.

> Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM." in: *Public Key Cryptography – PKC 2016.* Vol. 9614. LNCS. Springer, 2016.

In this work, we implement the SPHINCS-256 scheme as proposed in [BHH+15] on the Cortex-M3 Discovery board. The idea of applying the Treehash algorithm was already proposed in [BHH+15]. Part of this work was done as part of my master's thesis *"Implementing SPHINCS with restricted memory"*. Andreas Hülsing and Peter Schwabe acted as supervisors, providing context, explanation and suggestions. I evaluated and implemented the required changes based on the implementation presented in [BHH+15],

> Andreas Hülsing, Joost Rijneveld, and Fang Song. "Mitigating Multi-Target Attacks in Hash-based Signatures." In: *Public Key Cryptography – PKC 2016.* Vol. 9614. LNCS. Springer, 2016.

This work introduces XMSS-T and describes how to mitigate multi-target attacks, precisely analyzing the attack complexity in a multi-target setting. I was only tangentially involved in this work by modifying my implementation of XMSS to fit the new definitions and providing a performance comparison.

> Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. "Is Java Card ready for hash-based signatures?" In: *Advances in Information and Computer Security – IWSEC 2018.* Vol. 11049. LNCS. Springer, 2018.

This work was the result of a larger collaboration with the NLNCSA, where we implement XMSS^{MT} on the Java Card smart-card platform. Discussions with Ebo van der Laan and Jan Verschuren were crucial towards parameter selection for the relevant use-case, and Joeri de Ruiter provided pointers with regards to Java Card programming. I wrote the implementation, performed measurements and wrote the paper, with Erik Poll and Peter Schwabe acting as supervisors providing useful discussion on parallelism and performance expectations.

> Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme.* Request for Comments 8391. IETF, 2018.

This informational RFC describes XMSS and XMSS^{MT}. I got involved in a later stage by significantly rewriting and contributing to the reference implementation, effectively taking co-ownership of the code.

> 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, and Peter Schwabe. *SPHINCS^{+}*. Submission to the NIST Post-Quantum Cryptography Standardization project. 2017.

At the time of writing, SPHINCS^{+} is a second-round candidate in NIST's Post-Quantum Cryptography Standardization project. While the list of authors is long, it is only natural that a much smaller group participated prominently in the discussions regarding the details of the design, parameter selection, and writing the specification — I contributed significantly to this. Most importantly, I wrote and maintain the reference implementation and most of the optimized implementations, with notable contributions by Stefan Kölbl.

> Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS^{+} signature framework." In: *Conference on Computer and Communications Security – CCS '19. To appear.* ACM, 2019.

This paper accompanies the SPHINCS^{+} submission to NIST's Post-Quantum Cryptography Standardization project, defining it as a generic framework and providing a more thorough security analysis by presenting a unified approach that applies more generically to hash-based signature schemes. As with the submission to NIST, I contributed to the design, writing, and software.

Chapter 4: $\mathcal{MQ}$-based signatures

This chapter takes a non-standard approach towards designing signature schemes based on the $\mathcal{MQ}$ problem. Its main contributions are the introduction of the MQDSS and SOFIA schemes; it is primarily based on two academic papers, and also makes brief note of a subsequent technical document.

> Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "From 5-Pass $\mathcal{MQ}$-Based Identification to $\mathcal{MQ}$-Based Signatures." In: *Advances in Cryptology – ASIACRYPT 2016*. Vol. 10032. LNCS. Springer, 2016.

This work provides security proofs for a more generic Fiat-Shamir transform, and subsequently instantiates it as MQDSS. I contributed to finding faults in earlier work, but more notably to design discussions regarding MQDSS and the MQDSS-31-64 instance. Contributing to parameter selection and performance measurements, I implemented both the reference implementation and the optimized implementation of MQDSS-31-64; the latter with contributions by Ming-Shing Chen. Writing the paper was joint work by all authors.

> Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. *MQDSS*. Submission to NIST's Post-Quantum Cryptography Standardization project. 2017.

At the time of writing, MQDSS is a second-round candidate in NIST's Post-Quantum Cryptography Standardization project. I was only marginally involved in preparing the submission, primarily contributing by adjusting the implementation and making it amenable to the additional parameter set.

> Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "SOFIA: MQ-based signatures in the QROM." in: *Public Key Cryptography – PKC 2018*. Vol. 10770. LNCS. Springer, 2018.

Following up on the MQDSS work, this paper proposes a similar scheme using a different transform: SOFIA. The scheme design follows from Unruh's transform [Unr15], but we present several significant optimizations for this specific instantiation. I contributed to these design discussions, as well as the parameter-space exploration. I subsequently implemented the scheme and collaborated with Ming-Shing Chen to implement several optimized instances for the various parameter choices. As before, writing the paper was joint work by all authors.

Chapter 5: Lattice-based KEMs

The final technical chapter of this work concerns lattice-based key-encapsulation mechanisms. We focus on NTRU in particular, introducing the NTRU-HRSS scheme, but examine optimized implementations of related structures as well. Just like Chapter 4, this chapter is based on two academic papers and briefly discusses subsequent standardization efforts.

> Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. “High-speed key encapsulation from NTRU.” in: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Vol. 10529. LNCS. Springer, 2017.

In this work, we revisit the NTRU scheme, and choose parameters that result in a key-encapsulation mechanism that is competitive with more recent lattice-based schemes. We demonstrate feasibility by providing record-setting software; I primarily contributed by implementing the optimized polynomial arithmetic and writing the related parts of the paper.

> Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. *NTRU-KEM-HRSS: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2017.

We submitted NTRU-HRSS to NIST’s Post-Quantum Cryptography Standardization project. I made minor contributions by slightly adjusting the optimized NTRU-HRSS implementation; John Schanck took ownership of the reference software. In round 2, the NTRU and NTRUEncrypt submissions merged to form the NTRU submission. The result of this is listed separately.

> Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. *NTRU: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2019.

At the time of writing, NTRU is a second-round candidate in NIST’s Post-Quantum Cryptography Standardization project. I did not have significant contributions in the second-round submission other than performing updated benchmarks; Oussama Danba extended the optimized implementation to encompass the new parameter sets introduced in round 2.

> Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. "Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates." In: *Applied Cryptography and Network Security – ACNS 2019.* Vol. 11464. LNCS. Springer, 2019.

This work describes optimizing polynomial multiplication on the Cortex-M4. We optimize multiplication in $\mathbb{Z}_{2^m}[x]$, targeting several submission to NIST's Post-Quantum Cryptography Standardization project — one of which is the aforementioned NTRU-HRSS. Together, Matthias Kannwischer and I wrote the software and performed the measurements, frequently pair-programming and sharing a whiteboard. Writing the paper was joint work by all authors.

Off-by-many

If there is one practical commonality throughout the work described in this thesis, it is indexing. Whether nodes in a tree, a vector of field elements, or terms of a polynomial, every item gets a label. For readability, many indices in this work run from 1 to n. This, of course, is in stark contradiction to the well-established practice of starting from 0 and counting to $n - 1$, which often allows for easier computation. This mixing of conventions has doubtlessly led to errors, all of which I claim as my own. These and other errata will be collected and accounted for in the digital version of this work at `https://joostrijneveld.nl/thesis` as they become apparent.

Changing perspective

An issue that often comes up in discussions related to thesis-writing is the form of narrative. In particular, when a thesis is composed of a collection of already-published collaborative work with several coauthors, it is not immediately obvious what perspective to use: while the presented work is jointly authored, a single name is listed on the cover. This is sometimes resolved by avoiding first-person pronouns altogether, and using passive voice throughout.

Up to this point, I have been writing in first-person singular. For the remainder of this thesis, we will use the plural form. Here, *we* may refer to *us*, the authors of the discussed work, but also to *you*, the reader, and *I*, the author of this thesis. Occasionally, it will refer to all of us: *you*, the reader, and *us*, the authors. We (i.e., *you*, the reader, and *I*, the author) will have to rely on the assumption that this will become clear from context.

1.3.1 Software availability and data management

All work described in this thesis is accompanied by software. The source code is available online at `https://joostrijneveld.nl/thesis`, which will also include versioned updates as appropriate. This also includes scripts to reproduce benchmarks where relevant, as well as software that is only tangentially related to this thesis. Unless explicitly stated otherwise, all software has been placed in the public domain to the extent possible under law, and all copyright and related rights have been waived by applying the CC0 1.0 Public Domain Dedication waiver. Software accompanying a publication is implicitly coauthored by all authors.

In particular, this includes:

Getting started on STM32. Examples that demonstrate how to get started with programming STM32 Discovery boards, as well as wrappers around basic firmware functionality. See Section 2.4.2.

Merkle tree traversal algorithms. Python reimplementations of the Merkle tree traversal algorithms described in [BDS09]. See Section 3.2.5.

The XMSS reference code. The implementation accompanying the informational RFC 8391, specifying WOTS$^+$, XMSS, and XMSS^{MT}. It includes an implementation of the BDS tree traversal algorithm. This implementation was coauthored with Andreas Hülsing. See Sections 3.2.5 and 3.3.5.

The XMSS-T code. The implementation of XMSS-T, leading to the XMSS reference code. It contains tweaks that allow for comparison to SPHINCS-256 and to XMSS at lowered security levels, and accompanied the paper *“Mitigating Multi-Target Attacks in Hash-based Signatures”* [HRS16b]. See Section 3.3.3.

XMSS^{MT} on the Java Card. An implementation of the XMSS^{MT} scheme for the Java Card platform. It accompanied the paper *“Is Java Card ready for hash-based signatures?”* [LPR+18]. See Section 3.4.

The ARMed SPHINCS code. A modified version of the SPHINCS reference implementation and the XMSS^{MT} reference implementation, targeting the Cortex-M3. It accompanied the paper *“ARMed SPHINCS – Computing a 41 KB signature in 16 KB of RAM”* [HRS16a]. See Section 3.6.

The ChaCha permutation for the Cortex-M. An ARMv7E-M assembly implementation of the π_{ChaCha} permutation function. This implementation was initially used in ARMed SPHINCS [HRS16a]. See Section 3.6.

SPHINCS-256-py. A reimplementation of SPHINCS-256 in Python, aimed to provide a highly flexible framework for experimenting and comparison.

The SPHINCS$^+$ reference code. The reference implementation of SPHINCS$^+$, accompanying the SPHINCS$^+$ submission to NIST's Post-Quantum Cryptography Standardization project [BDE+17]. See Section 3.7.

PySPX. Python bindings for the SPHINCS$^+$ reference code, originally written for integration into The Update Framework. Also available on PyPI as `pyspx`.

The MQDSS code. The reference and AVX2-optimized implementation of MQDSS, accompanying the paper *"From 5-pass MQ-based identification to MQ-based signatures"* [CHR+16], and the software as part of the MQDSS submission to NIST's Post-Quantum Cryptography Standardization project. See Section 4.5.

The SOFIA code. The reference and optimized code accompanying the paper *"SOFIA: MQ-based signatures in the QROM"* [CHR+18]. See Section 4.8.

The NTRU-HRSS code. The AVX2-optimized implementation of NTRU-HRSS, accompanying the paper *"High-speed key encapsulation from NTRU"* [HRS+17a], and the updated software as part of the NTRU-HRSS submission to NIST's Post-Quantum Cryptography Standardization project. See Sections 5.1 and 5.2.

Bit permutations. Simulator for a subset of x86-64 with AVX2 extensions, used to construct efficient permutations on bit sequences for NTRU-HRSS [HRS+17a].

PQM4. A library and benchmarking and testing framework for post-quantum cryptography on the Cortex-M4, coauthored with Matthias Kannwischer, Peter Schwabe, and Ko Stoffelen. See also [KRS+19].

The $\mathbb{Z}_{2^m}[x]$ code. Code generation scripts for polynomial multiplication, accompanying the paper *"Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates"* [KRS19]. See Section 5.4.

PQClean. A testing framework and collection effort of clean implementations of post-quantum cryptography, coauthored with Matthias Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers.

Chapter 2

Preliminaries

Before we go into the technical content of this thesis, we first establish some context. This includes the necessary notation and security definitions, but also leaves room to discuss more general aspects that affect all of the discussed work, mostly related to cryptographic engineering. In particular, we briefly address various platforms and instruction sets, as well as more broadly applicable subjects such as code generation and timing-attack-resistant programming.

2.1 Definitions and notation

While it is often more comfortable to follow (or, indeed, explain) schemes and procedures in prose, there is no escaping the introduction of some ambiguity in this way. To ensure consistency, the prose is accompanied by formal descriptions that capture the same concepts. Throughout this thesis, we will require various recurring definitions and notation fragments. They are summed up in this section, accompanied, where relevant, by the appropriate security notions. For more specific, one-off, or varying definitions of symbols in different contexts (such as n, which is used to denote bit lengths, degrees of polynomials, numbers of variables, etc.), refer to the overview of symbols and acronyms on page 251.

We write $x = 42$ to mean equality, but also for definitions, aliases and unpacking; we use $x \leftarrow 42$ to denote variable assignment, typically as the result of computation. Similarly, we use $x \overset{\$}{\leftarrow} \mathcal{X}$ to draw uniformly random from a collection. When we write log without explicit base, we assume $\log_2$. To represent an arbitrary value that is polynomially related to n, we write $poly(n)$. The notation $\|collection\|$ is used for the size of a collection, but also for the number of possible subscripted elements (e.g., consider $\|node\|$ in relation to $node_i$). As an infix operator, $\|$ signifies string concatenation. When we write **for** $i \in collection$ **do**, we assume ordered enumeration over the collection, although the order typically does not matter.

2.1.1 The security parameter

A recurring issue in the algorithmic definition of cryptographic schemes is the relation to the desired security level. This is often captured by the notion of a ‘security parameter’, relating the targeted security of a scheme to its runtime and, in particularly, to that of the algorithm describing the adversary.

The notion of a security parameter is strongly connected to the seemingly colloquial classification of problems as *hard* and *easy*. These terms are typically used to convey a deeper mathematical idea: in this work, a problem is considered *hard* when there is no known algorithm with a runtime polynomial in the security level, and, conversely, *easy*, when such an efficient algorithm exists. Unsurprisingly, such algorithms are called *polynomial-time algorithms*. When their outcome depends on random chance, they are called *probabilistic* (sometimes *randomized*, in literature).

The nature of this work allows us, in terms of definitions, to lean towards the informal. As a consequence, algorithms and definitions in the following subsections are described such that they provide a basic understanding and a correct intuition, but may not be best suited as the basis for rigorous security proofs.

Closely related to the hardness of the underlying problems and the notion of polynomial runtime, we use the term ‘negligible’ and the phrase *negligible in the security parameter* to signify that the probability of some event diminishes exponentially as the security parameter is increased. Formally, we say that $p(x)\colon \mathbb{N} \to [0,1]$ is negligible if for any $x > 0$, there exists an n_x such that for all $n > n_x$, $p(x) < \frac{1}{x^n}$.

Not all cryptographic schemes allow the user to easily or granularly vary the achieved security level. Furthermore, the exact security level that follows from particular choices in the scheme design is not equally well-understood across families of cryptographic schemes, and it would be pretense to suggest that all concrete algorithms can be directly and unambiguously parameterized as such.

Still, in some contexts it is meaningful to be able to exactly quantify the security level (see, e.g., Section 3.3.3). When this is the case, we use k to denote the security parameter.[1] In all other cases, it is left implicit as a design consideration.

We differentiate between *classical* and *quantum* security levels. This reflects the natural distinction between security against classical and quantum adversaries. In particular, a quadratic difference typically follows from Grover’s algorithm [Gro96].

---

[1] The security parameter is often denoted 1^k rather than n or k. In complexity theory, ‘running in polynomial time’ signifies that the runtime is asymptotically polynomial in the *length* of the input. Parameterizing key generation with the unary string 1^k achieves exactly this, albeit somewhat artificially.

2.1.2 Symmetric primitives

This thesis focuses exclusively on asymmetric constructions. Still, such schemes typically require more fundamental primitives underlying their design. In this section, we define several such notions. We use various concrete instantiations throughout this thesis, which we will introduce where relevant. To avoid trivial cases, we implicitly assume efficient evaluation for all primitives defined below.

One of the main building blocks of modern cryptography is the one-way function. Intuitively, this is a function that is easy to compute, but, given an image, hard to invert. Note that this is only holds *in general*, and there may be a negligible number of images that are easy to invert. We formalize this as follows.

Definition 2.1.1 (One-way function) *A function* $f\colon\{0,1\}^* \to \{0,1\}^*$ *is called a one-way function if the following conditions hold:*

- *Given* $x \in \{0,1\}^*$, *there exists a polynomial-time algorithm to compute* $f(x)$.
- *For any probabilistic polynomial-time algorithm* $\mathcal{A}$, *given* $f(x)$ *for a random choice of* $x \in \{0,1\}^{poly(k)}$, *running* $\mathcal{A}$ *on input* $f(x)$, *the probability of finding any* $x' \in \{0,1\}^*$ *such that* $f(x) = f(x')$ *is negligible in* k.

Building upon the more general one-way function, we define the notion of a cryptographic hash function. Notably, hash functions map an infinite domain onto a finite range. We remark that not all uses of a hash function require it to satisfy all properties listed below. For example, for schemes that are resilient against collisions, a hash function that only satisfies (second-)preimage resistance suffices.

Definition 2.1.2 (Cryptographic hash function) *Let* $\mathcal{H}\colon\{0,1\}^* \to \{0,1\}^k$ *be a cryptographic hash function.*

- *We call* $\mathcal{H}$ *preimage resistant if, for any probabilistic polynomial-time algorithm* $\mathcal{A}$, *given* $f(x)$ *for a random choice of* $x \in \{0,1\}^{poly(k)}$, *running* $\mathcal{A}$ *on input* $f(x)$, *the probability of finding any* $x' \in \{0,1\}^*$ *such that* $f(x) = f(x')$ *is negligible in* k.
- *We call* $\mathcal{H}$ *second-preimage resistant if, for any probabilistic polynomial-time algorithm* $\mathcal{A}$, *given a randomly chosen* $x \in \{0,1\}^{poly(k)}$, *running* $\mathcal{A}$ *on input* x, *the probability of finding any* $x' \in \{0,1\}^*$ *such that* $x \neq x'$ *but* $f(x) = f(x')$ *is negligible in* k.

- *We call $\mathcal{H}$ collision resistant if, for any probabilistic polynomial-time algorithm $\mathcal{A}$, running $\mathcal{A}$, the probability of finding any $x, x' \in \{0,1\}^*$ such that $x \neq x'$ and $f(x) = f(x')$ is negligible in k.*

At the basis of the construction of many symmetric-key primitives lies the notion of the pseudorandom permutation. Pseudorandom permutations (PRPs) are typically defined as a function family: a keyed function that forms a collection of permutations. For this work we are only tangentially interested in PRPs, and limit ourselves to defining individual length-preserving permutations. For completeness, we explicitly define permutation functions, and stress that this is different from a *permutation on bits* as used in Section 5.2: the latter rearranges bits within a bit string, preserving its Hamming weight.

Definition 2.1.3 (Permutation function) *We call f a (length-preserving) permutation on S if $f\colon S^n \to S^n$ is a bijective function.*

In all schemes presented in this work, we make extensive use of randomly sampled values. Ensuring access to sufficient and reliable randomness is a hard problem on its own, and outside the scope of this work. To lessen the need for random values and to optimize storage requirements, we rely on pseudorandom generators (PRGs). Intuitively, a PRG takes a secret random seed, and expands this to a sequence of bits that are indistinguishable from random noise.

Definition 2.1.4 (Pseudorandom generator (PRG)) *For ℓ polynomial in k, we call a function $g\colon \{0,1\}^k \to \{0,1\}^\ell$ a pseudorandom generator if, for any probabilistic polynomial-time algorithm $\mathcal{A}$, given $U_0 = g(x)$ for a random choice of $x \in \{0,1\}^k$, a random $U_1 \in \{0,1\}^\ell$, and a random $b \in \{0,1\}$, the success probability of running $\mathcal{A}(U_b)$ to distinguish between $U_b = U_0$ and $U_b = U_1$ differs negligibly from guessing.*

Note that the adversarial model of indistinguishability of a PRG does not include making g or the input x available: the intended test is statistical on the output.

With the standardization of SHA-3 [NIST15b], a new primitive was popularized: the extendable output function (XOF). XOFs strongly relate the use of hash functions and PRGs, taking an input of arbitrary length to produce an output of arbitrary length while still satisfying the properties of a hash function. Defining security in terms of indistinguishability is not appropriate for XOFs: the security properties of hash functions include providing or even allowing free choice of the input, allowing a distinguisher to simply recompute the output of the XOF. This is addressed extensively in the literature on provable security of symmetric-key cryptography [MRH04; CDM+05; AMP10], but out of scope for the definitions presented here. For the purpose of this work, we simply consider a XOF to be a hash function with arbitrary-length output, skipping over more intricate properties relating to predictability of subsequent output. More formally, we define:

Definition 2.1.5 (Extendable output function (XOF)) $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^*$ *is an extendable output function if* $\mathcal{H}'\colon \{0,1\}^* \to \{0,1\}^\ell$*, obtained by truncating the output of* $\mathcal{H}$*, is a preimage resistant, second-preimage resistant and collision resistant hash function for any output length* ℓ*.*

It may be somewhat counterintuitive to consider deterministic functions when trying to construct (pseudo)random streams of data. The above definitions of a pseudorandom generator and an extendable output function work well when trying to expand a single random input to a stream of random data. When confronted with multiple, structured inputs, however, these primitives are not a good fit.

Instead, rather than using a single function to derive randomness, we can consider a *family* of pseudorandom functions. Rather than making an adversary distinguish between the output of a single function and uniform random data, we embrace the deterministic and pseudorandom nature and task the adversary to distinguish between a specific function and a randomly drawn function from the family. This specific function follows from choosing (and typically fixing) a key κ and using this to select a function from the family. We define this more formally, below. For simplicity, we assume the PRF is length-preserving in terms of the seed.

Definition 2.1.6 (Pseudorandom function (PRF)) *Consider a family of length-preserving functions* $F\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$*, that is, let* $F_\kappa\colon \{0,1\}^n \to \{0,1\}^n$ *be a length-preserving function for a key* $\kappa \in \{0,1\}^k$*. We call* F *a pseudorandom function if, for any probabilistic polynomial-time algorithm* $\mathcal{A}$ *and a randomly chosen key* $\kappa \in \{0,1\}^k$*, the success probability of running* $\mathcal{A}(F_\kappa)$ *to distinguish between* F_κ *and a random function* $f\colon \{0,1\}^n \to \{0,1\}^n$ *differs negligibly from random guessing.*

As opposed to the PRG setting, the adversary is allowed oracle access to the function it must classify. PRFs do inherit typical randomness properties, i.e., a slight difference in the input should make the output uncorrelated.

In the remainder of this thesis, we only casually touch upon symmetric primitives and their properties, simply assuming their existence and security. In particular, we select instantiations without justification according to the above definitions.

2.1.3 Digital signature schemes

The introduction in the previous chapter already briefly touched upon digital signatures. We now give a more formal definition.

Definition 2.1.7 (Digital signature scheme) *A digital signature scheme is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *defined as follows:*

- *The key-generation algorithm* KeyGen *is a probabilistic algorithm that outputs a public key* pk *and a secret key* sk*, i.e., a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- *The signing algorithm* Sign *is a possibly probabilistic algorithm that takes as input a message* m *and a secret key* sk *to produce a signature* σ*.*
- *The verification algorithm* Verify *is a deterministic algorithm that takes as input a message* m*, a signature* σ *and a public key* pk*. It outputs the Boolean value* True *to indicate that the signature is accepted, or* False *to indicate rejection.*

For correctness, we require that for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$, all messages m, and all signatures $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk})$, it holds that $\mathsf{Verify}(m, \sigma, \mathsf{pk}) = \mathsf{True}$. Informally, this expresses that all properly generated signatures are indeed accepted.

The standard security notion for signature schemes is existential unforgeability under adaptive chosen message attacks (EU-CMA) [GMR88]. Intuitively, this notion captures the idea that an adversary should not be able to create valid signatures for any message m (not necessarily of their specific choosing, as long as it has not been signed before), even after obtaining signatures on numerous other messages of their choice. We define this more formally in Definition 2.1.8.

Definition 2.1.8 (EU-CMA) *For a digital signature scheme* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$*, consider the following game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$*:*

1. $\mathcal{C}$ *runs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ *and sends* pk *to* $\mathcal{A}$*.*
2. $\mathcal{A}$ *sends a freely chosen message* m_i *to* $\mathcal{C}$*.*
3. $\mathcal{C}$ *responds with* $\sigma_i \leftarrow \mathsf{Sign}(m_i, \mathsf{sk})$*.*
4. $\mathcal{A}$ *repeats 2. and 3. for* q *iterations, with* q *polynomial in the security parameter.*
5. $\mathcal{A}$ *outputs a pair* (m', σ')*. This is a valid forgery if* $\mathsf{Verify}(m', \sigma', \mathsf{pk}) = \mathsf{True}$ *and* $m' \neq m_i$ *for all* $i \in \{1, \ldots, q\}$*.*

The signature scheme is said to be EU-CMA when any adversary $\mathcal{A}$*, running in time polynomial in the security parameter, has negligible success probability.*

2.1.4 Key-encapsulation mechanisms

In most classes of post-quantum cryptography, constructing a key exchange is not as elegant and straight-forward as the classical protocol by Diffie and Hellman. Instead of two parties performing analogous steps, one party generates a public key, which the other uses to encrypt and share a secret. This relates closely to public-key encryption; indeed, one follows from the other. We define both primitives below.

Definition 2.1.9 (Public-key encryption scheme (PKE)) *A public-key encryption scheme is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *defined as follows:*

- *The key-generation algorithm* KeyGen *is a probabilistic algorithm that outputs a public key* pk *and a secret key* sk*, i.e., a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- *The encryption algorithm* Enc *is a probabilistic algorithm that takes as input a message* m *and a public key* pk *to produce a ciphertext* c*.*
- *The decryption algorithm* Dec *is a deterministic algorithm that takes as input a ciphertext* c *and a secret key* sk *to produce a message* m*, or* False *for failure.*

Definition 2.1.10 (Key-encapsulation mechanism (KEM)) *A key-encapsulation mechanism is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$ *defined as follows:*[2]

- *The key-generation algorithm* KeyGen *is a probabilistic algorithm that outputs a public key* pk *and a secret key* sk*, i.e., a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- *The encapsulation algorithm* Encaps *is a probabilistic algorithm that takes as input a public key* pk *to produce a shared secret* ss *and a ciphertext* c*.*
- *The decapsulation algorithm* Decaps *is a deterministic algorithm acting on a ciphertext* c *and a secret key* sk *to output a shared secret* ss'*, or* False *for failure.*

Key-encapsulation mechanisms are sometimes designed in such a way[3] that the recipient of an encapsulated secret is not guaranteed to be able to successfully decapsulate the ciphertext. Depending on the construction, this leads to either an incorrect shared secret, or False. This behavior is captured in the notions of decryption and decapsulation failures. We define this for KEMs in Definition 2.1.11. An analogous definition can be given for PKEs.

---

[2] In Definition 2.1.10, we derive ss as an output of Encaps. Alternative formalizations of (non-contributory) key-encapsulation mechanisms sometimes list the shared secret as an *input*, instead. Encaps can then be a deterministic algorithm. In this work, we consider sampling the secret to be part of Encaps.

[3] Such failures are often permitted to improve performance, but are sometimes even unavoidable.

Definition 2.1.11 (Decapsulation failure) *A KEM is said to be correct when, for all key pairs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$*, and every run of* Encaps*, that is, given* $(\mathsf{ss}, \mathsf{c}) \leftarrow \mathsf{Encaps}(\mathsf{pk})$ *and* $\mathsf{ss}' \leftarrow \mathsf{Decaps}(\mathsf{c}, \mathsf{sk})$*, it holds that* $\mathsf{ss} = \mathsf{ss}'$*. A decapsulation failure refers to a run of* Decaps *that, for a given* c *and* sk*, returns* False*.*

Two notions typically define the security of public-key encryption schemes (PKEs) and KEMs: indistinguishability under chosen-plaintext attacks (IND-CPA) and indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2, to explicitly stress adaptivity, but often simply IND-CCA). Tracing back to Goldwasser and Micali [GM84], the literature contains many subtly different interpretations of both of these concepts, as well as equivalence proofs among some of their variants (but not all).[4] For the purpose of this work, we adhere to Definition 2.1.12 and Definition 2.1.13, below.

We first look at IND-CPA, in the context of public-key encryption schemes. The intuition here is that, even with knowledge of two candidates for the plaintext, an adversary is unable to reliably link one of these to a given ciphertext. Note that, in contrast to the EU-CMA game defined in Definition 2.1.8, an adversary is able to perform encryptions of the involved plaintexts; as Enc is probabilistic, this should not lead to an advantage [KL14].

Definition 2.1.12 (IND-CPA for PKEs) *Given a public-key encryption scheme* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$*, consider the following game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$*:*

1. $\mathcal{C}$ *runs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ *and sends* pk *to* $\mathcal{A}$*.*
2. $\mathcal{A}$ *may perform a polynomial number of calls to* Enc *and other operations.*
3. $\mathcal{A}$ *sends freely chosen messages* m_0 *and* m_1 *to* $\mathcal{C}$*.*
4. $\mathcal{C}$ *randomly chooses* $b \in \{0, 1\}$*, and responds with* $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$*.*
5. $\mathcal{A}$ *may perform a polynomial number of calls to* Enc *and other operations.*
6. $\mathcal{A}$ *outputs a guess* b'*. The attack is successful if* $b = b'$*.*

The public-key encryption scheme is said to be IND-CPA secure when any adversary $\mathcal{A}$*, running in time polynomial in the security parameter, has negligible advantage over random guessing.*

---

[4] E.g., in [BHK15], Bellare, Hofheinz, and Kiltz show that the formalization of various interpretations of the IND-CCA notion are equivalent for KEMs, but that this is not the case for PKEs.

We stress the fact that the adversary can make queries to Enc that include the messages m_0 and m_1. This is in stark contrast to the EU-CMA game for signatures, but a necessity to ensure indistinguishability: an Enc routine that is not sufficiently hiding for repeated encryptions of the same message should not meet the definition.

In the IND-CCA2 security game, the adversary is given access to an oracle that allows it to decrypt all but the challenged ciphertext. Still, the adversary should be unable to distinguish between two ciphertexts. The difference between IND-CCA2 and IND-CCA here relates to whether the adversary is allowed to request decryptions after having received the challenge ciphertext.

In the context of key encapsulation, the game is slightly different. Rather than the adversary picking two plaintexts for the challenger to encrypt, the challenger produces a properly encapsulated shared secret, and picks a random element in the same range. The adversary should then, given the ciphertext and both candidate secrets, be unable to distinguish the shared secret from the random element. For KEMs, there is no meaningful non-adaptivity notion.

Definition 2.1.13 (IND-CCA2 security for KEMs) *Given a key-encapsulation mechanism* (KeyGen, Encaps, Decaps)*, consider the following game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$*:*

1. $\mathcal{C}$ *runs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$.
2. $\mathcal{C}$ *runs* $(\mathsf{ss}_0, \mathsf{c}) \leftarrow \mathsf{Encaps}(\mathsf{pk})$.
3. $\mathcal{C}$ *randomly selects* ss_1 *in the range of* $\mathsf{Encaps}()$*, as well as* $b \in \{0, 1\}$.
4. $\mathcal{C}$ *sends* $\mathsf{pk}, \mathsf{ss}_b$ *and* c *to* $\mathcal{A}$.
5. $\mathcal{A}$ *may perform a polynomial number of operations, as well as queries to an oracle computing* $\mathsf{Decaps}(\mathsf{sk}, \mathsf{c}')$ *for ciphertexts* $\mathsf{c}' \neq \mathsf{c}$.
6. $\mathcal{A}$ *outputs a guess* b'*. The attack is successful if* $b = b'$.

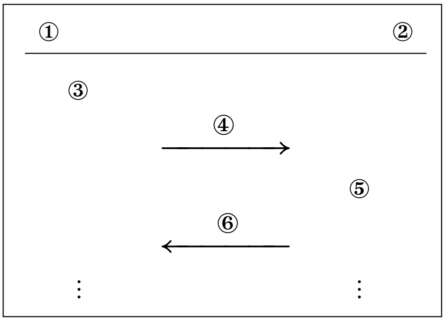
The key-encapsulation mechanism is said to be IND-CCA2 secure when any adversary $\mathcal{A}$*, running in time polynomial in the security parameter, has negligible advantage over random guessing.*

2.1.5 Algorithms and protocols

Throughout this work, we use pseudocode to describe various algorithms and protocols. See Algorithm 0 and Figure 0 for a description of the different fields, and, e.g., Algorithm 1 on page 46 and Figure 4.1 on page 115 for concrete instances.

```
Algorithm 0 ① (②)                                  ③
④
...
...
return ⑤
```

① Algorithm name. ② Algorithm parameters; can vary per system or algorithm instance. ③ Scheme-defined constants and functions. ④ Pseudocode statements. ⑤ Output.



① Protocol initiator; typically prover. ② Responder; typically challenger. ③ Operations(s) by initiator. ④ Message from initiator to responder. ⑤ Operations(s) by responder. ⑥ Message from responder to initiator. Et cetera.

Figure 0: Example protocol

2.2 NIST's Post-Quantum Cryptography Standardization project

Cryptographic research, while often quite academic, is strongly connected to real-world deployment. This is most clearly demonstrated by the ubiquity of protocols such as TLS[5] and EMV[6], with billions of uses every day; perhaps without noticing, society heavily relies on the correct and efficient functioning of cryptographic primitives. None of this would be possible without clear agreements on interactions between all involved systems. To ensure interoperability, standardization bodies issue cryptographic standards. One such body is the United States National Institute of Standards and Technology: NIST.

---

[5] Transport Layer Security; providing secure network communication, most famously underlying HTTPS.

[6] The global payment standard for chip-based smart-card payments.

While NIST is an agency of the United States government, its standards have worldwide impact. All federal agencies are required to be NIST-compliant through the Federal Information Security Management Act (FISMA),[7] and many international organizations and companies adhere to the same recommendations.

In cryptography, NIST is notably responsible for the Advanced Encryption Standard (AES) [NIST01] and the Secure Hashing Algorithms (SHA-1, SHA-2, and, recently, SHA-3) [NIST15a; NIST15b]. While SHA-1 and SHA-2 were designed by the National Security Agency (NSA), AES and SHA-3 are the result of a more open process. To replace the then-common Data Encryption Standard (DES), NIST ran an open competition and chose Rijndael [DR99] as AES in 2001. Keccak [BDP+11] was selected as SHA-3 to complement SHA-1 and SHA-2 in 2015 in much the same fashion. In 2016, NIST announced a project aiming to standardize post-quantum cryptography [NIST16]. This announcement followed a workshop in early 2015, with updated recommendations by the NSA heralding a transition to quantum-resistant cryptography [NSA15]. In contrast to the previous standardization efforts, NIST expressed the intention to publish a broader portfolio of approved algorithms, and explicitly refrains from labeling this project a competition.

NIST solicited submissions of public-key encryption schemes, key-encapsulation mechanisms and digital signature schemes. In the first round, 82 schemes were submitted, of which 69 were deemed 'complete and proper' [Moo18]. Of these, 26 subsequently survived to the second round [NIST19]. The work described in this thesis relates to three such submissions: SPHINCS$^+$, MQDSS and NTRU-HRSS (the latter subsequently merged with NTRUEncrypt to form NTRU). It is worth noting that NIST explicitly started a separate process to standardize stateful hash-based signatures, following the IETF. The IETF has since published RFC 8391 [HBG+18] and RFC 8554 [CMF19], the former of which will be discussed in Section 3.3.5.

In their call for proposals, NIST defines five categories to classify the security levels of the submissions [NIST16, Sec. 4.A.5]. The security of AES, SHA-2, and SHA-3 are used as a frame of reference, comparing the security to key-search attacks against AES and collision-finding attacks against SHA-2 and SHA-3. In increasing order of difficulty, categories 1 through 5 equate security to AES-128, SHA-256/SHA3-256, AES-192, SHA-384/SHA3-384, and AES-256, respectively. Throughout this work, we simply refer to these as 'category 1' through 'category 5'.

---

[7] Public Law 107-347 (Title III); December 17, 2002. Revised as the Federal Information Security Modernization Act as per Public Law 113-283; December 18, 2014.

2.3 Cryptographic engineering

The most obvious focus in the design of all schemes described in this thesis is resistance against quantum computers, but there is another important commonality. All work discussed here centers around real-world software implementations. Various sections in the upcoming chapters describe specific aspects of these implementations, detailing optimization techniques and considerations. Here, we discuss some aspects of cryptographic engineering in more general terms.

In general, cryptographic engineering is characterized by two —sometimes competing— fundamental goals: optimal use of resources and security. Towards optimality, we aim to minimize the computational costs (measured in clock cycles) as well as the memory usage (measuring both volatile memory usage and code size). While this may not matter for the average desktop user, large data centers filled with servers that terminate TLS connections may see the impact of every percentage point. To achieve security, we can attempt to counter a variety of implementation attacks, ranging from passive timing side-channel attacks to disruptive fault injection. There is extensive literature on both counts; we make no attempt to summarize that here, but instead discuss some intuitions that underly this work.

2.3.1 Programming abstractions

This thesis describes code written at three different abstract layers.[8] At the highest level of abstraction, we use Python for prototyping and experimental scripts; it is both quick to write and easy to read. For the majority of the code, we rely on C; it is portable and efficient, and provides some control over memory usage. For the nitty-gritty, we use assembly; while tedious and laborious to write, it allows for very precise optimizations, as well as a high level of control towards preventing side-channel leakage. Besides being able to meticulously schedule instructions, controlling which data is loaded into registers or stored in memory enables optimizations that are impossible in a compiled language like C.

Code generation

While on the subject of compilers, we must remark on code generation. When writing C, the code is typically compiled to processor instructions by a general-

---

[8] Four, if one insists to count the listed pseudocode separately.

purpose C compiler, such as GCC or Clang.[9,10] These compilers are the result of countless hours of engineering effort and extensive research and development. As a result, the code produced by modern versions of these compilers is extremely efficient. We must note, however, that cryptographic code is typically quite different from most C code. Whereas typical C code may see many complex memory accesses patterns, interrupts and context switches, cryptographic code is typically an extremely monolithic and straight-lined sequence of arithmetic operations.[11]

On one hand, this is ideal for compilers: without runtime decisions, the code paths are laid out at compile-time. On the other, it is clearly not the expectation they were designed with. Consequently, their behavior is somewhat erratic, and arithmetic that seems straight-forward sometimes gets mangled. Most importantly, it is often impossible for a compiler to keep track of the bigger picture. This is typically showcased by its decisions regarding which values to retain in registers, and which to swap out to memory. Still, manually keeping track of this and writing assembly by hand remains painful and error-prone.

It seems that there is value in a middle ground: code generation. None of the assembly code discussed in this thesis was written directly, by hand; literally all of it was produced by Python scripts. This comes with significant advantages — most relating to readability and ease of programming, but one should also not underestimate the power of perhaps only a hint of post-processing (see, e.g., the discussion in Section 5.4.1). The most noticeable benefit is similar to using assembly macros, but much more powerful: using an expressive language like Python allows us to close the loop on programming abstractions, natively dealing with lists, subroutines and objects while retaining the efficiency of custom assembly. For example, this allows us to transparently keep track of the 'state of rotation' of the words in the ChaCha permutation described in Section 3.6.3.

We should not forget that compilers are ever-improving. Recent experiments with PQM4 [KRS+18] showed as much as a 20% improvement merely by updating the compiler; for SHA-512 as used in [KRS19], we were hard-pressed to outperform compiler-generated code. In general, as processors become more and more complex, it becomes increasingly more difficult to write assembly by hand and take into account the various intricacies in one-off code generation scripts. One may hope that large compiler projects are able to account for this more thoroughly.

---

[9] `https://gcc.gnu.org`

[10] `https://clang.llvm.org`

[11] We come back to this shortly, when discussing timing side-channels.

Parallelism

Parallelism is a term that will return throughout this thesis in various forms. Parallelism is most notably benefited from through vectorization, to the point where it is sometimes tempting to use the two synonymously. We also use it to describe independent, parallel data streams: while the computations cannot actually occur at the same time, they happen independently and can be done in arbitrary order. It is important to always keep the relevant platform in mind — this will often underscore the relevant form of parallelism when this is not clear from context. We will discuss these platforms separately in the next section.

Throughout this work, we do not concern ourselves with 'high-level' parallelism, i.e., multi-threading and multi-core processors. For benchmarking purposes, our code always runs on a single CPU core. Still, vectorization allows us to benefit from parallelism in the various algorithms. Many algorithms contain subroutines that repeatedly apply the same operation (or sequence of operations) to independent blocks of data. Consider, e.g., summing terms in polynomial multiplication. Rather than using separate clock cycles to perform an operation (e.g., an addition) on individual operands, vector instructions allow us to operate in tuples of operands at the same time. This is achieved by replacing the narrow general-purpose registers by wider vector registers and simple arithmetic by the corresponding multi-operand instructions. It is important to realize that, e.g., 256-bit vector registers do not immediately allow 256-bit additions: the operands are still separate, and intermediate carry bits will be dropped. Indeed, an addition of 8 × 32 bits is exactly that: addition of eight operands of 32 bits.

Besides combining the arithmetic operation, the effect on memory operations is perhaps even more important. The wide vector registers allow loading of large blocks of memory at once, as well as maintaining many more intermediate values in active registers. This pairs well with the fact that typical vector instruction sets come with instructions that allow the operands to be moved around (such as the `vpshuf` instructions on AVX2) and their size liberally reinterpreted (i.e., interpreting 4 × 64-bit blocks as 16 × 16-bit integers).

Not every scheme lends itself well to vectorization. The cost of repacking the input data to fit the dimensions and memory layout suitable for vector registers can be more costly than the saved operations. Finding subroutines that parallelize well is what makes vectorization a versatile optimization technique. It is important to keep this in mind already during scheme design. During implementation, singling out the critical sections to implement in hand-optimized assembly combines well with making use of any inherent parallelism; while compilers are ever-improving, often a significant improvement can still be made by manually exploiting algorithm-level parallelism.

One should also not overlook the importance of operations that can be processed 'in parallel' because of data independence, even on platforms that do not support vectorization. Recognizing such blocks of instructions is crucial for efficient register allocation. Efficient register allocation, i.e., the assignment of abstract variables to concrete registers, directly leads to a reduction in the number of (expensive) memory operations. Separating parallel streams of operations typically reduces the number of operands that need to be kept available, reducing register pressure and thus simplifying allocation. This is visualized in dependency graphs, and lies at the root of more complex optimization problems such as the polynomial inversions described in Section 5.2, but also plays a fundamental role in seemingly simpler problems such as the schoolbook multiplication routines in Section 5.4.2.

Modern high-end processors typically perform out-of-order execution: by isolating independent data streams and sequences of instructions, the processor can effectively skip ahead to other instructions while it would otherwise be waiting for results of previous operations (including arithmetic, but in particular also memory accesses). This pairs well with algorithms designed with parallelism in mind, but directly competes with hand-optimized assembly, where the programmer carefully arranges the instructions in an order they consider optimal. While this can imply that hand-optimized assembly does not see much improvement from out-of-order execution, the ever-increasing complexity of processors sometimes leads to quite eccentric routines that prove to be optimal. We do not consider this effect in this thesis explicitly.

2.3.2 Side-channel resistance

This thesis will not at all touch upon the extensive and diverse field of side-channel attacks. Still, it remains of the utmost importance to consider the possibility of side channels when writing and publishing cryptographic code — one can never really be certain where it will end up being used. While it is impossible to counter all side channels without implementing countermeasures that fit the specific platform, threat model, and usage scenario, adhering to several basic principles already goes a long way. We briefly touch upon some intuitions.

The schoolbook examples of side channels are timing and power leakage due to the execution of different instructions. This occurs most notably when using branch instructions (e.g., if-statements) that depend on secret data. Here, secret data is anything that follows from parts of the secret key — most data is quickly ‘contaminated’ by interactions with secrets. Of course the final results of a computation also depend on secrets. This is why it is important to rigorously prove that the output and secret are sufficiently unrelated, for example by coupling this relation to solving a mathematically ‘hard’ problem. No such proofs exist for intermediate results: it is thus crucial not to reveal information about these values.

Branching on secrets

It often follows from secret data whether or not a certain arithmetic operation (and thus, an instruction) must be performed. Actually performing a different sequence of instructions can lead to a difference in runtime behavior, as well as different power-consumption patterns. To prevent this, this conditional behavior is captured in arithmetic that may lead to vacuous operations. For example, rather than conditionally adding a value based on a secret bit, consider first multiplying the value by the bit representing the condition and then adding the result.

Somewhat related, note that this kind of defensive coding requires a thorough understanding of the specific instruction set. The `DIV` instruction on Intel processors is a well-known example of an instruction that takes variable time depending on input values — the same is true for multiplication on some platforms.[12] For compiled code, modern compilers make it increasingly complex to give such guarantees without disassembling the result and carefully inspecting each instruction. Secret-aware compilers could be an important step in cryptographic engineering.

Furthermore, we must remark on the fact that avoiding branches on secret data can be directly at odds with performance optimization. For example, performing the maximum possible number of loop iterations is a typical defensive-but-suboptimal pattern. While this may be a reasonable trade-off for highly specific industry applications, we believe that open source software should be defensive by default. In particular, although it remains hard to give evidence of absence, all software described in this work is intended to be free of such secret-dependent branches.

---

[12] This is the case on the ARM Cortex-M3. Such variable-time instructions are notoriously poorly documented; for the Cortex-M3, time savings occur for operands below 65536 [ARMb], but also for powers of two and specifically for zero [Gro15].

Memory access

Closely related to secret-dependent branching is secret-dependent memory access. Here, the underlying idea is to exploit differences in the behavior of different parts of memory (see, e.g., the extensive literature on cache-timing attacks). When secrets are used to index parts of data structures that live in distinct parts of memory, this difference in behavior may expose which part of the structure was accessed. In turn, this reveals information about the index used to access it. Typical countermeasures include randomizing the access pattern or simply retrieving the entire data structure from memory for each access. It is not hard to imagine that this, too, is detrimental to performance.

Not all platforms described in this thesis actually provide multiple levels of memory — on embedded platforms, caches are often an optional peripheral. Still, as it is hard to control where the code ends up in practice, all software presented in this work aims to avoid such memory accesses.

Constant-time code

Underlying the above is the implicit suggestion that code must run in *constant time*. While this is often the term used in practice, it is somewhat misleading. Rather than demanding the exact same cycle count for every iteration, we require the execution time not to be related to the secrets in a revealing way. This is typically done by feeding the secret through a pseudo-random generator that leads to a statistically independent stream of output. This is relevant in particular in the context of post-quantum cryptography, where rejection sampling is a common practice, but also occurs in classical systems that require certain properties of random values (e.g., in primality testing).

Constant-time code is sometimes referred to as being *isochronous*. Perhaps even more confusingly, this term is occasionally used to describe code that runs in variable time, but where the timing does not depend on secret inputs. We will not use this term throughout the remainder of this thesis.

2.4 Platforms and architectures

Throughout this thesis, we will make passing note of various platforms and architectures. This section serves as a central point of reference, where we go into each one of them in some more detail.

2.4.1 Intel x86, x86-64 and AVX2

Found in most personal computers, laptops and servers, x86 is currently likely the most widespread architecture for high-end processors. Originally a 16-bit architecture from the 1970s, throughout this thesis we use x86-64, the modern 64-bit version. This architecture defines sixteen general-purpose registers of 64 bits each, the lower parts of which can be used as 32, 16 and 8-bit registers as well.

The x86 instruction set has seen numerous extensions. In this work, we are particularly interested in the AVX2 vector extensions. The AVX2 extensions give us access to another sixteen registers of 256 bits each, but these are far from general-purpose. Through 'single instruction, multiple data' (SIMD) instructions, these registers can be used as vectors of smaller values for efficient parallelization (see Section 2.3.1), but they also allow us to efficiently work with arithmetic on blocks of 16×256 bits. As with the general-purpose registers, the lower half of these registers can be addressed explicitly. There is still a strong lane boundary between the higher and the lower 128 bits, showing the 128-bit SSE heritage of AVX2: AVX2 instructions generally do not allow crossing between the high and low lanes, and naively mixing SSE and AVX2 instructions results in large penalties.

Other relevant extensions include CLMUL for carry-less multiplication (i.e., multiplication of polynomials over $\mathbb{F}_2$), BMI-1 and BMI-2 for bit manipulation (i.e., extracting, shuffling and inserting bits), and AES-NI for efficient AES operations. These will be briefly mentioned where relevant.

Intel Core i7 Haswell

While development can be done on any platform offering the required instruction set, accurate benchmarking requires somewhat more careful configuration. Unless otherwise specified, all benchmarks in this work were performed on a single core of an Intel Core i7-4770K Haswell running at 3.5 GHz. This core comes with 32 KiB of L1 instruction cache and data cache, 256 KiB of L2 cache and 8 MB of L3 (shared) cache. To aid comparison, we follow the standard practice of disabling TurboBoost and hyper-threading (see, e.g., benchmarking instructions by [BL]).

2.4.2 ARMv7 on the Cortex-M series

On smaller, embedded devices, one is unlikely to find the large and expensive x86 processors. Instead, the ARM Cortex-M series has been rapidly gaining popularity.

These microprocessors combine the ease of use and computational power of 32-bit arithmetic with low cost and a small energy footprint, quickly replacing 8-bit processors in small integrated controllers [ARMa].

In this work, we will discuss implementations targeting the Cortex-M3 and the Cortex-M4. These platforms respectively implement the ARMv7-M and ARMv7E-M architectures. Note that these processors are not sold and produced by ARM. Instead, the architectures are licensed to manufacturers such as STMicroelectronics, NXP Semiconductors and Atmel, each producing their own chips and development boards. We rely on the STM32 Discovery boards by STMicroelectronics. In particular, we use the STM32L100C and the STM32F407.

STM32 Discovery boards

The STM32L100C is ST's Cortex-M3 Discovery board. It features a Cortex-M3 running at 32 MHz, no data or instruction caches, 16 KiB of RAM and 256 KiB of persistent flash memory. Notably, the board contains a direct-memory-access (DMA) controller, allowing for fast access to memory through its serial communication interface. It implements the ARMv7-M architecture.

The STM32F407 is ST's Cortex-M4 Discovery board. Running at a maximum speed of 168 MHz and implementing the ARMv7E-M architecture, it is somewhat more powerful than the M3. In the context of this work, that becomes particularly apparent in the form of its 192 KiB of RAM (see the discussion in Section 3.6 and in [KRS+19], detailing how memory usage is typically the limiting factor). The STM32F407 has no data caches (see Section 5.5.2), and boasts 1 MiB of ROM.

For both boards, we make use of the libopencm3 firmware library.[13] Originally targeting the Cortex-M3, the library provides an abstraction layer for various Cortex-M microcontrollers. This makes it easier to write cross-platform code without worrying about specific microcontroller intricacies. Taking this one step further, we provide usage examples and a wrapper around functionality that is relevant to users implementing, testing and benchmarking cryptographic primitives (stm32-getting-started; see Section 1.3.1). The effectiveness of this approach is demonstrated in its use in various subsequent projects (most notably PQM4 [KRS+18]) and the yearly Cryptographic Engineering course at Radboud University. To compile for these platforms, we use the GNU ARM Embedded Toolchain.[14]

---

[13] `https://github.com/libopencm3/libopencm3`

[14] `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm`

ARMv7-M

Several variants of the ARMv7 architecture exist. For the Cortex-M series, we are interested in ARMv7-M and the ARMv7E-M variant. These architectures define sixteen 32-bit registers, three of which with a special purpose: `R13` is the stack pointer, `R14` is the link register, and `R15` contains the program counter. When hand-writing assembly code, `R14` is easily freed up for use as an additional general-purpose register. To reduce code size, ARM introduced the compressed Thumb instruction set containing a 16-bit-encoded subset of ARMv7. Thumb2 builds upon this by allowing for a mix of 16-bit and 32-bit instructions. Instructions at a non-32-bit offset can introduce a penalty; this requires careful consideration (see Section 5.4.1). ARMv7 specifies a simple three-stage pipeline, making it feasible for developers to reason about cycle-accurate execution times.

An important and distinguishing feature of ARMv7-M is the combined barrel shifter. Besides allowing for two source registers and a separate destination register when performing typical arithmetic instructions, one of the source registers can optionally be rotated or shifted by a fixed distance. Crucially, this does not induce a penalty over the standard cost of arithmetic. This is especially beneficial in cryptographic primitives, where rotations and shifts are often naturally surrounded by arithmetic operations.

Having established that vectorization is an important optimization mechanism, it is no surprise that ARMv7E-M introduces SIMD instructions. The DSP instruction set, designed for signal processing, adds arithmetic on pairs of 16-bit and vectors of 8-bit values, as well as a wide variety of instructions to combine multiplications and additions. We make extensive use of this in the work described in Section 5.3.

2.4.3 Java Card

We use the Java Card platform for the work described in Section 3.4. The situation here is quite different from the platforms discussed above: Java Card defines an environment standard rather than a precise architecture, specifying a Java API rather than instructions, registers and pipelines. Consequently, manufacturers have more freedom when designing the underlying hardware, and developers are guaranteed portability — naturally, this comes at the cost of performance.

We describe this platform in some more detail in Section 3.4.1.

Chapter 3

Hash-based signatures

> This chapter is based on the peer-reviewed papers *“ARMed SPHINCS – Computing a 41 KB signature in 16 KB of RAM”* [HRS16a], *“Mitigating Multi-Target Attacks in Hash-based Signatures”* [HRS16b], and *“Is Java Card ready for hash-based signatures?”* [LPR+18], the preprint *“SPHINCS$^+$”* [BHK+19], and the SPHINCS$^+$ submission to NIST’s Post-Quantum Cryptography Standardization project [BDE+17].

This is the first of two chapters on digital signatures. Here, we consider hash-based signature schemes: signature schemes that rely solely on the existence of a secure one-way function. The approach discussed here is often considered to be the most conservative option available — not just when facing an adversary equipped with a quantum computer, but also compared to its classical counterparts. In fact, it has been shown that the existence of a secure one-way function is a provably minimal requirement for the existence of any signature scheme [Rom90], and for hash-based signature schemes it is also sufficient. Hash-based signatures are among the oldest public-key cryptosystems, dating back to the publication of Lamport’s one-time signature scheme in 1979 [Lam79], and their security is well-understood.

A natural question one may be asking at this point is why these schemes have not been deployed already, to provide digital signatures in the presence of classical adversaries. The reasons for this are, unfortunately, manifold. Historically, the large signature size has been an important obstacle towards practical schemes. As there are various time/memory trade-offs to be made, this goes hand in hand with poor performance. Perhaps the biggest obstacle, however, was of an entirely different nature: producers of hash-based signatures were required to maintain and update a *state*. Rather than being able to generate a key pair once and then use it arbitrarily and indefinitely, the secret key effectively changes with every signature. This seemingly innocuous difference has a serious impact on real-world applications, and directly conflicts with typical user expectations.

The ever-growing threat of a large-scale quantum computer has renewed interest in this field, and over the last decade there has been a considerable amount of development. This resulted in XMSS [BDH11], demonstrating the viability of hash-based signatures, and subsequently in SPHINCS [BHH+15], a variant that does not need to maintain state. The former has since led to the publication of an RFC [HBG+18], while the latter laid the groundwork for the SPHINCS$^+$ submission to NIST's Post-Quantum Cryptography Standardization project.

The XMSS and SPHINCS constructions will be the main focus on this chapter, and are explained in detail in Sections 3.3 and 3.5. The remainder of this chapter concerns variations and tweaks to these schemes, as well as descriptions of practical implementations. Notably, the SPHINCS$^+$ framework is discussed in Section 3.7 Before getting to that, let us first consider the basic building block: one-time signature schemes.

3.1 One-time signature schemes

In Chapter 1 and Section 2.1.3, the concept of digital signatures was introduced. The signer produces a digital signature on a message of their choosing using their secret key. Afterwards, a verifier can use the corresponding public key to check that the signature was produced with that particular secret key, and corresponds to that specific message. This relation is fundamentally asymmetric: only the holder of the secret key can produce the signature, while anyone with access to the public key can verify the correctness of a signature for a given message.

For one-time signature schemes, an additional element of asymmetry is introduced. As the name suggests, a signer can use their secret key *only once*, i.e., only use it to produce a single signature. Security degrades when the same key pair is used for more than one message.[1] The situation for the verifier is unchanged: they can freely use the public key to verify multiple messages.

In terms of the EU-CMA security game specified in Definition 2.1.8, we require only a slight modification to cover this additional limitation. Rather than allowing the adversary to query the challenger for q signatures, they are only allowed to do so once. For completeness, we capture this formally in Definition 3.1.1.

Definition 3.1.1 (one-time EU-CMA) *Given a signature scheme* (KeyGen, Sign, Verify), *consider the following game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$*:*

---

[1] Just how quickly security degrades for repeated use strongly depends on the specific scheme [BH17].

1. *$\mathcal{C}$ runs* $(\mathsf{pk}, \mathsf{sk}) = \mathsf{KeyGen}()$ *and sends* pk *to* $\mathcal{A}$.
2. *$\mathcal{A}$ sends a freely chosen message m to $\mathcal{C}$.*
3. *$\mathcal{C}$ responds with* $\sigma = \mathsf{Sign}(m, \mathsf{sk})$.
4. *$\mathcal{A}$ outputs a pair (m', σ'). This is a valid forgery when* $\mathsf{Verify}(m', \sigma', \mathsf{pk}) = \mathsf{True}$ *and* $m' \neq m$.

The signature scheme is said to be one-time EU-CMA when any adversary $\mathcal{A}$, running in time polynomial in the security parameter, has negligible success probability.

3.1.1 Lamport's one-time signature scheme

The first hash-based signature scheme is typically credited to a technical report by Lamport [Lam79], although the same scheme is sometimes referred to as 'the Lamport-Diffie one-time signature scheme' (notably in direct follow-up work by Merkle [Mer90]). The description below is a slight reformulation,[2] so as to match presentation as is common in current literature. See Figure 3.1 for an illustration of a signature generated as described below, and Algorithms 1, 2 and 3 for pseudocode.

We assume a message $m \in \{0,1\}^n$, i.e., binary messages m of n bits. Define $F\colon \{0,1\}^k \to \{0,1\}^k$ to be a one-way function,[3] that is, given a value $y = F(x)$, it is infeasible to find an x' such that $F(x') = y$.

Before being able to produce signatures, the signer needs to generate a secret key and the corresponding public key. The secret key consists of $2n$ random values of k bits each. We label these $s_{i,j}$ for $i \in \{1, \ldots, n\}$ and $j \in \{0,1\}$. The public key consists of the result of applying F to each secret value. We write $p_{i,j} \leftarrow F(s_{i,j})$, and publish the public key $\mathsf{pk} = (p_{1,0}, p_{1,1}, \ldots, p_{n,0}, p_{n,1})$.

Given a message m, the signer selectively reveals the secrets $s_{i,j}$ corresponding to the values of the bits in m. More precisely, these are the values s_{i,m_i}, where m_i is the i^{th} bit of m. The list of these values make up the signature. Given such a signature, the verifier can also apply F to a revealed secret, and obtain $F(s_{i,m_i})$. The signature is to be considered valid if indeed $F(s_{i,m_i}) = p_{i,m_i}$ for all i bits of m.

---

[2] To the familiar reader, the original description by Lamport may be of interest. Rather than explicitly specifying messages to be a sequence of bits that indicate which secrets to reveal, they are more generally mapped to subsets of the domain of the one-way function.

[3] For now, we suffice with this intuitive definition, and defer discussion on collision resistance and multi-target attacks to Section 3.3.

Algorithm 1 Lamport.KeyGen ( ) n, k, F

```
for $i \in \{1, \ldots, n\}$ do
    for $j \in \{0, 1\}$ do
        $s_{i,j} \xleftarrow{\$} \{0, 1\}^k$
        $p_{i,j} \leftarrow F(s_{i,j})$
    end for
end for
return pk = $(p_{1,0}, p_{1,1}, \ldots, p_{n,0}, p_{n,1})$, sk = $(s_{1,0}, s_{1,1}, \ldots, s_{n,0}, s_{n,1})$
```

Algorithm 2 Lamport.Sign ($m, s_{i,j} \in$ sk) n, F

```
$(m_1, \ldots, m_n) = m$                    ▷ Split $m$ such that $m_i \in \{0, 1\}$
for $i \in \{1, \ldots, n\}$ do
    $\sigma_i = s_{i,m_i}$
end for
return $\sigma = (\sigma_1, \ldots, \sigma_n)$
```

Algorithm 3 Lamport.Verify ($m, \sigma_i \in \sigma, p_{i,j} \in$ pk) n, F

```
$(m_1, \ldots, m_n) = m$                    ▷ Split $m$ such that $m_i \in \{0, 1\}$
for $i \in \{1, \ldots, n\}$ do
    if $F(\sigma_i) \neq p_{i,m_i}$ then
        return False
    end if
end for
return True
```

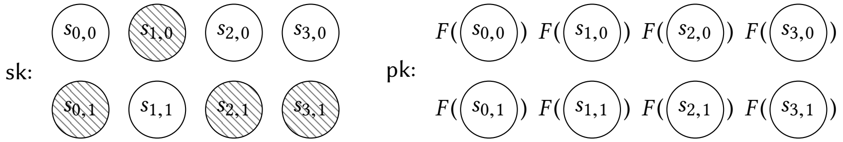

sk:
$s_{0,0}$ $s_{1,0}$ $s_{2,0}$ $s_{3,0}$
$s_{0,1}$ $s_{1,1}$ $s_{2,1}$ $s_{3,1}$
pk:
$F(s_{0,0})$ $F(s_{1,0})$ $F(s_{2,0})$ $F(s_{3,0})$
$F(s_{0,1})$ $F(s_{1,1})$ $F(s_{2,1})$ $F(s_{3,1})$

Figure 3.1: The Lamport signature scheme. Here, m = `1011` is used as an example message and the grayed nodes indicate the revealed secrets.

As a consequence of the one-wayness of F, an outside forger cannot create such a signature for any other m': they have no knowledge of any secrets $s_{i,j}$ other than the ones exactly corresponding to the bits of m. Changing as much as one bit in m would require including an unrevealed secret in the signature.

Intuitively, it should now be straight-forward to see why such a secret key cannot be used for multiple messages. If one was to also publish a signature for a different message m' based on the same secret values $s_{i,j}$, an outside observer would learn the secrets s_{i,m_i} as well as s_{i,m'_i}. This not only allows them to reproduce the signature for m and m' (which is not a problem, as these have been legitimately published already), but also for any message m'' that can be constructed by combining bits[4] of m and m', i.e., where $m''_i \in \{m_i, m'_i\}$ for all i.

Merkle's improvement

Besides the unmistakable downside of only being able to sign once per key pair, Lamport signatures are also quite large. At $2kn$ bits, signing a 256-bit hash with $k = 256$ leads to signatures of 128 KiB each. In [Mer90], Merkle demonstrates a slight improvement over Lamport's scheme. Rather than revealing secrets for both the 1-bits and 0-bits, only revealing the secrets corresponding to the 1-bits halves the size of the signature on average.

To prevent forgeries that flip 1-bits to 0 and simply omit the corresponding secret from the signature, the total number of zeros in m is appended to m and also signed. Flipping a 1 to a 0 in the original message will cause an inverse bit flip in this 'checksum' c: through carrying, the least significant 0-bit will flip to a 1. Signing this new 1-bit requires the forger to include an unrevealed secret.

---

[4] A similar problem is addressed in Lamport's original description, where he briefly examines a scheme equivalent to revealing only the secrets corresponding to the 1-bits, enabling forgeries for messages where 1-bits are flipped to 0; this is naturally prevented by revealing secrets for both m and its complement.

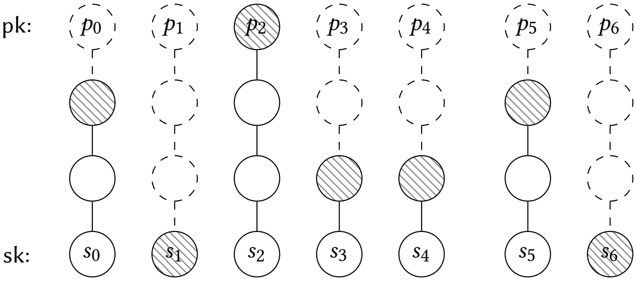


Figure 3.2: A Winternitz signature (marked in gray). This example illustrates $n = 10$ and $w = 4$, producing a signature for $m =$ `10 00 11 01 01`. Thus, $c = 8 =$ `10 00`.

3.1.2 Winternitz' improvement

Also in [Mer90], Merkle describes a variant credited to Winternitz that allows reducing the size of the signature at the cost of additional runtime. The intuition is that, rather than using the inclusion of a secret to authenticate one message bit, multiple message bits can be covered using a single secret and a varying number of applications of the one-way function. The applications of the one-way function are commonly referred to as *chains*, and the function itself the *chaining function*.

As an example, let us look at the case where $w = 2^4 = 16$, i.e., the case where bits are grouped in groups of four. For a message of n bits, we then need only $n/4$ secrets. For each such group, the signer applies the one-way function F as often as the value of the group indicates. Note that the maximum value, and thus the maximum number of required applications, is $w - 1$. Consequently, the public values need to be $p_i = F^{w-1}(s_i) = F^{15}(s_i)$. To sign the group $m_i = 1101$, i.e., the decimal value 13, the signer needs to compute and reveal $\sigma_i = F^{13}(s_i)$. This leaves $w - 1 - 13 = 2$ applications of F to the verifier, after which they can check that indeed $p_i = F^2(\sigma_i) = F^2(F^{13}(s_i))$. A smaller example of this is illustrated in Figure 3.2.

Like in Merkle's improvement described above, the naive scheme allows for a trivial way to construct forgeries. A forger could construct a message m' that is equal to m except for the i^{th} group of bits (we can choose any i and repeat this procedure without loss of generality). For this group, they choose m'_i such that $m'_i > m_i$. They can then obtain σ'_i by simply applying F an additional $m'_i - m_i$ times to $F(m_i)$, and complete the forgery. The verifier now applies one less iteration of F and concludes that indeed $p_i = F^{w-1-m'_i}(F^{m'_i-m_i}(F^{m_i}(s_i)))$.

As before, these forgeries can be prevented by including a negated checksum.[5] Let us introduce $\ell_1 = \lceil \frac{n}{\log(w)} \rceil$ for the number of values that are to be signed. The checksum is then computed as $c = \Sigma_{i=1}^{\ell_1}(w - 1 - m_i)$. Expressed in base w, this checksum requires $\ell_2 = \lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \rfloor + 1$ values, signed in the same way as the message. This implies that the keys must consist of $\ell = \ell_1 + \ell_2$ values s_i and p_i, in order to cover both the message and the checksum.

The scheme resulting from the above construction is typically referred to as the Winternitz One-Time Signature scheme (WOTS). See Algorithms 4, 5 and 6 for an algorithmic description. Note that the informal description presented above implicitly suggests w to be a power of 2 (e.g., by operating on 'groups of bits'). This is purely an artifact of convenience of implementation, and in principle the message can be converted to any base w. See Table 3.1 for a summary of common parameter choices.

Algorithm 4 WOTS.KeyGen () ℓ, k, w, F

```
for $i \in \{1, \dots, \ell\}$ do
    $s_i \xleftarrow{\$} \{0,1\}^k$
    $p_i \leftarrow F^{w-1}(s_{i,j})$
end for
return pk = $(p_1, \dots, p_\ell)$, sk = $(s_1, \dots, s_\ell)$
```

Algorithm 5 WOTS.Sign ($m, s_i \in$ sk) ℓ_1, ℓ_2, w, F

```
$(m_1, \dots, m_{\ell_1}) = m$    ▷ Convert $m$ to base-$w$, s.t. $m_i \in \{0, \dots, w-1\}$
for $i \in \{1, \dots, \ell_1\}$ do
    $\sigma_i = F^{m_i}(s_i)$
end for
$c = \Sigma_{i=1}^{\ell_1}(w - 1 - m_i)$
$(c_1, \dots, c_{\ell_2}) = c$    ▷ Convert $c$ to base-$w$, s.t. $c_i \in \{0, \dots, w-1\}$
for $i \in \{1, \dots, \ell_2\}$ do
    $\sigma_{\ell_1+i} = F^{c_i}(s_{\ell_1+i})$
end for
return $\sigma = (\sigma_1, \dots, \sigma_\ell)$
```

---

[5] Including the number of 0-bits, as in Merkle's improvement, can be seen as a specific instance of this.

Algorithm 6 WOTS.Verify $(m, \sigma_i \in \sigma, p_i \in \mathsf{pk})$ $\ell_1, \ell_2, \ell, w, F$

```
$(m_1, \ldots, m_{\ell_1}) = m$        ▷ Convert $m$ to base-$w$, s.t. $m_i \in \{0, \ldots, w-1\}$
for $i \in \{1, \ldots, \ell_1\}$ do
    if $F^{w-1-m_i}(\sigma_i) \neq p_i$ then
        return False
    end if
end for
$c \leftarrow \Sigma_{i=1}^{\ell_1}(w - 1 - m_i)$
$(c_1, \ldots, c_{\ell_2}) = c$        ▷ Convert $c$ to base-$w$, s.t. $c_i \in \{0, \ldots, w-1\}$
for $i \in \{1, \ldots, \ell_2\}$ do
    if $F^{w-1-c_i}(\sigma_{\ell_1+i}) \neq p_{\ell_1+i}$ then
        return False
    end if
end for
return True
```

n	k	w	ℓ_1	ℓ_2	ℓ	bytes
128	128	4	64	4	68	1088
128	128	16	32	3	35	560
128	128	256	16	2	18	288
192	192	4	96	5	101	2424
192	192	16	48	3	51	1224
192	192	256	24	2	26	624
256	256	4	128	5	133	4256
256	256	16	64	3	67	2144
256	256	256	32	2	34	1088

Table 3.1: Common WOTS parameter sets and the resulting signature sizes. Note that RFC 8391 [HBG+18] and the SPHINCS$^+$ submission to NIST's Post-Quantum Cryptography Standardization project [BDE+17] only list parameters where $w = 16$.

Public-key recovery

Given a WOTS signature, one can fully reconstruct the public key corresponding to the secret key that was used to produce the signature. In fact, the careful reader may have observed that this is exactly what happens during the verification routine (compare Algorithm 6 to Algorithm 4). During verification, the one-way function is repeatedly applied to all included values. This continues exactly until the output matches the public key; the actual 'verification' comes down to comparing the produced values to a previously obtained copy of the key to confirm this. Note that this can even be taken a step further by computing a digest over the public key, and comparing that to a previously obtained authentic digest. This effectively compresses the key to the size of a single digest.

In the next section, we will see how this seemingly unremarkable property is extremely useful when constructing a multi-signature scheme from WOTS. For now, we suffice by explicitly defining public-key recovery in Algorithm 7 (and, trivially, from the secret key in Algorithm 7b).

Algorithm 7 WOTS.RecoverPK ($m, \sigma_i \in \sigma$) $\ell_1, \ell_2, \ell, w, F$

```
$(m_1, \ldots, m_{\ell_1}) = m$        ▷ Convert $m$ to base-$w$, s.t. $m_i \in \{0, \ldots, w-1\}$
for $i \in \{1, \ldots, \ell_1\}$ do
    $p_i \leftarrow F^{w-1-m_i}(\sigma_i)$
end for
$c = \Sigma_{i=1}^{\ell_1}(w - 1 - m_i)$
$(c_1, \ldots, c_{\ell_2}) = c$        ▷ Convert $c$ to base-$w$, s.t. $c_i \in \{0, \ldots, w-1\}$
for $i \in \{1, \ldots, \ell_2\}$ do
    $p_{\ell_1+i} \leftarrow F^{w-1-c_i}(\sigma_{\ell_1+i})$
end for
return pk = $(p_1, \ldots, p_\ell)$
```

Algorithm 7b WOTS.RecoverPK ($s_i \in$ sk) ℓ, w, F

```
for $i \in \{1, \ldots, \ell\}$ do
    $p_i \leftarrow F^{w-1}(s_i)$
end for
return pk = $(p_1, \ldots, p_\ell)$
```

3.2 Merkle trees

All signature schemes we have seen so far have been one-time signature schemes. This is typically not the primitive that is useful in practice. We now discuss how to construct a many-time signature scheme from a one-time signature scheme, as proposed by Merkle [Mer90]. Note that a many-time signature scheme is still quite different from a digital signature scheme as defined in Definitions 2.1.7 and 2.1.8: the maximum number of signatures that can be produced is fixed, and the secret key continuously evolves as it is used to produce signatures. This is defined in [BM99], and formally addressed in the context of hash-based signature schemes in [Hül13a].

3.2.1 Many-time digital signature schemes

To properly express the correctness property and EU-CMA notion, we would have to separately define key updates. Instead, for ease of exposition, we make due with a slightly more loose definition that more closely resembles Definition 2.1.7 and real-world implementations (e.g., the API description in [HBG+18], discussed in Section 3.3.5). Notably, we omit an explicit KeyUpdate routine, and instead shoehorn this into the Sign algorithm.

Definition 3.2.1 (Many-time digital signature scheme) *A many-time digital signature scheme is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$*, that, for a given maximum number of signatures* p*, is defined as follows:*

- *The key-generation algorithm* KeyGen *is a probabilistic algorithm that outputs a public key* pk *and a secret key* sk_0*, i.e., a key pair* $(\mathsf{pk}, \mathsf{sk}_0)$.
- *The signing algorithm* Sign *is a possibly probabilistic algorithm that takes as input a message* m *and a secret key* sk_i *to produce a signature* σ*, and an updated secret key* sk_{i+1} *if* $i + 1 < p$*, or* Fail *otherwise.*
- *The verification algorithm* Verify *is a deterministic algorithm that takes as input a message* m*, a signature* σ *and a public key* pk*. It outputs the Boolean value* True *to indicate that the signature is accepted, or* False *to indicate rejection.*

Here, the intuition for the correctness property is that Verify is agnostic to the evolving of the secret key: for a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$, a signature $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}_i)$ on a message m should cause $\mathsf{Verify}(m, \sigma, \mathsf{pk})$ to return True as long as sk_i has evolved from sk_0 within p signatures.

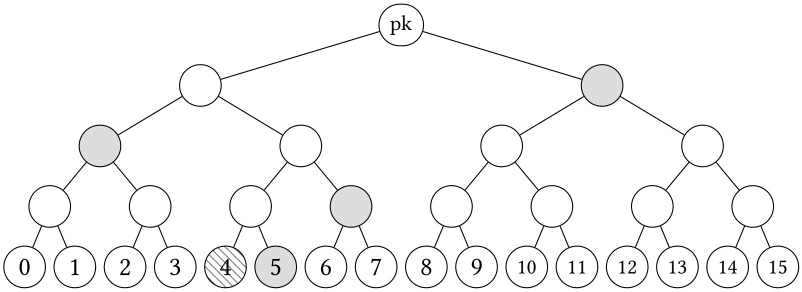


Figure 3.3: A binary hash tree. Here, $p = 16$. The marked nodes are included in the authentication path that authenticates pk_4.

3.2.2 From one to many

Assuming that the verifier has access to the required public key, a one-time signature provides the same functionality that we expect of any digital signature scheme. This assumption hits the sore spot, though: as a key pair can only be used once, each signature verification operation requires a different public key. This is easily remedied by attaching the relevant public key to the one-time signature, only to reveal the real problem: authenticating the attached public keys. In [Mer90], Merkle presents a construction to do precisely this.

For some $p = 2^h$ with $h \in \mathbb{N}$, generate p key pairs of the underlying one time signature scheme, i.e., pairs (pk_i, sk_i) with $i \in \{0, \ldots, 2^h - 1\}$. We then use these to construct a binary tree of height h. For the leaf nodes, simply use one of each of the public keys pk_i. The remainder of the tree can then be constructed from the bottom up: each parent node is computed by applying a hash function $H\colon \{0,1\}^k \times \{0,1\}^k \rightarrow \{0,1\}^k$ to the combination of its child nodes. This continues all the way to the root of the tree, which we label pk. Let us for now define pk to be the public key of the many-time signature scheme, and $\mathrm{sk}_0 = (sk_0, \ldots, sk_{p-1})$ to be the secret key. See Figure 3.3 for an illustration of an instance of such a tree, where $p = 16$. We include algorithmic descriptions that follow this structure as Algorithms 8, 9 and 10 on page 55.

Under the assumption that pk is externally authenticated, the problem of authenticating any of the public keys pk_j can now be solved: attach all of the public keys pk_i to the signature, allowing the verifier to reconstruct the entire tree and compare the root pk. This is, of course, terribly inefficient.

Authentication paths

The naive approach to authentication described above achieves the right goal (i.e., allowing the verifier to reconstruct the root node pk from a signature), but greatly increases the signature size in the process.

As always, [Mer90] provides a solution. Rather than focusing on reconstructing the tree, start at the leaf node corresponding to a given signature, and add only the nodes required to progress upwards towards pk. Given the leaf node that contains the one-time public key pk_j corresponding to a specific signature σ_j, this means we require its sibling leaf-node to construct its parent. From there, we require its parent's sibling to construct its grandparent, et cetera. Again, consider the example in Figure 3.3. Note that precisely one node is required per layer in the tree – in other words, the number of required nodes is logarithmic in the number of leaf nodes, i.e., the potential number of signatures. These nodes make up the 'authentication path.'

To summarize, a signature σ_j on a message m must thus contain the one-time signature on m produced using sk_j, the corresponding public key pk_j, the authentication path, as well as the index j to indicate *which* key pair was used. The latter is necessary for the verifier to correctly interpret the authentication path.

Note that for one-time signature schemes that allow for public-key recovery, such as the WOTS scheme, pk_j can be omitted from the signature; the verifier simply reconstructs it from the one-time signature. As we will see later, this is crucial to achieve practical signature sizes (e.g., in XMSS$^{\mathsf{MT}}$ and SPHINCS, described in Sections 3.3 and 3.5): the uncompressed public keys of the one-time signature schemes described so far are at least as large as their respective signatures.

The state

It was briefly alluded to in the introductory text of this section and in Definition 3.2.1: a many-time signature scheme, such as the Merkle signature scheme described in this section, comes with an important footnote. As each of the OTS key pairs can only be used once, the signer needs to keep track of which key pairs have already been used. The most straight-forward approach to achieve this is by including an index in the secret key, indicating which key pair is to be used next. Thus, a secret key is of the form $\mathrm{sk}_j = (j; sk_0, \ldots, sk_{p-1})$. The distinction between stateful and stateless signature schemes (and their respective downsides and merits) will be discussed in more detail in Sections 3.2.5 and 3.5.1.

Merkle's signature scheme

We describe the resulting signature scheme in Algorithms 8, 9 and 10, parameterized by an abstract one-time signature scheme OTS. As before, h is the height of the Merkle tree, allowing the signer to produce 2^h signatures. We define a subroutine to construct Merkle trees in Algorithm 8b to avoid repetition. Furthermore, we assume some way to retrieve an OTS public key based on the secret key;[6] Algorithm 7b defines this explicitly for the WOTS scheme.

Algorithm 8 Merkle.KeyGen () h, OTS, H

```
for $i \in \{0, \dots, 2^h - 1\}$ do
    $pk_i, sk_i \leftarrow OTS.\mathsf{KeyGen}()$
end for
$node_{i,j} \leftarrow \mathsf{Merkle.BuildTree}(pk_0, \dots, pk_{2^h-1})$   ▷ Let $0 \le i \le h$ and $0 \le j < 2^{h-i}$
$\mathsf{pk} \leftarrow node_{h,0}$
return $\mathsf{pk}, \mathsf{sk}_0 = (0; sk_0, \dots, sk_{2^h-1})$
```

Algorithm 8b Merkle.BuildTree $(\mathrm{leaf}_0, \dots, \mathrm{leaf}_{2^h-1})$ h, H

```
for $j \in \{0, \dots, 2^h - 1\}$ do
    $node_{0,j} \leftarrow \mathrm{leaf}_j$
end for
for $i \in \{1, \dots, h\}$ do
    for $j \in \{0, \dots, 2^{h-i} - 1\}$ do
        $node_{i,j} \leftarrow H(node_{i-1,2j}, node_{i-1,2j+1})$
    end for
end for
return $node_{i,j}$ for $i \in \{0, \dots, h\}, j \in \{0, \dots, 2^{h-i} - 1\}$
```

3.2.3 Treehash

The key generation and signing routines described in Algorithms 8 and 9 are quite expensive to execute, both in terms of computations and in terms of memory usage. Let us first examine memory usage. It is quickly observed that the peak memory requirement occurs when computing the Merkle tree.

---

[6] In general, this can be achieved trivially by including pk as part of sk during key generation.

Algorithm 9 Merkle.Sign (m, sk_{idx}) h, OTS

```
$(idx; sk_0, \dots, sk_{2^h-1}) = \mathrm{sk}_{idx}$
if $idx > 2^h - 1$ then
    return Fail
end if
$\mathrm{sk}_{idx+1} \leftarrow (idx + 1; sk_0, \dots, sk_{2^h-1})$
$\sigma_{OTS} \leftarrow OTS.\mathrm{Sign}(m, sk_{idx})$
for $j \in \{0, \dots, 2^h - 1\}$ do
    $pk_j \leftarrow OTS.\mathrm{RecoverPK}(sk_j)$
end for
$node_{i,j} \leftarrow \mathrm{Merkle.BuildTree}(pk_0, \dots, pk_{2^h-1})$    ▷ Let $0 \le i \le h$ and $0 \le j < 2^{h-i}$
for $i \in \{0, \dots, h - 1\}$ do
    $auth_i \leftarrow node_{i, \lfloor \frac{idx}{2^i} \rfloor \oplus 1}$
end for
$\sigma = (idx, pk_{idx}, \sigma_{OTS}, auth)$
return $\sigma, \mathrm{sk}_{idx+1}$
```

Algorithm 10 Merkle.Verify (m, σ, pk) h, OTS, H

```
$(idx, pk_{idx}, \sigma_{OTS}, auth) = \sigma$
if $\neg\, OTS.\mathrm{Verify}(m, \sigma_{OTS}, pk_{idx})$ then
    return False
end if
$node_0 \leftarrow pk_{idx}$
for $i \in \{0, \dots, h - 1\}$ do
    if $\lfloor \frac{idx}{2^i} \rfloor \bmod 2 = 0$ then
        $node_{i+1} \leftarrow H(node_i, auth_i)$
    else
        $node_{i+1} \leftarrow H(auth_i, node_i)$
    end if
end for
return $node_h \overset{?}{=} \mathrm{pk}$
```

By now the recurring pattern of this section must be clear: a solution is to be found in [Mer90]. In Section 7 of [Mer90], Merkle describes an algorithm that has become known as Treehash. This algorithm has since formed the basis of many procedures that generate Merkle trees; whether all at once to find the root node, as we assumed in the previous subsection, or incrementally, as we will see later.

Rather than building the tree layer by layer, Treehash incrementally considers one leaf node at a time. The core idea is to grow subtrees from the leaves upwards, gradually merging their roots. This allows us to only store the roots of each of these subtrees; all nodes are used exactly once to compute a parent node and are then no longer required (for the purpose of finding the root node).

Let us consider this somewhat more concretely. Initially, when the first leaf node is added, there are no heads of subtrees – the new leaf is considered to be a new subtree of its own. As soon as the second node is added, it can be merged into the initial subtree to compute a node on the second layer. The third leaf forms a subtree of its own again, and the forth leaf node triggers not one but two merges: it can be merged with the third leaf, and again with the head of the tree that had already grown to the second layer. This procedure is naturally continued until all leaf nodes have been added. See Figure 3.4 on the next page for an illustration.

When examining how the set of ‘relevant’ nodes evolves, we observe a strict ordering in when these nodes are used: it is no coincidence that this ordering is defined by their height in the tree, with the lowest nodes being used first. Similarly, it is easily observed that the lowest nodes were the most recently added, and the relevant nodes are consumed in a last-in-first-out manner; the set is really a stack.

As a consequence, note that there can never be two nodes of the same height on the stack – they would necessarily need to be siblings in the tree, and would thus immediately be merged to produce their parent node. This implies that the size of the stack is linear in tree height (or: logarithmic in the number of leaf nodes).

After the algorithm has finished and all leaf nodes have been processed, all that remains on the stack is the root node. This is sufficient to replace the above-described Merkle.BuildTree algorithm (Algorithm 8b) during key generation, but not as part of signing. Here, we also require the authentication path leading from one of the leaf nodes to the root. These nodes are generated during Treehash, but are not preserved by default. We slightly tweak the algorithm to recognize these nodes as they are produced, making use of the fact that the path contains exactly one node on each layer of the tree. See Algorithm 11 for an algorithmic description.

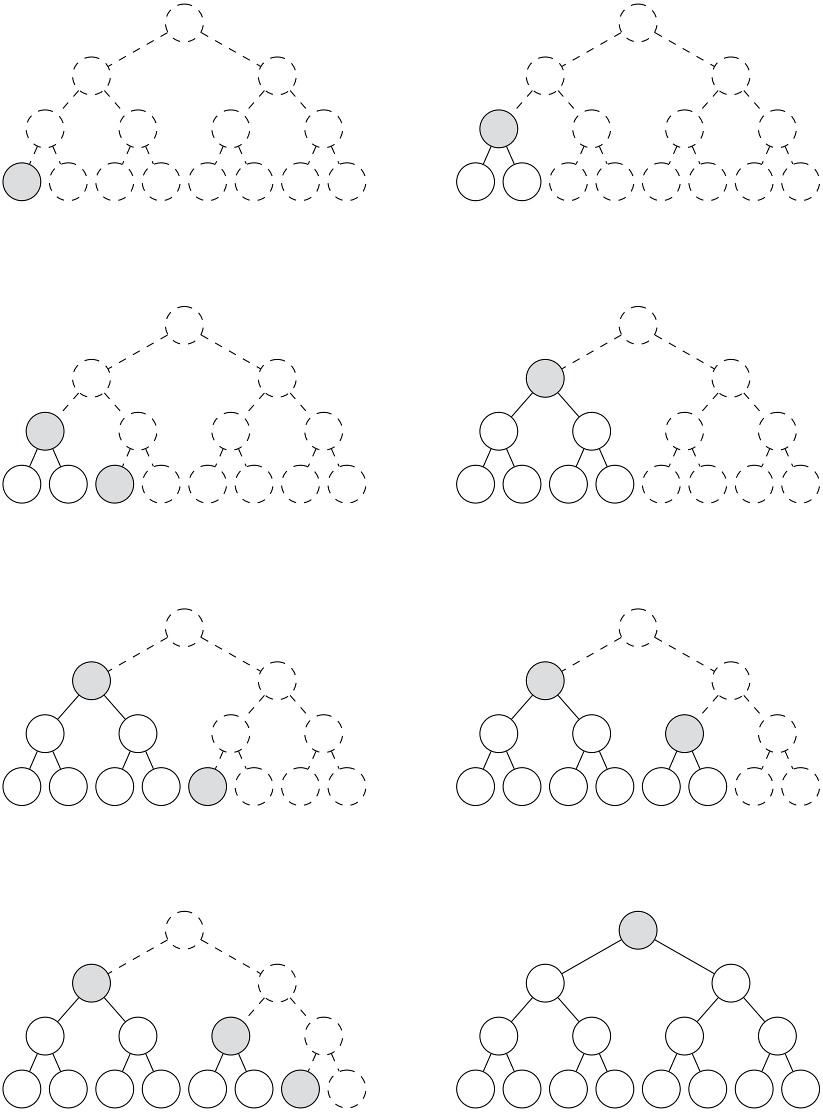


Figure 3.4: Treehash, with $h = 3$. The current roots of subtrees are marked gray. Each round introduces a leaf node, and updates the subtree roots where possible.

Algorithm 11 Merkle.Treehash $(idx, leaf_0, \ldots, leaf_{2^h-1})$ h, H

```
$stack \leftarrow []$
for $i \in \{0, \ldots, 2^h - 1\}$ do
    $node \leftarrow leaf_i$                ▷ Leaves would typically be generated in-place.
    if $idx = i \oplus 1$ then              ▷ If this is the sibling of $idx$..
        $auth_0 \leftarrow node$
    end if
                  ▷ We use the abstract height() to keep track of node heights.
    while height($stack$.head) = height($node$) do
        $sibling \leftarrow stack$.pop()
        if $\lfloor \frac{i}{2^{\mathrm{height}(node)}} \rfloor \bmod 2 = 0$ then
            $node \leftarrow H(node, sibling)$
        else
            $node \leftarrow H(sibling, node)$
        end if
        if $\lfloor \frac{idx}{2^{\mathrm{height}(node)}} \rfloor = \lfloor \frac{i}{2^{\mathrm{height}(node)}} \rfloor \oplus 1$ then
            $auth_{\mathrm{height}(node)} \leftarrow node$
        end if
    end while
    $stack$.push($node$)
end for
$root \leftarrow stack$.head
return $root, path = (auth_0, \ldots, auth_{h-1})$
```

3.2.4 Secret seeds

Up to this point we have defined secret keys to be large sequences of secret values. In Lamport's OTS and in WOTS, we assumed all these values to be sampled during key generation, stored as part of the secret key, and selectively revealed in the signature. The Merkle tree construction implies storing an array of such sequences, and ignored practical limitations: using WOTS, such keys consist of $2^h \cdot \ell \cdot n$ bits.

Rather than storing the entire keys, we can store a seed that allows us to generate the key when it is needed. This reduces the storage space to a k-bit seed, expanded at signing time using a pseudorandom generator (PRG; see Definition 2.1.4). This is first defined explicitly in [BGD+06], but a similar construction was described in [Gol87; Gol04].

The full construction is a bit more subtle. Simply expanding a seed to the full secret key (i.e. the secret keys of all 2^h WOTS instances) would require significant storage and computation, the output of which will mostly be discarded. Instead, we introduce a level of indirection, and use a global secret seed combined with the index i of a specific OTS key pair to derive the OTS-specific seed $seedOTS_i$. This OTS-specific seed can then be expanded as is required for that instance of the OTS. This allows the signer to directly target the OTS key pair that is required, rather than computing the secret keys for all 2^h key pairs.

The careful reader may have observed that this is not entirely the computational optimization it is made out to be. Notably, as part of a Merkle signature, the signer will also have to compute an authentication path. Using the algorithms we have seen so far, this requires computing all leaf nodes, which in turn requires computing all OTS keys. We revisit this in Section 3.2.5 on tree traversal, as well as when discussing the hypertree construction in Section 3.3.2.

Forward security

A related notion that is often mentioned in the same breath as Merkle signatures is forward security. Informally, this expresses the idea that, when a secret key is leaked at a certain moment t, an adversary should not be able to retroactively create signatures that appear to originate before t. In Merkle signatures, the chronology is intuitively captured by the evolving keys containing indices of the next OTS key pair. The signing routine described in Algorithm 9 can be trivially made forward-secure by changing line 5 to $\mathrm{sk}_{idx+1} \leftarrow (idx + 1; sk_{idx+1}, \ldots, sk_{2^h-1})$.

The global seed as introduced before breaks this. This seed remains fixed across signatures, and can be used to derive old secret keys. This is remedied by adding another level of indirection. The seed for sk_0 can be used to derive $seedOTS_0$ as well as a seed for sk_1, which is then used to derive $seedOTS_1$ and a seed for sk_2, et cetera. See [BDH11; BBH13] for more details on forward-secure Merkle signatures.

3.2.5 Tree traversal

After considering memory usage above, we now briefly examine computation time. So far we considered every instance of Merkle.Sign separately; while the secret key was updated to prevent reusing the index, no other state was kept across signatures. For each signature, the vast majority of the computation is spent constructing the same Merkle tree, so that the authentication path can be extracted.

If we stored this Merkle tree as part of the state, every authentication path would be directly available. This has the obvious downside of additional storage requirements — while the previous sections reduced runtime memory usage, this would incur extreme persistent-memory costs.

There is ample space for trade-offs between these two extremes, leading to a class of so-called tree-traversal algorithms. By maintaining a state that contains a carefully selected part of the tree, it is possible to achieve considerable performance improvements at little cost. An important observation here is that the authentication paths of subsequent signatures are often only slightly different: a node needs to change on the second layer for every second signature, on the third layer for every forth signature, on the fourth layer for every eighth, et cetera. A straight-forward optimization is achieved by only computing the changed nodes, but this introduces a large variance in the running time of the signing algorithm. As an example, consider the transition from the rightmost leaf of the left subtree to the leftmost leaf of the right subtree, where the complete authentication path is different. This results in an additional optimization goal (and potential trade-off) for tree-traversal algorithms, as reducing these differences invariably comes at a slight increase in cumulative cost — in terms of both memory and time.

Naturally, specific use-cases lend themselves well for specific algorithms, but an often-cited algorithm is the BDS traversal algorithm described in [BDS08]. Here, the core idea is to maintain several instances of Treehash (see Algorithm 11), each working towards recomputing parts of the tree. During key generation, when the entire tree invariably needs to be computed, the stacks of all of these Treehash instances are primed so that a minimal number of new computations is required. In addition to this, a number of nodes among the top layers of the tree is stored separately, as these are particularly expensive to recompute. This allows for a specific trade-off between state size and performance through a parameter which we will refer to as BDS_k, representing the number of stored layers.[7]

Each Treehash instance is allocated a ration of so-called 'updates' whenever a signature is created, so as to flatten out the runtime differences. These updates represent calls to underlying hash-function primitives, and counting them leads to a fairly accurate performance model. In [BDS08], the authors discuss lower bounds for the required number of updates for certain hypertree structures, guaranteeing that the correct nodes are available when an authentication path is put together.

The authors of [BDS08] remark that general-purpose tree-traversal algorithms typically optimize for the least number of nodes computed; indeed, when moving from one authentication path to the next, that seems like an intuitive heuristic. When considering a Merkle signature tree, this is the wrong metric to consider. It is crucial to observe that computing a parent node required just one call of H — computing a leaf node is equivalent to WOTS key generation, at the cost of hundreds of calls to F.

BDS traversal is implemented as part of the XMSS$^{\mathsf{MT}}$ software described in Section 3.3.5, as well as in the tree traversal scripts included in the software that is part of this work.

3.3 XMSS and XMSS$^{\mathsf{MT}}$

Up to here, we have limited ourselves to the early days of hash-based signatures, focusing on the foundations as laid out in the late 1970s and early 1980s. In this section, we jump ahead to the current state of the art, fast-forwarding past decades of gradual improvement and cherry-picking relevant results along the way.

3.3.1 Collision resilience

Intuitively, the security of the WOTS construction relies on the fact that it is hard to find a preimage for the one-way function F. The same intuition for the function H upholds the security of the Merkle tree construction [NSW05]. Unfortunately,

---

[7] In [BDS08], this parameter is referred to as k. We rename it here to avoid ambiguity.

the security arguments for these constructions do not capture this, and instead require a collision resistant hash function underlying the reduction [Gar05].

This is resolved by [DOT+08] and subsequently [BDE+11; Hül13b], proposing collision-resilient[8] variants of the hash-tree construction and WOTS. Both these works include reductions from finding preimages in the underlying hash function. This is achieved by randomizing the inputs to the hash-function calls, using masks that are included as part of the public key. In the case of WOTS, the resulting scheme is named WOTS$^+$. Note that the exact definition of WOTS$^+$ has slightly evolved throughout literature; collision resilience is the key differentiating characteristic.

In [BDH11], these results are combined to construct the 'eXtended Merkle Signature Scheme': XMSS. XMSS is EU-CMA-secure in the standard model, with a reduction from finding preimages in the underlying hash function. We refrain from giving a concrete specification of XMSS and its functions F and H here, as they are subsequently tweaked slightly in [HRS16b] (discussed in Section 3.3.3).

3.3.2 The hypertree

A recurring consideration with the Merkle tree construction is the fact that the number of potential signatures has to be fixed during key generation. The total number of signatures is a parameter that strongly influences the performance of the signature scheme. In addition to the typical runtime, size and security triad, the Merkle signature scheme introduces a limit to the number of signatures at which point the performance becomes prohibitive; when the sheer number of leaf nodes makes key generation infeasible. In [HRB13], this is mitigated by generalizing XMSS. Rather than scaling up the single tree of XMSS to accommodate more OTS key pairs, the authors construct a hypertree; a tree of XMSS trees.

On the lowest layer, an XMSS tree is used to sign a message as usual, using one of its leaf nodes and providing an authentication path to its root. On the higher layers, the leaf nodes are used to sign the root node of the tree below. An XMSS^{MT} signature thus contains several WOTS signatures, as well as several authentication paths, allowing the verifier to make its way all the way from the bottom to the top of the hypertree. This effectively serves as a certification tree: each tree certifies the root of the tree below, all the way to the public key anchor. See Figure 3.5 on page 65 for an illustration.

---

[8] Precisely the property informally described: to achieve the desired security goal without having to assume the underlying hash function to be collision resistant. See also the discussion in [BDL+11].

Crucially, all leaf nodes of all intermediate trees are deterministically generated WOTS$^+$ public keys that do not depend on any of the trees below it. This means that the complete hypertree is purely virtual: it never needs to be computed in full. During key generation, only the top-most subtree is computed to derive the public key. In the context of XMSS$^{\mathsf{MT}}$, we define the total tree to be of height h and the number of intermediate layers to be d, redefining the height of the subtrees to be h/d. We can now naturally view the single-tree XMSS scheme as the class of specific instances where $d = 1$.

This construction naively reduces the number of leaf-node computations for a signature from 2^h to $d \cdot 2^{h/d}$, but also opens up a range of time-memory trade-offs during tree traversal. In particular, this involves ensuring that the next tree is sufficiently prepared to be ready for use after h/d signatures. In single-tree XMSS, the BDS algorithm can make much stronger assumptions on the initial state, as the key generation routine has all nodes available (and can, e.g., cache nodes high up in the tree). In XMSS$^{\mathsf{MT}}$, the cost of computing the root of the next tree (and the WOTS$^+$ signature authenticating it) has to be spread out over signing operations.

All of this is also accompanied by an increase in the signature size: compared to XMSS trees of height h, the signature now includes $d - 1$ additional WOTS$^+$ signatures. For typical parameters, these make up the vast majority of the signature.

3.3.3 Multi-target attacks

Cryptographic primitives usually do not exist in a single, isolated instance. There may be multiple users relying on the same system parameters, or a system may rely on a composition of multiple keys to ensure its security. In such scenarios, it is important to consider the potential for multi-target attacks.

An example of this that makes its way into many introductory security courses is password hashing; any second-year student will be able to produce a calculation showing the orders of magnitudes between cracking one in many unsalted password hashes, and one of a salted set.[9] Similar problems occur in many-user settings such as TLS connections and software distribution. Solutions often include some form of domain separation or pinning to user-specific public keys [BDL+11; ADP+16], but rarely involve quantified analysis.

---

[9] For completeness: passwords are combined with large, user-specific random values (i.e., salts) before being hashed. This ensures that an adversary performing a preimage search needs to commit to attacking one specific user, as the preimages are effectively domain-separated.

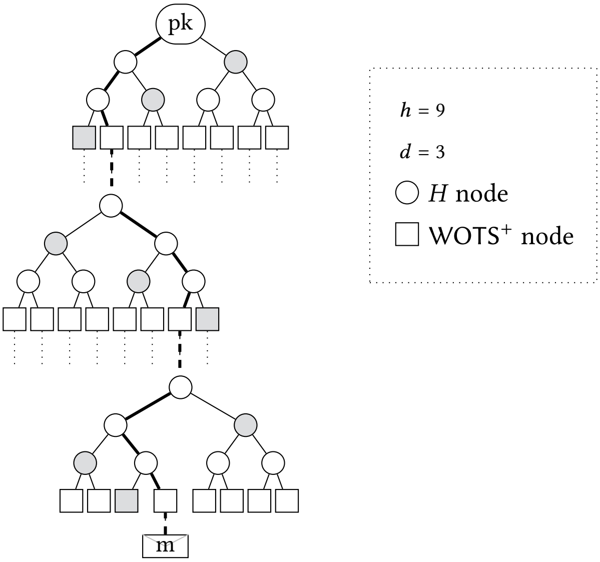


Figure 3.5: An XMSS$^{\text{MT}}$ hypertree. The nodes that make up the authentication path are marked in gray, and dashed lines signify a WOTS$^+$ signature.

In the context of hash-based signatures, this is a much more pressing issue. Rather than multiple instances creating an attack surface in a one-in-many setting, a single XMSS signature consists of a myriad of hash-function calls. Intuitively, given a hash function of output length k that is used q times, and assuming that finding a single preimage compromises security, the attack complexity effectively downgrades from $\mathcal{O}(2^k)$ to $\mathcal{O}(2^k/q)$. There is a probability of $q/2^k$ that a given preimage matches any one of the hashes, instead of the desired $1/2^k$. Where q is typically small in most systems and completely disappears in the asymptotics, it seriously affects practical parameter choices for hash-based signatures; in the largest XMSS$^{\text{MT}}$ parameter set defined in [HBG+18], roughly 2^{66} hash-function calls are involved.

In [HRS16b], we introduce XMSS-T as a multi-target attack resistant variant of the XMSS scheme presented in [BDH11; HRB13]. That work focuses on detailed security analysis, introducing new notions to analyze the common properties of hash functions (see Definition 2.1.2) in the context of multi-target attacks. The resulting single-function multi-target notions behave as one might expect, lowering the attack complexity linearly and by a square-root factor for classic and quantum attacks, respectively. For generic attacks in a multi-function multi-target setting, the complexity remains the same. In this chapter we limit ourselves to the constructive consequences for XMSS (and, in later sections, SPHINCS).

The main difference is the use of independent function keys and bitmasks for every call to a hash function, both inside the hash trees and inside WOTS. $\mathrm{XMSS}^{\mathrm{MT}}$ already moved in this direction, with fixed keys for function families and different, random bitmasks for different tree levels. As the keys and bitmasks are also needed for verification, they must be included as part of the public key. In XMSS-T, they are derived pseudorandomly from an included seed.

Addressing scheme

XMSS-T requires an addressing scheme to distinguish the hash-function calls. Every addressing scheme that assigns a unique address to every call to either F or H within the XMSS-T hypertree can be used. We use a hierarchical addressing scheme that enumerates sub-structures (e.g., a WOTS key pair) inside a larger structure (e.g., a tree). This makes it easy to locally modify parts of addresses while remaining agnostic towards the rest of the hypertree construction. The addressing scheme is public and part of the overall scheme definition: the same addresses are used across XMSS-T key pairs.

Modifying F and H

In the constructions we defined previously, we assumed the existence of the functions $F\colon \{0,1\}^k \rightarrow \{0,1\}^k$ and $H\colon \{0,1\}^k \times \{0,1\}^k \rightarrow \{0,1\}^k$. To differentiate, we supply a function-specific key as an additional k-bit input.[10] We assume a pseudorandom function $\mathcal{F}$ to derive this key based on the address and a public seed. This public seed is fixed for an XMSS-T key pair, and is part of the public key; the public key thus consists of this seed and the root of the XMSS-T hypertree.

Furthermore, before applying F and H, the inputs to these functions are masked. We again use a PRF, $\mathcal{F}'$, with the address and the public seed as inputs; this time to derive the masks. $\mathcal{F}$ and $\mathcal{F}'$ can be the instantiated using the same underlying primitive, as long as they are domain-separated. This can easily be done through a field in the address (see, e.g., the addressing scheme of [HBG+18]).

---

[10] See Sections 3.3.5 and 3.7.3 for examples on how to handle this for concrete instantiations based on SHA-2, SHAKE and Haraka.

For the remainder of this chapter, we assume their modified definitions to include the address as an input. Given the seed pk_{seed} and an address $addr$, an application of the one-way function F within WOTS would thus be

$$F(\mathcal{F}(\mathsf{pk}_{seed}, addr), s_i \oplus \mathcal{F}'(\mathsf{pk}_{seed}, addr)).$$

Here, s_i is a part of the WOTS secret key. Suddenly writing WOTS chains as repeated application of F is not as straight-forward anymore, as the address needs to differ between any two calls. Similarly, when applying H to two sibling nodes:

$$mask_l, mask_r \leftarrow \mathcal{F}'(\mathsf{pk}_{seed}, addr)$$

$$node_{i,j} \leftarrow H(\mathcal{F}(\mathsf{pk}_{seed}, addr), node_{i-1,2j} \oplus mask_l, node_{i-1,2j+1} \oplus mask_r).$$

3.3.4 Comparing XMSS and XMSS-T concretely

There is a non-negligible cost associated with the above-described alterations to F and H. This is offset by the increase in security level, affecting not only the number of function calls but also the input sizes. In order to quantify this difference, we implement XMSS and XMSS-T. We use the BDS traversal algorithm [BDS08] discussed in Section 3.2.5 to make these schemes practical. For parameter sets and the addressing scheme, we follow [HBG+18].

Matching parameters

First, we examine the schemes for two parameter sets — one single-tree and one multi-tree set. We set $h = 20$ and $d = 1$ for the first benchmarks, and use the same subtree height for the second configuration, setting $h = 60$ and $d = 3$. We use $w = 16$ and $n = 256$ in both cases, build on SHA-256 for F and H, and use ChaCha20 to instantiate $\mathcal{F}$. For the BDS trade-off (see Section 3.2.5), we set $BDS_k = 2$.

For XMSS, this leads to a security level of 212 bits classically and 106 bits quantumly for $h = 20$, using the formulas for bit security from [Hül13a]. The $h = 60$ parameter set results in 170 and 85 bits, respectively. Following the security analysis in [HRS16b], these parameters have a security level[11] of at least 256 bits

---

[11] Note that this ignores multi-target attacks against the message digest; a practicality we adjust for in the next subsection when we more carefully match the security levels of XMSS and XMSS-T. This is resolved more structurally in [HBG+18]. In fact, keeping the message digest at 256 bits actually gives XMSS-T a security level of 190 bits classically and 95 quantumly in the context of multi-target attacks across instances. This further skews this initial comparison, as XMSS-T could be downscaled to $n = 190$ to improve its performance without affecting its security level.

	h	d	clock cycles
XMSS	20	1	11 322 614
	60	3	12 547 967
XMSS-T	20	1	33 169 413
	60	3	36 897 222

Table 3.2: Average signing runtime for $n = 256$.

classically, and 128 bits quantumly for XMSS-T. While this comparison does not demonstrate the advantages of using the more secure XMSS-T, it provides some insight into the increase of computation cost. The results for these benchmarks are listed in Table 3.2. We used a single core of the Intel Core i7 described in Section 2.4.1 to carry out these benchmarks, but the implementation was not optimized specifically for this platform.

These results show that the difference in running time between XMSS and XMSS-T for the same parameters is quite significant. This was to be expected, as the running time of the schemes is dominated by applications of F and H – precisely the functions that were changed for XMSS-T. For plain XMSS with the aforementioned parameters, these functions merely consist of calls to SHA-256 with inputs of 256 and 512 bits, respectively. Each of these inputs fits within the internal block size of SHA-256 (512 bits). When considering the Merkle-Damgård construction [Mer79; Dam90] that defines the structure of SHA-256, this implies a single application of the internal compression function. Transforming F and H into keyed hash functions increases this input length. To ensure that the key and the input are in separate blocks, the key is prefixed with 256 zero-bits. This results in inputs of 768 and 1024 bits into SHA-256, respectively, implying the need for two blocks and two applications of the compression function. The straightforward calls to SHA-256 for F and H run in 1 072 and 1 924 cycles, while the keyed variants take 1 932 and 2 812 cycles, respectively. An even bigger factor that weighs down the calls to F and H is the time needed to generate the keys and bitmasks pseudorandomly. Both these values require calls to the pseudorandom generator. An application of F requires two output blocks of 256 bits each – H requires three. At an expense of 560 cycles per output block, generating randomness for the masks and keys carries a significant cost. Altogether, these experiments show that, for the same parameters, XMSS-T causes an increase in the runtime of roughly a factor 3.

Matching security levels

Arguably, the above comparison is not a fair or meaningful one. Rather than running both schemes with the same parameters, we scale up XMSS to match the security level of XMSS-T. For this we selected optimal parameters for XMSS and XMSS-T separately, targeting 256 bits of classical and 128 bits of quantum security. Before, we neglected multi-target attacks against the message digest required to accommodate arbitrary-length messages – here, we parameterize the schemes using $\|m\|$ to represent the digest length.[12]

For XMSS-T, achieving this security level means increasing the message digest length to $\|m\| = 276$ for $h = 20$ and $\|m\| = 316$ for $h = 60$, while keeping $n = 256$. For XMSS, we not only have to increase the message digest size to $\|m\| = 276$ for $h = 20$ and $\|m\| = 316$ for $h = 60$, but also increase n to $n = 300$ for $h = 20$ and $n = 342$ for $h = 60$ [Hül13a]. For $n > 256$ we instantiate F and H using SHA-512, truncating as required. See Table 3.3 for the benchmarks and Table 3.4 for the resulting signature and key sizes.

	$\|m\|$	n	h	d	clock cycles
XMSS	276	300	20	1	17 461 681
	316	342	60	3	22 529 760
XMSS-T	276	256	20	1	35 499 651
	316	256	60	3	44 882 383

Table 3.3: Average signing time for 256 bits classical and 128 bits quantum security.

Here, it turns out that the real increase in runtime for XMSS-T is only about a factor of 2. This seems like a reasonable price to pay for a significant decrease in signature size. For $h = 20$, XMSS-T achieves a size reduction of 18%, and as much as 36% for $h = 60$. The size reduction increases for greater values of d, as more intermediate layers require more WOTS signatures: signing nodes with a lower n implies a smaller ℓ, and thus also smaller WOTS signatures.

---

[12] Note that this length is often confusingly labeled m in the literature, at which point the message (digest) is renamed M, d (creating more confusing of its own), md, *msg* or *dgst*. Throughout this chapter we reserve m for the message or message digest and write $\|m\|$ for its length. Unfortunately, Definition 4.2.1 makes it virtually unjustifiable to establish this convention everywhere.

	$\|m\|$	n	h	d	signature	pk	sk
XMSS	276	300	20	1	$3.5k$	$1.5k$	$2.6k$
	316	342	60	3	$13.7k$	$1.7k$	$21.4k$
XMSS-T	276	256	20	1	$2.9k$	64	$2.2k$
	316	256	60	3	$8.8k$	64	$14.6k$

Table 3.4: Signature and key sizes (in bytes) for 256 bits classical and 128 bits quantum security. The secret key includes the BDS state but not the public key.

3.3.5 RFC 8391

As discussed in the introduction of this chapter, the security of hash-based signatures is comparatively well-understood. While XMSS can be considered a recent development, its foundations date back decades; the progress of the last years has worked towards making it practical for real-world applications. This is a fairly unique combination in the field of post-quantum cryptography, and has led to standardization efforts that precede NIST's standardization project.

Already in late 2015, the IETF published an Internet Draft on what would in May of 2018 become RFC 8391 [HBG+18]: "XMSS: eXtended Merkle Signature Scheme." This document is a so-called *Informational* Request For Comments, and does not formally define a standard that is approved for deployment. Still, it aims to describe WOTS$^+$, XMSS and XMSS^{MT} in a level of detail that allows for unambiguous interoperability between different implementations, and provides a reference implementation. This includes the naive approach that minimizes the state and reconstructs entire trees for each signature, as well as a variant using the BDS traversal algorithm [BDS08].

Previous sections have been littered with forward references to XMSS and XMSS^{MT} as defined by this RFC. XMSS has become somewhat of an overloaded term for various instantiations and variations. With the publication of the RFC, there is a more clear understanding of what the canonical scheme looks like.[13]

The XMSS^{MT} scheme described in RFC 8391 only differs from XMSS-T [HRS16b] in very subtle ways. In particular, the message digest is now also protected against multi-target attacks across users, simplifying both the security analysis and implementation by letting $n = \|m\|$ be equal to the security level.

---

[13] Note that XMSS is used as a primitive in the SPHINCS$^+$ specification in a way that is not fully compatible.

ℓ-trees

Besides Winternitz OTS key pairs in the leaf nodes and binary hash trees on top, there is another building block to XMSS that has so far gone unmentioned. In the note on public-key recovery in Section 3.1.2, we briefly mentioned compression of the public key. While WOTS$^+$ public keys are not part of an XMSS signature (as, indeed, they can be derived from the signature WOTS$^+$), they do need to be compressed before being used as input into H. In XMSS, this is done using so-called[14] ℓ-trees.

An ℓ-tree is an unbalanced binary tree, where the number of leaf nodes is not necessarily a power of two. The function H is applied as usual for each pair of nodes on a layer; when the number of nodes on a layer is odd, it is simply lifted to the layer above it. See Algorithm 12 for an algorithmic description. For ease of exposition, the key and mask construction for H is omitted. Note that in XMSS, all instances of H within an ℓ-tree are individually addressed, keyed and masked.

Algorithm 12 XMSS.$LTree$ ($p_i \in \mathrm{pk}$) ℓ, H

```
for $j \in \{0, \ldots, \ell - 1\}$ do
    $node_j \leftarrow pk_j$
end for
while $\|node\| > 1$ do
    for $j \in \{0, \ldots, \lceil \frac{\|node\|}{2} \rceil - 1\}$ do
        if $2j + 1 \leq \|node\|$ then
            $node_j \leftarrow H(node_{2j}, node_{2j+1})$
        else
            $node_j \leftarrow node_{2j}$
        end if
    end for
end while
return $node_0$
```

In fact, the use of an ℓ-tree in XMSS is not strictly necessary. The reason for using an ℓ-tree is to reduce the required number of masks; historically, these masks were not generated from a seed, but instead stored in the public key [BHH+15].

---

[14] The origin of the term ℓ-tree is unclear, and the construction is often referred to as an L-tree. We use ℓ-tree in this work, stressing the relation to the number of chain heads in a WOTS$^+$ public key.

With the pseudorandomly generated masks introduced in [HRS16b], there is no reason not concatenate, mask and hash all WOTS^+ public keys at once. This is observed in the SPHINCS^+ submission to NIST [BDE+17], unfortunately postdating RFC 8391.

3.4 XMSS^{MT} on the Java Card

Perhaps the first images that come to mind when considering cryptographic software in the real world are large data centers full of servers that terminate TLS, full disk encryption on laptops, or intricate PKI systems. It is easy to forget that most people carry several cryptographic devices in their pockets: smart cards. With estimates of over 10 billion[15] "secure elements" sold globally in 2018 [Eur17], this is undeniably an important market.

Smart cards are often used as authentication token – in an asymmetric-key setting, the card then stores the secret key and uses it to generate signatures. In this section, we examine a practical use case: setting up VPN connections using the popular OpenVPN application, for which we implement XMSS^{MT} [HRB13; HBG+18] (as described in Section 3.3) on the Java Card platform. Our implementation is compatible with XMSS^{MT} as specified in RFC 8391 [HBG+18], and we refer to this document for a byte-level technical specification. It is no coincidence that we chose the stateful XMSS^{MT} scheme; a smart-card implementation can conveniently record the state alongside the key material, hiding the complexity of the statefulness from applications that make use of its API. As before, we defer more elaborate discussion on statefulness to Section 3.5.1. Section 3.4.1 describes the Java Card platform and OpenVPN use case in more detail.

We are not the first to implement hash-based signatures on a smart card. In 2013, Busold, Buchmann, and Hülsing implemented a variant of XMSS on an Infineon-produced smart card [BBH13]; their work even makes on-card key generation practical – something that cannot possibly be said of our implementation. This builds upon earlier work [RED+08] that implements Merkle signatures with BDS traversal (see Section 3.2.5) on an 8-bit AVR, and expands it to the multi-tree scheme that would later evolve into XMSS^{MT}. Crucially, the implementation by Busold, Buchmann, and Hülsing uses low-level access to the underlying hardware, which is not publicly available or portable across manufacturers.

---

[15] Half of these are SIM cards — financial and governmental applications make up most of the remainder.

The results presented here are perhaps somewhat demoralizing. With signatures taking just shy of a minute (and a subsequent preparation step well over a minute and a half), this is impractical for many use cases; see Section 3.4.2 for a more detailed analysis. The main contribution here is clearly not to present speed records, but instead to provide a proof-of-concept and directions on how to improve the situation. Section 3.4.2 discusses our implementation of XMSS^{MT}; the issues we identify carry over into Section 3.4.3, where we provide suggestions for future improvements that could help make hash-based signature schemes more practical on the Java Card platform. The Java Card API has been extended in the past to support new protocols (notably the SAC/PACE protocol used in passports [ICAO14; BSI]), so we can expect future extensions when applications begin to require support for post-quantum cryptography.

3.4.1 Java Card platform and limitations

Java Card defines a standardized, vendor-independent programming platform for multi-application smart cards produced by different manufacturers. While the specification is controlled by Oracle, many of the large smart-card manufacturers[16] collaborate in the 'Java Card Forum' in defining the platform [For18]. The platform has proven popular, with over 20 billion cards sold at the time of its twentieth anniversary in 2016 [Sec17]. Java Card is often found in SIM cards and passports.

As the name suggests, Java Card is based on Java, but many language features have been restricted due to the limited resources. This shows prominently in the limited availability of types – a Java Card platform is only required to support 8-bit `byte` and 16-bit `short` types. Similarly, Java Card inherits the class-based object-oriented style of Java, but using objects is discouraged because of size constraints; moreover, garbage collection is optional for Java Card.

The APIs for Java and Java Card differ vastly. The Java Card API is extremely limited, but does provide a range of high-level methods for standard cryptographic use cases (e.g. signature generation, key storage, block encryption). This enables developers to quickly construct applets to perform basic cryptographic operations. The implementation of the API is left to the smart-card manufacturer, allowing implementations in native code or directly in hardware. This is crucial for performance: the Java Card virtual machine introduces considerable overhead, so

---

[16] At the time of writing, the Java Card Forum consists of Gemalto, Giesecke & Devrient, IDEMIA, Infineon, jNet ThingX, NXP Semiconductors and STMicroelectronics [For18]

implementing cryptographic primitives in Java Card bytecode would be unacceptably slow. Still, considerable overhead remains when calling these API functions, and this turns out to be a recurring theme throughout this work.

An important consideration is the limited amount of memory. Typical Java Cards have in the order of tens of KiB persistent memory (EEPROM or Flash), but the transient (RAM) memory is typically only a few KiB, presenting a serious bottleneck. Memory sizes can vary significantly between cards, so memory requirements should be carefully taken into account when developing applications.

Java Card is compatible with the ISO 7816 standard. This means that communication is done using APDUs (Application Protocol Data Units). These traditionally support a payload of up to 256 bytes, although recent cards support extended-length APDUs to push this limit. For compatibility, we avoid such extended APDUs.

In this work, we focus on compatibility with Java Card version 2.2.2 to 3.0.4.

Considerations for the OpenVPN use case

This work was done as part of a project involving a Java Card applet that was to provide authentication when establishing a VPN connection, tightly integrated into OpenVPN. The projected benefit of this was twofold: increased security and increased usability. Smart cards typically provide much more secure storage of the key material. By selecting the Java Card platform, the cross-platform applet can be easily combined with existing deployed systems. The tight integration with OpenVPN serves to improve the user experience, in particular by simplifying the setup process: we avoid third-party middleware (which would typically be required for the use of more generic solutions, such as hardware tokens relying on standards like PKCS#11 [PKCS11]) and store the configuration files for OpenVPN on the card.

This use case implies a set of assumptions and limitations. There is some margin in terms of signing time, as signing operations are fairly infrequent and users would expect some latency when establishing a connection. More importantly, the required throughput is low: after signing once, typical usage scenarios suggest a period of time during which the card is connected and powered, but not used to produce a new signature. Furthermore, we note that key generation can be done during issuance, and even outside of the card (assuming a secure issuance environment – this is arguably a reasonable assumption, as initialization also involves PINs and network configuration). In principle, there is a nice match between these properties and the XMSS^{MT} signature scheme. There are many time-memory trade-offs that can be flexibly tweaked, and there is ample opportunity for precomputation either during key generation or idle time. However, it is important to reiterate that memory (in particular the fast RAM) is a scarce resource on the card. The next section details these trade-offs.

Parameter selection

XMSS^{MT} offers several parameters that can be adjusted to make various trade-offs. For the remainder of this section, we fix a Winternitz parameter $w = 16$, which implies $\ell_1 = 64$ and $\ell_2 = 3$, and thus $\ell = 67$ chains (see also Table 3.1). Fixing this parameter is important for ease of implementation, as it allows us to make assumptions on the memory usage that would otherwise result in runtime decisions. Similarly, we fix $n = 256$, i.e., 32 bytes, as dictated by the RFC [HBG+18]. We leave the tree height h and number of layers of subtrees d configurable.[17] Crucially, different use-cases will imply different smart-card lifetimes, persistent memory availability and usage intensity, and may benefit from varying requirements on the total numbers of signatures.

3.4.2 Implementation

When designing a smart-card application, it is important to consider natural 'commands' that divide up and structure the computation. For a traditional RSA-2048 or ECC signature, signing a message could be a single command with a single APDU as response. For XMSS^{MT}, signatures are several kilobytes in size and must be spread out over multiple 256-byte response APDUs. This behavior is typical for hash-based signatures on small devices (as we will also see in Section 3.6); they are too large to comfortably fit in RAM but are very sequential in their construction, strongly suggesting an interface where the signature is streamed out incrementally.

There is much repetition of small subroutines to be found in the scheme. After initializing the signing routine by computing a message digest, a signature consists of a sequence of WOTS$^+$ signatures and authentication paths. Internally, the WOTS$^+$ signatures can be decomposed further into their separate chains. The $\ell = 67$ chains split naturally into 8 sets of 8 chains for the $\ell_1 = 64$ message digest chains, and $\ell_2 = 3$ chains for the checksum. For hashes of 32 bytes and $h/d \leq 8$,

---

[17] This is naturally bounded by the storage available on specific cards.

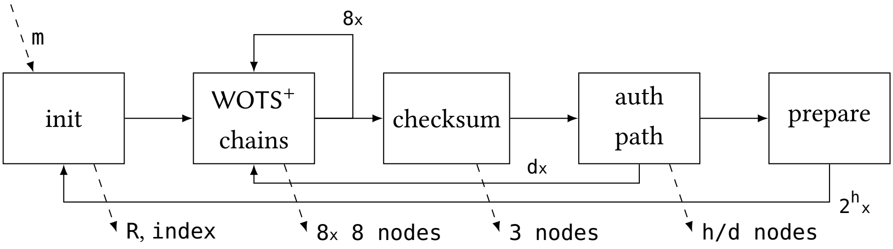


Figure 3.6: State diagram of the signing routine.

authentication paths within a subtree fit into one response APDU, and choosing $h/d > 8$ is not realistic on this platform because of resource constraints.[18] In order to reduce the latency of signature generation, we ensure that all relevant leaf nodes for the authentication path in each subtree on each layer are available in memory. We address this later in this section, and for now only note that maintaining this invariant introduces a preparation step after a signature is produced (and thus: a leaf node is consumed).

Figure 3.6 represents these states visually. Note that each state is triggered by a command APDU, of which only the initial command contains auxiliary data (i.e., the message). These have been omitted for simplicity.

Indices

As it is crucial that the smart card cannot be coerced into re-using a leaf, the first operation should be to increment the state index. Because Java Card does not guarantee a native 32-bit integer type, all indices are stored as tuples of two `shorts`, interpreted as 15-bit unsigned values (effectively ignoring the high bit). As a consequence, atomic increments are not possible without use of expensive transactions, and special care has to be taken in case of overflows – the conservative approach skips 2^{15} leaves in case of card tear,[19] rather than rolling back. Similar considerations apply when deriving indices of ‘next’ and ‘previous’ nodes during state generation. As we have limited h/d previously, internal tree indices can be represented with a single `short`.

---

[18] This would imply either computing or storing hundreds of WOTS$^+$ leaf nodes per tree layer.

[19] The physical attack of interrupting the power supply to the card, e.g., by removing it from the reader.

WOTS$^+$ leaf generation

To generate a WOTS$^+$ leaf, all ℓ chains must be fully computed and an ℓ-tree must be computed over their heads. As memory is limited, the natural choice here is to use the Treehash algorithm, as discussed in Section 3.2.3. Since $\ell = 67$ is not a power of 2, bringing the tree out of balance, there are some special cases to consider. Using Treehash makes it natural to not compute (and, crucially, store) all chains at once. As the value of ℓ is constant for all parameters we account for, this can be simplified by manually handling these special cases after performing Treehash. Altogether, this ensures that we require only 416 bytes of RAM for intermediate results when deriving a WOTS$^+$ public key.

State (re)generation

At least one WOTS$^+$ computation needs to be performed whenever a message is signed: exactly when signing the message digest. Without proper state management, however, one would be required to compute $d \cdot 2^{h/d}$ WOTS$^+$ leaves to derive the authentication paths. Instead, we keep a persistent array of the leaf nodes of the current tree on each of the d layers. If the secret key is generated off-card, the leaf nodes of the first trees can be preloaded; otherwise they can be computed during issuance. Similarly, the $d - 1$ WOTS$^+$ signatures that join the subtrees together can also be precomputed and cached. By keeping an additional array of such nodes and signatures for the 'next' tree on every layer[20] and computing one new node whenever one is consumed, it can be easily seen that we are guaranteed to always have all leaves available before they are consumed. Note that this introduces an imbalance in signing time cost (as consuming indices that introduce new nodes on multiple layers adds linear leaf-generation cost), but that this computation can be performed *after* outputting a signature. As discussed in Section 3.2.5, this variance is negligible compared to the recomputation of leaf nodes. Careful administration is required to guarantee that this is not neglected. Intuitively, one might consider decoupling the signing and preparation step, and allow the signature routine to effectively consume the nodes up to the point at which they were prepared. While this is certainly possible, the involved bookkeeping is more complicated than it may seem at first: memory requirements imply re-using arrays, not all leaves currently

---

[20] This is only required on layers where there is still a 'next' tree to consider, which is trivially not the case for the top-most tree.

in use can be overwritten,[21] and the next layer of leaves needs to be completed precisely when switching to the next subtree. Verifying these conditions combines poorly with the convoluted arithmetic on tuples of `shorts` that represent indices.

Hash functions

Performance is dominated by the cost of a call to the chaining function in WOTS$^+$ and the hash function in the binary trees. In essence, these functions consist of many applications of SHA-256 to small arrays of data (i.e. 32 to 128 bytes) and some XOR operations. This is not a particularly common pattern of operations in traditional cryptography – a signature operation typically requires just one hash-function call to digest the message, often negligible in the overall performance of the signing operation. Note also that there is significant cost associated with a single call to a hash function that is constant in the length of the input, likely representing the overhead of the function call, as shown in Table 3.5.

AES-based hashing

Instead of using a cryptographic hash function as a building block for the described functions, a block cipher can be used to construct a similar primitive using common constructions such as Davies-Meyer [Win84] and Matyas-Meyer-Oseas [MM+85] (the latter being used by [BBH13]). Some care would need to be taken to transform these to a security level equivalent to the second preimage resistance derived from SHA-256 in the context of XMSS$^{\mathrm{MT}}$. This would break compatibility with the RFC [HBG+18], but in principle this is not unsurmountable.

Some Java Cards appear to be equipped with an AES implementation in hardware, speeding up its performance significantly. This is evidenced by an even larger unbalance between constant and variable costs: encrypting larger blocks of data is only slightly more costly than smaller blocks, as shown in Table 3.6. The base cost of a single call to AES is still significant, however, putting the performance in the same ballpark as SHA-256 on short inputs. Note that these numbers cannot be directly compared to the cost of SHA-256 as listed in Table 3.5, as multiple iterations of AES would be required for one compression block.

There is another avenue to explore when relying on AES as a primitive, as the Java Card API supports a range of modes of operation for AES. Combining this

---

[21] Consider that authentication generation requires non-adjacent sets of leaf nodes to remain available.

Table 3.5: 1000 iterations of SHA-256

data (bytes)	-	32	64	128	256	-
runtime (seconds)	-	3.94	5.83	8.02	12.40	-

Table 3.6: 1000 iterations of AES-128 in ECB mode

data (bytes)	16	32	64	128	256	1024
runtime (seconds)	2.97	3.30	3.96	5.30	7.97	23.87

with the fact that we process a large amount of data at once suggests opportunities for parallel data streams; encrypting a large data stream using AES in ECB mode is functionally equivalent to performing independent AES encryption in parallel — *under the same key*. This last restriction is crucial, as a message-dependent block is used as key, ruling out precisely the constructions available to turn AES into a compression function. Other modes of operation suffer a similar faith; there is no clear way to exploit the AES implementation for parallel short-input hashing.

Memory usage and benchmarks

This section outlines the performance when running the applet on a Java Card. For this, we performed measurements and ran tests on NXP-produced JCOP cards, as well as a card of unclear origin (`ICFabricator=0005`). While this is somewhat indicative of relative performance, we note that measurements may vary wildly when comparing different cards by different manufacturers. Tables 3.5 and 3.6 give the individual benchmarks for the symmetric primitives on these cards.

For a WOTS$^+$ signature operation with the parameters described in Section 3.4.1, we measure an average time of approximately 33 seconds. The preparation step requires at least one WOTS$^+$ key generation, which takes approximately a minute.

When we consider a realistic parameter set, where $h = 20$ and $d = 4$, i.e., subtrees with 32 leaf nodes, we notice that the cost of authentication path generation starts to come into play. In particular, the access to nodes stored in persistent memory makes this more costly than a back-of-the-envelope computation would predict.[22] For

---

[22] A WOTS$^+$ signature costs 536 applications of the chaining function on average, as opposed to 63 hash-function calls in the tree.

these parameters, a signature takes roughly 54 seconds in the best case: every 32nd signature adds an additional WOTS$^+$ signature generation, every 256th signature adds two WOTS$^+$ signatures, et cetera. Similarly, preparation takes 85 seconds in the best case. Varying to $d = 5$ results in a slightly shorter signing time, coming in at 50 seconds in the best case (but more frequently requires new WOTS$^+$ signatures).

Besides a small number of bytes to store the keys and index, the requirements on persistent memory follow from the storage of WOTS$^+$ signatures and leaf nodes: $32 \cdot \ell \cdot (d - 1)$ bytes for the WOTS$^+$ signatures, and $32 \cdot (2 \cdot d - 1) \cdot 2^{\frac{h}{d}}$ bytes for the leaf nodes. For $d = 4$, this comes down to $6432 + 7168 = 13600 = 13.28$ KiB. Similarly, for $d = 5$, this adds up to $8576 + 4608 = 13.18$ KiB. Note that increasing d increases signature size because of the additional WOTS$^+$ signatures, but decreasing d while maintaining $h = 20$ sharply increases the memory requirements for node storage as well as the cost of (potentially off-card) key generation.

Considering the signing states described in Section 3.4.2, in particular in Figure 3.6, it can be easily seen that the signature is output in gradual stages as computation progresses. With the WOTS$^+$ chain computation taking up most of the computation, splitting this over eight APDUs levels out communication costs.

3.4.3 Java Card API recommendations and considerations

In the previous subsection, we touched upon several issues with implementing XMSS^{MT} (and hash-based signatures in general) using the current Java Card API (i.e., version 3.0.5 or below). This section discusses potential extensions to improve support for hash-based signatures. In the past the Java Card API has been extended to support new cryptographic algorithms.[23] If and when hash-based signatures become widely used in the future, one would expect extensions of the API for this, either as proprietary extensions of manufactures or ultimately as extensions of the Java Card standard.

An important design choice in such an API is the level of abstraction. One can opt for low-level methods providing more fine-grained (i.e., more primitive) operations, or for higher levels of abstractions, where the API methods provide bigger building blocks, or possibly even a complete signatures scheme.[24] Here we present four alternatives with an increasing level of abstraction.

---

[23] For example, version 3.0.5 introduces support for SAC/PACE [ICAO14; BSI], a protocol used in electronic passports.

[24] For example, in the case of the PACE protocol, the choice has been made not to provide a generic API method for elliptic curve point addition, which would enable applet developers to implement PACE,

Generally speaking, a more fine-grained API is likely to be easier to implement for manufacturers and offers more flexibility to applet developers. On the other hand, higher level, more monolithic API methods make it easier for developers that are less versed in the relevant cryptography to make the correct choices, allow for more performant black-box implementations, and enable manufacturers to provide more comprehensive side-channel countermeasures. Also, an API implementation may need memory to record state between API calls and scratchpad memory to record temporary results. Given that transient memory is extremely scarce, it is not acceptable that API methods need large amounts of RAM.

Another factor to consider when making an abstraction level trade-off is the fact that standardization and industry adoption efforts are still ongoing, and a high-level API leads to less agility to account for future scheme changes.

Parallel hashing

Performance of hash-based signatures is completely dependent on the ability to efficiently compute many hash digests over small amounts of data. While this can be sped up by implementing the hash function in hardware, Section 3.4.2 illustrates that this is only part of the solution. More critically, the execution time depends on being able to exploit the extreme levels of parallelism that are inherent to hash-based signatures. Here parallelism does not necessarily imply parallel execution, but rather independent parallel data streams.

The current interface to hash functions is provided by the `MessageDigest` class. After instantiating an object for a specific digest function, say SHA-256, a user can add additional data by calling the `update(byte[] inBuff, short inOffset, short inLength)` method, and obtain the final digest by calling `doFinal(byte[] inBuff, short inOffset, short inLength, byte[] outBuff, short outOffset)`.

We propose duals of these methods, following the API closely: `updateParallel( byte[] inBuff, short inOffset, short inBlockLength, short noOfBlocks)`, and `doFinalParallel(byte[] inBuff, short inOffset, short inBlockLength, short noOfBlocks, byte[] outBuff, short outOffset)`. Here, `inBuff` provides `noOfBlocks` sequential inputs of `inBlockLength` bytes, and output is written to `outBuff` analogously.

Providing an inconsistent number of inputs (i.e. different `noOfBlocks`) for `update` and `doFinal` calls could be treated as an error but it may be beneficial to

---

but rather to provide more higher-level operations to directly provide PACE as primitive.

instead fix the `noOfBlocks` at the time of construction of the `MessageDigest` object. For hash-based signatures this decision is equivalent, as the relevant hash-function calls all require arguments of the same form. Both options have serious effects on the underlying implementations, as these modifications suggest maintaining a (runtime-determined) number of intermediate hash function states. If this proves to be infeasible, a natural restriction would be to drop the parallel `updateParallel` method.[25] While this reduces flexibility for the applet developer, in particular when memory is constrained and rearranging input is costly, this allows underlying hardware to sequentially process each instance of the hash function without maintaining a variable-length intermediate state in addition to the caller-provided input and output buffers. This does not contradict the goal of achieving a speedup through internal parallelism, as the majority of the cost can be attributed to the Java stack on top of the underlying implementation (see Section 3.4.2, in particular on hash functions and benchmarks). As a result, implementations that would support a parallel `update` method would still likely opt for a sequential underlying hashing primitive to reduce area cost.

Complete WOTS$^+$ chains

Rather than providing a narrow API that allows a developer to efficiently make use of the underlying hash-function primitive, the parallelism can be made transparent to the implementer through a more abstract API: computing WOTS$^+$ chains and authentication paths.

In the WOTS$^+$ chains, there is a lot of opportunity for shared execution. Besides the natural 'horizontal' parallelism across many chains (which would be the primary candidate for optimizations discussed in the previous subsection), there is potential gain in coupling the 'vertical' computations that take place during WOTS$^+$ public key and signature generation. On top of the benefits achieved from only passing through the Java stack once, rather than repeatedly for every application of the chaining function, the input to many of the underlying SHA-256 compression function calls overlaps significantly. In particular, for a single WOTS$^+$ key pair, the input to the first compression call is completely identical across all $16 \cdot 67 = 1072$ calls of the chaining function. Note that this is a consequence of the specific instantiation of the chaining function in XMSS$^{\mathrm{MT}}$ as defined in [HBG+18],

---

[25] It is also possible to reach a similar invariance by fixing `noOfBlocks`, but this still requires multiple hash-function intermediate states in transient memory.

and does not immediately carry over to other function designs (in particular, the SPHINCS$^+$ construction does not benefit from this — see Section 3.7.3).

Such an API goes beyond a straight-forward parameter for the hash function specifying the number of iterations, as the iterated function is not simply SHA-256, but rather the address- and key-aware chaining function. Furthermore, to make it effective for WOTS$^+$ signature generation, it would require specifying the length of each individual chain, as well as a variable number of chains (as an entire WOTS$^+$ signature will likely not fit in RAM on most Java Cards).

This middle ground between abstracting away the parallelism of the hash functions but still requiring (or, indeed, allowing) the developer to puzzle together the pieces has its upsides, but is clearly not without added complexities. We stress that the API of such a hybrid solution needs to be carefully thought through to be sufficiently fine-grained to provide a benefit over an all-in-one API (as described later), yet convenient to use so that it actually reduces boilerplate code and development overhead when compared to a more straight-forward parallel hashing API.

WOTS$^+$ nodes and hash trees

Another unit of abstraction is a hash tree. In XMSS$^{\mathsf{MT}}$, there are two specific instances of hash trees: the tree in XMSS, and the ℓ-tree on top of a WOTS$^+$ key.

The computation of WOTS$^+$ nodes can be hidden behind an API with relative ease. Given its position in the hypertree and the secret seed, the only relevant output is the root node of the ℓ-tree, easily fitting into a single APDU (and thus appropriate as the result of a single function call). We note that in the SPHINCS$^+$ proposal, ℓ-trees have been eliminated altogether. It is not inconceivable that future updates to XMSS will include the same change — see Section 3.3.5.

Abstracting the hash trees in the hypertree behind a single function is somewhat more complicated. The reason for this is twofold. First and foremost, preventing recomputation of such trees is crucial to make XMSS$^{\mathsf{MT}}$ practical, which implies carefully maintaining a state (either by storing leaf nodes, as is done in the current implementation, or through more involved tree traversal techniques). This introduces a time/memory trade-off that strongly depends on the parameter choice; allowing more flexibility in terms of tree height and multi-tree depth significantly increases complexity of the underlying implementation. Secondly, as the relevant output comes in the form of an authentication path of multiple nodes, APDU size (and thus state machine management) becomes relevant as soon as $h > 8$.

Conversely, there is much to gain in terms of simplicity for the user if this is abstracted, as this prevents the users from having to re-implement the Treehash algorithm and make complex state-management decisions. We argue that this is a crucial requirement for non-expert usage.

Complete XMSS^{MT} signatures

At the far end of the spectrum, we consider an API that abstracts away as much of the internals of the scheme as possible. This matches the current approach of the Java Card API for public-key primitives: given a parameterized and keyed object and a message, there is a single API call that produces a signature. An `update` mechanism allows for longer messages, similar to how message digests work. Crucially, this is possible because of the small size of signatures; for standard parameters, a signature easily fits in RAM and even in a single output APDU.

When considering the multiple kilobytes of a typical XMSS^{MT} signature, such an API suggests writing the signature to persistent memory. This requires additional EEPROM or Flash and adds the extra cost of slow memory access. Still, this is likely to compare favorably when considering the potential for performance improvement by implementing the entire scheme natively.

Alternatively, the API could be split up in a similar way as is done in this implementation; we refer to the states described in Figure 3.6 – each state could represent an API call. This would still require the applet developer to implement the state machine, but makes conversion to output APDUs more natural.

Perhaps the most compelling argument for this high-level API is usability for applet developers. XMSS^{MT}, and tree traversal in general, is administratively notoriously tedious, and wrongly managing indices can easily degrade security. In particular, a high-level API is required to properly abstract the state preparation step, as this would otherwise heavily depend on implementation choices (i.e., what part of the state is cached, and how it is enumerated). Ease of use should not be underestimated as a critical factor towards adoption in real-world applications.

Side-channel countermeasures

Smart cards are a common target for physical attacks. To remedy this, manufacturers commonly implement a wide variety of platform-specific countermeasures. An API that abstracts away the usage of secret data is paramount for this to be effective. This requirement aligns well with the considerations of the rest of this section when considering the simplicity of the API exposed to the applet developer: a fine-grained API that requires the developer to implement the overarching scheme creates many potential pitfalls. To illustrate, the current lack of API required us to abuse the `AESKey` object to store sensitive key material in EEPROM, extracting it into RAM before use (although more recent versions of Java Card provide the `SensitiveArray` class for this purpose). Similarly, without API support, the expanded WOTS$^+$ seeds live plainly in transient memory. While in general hash-based signatures have a history of robustness against side-channel attacks, it is precisely this usage of the PRF that has recently been under scrutiny [KGB+18].

3.5 SPHINCS

All Merkle signature constructions we have seen so far have been stateful. In order to guarantee that a one-time signature key pair on the leaf node is not reused, we must, at the very least, maintain the index of the last-used leaf node. Additionally, the state can be used to optimize signature generation using tree-traversal algorithms.

Being stateful also comes with notable downsides. In particular, it greatly complicates key management: sharing keys across devices suddenly becomes a complex synchronization problem, and restoring from backups can easily lead to undesired rollbacks. This contradicts typical API definitions as well as many assumptions implementors and practitioners may make, to the point where Adam Langley famously referred to stateful hash-based signatures as *"a huge foot-cannon."*[26] In fact, such an interface with a changing secret key directly conflicts with the very definition of digital signature schemes (compare Definition 2.1.7 to Definition 3.2.1).

In this section, we review SPHINCS: a hash-based signature scheme [BHH+15] that does not need to maintain state. This scheme demonstrates that, at the cost of some performance, it is possible to make stateless hash-based schemes practical.

3.5.1 Eliminate the state

The fact that one can construct hash-based signatures without maintaining a state is a result from long ago, dating back to work by Goldreich in the eighties [Gol87; Gol04]. The core idea is to create an authentication tree of such depth that, when randomly choosing a leaf node for each signature, the chance of reusing the same

---

[26] `https://www.imperialviolet.org/2013/07/18/hashsig.html`

OTS key pair is negligible. This eliminates the need to keep track of the already-used OTS key pairs.

The obvious problem with this construction is practically creating such a tree in the first place: in order for the probability of collisions to become negligible, the number of leaves has to be so big that it is infeasible to enumerate them. As we have already seen in Section 3.3.2, this can be avoided by using OTS key pairs to link together parts of a hypertree. Goldreich's construction can be seen as an instance of the XMSS$^{\text{MT}}$ hypertree where $h = d$; leaf nodes are used to sign messages, and non-leaves are used to sign the hash of two child nodes. This allows the signer to start from any leaf node, and authenticate it by providing OTS signatures leading to the root of the tree. As with XMSS$^{\text{MT}}$, this requires that all OTS key pairs can be deterministically generated based on their position in the tree.

While Goldreich's system solves the issue of having to maintain a state, it introduces a new problem. As it replaces hashing with signing throughout the tree, it also replaces hash digests with OTS signatures for all nodes included in the authentication path. This creates a new hurdle for practical use, as it results in tremendously large signatures: over a megabyte for real-world parameters.

In 2015, after XMSS$^{\text{MT}}$ had been introduced and early drafts for RFC 8391 started taking shape, Goldreich's construction was revisited in a EUROCRYPT paper by Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe and Wilcox-O'Hearn [BHH+15]. This work introduced SPHINCS: stateless practical hash-based signature schemes. In essence, the construction combines Goldreich's large trees with the hypertree construction of XMSS$^{\text{MT}}$. The presented instance, SPHINCS-256, produces signatures of 41 KiB at a rate of *"hundreds per second"* on a modern Intel desktop processor.

The fact that SPHINCS is stateless prevents optimizations through tree traversal algorithms. As a consequence, a subtree on each layer must be computed from scratch, making the signing routine significantly more costly than that of similar XMSS$^{\text{MT}}$ instances. In terms of XMSS$^{\text{MT}}$ parameters, SPHINCS-256 consists of 12 subtrees, each of depth 5, for a total of 2^{60} leaf nodes.

To further increase the security level, which is now strongly coupled to the probability of repeating a leaf node, SPHINCS does not directly rely on a one-time signature scheme to sign messages. Instead, it relies on a few-time signature scheme (FTS), the public keys of which are authenticated using the one-time key pairs on the leaf nodes.

As the name implies, the security of an FTS does not immediately deteriorate when it is used more than once; it can be used several times before too much of its secret key is revealed. As a consequence, SPHINCS does not require as many leaf nodes – the probability of selecting the same leaf node can be much high before this becomes the defining factor for the security level.

One may wonder why a layer of OTS leaf nodes is included, rather than immediately constructing a Merkle tree on top of the FTS nodes. Indeed, this adds an additional OTS signature to the resulting SPHINCS signature. In practice, reconstructing an FTS public key is much more expensive than an OTS public key. Adding this additional layer of indirection ensures that the signer needs only to compute one FTS leaf node (i.e., the one used to sign the message).

We now briefly review HORST, the FTS proposed in [BHH+15].

3.5.2 HORST

HORST is a variant of the HORS FTS [RR02], adding a Merkle tree on top of the HORS public key to achieve key compression.

Like in WOTS and Lamport's OTS, a HORST secret key consists of an array of random values (potentially generated from a seed). We define $\mathrm{sk} = (s_0, \dots, s_{t-1})$ to consist of t random k-bit values, where t is a power of two. As before, we apply a one-way function F to obtain $p_i = F(s_i)$. The values p_i are then placed on leaf nodes of a Merkle tree, and combined to derive the root node pk.

To create a signature on a message m of $\|m\|$ bits, the message is split into $\frac{\|m\|}{\log(t)}$ components[27] of $\log(t)$ bits. To provide an intuition, consider that SPHINCS-256 uses $\|m\| = 512$ and $t = 2^{16}$. We label these m_j for $j \in \{0, \dots, \frac{\|m\|}{\log(t)}\}$. The bit string representing each component is interpreted as an unsigned integer, and the corresponding value s_{m_j} is revealed. A HORST signature consists of these revealed values s_{m_j}, as well as nodes along the authentication path to pk. Verification works as one might expect: for each message component, p_{m_j} is computed by applying F, and each authentication path is verified to lead to the public key pk.

The algorithmic description of HORST in [BHH+15] introduces a slight optimization that is especially worth mentioning in the context of Section 3.6. As the signature contains multiple authentication paths within the same tree, the

---

[27] Note that the definition of HORST in [BHH+15] is not expressed in terms of a fixed message length m, but rather introduces a parameter k that tweaks the number of revealed components. The message length then equivalently follows as $k \log(t)$.

paths are guaranteed to overlap. The point at which the paths overlap varies per signature, but as the layers of the tree get narrower as we progress towards the top, the number of nodes in a layer is surpassed by the number of authentication paths. At this point, it is more efficient to truncate the paths and fully include this layer, instead. This is expressed by finding a threshold x, signifying the point at which the number of nodes, $\frac{\|m\|}{\log(t)} \cdot (\log(t) - x + 1) + 2^x$, is minimal.

The careful observer will note that the signature will now often include redundant nodes. Some of the nodes on layer x can be derived from the authentication paths, yet layer x is included in full. Arguably the reason for this is to be found in the worst case, where all authentication paths converge onto a single node on layer x; stripping the redundant nodes from layer x would then save only one node. Other than increased code complexity, there appears to be no clear reason not to perform this single-node optimization.

This size reducing optimization is extended upon in Gravity-SPHINCS [AE17; AE18], which introduces PORST. In PORST, the overlapping paths are combined dynamically, resulting in variable-length signatures. While improving the average signature size, this still performs similar to HORST in the worst case. We briefly touch upon this again when discussing the SPHINCS$^+$ submission in Section 3.7, where we introduce another few-time signature scheme: FORS.

See Algorithms 13, 14 and 15 for a description of HORST. Note that, like WOTS, HORST performs verification by public-key recovery. We rely on the Merkle.BuildTree subroutine introduced in Algorithm 8b, setting $h = \log(t)$.

Algorithm 13 HORST.KeyGen () k, t, F, H

```
for $i \in \{0, \ldots, t-1\}$ do
    $s_i \xleftarrow{\$} \{0,1\}^k$
    $p_i \leftarrow F(s_i)$
end for
$node_{i,j} \leftarrow$ Merkle.BuildTree($p_0, \ldots, p_{t-1}$)    ▷ Let $0 \le i \le \log(t)$ and $0 \le j < \frac{t}{2^i}$
pk $\leftarrow node_{\log(t),0}$
return pk, sk $= (s_0, \ldots, s_{t-1})$
```

What makes HORS a few-time signature scheme rather than an OTS is the ratio between the number of leaf nodes t and the message m. As long as the difference is sufficiently large, only a small subset of the secret key is revealed for each signature. Crucially, a forger should not be able to freely choose the messages. To achieve

Algorithm 14 HORST.Sign $(m, s_i \in \mathsf{sk})$ $x, \|m\|, t, F, H$

```
for $i \in \{0, \dots, t-1\}$ do
    $p_i \leftarrow F(s_i)$
end for
$node_{i,j} \leftarrow$ Merkle.BuildTree$(p_0, \dots, p_{t-1})$    ▷ Let $0 \le i \le \log(t)$ and $0 \le j < \frac{t}{2^i}$
$(m_0, \dots, m_{\frac{\|m\|}{\log(t)}-1}) = m$    ▷ Split $m$ into strings of $\log(t)$ bits
for $idx \in \{0, \dots, \frac{\|m\|}{\log(t)} - 1\}$ do
    for $i \in \{0, \dots, \log(t) - x - 1\}$ do
        $auth_{idx,i} \leftarrow node_{i, \lfloor \frac{m_{idx}}{2^i} \rfloor \oplus 1}$
    end for
    $\sigma_{idx} \leftarrow s_{m_{idx}}, auth_{idx}$
end for
$\sigma_x = node_{x,j}$ for $j \in \{0, \dots, 2^x - 1\}$
return $\sigma_{idx}$ for $idx \in \{0, \dots, \frac{\|m\|}{\log(t)} - 1\}, \sigma_x$
```

Algorithm 15 HORST.Verify (m, σ, pk)

```
$\sigma_{idx}$ for $idx \in \{0, \dots, \frac{\|m\|}{\log(t)} - 1\}, \sigma_x \leftarrow \sigma$
$(m_0, \dots, m_{\frac{\|m\|}{\log(t)}-1}) = m$    ▷ Split $m$ into strings of $\log(t)$ bits
$node'_{i,j} \leftarrow$ Merkle.BuildTree$(\sigma_x)$    ▷ Let $0 \le i \le \log(t)$ and $0 \le j < \frac{t}{2^i}$
for $idx \in \{0, \dots, \frac{\|m\|}{\log(t)} - 1\}$ do
    $s_{m_{idx}}, auth_{idx} \leftarrow \sigma_{idx}$
    $node_{idx,0} \leftarrow F(s_{m_{idx}})$
    for $i \in \{0, \dots, \log(t) - x - 1\}$ do
        if $\lfloor \frac{m_{idx}}{2^i} \rfloor \bmod 2 = 0$ then
            $node_{idx,i+1} \leftarrow H(node_{idx,i}, auth_{idx,i})$
        else
            $node_{idx,i+1} \leftarrow H(auth_{idx,i}, node_{idx,i})$
        end if
    end for
    if $node_{idx,\log(t)-x} \ne node'_{0,\lfloor \frac{m_{idx}}{2^{\log(t)-x}} \rfloor}$ then
        return False
    end if
end for
return $node'_{x,0} \overset{?}{=} \mathsf{pk}$    ▷ $\sigma_x$ contains $2^x$ nodes, so $node'_{x,0}$ is the root
```

unforgeability (to the extent possible in the scope of 'few' message queries), it is necessary to first randomize the message and derive an n-bit message digest; this is the 'message' that is signed using HORST. This can be done by including a pseudorandom value R with the message as input to a one-way function, deriving R deterministically from the secret key. The value R then needs to be included in the signature, so that the verifier can recompute the digest from the message. This is covered in more detail in [BHH+15] and briefly discussed in Section 3.7.

3.5.3 High-performance hash functions

To instantiate SPHINCS, one must define the one-way function F and the compressing hash function H, as well as message digest functions we have up to now skimmed over. The SPHINCS-256 instance is designed to target high performance, and is not necessarily restricted to standards. As a consequence, its designers opted to use the ChaCha [Ber08] permutation to underly F and H, and use the BLAKE-512 function [AHM+08] for message compression. The pseudorandom generators used to derive the masks and the WOTS$^+$ and HORST secret keys are also generated using ChaCha (in particular the 12-round ChaCha12) and BLAKE.

The ChaCha permutation, typically denoted π_{ChaCha}, operates on a 512-bit state. With $n = 256$, the input to F needs to be padded. Similarly, H is constructed by padding both input nodes and applying the ChaCha permutation twice. In a construction that is reminiscent of modern sponge-based hash functions, the output is generated by truncating the state to n output bits.

The functions F and H are instantiated as follows. Here, Chop_{256} truncates its input to 256 bits, C is the ASCII string "`expand 32-byte to 64-byte state!`", and O is a string of 256 0-bits.

$$
\begin{aligned}
F(M) &= \mathrm{Chop}_{256}(\pi_{\mathsf{ChaCha}}(M\|C))\\
H(M_1\|M_2) &= \mathrm{Chop}_{256}(\pi_{\mathsf{ChaCha}}(\pi_{\mathsf{ChaCha}}(M_1\|C) \oplus (M_2\|O)))
\end{aligned}
$$

Here, we should note that [BHH+15] predates our work on mitigating multi-target attacks [HRS16b]. As a consequence, SPHINCS uses bitmask constructions as seen in earlier versions of XMSS$^{\mathsf{MT}}$, included as part of the public key rather than pseudorandomly generated, and addressed less granular. We refer to [BHH+15] for exact details.

3.6 ARMed SPHINCS

In the previous section, we briefly considered SPHINCS and the SPHINCS-256 instance. SPHINCS-256 was designed for high performance on desktop or server-grade platforms. This shows in the choice of parameters, but also in the algorithmic choices in HORST. Such platforms typically have access to processors with SIMD instructions (see Section 2.4.1) and an abundance of random-access memory.

It is not obvious that SPHINCS is a feasible solution for small, embedded devices. In this section, we look at SPHINCS-256 on a much more constrained platform: the Cortex-M3. Given the memory usage of the reference implementation and the fact that a SPHINCS-256 signature alone requires 41 KiB, it is not immediately clear that this is feasible on a device with 16 KiB of memory available. We show that while it is possible to construct and verify SPHINCS-256 signatures, performance results indicate that practical applications are limited to non-interactive contexts (such as sensor nodes sending signed data several times a day). To illustrate the cost of eliminating the state, we also implement and benchmark XMSS$^{\mathrm{MT}}$ on the same platform. Rather than implementing the XMSS$^{\mathrm{MT}}$ as described in RFC 8391, we use the same performance-oriented hash functions that underly SPHINCS-256.

Related work

In [RED+08], the potential for hash-based signature schemes on constrained microprocessors was first demonstrated. The authors establish that it is possible, to implement GMSS [BDK+07], an improvement of Merkle's original hash-based signature scheme, on an 8-bit AVR microprocessor at a speed comparable to RSA and ECDSA, although without key generation. The described platform offers 8 KiB of program memory and 4 KiB of SRAM.

As mentioned in Section 3.4, a variant of XMSS was implemented on a 16-bit smart card [BBH13]. The authors show that key generation can be done on the device and get even faster speeds than [RED+08], further demonstrating practicality of (stateful) hash-based signature schemes on constrained devices.

Extensive side-channel analysis of a fast Merkle signature scheme implementation on an AVR ATxmega is presented in [EVMY14]. This paper introduces a new algorithm for the computation of authentication paths in a Merkle tree to significantly reduce (and actually bound) side-channel leakage during this computation.

Other post-quantum schemes also show promising results on embedded systems. In [GLP12], a lattice-based signature scheme is shown to produce signatures of 9 KiB, with keys of 2 KiB and 12 KiB in size, beating RSA in terms of speed, on a Xilinx Spartan-6 FPGA. While memory usage is slightly higher compared to hash-based schemes, [OPG14] shows that lattice-based signatures can be very fast by providing an implementation on a Cortex-M4F. Multivariate-quadratic systems have also been implemented and proven to be practical on low-resource devices as well as ASICs, with keys of practical size [YCC+06].

At the time of publication, the software presented in this section was the first (stateless) signature to target 128 bits of security against quantum attackers on an embedded microcontroller (although it has since been shown that this was not achieved [BHK+19]), complicating comparison to previous results. On the one hand, none of the previous papers targets 128 bits of post-quantum security, and, unlike our software, both [EVMY14] and [BBH13] use hardware accelerators for fast hashing. On the other hand, 8-bit AVR microcontrollers used in [BDK+07] and [EVMY14] and the 16-bit Infineon SLE 78 used in [BBH13] are less powerful (and offer less RAM and ROM) than the more recent Cortex-M3 used in this work. For many applications there is a trend to move from 8-bit and 16-bit microcontrollers towards more powerful 32-bit processors like the Cortex-M; mainly towards the low-end Cortex-M0, which is explicitly advertised to "achieve 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices" [ARMa]. The Cortex-M3 is considerably more powerful than the Cortex-M0 and offers more persistent and volatile memory. Our memory usage suggests that it might be feasible to bring SPHINCS to the Cortex-M0 with 8 KiB of RAM, but this would not leave any space for other applications and would largely be an academic exercise.

Since the publication of the paper underlying this section [HRS16a], several more post-quantum schemes have been implemented on Cortex-M series microprocessors. In particular, many of these results are part of the PQM4 project [KRS+18; KRS+19] — this is briefly discussed in the software listing in Section 1.3.1. At the time of writing, optimized schemes primarily include key-exchange primitives, but also several lattice-based signature schemes.

3.6.1 The Cortex-M3

The Cortex-M3 is a 32-bit microprocessor that implements the ARMv7-M instruction set. The board used in this work is part of the STM32 Discovery line: the STM32L100C. This microcontroller is commonly found in embedded systems used in the automotive industry, small industrial systems and (wireless) sensors. See Section 2.4.2 for more details on the architecture and the STM32 Discovery boards; for this work, the most relevant aspect is the highly constraining 16 KiB of RAM.

We make use of serial communication over USART to communicate with a host device at runtime. In most of the work presented in this thesis communication overhead is irrelevant and not explicitly considered, but the streaming-oriented approach of this implementation makes it a crucial consideration for performance here. This can be done efficiently using the direct-memory-access (DMA) controller to prevent blocking the computation while waiting for the communication interface. Doing this, we are able to communicate reliably at 921 600 Bd.

3.6.2 Implementing SPHINCS-256 on the Cortex-M3

In this section, we describe implementation-specific design choices and present the achieved speed results that come with running SPHINCS-256 on the Cortex-M3. This implementation makes use of code from the SPHINCS reference implementation [BHH+15] as well as (parts of) implementations of BLAKE-256 and BLAKE-512 [AHM+08] and the ChaCha12 stream cipher [Ber08].

While the data structures used in signing and verification may seem similar, the different nature of the performed operations ensures that verification is straight-forward with little memory, while signing is non-trivial. In particular, verification does not require operating on entire subtrees, but rather iterates along authentication paths one layer at a time.

As a general approach to reduce the memory usage of the signing operation, we split the computation into disjunct parts as dictated by the structure of the output. This allows us to focus on and optimize memory requirements of the individual subroutines separately, carrying over minimal memory allocation between them.

Tree storage

The SPHINCS scheme consists of a number of clearly distinct components, with the HORST trees and hash trees as the two most prominent subdivisions. While the memory usage is typically large at the base of a tree, it fans in as we progress towards the root. As each tree is stacked on top of the one below, it is not necessary to ever store more than one tree in memory before proceeding on to the next – this progression is highly sequential.

For the hash trees, the available memory is not an immediate problem. At $32 \cdot 67 = 2144$ bytes, the WOTS$^+$ public key required to produce a leaf node is costly, but can be computed in-place. Such a key can be immediately reduced to its 32-byte root node by constructing an ℓ-tree[28] (as described in Section 3.3.5). After processing all leaf nodes in this fashion, one is left with 32 leaf nodes of 32 bytes each. Each authentication tree contains only $h/d = 5$ layers of hashing, resulting in a total of $2^6 - 1 = 63$ nodes. We simply compute the entire tree and extract the required authentication path.

HORST is a different beast entirely. Given $t = 2^{16}$, the trees contain 131071 nodes spread over 16 layers of hashing, making these trees much higher than the hash trees on top of the WOTS$^+$ leaf nodes. This means that constructing the entire tree and then extracting the authentication paths is not possible. At 32 bytes per node, the nodes alone would require 4 MB of storage. There is no need to store the entire tree, though, as only a very specific set of nodes is relevant for the signature: the nodes along the $\frac{\|m\|}{\log(t)} = 32$ authentication paths, as well as the root node. As we do require the root node to authenticate the tree, there is definitely no escaping having to *compute* the entire tree. Here, we use a slight variant of Treehash as described in Section 3.2.3, and extract all authentication paths in a single pass.

Treehash

In [BHH+15], the authors mention that RAM usage and code size was not one of the concerns when writing the optimized implementation – the implementation was optimized for speed on a platform where memory was available in abundance. They remark that, if saving memory is a concern, the Treehash algorithm could be used. Here, we discuss the specifics of applying it as part of HORST.

The routine as described in Algorithm 11 performs some bookkeeping to recognize nodes that occur in the authentication path. As HORST involves many nodes and multiple authentication paths within the same tree, this quickly becomes cumbersome. Navigating through the tree without actually computing the node values is cheap, allowing us to trace the authentication paths from leaf to root and observe which nodes will need to be included in the signature. Rather than compiling a list of these nodes and performing costly lookups, we can compute and store in which Treehash round they will be produced, as well as their position

---

[28] As this 2144 bytes is not the bottleneck, we did not apply Treehash here, but note that this could have further reduced the memory usage by not requiring all chain heads to be stored at once.

in the signature; as nodes of the various authentication paths will be generated interleaved, it is necessary to rearrange them accordingly.

Consider that the tree consists of $2^{17} - 1 = 131071$ nodes, but only 320 nodes[29] are used. Because of this, only a small subset of all Treehash rounds contains relevant nodes, making it especially important to optimize recognizing these rounds.

Storing a bit mask for each of the relevant rounds allows for an efficient way to recognize which nodes need to be included in the signature. Here, each bit indicates whether a node should be included in the respective authentication path. We sort these bit masks by their round index, so that we can iterate over the mask-index pairs while processing each of the leaf nodes. Pointing an iterator at the current mask-index pair and only incrementing it when the index is equal to the index of the current leaf node will result in an overhead of only one comparison for each non-relevant round.

Streaming out signature data

So far we have glossed over an important aspect of the signing process: putting together the signature. Where an implementation with an abundance of memory available would simply allocate 41 KiB of memory and insert the different pieces of the signature in the right place as they are computed, this is not possible on our device. Instead, the signature is streamed out of the board over the serial port throughout the computation. For many use-cases this is not different from receiving the entire signature all at once after the entire computation has finished, so we believe this should not pose any immediate usability concerns.

As discussed before, a SPHINCS signature primarily[30] consists of a HORST signature, d WOTS$^+$ signatures and d authentication paths. The WOTS$^+$ signatures and authentication paths are computed in the order in which they should occur as part of the signature; instead of storing them in memory, they can be written to the output stream as they are computed.

The HORST signature is a bit more complicated. It consists of $\log(t)$ secret keys belonging to leaf nodes and their respective authentication paths. As remarked in

---

[29] One might expect to require $32 \cdot 16 = 512$ nodes, as each of the 32 authentication paths results in 16 neighboring nodes. However, in order to prevent needless duplication in the top layers, the HORST signature always includes layer 6 in its entirety and truncates the authentication paths after 10 nodes, leaving it to the verifier to reconstruct the paths.

[30] The signature also contains the leaf index and a randomization value, but these can be written to the output stream immediately.

footnote 29 on page 95, all nodes on layer 6 are always included, so the last 6 nodes of these sequences are truncated. The issue here is the fact that the authentication paths are not produced one at a time, but are each grown in an interleaved fashion as more and more of the tree is computed. This does not pose a problem when storing the hash values in a signature in memory – each node value can be inserted in the right place. When streaming the output, however, one cannot go back and insert a node value. Instead, the node values will have to be tagged with what should have been their location in the signature, and rearranged accordingly on the receiving end. For each 32-byte node value, this adds an overhead of two bytes. This results in a communication overhead of 832 bytes (640 for the authentication path nodes, 128 for the nodes on layer 6 and 64 bytes for the secret key values), or 2% on top of the 41 KiB signature.

HORST key material

Similarly, generating a HORST secret key results in too much key material to fit in memory. With 2^{16} leaf nodes of 32 bytes each, this would amount to 2 MiB. Instead, we can once more rely on the fact that Treehash rounds consume the leaf nodes sequentially, and only generate the leaf node values when they are required. In SPHINCS, the HORST secret key is expanded from a seed using ChaCha12. As ChaCha12 is used in counter mode, we do not immediately generate the entire key, but keep track of the counter and perform the next iteration whenever more key data is required. With ChaCha12 producing output blocks of 512 bits, every other leaf node requires a new chunk of output to be generated.

3.6.3 Performance

So far we have focused on the adjustments required to be able to generate SPHINCS signatures with only 16 KiB of volatile memory. Besides memory usage, runtime also remains a defining property to consider for practical feasibility; we provide benchmark results and briefly discuss performance in the next subsections.

ChaCha permutation

In SPHINCS-256, the ChaCha permutation is the fundamental building block for both WOTS$^+$ and HORST, as well as the hashing in the authentication trees. Recall that $t = 2^{16} = 65536$. To generate a HORST key and produce a signature, π_{ChaCha} is called $\frac{1}{2} \cdot t = 32768$ times to expand the seed and generate the secret keys, as the permutation outputs 512 bits and the keys are 256 bits each. These secret keys are then hashed using F to construct the leaf nodes at the cost of another $t = 65536$ permutations. Subsequently, Treehash is used to hash t leaf nodes, at a cost of two ChaCha permutations per execution of H, resulting in another $2 \cdot (t - 1) = 131070$ permutations. This adds up to 229374 permutation calls for one HORST signature.

WOTS$^+$ is significantly cheaper. Recall that $\ell = 67$ and $w = 16$. Generating a WOTS$^+$ key pair requires ℓ secret keys, which costs $\lceil \frac{1}{2} \cdot \ell \rceil = 34$ permutations to expand the seed, $\ell \cdot (w - 1) = 1005$ invocations of F at one permutation each for the chaining function and 66 invocations of H to build the ℓ-tree, totaling $34 + 1005 + 2 \cdot 66 = 1171$ permutations. Given $h = 60$ and $d = 12$, each of the trees in the hypertree has 32 WOTS$^+$ leaves,[31] for a total of 37472 permutations per tree.

Constructing a tree with WOTS$^+$ key pairs on the leaf nodes costs an additional 31 invocations of H. One of the WOTS$^+$ nodes is used to produce a signature on the sub-tree below, at the average cost of $\lceil \frac{1}{2} \cdot \ell \cdot (w - 1) \rceil = 503$ more invocations of F. As there are 12 trees in the hypertree, this accumulates to a total of $12 \cdot (37472 + 2 \cdot 31 + 503) = 456444$. Combining the cost of HORST and the WOTS$^+$ trees, we arrive at a grand total of $229374 + 456444 = 685818$ permutations.

Because we perform so many ChaCha permutations, it is worthwhile to carefully optimize this routine in ARMv7-M assembly. Internally, π_{ChaCha} operates on words of 32 bits each. These fit precisely in the 32-bit registers that are available to us on this platform, and the arithmetic in ChaCha is very simple to perform once the words are accessible. Additionally, many of the arithmetic operations come for free, as the ARMv7-M instruction set provides instructions that take rotated registers as arguments (using the so-called barrel shifter). This enables us to perform nearly all of the rotation operations implicitly. There are not enough registers available for all sixteen 32-bit words, though, as register 13, 14 and 15 are reserved for the stack pointer, link register and program counter, respectively. This would imply that three of the sixteen words need to be saved in memory at all times, at the cost of a load and a store whenever one of these is needed. While we need the program counter and stack pointer for the code to run properly, we are not making any function calls that require the link register. The extra cost of having to pop it from the stack in the end is easily compensated for by the benefit

---

[31] At the cost of some code complexity, one can avoid computing the public key corresponding to the WOTS$^+$ leaf node used to sign; it is not required for any of the nodes in the authentication path.

of an extra general-purpose register, allowing us to keep fourteen of the sixteen words in registers. We can arrange the order of the round internals of the ChaCha permutations such that, on average, we only need to switch out the two words on the stack once every round. Doing so, we arrive at 542 cycles for one permutation; in the context of ChaCha12, this corresponds to around 17 cycles per byte.

Key generation

Generating a SPHINCS-256 key takes 28 205 671 cycles. As was to be expected, virtually all of these cycles can be attributed to WOTS$^+$ key generation. At 32 MHz, this amounts to just below a second. This suggests that key generation is not only feasible but also practical. It should be noted that the STM32L100C Discovery board is not equipped with a random number generator (TRNG). Instead, for benchmarking purposes, we fix a 32-byte seed and write it to persistent memory when flashing the device. As the cost of properly generating this seed is negligible, our results directly carry over to boards that do come equipped with a TRNG.

Signing

Producing a signature takes 589 018 151 cycles, or approximately 18.41 seconds. As described above, we cannot store the signature on the board – this requires communication to a host outside the board. Using the direct-memory-access (DMA) interface, we can efficiently interleave control of this communication with computations. If we disable communication and instead discard the signature as it is being produced, the signing procedure requires 584 384 791 cycles (for messages of small length, so as to focus the benchmark on penalty of signature output). This shows that the communication overhead is noticeable but not significant. In practice, this is a factor that may vary slightly depending on the specific context and interfaces available.

In terms of RAM usage, the signing procedure ends up using 8 755 bytes of stack space. Note that some of this stack usage is the result of function inlining by the compiler. When disabling this behavior, the stack space consumption is reduced to 6 619 bytes. Furthermore, we observe that the current implementation requires 25 KiB of flash memory (or 19 KiB, without inlining). These results show that there is a sufficient amount of memory left on the device (in terms of both RAM and ROM) for other applications, but also indicate that moving to even smaller devices (such as the Cortex-M0) would be quite challenging.

Verification

Verification is much more straight-forward. The memory limit does not necessitate any significant changes like it did for signature generation, as the verification procedure never requires the construction of a full tree. The signature needs to be streamed to the device, but this does not complicate processing, as the node values arrive in the order in which they are to be consumed. At 16 414 251 cycles, verification takes approximately 513 milliseconds. When ignoring the communication by operating on bogus data instead, verification requires 5 991 643 cycles. The communication penalty is in the same ballpark as the one incurred when signing, but still noticeably bigger. This can be accounted for by the way in which communication and computation can be interleaved in the two procedures: for verification, the windows during which communication and computation can be performed in parallel are much smaller, making it more difficult to schedule the communication efficiently.

3.6.4 Comparing to XMSS^{MT}

The Cortex-M3 is typically found in embedded applications, making it a possible candidate for stateful hash-based signatures; the relative simplicity of such deployments allows for a clear understanding of possible interrupts, shared memory usage and key management. One may wonder whether it is worth deploying SPHINCS on such platforms, given the cost of eliminating the state. We now discuss an implementation of XMSS^{MT} on the STM32L100C and compare its performance. We parameterize XMSS^{MT} such that it provides a comparable security level using similar primitives. Note that, like SPHINCS [BHH+15], this work predates the advances in resistance against multi-target attacks discussed in Section 3.3.3.

Parameters

For the parameters selection, we tried to conform to the settings proposed in RFC 8391 (at the time only just submitted as an Internet Draft [HBG+15]). This lead to the choice of $\|m\| = 256$ and $n = 256$ bits for the function output sizes, a tree with a total height of $h = 20$, $d = 2$ subtree layers and a Winternitz parameter $w = 16$ (resulting in a length of $\ell = 67$).

In terms of running time, the performance would have benefited significantly from a larger number of subtree layers d. However, each layer implies the need to store an additional WOTS$^+$ signature, quickly exceeding our memory constraint. Moreover, a signature contains one WOTS$^+$ signature per layer, increasing the signature size significantly. For the BDS algorithm (see Section 3.2.5), we set $BDS_k = 6$. This allows caching of a large number of expensive nodes in the limited memory that is available.

In order to be able to fairly compare XMSS$^{\mathsf{MT}}$ to SPHINCS-256, we do not use SHA-256 and SHA-512 to compute the message digest or the parent nodes in the hash trees. Instead, we rely on the BLAKE hash functions [AHM+08] for the message digest, and use a construction based on the ChaCha permutation similar to the ones described in Section 3.5.3 for the functions H and F. For pseudorandom number generation, we replace ChaCha20 with ChaCha12, matching the choice in SPHINCS-256. All of this implies that we can use the same ARMv7-M assembly implementation of the ChaCha permutation that we used for SPHINCS.

Performance

The difficulty with an accurate performance estimate for XMSS$^{\mathsf{MT}}$ is that it highly depends on the practicalities of the platform it is deployed on, as well as the precise use-case. This is a result of the extra administration that comes with dealing with the state. Part of the state is crucial for the security of the scheme (namely the index of the last processed leaf node), while the structures that need to be stored for BDS traversal are needed for signing time optimization purposes. Writing persistent data is a relatively costly operation on most platforms, so different decisions will need to be made depending on use case specific requirements. On the STM32L100C, writing a well-aligned 4-byte word to non-volatile memory costs roughly 216 500 cycles on average, and scales linearly with the number of words written.

For our experiments, we assume that the device is powered on for a longer period of time, and is being queried for multiple signatures over this interval. This is an especially interesting scenario for XMSS$^{\mathsf{MT}}$, as this is where the benefit of the BDS state comes into play most prominently.

Before outputting each new signature, it is necessary to write the updated secret key to persistent memory. This prevents re-use of a leaf node (and thus compromise of the key) when the power gets cut. As the BDS state is much larger and thus more expensive to store, it is only written to persistent memory when a graceful power-off occurs. In case this state is lost, it can be reinitialized based on the secret key seed and leaf node index. Note that this is a costly operation, roughly equivalent to a complete key generation run.

Compared to SPHINCS, the key generation phase for XMSS^{MT} is much more expensive, especially in the setting described here. The main reason for this is the fact that the two trees consist of 10 levels each, resulting in the computation of 2048 WOTS$^+$ leaves (1024 on each level). The generation of two such trees is necessary to initialize the BDS state. Additionally, a WOTS$^+$ signature needs to be computed and stored to link the trees. For the specified parameters, the initialization phase takes 8 857 708 189 cycles. Each WOTS$^+$ leaf computation costs 4 299 598 cycles, and the WOTS$^+$ signature costs 1 079 936 cycles. As expected, the WOTS$^+$ operations account for most of the work, leaving only a small fraction for the hash trees.

For signing, the cycle count is not precisely identical for each signature. The BDS algorithm tries to distribute costs equally among signature generations by running a fixed amount of Treehash 'updates' for each signature (see Section 3.2.5). For the first few signatures, not all these updates are needed as all structures are initialized during key generation and only few values have to be computed during each signature generation. Similarly, not all updates can be distributed optimally without incurring costly memory operations. It turns out that during this 'start-up phase' it is slightly more costly to update the state for 'right' leaf nodes than for their 'left' neighbors, and signatures using a left leaf node come in at 21 551 730 cycles, while right nodes cost 17 308 759 cycles. For our implementation, transitioning from one tree to the next does cost significantly more cycles than a regular signature: signatures that require renewing the WOTS$^+$ signature that binds the subtrees together cost 28 344 774 cycles. Overall, the average signing time is 19 441 021 cycles.

As one would expect of a hash-based signature scheme, verification remains a much cheaper operation. At 4 961 447 cycles, the relative gain compared to SPHINCS is not as dramatic as it is for signing, but the difference is significant.

3.7 SPHINCS$^+$

Even though the introduction of SPHINCS made stateless hash-based signatures much more practical, there is still ample room for improvement. In this section, we describe steps in that direction by introducing its direct successor: SPHINCS$^+$. Improving both in terms of signature size and runtime performance, we describe not one specific signature scheme instance but a framework that allows for a variety of trade-offs. This allows users to make highly application-specific trade-offs with regards to the signature size, the signing speed, the required number of signatures and the desired security level, and even account for platform considerations such as memory limits or hardware support for specific hash function.

SPHINCS$^+$ was originally described in the submission to NIST's Post-Quantum Cryptography Standardization project in November of 2017 [BDE+17]. It has since progressed to the second round, where it saw minor tweaks to further improve performance. In [BHK+19], we address SPHINCS$^+$ from a more academic viewpoint, providing a more thorough security analysis and comparison to instances of SPHINCS [BHH+15], Gravity-SPHINCS [AE17; AE18] and Picnic [CDG+19].

As the general structure of SPHINCS$^+$ closely resembles SPHINCS and XMSS^{MT}, we refrain from describing the full proposal here, and focus on aspects that differ from its predecessors. We construct a hypertree using Winternitz' one-time signature scheme, and, like SPHINCS, use a few-time signature scheme at the leaves. Instead of committing to a specific hash-function construction, however, we define the framework in terms of *tweakable* hash functions. We elaborate on this below.

Among the main distinguishing contributions of SPHINCS$^+$ is the introduction of a new few-time signature scheme: FORS, introduced in Subsection 3.7.2. Another important change from SPHINCS to SPHINCS$^+$ is the way leaf nodes are chosen. SPHINCS$^+$ uses publicly verifiable index selection, preventing an attacker from freely selecting a seemingly random index and combining it with a message of their choice. These two changes together make it harder to attack SPHINCS$^+$ via the few-time signature scheme and thus allow us to choose smaller parameters. With the same goal, we apply multi-target attack mitigation techniques as proposed in [HRS16b], making it harder to attack SPHINCS$^+$ using a (second-)preimage attack. We give a security reduction that formally shows these claims in [BHK+19].

In the remainder of this section, we describe the framework and the selected instances as defined in the second-round revision of the NIST submission. We omit minute changes, such as the revised WOTS$^+$ key compression. Refer to the specification document for a byte-accurate description [BDE+17].

3.7.1 Tweakable hash functions

As introduced in Section 3.3.3, there has been a line of work [DOT+08; BDE+11; BDH11; Hül13b; HRS16b] focusing on reducing the assumptions necessary to prove the security of hash-based signature schemes. In some constructions inputs to hash functions are masked with random values, while in others, functions are defined to take a separate key as input. Some proposals do both, and some simply prefix or append additional data to the inputs before hashing. Overall, though, the high-level structures remain the same. Although the differences in schemes are somewhat local, each work redid a full security analysis of the whole signature scheme. While these security analyses were already complex for stateful hash-based signature schemes, the case of stateless schemes adds further complexity.

We now introduce *tweakable* hash functions as an intermediate abstraction. Tweakable hash functions allow us to unify the general description of hash-based signature schemes, abstracting away the details of how exactly hashing is done. and allowing for a separation of the analysis of the high level construction. In [BHK+19], we show that this captures typical constructions from the literature, and use it to construct the security reduction of SPHINCS$^+$.

In addition to the message input, a *tweakable* hash function takes public parameters P and context information in the form of a tweak T. The public parameters can be thought of as a function key or index. The tweak can be seen as a nonce.

Definition 3.7.1 (Tweakable hash function) *Let n be the security parameter,*[32] *$\alpha \in \mathbb{N}$, $\mathcal{P}$ the public parameters space of size exponential in n, and $\mathcal{T}$ the tweak space. A tweakable hash function is an efficient function Th_α,*

$$Th_\alpha : \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha \to \{0,1\}^n$$
$$\mathsf{md} \leftarrow Th_\alpha(P, T, m)$$

mapping an α-bit message m to an n-bit hash value md *using public parameters $P \in \mathcal{P}$ and a tweak $T \in \mathcal{T}$.*

In the remainder of this section we write n for the length of digests in bits, which, by the above construction, is equal to the security parameter. For convenience and consistency, we use F as a shorthand for Th_n and H for Th_{2n}.

In SPHINCS$^+$, the public parameter is a public seed pk_{seed} of n bits that is part of the public key. As tweak, we use the 256-bit address *addr* that identifies the position of the hash function call within the hypertree. This combination makes all hash-function calls across every SPHINCS$^+$ key pair and position in the hypertree

---

[32] We use n rather than k here, both for consistency with literature and because k is used in the definition of FORS in [BHK+19]. In previous sections it was important to distinguish between the digest length and the security parameter, but the introduction of tweakable hash functions obviates this complication.

fully independent. We defer the exact requirements on the security properties of tweakable hash functions and their instances to [BHK+19].

3.7.2 FORS

As the few-time signature scheme in SPHINCS$^+$, we define FORS [fɔːrs]: Forest of Random Subsets. This scheme serves as a drop-in replacement for HORST [BHH+15] (see Section 3.5.2). In [BHK+19], we strengthen the notion of target subset resilience as previously used to analyze HORS and HORST. The design of FORS closely follows from this notion, but can be described independently.

FORS is defined in terms of a power-of-two integer t, defining the size of a single binary tree, and an integer κ, defining the size of the forest, i.e., the number of binary trees. These parameters dictate the length of the message that can be signed: given κ and t, a FORS key pair can be used to sign strings of $\kappa \log t$ bits. In the context of SPHINCS$^+$, this means we must ensure that the message digest to be signed at the bottom of the hypertree is at least of this length.

The FORS key pair

The FORS secret key consists of κt random n-bit values, grouped together into κ sets of t values each. In SPHINCS$^+$, these values are deterministically derived from a seed together with the address of the key in the hypertree. We label these values $\mathrm{sk} = (s_0^{(0)}, \ldots, s_{t-1}^{(\kappa-1)})$. To construct the FORS public key, we first construct κ binary hash trees on top of the sets of secret-key elements. As in HORST, we hash the secret-key elements to obtain $(p_0^{(0)}, \ldots, p_{t-1}^{(\kappa-1)})$, this time using a tweakable hash function. Each of the t resulting values in a set is used as a leaf node, resulting in κ trees of height $\log t$. We use H to construct the hash tree as before, addressed using its unique position within the FORS trees and the location of the FORS key pair in the hypertree. We then compress the root nodes using a call to a tweakable hash function with input length κn. The resulting n-bit value is the FORS public key. See Algorithm 16. Note that we use Merkle.Treehash described in Algorithm 11 as a subroutine with $h = \log t$, implicitly tweaking H with the address of each call.

FORS signatures

Given a message of $\kappa \log t$ bits, we extract κ strings of $\log t$ bits. Each of these bit strings is interpreted as the index of a single leaf node in each of the κ FORS trees.

Algorithm 16 FORS.KeyGen () $\kappa, n, t, P, T, F, H, Th_{\kappa n}$

```
for $i \in \{0, \ldots, \kappa - 1\}$ do
    for $j \in \{0, \ldots, t - 1\}$ do
        $s_j^{(i)} \xleftarrow{\$} \{0, 1\}^n$
        $p_j^{(i)} \leftarrow F(s_j^{(i)})$
    end for
    $r_i, path^{(i)} \leftarrow$ Merkle.Treehash$(0, p_0^{(i)}, \ldots, p_{t-1}^{(i)})$
end for
pk $\leftarrow Th_{\kappa n}(P, T, r_0 \| \ldots \| r_{\kappa-1})$
sk $= s_j^{(i)}$ for $i \in \{0, \ldots, \kappa - 1\}, j \in \{0, \ldots, t - 1\}$
return pk, sk
```

The signature consists of each of these nodes and their respective authentication paths. Signature generation is illustrated in Figure 3.7; see Algorithm 17 for pseudocode. As before, in Treehash, assume H implicitly tweaked and that $h = \log t$.

To verify the signature, one reconstructs the public key and confirms that it corresponds to the given public key. The verifier reconstructs each of the root nodes using the authentication paths; combining these leads to the candidate public key. See Algorithm 18. Like in SPHINCS, a FORS signature is never verified explicitly as part of SPHINCS$^+$. Instead, the resulting public key is used as a message on the next layer of the hypertree, to be authenticated using a WOTS$^+$ signature. As with WOTS$^+$ and HORST, the FORS verification algorithm described below is trivially turned into a RecoverPK routine that instead outputs the public key.

Algorithm 17 FORS.Sign $\left(m, s_j^{(i)} \in \text{sk}\right)$ κ, t, F, H

```
$(m_0, \ldots, m_{\kappa-1}) = m$                ▷ Split $m$ into strings of log$(t)$ bits
for $i \in \{0, \ldots, \kappa - 1\}$ do
    for $j \in \{0, \ldots, t - 1\}$ do
        $p_j^{(i)} \leftarrow F(s_j^{(i)})$
    end for
    $r_i, path^{(i)} \leftarrow$ Merkle.Treehash$(m_i, p_0^{(i)}, \ldots, p_{t-1}^{(i)})$
    $\sigma_i = s_{m_i}^{(i)}, path^{(i)}$
end for
return $\sigma_i$ for $i \in \{0, \ldots, \kappa - 1\}$
```

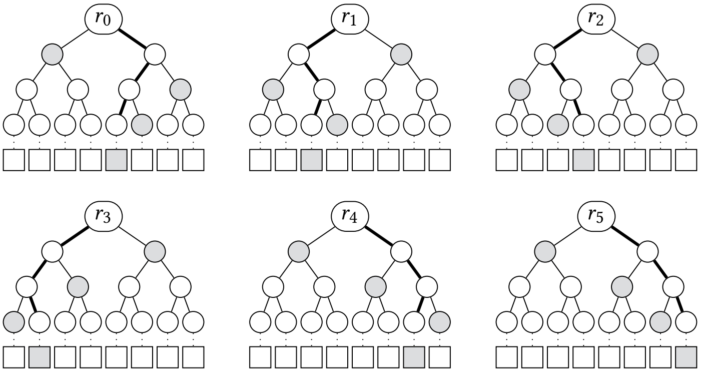


Figure 3.7: An illustration of a FORS signature with $\kappa = 6$ and $t = 2^3$, for the message `100 010 011 001 110 111`. We get $\text{pk} \leftarrow Th_{\kappa n}(\text{pk}_{seed}, addr, r_0 \| r_1 \| r_2 \| r_3 \| r_4 \| r_5)$.

3.7.3 Instances

As SPHINCS$^+$ is a versatile framework that allows for many trade-offs and choices, we define several concrete instances. These instances define parameters for FORS, WOTS$^+$ and the general hypertree structure, but also instantiate the tweakable hash functions using concrete hash-function primitives.

We select the hypertree parameters h and d, the FORS parameters t and κ, and the Winternitz parameter w by fixing the maximum number of signatures and the target security level, and then automatically searching through a large space of possible parameter sets. For each of the security categories 1, 3 and 5 as defined by NIST (see Section 2.2), we select one parameter set that targets *fast signing* and one parameter set that targets *small signatures.* The Sage script we used for parameter selection is included as an appendix to [BHK+19]. See Table 3.7 for an overview of these parameter sets.

Tweakable hash functions

Finally, we propose a total of 6 different instantiations of the tweakable hash functions, based on two constructions, subsequently instantiated with one of three underlying hash functions: SHA-256 [NIST15a], SHAKE256 [NIST15b], and Haraka [KLM+17]. Note that the instantiations using Haraka cannot reach the same security levels that can be reached with SHA-256 or SHAKE256. This is due

Algorithm 18 FORS.Verify $(m, \sigma_i \in \sigma, \mathrm{pk})$ $\kappa, n, t, P, T, F, H, Th_{\kappa n}$

```
$(m_0, \ldots, m_{\kappa-1}) = m$    ▷ Split $m$ into strings of $\log(t)$ bits
for $i \in \{0, \ldots, \kappa - 1\}$ do
    $(s^{(i)}_{m_i}, path^{(i)}) = \sigma_i$
    $(auth^{(i)}_0, \ldots, auth^{(i)}_{t-1}) = path^{(i)}$
    $node^{(i)}_0 \leftarrow F(s^{(i)}_{m_i})$
    for $j \in \{0, \ldots, t - 1\}$ do
        if $\lfloor \frac{m_i}{2^j} \rfloor \mod 2 = 0$ then
            $node^{(i)}_{j+1} \leftarrow H(node^{(i)}_j \| auth^{(i)}_j)$    ▷ Assume $H$ is implicitly tweaked
        else
            $node^{(i)}_{j+1} \leftarrow H(auth^{(i)}_j \| node^{(i)}_j)$    ▷ Assume $H$ is implicitly tweaked
        end if
    end for
    $r_i = node^{(i)}_{\log(t)}$
end for
$\mathrm{pk}' \leftarrow Th_{\kappa n}(P, T, r_0 \| \ldots \| r_{\kappa-1})$
return $\mathrm{pk}' \overset{?}{=} \mathrm{pk}$
```

Table 3.7: Parameter sets for SPHINCS$^+$ targeting different security levels and different trade-offs between size and speed.

	n	h	d	$\log t$	κ	w	bitsec	NIST	$\lvert sig \rvert$
SPHINCS$^+$-128s	128	64	8	15	10	16	133	1	8 080
SPHINCS$^+$-128f	128	60	20	9	30	16	128	1	16 976
SPHINCS$^+$-192s	192	64	8	16	14	16	196	3	17 064
SPHINCS$^+$-192f	192	66	22	8	33	16	194	3	35 664
SPHINCS$^+$-256s	256	64	8	14	22	16	255	5	29 792
SPHINCS$^+$-256f	256	68	17	10	30	16	254	5	49 216

to a generic meet-in-the-middle attack computing collisions in the internal state, which has (classical) complexity 2^{128}.

For the *robust* instances, we first generate pseudorandom bitmasks which are then applied to the input message. This results in the following construction, closely following [HRS16b]. Given two hash functions $H_1\colon \{0,1\}^{2n} \times \{0,1\}^{\alpha} \to \{0,1\}^n$ with $2n$-bit keys, and $H_2 : \{0,1\}^{2n} \to \{0,1\}^{\alpha}$, we define

$$\begin{aligned} \mathcal{P} = \mathcal{T} &= \{0,1\}^n, \\ Th_\alpha(P,T,m) &= H_1(P\|T, m^{\oplus})\,, \text{where} \\ m^{\oplus} &= m \oplus H_2(P\|T)\,. \end{aligned}$$

For the *simple* instances, we take an approach inspired by the LMS proposal for stateful hash-based signatures [CMF19], and omit the bitmasks. Given a hash function $H : \{0,1\}^{2n+\alpha} \to \{0,1\}^n$, we define

$$\begin{aligned} \mathcal{P} = \mathcal{T} &= \{0,1\}^n\,, \\ Th_\alpha(P,T,m) &= H(P\|T\|m)\,. \end{aligned}$$

Naturally, the ‘simple’ instances are faster, as they do not require the generation of bitmasks. When combined with compressed addresses in the SHA-256 case (see below) this can lead to an estimated reduction of the number of compression function calls by a factor of almost 4. This comes at the cost of a security argument that relies on the random oracle model.

We now explicitly define F and H for the above-mentioned variants. We also show how to define Th_α for generic lengths, e.g. for WOTS$^+$ or FORS key compression. We defer defining functions specific to SPHINCS$^+$ (such as a PRF and the message hashing function) to the formal specification document. In general, if a parameter set requires an output length $n < 256$ bits for F or H, we take the first n bits of the output and discard the remaining.

SPHINCS$^+$-SHAKE256

For the robust variant of SPHINCS$^+$-SHAKE256, we define

$$\begin{aligned} F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed}\|addr\|m_1^{\oplus}, n), \\ H(\mathsf{pk}_{seed}, addr, m_1\|m_2) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed}\|addr\|m_1^{\oplus}\|m_2^{\oplus}, n), \\ Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed}\|addr\|m^{\oplus}, n). \end{aligned}$$

For the simple variant, we instead define

$$\begin{aligned}
F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed} \| addr \| m_1, n),\\
H(\mathsf{pk}_{seed}, addr, m_1 \| m_2) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed} \| addr \| m_1 \| m_2, n),\\
Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{SHAKE256}(\mathsf{pk}_{seed} \| addr \| m, n).
\end{aligned}$$

SHAKE256 can be used as an extendable output function, which allows us to generate the bitmasks for arbitrary length messages directly. For a message m with l bits we compute

$$m^{\oplus} = m \oplus \mathsf{SHAKE256}(\mathsf{pk}_{seed} \| addr, l).$$

SPHINCS$^+$-SHA-256

In a similar way we define the functions for SPHINCS$^+$-SHA-256. For the robust variant, we further define the tweakable hash functions as

$$\begin{aligned}
F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m_1^{\oplus}),\\
H(\mathsf{pk}_{seed}, addr, m_1 \| m_2) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m_1^{\oplus} \| m_2^{\oplus}),\\
Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m^{\oplus}).
\end{aligned}$$

For the simple variant, we instead define the tweakable hash functions as

$$\begin{aligned}
F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m_1),\\
H(\mathsf{pk}_{seed}, addr, m_1 \| m_2) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m_1 \| m_2),\\
Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{SHA\text{-}256}(\mathsf{pk}_{seed} \| 0^{64-n/8} \| addr^c \| m).
\end{aligned}$$

SHA-256 can be turned into an extendable output function using MGF1, which allows us to generate the bitmasks for arbitrary length messages directly. Here, we use MGF1 as defined in [KS98a]. Note that MGF1 takes as the last input the output length in bytes. For a message m with l bytes we compute

$$m^{\oplus} = m \oplus \mathsf{MGF1\text{-}SHA\text{-}256}(\mathsf{pk}_{seed} \| addr^c, l).$$

Each of the instances of the tweakable hash function take pk_{seed} as its first input, which is constant for a given key pair and, thus, across a single signature. This leads to redundant computation. To remedy this, we pad pk_{seed} to the length of a full 64-byte SHA-256 input block. Given the Merkle-Damgård construction [Mer79; Dam90] that underlies SHA-256, we improve performance by reusing the intermediate SHA-256 state after the initial call to the compression function.

To ensure that we require the minimal number of calls to the SHA-256 compression function, we use a compressed *addr* for each of these instances. Where realistic, this ensures that the SHA-2 padding fits within the last input block. Rather than storing the layer address and type field in a full 4-byte word each, we only include the least-significant byte of each. Similarly, we only include the least-significant 8 bytes of the 12-byte tree address. This reduces the address from 32 to 22 bytes. We denote such compressed addresses as $addr^c$.

SPHINCS$^+$-Haraka

Our third instantiation is based on the Haraka short-input hash function. Note that Haraka has seen little cryptanalysis and is not a NIST-approved hash function, making this parameter set highly experimental. We specify SPHINCS$^+$-Haraka 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 and a capacity of 256-bit. We adhere to the same padding scheme as used in SHAKE256. We denote this sponge as $\mathsf{HarakaS}(m, d)$, where m is the padded message and d is the length of the message digest in bits.

For a more efficient construction we generate the round constants of Haraka using pk_{seed}.[33] As pk_{seed} is the same for all hash-function calls for a given key pair we expand pk_{seed} using HarakaS and use the result for the round constants in all instantiations of Haraka used in SPHINCS$^+$. In total there are forty 128-bit round constants, defined as

$$RC_0, \ldots, RC_{39} = \mathsf{HarakaS}(\mathsf{pk}_{seed}, 5120).$$

As in the initial compression call in SHA-256, this only has to be done once for each key pair. We denote Haraka with constants derived from pk_{seed} as $\mathsf{Haraka}_{\mathsf{pk}_{seed}}$. For the robust variant, we define the tweakable hash functions as

$$\begin{aligned} F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{Haraka512}_{\mathsf{pk}_{seed}}(addr \| m_1^{\oplus}), \\ H(\mathsf{pk}_{seed}, addr, m_1 \| m_2) &= \mathsf{HarakaS}_{\mathsf{pk}_{seed}}(addr \| m_1^{\oplus} \| m_2^{\oplus}, n), \\ Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{HarakaS}_{\mathsf{pk}_{seed}}(addr \| m^{\oplus}, n). \end{aligned}$$

---

[33] This is similar to the ideas used for the MDx-MAC construction [PO95].

For the simple variant, we instead define the tweakable hash functions as

$$
\begin{aligned}
F(\mathsf{pk}_{seed}, addr, m_1) &= \mathsf{Haraka512}_{\mathsf{pk}_{seed}}(addr \| m_1),\\
H(\mathsf{pk}_{seed}, addr, m_1 \| m_2) &= \mathsf{HarakaS}_{\mathsf{pk}_{seed}}(addr \| m_1 \| m_2, n),\\
Th_\alpha(\mathsf{pk}_{seed}, addr, m) &= \mathsf{HarakaS}_{\mathsf{pk}_{seed}}(addr \| m, n).
\end{aligned}
$$

For F, we pad m_1 and $m_1^{\oplus}$ with zero if $n < 256$. For efficiency reasons, the mask for the message used in F is generated by computing

$$m_1^{\oplus} = m_1 \oplus \mathsf{Haraka256}_{\mathsf{pk}_{seed}}(addr).$$

For all other purposes the masks are generated using HarakaS. For a message m with l bytes we compute

$$m^{\oplus} = m \oplus \mathsf{HarakaS}_{\mathsf{pk}_{seed}}(addr, l).$$

3.7.4 Performance

To illustrate the performance of instances of the SPHINCS$^+$ framework, we provide optimized implementations for all of the parameter sets discussed above. These implementations make use of AVX2 vector instructions to parallelize instances of the hash-function calls. For Haraka, we additionally make use of the AES-NI extension for the internal AES rounds. Altogether, the optimization target, hash-function construction, underlying hash function, and security level result in $2 \times 2 \times 3 \times 3 = 36$ parameter sets. See Table 3.8.

In [BHK+19], we discuss performance numbers in relation to SPHINCS-256, Picnic [CDG+19] and Gravity-SPHINCS [AE17]. For a fair comparison, we instantiate SPHINCS$^+$ so that its security corresponds to the scheme we compare to. These custom parameter sets include adjustments to account for assumptions such as the number of signatures under the same key (e.g., 2^{50} for SPHINCS-256, rather than the 2^{64} required by NIST). Besides demonstrating competitiveness, all of this underscores the flexibility of the SPHINCS$^+$ framework.

Table 3.8: Performance of optimized implementations for all SPHINCS$^+$ signature instances proposed to NIST. Benchmarks were performed on a 3.5 GHz Intel Xeon E3-1275 V3 (Haswell).

Parameter set		Cycles			Bytes	
	keypair	sign	verify	sig	pk	sk
SHAKE256-128s-simple	128 154 676	2 041 365 350	3 951 142	8 080	32	64
SHAKE256-128s-robust	250 818 474	3 701 426 810	7 615 270	8 080	32	64
SHAKE256-128f-simple	4 018 144	131 989 768	9 557 542	16 976	32	64
SHAKE256-128f-robust	7 851 034	245 065 142	18 993 432	16 976	32	64
SHAKE256-192s-simple	194 000 638	4 378 342 330	5 923 106	17 064	48	96
SHAKE256-192s-robust	374 059 710	7 584 715 214	11 398 728	17 064	48	96
SHAKE256-192f-simple	6, 079, 376	173 513 530	15 523 074	35 664	48	96
SHAKE256-192f-robust	11 695 144	326 736 564	29 729 294	35 664	48	96
SHAKE256-256s-simple	253 651 290	3 086 754 562	7 783 684	29 792	64	128
SHAKE256-256s-robust	480 242 128	5 551 086 830	15 116 818	29 792	64	128
SHAKE256-256f-simple	15 875 308	373 185 700	15 397 090	49 216	64	128
SHAKE256-256f-robust	30 041 464	682 683 022	30 727 218	49 216	64	128
SHA-256-128s-simple	49 078 104	835 272 076	2 348 916	8 080	32	64
SHA-256-128s-robust	94 988 100	1 624 566 118	4 700 588	8 080	32	64
SHA-256-128f-simple	1 602 368	51 805 308	5 676 578	16 976	32	64
SHA-256-128f-robust	2 978 018	96 974 576	11 401 188	16 976	32	64
SHA-256-192s-simple	69 860 954	1 737 629 602	3 662 790	17 064	48	96
SHA-256-192s-robust	134 664 612	3 024 929 742	7 784 118	17 064	48	96
SHA-256-192f-simple	2 116 010	66 380 214	9 611 814	35 664	48	96
SHA-256-192f-robust	4 390 738	133 192 018	19 219 918	35 664	48	96
SHA-256-256s-simple	85 946 882	1 121 074 298	4 903 926	29 792	64	128
SHA-256-256s-robust	350 260 762	4 064 645 574	13 790 402	29 792	64	128
SHA-256-256f-simple	5 298 662	133 374 038	9 408 596	49 216	64	128
SHA-256-256f-robust	21 672 826	495 051 104	26 825 462	49 216	64	128
Haraka-128s-simple	19 984 598	383 658 068	545 352	8 080	32	64
Haraka-128s-robust	25 340 702	526 821 772	829 266	8 080	32	64
Haraka-128f-simple	643 370	22 936 196	1 188 352	16 976	32	64
Haraka-128f-robust	809 006	30 719 668	1 890 584	16 976	32	64
Haraka-192s-simple	29 838 170	830 939 210	764 448	17 064	48	96
Haraka-192s-robust	39 650 538	1 312 001 676	1 451 896	17 064	48	96
Haraka-192f-simple	956 708	27 551 500	1 906 088	35 664	48	96
Haraka-192f-simple	1 260 024	38 911 468	3 482 634	35 664	48	96
Haraka-256s-simple	40 094 962	572 899 448	1 091 290	29 792	64	128
Haraka-256s-robust	51 961 586	807 399 570	1 799 156	29 792	64	128
Haraka-256f-simple	2 528 384	65 363 906	2 037 918	49 216	64	128
Haraka-256f-robust	3 268 332	90 442 914	3 351 188	49 216	64	128

Chapter 4

MQ-based signatures

> This chapter is based on the peer-reviewed papers *"From 5-pass $\mathcal{MQ}$-based identification to $\mathcal{MQ}$-based signatures"* [CHR+16] and *"SOFIA: $\mathcal{MQ}$-based signatures in the QROM"* [CHR+18], and the MQDSS submission to NIST's Post-Quantum Cryptography Standardization project [CHR+17].

In the previous chapter, we have discussed the most conservative and perhaps most well-established approach to post-quantum signatures. Here, we venture somewhat into the unknown, exploring signatures based on solving large systems of multivariate quadratic equations over finite fields: the $\mathcal{MQ}$ problem.

For random instances this problem is NP-complete [GJ79]. However, all schemes in this class that have been proposed with actual parameters for practical use share two properties that often raise concerns about their security. First, their security arguments are rather ad-hoc; there is no reduction from the hardness of $\mathcal{MQ}$. The reason for this is the second property, namely that these systems require a hidden structure in the system of equations; this implies that their security inherently also relies on the hardness of the so-called isomorphism-of-polynomials (IP) problem [Pat96] (or, more precisely, the Extended IP problem [DHY+06] or the similar IP with partial knowledge [Tho13] problem). Time has shown that IP in many of the proposed schemes actually relies on the MinRank problem [Cou01; FLP08], and unfortunately, more than often, on an easy instance of this problem. Therefore, many proposed schemes have been broken not by targeting $\mathcal{MQ}$, but by targeting IP (and thus exploiting the structure in the system of equations). Examples of broken schemes include Oil-and-Vinegar [Pat97] (broken in [KS98b]), SFLASH [CGP] (broken in [DFS+07]), MQQ-Sig [GØJ+11] (broken in [FGP+15]), (Enhanced) TTS [YCC04a; YC05b] (broken in [TW12]), and Enhanced STS [TGT+10] (broken in [TW12]). There are essentially only two proposals from the "$\mathcal{MQ}$ +IP" class of schemes that are still standing: HFEv- variants [PCG01;

PCY+15] and Unbalanced Oil-and-Vinegar (UOV) variants [KPG99; DS05]. Predating NIST's Post-Quantum Cryptography Standardization project, the literature did not, to the best of our knowledge, describe any instantiation of those schemes with parameters that achieve a conservative *post-quantum* security level.

In the realm of $\mathcal{MQ}$-based signatures, one might hope for a scheme that has a tight reduction from $\mathcal{MQ}$ in the quantum random oracle model (QROM) or even the standard model, has small keys and signatures, and is also efficient to perform when instantiated with parameters that offer 128 bits of post-quantum security.

This chapter presents significant steps towards such a scheme. We introduce signature schemes with reductions from $\mathcal{MQ}$ in both the random oracle model (ROM) and the QROM, parameterized to target 128 bits of post-quantum security. We then provide highly optimized implementations that demonstrate practicality.

We first introduce MQDSS [CHR+16]. Fundamentally, this scheme relies on applying the Fiat-Shamir transform to an $\mathcal{MQ}$-based identification scheme by Sakumoto, Shirai, and Hiwatari [SSH11]. We discuss that this idea is not new, but show that earlier attempts were unsuccessful. At roughly 41 KiB the signature we construct is comparable in size to SPHINCS-256 [BHH+15], but the runtime performance is considerably better. In Section 4.6.4, we examine parameterizations of MQDSS in the context of NIST's Post-Quantum Cryptography Standardization project and describe some further tweaks. While the scheme comes with a security reduction from the $\mathcal{MQ}$ problem, this reduction is non-tight.

We also introduce SOFIA [CHR+18]. Based on the same identification scheme, we apply Unruh's transform [Unr15] to derive a signature scheme with a reduction from $\mathcal{MQ}$ in the QROM, as well as a tight reduction in the ROM. With a signature three times the size of MQDSS, SOFIA is somewhat less practical; comparison to other signature schemes with QROM proofs shows that this is a broader problem.

4.1 Identification schemes

An identification scheme (IDS) is a protocol that allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of its identity. Formally, this is covered by the following definition.

Definition 4.1.1 (Identification scheme) *An identification scheme consists of a tuple of probabilistic, polynomial-time algorithms* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$ *such that:*

- *the key-generation algorithm* KeyGen *is a probabilistic algorithm that outputs a public key* pk *and a secret key* sk, *i.e., a key pair* $(\mathsf{pk}, \mathsf{sk})$.

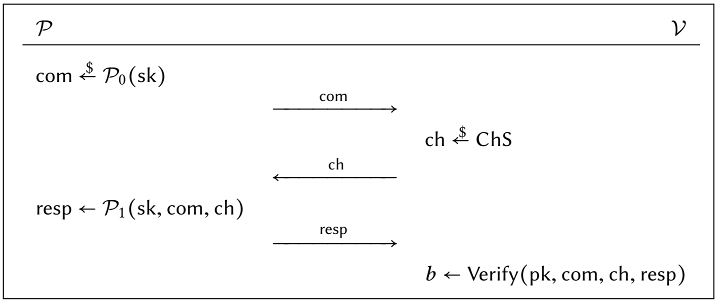

$\mathcal{P}$
$\mathcal{V}$
$\mathsf{com} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk})$
com
$\mathsf{ch} \xleftarrow{\$} \mathsf{ChS}$
ch
$\mathsf{resp} \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}, \mathsf{ch})$
resp
$b \leftarrow \mathsf{Verify}(\mathsf{pk}, \mathsf{com}, \mathsf{ch}, \mathsf{resp})$

Figure 4.1: The canonical 3-pass identification scheme

- *$\mathcal{P}$ 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 Boolean value* True *to indicate an 'accepting' run of the protocol, or* False *to indicate rejection.*

We implicitly assume that $\mathcal{P}$ and $\mathcal{V}$ can maintain state across their subroutines.[1] *For correctness of the scheme we require that for all* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ *we have* $\Pr\left[\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk})\rangle = 1\right] = 1$*, 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 $\mathcal{V}$ with input* pk.

We often categorize identification schemes based on the number of exchanged messages. In general, an n-pass scheme signifies n communication steps between the prover and the verifier. Every concrete instance of an identification scheme discussed in this chapter is either 3-pass or 5-pass, but we will occasionally touch upon generalizations to $(2n + 1)$-pass schemes.

In its so-called 'canonical form', a 3-pass identification scheme can be described by the protocol depicted in Figure 4.1. Here, com is a commitment to some random value, ChS is the challenge space from which a challenge ch is drawn, and resp is the corresponding response. The intuition here is that, by selecting and publishing a commitment, the prover binds themselves to a certain limited set of valid responses to challenges, from which the verifier then requests one.

We define security in terms of two properties: *soundness* and *honest-verifier zero-knowledge.* Cheating provers (i.e., provers unable to respond to arbitrary challenges) may attempt to prepare a commitment so that they are able to correctly

---

[1] In Section 4.7.1, when discussing Unruh's transform [Unr15], running subroutines of the prover based on specific progressions of the state becomes more relevant. There, we do make the state explicit.

respond to one or more possible challenges. The probability of success against a verifier that randomly selects a challenge is referred to as the *soundness error*; a large soundness error implies that a cheating prover is likely to succeed. Conversely, an eavesdropper may attempt to retrieve the secret key sk based on the exchanged messages in a valid exchange. We require the scheme to be honest-verifier zero-knowledge: honest exchanges should not leak information. This is demonstrated by showing that one can recreate a valid transcript without knowledge of the secret key. At first this proving technique may seem to contradict soundness, but consider the fact that such a transcript can be created out of order. A *simulator* creating such a transcript can start from a given challenge, and then adjust the commitment and response accordingly without knowledge of sk.

We are only concerned with passively secure identification schemes; as we use identification schemes as an intermediate step towards the non-interactive construction of signatures, active third-party attackers are not relevant. We now define soundness and honest-verifier zero-knowledge more formally.

Definition 4.1.2 (Soundness) *For an identification scheme* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$*, we say that it is sound with soundness error* κ *if for every polynomially-bounded adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{l}(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}() \\ \langle \mathcal{A}(\mathsf{pk}), \mathcal{V}(\mathsf{pk}) \rangle = 1\end{array}\right] \leq \kappa + \mathsf{negl}(k).$$

Here, k is the security parameter (see Section 2.1.1). Of course, the goal is to obtain an identification scheme with negligible soundness error. This can be achieved by running r rounds of the protocol for a sufficiently large r such that κ^r becomes negligible in the security parameter.

For the following definition, we need a slightly more formal 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 as $\mathsf{trans}(\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk}) \rangle)$.

Definition 4.1.3 ((statistical) Honest-verifier zero-knowledge) *For an identification scheme* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$*, we say that it is statistical honest-verifier zero-knowledge if there exists a probabilistic polynomial-time algorithm* $\mathcal{S}$*, called the simulator, such that the statistical distance between the following two distribution ensembles is negligible in the security parameter.*

$$\{(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}() : (\mathsf{sk}, \mathsf{pk}, \mathsf{trans}(\langle \mathcal{P}(\mathsf{sk}), \mathcal{V}(\mathsf{pk}) \rangle))\}$$
$$\{(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}() : (\mathsf{sk}, \mathsf{pk}, \mathcal{S}(\mathsf{pk}))\}.$$

4.1.1 The Fiat-Shamir transform

In order to construct a digital signature scheme from an identification scheme, we can apply the Fiat-Shamir transform [FS86] (sometimes referred to as the Fiat-Shamir heuristic).[2] This generic transformation takes a 3-pass IDS and outputs a digital signature scheme (as defined in Definition 2.1.7). The construction relies on the idea of replacing the interactive part of the verifier by a random oracle, allowing the prover to unpredictably select a challenge for themselves.

Before applying the transform, we first parallelize the protocol to ensure a negligible soundness error. Here, we apply parallel composition: the prover first publishes all commitments, then receives all challenges, and finally outputs all responses. While the reduction of the soundness error behaves the same as it would for sequential composition, this ensures the protocol remains in the correct format so that the Fiat-Shamir transform can be seamlessly applied. Indeed, the resulting protocol can still be written as an instance of the canonical 3-pass scheme shown in Figure 4.1. We depict the parallel composition of a 3-pass IDS in Figure 4.2.

As mentioned above, part of the role of the verifier is replaced by a randomized process that the prover, whom we will now refer to as the *signer*, can perform. While the theoretical model requires this to be a random oracle, in practice this takes the form of a cryptographic hash function (see Definition 2.1.2). In Figure 4.3, we denote this function as $\mathcal{H}$. It is important to carefully consider how parallel composition of the challenges is done: all challenges are sampled using a single hash-function call. If each challenge was sampled individually, a cheating prover would be able to adaptively select commitments per round to sample the challenge of their choosing. This follows from the fact that the image space of the hash function is coupled to the challenge space; using the output to select only one challenge tremendously restricts the number of distinct images, trivializing preimage search.

After sampling commitments and the corresponding challenges, the signer computes the respective responses. The signature then consists of these commitments and responses. Crucially, the challenges also depend on the message. The verifier independently computes the challenges and verifies the transcripts.

---

[2] The transform actually requires a slightly stronger property of the underlying identification scheme: special soundness. See Definition 4.4.3 on page 126. Intuitively, this relates to the possibility of extracting a solution to the underlying hard problem from two valid transcripts. By constructing such an extractor, one demonstrates that no two such transcripts can be found: the extracted solution would contradict the hardness assumption. For details on the security of the Fiat-Shamir transform, refer to the original definitions and proof in [FS86; CDS94; PS96], but also the preliminaries of, e.g., [Unr17; DFM+19].

$$
\begin{array}{lcl}
\mathcal{P} & & \mathcal{V} \\
\hline
\mathsf{com}^{(1)} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk}) & & \\
\qquad\vdots & & \\
\mathsf{com}^{(r)} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk}) & & \\
 & \xrightarrow{\forall_i\, \mathsf{com}^{(i)}} & \\
 & & \mathsf{ch}^{(1)} \xleftarrow{\$} \mathsf{ChS} \\
 & & \qquad\vdots \\
 & & \mathsf{ch}^{(r)} \xleftarrow{\$} \mathsf{ChS} \\
 & \xleftarrow{\forall_i\, \mathsf{ch}^{(i)}} & \\
\mathsf{resp}^{(1)} \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}^{(1)}, \mathsf{ch}^{(1)}) & & \\
\qquad\vdots & & \\
\mathsf{resp}^{(r)} \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}^{(r)}, \mathsf{ch}^{(r)}) & & \\
 & \xrightarrow{\forall_i\, \mathsf{resp}^{(i)}} & \\
 & & b \leftarrow \forall_i\, \mathsf{Verify}(\mathsf{pk}, \mathsf{com}^{(i)}, \mathsf{ch}^{(i)}, \mathsf{resp}^{(i)})
\end{array}
$$

Figure 4.2: Parallel composition of 3-pass identification scheme

$$
\begin{array}{lcl}
\mathcal{P} & & \mathcal{V} \\
\hline
\mathsf{com}^{(1)} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk}) & & \\
\qquad\vdots & & \\
\mathsf{com}^{(r)} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk}) & & \\
\sigma_0 \leftarrow \mathsf{com}^{(1)}, \mathsf{com}^{(2)}, \cdots \mathsf{com}^{(r)} & & \\
\mathsf{ch}^{(1)}, \mathsf{ch}^{(2)}, \cdots, \mathsf{ch}^{(r)} \leftarrow \mathcal{H}(\sigma_0, m) & & \\
\mathsf{resp}^{(1)} \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}^{(1)}, \mathsf{ch}^{(1)}) & & \\
\qquad\vdots & & \\
\mathsf{resp}^{(r)} \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}^{(r)}, \mathsf{ch}^{(r)}) & & \\
\sigma_1 \leftarrow \mathsf{resp}^{(1)}, \mathsf{resp}^{(2)}, \cdots \mathsf{resp}^{(r)} & & \\
 & \xrightarrow{m, \sigma = (\sigma_0, \sigma_1)} & \\
 & & \mathsf{com}^{(1)}, \mathsf{com}^{(2)}, \cdots, \mathsf{com}^{(r)} \leftarrow \sigma_0 \\
 & & \mathsf{ch}^{(1)}, \mathsf{ch}^{(2)}, \cdots, \mathsf{ch}^{(r)} \leftarrow \mathcal{H}(\sigma_0, m) \\
 & & \mathsf{resp}^{(1)}, \mathsf{resp}^{(2)}, \cdots, \mathsf{resp}^{(r)} \leftarrow \sigma_1 \\
 & & b \leftarrow \forall_i\, \mathsf{Verify}(\mathsf{pk}, \mathsf{com}^{(i)}, \mathsf{ch}^{(i)}, \mathsf{resp}^{(i)})
\end{array}
$$

Figure 4.3: Transformed 3-pass identification scheme

Note that the above transform was designed for 3-pass identification schemes. Further on in this chapter, in Section 4.4, we will look at the Fiat-Shamir transform in the context of a special case of 5-pass identification schemes. To motivate this, we first examine a concrete 5-pass IDS. This requires us to define the hard problem underlying all concrete instances in this chapter: the $\mathcal{MQ}$ problem.

4.2 The MQ problem

We now define the search variant of the $\mathcal{MQ}$ problem, and the related polar form.

Definition 4.2.1 ($\mathcal{MQ}$ problem) *Let $m, n, q \in \mathbb{N}$ and let $\mathbf{x} = (x_1, \ldots, x_n)$ denote a vector of variables x_i over $\mathbb{F}_q$. Then, let $\mathcal{MQ}(n, m, \mathbb{F}_q)$ denote the family of vectorial quadratic functions $\mathbf{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$.*

$$\mathcal{MQ}(n,m,\mathbb{F}_q) = \{\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})) | f_s(\mathbf{x}) = \sum_{i,j} a_{i,j}^{(s)} x_i x_j + \sum_i b_i^{(s)} x_i |_{s=1}^m\}.$$

An instance of the $\mathcal{MQ}$ problem is determined by $\mathbf{F}$ and $\mathbf{v}$, and defined as follows. 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 $\mathbf{F}(\mathbf{s}) = \mathbf{v}$.

Given $\mathbf{v} \in \mathbb{F}_q^m$ we will refer to $\mathbf{F}(\mathbf{x}) = \mathbf{v}$ as a system of m quadratic equations in n variables. We will omit m, n, q whenever they are clear from the context.

Definition 4.2.2 (Polar form) *Let $\mathbf{F} \in \mathcal{MQ}(n, m, \mathbb{F}_q)$. The function $\mathbf{G}(\mathbf{x}, \mathbf{y}) = \mathbf{F}(\mathbf{x} + \mathbf{y}) - \mathbf{F}(\mathbf{x}) - \mathbf{F}(\mathbf{y})$ is called the polar form of $\mathbf{F}$.*

It follows that the polar form is bilinear, that is, for every vector $\mathbf{a}_1, \mathbf{a}_2, \mathbf{b} \in \mathbb{F}_q^n$ and scalar $c \in \mathbb{F}_q$, it holds that

$$\begin{aligned} \mathbf{G}(c(\mathbf{a}_1 + \mathbf{a}_2), \mathbf{b}) &= c\mathbf{G}(\mathbf{a}_1, \mathbf{b}) + c\mathbf{G}(\mathbf{a}_2, \mathbf{b}) \text{ and} \\ \mathbf{G}(\mathbf{b}, c(\mathbf{a}_1 + \mathbf{a}_2)) &= c\mathbf{G}(\mathbf{b}, \mathbf{a}_1) + c\mathbf{G}(\mathbf{b}, \mathbf{a}_2). \end{aligned}$$

The decisional version of the $\mathcal{MQ}$ problem is NP-complete [GJ79]. It is widely believed that the $\mathcal{MQ}$ problem is intractable even for quantum computers in the average case, i.e., that there exists no polynomial-time quantum algorithm that given $\mathbf{F} \xleftarrow{\$} \mathcal{MQ}(n, m, \mathbb{F}_q)$ and $\mathbf{v} = \mathbf{F}(\mathbf{s})$ (for random $\mathbf{s} \xleftarrow{\$} \mathbb{F}_q^n$) outputs a solution $\mathbf{s}'$ to the $\mathcal{MQ}(\mathbf{F}, \mathbf{v})$ problem with non-negligible probability.

4.3 The [SSH11] 5-pass identification scheme

In [SSH11], Sakumoto, Shirai, and Hiwatari propose two new identification schemes: a 3-pass and a 5-pass IDS, based on the intractability of the $\mathcal{MQ}$ problem. Assuming existence of a non-interactive commitment scheme that is statistically hiding and computationally binding, they show that their schemes are statistical zero knowledge and argument of knowledge. They further show that the parallel composition of their protocols is secure against a passive attacker.

The novelty of the approach of Sakumoto, Shirai, and Hiwatari [SSH11] is that, unlike previous public-key schemes, their solution provably relies only on the $\mathcal{MQ}$ problem and the security of the commitment scheme. Crucially, it does not rely on other related problems in multivariate cryptography such as the Isomorphism of Polynomials (IP) problem [Pat96], the related Extended IP [DHY+06] and IP with partial knowledge [Tho13] problems, or the MinRank problem [Cou01; FLP08].

The construction centers around splitting the secret input using the polar form defined above. 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, due to the linearity 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 $\mathbf{s}$. The 5-pass IDS is given in Figure 4.4, where $(\mathsf{pk}, \mathsf{sk}) = (\mathbf{v}, \mathbf{s}) \leftarrow \mathsf{KeyGen}()$. We will look into the practicalities of the IDS in more detail in Section 4.5. In Appendix 4.A, we also introduce the related 3-pass scheme.

Sakumoto, Shirai, and Hiwatari have proven that their 5-pass scheme is statistically zero knowledge when the commitment scheme *Com* is statistically hiding, which implies (honest-verifier) zero knowledge. Here, we prove the soundness property of the scheme.[3]

Theorem 4.3.1 *The 5-pass identification scheme by Sakumoto, Shirai, and Hiwatari (see Figure 4.4) [SSH11] is sound with soundness error* $\frac{1}{2} + \frac{1}{2q}$ *when the commitment scheme Com is computationally binding.*

Before we prove the theorem statement, we first present a lower bound on the cheating probability as it helps to understand the scheme. We show that there

---

[3] Sakumoto, Shirai, and Hiwatari sketch a proof that their 5-pass protocol is argument of knowledge when Com is computationally binding, but our security arguments rely on the weaker notion of soundness.

$\mathcal{P}$		$\mathcal{V}$
$\mathbf{r}_0, \mathbf{t}_0 \xleftarrow{\$} \mathbb{F}_q^n, \mathbf{e}_0 \xleftarrow{\$} \mathbb{F}_q^m$ $\mathbf{r}_1 \leftarrow \mathbf{s} - \mathbf{r}_0$ $c_0 \leftarrow Com(\mathbf{r}_0, \mathbf{t}_0, \mathbf{e}_0)$ $c_1 \leftarrow Com(\mathbf{r}_1, \mathbf{G}(\mathbf{t}_0, \mathbf{r}_1) + \mathbf{e}_0)$		
	$\xrightarrow{(c_0, c_1)}$	
		$\alpha \xleftarrow{\$} \mathbb{F}_q$
	$\xleftarrow{\alpha}$	
$\mathbf{t}_1 \leftarrow \alpha\mathbf{r}_0 - \mathbf{t}_0$ $\mathbf{e}_1 \leftarrow \alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_0$		
	$\xrightarrow{\mathsf{resp}_1 = (\mathbf{t}_1, \mathbf{e}_1)}$	
		$\mathsf{ch}_2 \xleftarrow{\$} \{0, 1\}$
	$\xleftarrow{\mathsf{ch}_2}$	
If $\mathsf{ch}_2 = 0$, $\mathsf{resp}_2 \leftarrow \mathbf{r}_0$ Else $\mathsf{resp}_2 \leftarrow \mathbf{r}_1$		
	$\xrightarrow{\mathsf{resp}_2}$	
		If $\mathsf{ch}_2 = 0$, Parse $\mathsf{resp}_2 = \mathbf{r}_0$, check $c_0 \overset{?}{=} Com(\mathbf{r}_0, \alpha\mathbf{r}_0 - \mathbf{t}_1, \alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_1)$ Else Parse $\mathsf{resp}_2 = \mathbf{r}_1$, check $c_1 \overset{?}{=} Com(\mathbf{r}_1, \alpha(\mathbf{v} - \mathbf{F}(\mathbf{r}_1)) - \mathbf{G}(\mathbf{t}_1, \mathbf{r}_1) - \mathbf{e}_1)$

Figure 4.4: The [SSH11] 5-pass IDS

exists an adversary $\mathcal{C}$, the cheater, that can cheat with probability $\frac{1}{2} + \frac{1}{2q}$. This cheater $\mathcal{C}$ simply follows the protocol using some random $\mathbf{s}'$, with a little difference: they guess α and manipulate the second part of the commitment (c_0, c_1). The reason this works is that for $\mathsf{ch}_2 = 0$, nothing is checked using the verification key $\mathbf{v}$. Thus, in this case the cheater can always win. This gives a success probability of at least 1/2. Furthermore, manipulating c_1 does not influence the success probability when $\mathsf{ch}_2 = 0$, but does increase the success probability when $\mathsf{ch}_2 = 1$.

More formally, for the public $\mathsf{pk} = \mathbf{v}$, the cheater $\mathcal{C}$ chooses $\alpha^* \in \mathbb{F}_q$ as a prediction of what the verifier $\mathcal{V}$ will use in the protocol later on. Then $\mathcal{C}$ chooses $\mathbf{s}', \mathbf{r}_0, \mathbf{t}_0 \in \mathbb{F}_q^n$, $\mathbf{e}_0 \in \mathbb{F}_q^m$ at random, and computes $\mathbf{r}_1 \leftarrow \mathbf{s}' - \mathbf{r}_0$, and $\mathbf{t}_1 \leftarrow \alpha^*\mathbf{r}_0 - \mathbf{t}_0$. Now $\mathcal{C}$ computes the commitment (c_0, c_1) as

$$c_0 \leftarrow Com(\mathbf{r}_0, \mathbf{t}_0, \mathbf{e}_0) and$$

$$c_1 \leftarrow Com(\mathbf{r}_1, \alpha^*(\mathbf{v} - \mathbf{F}(\mathbf{r}_1)) - \mathbf{G}(\mathbf{t}_1, \mathbf{r}_1) - \alpha^*\mathbf{F}(\mathbf{r}_0) + \mathbf{e}_0),$$

and computes $\mathbf{e}_1 \leftarrow \alpha^*\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_0$. It then follows the remainder of the protocol.

Now, when $\mathsf{ch}_2 = 0$, $\mathcal{C}$ always wins, regardless of what α the verifier has chosen. When $\mathsf{ch}_2 = 1$, $\mathcal{C}$ wins when $\alpha = \alpha^*$, i.e., the first challenge was correctly guessed, which happens with probability $1/q$.

This gives $\frac{1}{2} + \frac{1}{2q}$ as a lower bound on the success probability of an adversary. What we want to show now is that there cannot exist a cheater that wins with significantly higher probability as long as the $\mathcal{MQ}$ problem is hard and the used commitment scheme is computationally binding.

Towards a contradiction, assume that there exists a malicious polynomially-bounded cheater $\mathcal{C}$ such that it holds that

$$\epsilon := \Pr[\langle \mathcal{C}(1^k, \mathbf{v}), \mathcal{V}(\mathbf{v})\rangle = 1] - (\frac{1}{2} + \frac{1}{2q}) = \frac{1}{P(k)}$$

for some polynomial function $P(k)$. We show that this implies that there exists a polynomially-bounded adversary $\mathcal{A}$ with access to $\mathcal{C}$ that can either break the binding property of Com or can solve the $\mathcal{MQ}$ problem $\mathcal{MQ}(\mathbf{F}, \mathbf{v})$.

$\mathcal{A}$ can achieve this if they can obtain four accepting transcripts from $\mathcal{C}$ with some internal random tape, equation system $\mathbf{F}$, and public key $\mathbf{v}$, such that for two different α there are two transcripts for each α with different ch_2. This is done by rewinding $\mathcal{C}$ and feeding it with all possible combinations of $\alpha \in [0, q-1]$ and $ch_2 \in \{0, 1\}$. That way we obtain $2q$ different transcripts. Now, per assumption

$\mathcal{C}$ produces an accepting transcript with probability $\frac{1}{2}+\frac{1}{2q}+\epsilon$. Hence, with non-negligible probability ϵ we get at least $q+2$ accepting transcripts. A simple counting argument gives that there has to be a set of four transcripts fulfilling the above conditions. Let these transcripts be $((c_0,c_1),\alpha^{(i)},(\mathbf{t}_1^{(i)},\mathbf{e}_1^{(i)}),\mathrm{ch}_2^{(i)},\mathrm{resp}_2^{(i)})$, where $\alpha^{(1)}=\alpha^{(2)}\neq\alpha^{(3)}=\alpha^{(4)}$, $\mathbf{t}_1^{(1)}=\mathbf{t}_1^{(2)}\neq\mathbf{t}_1^{(3)}=\mathbf{t}_1^{(4)}$, $\mathbf{e}_1^{(1)}=\mathbf{e}_1^{(2)}\neq\mathbf{e}_1^{(3)}=\mathbf{e}_1^{(4)}$, $\mathrm{ch}_2^{(1)}=\mathrm{ch}_2^{(3)}=0$, $\mathrm{ch}_2^{(2)}=\mathrm{ch}_2^{(4)}=1$, $\mathrm{resp}_2^{(1)}=\mathbf{r}_0^{(1)}$, $\mathrm{resp}_2^{(3)}=\mathbf{r}_0^{(3)}$, $\mathrm{resp}_2^{(2)}=\mathbf{r}_1^{(2)}$, $\mathrm{resp}_2^{(4)}=\mathbf{r}_1^{(4)}$. Since the commitment (c_0,c_1) is the same in all four transcripts, we have

$$\begin{gathered}Com(\mathbf{r}_0^{(1)},\alpha^{(1)}\mathbf{r}_0^{(1)}-\mathbf{t}_1^{(1)},\alpha^{(1)}\mathbf{F}(\mathbf{r}_0^{(1)})-\mathbf{e}_1^{(1)})=\\ Com(\mathbf{r}_0^{(3)},\alpha^{(3)}\mathbf{r}_0^{(3)}-\mathbf{t}_1^{(3)},\alpha^{(3)}\mathbf{F}(\mathbf{r}_0^{(3)})-\mathbf{e}_1^{(3)})\text{ and}\end{gathered}\tag{4.1}$$

$$\begin{gathered}Com(\mathbf{r}_1^{(2)},\alpha^{(2)}(\mathbf{v}-\mathbf{F}(\mathbf{r}_1^{(2)}))-\mathbf{G}(\mathbf{t}_1^{(2)},\mathbf{r}_1^{(2)})-\mathbf{e}_1^{(2)})=\\ Com(\mathbf{r}_1^{(4)},\alpha^{(4)}(\mathbf{v}-\mathbf{F}(\mathbf{r}_1^{(4)}))-\mathbf{G}(\mathbf{t}_1^{(4)},\mathbf{r}_1^{(4)})-\mathbf{e}_1^{(4)}).\end{gathered}\tag{4.2}$$

If any of the arguments of *Com* on the left-hand side is different from the one on the right-hand side in (4.1) or in (4.2), then we get two different openings of *Com*, which breaks its computationally binding property.

If they are the same in both (4.1) and (4.2), then from (4.1):

$$(\alpha^{(1)}-\alpha^{(3)})\mathbf{r}_0^{(1)}=\mathbf{t}_1^{(1)}-\mathbf{t}_1^{(3)}$$

$$(\alpha^{(1)}-\alpha^{(3)})\mathbf{F}(\mathbf{r}_0^{(1)})=\mathbf{e}_1^{(1)}-\mathbf{e}_1^{(3)}$$

and from (4.2):

$$(\alpha^{(2)}-\alpha^{(4)})(\mathbf{v}-\mathbf{F}(\mathbf{r}_1^{(2)}))=\mathbf{G}(\mathbf{t}_1^{(2)}-\mathbf{t}_1^{(4)},\mathbf{r}_1^{(2)})+\mathbf{e}_1^{(2)}-\mathbf{e}_1^{(4)}$$

Combining the two,

$$(\alpha^{(2)}-\alpha^{(4)})(\mathbf{v}-\mathbf{F}(\mathbf{r}_1^{(2)}))=(\alpha^{(2)}-\alpha^{(4)})\mathbf{G}(\mathbf{r}_0^{(1)},\mathbf{r}_1^{(2)})+(\alpha^{(2)}-\alpha^{(4)})\mathbf{F}(\mathbf{r}_0^{(1)}),$$

and since $\alpha^{(2)}\neq\alpha^{(4)}$ we get $\mathbf{v}=\mathbf{F}(\mathbf{r}_1^{(2)})+\mathbf{G}(\mathbf{r}_0^{(1)},\mathbf{r}_1^{(2)})+\mathbf{F}(\mathbf{r}_0^{(1)})$. Thus, $\mathbf{r}_0^{(1)}+\mathbf{r}_1^{(2)}$ is a solution to the given $\mathcal{MQ}$ problem. □

4.4 Fiat-Shamir for 5-pass identification schemes

The most efficient identification schemes are often 5-pass schemes. Here, efficiency refers to the combined size of all communication of sufficient rounds to make the soundness error negligible. This, of course, is tightly coupled to the signature size of the resulting signature scheme.

In [EDV+12], the authors present a Fiat-Shamir style transform for $(2n+1)$-pass identification schemes fulfilling a canonical structure. Intuitively, a 5-pass IDS is called canonical in the above sense if $\mathcal{P}$ starts with a commitment com_1, $\mathcal{V}$ replies with a challenge ch_1, $\mathcal{P}$ sends a first response resp_1, $\mathcal{V}$ replies with a second challenge ch_2 and finally $\mathcal{P}$ returns a second response resp_2. Based on the transcript of this exchange, $\mathcal{V}$ then accepts or rejects. See Figure 4.5; this is the natural extension of the 3-pass canonical form described in Section 4.1. The authors of [EDV+12] also present a security reduction for signature schemes derived from such IDS using a security property of the IDS which they call *special n-soundness*. Intuitively, this property says that given two transcripts that agree on all messages but the last challenge and possibly the last response, one can extract a valid secret key. We define this more formally in Definition 4.4.3 on page 126.

In this section, we first show that any $(2n+1)$-pass IDS that fulfills special n-soundness (as required by the security reduction in [EDV+12]) can be converted into a 3-pass IDS by letting $\mathcal{P}$ choose all but the last challenge uniformly at random himself. On the other hand, we argue that existing 5-pass schemes in the literature do not fulfill special n-soundness and prove it for the 5-pass $\mathcal{MQ}$-IDS from [SSH11]. Hence, they can neither be turned into 3-pass schemes, nor does the security reduction from [EDV+12] apply. In the remainder of this section, we define a less generic class of 5-pass IDS which covers many 5-pass IDS, including [CVE10], [Ste93] and [PP03]. In particular, it covers the 5-pass $\mathcal{MQ}$ scheme from [SSH11]. We describe an extractor and show how to apply the Fiat-Shamir transform, but defer the security proof to the underlying paper [CHR+16].

4.4.1 The [EDV+12] proof

Before we can make any statement about IDS that fall into the case of [EDV+12], we have to define the target of our analysis. A $(2n+1)$-pass IDS is an IDS where the prover and the verifier exchange n challenges and replies. More formally:

Definition 4.4.1 (Canonical $(2n+1)$-pass identification schemes) *Define the identification scheme* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$ *to be a* $(2n+1)$*-pass identification scheme with n challenge spaces* ChS_j, $0 < j \leq n$. *We call* IDS *a canonical* $(2n+1)$*-pass identification scheme if the prover can be split into* $n+1$ *subroutines* $\mathcal{P} = (\mathcal{P}_0, \ldots, \mathcal{P}_n)$ *and the verifier into* $n+1$ *subroutines* $\mathcal{V} = (\mathsf{ChS}_1, \ldots, \mathsf{ChS}_n, \mathsf{Verify})$ *such that*

- $\mathcal{P}_0(\mathsf{sk})$ *computes the initial commitment* com *sent as the first message.*

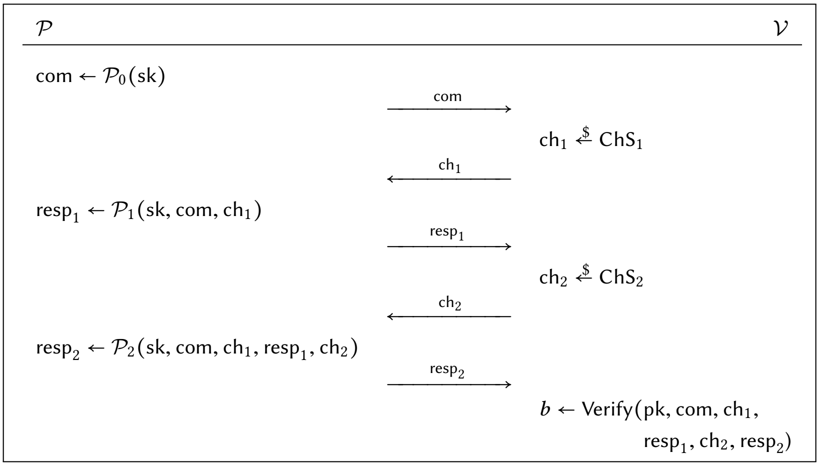

$\mathcal{P}$
$\mathcal{V}$
$\mathsf{com} \leftarrow \mathcal{P}_0(\mathsf{sk})$
com
$\mathsf{ch}_1 \overset{\$}{\leftarrow} \mathsf{ChS}_1$
ch_1
$\mathsf{resp}_1 \leftarrow \mathcal{P}_1(\mathsf{sk}, \mathsf{com}, \mathsf{ch}_1)$
resp_1
$\mathsf{ch}_2 \overset{\$}{\leftarrow} \mathsf{ChS}_2$
ch_2
$\mathsf{resp}_2 \leftarrow \mathcal{P}_2(\mathsf{sk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \mathsf{ch}_2)$
resp_2
$b \leftarrow \mathsf{Verify}(\mathsf{pk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \mathsf{ch}_2, \mathsf{resp}_2)$

Figure 4.5: Canonical 5-pass IDS

- $\mathsf{ChS}_j, j \leq n$ *computes the j-th challenge message* $\mathsf{ch}_j \overset{\$}{\leftarrow} \mathsf{ChS}_j$*, sampling a random element from the j-th challenge space.*

- $\mathcal{P}_i(\mathsf{sk}, \mathsf{trans}_{2i}), 0 < i \leq n$ *computes the i-th response of the prover given access to the secret key and* trans_{2i}*, the transcript so far, containing the first* $2i$ *messages.*

- $\mathsf{Verify}(\mathsf{pk}, \mathsf{trans})$*, upon access to the public key and the whole transcript outputs* $\mathcal{V}$*'s final decision.*

The definition implies that a canonical $(2n + 1)$-pass IDS is *public coin*. In this context, this is a way of expressing that the challenges are sampled from the respective challenge spaces using the uniform distribution.

El Yousfi Alaoui, Dagdelen, Véron, Galindo, and Cayrel propose a generalized Fiat-Shamir transform that turns a canonical $(2n + 1)$-pass IDS into a digital signature scheme. The algorithms of the obtained signature scheme make use of the IDS algorithms as follows.

The key generation is identical to the IDS key generation. The signature algorithm simulates an execution of the IDS, replacing challenge ch_j by the output of a hash function (mapping into ChS_j) that takes as input the concatenation of the message to be signed and all $2(j - 1) + 1$ messages that have been exchanged so far. The signature is simply a transcript of the messages produced by $\mathcal{P}$. The verification algorithm uses the signature and the message to be signed to generate a full transcript, recomputing the challenges using the hash function. Then the verifier runs the Verify routine on the public key and the computed transcript.

El Yousfi Alaoui, Dagdelen, Véron, Galindo, and Cayrel give a security reduction for the resulting signature scheme if the used IDS is honest-verifier zero-knowledge and fulfills *special n-soundness* defined below. The latter is a generalization of the typical special soundness property, which we recall in Definition 4.4.2.

Definition 4.4.2 (Special soundness) *A canonical 3-pass IDS fulfills special soundness if there exists a polynomial-time algorithm $\mathcal{E}$, the extractor, that, given two accepting transcripts* $\mathsf{trans} = (\mathsf{com}, \mathsf{ch}, \mathsf{resp})$ *and* $\mathsf{trans}' = (\mathsf{com}, \mathsf{ch}', \mathsf{resp}')$ *with* $\mathsf{ch} \neq \mathsf{ch}'$, *as well as the public key* pk, *finds a matching secret key* sk *with non-negligible probability.*

Definition 4.4.3 (Special n-soundness) *A canonical $(2n+1)$-pass IDS fulfills special n-soundness if there exists a polynomial-time algorithm $\mathcal{E}$, the extractor, that, given two accepting transcripts* $\mathsf{trans} = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{ch}_n, \mathsf{resp}_n)$ *and* $\mathsf{trans}' = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{ch}'_n, \mathsf{resp}'_n)$ *with* $\mathsf{ch}_n \neq \mathsf{ch}'_n$, *as well as the public key* pk, *finds a matching secret key* sk *with non-negligible probability.*

Special soundness for canonical 3-pass IDS can thus be trivially formulated as special 1-soundness. Note that El Yousfi Alaoui, Dagdelen, Véron, Galindo, and Cayrel define special n-soundness for the resulting signature scheme, which in turn requires the underlying identification scheme to provide special n-soundness. We decided to follow the more common approach, defining the soundness properties for the identification scheme.

We now show that every canonical $(2n+1)$-pass IDS that fulfills special n-soundness can be turned into a canonical 3-pass IDS fulfilling special soundness.

Theorem 4.4.4 *Let* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$ *be a canonical $(2n+1)$-pass IDS that fulfills special n-soundness. Then we can construct a 3-pass identification scheme* $\mathsf{IDS}' = (\mathsf{KeyGen}, \mathcal{P}', \mathcal{V}')$ *that is canonical and fulfills special soundness.*

IDS' *is obtained from* IDS *by moving* $\mathsf{ChS}_j, 0 < j < n$, *(i.e. all but the last challenge generation algorithm) from $\mathcal{V}$ to $\mathcal{P}$, as follows. $\mathcal{P}'$ computes the commitment* $\mathsf{com}' = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{resp}_{n-1}, \mathsf{ch}_{n-1})$ *using $\mathcal{P}_0, \ldots, \mathcal{P}_{n-1}$ and randomly sampling from* $\mathsf{ChS}_1, \ldots, \mathsf{ChS}_{n-1}$. *After $\mathcal{P}'$ sent* com', *$\mathcal{V}'$ replies with* $\mathsf{ch}'_1 \leftarrow \mathsf{ChS}_n$. *$\mathcal{P}'$ then computes* $\mathsf{resp}'_1 \leftarrow \mathcal{P}_n(\mathsf{sk}, \mathsf{trans}_{2n})$ *and $\mathcal{V}'$ verifies the transcript using* Verify.

Proof. By construction, IDS' fits the definition of a canonical 3-pass IDS. We now show that it is honest-verifier zero-knowledge and that it fulfills special soundness.

The latter is straightforward, as two transcripts for IDS' that fulfill the conditions for soundness can be turned into two transcripts for IDS fulfilling the conditions in the n-soundness definition by splitting $\mathsf{com}' = (\mathsf{com}, \mathsf{ch}_1, \mathsf{resp}_1, \ldots, \mathsf{ch}_{n-1}, \mathsf{resp}_{n-1})$ into its parts. Consequently, we can use any extractor for IDS as an extractor for IDS' running in the same time with the same success probability.

Showing honest-verifier zero-knowledge works similarly. A simulator $\mathcal{S}'$ for IDS' can be obtained from any simulator $\mathcal{S}$ for IDS: $\mathcal{S}'$ runs $\mathcal{S}$ to obtain a transcript and regroups the messages to produce a valid transcript for IDS'. Again, $\mathcal{S}'$ runs in essentially the same time as $\mathcal{S}$ and achieves the exact same statistical distance. □

The above result raises the question whether this property was overlooked and we can turn all the 5-pass schemes in the literature into 3-pass schemes. This would have the benefit that we could use the classical Fiat-Shamir transform to turn the resulting schemes into signature schemes.

Sadly, this is not the case. The reason is that the extractors for those IDS need more than two transcripts. For example, the extractor for the 5-pass IDS from [SSH11] needs four unique transcripts that all agree on com. The transcripts have to form two pairs such that within a pair the transcripts agree on ch_1 but not on ch_2, and disagree on ch_1 across the pairs. The proof given by El Yousfi Alaoui, Dagdelen, Véron, Galindo, and Cayrel is flawed. The authors miss that the two secret shares $\mathbf{r}_0$ and $\mathbf{r}_1$ obtained from two different transcripts do not have to be shares of a valid secret key. We now give a formal proof.

Theorem 4.4.5 *The 5-pass identification scheme from [SSH11] does not fulfill special n-soundness if the computational $\mathcal{MQ}$-problem is hard.*

Proof. We prove this by showing that there exist pairs of transcripts, fulfilling the special n-soundness criteria that can be generated by an adversary without knowledge of the secret key simulating just two executions of the protocol. As a key pair for the $\mathcal{MQ}$-IDS by Sakumoto, Shirai, and Hiwatari is a random instance of the $\mathcal{MQ}$ problem, special n-soundness of the 5-pass $\mathcal{MQ}$-IDS would imply that the $\mathcal{MQ}$ problem can be solved in probabilistic polynomial time.

Towards a contradiction, assume there exists a polynomial-time extractor $\mathcal{E}$ against the 5-pass $\mathcal{MQ}$-IDS that fulfills Definition 4.4.3. Using this, we show how to build a polynomial-time solver $\mathcal{A}$ for the $\mathcal{MQ}$ problem.

Given an instance of the $\mathcal{MQ}$ problem $\mathbf{v}$, $\mathcal{A}$ sets $\mathsf{pk} = \mathbf{v}$, which is a valid public key for the $\mathcal{MQ}$-IDS. Next, $\mathcal{A}$ computes two transcripts as follows. $\mathcal{A}$ samples a random $\alpha \in \mathbb{F}_q$ and random $\mathbf{s}, \mathbf{r}_0, \mathbf{t}_0 \in \mathbb{F}_q^n$, $\mathbf{e}_0 \in \mathbb{F}_q^m$, and computes $\mathbf{r}_1 \leftarrow \mathbf{s} - \mathbf{r}_0$, and $\mathbf{t}_1 \leftarrow \alpha\mathbf{r}_0 - \mathbf{t}_0$. Then, $\mathcal{A}$ simulates two successful protocol executions, one for $\mathsf{ch}_2 = 0$, one for $\mathsf{ch}_2 = 1$. To do so, $\mathcal{A}$ impersonates $\mathcal{P}$ and replaces the ch_1 with α, and ch_2 with 0 for the first run and 1 for the second run. In addition, $\mathcal{A}$ uses the knowledge of α to compute the commitments as

$$c_0 \leftarrow Com(\mathbf{r}_0, \mathbf{t}_0, \mathbf{e}_0), \text{and}$$

$$c_1 \leftarrow Com(\mathbf{r}_1, \alpha(\mathbf{v} - \mathbf{F}(\mathbf{r}_1)) - \mathbf{G}(\mathbf{t}_1, \mathbf{r}_1) - (\alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_0)).$$

$\mathcal{A}$ then computes $\mathbf{e}_1 \leftarrow \alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_0$ and sets the second commitment in both runs to $(\mathbf{t}_1, \mathbf{e}_1)$. For $\mathsf{ch}_2 = 0$, $\mathcal{A}$ sets $\mathsf{resp} = \mathbf{r}_0$, and for $\mathsf{ch}_2 = 1$, $\mathcal{A}$ sets $\mathsf{resp} = \mathbf{r}_1$.

Now, the first transcript (where $\mathsf{ch}_2 = 0$) is valid, since by construction of $\mathbf{t}_1$ and $\mathbf{e}_1$, it follows that $\mathbf{t}_0 = \alpha\mathbf{r}_0 - \mathbf{t}_1$ and $\mathbf{e}_0 = \alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_1$. The second transcript (where $\mathsf{ch}_2 = 1$) is also trivially valid, as $\mathbf{e}_1 = \alpha\mathbf{F}(\mathbf{r}_0) - \mathbf{e}_0$ by construction.

Finally, $\mathcal{A}$ feeds the transcripts to $\mathcal{E}$ and outputs whatever $\mathcal{E}$ outputs. $\mathcal{A}$ has the same success probability as $\mathcal{E}$ and runs in essentially the same time. As $\mathcal{E}$ is a polynomial-time algorithm per assumption, this contradicts the hardness of the computational $\mathcal{MQ}$ problem. □

From this, it directly follows that we can also use $\mathcal{A}$ to deal with a parallel execution of many rounds of the scheme. A similar situation arises for all the 5-pass IDS schemes that we found in the literature.

4.4.2 A Fiat-Shamir transform for most $(2n + 1)$-pass IDS

In the previous subsection, we established that the literature does not provide security arguments for signature schemes derived from $(2n + 1)$-pass identification schemes. Here, we present definitions that enable such arguments for most $(2n+1)$-pass IDS in the literature. As most of these IDS are 5-pass schemes that follow a certain structure, we restrict ourselves to these cases. Generalizations to $(2n + 1)$-pass schemes exist, but greatly complicate accessibility of our statements. We will consider a particular type of 5-pass identification protocols where the length of the two challenges is restricted to q and 2.

Definition 4.4.6 ($q2-$identification scheme) *Given $q \in \mathbb{N}$, a $q2-$identification scheme* IDS *is a canonical 5-pass identification scheme where, for the challenge spaces* ChS_1 *and* ChS_2*, it holds that* $|\mathsf{ChS}_1| = q$ *and* $|\mathsf{ChS}_2| = 2$*. Moreover, the probability that the commitment* com *takes a given value is negligible in the security parameter, where the probability is taken over the random choice of the input and the used randomness.*

To keep the security reduction somewhat generic, we also need a property that defines when an extractor exists for a $q2$-IDS. As we have seen, special n-soundness is not applicable. Hence, we give a less generic definition.

Definition 4.4.7 ($q2$-extractor) *We say that a $q2$-identification scheme* IDS *has a* $q2$-extractor *if there exists a polynomial-time algorithm $\mathcal{E}$, the extractor, that given a public key* pk *and four transcripts* $\mathsf{trans}^{(i)} = (\mathsf{com}, \mathsf{ch}_1^{(i)}, \mathsf{resp}_1^{(i)}, \mathsf{ch}_2^{(i)}, \mathsf{resp}_2^{(i)})$, $i \in \{1,2,3,4\}$*, with*

$$\begin{aligned} \mathsf{ch}_1^{(1)} = \mathsf{ch}_1^{(2)} \neq \mathsf{ch}_1^{(3)} = \mathsf{ch}_1^{(4)}, \\ \mathsf{ch}_2^{(1)} = \mathsf{ch}_2^{(3)} \neq \mathsf{ch}_2^{(2)} = \mathsf{ch}_2^{(4)}, \end{aligned} \tag{4.3}$$

valid with respect to pk*, outputs a matching secret key* sk *for* pk *with non-negligible success probability.*

In the next definition, let $\mathsf{IDS}^r = (\mathsf{KeyGen}, \mathcal{P}^r, \mathcal{V}^r)$ be the parallel composition of r rounds of the identification scheme $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$: as before, this is used to amplify the constant soundness error. The construction below uses a polynomial number of rounds r to obtain an IDS with negligible soundness error as an explicit intermediate step.

Construction 4.4.8 (Fiat-Shamir transform for $q2-$IDS) *Let $k \in \mathbb{N}$ be the security parameter, and* $\mathsf{IDS} = (\mathsf{KeyGen}, \mathcal{P}, \mathcal{V})$ *a $q2$-identification scheme that achieves soundness with soundness error κ. Select r, the number of (parallel) rounds of* IDS*, such that $\kappa^r = \mathsf{negl}(k)$, and that the challenge spaces of the composition* IDS^r*,* $\mathsf{ChS}_1^r, \mathsf{ChS}_2^r$ *have exponential size in k. Moreover, select cryptographic hash functions $H_1 : \{0,1\}^* \to \mathsf{ChS}_1^r$ and $H_2 : \{0,1\}^* \to \mathsf{ChS}_2^r$. The $q2$-signature scheme derived from* IDS *is the triplet of algorithms* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *in accordance to Definition 2.1.7, where:*

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}()$,

- $\sigma = (\sigma_0, \sigma_1, \sigma_2) \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{m})$ *where* $\sigma_0 = \mathsf{com} \leftarrow \mathcal{P}_0^r(\mathsf{sk})$, $h_1 = H_1(\mathsf{m}, \sigma_0)$, $\sigma_1 = \mathsf{resp}_1 \leftarrow \mathcal{P}_1^r(\mathsf{sk}, \sigma_0, h_1)$, $h_2 = H_2(\mathsf{m}, \sigma_0, h_1, \sigma_1)$, *and* $\sigma_2 = \mathsf{resp}_2 \leftarrow \mathcal{P}_2^r(\mathsf{sk}, \sigma_0, h_1, \sigma_1, h_2)$.
- $\mathsf{Verify}(\mathsf{pk}, \mathsf{m}, \sigma)$ *parses* $\sigma = (\sigma_0, \sigma_1, \sigma_2)$, *computes the values* $h_1 = H_1(\mathsf{m}, \sigma_0)$, $h_2 = H_2(\mathsf{m}, \sigma_0, h_1, \sigma_1)$ *as above and outputs* $\mathcal{V}^r(\mathsf{pk}, \sigma_0, h_1, \sigma_1, h_2, \sigma_2)$.

Correctness of the scheme follows immediately from the correctness of IDS.

From these properties, it is now possible to construct a security reduction that proves EU-CMA security of the signature scheme in the random oracle model. The proof, based on a variant of the forking lemma [PS96], is considered out of scope for this thesis, and we refer the interested reader to the original publication [CHR+16].

4.5 MQDSS

Having defined the Fiat-Shamir transform for 5-pass identification schemes that use a q-ary and a binary challenge, we use this section to apply the transform to the IDS of [SSH11] (see Section 4.3). Before discussing the 5-pass scheme, which we dub MQDSS, we first briefly examine the results of applying the traditional Fiat-Shamir transform to the 3-pass IDS in [SSH11]. This serves as a baseline and a starting point for comparison. We then present a generic description of MQDSS, and instantiate it based on state-of-the-art attacks that solve the $\mathcal{MQ}$ problem.

The IDS requires an $\mathcal{MQ}$ system $\mathbf{F}$ as input, potentially system-wide. We could simply select one function $\mathbf{F}$ and define it as a system parameter for all users. Instead, we choose to derive it from a unique seed that is included in each public key. This increases the size of pk by k bits, and adds some cost for seed expansion when signing and verifying. However, selecting a single system-wide $\mathbf{F}$ might allow an attacker to focus their efforts on a single $\mathbf{F}$ for all users, and would require whoever selects this system parameter to convince all users of its randomness (which is not trivial [BCC+14]). For consistency with literature, we still occasionally refer to $\mathbf{F}$ as the 'system parameter'.

Note that the signing procedure described below is slightly more involved than is suggested by Construction 4.4.8. Where the transformed construction operates directly on the message, we first apply what is effectively a randomized hash function. As discussed in [HK06], this extra step provides resilience against collisions in the hash function at only little extra cost. A similar construction appears, e.g., in SPHINCS [BHH+15]. The digest (and thus the signature) is still deterministically derived from the message and the secret key.

Establishing a baseline using the 3-pass scheme over $\mathbb{F}_2$

In the interest of brevity and in order not to interrupt the main storyline, we will not go into the details of the derived signature scheme here – instead, we defer this to Appendix 4.A at the end of this chapter and merely summarize the results here.

For the 3-pass scheme, we select $n = m = 256$ over $\mathbb{F}_2$. This results in signatures of 54.81 KiB, and 64 bytes for each of the secret and public keys. We ran benchmarks on a single 3.5 GHz core of an Intel Core i7-4770K CPU, measuring 118 088 992 cycles for signature generation, 8 066 324 cycles for key generation, and 82 650 156 cycles for signature verification (or 33.7 ms, 2.30 ms, and 23.6 ms, respectively).

4.5.1 The 5-pass scheme over $\mathbb{F}_{31}$

The plain 3-pass scheme over $\mathbb{F}_2$ is quite inefficient, both in terms of signature size and signing speed. This is a direct consequence of the large number of variables and equations required to achieve 128 bits of post-quantum security using $\mathcal{MQ}$ over $\mathbb{F}_2$, as well as the high number of rounds required (see Appendix 4.A for an analysis). Using a 5-pass scheme over $\mathbb{F}_{31}$ allows for a smaller n and m, as well as a smaller number of rounds. One might wonder why we do not consider different fields for the 3-pass scenario, instead. This turns out to be suboptimal: contrary to the 5-pass scheme, this does not result in a soundness error reduction, but does increase the transcript size per round.

We now construct the functions KeyGen, Sign and Verify in accordance with Definition 2.1.7. Specific instantiations for the parameters that achieve 128-bit post-quantum security are given in the next subsection; we start by presenting the parameters of the scheme in general.

Parameters

MQDSS is explicitly parameterized by the security parameter $k \in \mathbb{N}$, and $m, n \in \mathbb{N}$ such that the security level of the underlying $\mathcal{MQ}$ instance $\mathcal{MQ}(n, m, \mathbb{F}_{31})$ is at least k bits. The parameters m and n define the size of the equation system $\mathbf{F}$, which we denote as $F_{len} = m \cdot (\frac{n \cdot (n+1)}{2} + n)$. This accounts for both the quadratic and linear terms. We will use r to refer to the number of iterations of the underlying

IDS, which follows directly from the required security level and the soundness error.[4] Note that this should not be confused with $\mathbf{r}_0$ and $\mathbf{r}_1$, which are vectors of elements of $\mathbb{F}_{31}$. We will require the following functions:

- cryptographic hash functions $\mathcal{H} : \{0,1\}^* \to \{0,1\}^k$, $H_1 : \{0,1\}^k \times \{0,1\}^k \to {\mathbb{F}_{31}}^r$, and $H_2 : \{0,1\}^k \times \{0,1\}^k \times \mathbb{F}_{31}^r \times \mathbb{F}_{31}^{rn+rm} \to \{0,1\}^r$,
- two string commitment functions $Com_0 : {\mathbb{F}_{31}}^n \times {\mathbb{F}_{31}}^n \times {\mathbb{F}_{31}}^m \to \{0,1\}^k$ and $Com_1 : {\mathbb{F}_{31}}^n \times {\mathbb{F}_{31}}^m \to \{0,1\}^k$,
- two pseudorandom generators $G_s : \{0,1\}^k \to {\mathbb{F}_{31}}^n$ and $G_{\mathrm{rte}} : \{0,1\}^k \times \{0,1\}^k \to {\mathbb{F}_{31}}^{r\cdot(2n+m)}$, and
- an extendable output function[5] $\mathrm{XOF}_F : \{0,1\}^k \to {\mathbb{F}_{31}}^{F_{len}}$.

We now describe the key generation, signing and verification procedures. See Algorithms 19, 20 and 21 for summarized descriptions in pseudocode.

Key generation

We first sample a secret seed of k bits $\mathcal{S}_{\mathrm{sk}} \xleftarrow{\$} \{0,1\}^k$, as well as[6] a public seed $\mathcal{S}_F \xleftarrow{\$} \{0,1\}^k$. We then select an $\mathcal{MQ}$ system $\mathbf{F}$ from $\mathcal{MQ}(n, m, \mathbb{F}_{31})$ by expanding $\mathcal{S}_F$ using the extendable output function XOF_F.

In order to compute the public value $\mathbf{v}$, we require a secret as input to the $\mathcal{MQ}$ function defined by $\mathbf{F}$. We use $\mathcal{S}_{\mathrm{sk}}$ as input to the pseudorandom generator G_s, and derive $\mathbf{s} = G_s(\mathcal{S}_{\mathrm{sk}})$. We then evaluate the $\mathcal{MQ}$ function to compute $\mathbf{v} = \mathbf{F}(\mathbf{s})$. The secret key $\mathrm{sk} = (\mathcal{S}_{\mathrm{sk}}, \mathcal{S}_F)$ and the public key $\mathrm{pk} = (\mathcal{S}_F, \mathbf{v})$ require $2 \cdot k$ and $k + 5 \cdot m$ bits respectively, using 5 bits per $\mathbb{F}_{31}$ element.

Signing

The signature algorithm takes as input a message m and a secret key $\mathrm{sk} = (\mathcal{S}_{\mathrm{sk}}, \mathcal{S}_F)$. Similarly as in the key generation, we derive $\mathbf{F} = \mathrm{XOF}_F(\mathcal{S}_F)$. Then, we derive a message-dependent random value $R = \mathcal{H}(\mathcal{S}_{\mathrm{sk}} \parallel \mathsf{m})$, where "$\parallel$" is string concatenation. Using this random value R, we compute the randomized message digest $\mathrm{md} = \mathcal{H}(R \parallel \mathsf{m})$. The value R must be included in the signature, so that a verifier can derive the same randomized digest.

---

[4] Recall that the soundness error of the 5-pass scheme is $\frac{1}{2} + \frac{1}{2q}$; for $q = 31$, this is $\frac{16}{31}$.

[5] In MQDSS as presented in [CHR+16], this function is also modeled as an a pseudorandom generator.

[6] In practice, one would even simply derive $\mathcal{S}_F$ from $\mathcal{S}_{\mathrm{sk}}$.

```
Algorithm 19 MQDSS.KeyGen ()                                  $k, G_s, \mathrm{XOF}_F$
    $\mathcal{S}_{\mathrm{sk}} \xleftarrow{\$} \{0,1\}^k$
    $\mathcal{S}_F \xleftarrow{\$} \{0,1\}^k$
    $\mathbf{F} \leftarrow \mathrm{XOF}_F(\mathcal{S}_F)$
    $\mathbf{s} \leftarrow G_s(\mathcal{S}_{\mathrm{sk}})$
    $\mathbf{v} \leftarrow \mathbf{F}(\mathbf{s})$
    return $\mathsf{pk} = (\mathcal{S}_F, \mathbf{v}), \mathsf{sk} = (\mathcal{S}_{\mathrm{sk}}, \mathcal{S}_F)$
```

Given $\mathcal{S}_{\mathrm{sk}}$ and md, we now compute $G_{\mathrm{rte}}(\mathcal{S}_{\mathrm{sk}}, \mathsf{md})$ to produce $(\mathbf{r}_0^{(1)}, \ldots, \mathbf{r}_0^{(r)}, \mathbf{t}_0^{(1)}, \ldots, \mathbf{t}_0^{(r)}, \mathbf{e}_0^{(1)}, \ldots, \mathbf{e}_0^{(r)})$. Using these values, we compute $c_0^{(i)}$ and $c_1^{(i)}$ for each round i, as defined in the IDS. Recall that $\mathbf{G}(\mathbf{x}, \mathbf{y}) = \mathbf{F}(\mathbf{x} + \mathbf{y}) - \mathbf{F}(\mathbf{x}) - \mathbf{F}(\mathbf{y})$, and that Com_0 and Com_1 are string commitment functions:

$$c_0^{(i)} = Com_0(\mathbf{r}_0^{(i)}, \mathbf{t}_0^{(i)}, \mathbf{e}_0^{(i)}) \quad \text{and} \quad c_1^{(i)} = Com_1(\mathbf{r}_1^{(i)}, \mathbf{G}(\mathbf{t}_0^{(i)}, \mathbf{r}_1^{(i)}) + \mathbf{e}_0^{(i)}).$$

As mentioned in [SSH11] (and originally suggested in [Ste96]), it is not necessary to include all $2r$ commitments in the transcript.[7] Instead, we include a digest over the concatenation of all commitments $\sigma_0 = \mathcal{H}(c_0^{(1)} \| c_1^{1)} \| \ldots \| c_0^{(r)} \| c_1^{(r)})$. We derive the challenges $\alpha^{(i)} \in \mathbb{F}_{31}$ (for $0 \leq i < r$) by applying H_1 to $h_1 = (\mathsf{md}, \sigma_0)$. Using these $\alpha^{(i)}$, the vectors $\mathbf{t}_1^{(i)} = \alpha^{(i)} \cdot \mathbf{r}_0^{(i)} - \mathbf{t}_0^{(i)}$ and $\mathbf{e}_1^{(i)} = \alpha^{(i)} \cdot \mathbf{F}(\mathbf{r}_0^{(i)}) - \mathbf{e}_0^{(i)}$ can be computed.

Let $\sigma_1 = (\mathbf{t}_1^{(1)}, \ldots, \mathbf{t}_1^{(r)}, \mathbf{e}_1^{(1)}, \ldots, \mathbf{e}_1^{(r)})$. We compute h_2 by applying H_2 to the tuple $(\mathsf{md}, \sigma_0, h_1, \sigma_1)$ and use it as r binary challenges $\mathsf{ch}_2^{(i)} \in \{0,1\}$.

Now we define $\sigma_2 = (\mathbf{r}^{(1)}_{\mathsf{ch}_2^{(1)}}, \ldots, \mathbf{r}^{(r)}_{\mathsf{ch}_2^{(r)}}, c^{(1)}_{(1-\mathsf{ch}_2^{(1)})}, \ldots, c^{(r)}_{(1-\mathsf{ch}_2^{(r)})})$. Note that here we also need to include the challenges $c_{1-\mathsf{ch}_2^{(i)}}$ that the verifier cannot recompute. We then output $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$ as the signature. At 5 bits per $\mathbb{F}_{31}$ element, the size of the signature is $(2 + r) \cdot k + 5 \cdot r \cdot (2 \cdot n + m)$ bits.

Verification

The verification algorithm takes as input the message, the signature $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$ and the public key $\mathsf{pk} = (\mathcal{S}_F, \mathbf{v})$. As above, we use R and m to compute md, and derive $\mathbf{F}$ from $\mathcal{S}_F$ using XOF_F. As the signature contains σ_0, we can compose h_1 and, consequentially, compute the challenge values $\alpha^{(i)}$ for all r rounds by using H_1. Similarly, the values $\mathsf{ch}_2^{(i)}$ are computed by applying H_2 to $(\mathsf{md}, \sigma_0, h_1, \sigma_1)$. For

---

[7] See Section 4.8.1 for a more elaborate description of this.

Algorithm 20 MQDSS.Sign $(\mathsf{m}, \mathsf{sk})$ $\quad r, Com_0, Com_1, G_s, G_{\text{rte}}, \mathcal{H}, H_1, H_2, \text{XOF}_F$

```
$\mathcal{S}_{\mathsf{sk}}, \mathcal{S}_F = \mathsf{sk}$
$\mathbf{F} \leftarrow \text{XOF}_F(\mathcal{S}_F)$
$R \leftarrow \mathcal{H}(\mathcal{S}_{\mathsf{sk}} \parallel \mathsf{m})$
$\mathbf{s} \leftarrow G_s(\mathcal{S}_{\mathsf{sk}})$
$\mathsf{md} \leftarrow \mathcal{H}(R \parallel \mathsf{m})$
$\mathbf{r}_0^{(1)}, \dots, \mathbf{r}_0^{(r)}, \mathbf{t}_0^{(1)}, \dots, \mathbf{t}_0^{(r)}, \mathbf{e}_0^{(1)}, \dots, \mathbf{e}_0^{(r)} \leftarrow G_{\text{rte}}(\mathcal{S}_{\mathsf{sk}}, \mathsf{md})$
for $i \in \{1, \dots, r\}$ do
    $c_0^{(i)} \leftarrow Com_0(\mathbf{r}_0^{(i)}, \mathbf{t}_0^{(i)}, \mathbf{e}_0^{(i)})$
    $c_1^{(i)} \leftarrow Com_1(\mathbf{r}_1^{(i)}, \mathbf{G}(\mathbf{t}_0^{(i)}, \mathbf{r}_1^{(i)}) + \mathbf{e}_0^{(i)})$
end for
$\sigma_0 \leftarrow \mathcal{H}(c_0^{(1)} \| c_1^{(0)} \| \dots \| c_0^{(r)} \| c_1^{(r)})$
$h_1 = (\alpha^{(1)}, \dots, \alpha^{(r)}) \leftarrow H_1(\mathsf{md}, \sigma_0)$
for $i \in \{1, \dots, r\}$ do
    $\mathbf{t}_1^{(i)} \leftarrow \alpha^{(i)} \cdot \mathbf{r}_0^{(i)} - \mathbf{t}_0^{(i)}$
    $\mathbf{e}_1^{(i)} \leftarrow \alpha^{(i)} \cdot \mathbf{F}(\mathbf{r}_0^{(i)}) - \mathbf{e}_0^{(i)}$
end for
$\sigma_1 = (\mathbf{t}_1^{(1)}, \dots, \mathbf{t}_1^{(r)}, \mathbf{e}_1^{(1)}, \dots, \mathbf{e}_1^{(r)})$
$\mathsf{ch}_2^{(1)}, \dots, \mathsf{ch}_2^{(r)} \leftarrow H_2(\mathsf{md}, \sigma_0, h_1, \sigma_1)$
$\sigma_2 = (\mathbf{r}^{(1)}_{\mathsf{ch}_2^{(1)}}, \dots, \mathbf{r}^{(r)}_{\mathsf{ch}_2^{(r)}}, c^{(1)}_{(1-\mathsf{ch}_2^{(1)})}, \dots, c^{(r)}_{(1-\mathsf{ch}_2^{(r)})})$
return $\sigma = (R, \sigma_0, \sigma_1, \sigma_2)$
```

each round i, the verifier extracts vectors $\mathbf{t}_1^{(i)}$ and $\mathbf{e}_1^{(i)}$ from σ_1 and $\mathbf{r}^{(i)}$ from σ_2. Half the commitments can now be computed; which ones depends on $\mathsf{ch}_2^{(i)}$.

$$\begin{array}{ll} \text{if } \mathsf{ch}_2^{(i)} = 0 & c_0^{(i)} = Com_0(\mathbf{r}^{(i)}, \alpha^{(i)} \cdot \mathbf{r}^{(i)} - \mathbf{t}_1^{(i)}, \alpha^{(i)} \cdot \mathbf{F}(\mathbf{r}^{(i)}) - \mathbf{e}_1^{(i)}) \\ \text{if } \mathsf{ch}_2^{(i)} = 1 & c_1^{(i)} = Com_1(\mathbf{r}^{(i)}, \alpha^{(i)} \cdot (\mathbf{v} - \mathbf{F}(\mathbf{r}^{(i)})) - \mathbf{G}(\mathbf{t}_1^{(i)}, \mathbf{r}^{(i)}) - \mathbf{e}_1^{(i)}) \end{array}$$

Extracting the missing commitments $c^{(i)}_{(1-\mathsf{ch}_2^{(i)})}$ from σ_2, the verifier now computes $\sigma_0' = \mathcal{H}(c_0^{(1)} \| c_1^{(0)} \| \ldots \| c_0^{(r)} \| c_1^{(r)})$. For verification to succeed, test if $\sigma_0' = \sigma_0$.

Algorithm 21 MQDSS.Verify (m, σ, pk) $r, \mathcal{H}, \mathrm{XOF}_F, Com_0, Com_1, H_1, H_2$

```
(S_F, v) = pk  ->  $(\mathcal{S}_F, \mathbf{v}) = \mathsf{pk}$
```

```
$(\mathcal{S}_F, \mathbf{v}) = \mathsf{pk}$
$(R, \sigma_0, \sigma_1, \sigma_2) = \sigma$
$\mathbf{F} \leftarrow \mathrm{XOF}_F(\mathcal{S}_F)$
$\mathsf{md} \leftarrow \mathcal{H}(R \| \mathsf{m})$
$h_1 = \alpha^{(1)}, \ldots, \alpha^{(r)} \leftarrow H_1(\mathsf{md}, \sigma_0)$
$\mathsf{ch}_2^{(1)}, \ldots, \mathsf{ch}_2^{(r)} \leftarrow H_2(\mathsf{md}, \sigma_0, h_1, \sigma_1)$
$(\mathbf{t}_1^{(1)}, \ldots, \mathbf{t}_1^{(r)}, \mathbf{e}_1^{(1)}, \ldots, \mathbf{e}_1^{(r)}) = \sigma_1$
$(\mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(r)}, c^{(1)}_{(1-\mathsf{ch}_2^{(1)})}, \ldots, c^{(r)}_{(1-\mathsf{ch}_2^{(r)})}) = \sigma_2$
for $i \in \{1, \ldots, r\}$ do
    if $\mathsf{ch}_2^{(i)} = 0$ then
        $c_0^{(i)} \leftarrow Com_0(\mathbf{r}^{(i)}, \alpha^{(i)} \cdot \mathbf{r}^{(i)} - \mathbf{t}_1^{(i)}, \alpha^{(i)} \cdot \mathbf{F}(\mathbf{r}^{(i)}) - \mathbf{e}_1^{(i)})$
    else if $\mathsf{ch}_2^{(i)} = 1$ then
        $c_1^{(i)} \leftarrow Com_1(\mathbf{r}^{(i)}, \alpha^{(i)} \cdot (\mathbf{v} - \mathbf{F}(\mathbf{r}^{(i)})) - \mathbf{G}(\mathbf{t}_1^{(i)}, \mathbf{r}^{(i)}) - \mathbf{e}_1^{(i)})$
    end if
end for
$\sigma_0' \leftarrow \mathcal{H}(c_0^{(1)} \| c_1^{(0)} \| \ldots \| c_0^{(r)} \| c_1^{(r)})$
return $\sigma_0' \overset{?}{=} \sigma_0$
```

Security reduction

In [CHR+16], we present a game-based security reduction for MQDSS in the random oracle model (ROM). In particular, we show that MQDSS is EU-CMA-secure in the ROM if:

- the $\mathcal{MQ}$ problem is intractable,

- 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 the probability that their output takes a given value is negligible in the security parameter,
- the extendable output function XOF_F is modeled as random oracle, and
- the pseudorandom generators G_s and G_{rte} have outputs computationally indistinguishable from random.

This proof naturally relies on Construction 4.4.8, as introduced in the previous section. As this approach is non-tight, the proof only covers 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. Also, the reduction is in the random oracle model and not in the quantum random oracle model (QROM), limiting applicability in the post-quantum setting. We consider strengthening this statement important future work; research in this direction is ongoing [DFG13; ARU14; Unr17; KLS18; LZ19; DFM+19]. In Section 4.8, we introduce SOFIA, a signature scheme based on the [SSH11] identification scheme that does allow for a proof in the QROM.

4.6 MQDSS-31-64

In this section, we provide a concrete instance of MQDSS. We discuss a suitable set of parameters to achieve the desired security level, discuss an optimized software implementation, and present benchmark results.

4.6.1 Parameter selection

For the 5-pass scheme, the soundness error κ is affected by the size of q. This motivates a field choice larger than $\mathbb{F}_2$ in order to reduce the number of rounds required. From an implementation point of view, it is beneficial to select a small prime, allowing very cheap multiplications as well as comparatively cheap field reductions. We choose $\mathbb{F}_{31}$ with the intention of storing it in a 16-bit value – the benefits of which become clear in the next subsection, where we discuss the required modular reductions.

We now consider the choice of $\mathcal{MQ}(n, m, \mathbb{F}_{31})$, i.e. the parameters n and m. Several generic classical algorithms exist for solving systems of quadratic equations over finite fields, such as the F4 algorithm [Fau99] and the F5 algorithm [Fau02; BFS15] using Gröbner basis techniques, the Hybrid approach [BFP09; BFP12], and the XL algorithm [CKP+00; Die04] and variants [YC05a].

At the time of writing, for fields $\mathbb{F}_q$ where $q \geqslant 4$, the best known technique for solving overdetermined systems of equations over $\mathbb{F}_q$ is combining equation solvers with exhaustive search. The Hybrid Approach and the FXL variant of XL [YC05a] use this paradigm. We analyze the complexity using the Hybrid approach; the complexity for the XL family of algorithms is similar [YCY13].

The F5 algorithm as well as the Hybrid approach perform better when the number of equations is bigger than the number of variables, so from this point of view there is no incentive in choosing $m > n$. On the other hand, if $m < n$, then we can simply fix $n - m$ variables and reduce the problem to a smaller one, with m variables. Therefore, in terms of classical security the best choice is $m = n$.

Following the analysis from [BFP09; BFP12] and considering our goal of obtaining classical security of at least 128 bits, we need to choose $n \geq 51$, so that the Hybrid approach would need at least 2^{128} operations.[8] For implementation reasons, we scale this up even further, and choose $n = 64$. In particular, a multiple of 16 suggests efficient register usage for vectorized implementations. In this case, the complexity of the Hybrid approach is approximately 2^{177}.

Regarding post-quantum security, there was no dedicated quantum algorithm for solving systems of quadratic equations at the time of writing. This has slightly changed since [BY18]. We briefly come back to this in Section 4.9.1. For MQDSS, we worked under the assumption that one could use Grover's search algorithm [Gro96] to directly attack the $\mathcal{MQ}$ problem, or use Grover's algorithm for the search part in a quantum implementation of the Hybrid method. The latter requires an efficient quantum implementation of the F5 algorithm, which, given its expensive memory usage, is far from trivial. We provide a brief analysis of this approach in [CHR+16], estimating that the quantum version of the Hybrid method has a time complexity of approximately 2^{139} operations.

To achieve EU-CMA for 128 bits of post-quantum security, at the time of writing, we required that $\kappa^r \leq 2^{-256}$, as an adversary could perform a preimage search to

---

[8] The Hybrid approach is parameterized by a so-called 'linear algebra constant' ω, signifying assumptions on the effectiveness of the attack. This estimate uses a highly conservative $\omega = 2$. If we set the more realistic value of $\omega = 2.3$, the minimum is $n = 45$.

effectively control the challenges. As $\kappa = \frac{q+1}{2q}$ with $q = 31$, this leads to $r = 269$.[9] To complete the scheme, we instantiate the functions $\mathcal{H}$, Com_0 and Com_1 with SHA3-256, and use SHAKE-128 for H_1, H_2, XOF_F, G_{rte}, and G_s [BDP+11]. Domain separation is achieved implicitly through difference in input length and repeated application of SHAKE-128. In order to convert between the output of SHAKE-128 and functions that map to vectors over $\mathbb{F}_{31}$, we simply reject and resample values that are not in $\mathbb{F}_{31}$ (effectively applying an instance of the second TSS08 construction from [WHC+13]).

We refer to this instance of the scheme as MQDSS-31-64.

4.6.2 Implementation details

The central and most costly computation in this signature scheme is the evaluation of **F** (and, by corollary, **G**). The signing procedure requires one evaluation of each for every round, and the verifier needs to compute either **F** (if $\mathrm{ch}_2 = 0$) or both **F** and **G** (if $\mathrm{ch}_2 = 1$), for each round. Other than these functions, the computational effort is made up of seed expansion, several hash-function applications and a small number of additions and subtractions in $\mathbb{F}_{31}$. For SHA-3-256 and SHAKE-128, we rely on existing code from the Keccak Code Package [BDP+18]. Clearly, the focus for an optimized implementation should be on the $\mathcal{MQ}$ function. Previous work [CCC+09] shows that modern CPUs offer interesting and valuable methods to efficiently implement this primitive, in particular by exploiting the high level of internal parallelism.

Compared to the binary 3-pass scheme (see Appendix 4.A), the implementation of the 5-pass scheme over $\mathbb{F}_{31}$ presents more challenges. As regular integer multiplication and addition will typically take us outside of $\mathbb{F}_{31}$, results of computations need to be reduced to smaller representations. We generally represent field elements as unsigned 16-bit values to avoid having to this too frequently, During specific parts of the computation, we vary this representation as needed.

The evaluation of **F** can roughly be divided into two parts: the generation of all monomials, and computation of the resulting polynomials for known monomials.

---

[9] Actually, $k = 128$ implies that $r = 135$ suffices to achieve the desired security level. This was fixed in a revision of the MQDSS submission to NIST's Post-Quantum Cryptography Standardization project. For the results presented in this section (and for the 3-pass scheme as presented in Appendix 4.A), we adhere to parameters as presented in [CHR+16] for consistency. See Section 4.6.4 and the official comment at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/official-comments/MQDSS-official-comment.pdf`.

&	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	00	11	22	33	04	15	26	37	08	19	2A	3B	0C	1D	2E	3F
	10	21	32	03	14	25	36	07	18	29	3A	0B	1C	2D	3E	0F
	-	-	-	-	44	55	66	77	48	59	6A	7B	4C	5D	6E	7F
	50	61	72	43	54	65	76	47	58	69	7A	4B	5C	6D	7E	4F
	-	-	-	-	-	-	-	-	88	99	AA	BB	8C	9D	AE	BF
	90	A1	B2	83	94	A5	B6	87	98	A9	BA	8B	9C	AD	BE	8F
	-	-	-	-	-	-	-	-	-	-	-	-	CC	DD	EE	FF
	D0	E1	F2	C3	D4	E5	F6	C7	D8	E9	FA	CB	DC	ED	FE	CF
	02	13	-	-	42	53	-	-	82	93	-	-	C2	D3	-	-
	06	17	-	-	46	57	-	-	86	97	-	-	C6	D7	-	-
	0A	1B	-	-	4A	5B	-	-	8A	9B	-	-	CA	DB	-	-
	0E	1F	-	-	4E	5F	-	-	8E	9F	-	-	CE	DF	-	-

Figure 4.6: Arrangement of AND of four registers with four $\mathbb{F}_{31}$ elements each.

Generating the quadratic monomials based on the given linear monomials requires $n \cdot \frac{n+1}{2}$ multiplications. For the second part, we require $m \cdot (n + n \cdot \frac{n+1}{2})$ multiplications to multiply the coefficients of the system parameter with the quadratic monomials, as well as a number of additions to accumulate all results.

For the initial part of the computation, we represent the monomials as 16-bit values. As just discussed, this is done to help prevent intermediate reductions. In the 3-pass case, we rotated a `YMM` vector register to efficiently compute all quadratic monomials (see Appendix 4.A, in particular Figure 4.A.2 and 4.A.3).[10] While that was already expensive, rotating four `YMM` registers as if it were one 1024-bit value is considerably more costly. Additionally, storing both the original and rotated state would require many registers. Instead, one can arrange the products in such a way that the blocks of 16 elements do not need to be mixed, but can each be rotated individually and multiplied with the unrotated originals. This is especially beneficial when computing **G**, in which we process 8 blocks of 256 bits. Two caveats appear in the first and last rows: duplicates need to be avoided when unrotated originals are combined in the first row, while the last (half-)row needs to be used to produce missing products by pairwise multiplying the high half of each register with every low half. See Figure 4.6 for an intuition of this arrangement with four registers containing four field elements. The software that is part of this work includes a script that generates this arrangement.

To efficiently compute all polynomials for a given set of monomials, we keep all required data in registers to avoid the cost of register spilling throughout the

---

[10] The interested reader is advised to briefly digress and review the implementation details of the 3-pass scheme before continuing into the remainder of this paragraph.

computation. Given that $n = m = 64$, for this part of the computation we represent the 64 input values in $\mathbb{F}_{31}$ as 8-bit values and the resulting 64 elements in $\mathbb{F}_{31}$ as 16-bit values, costing us two and four `YMM` registers, respectively. The coefficients of **F** can be represented as a column major matrix with every column containing all coefficients that correspond to a specific monomial, i.e. one for each output value. That implies that every row of the matrix represents one polynomial of **F**. In this representation, each result term is computed by accumulating the products of a row of coefficients with each monomial, which is exactly the same as computing the product of the matrix **F** and the vector containing all monomials. This allows us to efficiently accumulate output terms, minimizing the required output registers.

In order to perform the required multiplications and additions as quickly as possible, we heavily rely on the AVX2 instruction `vpmaddubsw`. In one instruction, this computes two 8-bit SIMD multiplications and a 16-bit SIMD addition. However, this instruction operates on 8-bit input values that are stored adjacently. This requires a slight variation on the representation of **F** described above: instead, we arrange the coefficients of **F** in a column major matrix with 16-bit elements, each corresponding to two concatenated monomials.

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: $\{0, \ldots, 30\}$). For the coefficients in the system parameter **F**, we can then freely assume the values are in $\{-15, \ldots, 15\}$, as these are the direct result of a pseudorandom generator. It turns out to be efficient to immediately reduce the quadratic monomials back to $\{0, \ldots, 30\}$ 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[11] $64 \cdot 31 \cdot 15 = 29760$. As this does not exceed 32768, we only have to perform reductions on each individual accumulator at the very end.

One should note that [CCC+09] approaches this problem from a slightly different angle. In particular, they accumulate each individual output element sequentially, allowing them to keep the intermediate results in the 32-bit representation that is the output of their combined multiplication and addition instructions. This has the natural consequence of also avoiding early reductions.

---

[11] This follows from the fact that we combine 64 such monomials in two `YMM` registers.

4.6.3 Performance

We now present benchmarks of our optimized MQDSS-31-64 implementation, targeting large Intel processors with AVX2 support. Measurements were carried out on a single core of the Intel Core i7-4770K system described in Section 2.4.1.

Signature and key sizes

The signature size of MQDSS-31-64 is considerably smaller than that of the 3-pass scheme. Recall that one field element is represented by 5 bits. The obvious factor in this is the decreased ratio between the vector size (which, in packed form, now require $n \cdot 5 = 64 \cdot 5 = 320$ bits each) and the number of rounds, resulting in a signature size of $2 \cdot k + r \cdot (k + 5 \cdot (2 \cdot n + m)) = 2 \cdot 256 + 269 \cdot (256 + 5 \cdot (2 \cdot 64 + 64)) = 327\,616$ bits, or 40 952 bytes (39.99 KiB). The structure of the keys does not change compared to 3-pass scheme, but since a vector of field elements now requires 320 bits, the public key is 72 bytes. The secret key remains 64 bytes.

Runtime

As the $\mathcal{MQ}$ function is the most costly part of the computation, parameters are chosen in such a way that its performance is maximized. The required number of multiplications and additions (expressed as functions of n and m) does not change dramatically compared to the 3-pass baseline,[12] but the actual values of n and m are only a quarter of what they were. As the relation between n and m and the number of multiplications is quadratic for the monomials and cubic for the system parameter masking, and we see only a linear increase in the number of registers needed to operate on, the entire sequence of multiplications and additions becomes much cheaper. This especially impacts operations that involve the accumulators. As the representation allows us to keep reductions out of this innermost repeated loop, we perform (only) $\frac{67 \cdot 4}{2} + 4 = 136$ reductions[13] throughout the main computation and 66 when preparing quadratic monomials. As we were able to arrange the registers in such a way that they do not need to rotate across multiple registers, we greatly reduce the number of rotations required compared to the 3-pass scenario. Following the same flattened-triangle structure, we count a total of $67 \cdot 16 \cdot 4 = 4288$

---

[12] A slight difference is introduced by cancellation of the monomials in the $\mathbb{F}_2$ setting.

[13] This follows from the fact that we need a total of $\frac{64+64 \cdot 65}{2 \cdot 32} = 67$ `YMM` registers worth of space to store the monomials and perform 4 reductions after accumulating 2 `YMM` monomials.

`vpmaddubsw` instructions for the core computations.

For one iteration of the $\mathcal{MQ}$ function **F**, we measure 6 616 cycles (**G** is slightly less costly, at 6 396 cycles). We measure a total of 8 510 616 cycles for the complete signature generation. Key generation costs 1 826 612 cycles, and verification consumes 5 752 612 cycles. On the given platform, that translates to roughly 2.43 ms, 0.52 ms and 1.64 ms, respectively. Verification is expected to require on average $\frac{3}{2}$ calls to an $\mathcal{MQ}$ function per round, while signature generation always requires two. This explains the ratio; note that both signer and verifier incur additional costs besides the $\mathcal{MQ}$ functions, e.g. for seed expansion.

In order to compare these results to the state of the art, we consider the performance figures reported in [CCC+09]. In particular, we examine the Rainbow(31, 24, 20, 20) instance, as the 'public map' in this scheme is effectively the $\mathcal{MQ}$ function over $\mathbb{F}_{31}$ with $n = 64$, as used above. The number of equations differs (i.e. $m = 40$ as opposed to $m = 64$), but this can be approximated by normalizing linearly. In [CCC+09], the authors report a time measurement of $17.7\mu s$, which converts to 50 144 cycles on their 2.833 GHz Intel C2Q Q9550. After normalizing for m, this amounts to 80 230 cycles. Results from the eBACS benchmarking project further show that running the Rainbow verification function from [CCC+09] on a Haswell CPU requires approximately 46 520 cycles (and thus 74 432 after normalizing); verification is dominated by the public map. Using their (by now arguably outdated) SSE2-based code to evaluate a public map with $m = 64$ consumes 60 968 cycles on our Intel Core i7-4770K. These results demonstrate the competitiveness of our implementation, even when considering the extensive use of AVX2 instructions.

4.6.4 The NIST submission

In November of 2017, the MQDSS signature scheme was submitted to the NIST's Post-Quantum Cryptography Standardization project (see Section 2.2). In January of 2019, NIST announced that MQDSS has progressed as one of the second-round candidates. The description of MQDSS as presented in the previous section of this chapter was largely based on the version defined in the ASIACRYPT 2016 paper *"From 5-pass MQ-based identification to MQ-based signatures"* [CHR+16]. Here, we review the current state of MQDSS as submitted to NIST.

Untouched by concrete advances in cryptanalysis, the scheme has largely remained the same. The security categories as defined by NIST do allow for a slight change in parameter choice: where the instance presented so far targets a 128-bit post-quantum security level, the lower NIST security categories allow for less conservative choices. This leads to improved runtime performance, and, perhaps more importantly, a significant reduction in signature size.

Parameters

We define two parameter sets as part of the submission. Targeting security category 1-2, we define MQDSS-31-48, where we choose $n = m = 48$. We stick to arithmetic over $\mathbb{F}_{31}$, but require only $r = 135$ rounds. As before, this follows from ${\frac{2q}{q+1}}^r < 2^{-k}$, and thus $r = \lceil k / \log \frac{2q}{q+1} \rceil$. Furthermore, all hash values are truncated to 256 bits, and we sample 128 bits for a seed. At category 3-4, MQDSS-31-64 uses an $\mathcal{MQ}$ system with $n = m = 64$ as described above, now with the reduced $r = 202$ (see Footnote 9 on page 138). In this category we require hash-function outputs of 384 bits, and 192-bit seeds. Note that this differs from the instance presented in [CHR+16], where we set both the hash and seed length to k. Here, we take a more conservative approach with respect to collision resistance of the hash function when estimating the post-quantum security level.

For simplicity, all functions previously instantiated using SHA3-256 and SHAKE-128 are now instantiated using SHAKE-256. As the inputs to these functions are invariably smaller than the 1088-bit rate of SHAKE-256, there is no performance penalty for making a more conservative choice. The proposed parameter sets are summarized in Table 4.1, below.

Randomized string commitment

Something we have silently assumed in the original MQDSS scheme [CHR+16] was the fact that the commitment functions are statistically hiding. We wrongfully assumed that the randomness of their inputs would be sufficient to make the commitment functions computationally hiding. A recent result by Leichtle [Lei18] shows that we require an additional randomization value of $2k$ bits as input to Com_0 and Com_1. We expand these values from a seed $\mathcal{S}_\rho$ that is derived from the secret key, and include them as part of the signature.

Other minor changes

Aside from the above, the submission to NIST contains a number of other additions and minor changes. This includes small tweaks such as explicitly deriving the seeds

Table 4.1: Key and signature sizes for the NIST parameter sets.

	category	k	q	n	r	pk	sk	signature
MQDSS-31-48	1-2	128	31	48	135	46	16	20 854
MQDSS-31-64	3-4	192	31	64	202	64	24	43 728

$\mathcal{S}_F$, $\mathcal{S}_{\mathrm{sk}}$, $\mathcal{S}_\rho$ and $\mathcal{S}_{\mathrm{rte}}$ from the secret key, absorbing the public key when deriving the message digest, but also a more accurate security analysis of the $\mathcal{MQ}$ problem and a broader exploration of the parameter space. Please refer to the specification as submitted to NIST for the details of the latest revision of the scheme.

Fiat-Shamir in the QROM

As mentioned in Section 4.5.1, the Fiat-Shamir transform lacks a proof in the QROM. In particular, the forking lemma that is traditionally used to prove its security [PS96] does not carry over to the QROM setting. Prior to the NIST submission, two methods of showing the security of the transform in the QROM have been proposed [DFG13; KLS18], both imposing additional assumptions that are not immediately obviously satisfied. Just before the deadline for second-round modifications, Don, Fehr, Majenz and Schaffner publicized a paper on ePrint [DFM+19] that claims to prove the Fiat-Shamir transform secure in the quantum random oracle model. Liu and Zhandry make similar claims [LZ19], and demonstrate concrete results by proving a lattice-based signature scheme secure. If these proofs generalize to the class of 5-pass identification schemes as presented above, these results may lead to a QROM proof that supports MQDSS.

Performance

In Table 4.1, we list the key and signature sizes for the proposed parameter sets. The public key is $k + n\lceil \log q \rceil$ bits, the secret key is k bits, and the signature adds up to $4k + r \cdot (4k + 3n\lceil \log q \rceil)$ bits. Note that this is slightly different from the computation presented in Section 4.6.3 because of the increased hash function length and the additional randomization string.

In Table 4.2 and Table 4.3, we present runtime benchmarks for key generation, signature generation and verification. The former contains performance numbers for the reference implementation in plain C as required by NIST, while the latter

Table 4.2: Runtime (in cycles) of the reference implementation.

	key generation	signing	verification
MQDSS-31-48	1 192 984	26 630 590	19 840 136
MQDSS-31-64	2 767 384	85 268 712	62 306 098

Table 4.3: Runtime (in cycles) of the AVX2 implementation.

	key generation	signing	verification
MQDSS-31-48	1 074 644	3 816 106	2 551 270
MQDSS-31-64	2 491 050	9 047 148	6 132 948

describes an implementation that makes use of the AVX2 instruction set. As before, benchmarks were obtained on an Intel Core i7-4770K (see also Section 2.4.1).

The AVX2 implementation relies heavily on the optimized evaluation of the $\mathcal{MQ}$ function over $\mathbb{F}_{31}$ described in Section 4.6.2. 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.

4.7 $\mathcal{MQ}$-based signatures in the QROM

In the previous sections, we have discussed MQDSS: an $\mathcal{MQ}$-based signature scheme with a security reduction from the $\mathcal{MQ}$ problem in the random oracle model. Unlike previous $\mathcal{MQ}$ signature schemes, MQDSS comes with a reduction from a random instance of $\mathcal{MQ}$. Unfortunately, this reduction remains highly non-tight. In the remainder of this chapter, we focus on SOFIA: a digital signature scheme that is provably EU-CMA secure in the QROM if the $\mathcal{MQ}$ problem is hard, and allows for a tight reduction in the ROM (albeit not in the QROM). Rather than approaching the herculean task of proving the Fiat-Shamir transform in the QROM, we start from a transform designed with the QROM in mind [Unr15].

Like MQDSS, SOFIA builds on the 5-pass $\mathcal{MQ}$-based IDS from [SSH11], as described in Section 4.3. While [SSH11] also introduces a 3-pass IDS, the reduced soundness error of the 5-pass scheme leads to smaller signatures. Consequently, in much the same spirit as the Fiat-Shamir transform for MQDSS, we do not simply apply Unruh's transform to the 3-pass IDS. Instead, we extend it such that it applies to any 5-pass IDS with a binary second challenge (dubbed $q2$-IDS, as per Definition 4.4.6), and thus to the $\mathcal{MQ}$-based 5-pass IDS from [SSH11]. Rather than simply relying on the generic transform, we provide various optimizations particularly suited for this specific IDS. These optimizations almost halve the size of the signature compared to the non-optimized transform. We discuss these in more detail in Section 4.8; some of these have already made an appearance in the algorithmic description of MQDSS.

We instantiate SOFIA with carefully optimized parameters targeting 128-bit post-quantum security level: SOFIA-4-128. A comparison with MQDSS-31-64 from [CHR+16] shows that, at the same security level, the improvements in security assumptions come at a cost: with 123 KiB, SOFIA-4-128 signatures are about three times as large as MQDSS-31-64 signatures and our optimized SOFIA-4-128 software takes about three times as long for both signing and verification than the software presented in [CHR+16] and discussed earlier in this chapter. However, like MQDSS, SOFIA features extremely short keys, making the sum of its public key and signature competitive with state-of-the-art, unproven, $\mathcal{MQ}$-based schemes. We discuss these implementation aspects in more detail in Section 4.9.

SOFIA is not the first concrete signature scheme with a proof in the QROM. Notably, TESLA-2 [ABB+15] is a lattice-based signature scheme with a reduction in the QROM, and Picnic-10-38 [CDG+17] is the result of constructing a signature scheme from a symmetric primitive using Unruh's transform.[14] Relying on even more conservative assumptions, the hash-based signature scheme SPHINCS-256 [BHH+15] (see also Sections 3.5 and 3.7) has a proof in the standard model.[15] Although SOFIA-4-128 remains faster than SPHINCS[16] (which is, because of its standard-model assumptions, arguably the 'scheme to beat'), it does significantly exceed SPHINCS' approximately 40 KiB signatures. Conversely, but on a similar note, SOFIA-4-128 outperforms Picnic-10-38 both in terms of signing speed and signature size. TESLA-2 remains the 'odd one out' with its small signatures but much larger keys; it strongly depends on context whether this is an upside or a problem. See Table 4.5 and Table 4.6 for a numeric overview of the comparison.

---

[14] Note that, for this work, we compare against Picnic as originally published at ACM CCS 2017. The submission to NIST's Post-Quantum Cryptography Standardization project also contains variants that make use of the Fiat-Shamir transform, and has improved parameters.

[15] At the time of writing, the security analysis of [BHK+19] demonstrating the multi-target tightness gap in SPHINCS-256 (and fixing it in SPHINCS$^+$) had not yet been done.

[16] For the sake of this comparison, the differences between SPHINCS-256 and the more recent SPHINCS$^+$ instances is not directly relevant.

Before introducing SOFIA in more detail, we first briefly examine the Unruh transform (and its application to $q2$-identification schemes) more generically.

4.7.1 Unruh's transform

In [Unr15], Unruh proposes a transform that turns 3-pass zero-knowledge proofs into non-interactive schemes, accompanied by a reduction in the QROM. This transform finds its basis in Fischlin's transform [Fis05], published ten years prior. Unruh then shows how to use his transform to obtain a signature scheme, which we now briefly review. Rather than relying on the forking lemma [PS96], which requires 'rewinding' the signer, the transform works by making the signer generate several candidate transcripts for a commitment. Here, a candidate transcript follows the notion of a transcript as introduced in Section 4.1: the commitment, challenge and response as exchanged between a prover and a verifier in the underlying interactive protocol. The challenges are randomly sampled from the challenge space, but unique across transcripts per commitment. To parallelize the protocol (as in Figure 4.2), this process is iterated for several initial commitments. Next, the signer 'blinds' all responses in the transcripts by applying a length-preserving one-way function. Like in the Fiat-Shamir transform, these commitment-response pairs are then used as input into a hash function to non-interactively sample a list of challenges. The resulting signature consists of all commitments with their blinded responses, as well as unblinded responses for all transcripts corresponding to the selected challenges.

The strategy underlying this transform assumes that, without knowledge of the secret key, a forger cannot include valid unblinded responses for sufficiently many commitments. In the context of a reductionist security proof, however, the length-preserving one-way function is replaced by a random permutation. This makes the blinding invertible, allowing for an adversary to recover multiple transcripts corresponding to the same commitment, and running an extractor.

Before being able to apply Unruh's transform to the $\mathcal{MQ}$-based identification scheme proposed in [SSH11], we must first extend it to operate on 5-pass identification schemes. Here, we again limit ourselves to the class of $q2$-identification schemes as introduced in Definition 4.4.6. The intuition here is to repeat the blinding process across both challenges; this requires candidate transcripts for the direct product of the two challenge spaces. As the second challenge is binary, this implies two candidate transcripts per initial challenge.

Extending Unruh's transform to $q2$-IDS

We define IDS = (KeyGen, $\mathcal{P}, \mathcal{V}$) to be a $q2$-identification scheme, with $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2)$ and $\mathcal{V} = (\mathsf{ChS}_1, \mathsf{ChS}_2, \mathsf{Verify})$. We further require system parameters $r, t \in \mathbb{N}$, with $2 \leqslant t \leqslant q$, which represent a trade-off between the security level, computation time and signature size. The selection of these parameters are discussed in somewhat more detail in Section 4.9.1.

Let $H_1 : \{0,1\}^{|\mathsf{resp}_1|} \to \{0,1\}^{|\mathsf{resp}_1|}$ and $H_2 : \{0,1\}^{|\mathsf{resp}_2|} \to \{0,1\}^{|\mathsf{resp}_2|}$ be length-preserving one-way functions, and $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{\lceil \log 2t \rceil r}$ a hash function, all of which to be modeled as random oracles. We then construct the digital signature scheme (KeyGen, Sign, Verify), resulting from the transform, as specified in Algorithms 22 and 23; the key generation algorithm is identical to key generation of the underlying identification scheme.

In [CHR+18], we provide a reductionist proof to show a quantum variant of EU-CMA security for the transformed scheme in the random oracle model. We then show that this proof carries over to the quantum random oracle model with only minor changes. This is considered out of scope for the purpose on this thesis.

4.8 SOFIA

Now that we have generically defined a signature scheme as the result of a transformed $q2$-IDS scheme, we instantiate it with the 5-pass identification scheme proposed in [SSH11]. We define the signature scheme by specifying the functions KeyGen, Sign and Verify, and defer giving concrete parameters (m, n, and $\mathbb{F}_q$, as well as r and t) to the next section, where we consider their effect on implementation aspects. There, we also explicitly instantiate the pseudorandom generators (G_{sk} and G_{rte}) and extendable output functions (XOF_F and $\mathsf{XOF}_{\mathsf{trans}}$). For now we only fix $2 \leqslant t \leqslant q$ distinct[17] elements of the field $\mathbb{F}_q$. Without loss of generality, we denote them as $\alpha_1, \ldots, \alpha_t$.

Key generation

The SOFIA key generation algorithm is almost identical to the key generation of MQDSS, as we have seen in Algorithm 19. We redefine it in Algorithm 24. Note, however, that the function definitions have slightly different domains and ranges.

---

[17] See the 'Fixing the challenge space' optimization later in this section.

Algorithm 22 Unruh.Sign $(\mathsf{m}, \mathsf{sk})$ $r, t, \mathsf{ChS}_1, \mathcal{H}, H_1, H_2, \mathcal{P}$

```
for $j \in \{1, \ldots, r\}$ do
    $\mathsf{state}^{(j)}, \mathsf{com}^{(j)} \xleftarrow{\$} \mathcal{P}_0(\mathsf{sk})$
    for $i \in \{1, \ldots, t\}$ do
        $\mathsf{ch}_1^{(i,j)} \xleftarrow{\$} \mathsf{ChS}_1 \setminus \{\mathsf{ch}_1^{(1,j)}, \ldots, \mathsf{ch}_1^{(i-1,j)}\}$
        $\mathsf{state}^{(i,j)}, \mathsf{resp}_1^{(i,j)} \leftarrow \mathcal{P}_1(\mathsf{state}^{(j)}, \mathsf{ch}_1^{(i,j)})$
        $\mathsf{cr}_1^{(i,j)} \leftarrow H_1(\mathsf{resp}_1^{(i,j)})$
        $\mathsf{resp}_2^{(i,j,0)} \leftarrow \mathcal{P}_2(\mathsf{state}^{(i,j)}, \mathsf{ch}_2 = 0)$
        $\mathsf{resp}_2^{(i,j,1)} \leftarrow \mathcal{P}_2(\mathsf{state}^{(i,j)}, \mathsf{ch}_2 = 1)$
        $\mathsf{cr}_2^{(i,j,0)} \leftarrow H_2(\mathsf{resp}_2^{(i,j,0)})$
        $\mathsf{cr}_2^{(i,j,1)} \leftarrow H_2(\mathsf{resp}_2^{(i,j,1)})$
    end for
    $\mathsf{trans}_{\mathsf{full}}^{(j)} = \left(\mathsf{com}^{(j)}, \left\{\mathsf{ch}_1^{(i,j)}, \mathsf{cr}_1^{(i,j)}, \mathsf{cr}_2^{(i,j,0)}, \mathsf{cr}_2^{(i,j,1)}\right\}_{i=1}^{t}\right)$
end for
$\mathsf{md} \leftarrow \mathcal{H}\left(\mathsf{pk}, \mathsf{m}, \left\{\mathsf{trans}_{\mathsf{full}}^{(j)}\right\}_{j=1}^{r}\right)$
$((I_1, B_1), \ldots, (I_r, B_r)) = \mathsf{md}$        ▷ Parse md such that $I_j \in \mathsf{ChS}_1$, $B_j \in \{0, 1\}$
for $j \in \{1, \ldots, r\}$ do
    $\mathsf{trans}_{\mathsf{red}}^{(j)} = \left(\mathsf{com}^{(j)}, \left\{\mathsf{ch}_1^{(i,j)}, \mathsf{cr}_1^{(i,j)}, \mathsf{cr}_2^{(i,j,0)}, \mathsf{cr}_2^{(i,j,1)}\right\}_{i \neq I_j, i=1}^{t}\right)$
end for
return $\sigma = \left(\mathsf{md}, \left\{\mathsf{trans}_{\mathsf{red}}^{(j)}, \mathsf{ch}_1^{(I_j,j)}, \mathsf{resp}_1^{(I_j,j)}, \mathsf{resp}_2^{(I_j,j,B_j)}, \mathsf{cr}_2^{(I_j,j,\neg B_j)}\right\}_{j=1}^{r}\right)$
```

Algorithm 23 Unruh.Verify $(\mathsf{m}, \sigma, \mathsf{pk})$ $\quad r, t, \mathsf{ChS}_1, H_1, H_2, \mathcal{H}, \mathsf{Verify}$

```
$\left(\mathsf{md}, \left\{\mathsf{trans}_{\mathsf{red}}^{(j)}, \mathsf{ch}_1^{(I_j,j)}, \mathsf{resp}_1^{(I_j,j)}, \mathsf{resp}_2^{(I_j,j,B_j)}, \mathsf{cr}_2^{(I_j,j,\neg B_j)}\right\}_{j=1}^{r}\right) = \sigma$
$((I_1, B_1), \ldots, (I_r, B_r)) = \mathsf{md}$        ▷ Parse md such that $I_j \in \mathsf{ChS}_1$, $B_j \in \{0, 1\}$
for $j \in \{1, \ldots, r\}$ do
    $\mathsf{cr}_1^{(I_j,j)} \leftarrow H_1(\mathsf{resp}_1^{(I_j,j)})$
    $\mathsf{cr}_2^{(I_j,j,B_j)} \leftarrow H_2(\mathsf{resp}_2^{(I_j,j,B_j)})$
    $\left(\mathsf{com}^{(j)}, \left\{\mathsf{ch}_1^{(i,j)}, \mathsf{cr}_1^{(i,j)}, \mathsf{cr}_2^{(i,j,0)}, \mathsf{cr}_2^{(i,j,1)}\right\}_{i \neq I_j, i=1}^{t}\right) = \mathsf{trans}_{\mathsf{red}}^{(j)}$
    $\mathsf{trans}_{\mathsf{full}}^{(j)} = \left(\mathsf{com}^{(j)}, \left\{\mathsf{ch}_1^{(i,j)}, \mathsf{cr}_1^{(i,j)}, \mathsf{cr}_2^{(i,j,0)}, \mathsf{cr}_2^{(i,j,1)}\right\}_{i=1}^{t}\right)$
end for
$\mathsf{md}' \leftarrow \mathcal{H}\left(\mathsf{pk}, \mathsf{m}, \left\{\mathsf{trans}_{\mathsf{full}}^{(j)}\right\}_{j=1}^{r}\right)$
if $\mathsf{md} \neq \mathsf{md}'$ then
    return False
end if
for $j \in \{1, \ldots, r\}$ do
    if $\exists_{i \in \{1, \ldots, r\}, i \neq I_j}, \mathsf{ch}_1^{(I_j,j)} = \mathsf{ch}_1^{(i,j)}$ then        ▷ If $\mathsf{ch}_1^{(I_j,j)}$ is not unique
        return False
    end if
    if $\neg\mathsf{Verify}(\mathsf{pk}, \mathsf{com}^{(j)}, \mathsf{ch}_1^{(I_j,j)}, \mathsf{resp}_1^{(I_j,j)}, B_j, \mathsf{resp}_2^{(I_j,j,B_j)})$ then
        return False
    end if
end for
return True
```

In particular, as we have not yet selected a specific field $\mathbb{F}_q$, we leave the exact output lengths of XOF_F and G_{sk} unspecified for now. As remarked in footnote 6 on page 132, we can reduce the secret key to a single seed by explicitly deriving $\mathcal{S}_F$.

Algorithm 24 SOFIA.KeyGen () $k, G_{\mathsf{sk}}, \mathrm{XOF}_F$

```
$\mathcal{S}_{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^k$
$\mathcal{S}_F, \mathbf{s}, \mathcal{S}_{\mathsf{rte}} \leftarrow G_{\mathsf{sk}}(\mathcal{S}_{\mathsf{sk}})$
$\mathbf{F} \leftarrow \mathrm{XOF}_F(\mathcal{S}_F)$
$\mathbf{v} \leftarrow \mathbf{F}(\mathbf{s})$
return $\mathsf{pk} = (\mathcal{S}_F, \mathbf{v}), \mathsf{sk} = \mathcal{S}_{\mathsf{sk}}$
```

Signing

For the signing procedure, we assume as input a message $\mathsf{m} \in \{0,1\}^*$ and a secret key sk. Note that the scheme definition includes several optimizations to reduce the signature size. We discuss these later in this section.

The signer begins by effectively performing KeyGen() to obtain pk and **F**, and subsequently iterates through r rounds of the transformed identification scheme, computing and blinding the responses to obtain the transcript. Instantiating $\mathcal{H}$ as required in Algorithm 22 using $\mathrm{XOF}_{\mathsf{trans}}$, they then derive a sequence of indices $((I_1, B_1), \ldots, (I_r, B_r))$. As before, these indices dictate the responses that should be included in the signature. See the full description in Algorithm 25.

Verification

Upon receiving a message m, a signature σ, and a public key $\mathsf{pk} = (\mathcal{S}_F, \mathbf{v})$, the verifier begins by obtaining the system parameter **F** and parsing the signature σ. The verification routine that follows is listed in Algorithm 26, resembling the generic template of 23. As a consequence of the preselected, fixed set of possible challenges $\{\alpha_i\} \subseteq \mathbb{F}_q$, the correctness checks are greatly simplified.[18] Furthermore, because of the commitment reconstruction by the verifier, the satisfaction conditions of Verify are checked implicitly when md and md′ are compared. This pattern is very similar in spirit to the comparison of σ_0 and σ_0' in the verification of MQDSS signatures as described in Algorithm 21 on page 135.

---

[18] See the next subsection for some more discussion on consequences of this.

Algorithm 25 SOFIA.Sign $(\mathsf{m}, \mathsf{sk})$ $\alpha, r, t, Com, G_{\mathsf{sk}}, G_{\mathrm{rte}}, \mathcal{H}, H_1, H_2, \mathrm{XOF}_F, \mathrm{XOF}_{\mathsf{trans}}$

```
$\mathcal{S}_{\mathsf{sk}} = \mathsf{sk}$
$\mathcal{S}_F, \mathbf{s}, \mathcal{S}_{\mathrm{rte}} \leftarrow G_{\mathsf{sk}}(\mathcal{S}_{\mathsf{sk}})$
$\mathbf{F} \leftarrow \mathrm{XOF}_F(\mathcal{S}_F)$
$\mathbf{v} \leftarrow \mathbf{F}(\mathbf{s})$
$\mathsf{pk} = (\mathcal{S}_F, \mathbf{v})$
$\mathbf{r}_0^{(1)}, \ldots, \mathbf{r}_0^{(r)}, \mathbf{t}_0^{(1)}, \ldots, \mathbf{t}_0^{(r)}, \mathbf{e}_0^{(1)}, \ldots, \mathbf{e}_0^{(r)} \leftarrow G_{\mathrm{rte}}(\mathcal{S}_{\mathrm{rte}}, \mathsf{m})$
**for** $j \in \{1, \ldots, r\}$ **do**
    $\mathbf{r}_1^{(j)} \leftarrow \mathbf{s}^{(j)} - \mathbf{r}_0^{(j)}$
    $c_0^{(j)} \leftarrow Com(\mathbf{r}_0^{(j)}, \mathbf{t}_0^{(j)}, \mathbf{e}_0^{(j)})$
    $c_1^{(j)} \leftarrow Com(\mathbf{r}_1^{(j)}, \mathbf{G}(\mathbf{t}_0^{(j)}, \mathbf{r}_1^{(j)}) + \mathbf{e}_0^{(j)})$
    $\mathsf{com}^{(j)} = (c_0^{(j)}, c_1^{(j)})$
    **for** $i \in \{1, \ldots, t\}$ **do**
        $\mathbf{t}_1^{(i,j)} \leftarrow \alpha_i \mathbf{r}_0^{(j)} - \mathbf{t}_0^{(j)}$
        $\mathbf{e}_1^{(i,j)} \leftarrow \alpha_i \mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_0^{(j)}$
        $\mathsf{resp}_1^{(i,j)} = (\mathbf{t}_1^{(i,j)}, \mathbf{e}_1^{(i,j)})$
        $\mathsf{cr}_1^{(i,j)} \leftarrow H_1(\mathsf{resp}_1^{(i,j)})$
    **end for**
    $\mathsf{resp}_2^{(j,0)} = \mathbf{r}_0^{(j)}$
    $\mathsf{resp}_2^{(j,1)} = \mathbf{r}_1^{(j)}$
    $\mathsf{cr}_2^{(j,0)} \leftarrow H_2(\mathsf{resp}_2^{(j,0)})$
    $\mathsf{cr}_2^{(j,1)} \leftarrow H_2(\mathsf{resp}_2^{(j,1)})$
    $\mathsf{trans}_{\mathsf{full}}(j) = \left(\mathsf{com}^{(j)}, \left\{\mathsf{cr}_1^{(i,j)}\right\}_{i=1}^{t}, \mathsf{cr}_2^{(j,0)}, \mathsf{cr}_2^{(j,1)}\right)$
**end for**
$\mathsf{md} \leftarrow \mathcal{H}\left(\mathsf{pk}, \mathsf{m}, \left\{\mathsf{trans}_{\mathsf{full}}^{(j)}\right\}_{j=1}^{r}\right)$
$((I_1, B_1), \ldots, (I_r, B_r)) \leftarrow \mathrm{XOF}_{\mathsf{trans}}(\mathsf{md})$
**for** $j \in \{1, \ldots, r\}$ **do**
    $\mathsf{trans}_{\mathsf{red}}^{(j)} = \left(c_{\neg B_j}^{(j)}, \left\{\mathsf{cr}_1^{(i,j)}\right\}_{i \neq I_j, i=1}^{t}, \mathsf{cr}_2^{(j, \neg B_j)}\right)$
**end for**
**return** $\sigma = \left(\mathsf{md}, \left\{\mathsf{trans}_{\mathsf{red}}^{(j)}, \mathsf{resp}_1^{(I_j,j)}, \mathsf{resp}_2^{(j,B_j)}\right\}_{j=1}^{r}\right)$
```

Algorithm 26 SOFIA.Verify $(\sigma, \mathsf{pk}, \mathsf{m})$ $\alpha, r, t, Com, \mathcal{H}, H_1, H_2, \mathrm{XOF}_{\mathsf{trans}}$

```
return $\left(\mathsf{md}, \left\{\mathsf{trans}^{(j)}_{\mathsf{red}}, \mathsf{resp}^{(I_j,j)}_1, \mathsf{resp}^{(j,B_j)}_2\right\}^r_{j=1}\right) = \sigma$
$(\mathcal{S}_F, \mathbf{v}) = \mathsf{pk}$
$\mathbf{F} \leftarrow \mathrm{XOF}_F(\mathcal{S}_F)$
$((I_1,B_1),\ldots,(I_r,B_r)) \leftarrow \mathrm{XOF}_{\mathsf{trans}}(\mathsf{md})$
for $j \in \{1,\ldots,r\}$ do
    $\mathsf{cr}_1^{(I_j,j)} \leftarrow H_1(\mathsf{resp}_1^{(I_j,j)})$
    $\mathsf{cr}_2^{(I_j,B_j)} \leftarrow H_2(\mathsf{resp}_2^{(I_j,B_j)})$
    if $B_j = 0$ then
        $\mathbf{r}_0^{(j)} = \mathsf{resp}_2^{(I_j,B_j)}$
        $c_0^{(j)} \leftarrow Com(\mathbf{r}_0^{(j)}, \alpha_{I_j}\mathbf{r}_0^{(j)} - \mathbf{t}_1^{(I_j,j)}, \alpha_{I_j}\mathbf{F}(\mathbf{r}_0^{(j)}) - \mathbf{e}_1^{(I_j,j)})$
    else
        $\mathbf{r}_1^{(j)} = \mathsf{resp}_2^{(I_j,B_j)}$
        $c_1^{(j)} \leftarrow Com(\mathbf{r}_1^{(j)}, \alpha_{I_j}(\mathbf{v} - \mathbf{F}(\mathbf{r}_1^{(j)})) - \mathbf{G}(\mathbf{t}_1^{(I_j,j)}, \mathbf{r}_1^{(j)}) - \mathbf{e}_1^{(I_j,j)})$
    end if
    $\left(c^{(j)}_{\neg B_j}, \left\{\mathsf{cr}_1^{(i,j)}\right\}^t_{i\neq I_j, i=1}, \mathsf{cr}_2^{(j,\neg B_j)}\right) = \mathsf{trans}^{(j)}_{\mathsf{red}}$
    $\mathsf{trans}^{(j)}_{\mathsf{full}} = \left(\mathsf{com}^{(j)}, \left\{\mathsf{ch}_1^{(i,j)}, \mathsf{cr}_1^{(i,j)}, \mathsf{cr}_2^{(i,j,0)}, \mathsf{cr}_2^{(i,j,1)}\right\}^t_{i=1}\right)$
end for
$\mathsf{md}' \leftarrow \mathcal{H}\left(\mathsf{pk}, \mathsf{m}, \{\mathsf{trans}_{\mathsf{full}}(j)\}^r_{j=1}\right)$
return $\mathsf{md}' \overset{?}{=} \mathsf{md}$
```

4.8.1 Tweaks and optimizations

There are several optimizations that can be applied to signatures resulting from a transformed $q2$-IDS. Some are specific for SOFIA and some are more general; similar and related optimizations were suggested in [Unr15], [CHR+16] and [CDG+17], and several of these optimizations were already implicitly included in MQDSS as described in Section 4.5. We now attempt to provide a comprehensive overview.

Excluding unnecessary blindings

The signature contains blindings of all computed responses, as well as a selection of opened responses $\mathsf{resp}_1^{(I_j,j)}$ and $\mathsf{resp}_2^{(j,B_j)}$. It is redundant to include the values $\mathsf{cr}_1^{(I_j,j)}$ and $\mathsf{cr}_2^{(j,B_j)}$, as these can be recomputed based on the opened responses. This optimization was already proposed in the generic Unruh transform [Unr15], and applies to any construction similar to Unruh's and ours. For the verifier to know which responses were actually opened, they must be able to reproduce the indices $((I_1, B_1), \ldots, (I_r, B_r))$, which are derived from the transcript. Without the blinded responses, this transcript is incomplete. To solve this circular dependency, we could include the selected indices in the signature. Depending on the choice of parameters (e.g., the parameters defined in 4.9.1), we can do this more efficiently by instead breaking $\mathsf{XOF}_{\mathsf{trans}}$ into two parts, composing it of a hash function over the transcript $\mathcal{H}$ and an extendable output function XOF_{IB} to derive the indices from the hash output. We then include $\mathcal{H}\left(\mathsf{pk}, \mathsf{m}, \{\mathsf{trans}_{\mathsf{full}}(j)\}_{j=1}^{r}\right)$ as part of the signature, so that the verifier can reconstruct the indices, blind the corresponding responses, construct $\mathsf{trans}_{\mathsf{full}}$, and recompute the same hash for comparison.

Fixing the challenge space

Following the generic description of the signing algorithm, the selected $\alpha^{(i,j)}$ are included in the signature. Depending on the specific choice of t and q, it may be more efficient to include the challenges $\alpha^{(i,j)}$ that were *not* selected. There is no reason not to take this a step further and simply fix a challenge space ch_1 of t elements. That way, all the α's from ch_1 will be selected and there is no need to include them in the signature. This not only reduces the signature size, but also simplifies the implementation. By preselecting these elements to be distinct and prescribing an ordering, we avoid the additional complexity of mapping indices to challenges, as well as checking for uniqueness in the verification routine.

Excluding unnecessary second responses

The underlying IDS from [SSH11] has a specific property: the second responses do not depend on the previous state (that is, on the first challenge and response). Regardless of the value of α, the second responses always have the same value. Consequently, they need to be included only once per commitment rather than repeated t times. Combined with the exclusion of unnecessary blindings, one of the one of the second responses is included open, and the other remains blinded.

Omitting commitments

The check that the verifier performs for each round consists of recomputing $c_{B_j}^{(j)}$, and comparing it to one of the commits supplied by the signer. Similar to the above, and as already suggested in [SSH11], the signer can omit the commits that the verifier will recompute. A hash over all commits could be included instead, which the verifier can reconstruct using the commits $c_{B_j}^{(j)}$ they recompute and the commits $c_{\neg B_j}^{(j)}$ the signer includes. As it turns out, this hash is not necessary either: as these commitments are part of the transcript and the verifier is already checking the correctness of the transcript as per the first optimization, the correctness of the recomputed commitments is implicitly checked when comparing md and md'.

While constructing this scheme, we experimented with several other variants of the above-described optimizations.[19] Notably, we explored opening for multiple α-challenges, but that led to no improvement in the number of rounds, and, in some cases, to a contradiction of the zero-knowledge property. Variants that employ a form of internal parallelization by committing to multiple values for $\mathbf{t}_0$ do reduce the number of rounds, but increase the size of the transcript disproportionately. Altogether, the above optimizations are crucial: for the practical instance described in the next section, they more than halve the signature size compared to the scheme that would result from naively applying the transform.

Security of SOFIA

In Section 4.7.1 we described an extension of Unruh's transform to $q2$-IDS. We left the proof of its EU-CMA security to the original publication [CHR+18], but

---

[19] In the paper describing SOFIA [CHR+18], we propose an optimization that suggests relying on the randomness of the inputs to the commitment functions instead of including a randomization string. As mentioned in Section 4.6.4, recent work by Leichtle [Lei18] shows that this is insufficient.

remark that it immediately applies to the scheme resulting from the 5-pass scheme from [SSH11]. After applying the optimizations described above, SOFIA deviates significantly from the direct output of the transform, and thus no longer adheres to the generic construction to which the proof applies. Fortunately, only minor changes to the proof are required. In [CHR+18], we enumerate the optimizations, and individually address how each of them impacts the proof. Using a game-hopping argument, one can show that the success probability of an EU-CMA adversary in the quantum random oracle model against SOFIA is negligibly close to the success probability of the same adversary against the unmodified direct result of the transform.

4.9 SOFIA-4-128

Having described the scheme in general terms, we now provide concrete parameters that allow us to specify a specific instance, which we will refer to as SOFIA-4-128. We present an optimized software implementation and list the results, in particular in comparison to MQDSS-31-64. All speed comparisons mentioned below are based on benchmarks obtained on the same Intel Core i7-4770K Haswell system described in Section 2.4.1, using gcc 4.9.2-10.

4.9.1 Parameter selection

The previous section assumed a number of parameters and functions. Notably, we must define $\mathbb{F}_q$, the field in which we perform the arithmetic, and n and m, the number of variables and equations defining the $\mathcal{MQ}$ problem. The number of rounds r follows from t (i.e. the number of responses $\text{resp}_1^{(i,j)}$, bounded by q) and the targeted security level.

Parameters for $\mathcal{MQ}(m, n, \mathbb{F}_q)$

For MQDSS-31-64, the choice of $\mathbb{F}_{31}$ was motivated by the fact that it brings the soundness error close to $1/2$ while providing convenient characteristics for fast implementation [CHR+16]. For SOFIA-4-128, our primary focus is on optimizing for signature size, although we do still try to strike a balance that maintains reasonable efficiency. To do so, we compute signature sizes for a wide range of candidates, and investigate several in more detail by implementing and measuring the resulting $\mathcal{MQ}$ evaluation functions. In particular, we look at the results of $\mathcal{MQ}(128, 128, \mathbb{F}_4)$,

$\mathcal{MQ}(96, 96, \mathbb{F}_7)$ and $\mathcal{MQ}(72, 72, \mathbb{F}_{16})$, and compare to $\mathcal{MQ}(64, 64, \mathbb{F}_{31})$ as used in MQDSS. Of these, $\mathcal{MQ}(128, 128, \mathbb{F}_4)$ is the decisive winner, resulting in the smallest signatures while still providing decent performance. This is also the minimum amongst all candidate systems we looked at – it is not only beating $\mathbb{F}_7$ and $\mathbb{F}_{16}$, but also less common options such as $\mathbb{F}_5$ and $\mathbb{F}_8$. See Table 4.4 for benchmarks of single evaluation functions and the related signature sizes. Note that, as the number of rounds r does not depend on the choice of $\mathbb{F}_q$ but merely on t, the signing time scales proportionally.

All of the suggestions for q, m and n mentioned above are chosen to target 128-bit post-quantum security. Like for MQDSS in Section 4.6.1, the estimates for the security of the $\mathcal{MQ}$ problem are based on the best known attacks.

A straightforward method for solving systems of m quadratic equations in n variables over $\mathbb{F}_q$ is by performing exhaustive search on all possible q^n values for the variables, and testing whether they satisfy the system. Currently, [BCC+10] provide the fastest enumeration algorithm for systems over $\mathbb{F}_2$, needing $4 \log n \cdot 2^n$ operations. The techniques from [BCC+10] can be extended to other fields $\mathbb{F}_q$ with the same expected complexity of $\Theta(\log_q n \cdot q^n)$.

In Section 4.6.1, we already touched upon some algebraic methods, including the F4/F5 family [Fau99; Fau02; BFS15; BFP12] and the variants of the XL algorithm [CKP+00; Die04; YC05a; YC04].

In the Boolean case, today's state of the art algorithms BooleanSolve [BFS+13] and FXL [YC04] 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 recent algorithm, the Crossbred algorithm [JV17] over $\mathbb{F}_2$, considerably improves on these bounds: the authors report that it passes the exhaustive search barrier already for 37 Boolean variables. At the time of this work, the preprint did not include a detailed complexity analysis that we could take into account.

BooleanSolve [BFS+13], FXL [YC04; YC05a], the Crossbred algorithm [JV17] and the Hybrid approach [BFP12] all combine algebraic techniques with exhaustive search. This immediately allows for improvement in their quantum version using Grover's quantum search algorithm [Gro96], provided the cost of running them on a quantum computer does not diminish the gain from Grover. At the time of this work, no quantum version of these algorithms had been analyzed yet, and the literature only covered pure enumeration [WS16] using Grover. Since then,

Table 4.4: Benchmarks for varying parameter sets

	cycles[b]	size $t = 3, r = 438$	size $t = 4, r = 378$
$\mathcal{MQ}(128, 128, \mathbb{F}_4)$	21 412	123.22 KiB	129.97 KiB
$\mathcal{MQ}(96, 96, \mathbb{F}_7)$	36 501	129.00 KiB[a]	136.20 KiB[a]
$\mathcal{MQ}(72, 72, \mathbb{F}_{16})$	25 014	136.91 KiB	144.73 KiB
$\mathcal{MQ}(64, 64, \mathbb{F}_{31})$	6 616	149.34 KiB[a]	158.15 KiB[a]

[a] Assumes optimally packing the elements of $\mathbb{F}_q$, which may be impractical.

[b] For a single evaluation. In practice, batching provides a speedup; see Section 4.9.3.

Bernstein and Yang have instantiated the XL approach with Grover and conclude that the two can be meaningfully combined [BY18]. In the original publication of SOFIA [CHR+18], we provide an analysis of the quantum version of the Hybrid approach and BooleanSolve. It turns out that the results by Bernstein and Yang do not affect our choice of parameters.

Number of rounds and blinded responses

The choice of t, the number of blinded responses per round, provides a trade-off between size and speed: a larger t implies a smaller error, resulting in fewer rounds, but more included blinded responses per round.[20] Interestingly, $t = 3$ provides the minimal size, followed by $t = 4$, and, only then, $t = 2$. The decrease in rounds quickly diminishes, making $t = 3$ and $t = 4$ the most attractive choices. Note that t is naturally bounded by q, making these the *only* options for $\mathbb{F}_4$.

Given the above considerations (and with a prospect of some convenience of implementation), we select the parameters $n = m = 128$, $q = 4$ and $t = 3$. We then set the number of rounds r such that a forger has success probability negligible in the security parameter. For a security level of 128 bits post-quantum security, it follows from Theorem 3.6 of [CHR+18] that we must select r such that $2^{-(r \log \frac{2t}{t+1})/2} < 2^{-128}$. This implies $r = 438$.

---

[20] The increase in computational cost that comes with additional blinded responses is insignificant in comparison to evaluations of the $\mathcal{MQ}$ function.

Functions

Before being able to implement the scheme, we must still define several of the functions we have assumed to exist. In particular, we need a string commitment function *Com*, pseudorandom generators G_{sk} and G_{rte}, extendable output functions XOF_F and XOF_{IB}, permutation functions H_1 and H_2, and a cryptographic hash function $\mathcal{H}$.

We instantiate the extendable output functions, the string commitment functions,[21] the permutations and the hash function with SHAKE-128 [BDP+11]. This applies trivially, except for XOF_{IB}, of which the output domain is a series of ternary and binary indices (since $t = 3$). We resolve this by applying rejection sampling to the output of SHAKE-128. Note that this does not enable a timing attack, as the input to SHAKE-128 (i.e., md, the digest of the transcript) is public. For XOF_F, we achieve a significant speedup by dividing its output in four separate pieces, generating each of them with a domain-separated call to cSHAKE-128 [BDP+11]. For the application of $\mathcal{H}$ to the public key, the message and the transcript, collision resilience is achieved by absorbing the transcript into the SHAKE-128 state first, as the included randomness prevents internal collisions.

In principle we suggest to also instantiate G_{rte} and G_{sk} with SHAKE-128, but note that implementations can make different choices without breaking compatibility. In fact, for the optimized Haswell implementation discussed in the next section, we instantiate G_{rte} with AES in counter mode, using the AES-NI instruction set.

4.9.2 Implementation details

As part of this work, we provide a C reference implementation and an implementation optimized for AVX2. The focus of this section is the evaluation of the $\mathcal{MQ}$ function, given the aforementioned parameter set $\mathcal{MQ}(128, 128, \mathbb{F}_4)$. The rest of the scheme depends on fairly straight-forward operations (such as multiplying vectors of $\mathbb{F}_4$ elements by a constant scalar) and applications of existing implementations of AES-CTR and SHAKE-128. The used AES-CTR and SHAKE-128 implementations are in the public domain and run in constant time.

Before discussing the computation, we note that the chosen parameters lend themselves to a very natural data representation. Throughout the entire scheme,

---

[21] At the time of writing, we, as was the case for MQDSS, assumed that the inputs to the commitments provided sufficient randomization. This does not take into account the recent work by Leichtle [Lei18] yet. See also 4.6.4.

we interpret 256-bit vectors as vectors of 128 bitsliced $\mathbb{F}_4$ elements: the low 128 bits make up the lower bits of the two-bit elements, and the high 128 bits make up the higher bits of each element. This makes operations such as scalar multiplication convenient in C code, as this can be easily expressed as logical operations on bit sequences, but provides an even more important benefit for AVX2 assembly code. Notably, one vector of $\mathbb{F}_4$ elements fits exactly into one 256-bit vector register, with the lower bits now fitting into the low lane and the higher bits into the high lane. Other parameter sets might have resulted in having to consider crossing the lanes, but for these dimensions the separation fits perfectly.

When sampling elements in $\mathbb{F}_4$ from the output of SHAKE-128 or AES-CTR, we can freely interpret the random data to already be in bitsliced representation. Similarly, we include the elements in the signature in this representation, as signature verification enjoys precisely the same benefits. Throughout the entire scheme, there is no point at which we need to actually perform a bitslicing operation.

As a side effect of this choice of representation, the naive approach to performing the $\mathcal{MQ}$ evaluation runs in constant time. While bigger underlying fields might have implied approaches based on lookup tables, for vectors over $\mathbb{F}_4$ it is hard to imagine a platform on which it would not be faster to perform the evaluation using bitsliced field arithmetic.

Evaluating $\mathcal{MQ}$

For a given input $\mathbf{x}$, we split the evaluation into two phases: computing all quadratic monomial terms $x_i x_j$, and composing them to evaluate the quadratic polynomials.

To perform the first step, we use a similar approach as described in Sections 4.6.2 and 4.A.2. It can be seen as a combination of the approach for $\mathbb{F}_2$ and for $\mathbb{F}_{31}$, as we now operate on a single register that contains all input elements but view each lane as 16 separate single-byte registers. We use `vpshufb` instructions to arrange the elements such that all multiplications can be performed using only a minimal number of rotations. To do this, we modify the script to generate arrangements that was described in Section 4.6.2.

A bitsliced multiplication in $\mathbb{F}_4$ can be efficiently performed using only a few logical operations. The inputs to these multiplications are a register containing $\mathbf{x}$ and a register containing some rotated arrangement of $\mathbf{x}$. Some of these operations require the low and high lanes of the vector registers to interact, which is typically costly. As $\mathbf{x}$ is constant, we speed up these multiplications by rewriting them as shown below, combined with presetting two registers that contain $[\mathbf{x}_{high}|\mathbf{x}_{high}]$ and $[\mathbf{x}_{high} \oplus \mathbf{x}_{low}|\mathbf{x}_{low}]$, respectively. Note that all of these operations are not performed on single bits, but rather on 128-bit vector lanes. The multiplication of 128 elements then requires only two `vpand` instructions and one `vperm` instruction, and a `vpxor` to combine the results.

$$\begin{aligned} c_{high} &= (a_{high} \wedge (b_{low} \oplus b_{high})) \oplus (a_{low} \wedge b_{high}) \\ c_{low} &= (a_{low} \wedge b_{low}) \oplus (a_{high} \wedge b_{high}) \end{aligned}$$

We focus on two approaches to perform the second and most costly part of the evaluation, in which all of the above monomials need to be multiplied with coefficients from **F** and added into the output vector. They are best described as iterating either 'horizontally' or 'vertically' through the required multiplications. For the vertical approach, we iterate over all[22] registers of monomials, broadcasting each of the monomials to each of the 128 possible positions (using rotations), before multiplying with a sequence of coefficients from **F** and adding into an accumulator. Alternatively, we iterate over the output elements in the outer-most loop. For each output element, we iterate over all registers of monomials, perform the multiplications, and horizontally sum the results using the `popcnt` instruction.

Intuitively, the latter approach may seem like more work, in particular because it requires more loads from memory, but in practice it turns out to be faster for our parameters. The main reason for this is that by maintaining multiple separate accumulators, loaded monomials can be re-used while still maintaining chains of logic operations that operate on independent results, as the accumulators are only joined together later. This leads to highly efficient scheduling.

For both approaches, delaying part of the multiplication in $\mathbb{F}_4$ provides a significant speedup. This is done by computing both $[\hat{\mathbf{x}}_{high} \wedge \mathbf{f}_{high}|\hat{\mathbf{x}}_{low} \wedge \mathbf{f}_{low}]$ and $[\hat{\mathbf{x}}_{low} \wedge \mathbf{f}_{high}|\hat{\mathbf{x}}_{high} \wedge \mathbf{f}_{low}]$, with $\mathbf{f}$ from **F** and $\hat{\mathbf{x}}$ a sequence of quadratic monomials, and accumulating these results separately. After accumulating, all multiplications and reductions can be completed at once, eliminating duplicate operations that would otherwise be performed for each of the 65 multiplications.

---

[22] There are $\frac{n \cdot (n+1)}{2} = 8256$ such monomials, which results in $64\frac{1}{2}$ 256-bit sequences. We round up to 65 by zeroing out half of the high and half of the low lane. To still get results that are compatible with implementations on other platforms, we create similar gaps in the stream of random values used to construct **F**, ensuring that the same random elements are aligned with the original coefficients.

Evaluating $\mathcal{MQ}$ instances in parallel

As each of the coefficients in **F** is used only once, loading these elements from memory causes a considerable burden. We observe that **F** is constant *across* evaluations, leading to a significant speedup when processing multiple instances of the $\mathcal{MQ}$ function at once. This applies in particular to the vertical approach, as its critical section leaves several registers unused. Horizontally, there is a trade-off with registers used for parallel accumulators, but we still benefit from parallelizing evaluations. For SOFIA-4-128, the signer evaluates $r = 438$ instances of **F** and its polar form **G** on completely independent inputs, which can be trivially batched.

Parallel SHAKE-128 and cSHAKE-128

As will be apparent in the next section, many cycles are spent computing the Keccak permutation (as part of either SHAKE-128 or cSHAKE-128). Some of the main culprits are the commitments, the blinding of responses and the expansion of **F**. While the Keccak permutation does not lend itself well to internal parallelism, it is straightforward to compute four instances in parallel in a 256-bit vector register. This allows us to seriously speed up the many commitments and blindings, as these are all fully independent and can be grouped together across rounds. Deriving **F** can be parallelized by splitting it into four domain-separated cSHAKE-128 calls operating on the same seed, as was alluded to in Section 4.9.1.

4.9.3 Performance

Evaluating the $\mathcal{MQ}$ function horizontally in batches of three turns out to give the fastest results, measuring in at 17 558 cycles per evaluation. Evaluating vertically costs 18 598 cycles. The cost for evaluating the polar form **G** is not significantly different, differing by approximately a hundred cycles from the regular $\mathcal{MQ}$ function; generating the monomial terms $x_i y_j + x_j y_i$ is somewhat more costly, but this is countered by the fact that the linear terms cancel out.

We spend 21 305 472 cycles to generate a signature. Of this, 15 420 520 cycles can be attributed to evaluating $\mathcal{MQ}$, and 43 954 to AES-CTR. The remainder is almost entirely accounted for by the various calls to SHAKE-128 and cSHAKE-128 for the commitments, blindings and randomness expansion. In particular, expanding **F** costs 1 120 782 cycles. Note, however, that if many signatures are to be generated, this expansion only needs to be done once and **F** can be kept in memory across

Table 4.5: Benchmark overview

	key generation	signing	verification
SOFIA-4-128	1 157 112	21 305 472	15 492 686
MQDSS-31-64	1 826 612	8 510 616	5 752 612
SPHINCS-256[a]	3 237 260	51 636 372	1 451 004
Picnic-10-38	≈36 000	≈112 716k	≈58 680 000
TESLA-1	?[b]	143 402 231	19 284 672

[a] Benchmarked on an Intel Xeon E3-1275 (Haswell).

[b] The benchmarks in [ABB+17] omit key generation. In [CDG+17], a measurement of approximately 173 billion cycles is reported for the smaller TESLA-768 [ABB+15] scheme.

Table 4.6: Overview of key and signature sizes

	pk	sk	signature
SOFIA-4-128	64	32	126 176
MQDSS-31-64	72	64	40 952
SPHINCS-256	1056	1088	41 000
Picnic-10-38	64	32	195 458
TESLA-1	11 653k	6 769k	2 444
TESLA-2	≥21 799k[a]	≥7 700k[a]	≥4.0k[a]

[a] *"Sizes are theoretic sizes for fully compressed keys and signatures"* [ABB+17].

subsequent signatures. Verification costs 15 492 686 cycles, and key generation costs 1 157 112; key generation is dominated by expansion of $\mathbf{F}$.

The keys of SOFIA-4-128 are small, with the secret key consisting of only a single 32 byte seed, and the 64 byte public key of a seed and a single $\mathcal{MQ}$ output.

The natural candidate for direct comparison is MQDSS-31-64 [CHR+16]. While MQDSS has a proof in the ROM, we focus further comparison on post-quantum schemes that have proofs in the QROM or standard model. See Table 4.5 and Table 4.6, below; we include SPHINCS-256 [BHH+15], which has a proof in the standard model, and Picnic-10-38 [CDG+17] and TESLA-2 [ABB+17], which come with proofs in the QROM. Since [ABB+17] does not implement the TESLA-2 parameter set, we include the ROM variant, TESLA-1, for context. Except for SPHINCS-256, all benchmarks were obtained on an Intel Core i7 (Haswell).

Appendices to Chapter 4

4.A The 3-pass scheme over $\mathbb{F}_2$

In this appendix, we discuss the result of applying the Fiat-Shamir transform to the 3-pass IDS introduced in [SSH11] (see Figure 4.A.1).

As the resulting scheme is similar to the transformed 5-pass scheme (MQDSS) described in Section 4.5 in many ways, we occasionally refer back to that description to prevent needless redundancy and duplication.

The scheme is parameterized by $k, m, n \in \mathbb{N}$ in much the same way as MQDSS. While the 5-pass variant required several functions that ranged over $\mathbb{F}_{31}$, however, we can suffice with a slightly simpler setup for the 3-pass scheme. We require the following functions:

- Cryptographic hash functions $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^k$ and $H_1 : \{0,1\}^k \times \{0,1\}^k \rightarrow \{0,1,2\}^r$,
- a string commitment function $Com : \mathbb{F}_2^n \times \mathbb{F}_2^m \rightarrow \{0,1\}^k$, and
- pseudorandom generators $G_F : \{0,1\}^k \rightarrow \mathbb{F}_2^{F_{len}}$ and $G_{\text{rte}} : \{0,1\}^k \times \{0,1\}^k \rightarrow \mathbb{F}_2^{r\cdot(2n+m)}$.

Key generation

As before, we randomly sample $\mathcal{S}_{\text{sk}} \xleftarrow{\$} \{0,1\}^k$ and $\mathcal{S}_F \xleftarrow{\$} \{0,1\}^k$. Given parameters $n, m \in \mathbb{N}$, we expand $\mathcal{S}_F$ to obtain $\mathbf{F} = G_{\mathcal{S}_F}(\mathcal{S}_F)$, a random system from $\mathcal{MQ}(n, m, \mathbb{F}_2)$. The equation system $\mathbf{F}$ is defined by $F_{len} = m \cdot \frac{n\cdot(n+1)}{2}$ elements[23] in $\mathbb{F}_2$. We then apply $\mathbf{F}$ to $\mathcal{S}_{\text{sk}}$ to obtain the rest of the public key, $\mathbf{v} = \mathbf{F}(\mathcal{S}_{\text{sk}})$. The key generation algorithm outputs $\text{sk} = (\mathcal{S}_{\text{sk}}, \mathcal{S}_F)$ and $\text{pk} = (\mathcal{S}_F, \mathbf{v})$ as the key pair.

---

[23] As we are computing over $\mathbb{F}_2$, we have $x_i \cdot x_i = x_i \wedge x_i = x_i$. This allows us to merge the linear monomial terms into the quadratics, reusing $n \cdot m$ field elements in $\mathbf{F}$.

$$\begin{array}{lcl}
\mathcal{P} & & \mathcal{V} \\
\hline
\mathbf{r}_0, \mathbf{t}_0 \xleftarrow{\$} \mathbb{F}_q^n & & \\
\mathbf{e}_0 \xleftarrow{\$} \mathbb{F}_q^m & & \\
\mathbf{r}_1 \leftarrow \mathbf{s} - \mathbf{r}_0 & & \\
\mathbf{t}_1 \leftarrow \mathbf{r}_0 - \mathbf{t}_0 & & \\
c_0 \leftarrow Com(\mathbf{r}_1, \mathbf{G}(\mathbf{t}_0, \mathbf{r}_1) + \mathbf{e}_0) & & \\
c_1 \leftarrow Com(\mathbf{t}_0, \mathbf{e}_0) & & \\
c_2 \leftarrow Com(\mathbf{t}_1, \mathbf{e}_1) & & \\
 & \xrightarrow{(c_0, c_1, c_2)} & \\
 & & \mathrm{ch} \xleftarrow{\$} \{0, 1, 2\} \\
 & \xleftarrow{\mathrm{ch}} & \\
\text{If ch} = 0,\ \text{resp} \leftarrow (\mathbf{r}_0, \mathbf{t}_1, \mathbf{e}_1) & & \\
\text{If ch} = 1,\ \text{resp} \leftarrow (\mathbf{r}_1, \mathbf{t}_1, \mathbf{e}_1) & & \\
\text{If ch} = 2,\ \text{resp} \leftarrow (\mathbf{r}_1, \mathbf{t}_0, \mathbf{e}_0) & & \\
 & \xrightarrow{\text{resp}} & \\
 & & \text{If ch} = 0,\ \text{Parse resp} = (\mathbf{r}_0, \mathbf{t}_1, \mathbf{e}_1),\ \text{check} \\
 & & \quad c_1 \overset{?}{=} Com(\mathbf{r}_0 - \mathbf{t}_1, \mathbf{F}(\mathbf{r}_0) - \mathbf{e}_1) \\
 & & \quad c_2 \overset{?}{=} Com(\mathbf{t}_1, \mathbf{e}_1) \\
 & & \text{If ch} = 1,\ \text{Parse resp} = (\mathbf{r}_1, \mathbf{t}_1, \mathbf{e}_1),\ \text{check} \\
 & & \quad c_0 \overset{?}{=} Com(\mathbf{r}_1, \mathbf{v} - \mathbf{F}(\mathbf{r}_1) - \mathbf{G}(\mathbf{t}_1, \mathbf{r}_1) - \mathbf{e}_1) \\
 & & \quad c_2 \overset{?}{=} Com(\mathbf{t}_1, \mathbf{e}_1) \\
 & & \text{If ch} = 2,\ \text{Parse resp} = (\mathbf{r}_1, \mathbf{t}_0, \mathbf{e}_0),\ \text{check} \\
 & & \quad c_0 \overset{?}{=} Com(\mathbf{r}_1, \mathbf{G}(\mathbf{t}_0, \mathbf{r}_1) + \mathbf{e}_0) \\
 & & \quad c_1 \overset{?}{=} Com(\mathbf{t}_0, \mathbf{e}_0)
\end{array}$$

Figure 4.A.1: The [SSH11] 3-pass IDS

Signing

The signature algorithm takes as input a message $\mathsf{m} \in \{0,1\}^*$ and a secret key $\mathsf{sk} = (\mathcal{S}_{\mathsf{sk}}, \mathcal{S}_F)$. The message-dependent randomness R and digest md are derived in the same way as was done for the 5-pass scheme, and $\mathbf{F} = G_{\mathcal{S}_F}(\mathcal{S}_F)$ as during key generation. Let r be the required number of rounds.

The pair $(\mathcal{S}_{\mathsf{sk}}, \mathsf{md})$ is expanded using G_{rte} to produce the values $(\mathbf{r}_0^{(1)}, \dots, \mathbf{r}_0^{(r)}, \mathbf{t}_0^{(1)}, \dots, \mathbf{t}_0^{(r)}, \mathbf{e}_0^{(1)}, \dots, \mathbf{e}_0^{(r)})$. We then compute $c_0^{(i)}$, $c_1^{(i)}$ and $c_2^{(i)}$ using the string commitment function Com, as follows:

$$\begin{aligned} c_0^{(i)} &= Com(\mathbf{r}_1^{(i)}, \mathbf{G}(\mathbf{t}_0^{(i)}, \mathbf{r}_1^{(i)}) + \mathbf{e}_0^{(i)}) \\ c_1^{(i)} &= Com(\mathbf{t}_0^{(i)}, \mathbf{e}_0^{(i)}) \\ c_2^{(i)} &= Com(\mathbf{t}_1^{(i)}, \mathbf{e}_1^{(i)}) \end{aligned}$$

Let $\sigma_0 = \mathcal{H}(c_0^{(1)} \| c_1^{(1)} \| c_2^{(1)} \| \dots \| c_0^{(r)} \| c_1^{(r)} \| c_2^{(r)})$. We now derive the challenges ch_i from the pair (md, σ_0) using H_1. Then, for each of the rounds, we include the responses to each challenge, i.e., $(\mathbf{r}_0^{(i)}, \mathbf{t}_1^{(i)}, \mathbf{e}_1^{(i)})$, $(\mathbf{r}_1^{(i)}, \mathbf{t}_1^{(i)}, \mathbf{e}_1^{(i)})$ or $(\mathbf{r}_1^{(i)}, \mathbf{t}_0^{(i)}, \mathbf{e}_0^{(i)})$, respectively), in σ_1. For each round, σ_1 must also contain the one commitment the verifier cannot recompute: $c_{\mathsf{ch}^{(i)}}$. The resulting signature is $\sigma = (R, \sigma_0, \sigma_1)$, for a total of $2 \cdot k + r \cdot (2 \cdot n + m + k)$ bits.

Verification

The verification algorithm takes as input the message m, the signature $\sigma = (R, \sigma_0, \sigma_1)$ and the public key $\mathsf{pk} = (\mathcal{S}_F, \mathbf{v})$.

As for MQDSS, the verifier uses R and m to compute $\mathsf{md} = \mathcal{H}(R \parallel \mathsf{m})$ and derives $\mathbf{F}$ from $\mathcal{S}_F$ using $G_{\mathcal{S}_F}$, which is available in pk. Once more mimicking the signing procedure, the verifier can now derive $\mathsf{ch}^{(i)}$ for all r rounds using H_1 and the pair (md, σ_0). They then extract $(\mathbf{r}^{(i)}, \mathbf{t}^{(i)}, \mathbf{e}^{(i)})$ from σ_1, and, depending on the values of $\mathsf{ch}^{(i)}$, computes two thirds of the committed values as follows:

$$\text{if } \mathsf{ch}^{(i)} = 0 \begin{cases} c_1^{(i)} = Com(\mathbf{r}^{(i)} - \mathbf{t}^{(i)}, \mathrm{F}(\mathbf{r}^{(i)}) - \mathbf{e}^{(i)}) \\ c_2^{(i)} = Com(\mathbf{t}^{(i)}, \mathbf{e}^{(i)}) \end{cases}$$

$$\text{if } \mathsf{ch}^{(i)} = 1 \begin{cases} c_0^{(i)} = Com(\mathbf{r}^{(i)}, \boldsymbol{PK_v} - \mathrm{F}(\mathbf{r}^{(i)}) - \mathrm{G}(\mathbf{t}^{(i)}, \mathbf{r}^{(i)}) - \mathbf{e}^{(i)}) \\ c_2^{(i)} = Com(\mathbf{t}^{(i)}, \mathbf{e}^{(i)}) \end{cases}$$

$$\text{if } \mathsf{ch}^{(i)} = 2 \begin{cases} c_0^{(i)} = Com(\mathbf{r}^{(i)}, \mathrm{G}(\mathbf{t}^{(i)}, \mathbf{r}^{(i)}) + \mathbf{e}^{(i)}) \\ c_1^{(i)} = Com(\mathbf{t}^{(i)}, \mathbf{e}^{(i)}) \end{cases}$$

For each round, the other commitments $c^{(i)}_{\mathsf{ch}^{(i)}}$ can be extracted from σ_1, allowing the verifier to compute $\sigma_0' = \mathcal{H}(c_0^{(1)} \| c_1^{(1)} \| c_2^{(1)} \| \dots \| c_0^{(r)} \| c_1^{(r)} \| c_2^{(r)})$. If $\sigma_0' = \sigma_0$, verification succeeds.

4.A.1 Parameter selection

As in the case of the 5-pass scheme we motivate our choice of parameters both from security and from implementation point of view. First of all, the security arguments for $n = m$ are the same as for the 5-pass scheme. From an implementation point of view, choosing $n = m$ as a power of two provides various benefits. Most notably, the fact that we operate over a binary field in combination with a number of elements that neatly fit typical registers greatly enhances the ease of using bitwise operations. Therefore, we choose $n = 256$.

In terms of classical security, we can use the known generic algorithms for solving systems of quadratic equations [Fau99; Fau02; CKP+00], or the more recently proposed BooleanSolve [BFS+13], crafted specifically for the Boolean case. This algorithm performs similar to XL over $\mathbb{F}_2$. Indeed, the analysis in [YC04; YCC04b] and that of [BFS+13] both show that for large enough systems (i.e. where $n \geqslant 200$) it is possible to outperform exhaustive search in terms of time complexity. When $n = m$, the asymptotic complexity of the FXL algorithm is $\mathcal{O}(2^{0.875n})$ [YC04], and of the BooleanSolve algorithm it is $\mathcal{O}(2^{0.841n})$ in the deterministic variant and $\mathcal{O}(2^{0.792n})$ in the probabilistic variant [BFS+13]. At $n = 256$, this would lead to a computation complexity of 2^{202} operations. Using the same reasoning as for the 5-pass scheme, the complexity of Grover's algorithm will be $\approx 2^{128}$ operations.

As was also mentioned in Section 4.6.1, we aim for $\kappa^r \leq 2^{-256}$ in order to achieve 128 bits of post-quantum security. Thus, given the soundness error $\kappa = \frac{2}{3}$, we must perform $r = 438$ rounds of the transformed identification scheme.[24]

As before, we choose SHA3-256 for the functions $\mathcal{H}$ and *Com*. For H_1, $G_{\mathcal{S}_F}$ and G_{rte}, we select the SHAKE-128 extendable output function [BDP+11]. To map to the appropriate output domain, i.e., sample ternary challenges, we reject and resample unusable output from H_1.

---

[24] In retrospect, this number of rounds is excessive. See Footnote 9 on page 138.

&	0	1	2	3	4	5	6	7
0	00	01	02	03	04	05	06	07
1	-	11	12	13	14	15	16	17
2	-	-	22	23	24	25	26	27
3	-	-	-	33	34	35	36	37
4	-	-	-	-	44	45	46	47
5	-	-	-	-	-	55	56	57
6	-	-	-	-	-	-	66	67
7	-	-	-	-	-	-	-	77

Figure 4.A.2: Naive AND of 8 bits

&	0	1	2	3	4	5	6	7
	00	11	22	33	44	55	66	77
	10	21	32	43	54	65	76	07
	20	31	42	53	64	75	06	17
	30	41	52	63	74	05	16	27
	40	51	62	73	-	-	-	-

Figure 4.A.3: Efficient AND of 8 bits

4.A.2 Implementation details

We operate on $n = 256$ elements from $\mathbb{F}_2$, which means that an input vector $\mathbf{x}$ fits precisely in one 256-bit SIMD vector register. In essence, the computation of $\mathbf{F}$ comes down to computing the product of all unique pairs of elements in $\mathbf{x}$. We can visualize such a multiplication as the triangle depicted in Figure 4.A.2 (note that we do not incur any costs for register-wide gaps in the arrangement). As we are using vector registers, we can perform 256 such multiplications in parallel. In $\mathbb{F}_2$, such a multiplication is simply a bitwise AND operations. To do this, we rearrange the products in a way that makes them more easily computable – see Figure 4.A.3.

The elements as arranged in Figure 4.A.3 can be generated by rotating the original vector and computing the bitwise AND with the original after each rotation. Consequently, we require only $\frac{n}{2} + 1$ vectorized AND operations (one of which is the last half-row). After computing a row of products, each of the output bits in the result is multiplied with m bits from the system parameter, one at a time.

To compute $\mathbf{G}(\mathbf{x}, \mathbf{y})$, one can iterate through two instances of Figure 4.A.3 in parallel; one for each input. Computing $a_{i,j,l} \wedge ((x_i \wedge y_j) \oplus (x_j \wedge y_i))$, where $a_{i,j,l}$ is an element of $\mathbf{F}$, is then a matter of computing a crosswise XOR before masking with system parameter bits. Note that the first row can be skipped, as $(x_i \wedge y_i) \oplus (x_i \wedge y_i)$ trivially results in 0.

4.A.3 Performance

For the 3-pass scheme, the signature size is fairly large because of the high number of rounds, as well as the large n and m. With each vector consuming 256 bits and each of the 438 rounds adding three vectors and a commitment hash, the total amounts to $2 \cdot 256 + 438 \cdot (3 \cdot 256 + 256) = 449\,024$ bits, or 54.81 KiB. Both the secret key and the public key are 64 bytes.

As one would expect, the calls to the $\mathcal{MQ}$ function are the most time-consuming. Theoretically, we can calculate that we require $n \cdot \frac{n+1}{2}$ AND operations to compute all quadratic monomials in a call to **F**. For each of the m output bits, the results are masked with bits from the system parameter, incurring another $m \cdot n \cdot \frac{n+1}{2}$ AND operations, as well as $m \cdot (n \cdot \frac{n+1}{2} - 1)$ XOR operations. Using vector instructions, the most optimal scenario would allow us to do 256 such operations in parallel. While this is almost achieved (although not fully for the half-row of monomials, and because the accumulators used to compute the final result need to be folded onto themselves at the end), this is offset greatly by the amount of loads and stores required to manage the 256 accumulators for the output results. Some additional costs are also incurred for having to rotate the vector register for every 256 monomials, at the cost of two shifts, a quadword permute and an OR operation. Costs of the application of the function **G** are similar, although distributed slightly differently: each 'binomial' costs two AND operations and a XOR operation, and two vectors need to be rotated, but as the symmetric monomials can be omitted, one entire iteration of updating accumulators can be spared.

For one iteration of the $\mathcal{MQ}$ function, we measure[25] 122 564 cycles (again, **G** is marginally cheaper: 121 928 cycles), while we measure 118 088 992 cycles for the complete signature generation. Key generation comes in at 8 066 324 cycles, and verification costs 82 650 156 cycles. On the used CPU, that comes down to 33.7 ms, 2.30 ms and 23.6 ms, respectively. On average, verification should require $1\frac{1}{3}$ calls to an $\mathcal{MQ}$ function, varying with the challenge value.

---

[25] As mentioned in Section 4.5, benchmarks were performed on an Intel Core i7-4770K CPU at 3.5 GHz.

Chapter 5

Lattice-based KEMs

> This chapter is based on the peer-reviewed papers *"High-speed key encapsulation from NTRU"* [HRS+17a] and *"Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates"* [KRS19], and the NTRU-HRSS / NTRU submission to NIST's Post-Quantum Cryptography Standardization project [HRS+17b; CDH+19].

So far, we have focused on one of two basic public-key primitives. After discussing two methods of producing post-quantum digital signatures, this chapter will touch upon key-encapsulation mechanisms (KEMs). In terms of speed, key size and ciphertext size, arguably the most promising candidates are found in the area of lattice-based cryptography. It is not surprising that various recent papers propose a variety of constructions and parameters for lattice-based encryption schemes and KEMs, often together with implementations. See, for example, [BCN+15; ADP+16; BCD+16; CKL+18; BCL+17; Saa17b; CHK+17; PLP16]. These schemes differ in terms of security notions (e.g., passive vs. active security), underlying hard problems (e.g., learning-with-errors vs. learning with rounding), structure of the underlying lattices (standard vs. ideal lattices), cryptographic functionality (encryption vs. key encapsulation), and performance in terms of speed and sizes. It is also no coincidence that Google chose lattice-based schemes for their experiments with real-world deployment of post-quantum TLS [Bra16; Lan18]. The first of these experiments involved NewHope [ADP+16], a key exchange based on the ring-learning-with-errors (Ring-LWE or RLWE) problem. The second, two years later, is based on NTRU-HRSS [HRS+17b], a KEM based on the NTRU problem. It is the latter of these two schemes that we will discuss in the upcoming sections.

Dating back to the nineties, NTRU [HPS96; HPS98] is one of the oldest lattice-based schemes. Since then, several variants [HGSS+03; Con03; HGSW05; IEE09] have been proposed, and their security analyzed [May99; Sch03; BL06; Lud03; HG07; HHHG+09; Wun16]. Perhaps most famously, variants of the NTRU construction have been encumbered with patents, hindering its deployment. The NTRU cryptosystem was patented in [HPS00], and NTRU with "product-form keys" was patented in [HS06]. The former patent was due to expire on August 19, 2017, but in March of that year Security Innovation released both patents [Sec17], placing NTRU into the public domain.

Recent work in lattice-based cryptography may have given the impression that NTRU has been superseded. Indeed, both schemes based on Ring-LWE [LPR10] as well as NTRU Prime [BCL+17] present interesting alternatives. We argue that the choice is not as clear-cut, both in terms of performance and in terms of security.

As part of this work, we revisit the original proposal [HPS98] (sometimes referred to as 'classic NTRU'), and use it to construct an IND-CCA2-secure KEM, satisfying Definition 2.1.13. We describe the resulting scheme in detail in Section 5.1, and touch upon the subsequent submission to NIST's Post-Quantum Cryptography Standardization project in Section 5.1.4.

In the remainder of this chapter, we shift the focus towards high-speed software implementations. Initially, we will look at what it takes to speed up NTRU-HRSS. Part of the motivation underlying the work of [HRS+17a] was showing that classic NTRU can indeed lead to a highly performant key-encapsulation mechanism, even when instantiated without fixed-weight sampling, guaranteeing correctness, and implemented to run in constant time. We describe a highly optimized implementation, and demonstrate its competitiveness.

We then broaden our scope somewhat, discussing optimizations that apply to a larger class of lattice-based schemes. In particular, we optimize the polynomial arithmetic underlying several submissions to NIST that make use of a ring similar to the one chosen for NTRU-HRSS, i.e., rings of the form $\mathbb{Z}_{2^m}[x]$. We specifically target the Cortex-M4 microcontroller, and use code generation to explore a wide range of multiplication strategies.

While it may seem counterintuitive, this chapter will not discuss lattices at all. The problems that underly the hardness of the discussed schemes find their origin in lattice problems, but, given a set of parameters, merely examining the properties and arithmetic of these constructions requires no context in lattices.

5.1 NTRU-HRSS

In this section, we review and instantiate the classic NTRU cryptosystem. As we did not name the resulting system in the paper [HRS+17a] presenting it, its name stems from subsequent citations: NTRU-HRSS. We later adopted this name for the round 1 submission[1] to NIST's Post-Quantum Cryptography Standardization project, which we will discuss in Section 5.1.4. Throughout this thesis, we refer to the scheme as NTRU-HRSS.

We begin by introducing the underlying public-key encryption scheme. This PKE is passively CPA-secure. Directly constructing an IND-CCA2-secure scheme from NTRU appears to be non-trivial [HGSS+03]. However, already [Sta05] and [Sak07] showed that most of this complexity can be avoided when constructing an NTRU-based IND-CCA2-secure KEM. We follow this approach, and apply a modified version of a transform by Dent [Den03], which finds its roots in the popular Fujisaki-Okamoto transform [FO99]. Before presenting any construction, we must first establish some parameters and context.

5.1.1 Parameters

NTRU operates over quotient rings — in particular, the rings $\mathbb{Z}[x]/(p, x^n - 1)$ and $\mathbb{Z}[x]/(q, x^n - 1)$ with p, q, and n co-prime integers. For ease of exposition, we typically denote these rings as $\mathbb{Z}_p[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/(x^n - 1)$, and even as $\mathcal{R}_p$ and $\mathcal{R}_q$, respectively, leaving the irreducible polynomial $x^n - 1$ implicit.[2]

For NTRU-HRSS, we further require subrings of $\mathcal{R}_p$ and $\mathcal{R}_q$, which we define using cyclotomic polynomials. We denote the d^{th} cyclotomic polynomial as Φ_d. In particular, we remark that $\Phi_1 = x - 1$, and, if d is prime, $\Phi_d = 1 + x + x^2 + \cdots + x^{d-1}$. We mimic the shorthand notation above to write $\mathcal{S}_p$ for $\mathbb{Z}[x]/(p, \Phi_n)$ and $\mathcal{S}_q$ for $\mathbb{Z}[x]/(q, \Phi_n)$, with p, q, and n as fixed before. When n is prime, we have $x^n - 1 = \Phi_1\Phi_n$ and thus $\mathcal{R}_p \cong \mathbb{Z}[x]/(p, \Phi_1) \times \mathcal{S}_p$ and $\mathcal{R}_q \cong \mathbb{Z}[x]/(q, \Phi_1) \times \mathcal{S}_q$.

NTRU-HRSS differs from other NTRU instantiations in several ways. First, we work directly with $\mathcal{S}_p$ and $\mathcal{S}_q$ to avoid common security issues associated with the $\mathbb{Z}[x]/(\Phi_1)$ subring. While it is possible to instantiate NTRU directly in $\mathbb{Z}[x]/(\Phi_n)$ and not use $\mathbb{Z}[x]/(x^n - 1)$ at all, we still lift elements of $\mathcal{S}_p$ and $\mathcal{S}_q$ to $\mathcal{R}_p$ and $\mathcal{R}_q$ to take advantage of convenient computational and geometric features of the larger ring. Second, we choose parameters so that decryption failures are completely eliminated, and we do this without restricting the key and message spaces. Finally, we eliminate any need for fixed-weight distributions like those used in [HPS98;

---

[1] For the round 2 update, the merger of NTRU-HRSS and NTRUEncrypt was renamed to NTRU.

[2] Note that this notation slightly differs from the notation as used in [HRS+17a], but more closely relates to notation used in RLWE literature. We use the same notation in [KRS19].

Con03; HGSW05; HHHG+09; HPS+17; BCL+17; PLP16]. All of our sampling routines are chosen to admit simple and efficient constant-time implementations.

For the remainder of this section, we let $p = 3$; we often explicitly write 3 instead of p. In [HRS+17a], we show that we then require $q \geq 8192$ to achieve zero decryption failures. By choosing n such that Φ_n is irreducible modulo p and q, we guarantee that all polynomials computed during key generation are invertible.[3] This simplifies key generation, and makes it easier to implement the entire routine in constant time. With only several candidates left, we choose $n = 701$.

We claim that our $n = 701$ parameter set offers 128-bit security in a post-quantum setting. As a consequence of the parameter selection described above, n is the only free parameter; for $n = 701$ we have $p = 3$ and $q = 8192$. The claim of 128-bit post-quantum security is based on two separate numerical analyses. First, an analysis of the "known quantum" primal attack described in [ADP+16] with the cost model of the same paper. Second, an analysis of the hybrid attack [HG07] using the cost model of [HPS+17]. We defer the details of this to [HRS+17a], where we also review the cryptanalytic literature around NTRU and provide some insight into how security analyses of NTRU have evolved since 1996.

5.1.2 CPA-secure NTRU encryption

We now define an intermediate public-key encryption scheme NTRU-HRSS-PKE by subsequently defining KeyGen, Enc and Dec. These functions explicitly serve as building blocks towards constructing NTRU-HRSS.

Key generation

For NTRU-HRSS-PKE, the secret key is a non-zero element $f \in \mathcal{S}_3$. From f, we compute the public key h as $h = \Phi_1 \cdot g \cdot f^{-1} \in \mathcal{R}_q$ for some $g \in \mathcal{S}_3$. This requires inverting f with respect to $\mathcal{R}_q$. To aid decryption, we also compute f^{-1} with respect to $\mathcal{S}_3$ and store it as part of the secret key. To avoid confusion, we refer to these inverses as f_q^{-1} and f_3^{-1}, respectively. By construction, f is invertible in both $\mathcal{S}_3$ and $\mathcal{S}_q$. Previous NTRU variants have typically taken f to be an element of $\mathcal{R}$, requiring invertibility testing. We must still compute f^{-1} both with respect

---

[3] A similar condition has been recommended since the original description of NTRU [HPS96; HPS98], but has not previously been a requirement. Streamlined NTRU Prime has an analogous requirement for q, but not for p [BCL+17].

to $\mathcal{S}_3$ and to $\mathcal{S}_q$, but this never fails. It suffices that f is invertible in $\mathcal{S}_q$ rather than $\mathcal{R}_q$, as multiplication by Φ_1 during decryption accounts for this difference.

Rather than directly sampling f and g from $\mathcal{S}_3$, we sample them from the subset

$$\mathcal{T}^+ = \{v \in \mathcal{S}_3 : \langle xv, v \rangle \geq 0\}.$$

This correlation restriction is new in NTRU-HRSS. In Section 3.4 and 3.5 of the original publication [HRS+17a], we discuss how sampling from $\mathcal{T}^+$ is simple and efficient in constant-time, as well as showing how it enables the correctness proof.

```
Algorithm 27 NTRU-HRSS-PKE.KeyGen ()                    $\mathcal{R}_q, \mathcal{S}_3, \Phi_1, \mathcal{T}^+$
  $f \xleftarrow{\$} \mathcal{T}^+$
  $f_3^{-1} \leftarrow f^{-1} \in \mathcal{S}_3$
  $f_q^{-1} \leftarrow f^{-1} \in \mathcal{S}_q$
  $g \xleftarrow{\$} \mathcal{T}^+$
  $h \leftarrow \Phi_1 \cdot g \cdot f_q^{-1} \in \mathcal{R}_q$
  return pk = $h$, sk = $(f, f_3^{-1})$
```

Encryption and decryption

Let $m \in \mathcal{S}_3$ be an encoding of the message, and $r \in \mathcal{S}_3$ a random element.[4] The computationally hard problem underlying NTRU variants is that it is now difficult to find m, given a relation $rh + m$. We compute the ciphertext e as

$$e \leftarrow p \cdot r \cdot h + m \in \mathcal{R}_q.$$

Note that this requires lifting m from $\mathcal{S}_3$ to $\mathcal{R}_q$. When decrypting, we first compute $e \cdot f \in \mathcal{R}_q$. We then take this result, reduce it to $\mathcal{S}_3$, and multiply by f_3^{-1}. This results in a message m'. See Algorithms 28 and 29.

```
Algorithm 28 NTRU-HRSS-PKE.Enc ($m, h$ = pk)                    $p, \mathcal{R}_q, \mathcal{S}_3$
  $r \xleftarrow{\$} \mathcal{S}_3$
  $e \leftarrow p \cdot r \cdot h + [m]_3 \in \mathcal{R}_q$
  return $e$
```

---

[4] Previous NTRU variants often required m and r to be balanced, i.e., have coefficients in $\{-1, 0, 1\}$ with each coefficient occurring a fixed number of times. Our variant does not have such a restriction, simplifying random sampling.

```
Algorithm 29 NTRU-HRSS-PKE.Dec $\left(e, (f, f_3^{-1}) = \mathsf{sk}\right)$        $\mathcal{R}_q, \mathcal{S}_3$
    $m' \leftarrow [e \cdot f]_q \cdot f_3^{-1} \in \mathcal{S}_3$
    return $m'$
```

It is not immediately trivial to observe correctness, i.e., that $m = m'$ holds after encapsulation and decapsulation; the mixed operations in $\mathcal{S}_3$ and $\mathcal{R}_q$ make this somewhat hard to grasp. We justify this in more detail in the proof of correctness in [HRS+17a], where we explicitly define the lifting operation between the rings. In the derivation below, we briefly use the notation $[\dots]_q$ and $[\dots]_3$ to make explicit in what ring a computation takes place. Here, we note that

$$
\begin{aligned}
m' &\equiv [[e \cdot f]_q \cdot f_3^{-1}]_3 && \\
&\equiv [[(p \cdot r \cdot h + [m]_3) \cdot f]_q \cdot f_3^{-1}]_3 && \\
&= [[p \cdot r \cdot h \cdot f + [m]_3 \cdot f]_q \cdot f_3^{-1}]_3 && \\
&\equiv [[p \cdot r \cdot g \cdot \Phi_1 \cdot f_q^{-1} \cdot f + [m]_3 \cdot f]_q \cdot f_3^{-1}]_3 && h \equiv g \cdot \Phi_1 \cdot f_q^{-1} \\
&\equiv [[p \cdot r \cdot g \cdot \Phi_1 + [m]_3 \cdot f]_q \cdot f_3^{-1}]_3 && \\
&= [3 \cdot [r \cdot g \cdot \Phi_1]_q + [[m]_3 \cdot f]_q \cdot f_3^{-1}]_3 && p = 3 \\
&\equiv [[[m]_3 \cdot f]_q \cdot f_3^{-1}]_3 && [3 \cdot [r \cdot g \cdot \Phi_1]_q]_3 \equiv 0 \\
&\approxeq [m \cdot f \cdot f_3^{-1}]_3 && \mathcal{R}_q \to \mathcal{S}_3 \\
&\equiv [m]_3 \in \mathcal{S}_3.
\end{aligned}
$$

5.1.3 Fujisaki-Okamoto and an IND-CCA2-secure KEM

We now discuss how to turn the previously described CPA-secure encryption scheme into an IND-CCA2-secure key-encapsulation mechanism: NTRU-HRSS. We achieve this using a generic transform by Dent [Den03, Table 5]. Similar transforms have already been used for the NTRU-based KEMs described in [Sta05; Sak07] and [BCL+17]. This transform comes with a security reduction in the random oracle model. As we are interested in post-quantum security, we have to deal with the quantum-accessible random oracle model (QROM) [BDF+11]. As it turns out, Dent's transform can be viewed as a variant of the Fujisaki-Okamoto transform [FO99]. In [TU15], Targhi and Unruh show how to modify this transform to obtain a security reduction in the QROM.

The resulting KEM works as follows. First, a random string m is sampled from the message space of the encryption scheme. This string is encrypted using random 'coins' to seed the randomness of the PKE, deterministically derived from m using a hash function. This hash function is modeled as a random oracle (RO) in the proof — in practice, we use domain-separated extendable output functions to instantiate all random oracles. The shared secret is derived from m by applying another random oracle. Finally, the ciphertext and the shared secret are output.

The decapsulation algorithm decrypts the ciphertext to obtain m, derives the random coins from m, and re-encrypts m using these coins. If the resulting ciphertext matches the received one, it generates the shared secret from m.

In the QROM setting, Targhi and Unruh add a hash of m to the ciphertext for the sake of the proof. In the QROM, a reduction that involves simulating the random oracles has no way of learning the actual content of adversarial RO queries. This issue can be circumvented using a length-preserving hash function in much the same way as we have seen in Section 4.7.1 when discussing Unruh's transform [Unr15] for signatures. In the proof, this function is simulated using an invertible function and inverts it to recover the corresponding input.

For our specific parameters, this 'confirmation hash' leads to an increase of the ciphertext of 141 bytes. This accounts for 11% of the final encapsulation size, and users that do not consider a QROM proof necessary can simply omit the hash.

In Algorithms 31 and 32, we explicitly describe the encapsulation and decapsulation routines that result from applying the transform described above. Note that the random polynomial r is the instantiation of the random coins, and that we assume three hash functions: $\mathrm{H}_r\colon \mathcal{S}_3 \to \mathcal{S}_3$, $\mathrm{H}_{\mathrm{ss}}\colon \mathcal{S}_3 \to \{0,1\}^k$, and $\mathrm{H}_{\mathrm{qrom}}\colon \mathcal{S}_3 \to \{0,1\}^*$. Here, k is the security parameter, dictating the length of the resulting shared secret. We do not necessarily care about the range of $\mathrm{H}_{\mathrm{qrom}}$, but only require it to be length-preserving. For completeness, we also include the practically unchanged key generation in Algorithm 30; we include h in sk to allow re-encryption.

5.1.4 The NIST submission

In November of 2017, NTRU-HRSS was submitted to the NIST's Post-Quantum Cryptography Standardization project (see Section 2.2). In January of 2019, NIST announced that NTRU-HRSS has progressed as one of the second-round candidates. After merging with the NTRUEncrypt submission, the two schemes continue under the joint banner of NTRU.

Algorithm 30 NTRU-HRSS.KeyGen () $\mathcal{R}_q, \mathcal{S}_3, \Phi_1, \mathcal{T}^+$

```
$f \xleftarrow{\$} \mathcal{T}^+$
$f_3^{-1} \leftarrow f^{-1} \in \mathcal{S}_3$
$f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}_q$
$g \xleftarrow{\$} \mathcal{T}^+$
$h \leftarrow \Phi_1 \cdot g \cdot f_q^{-1} \in \mathcal{R}_q$
return $\mathsf{pk} = h, \mathsf{sk} = (f, f_3^{-1}, h)$
```

Algorithm 31 NTRU-HRSS.Encaps (h = pk) $p, \mathcal{R}_q, \mathcal{S}_3, \mathrm{H}_r, \mathrm{H}_{\mathrm{ss}}, \mathrm{H}_{\mathrm{qrom}}$

```
$m \xleftarrow{\$} \mathcal{S}_3$
$r \leftarrow \mathrm{H}_r(m)$
$\mathsf{ss} \leftarrow \mathrm{H}_{\mathrm{ss}}(m)$
$e_1 \leftarrow p \cdot r \cdot h + [m]_3 \in \mathcal{R}_q$
$e_2 \leftarrow \mathrm{H}_{\mathrm{qrom}}(m)$
return $e_1, e_2, \mathsf{ss}$
```

Algorithm 32 NTRU-HRSS.Decaps (e_1, e_2, sk) $p, \mathcal{R}_q, \mathcal{S}_3, \mathrm{H}_r, \mathrm{H}_{\mathrm{ss}}, \mathrm{H}_{\mathrm{qrom}}$

```
$(f, f_3^{-1}, h) = \mathsf{sk}$
$m' \leftarrow [e_1 \cdot f]_q \cdot f_3^{-1} \in \mathcal{S}_3$
$r' \leftarrow \mathrm{H}_r(m')$
$\mathsf{ss} \leftarrow \mathrm{H}_{\mathrm{ss}}(m')$
$e_1' \leftarrow p \cdot r' \cdot h + [m']_3 \in \mathcal{R}_q$
$e_2' \leftarrow \mathrm{H}_{\mathrm{qrom}}(m')$
if $(e_1, e_2) \neq (e_1', e_2')$ then
    $\mathsf{ss} \leftarrow \perp$
end if
return $\mathsf{ss}$
```

The description of NTRU-HRSS as given above very closely aligns with the submission to the first round of the NIST project, recommending the presented parameter set where $n = 701$. The second-round submission contains several changes. While these are largely outside the scope of this thesis, we briefly outline the most significant differences for completeness.

NTRU-HRSS and NTRU-HPS

In the NTRU submission, we specify four distinct parameter sets: ntruhps2048509, ntruhps2048677, ntruhps4096821, and ntruhrss701. While ntruhrss701 closely follows the previously described NTRU-HRSS design, the NTRU-HPS instances differ slightly in their use of fixed-weight sampling of polynomials, following the original work by Hoffstein, Pipher and Silverman [HPS96; HPS98]. This allows for a more flexible combination of choices for q and n. In all other aspects, the two proposed variants share design choices — in particular, all parameter sets are selected to guarantee absence of decryption failures.

The [SXY18] transform

Saito, Xagawa, and Yamakawa [SXY18] propose an alternative transform to obtain an IND-CCA2 KEM, based on a deterministic public-key encryption (DPKE) scheme. They demonstrate this by transforming a variant of the NTRU-HRSS-PKE scheme that underlies NTRU-HRSS. This transform allows for a tight reduction to the CPA-security of the PKE in the ROM, as well as a reduction in the QROM – the QROM reduction is also tight under slightly stronger assumptions. Whereas the transform described in Section 5.1.3 required a confirmation hash, the [SXY18] transform avoids this. Besides requiring the PKE to be deterministic, the proof also requires implicit rejection: rather than returning $\perp$ in case of failure, the shared secret ss is set to a random bit string.[5] In the round 2 submission, we apply an interoperable variant of the [SXY18] transform, relying on work by [BP18] to construct the DPKE without requiring the re-encryption step that [SXY18] introduced as part of the DPKE decryption.

---

[5] While this is a minor technicality in the proof, arising from preventing distinguishability of simulated ciphertexts, it has some consequences for the API of actual implementations. In particular, this shifts the burden of detecting failure from the decapsulation routine to the overarching protocol that later relies on a mutually shared secret. Note that this is not limited to decryption failure as captured in Definition 2.1.11, but also includes, e.g., failure because of ciphertexts outside the domain of Decaps.

5.2 High-speed key encapsulation

As part of the software accompanying [HRS+17a] and the submission to NIST's Post-Quantum Cryptography Standardization project [HRS+17b], we provide a portable reference implementation. More importantly in context of this thesis, we also provide an optimized implementation using vector instructions from the AVX2 instruction set. Both implementations run in constant time. For the AVX2 implementation, we strongly rely on the specific properties of the parameter set defined by $n = 701$ (and thus $q = 8192$ and $p = 3$). This section highlights some of the relevant building blocks to consider when implementing the scheme, focusing on the AVX2 implementation. Recall that the AVX2 extensions (see Section 2.4.1) provide 16 vector registers of 256 bits that support a wide range of SIMD instructions. As before, all benchmarks in this section were obtained on one core of an Intel Core i7-4770K Haswell CPU at 3.5 GHz unless otherwise specified.

5.2.1 Polynomial multiplication

It will come as no surprise that one of the most crucial implementation aspects is polynomial multiplication. For NTRU-HRSS, we require multiplication in $\mathcal{R}_q$ during key generation as well as during encryption and decryption. Additionally, decryption uses multiplication in $\mathcal{S}_3$. Furthermore, we use multiplication of binary polynomials in order to perform inversion in $\mathcal{S}_q$, which we will describe in Section 5.2.2. Refer to Algorithms 30, 31, and 32 for the respective pseudocode. Before looking at specific routines, we discuss multiplication strategies more generally.

Popular choices for the ring in Ring-LWE schemes typically make it convenient to use the number-theoretic transform (NTT) to perform multiplication. As is often the case for NTRU variants, however, the ring of choice is particularly unsuitable (see, e.g., the discussion in [BCL+17]).[6] This is a consequence of q being a power of two, and the polynomials being of prime degree. Instead, we rely on more generically applicable multiplication methods.

Schoolbook multiplication

At the heart of almost any multiplication method lies 'schoolbook' multiplication. This is literally the way multiplication is typically taught in schools: the operands

---

[6] Recently, Lyubashevsky and Seiler demonstrated that this is not necessarily the case [LS19], describing and implementing a record-shattering NTRU variant over a ring that permits the use of the NTT.

are split into parts, the parts are multiplied pairwise, and the resulting products are accumulated. This apparently trivial operation leaves much room for optimization, and a long history of extensive literature exists [Bar86; Com90; GPW+04; HW11] on exactly how to schedule the multiplications. In practice, memory-access patterns have a defining impact on performance, and processors may support operations that efficiently combine certain multiplications and additions. Overall, though, the complexity of schoolbook multiplication is clear, and its cost predictable: the number of additions and multiplications is quadratic in the size of the operands. We discuss this in some more detail in the next section, when we more carefully optimize schoolbook multiplications on a more constrained device. In the implementation described here, we simply decompose multiplications using Toom-Cook and Karatsuba until a schoolbook multiplication can be performed within the available registers.

Karatsuba and Toom-Cook

In the early 1960s, Karatsuba [KO63] showed that the asymptotic complexity of multiplication is, in fact, not plainly quadratic. Instead, using a method that would later be named after him, Karatsuba showed that the complexity is $\Theta(n^{\log_2 3})$. We briefly illustrate the key idea of the method using an example of multiplication of polynomials (but note that it applies more generally).

Let a and b be polynomials of n terms, which we write as a composition of limbs of $n/2$ terms, i.e., $a = a_h \cdot x^{n/2} + a_l$ and $b = b_h \cdot x^{n/2} + b_l$. We then see that:

$$\begin{aligned} a \cdot b &= (a_h \cdot x^{n/2} + a_l) \cdot (b_h \cdot x^{n/2} + b_l) \\ &= a_h b_h \cdot x^n + (a_l b_h + a_h b_l) \cdot x^{n/2} + a_l b_l \\ &= a_h b_h \cdot x^n + ((a_h + a_l) \cdot (b_h + b_l) - a_h b_h - a_l b_l) \cdot x^{n/2} + a_l b_l \end{aligned}$$

Plainly counting operations now suggests that the computational load increased from 4 to 5 multiplications of limbs, and several extra additions. However, crucially, $a_h b_h$ and $a_l b_l$ occur twice, and can be reused after the initial computation. This results in only three multiplications with operands of $n/2$ terms.

The derivation above suggests three n-sized additions and subtractions,[7,8] as well as two $n/2$-sized additions. A further optimization, commonly known as

---

[7] Three additions, and not four: the resultant limbs $a_h b_h \cdot x^n$ and $a_l b_l$ do not overlap at all, and thus do not introduce any additions of coefficients.

[8] To be exact, the full-size additions are actually not n-sized. Multiplying two polynomials of $n/2$ terms produces $n - 1$ resultant terms: no term of degree $n - 1$ is produced.

'refined Karatsuba' [Ber09], reduces this to two n-sized additions and three $n/2$-sized additions by sharing additions between the subtraction of $a_h b_h \cdot x^{n/2}$ and $a_l b_l \cdot x^{n/2}$.

Toom [Too63] subsequently generalized Karatsuba's method, setting lower bounds in the asymptotics. The algorithm is typically called Toom-Cook, as Cook [Coo66] is credited for cleaning up the algorithm and making it accessible. The intuition behind this algorithm is to partially evaluate polynomials and multiply the results rather than directly multiplying the polynomials themselves. To this end, each of the two input polynomials is first evaluated at a fixed set of points, resulting in two sets of polynomials of a lower degree. After multiplying these smaller polynomials, the results are interpolated with respect to the chosen points, producing limbs that are then recomposed to derive the output polynomial.

While Karatsuba's method can be applied somewhat more generally to produce six multiplications (instead of nine) for a three-way split, or recursively to reduce the sixteen multiplications of a four-way split to only nine, Toom-Cook turns nine multiplications into five ('Toom-3') and sixteen into seven ('Toom-4'). Naturally this comes at the cost of a more involved process of additions and subtractions, making the trade-off less worthwhile for low-degree polynomials. Both Karatsuba's method and Toom-Cook's algorithm scale arbitrarily, and can be arbitrarily and recursively combined. As we will see in the next sections, polynomials of degrees relevant in lattice-based cryptography typically require one or two layers of Toom-Cook before Karatsuba becomes the optimal decomposition strategy.

There is more to multiplication than schoolbook, Karatsuba and Toom-Cook (such as the work by Schönhage and Strassen [SS71], Fürer's algorithm [Für09], and the recent improvements by Harvey, Van Der Hoeven and Lecerf [HHL16]). For the context of the work presented here, Karatsuba and Toom-Cook suffice.

Multiplication in $\mathcal{R}_q$

We now describe how the multiplication in $\mathcal{R}_q$ can be composed of smaller instances by combining Toom-Cook multiplication with Karatsuba's method. Consider that elements of $\mathcal{R}_q$ are polynomials with 701 coefficients in $\mathbb{Z}_{8192}$; 16 such coefficients fit in a vector register. With this in mind, we look for a sequence of decompositions that results in multiplications best suited for parallel computation.

By applying Toom-Cook to split into 4 limbs, we decompose into 7 multiplications of polynomials of degree 176. We decompose each of those by recursively applying two instances of Karatsuba to obtain 63 multiplications of polynomials of 44 coefficients. Consider the inputs to these multiplications as elements of a matrix, rounding the dimensions up to 64 and 48. By transposing this matrix we can efficiently perform the 63 multiplications in a vectorized manner — rather than storing fragments of polynomials in vector registers, we designate registers only for leading coefficients, for secondary coefficients, et cetera. As a consequence, from this point onwards, we are no longer bound to polynomials with degrees that approximate multiples of 16. Using three more applications of Karatsuba, we decompose first into 22 and 11 coefficients, until finally we are left with polynomials of degree 5 and 6. At this point a straight-forward schoolbook multiplication can be performed completely in registers, without any memory interaction.

The full sequence of operations is as follows. We first combine the evaluation step of Toom-4 and the two layers of Karatsuba. Then, we transpose the obtained 44-coefficient results by applying transposes of 16x16 matrices, and perform the block of 63 multiplications. The 88-coefficient products remain in 44-coefficient form (i.e. aligned on the first and 45th coefficient), allowing for easy access and parallelism during interpolation: limbs of 44 coefficients are the smallest interacting elements that interact during this phase, making it possible to operate on each part individually and keep register pressure low.

Multiplication in $\mathcal{R}_q$ costs 11 722 cycles. Of this, 512 cycles are spent on point evaluation, 3 692 cycles are used to transpose, 4 776 are spent computing 64-way parallel multiplications, and interpolation and recomposition take 2 742 cycles.

Multiplication in $\mathcal{S}_3$

In the ring $\mathcal{S}_3$, it appears to be efficient to decompose the multiplication by applying Karatsuba recursively five times, resulting in 243 multiplications of polynomials of degree 22. One could then bitslice the two-bit coefficients into 256-bit registers with only very minimal wasted space, and perform schoolbook multiplication on the 22-register operands, or even decide to apply another layer of Karatsuba.

For our implementation, however, we instead decide to use our $\mathcal{R}_q$ multiplication as though it were a generic $\mathbb{Z}[x]/(x^n - 1)$ multiplication. Even though in general these operations are not compatible, it works out for our parameters. After multiplication and summation of the products, each result is at most $701 \cdot 4 = 2804$, staying well below the threshold of 8192. While a dedicated $\mathcal{S}_3$ multiplication would out-perform this use of $\mathcal{R}_q$ multiplication, the choice of parameters makes this an attractive alternative at a marginal cost.

Multiplication in $\mathbb{Z}_2[x]$

Dedicated processor instructions have made multiplications in $\mathbb{Z}_2[x]$ considerably easier. As part of the CLMUL instruction set, the `pclmulqdq` instruction computes a carry-less multiplication of two 64-bit quadwords, performing a multiplication of two 64-coefficient polynomials over $\mathbb{Z}_2$.

We set out to efficiently decompose into polynomials of degree close to 64, and do so by recursively applying a Karatsuba layer of degree 3 followed by a regular Karatsuba layer and a schoolbook multiplication. This reduces the full multiplication to 72 multiplications of 59-bit operands, which we perform using `pclmulqdq`. By interleaving the evaluation and interpolation steps with the multiplications, we require no intermediate loads and stores, and a single multiplication ends up measuring in at only 244 cycles.

5.2.2 Inverting polynomials

Computing the inverse of f plays a critical role in the performance of key generation. Recall that we compute $f^{-1} \in \mathcal{S}_q$ when producing the public key, but also pre-compute $f^{-1} \in \mathcal{S}_3$ as part of the secret key, to be used during decryption.

Inversion in $\mathcal{S}_q$

We first compute f^{-1} with respect to $\mathcal{S}_2$ and then apply a variant of Newton iteration [Sil99] in $\mathcal{R}_q$ to obtain $f_q^{-1} \equiv f^{-1} \pmod{(q, \Phi_n)}$. It may not be the case that $f_q^{-1} = f^{-1} \in \mathcal{S}_q$, i.e., that f_q is the smallest representative, but the difference this makes in the calculation of h is eliminated after the multiplication by Φ_1 (see Algorithm 30). The Newton iteration adds an additional cost of eight multiplications in $\mathcal{R}_q$ on top of the cost of an inversion in $\mathcal{S}_2$.

Finding an inverse in $\mathcal{S}_2$ is done using the fact that $f^{2^{n-1}-1} \equiv 1 \pmod{(2, \Phi_n)}$, and thus $f^{2^{700}-2} \equiv f^{-1} \pmod{(2, \Phi_{701})}$ [IT88]. This exponentiation can be done efficiently using an addition chain, resulting in twelve multiplications, interleaved with thirteen series of repeated squarings.

Performing a squaring operation in $\mathbb{Z}_2[x]$ is equivalent to inserting 0-bits between the bits representing the coefficients: the odd-indexed products cancel out in $\mathbb{Z}_2$. When working modulo $x^n - 1$ with odd n, the subsequent reduction of the polynomial causes the terms with degree exceeding x^n to wrap around and fill the empty coefficients. Consider the toy example in Figure 5.1. This allows us

$$\begin{aligned}
f &= x^6 + x^5 + x^3 + x + 1 && 0000\,0000\,0110\,1011 \\
f^2 &= x^{12} + 2x^{11} + x^{10} + 2x^9 + 2x^8 + 2x^7 + 5x^6 + 2x^5 + 2x^4 + 2x^3 + x^2 + 2x + 1 && \\
&\equiv x^{12} + x^{10} + x^6 + x^2 + 1 && 0001\,0100\,0100\,0101 \\
& && \ldots \rightarrow 0\,0010\,1000 \\
&\equiv x^6 + x^5 + x^3 + x^2 + 1 && 0000\,0000\,0110\,1101
\end{aligned}$$

Figure 5.1: Squaring binary polynomials modulo $x^7 - 1$

to express the problem of computing a squaring as performing a permutation on bits. More importantly: repeated squaring operations can be considered repeated permutations, which compose into a single bit permutation.

Rewording the problem to that of performing bit permutations allows for different approaches — both generically and for specific permutations. In order to aid in constructing routines that perform these permutations, we have developed a tool to simulate a subset of the assembly instructions related to bit movement. Rather than representing the bits by their value, we label them by *index*, making it significantly easier to maintain an overview. The assembly code corresponding to the simulated instructions is generated as output. While we have used this tool to construct permutations that represent multi-squarings, it may be of interest in a broader context. We include it with this work as separately packaged software.

We use two distinct generic approaches to construct permutation routines, based respectively on `pext`/`pdep` from the BMI2 instruction set, and on `vshufb`.

The first approach amounts to extracting and depositing bits that occur within the same 64-bit block in both the source and destination bit sequence, under the constraint that their order remains unchanged. By relabeling the bits according to their destination and using the patience sorting algorithm [Mal63], we iteratively find the longest increasing subsequence in each block until every bit has been extracted. Note that the number of required bit extractions is equal to the number of piles patience sort produces. To minimize this, we examine the result of each possible input rotation, and rotate it by the offset that produces the least amount of disjunct increasing subsequences. Heuristically keeping the most recently used masks in registers reduces the number of load operations, as the BMI2 instructions do not allow operands from memory. Further improvements could include dynamically finding the right trade-off between rotating registers and re-using masks, as well as grouping similar extractions. For the permutations we required, these changes do not seem to hold any promises towards significant improvements.

The second approach uses byte-wise shuffling to position the bits within 256-bit registers. We consider all eight rotations of the input bytes and use `vshufb` to reposition the bytes (as well as `vpermq` to cross the 128-bit lanes). The number of required shuffles is minimized by gathering bytes for all three destination registers at the same time, and where possible, rotation sequences are replaced by shifts (as the rotated bits often play no role in the bit deposit step, and shifts are significantly cheaper). The bit extraction approach works for well-structured permutations, but it is beaten by the more constant shuffling-based method for more erratic transpositions. There is room for improvement by hand-crafting permutations, but it turns out to be non-trivial to beat the generated multi-squarings.

The multi-squaring routines vary around 235 cycles, with a single squaring taking only 58. Including converting from $\mathcal{R}_q$ to $\mathcal{S}_2$, an inversion in $\mathcal{S}_2$ costs 10 332 cycles. Combining this with the multiplication in $\mathcal{R}_q$ described above, the full inversion in $\mathcal{S}_q$ takes 107 726 cycles.

Inversion in $\mathcal{S}_3$

Inversion in $\mathcal{S}_3$ is done using the 'Almost Inverse' algorithm described in [SOO+95] and [Sil99].[9] However, the algorithm as described in [Sil99] does not run in constant time. Notably, it performs a varying number of consecutive divisions and multiplications by x depending on the coefficients in f, and halts as soon as f has degree zero. We eliminate this issue by iterating through every term in f, (i.e., including potential zero terms, up to the n^{th} term), and always performing the same operations for each term (i.e., constant-time swaps and always performing the same addition, multiplied with a sign flag). We list the original algorithm as Algorithm 33, and the constant-time variant in Algorithm 34. Note that `fmadd(f, g, s)` is the operation $f_i + s \cdot g_i \mod 3$, for all coefficients f_i and g_i of the polynomials f and g.

While the number of loop iterations is constant, the final value of the rotation counter k is not — the `done` flag may be set before the final iteration. We compensate for k after the loop has finished by rotating 2^i bits for each bit in the binary representation of k (with $i = 0$ indicating the least-significant bit), and subsequently performing a constant-time move when the respective bit is set.

Benefiting from the width of the vector registers, we operate on bitsliced vectors of coefficients. This allows us to efficiently perform the multiplications and additions in parallel modulo 3, and makes register swaps comparatively easy.

---

[9] The latest software instead uses the inversion algorithm described by Bernstein and Yang in [BY19].

Algorithm 33 AlmostInverse(f) n

```
$k \leftarrow 1, b \leftarrow 1, c \leftarrow 0, g \leftarrow x^n - 1$
while $f_0 = 0$ do
    $f \leftarrow f/x$
    $c \leftarrow c \cdot x$
    $k \leftarrow k + 1$
end while
if $deg(f) < deg(g)$ then
    swap $f$ and $g$, swap $b$ and $c$
end if
if $f = \pm 1$ then
    return $\pm x^{n-k} \cdot b \mod x^n - 1$
end if
if $f_0 = g_0$ then
    $f \leftarrow f - g \mod 3$
    $b \leftarrow b - c \mod 3$
else
    $f \leftarrow f + g \mod 3$
    $b \leftarrow b + c \mod 3$
end if
return $b$
```

Algorithm 34 AlmostInverseConst(f) n

```
$k \leftarrow 1, b \leftarrow 1, c \leftarrow 0, g \leftarrow x^n - 1, deg_f \leftarrow n - 1, deg_g \leftarrow n - 1$
for $i \in \{1, \ldots, 2 \cdot (n-1)\}$ do
    $s \leftarrow 2 \cdot f_0 \cdot g_0 \mod 3$
    $swap \leftarrow s \wedge \neg done \wedge (deg_f < deg_g)$
    cswap($f, g, swap$)
    cswap($b, c, swap$)
    cswap($deg_f, deg_g, swap$)
    fmadd($f, g, s \cdot \neg done$)
    fmadd($b, c, s \cdot \neg done$)
    $f \leftarrow f/x$
    $c \leftarrow c \cdot x$
    $deg_f \leftarrow deg_f - \neg done$
    $k \leftarrow k + \neg done$
    $done \leftarrow (deg_f = 0)$
end for
return $f_0 \cdot x^{n-k} b \mod x^n - 1$
```

On the other hand, shifts are still fairly expensive, and two are performed for each loop iteration to multiply and divide by x. With 159 606 cycles, the inversion remains a costly operation that determines a large chunk of the cost of the key generation operation. As each instruction in the loop is executed 1400 times, small incremental improvements in the critical section may still have significant impact.

5.2.3 Performance and comparison

Table 5.1 gives an overview of the performance of various lattice-based encryption schemes and KEMs. Note that these numbers were collected at the time of this work, considerably before many of these schemes were revisited and revised in the context of their submission to NIST's Post-Quantum Cryptography Standardization project, and a plethora of new schemes were designed. We consider maintaining an up-to-date comparison out of scope for this thesis.

As memory is typically not a big concern on the given platforms, concrete memory usage figures are often not available and we do not attempt to include this in the comparison. In the same spirit, our reference implementation uses almost

11 KiB of stack space and our AVX2 software uses over 43 KiB, but this should not be considered to be a lower bound. We warn the reader that direct comparison of the listed schemes and implementations is near impossible for various reasons. First of all, there are significant differences in the security level; however, at least most schemes aim at a level of around 128 bits of post-quantum security. More importantly, the passively secure KEMs have a very fast decapsulation routine, but turning them into CCA2-secure KEMs via the Targhi-Unruh transform would add the cost of encapsulation to decapsulation. Also, the level of optimization of implementations is different. For example, we expect that Frodo as presented in [BCD+16] or the spLWE-based KEM from [CHK+17] could be sped up through vectorization. Finally, not all implementations protect against timing attacks and adding protection may incur a serious overhead. However, the results show that carefully optimized NTRU is very competitive, even for key generation and even with full protection against timing attacks.

The column 'ct?' indicates whether the software is running in constant time, i.e., with protection against timing attacks. The results come with several footnotes, which we enumerate here for readability. [a] According to the conservative estimates obtained by the approach from [ADP+16]. [b] Benchmarked on a 2.6 GHz Intel Xeon E5 (Sandy Bridge). [c] As reported by SUPERCOP, version 20170725 [BL] for ntruprime-20170815 on Intel Core i-7 4770K (Haswell). [d] Benchmarked on "PC (Macbook Pro) with 2.6 GHz Intel Core i5". [e] Benchmarked by eBACS [BL] on Intel Xeon E3-1275 (Haswell). [f] Unlike most others, the secret key of Lizard does not include the public key required for decryption in the Targhi-Unruh transform. [g] According to the authors' analysis, i.e., not following [ADP+16]. [h] Derived from the implementation – can be compressed to 10/16 of its size at a marginal increase in cost of K, E and D by representing each coefficient using $\log(q)$ bits.

5.3 Polynomials in $\mathbb{Z}_{2^m}[x]$

Having discussed and optimized NTRU-HRSS, we now broaden our scope somewhat. The work discussed in this section follows from attempts to optimize NTRU-HRSS for the ARM Cortex-M4 microcontroller, which quickly dissolved into more generally applicable optimized arithmetic. In particular, we look at the polynomial arithmetic underlying schemes that operate in the ring $\mathbb{Z}_{2^m}[x]$.

In 2017, NIST's Post-Quantum Cryptography Standardization project received 69 "complete and proper" submissions to round 1. By December 2018, five of these had subsequently been withdrawn. Out of the remaining schemes, 22 are based on

Table 5.1: Comparison of lattice-based KEMs and public-key encryption.

scheme	PQ sec.	ct?		cycles		bytes
Passively secure KEMs						
BCNS [BCN+15]	78^{a}	yes	K:	≈ 2 477 958	*sk:*	4096
			E:	≈ 3 995 977	*pk:*	4096
			D:	≈ 481 937	*c:*	4224
NewHope [ADP+16]	255^{a}	yes	K:	88 920	*sk:*	1792
			E:	110 986	*pk:*	1824
			D:	19 422	*c:*	2048
Frodo [BCD+16]	130^{a}	yes	K:	≈ 2 938 000^{b}	*sk:*	11 280
(recommended parameters)			E:	≈ 3 484 000^{b}	*pk:*	11 296
			D:	≈ 338 000^{b}	*c:*	11 288
CCA2-secure KEMs						
Streamlined NTRU	137^{a}	yes	K:	6 115 384^{c}	*sk:*	1600
Prime 4591^{761} [BCL+17]			E:	59 600	*pk:*	1218
			D:	97 452	*c:*	1047
spLWE-KEM [CHK+17]	128^{g}	?	K:	≈ 336 700^{d}	*sk:*	?
(128-bit PQ parameters)			E:	≈ 813 800^{d}	*pk:*	?
			D:	≈ 785 200^{d}	*c:*	804
Kyber [BDK+17]	161^{a}	yes	K:	77 892	*sk:*	2400
(AVX2 optimized)			E:	119 652	*pk:*	1088
			D:	125 736	*c:*	1184
NTRU-HRSS	123^{a}	yes	K:	307 914	*sk:*	1422
			E:	48 646	*pk:*	1140
			D:	67 338	*c:*	1281
CCA2-secure public-key encryption						
NTRU ees743ep1[HPS+17]	159^{a}	no	K:	1 194 816	*sk:*	1 120
			E:	57 440	*pk:*	1 027
			D:	110 604	*c:*	980
Lizard [CKL+18]	128^{g}	no	K:	≈ 97 573 000	*sk:*	466 $944^{f,h}$
(recommended parameters)			E:	≈ 35 000	*pk:*	2 031 616^{h}
			D:	≈ 80 800	*c:*	1 072

lattice structures. Most of those lattice-based schemes use structured lattices and, as a consequence, require fast arithmetic in a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/f$ for some n-coefficient polynomial $f \in \mathbb{Z}_q[x]$. Typically the largest performance bottleneck of these schemes is multiplication in $\mathcal{R}_q$; the optimization of NTRU-HRSS in the previous section is a testament to this.

As noted in Section 5.2.1, many proposals, such as NewHope [ADP+16; AAB+17], Kyber [ABD+17], and LIMA [SAL+17], choose q, n, and f such that multiplication in $\mathcal{R}_q$ can be done using the number-theoretic transform (NTT). Because of the choice of q of the form 2^m, this does not apply to NTRU-HRSS (recall the brief discussion in Section 5.2.1). Similarly, five other submissions choose $q = 2^m$ with some small m: Round2 [GMZB+17], Saber [DKR+17], NTRUEncrypt [ZCH+17], Kindi [Ban17], and RLizard [CPL+17]. Round2 merged with Hila5 [Saa17a] into Round5 [BGML+18], and the Round5 team presented optimized software for the ARM Cortex-M4 processor in [SBGM+18]; the multiplication in Round5 has more structure, allowing for a specialized high-speed routine. Here, we focus on optimizing the other five schemes (i.e., including NTRU-HRSS [HRS+17b]) on the Cortex-M4. We do this by programmatically exploring a large design space for the routines to perform their multiplication operations, combining various instances of Karatsuba's method and Toom-Cook's algorithm.

In related work, we note that Saber has previously been optimized on the ARM Cortex-M4 [KMR+18] as well; our multiplication implementation outperforms the results by 42% which improves the overall performance of key generation by 22%, encapsulation by 20%, and decapsulation by 22%. For the other four schemes the only software that was readily available for the Cortex-M4 was the reference implementation and, unsurprisingly, our optimized code significantly outperforms these implementations. For example, our optimized versions of RLizard-1024 and Kindi-256-3-4-2 encapsulation and decapsulation are more than a factor of 20 faster. Our implementation of NTRU-HRSS encapsulation and decapsulation solidly outperform the optimized Round5 software presented in [SBGM+18].

Most of the work presented in this section was performed between the publication of round-1 candidates to NIST's Post-Quantum Cryptography Standardization project and the elimination of several of these in round 2. In particular, at the current time of writing, Kindi and RLizard are no longer being considered. Round5, Saber, NTRU-HRSS, and NTRUEncrypt made it into the second round. As described in Section 5.1.4, the latter two merged into the NTRU [CDH+19] scheme.

Before discussing the multiplication routines, we briefly introduce Saber, NTRUEncrypt, Kindi and RLizard. We follow with an overview of the relevant extensions to the ARMv7E-M instruction set, and then detail our approach to exploring different Toom-Cook and Karatsuba decomposition strategies for multiplication in $\mathcal{R}_q$. After decomposition, the multiplications dissolve into small schoolbook routines, which we carefully hand-optimize. We conclude this section (and, indeed, this chapter) with a comparison, presenting record-setting benchmarks for post-quantum key encapsulation on the Cortex-M4.

5.3.1 Kindi, NTRUEncrypt, RLizard, and Saber

We now present simplified algorithmic descriptions of the optimized schemes; we omit NTRU-HRSS, as it was discussed in detail in the previous sections.[10] For the sake of brevity, we only provide an overview of the other schemes, and highlight the aspects relevant in the context of this work. Refer to their respective specifications as submitted to NIST for complete descriptions. Readers solely interested in the optimized multiplication routine are encouraged to skip ahead to the next section.

As we have seen with NTRU-HRSS, an IND-CCA2-secure KEM can be constructed by first constructing a passively secure public-key encryption scheme and applying a transformation. We only include algorithmic descriptions of the underlying encryption schemes, omitting the generic transformations. This is sufficient to show the relevant arithmetic. In particular, we highlight the multiplications in $\mathcal{R}_q$ by denoting these operations using the infix $\circledast$.

Similarly, we do not go into any detail with respect to the sampling of random bit strings, polynomials, or matrices, and simply denote all of these functions as Sample_S, where S is the set from which the elements are drawn. While we specify a set to which the sampled elements belong, we leave the distribution according to which they are sampled unspecified. Where deterministic sampling from a specific seed is relevant, Sample_S is parameterized with this seed.

Finally, most schemes make use of rounding coefficients of polynomials. We denote any such rounding operation by $\lfloor \ldots \rceil$ and specify the domain in which the result lives, but again omit the details of how the rounding operation is defined.

---

[10] In the context of this work, we do not optimize all arithmetic involved in NTRU-HRSS. Notably, by only focusing on the multiplications, we address the core operations required in encapsulation and decapsulation, but not all inversions as performed during key generation. As before, we note that the multiplications in the smaller subring can be performed using the multiplication in $\mathcal{R}_q$.

RLizard

RLizard is part of the Lizard proposal [CPL+17]. It is a cryptosystem based on the Ring-Learning-with-Errors (Ring-LWE) and Ring-Learning-with-Rounding (Ring-LWR) problems. As the names suggest, these problems are closely related, and efficient reductions exist [BPR12; BGM+16]. The submission motivates the choice for the Learning-with-Rounding problem by stressing its deterministic encryption routine and reduced ciphertext size compared to Learning-with-Errors. The main structure underlying RLizard is the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, but coefficients of the ciphertext are ultimately reduced to $\mathcal{R}_p$, where $p < q$. We consider the parameter set where $n = 1024$, $q = 2048$ and $p = 512$. In the submission the derived KEM is referred to as `RING_CATEGORY3_N1024` – for the sake of brevity, we denote it as RLizard-1024 from this point onwards. As both p and q are powers of 2, all multiplications in RLizard fit the structure that we target in this work.

```
Algorithm 35 RLizard.KeyGen ()                                    $\mathcal{R}_q$
$a, s, e \leftarrow \text{Sample}_{\mathcal{R}_q}$
$b \leftarrow -a \circledast s + e \in \mathcal{R}_q$
return $(\text{pk} = (a, b), \text{sk} = s)$
```

```
Algorithm 36 RLizard.Enc ($m, (a, b) = \text{pk}$)                $p, q, \mathcal{R}_p, \mathcal{R}_q$
$r \leftarrow \text{Sample}_{\mathcal{R}_q}$
$c_1' \leftarrow a \circledast r \in \mathcal{R}_q$
$c_2' \leftarrow b \circledast r \in \mathcal{R}_q$
$c_1 \leftarrow \lfloor (p/q) \cdot c_1' \rceil \in \mathcal{R}_p$
$c_2 \leftarrow \lfloor (p/q) \cdot ((q/2) \cdot m + c_2') \rceil \in \mathcal{R}_p$
return $(c_1, c_2)$
```

```
Algorithm 37 RLizard.Dec ($(c_1, c_2), s = \text{sk}$)             $p, \mathcal{R}_2$
$m' \leftarrow \lfloor (2/p) \cdot (c_2 + c_1 \circledast s) \rceil \in \mathcal{R}_2$
return $m'$
```

NTRUEncrypt

Like NTRU-HRSS, the NTRUEncrypt scheme [ZCH+17] is based on the standard NTRU construction [HPS98]. Its choice of parameters is based on a recent revisiting in [HPS+17]. The NIST submission of NTRUEncrypt [ZCH+17] presents several instantiations, but we limit ourselves to the instances where $q = 2^k$. We look at the parameter set NTRU-KEM-743, where $p = 3$, $q = 2048$, and $n = 743$; the arithmetic takes place in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$, but coefficients are also reduced modulo p when moving to $\mathcal{R}_p$. The optimizations in this work carry over to the smaller NTRU-KEM-443 parameter set, but not to NTRU-KEM-1024 (which uses a prime q). As in NTRU-HRSS, the relevant multiplication occurs when the noise polynomial r is multiplied with the public key h, but we also utilize our multiplication routine for the other multiplication in Dec.

```
Algorithm 38 NTRUEncrypt.KeyGen ()                                   $p, \mathcal{R}_q$
$f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
$h \leftarrow (p \cdot g)/(p \cdot f + 1) \in \mathcal{R}_q$
return $(\mathsf{pk} = h, \mathsf{sk} = (f, h))$
```

```
Algorithm 39 NTRUEncrypt.Enc ($m, h = \mathsf{pk}$)                     $p, \mathcal{R}_p, \mathcal{R}_q$
$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
$t \leftarrow r \circledast h$
$m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
$m' \leftarrow m - m_{mask} \in \mathcal{R}_p$
$c \leftarrow t + m'$
return $c$
```

Saber

Like Lizard and RLizard, Saber [DKR+17; DKR+18] also relies on the Learning-with-Rounding problem. Rather than directly targeting LWR or the ring variant, it positions itself in the middle-ground formed by the Module-LWR problem [BPR12; DKR+18]. Like RLizard, Saber operates in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, and in the smaller $\mathcal{R}_p$. Because of the Module-LWR structure, however, n is fixed to 256 for all parameter sets. Instead of varying the degree of the polynomial, Saber variants use matrices of varying sizes with entries in the polynomial ring (denoted $\mathcal{R}^{\ell \times \ell}$).

Algorithm 40 NTRUEncrypt.Dec $(c, (f, h) = \mathsf{sk})$ $p, \mathcal{R}_p, \mathcal{R}_q$

```
$m' \leftarrow f \circledast c \in \mathcal{R}_p$
$t \leftarrow c - m'$
$m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
$m \leftarrow m' + m_{mask} \in \mathcal{R}_p$
$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
if $p \cdot r \circledast h = t$ then
    return $m$
else
    return $\perp$
end if
```

With the fixed $q = 8192$, this ensures that an optimized routine for multiplication in $\mathcal{R}_q$ directly applies to the smaller LightSaber and the larger FireSaber instances as well. Other parameters p and t are powers of 2 smaller than q; for the Saber instance,[11] $p = 1024$ and $t = 8$. The vector h is a fixed constant in $\mathcal{R}_q^{\ell}$.

Note that some of the multiplications in Saber are in $\mathcal{R}_q$ and some are in $\mathcal{R}_p$ — in our software, both use the same routine. As we will explain in Section 5.4, the smaller value of p would in principle allow us to explore a larger design space for multiplications in $\mathcal{R}_p$, but for the small value of $n = 256$ there is nothing to be gained from using, e.g., higher-degree Toom-Cook instances.

Algorithm 41 Saber.KeyGen () $h, \ell, \mathcal{R}_p, \mathcal{R}_q$

```
$\rho \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho)$
$s \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$b \leftarrow \lfloor A \circledast s + h \rceil \in \mathcal{R}_p^{\ell}$
return $(\mathsf{pk} = (\rho, b), \mathsf{sk} = s)$
```

KINDI

In the same vein as Saber, Kindi [Ban17] is based on a matrix of polynomials, relating it to the Module-LWE problem [LS15]. Somewhat more intricate than the standard approach, it relies on a trapdoor construction, and constructs a CPA-

---

[11] Note that both the scheme and the category-3 parameter set are called Saber.

Algorithm 42 Saber.Enc $(m, (\rho, b) = \mathsf{pk})$ $p, \ell, \mathcal{R}_{2t}, \mathcal{R}_p, \mathcal{R}_q$

```
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho)$
$s' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$b' \leftarrow \lfloor A \circledast s' + h \rceil \in \mathcal{R}_p^{\ell}$
$v' \leftarrow b \circledast \lfloor s' \rceil \in \mathcal{R}_p$
$c_m \leftarrow \lfloor v' + (p/2) \cdot m \rceil \in \mathcal{R}_{2t}$
return $(c_m, b')$
```

Algorithm 43 Saber.Dec $((c_m, b'), s = \mathsf{sk})$ $p, t, \mathcal{R}_2, \mathcal{R}_p$

```
$v \leftarrow b' \circledast \lfloor s \rceil \in \mathcal{R}_p$
$m' \leftarrow \lfloor v - (p/(2t)) \cdot c_m + h \rceil \in \mathcal{R}_2$
return $m'$
```

secure PKE that is already close to a key-encapsulation mechanism. Kindi operates in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $q = 2^k$, the more general $\mathcal{R}_b = \mathbb{Z}_b[x]/(x^n + 1)$ for some integer b, and in the polynomial ring with integer coefficients $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$. The relevant arithmetic primarily happens in the ring $\mathcal{R}_q$, though, meaning that the performance of Kindi still considerably improves as a consequence of this work. We consider the parameter set Kindi-256-3-4-2, where $n = 256$ and $q = 2^{14}$. This also defines $g \in \mathcal{R}_q$, $\ell = 3$ and $p = 4$.

Algorithm 44 Kindi.KeyGen() $\ell, \mathcal{R}_q$

```
$\mu \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$r, r' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$b \leftarrow A \circledast r + r'$
return $(\mathsf{pk} = (b, \mu), \mathsf{sk} = (r, b, \mu))$
```

5.3.2 ARM Cortex-M4

Our target platform is the ARM Cortex-M4, which implements the ARMv7E-M architecture. See Section 2.4.2. We recall the most relevant properties here.

The architecture defines 16 general-purpose registers, of which 14 are freely usable by the developer. In contrast to smaller architectures like the Cortex-M3, the Cortex-M4 supports the DSP instructions `smuad`, `smuadx`, `smlad`, and `smladx`, which

Algorithm 45 Kindi.Enc $(m, (b, \mu) = \mathrm{pk})$ $g, \ell, n, p, \mathcal{R}_2, \mathcal{R}_p, \mathcal{R}_q$

```
$s_1 \leftarrow \mathsf{Sample}_{\mathcal{R}_2}$
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$\mathrm{p} \leftarrow b + g$
$\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
$(s_2, \dots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
$s \leftarrow (s_1 + 2 \cdot \bar{s_1} - [p], s_2 - [p], \dots, s_\ell - [p]) \in \mathcal{R}_q^\ell$
$\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
$u \leftarrow \bar{u} \oplus m$
$e \leftarrow (u_1 - [p], \dots, u_\ell - [p]) \in \mathcal{R}_q^\ell$
$e_{\ell+1} \leftarrow u_{\ell+1} - [p]$
$(c, c_{\ell+1}) \leftarrow (A \circledast s + e, \mathrm{p} \circledast s + g \cdot [p] + e) \in \mathcal{R}_q^{\ell+1}$
return $(c, c_{\ell+1})$
```

Algorithm 46 Kindi.Dec $((c, c_{\ell+1}), (r, b, \mu) = \mathrm{sk})$ $g, \ell, n, p, q, \mathcal{R}_2, \mathcal{R}_p, \mathcal{R}_q$

```
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$\mathrm{p} \leftarrow b + g$
$v \leftarrow c_{\ell+1} - c \circledast r$
$s_1 \leftarrow (\lfloor v_1 / 2^{\log q - 1} \rceil, \dots, \lfloor v_n / 2^{\log q - 1} \rceil) \in \mathcal{R}_2$
$\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
$(s_2, \dots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
$s \leftarrow (s_1 + 2 \cdot \bar{s_1} - [p], s_2 - [p], \dots, s_\ell - [p])$
$\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
$(e, e_{\ell+1}) \leftarrow (c - A \circledast s, c_{\ell+1} - \mathrm{p} \circledast s) \in \mathcal{R}_q^{\ell+1}$
$u \leftarrow (e_1 + [p], \dots e_\ell + [p])$
$u_{\ell+1} \leftarrow e_{\ell+1} + [p]$
$m \leftarrow u \oplus \bar{u}$
return $m$
```

Table 5.2: Dual 16-bit multiplication instructions supported by the ARM Cortex-M4.

instruction	semantics
`smuad Ra, Rb, Rc`	$\mathsf{Ra} \leftarrow \mathsf{Rb}_\mathsf{L} \cdot \mathsf{Rc}_\mathsf{L} + \mathsf{Rb}_\mathsf{H} \cdot \mathsf{Rc}_\mathsf{H}$
`smuadx Ra, Rb, Rc`	$\mathsf{Ra} \leftarrow \mathsf{Rb}_\mathsf{L} \cdot \mathsf{Rc}_\mathsf{H} + \mathsf{Rb}_\mathsf{H} \cdot \mathsf{Rc}_\mathsf{L}$
`smlad Ra, Rb, Rc, Rd`	$\mathsf{Ra} \leftarrow \mathsf{Rb}_\mathsf{L} \cdot \mathsf{Rc}_\mathsf{L} + \mathsf{Rb}_\mathsf{H} \cdot \mathsf{Rc}_\mathsf{H} + \mathsf{Rd}$
`smladx Ra, Rb, Rc, Rd`	$\mathsf{Ra} \leftarrow \mathsf{Rb}_\mathsf{L} \cdot \mathsf{Rc}_\mathsf{H} + \mathsf{Rb}_\mathsf{H} \cdot \mathsf{Rc}_\mathsf{L} + \mathsf{Rd}$

we use to significantly speed up low-degree polynomial multiplication using the schoolbook method. Those low-degree multiplication routines are used as a core building block for higher-degree polynomial multiplication. The DSP instructions perform two half-word multiplications, accumulate the two products and optionally add another 32-bit word in one clock cycle (as illustrated in Table 5.2). There is strong synergy between these DSP instructions and the fact that loading a 32-bit word using `ldr` is as expensive as loading a halfword using `ldrh`.

As we will be performing loops of the same operation across polynomials, we stress that it is important to perform load operations sequentially (i.e., uninterrupted by other instructions) to benefit from pipelining. This shows in the `ldm` instruction, but also when simply adjoining multiple `ldr` instructions. While the same behavior occurs for store instructions, combining loads and stores only incurs pipelining benefits when stores follow loads, but not when loads follow stores.

The ARMv7E-M instruction set contains support for 16-bit Thumb instructions, such as simple arithmetic and memory operations with register parameters. Using these instructions has an obvious benefit for code size, but comes at the cost of introducing misalignment: instruction fetching is significantly more expensive when instruction offsets are not aligned to multiples of four bytes. To combat this, Thumb instructions can be expanded to full-word width using the `.w` suffix.

In our experiments we use the STM32F407, featuring 1 MiB of Flash ROM, 192 KiB of RAM, and a maximum frequency of 168 MHz. See Section 2.4.2 for more details on this platform. For benchmarking, we use the reduced clock frequency of 24 MHz to not be impacted by wait states caused by slow memory [SS17]. We use the GNU ARM Embedded Toolchain[12] (arm-none-eabi) with arm-none-eabi-gcc-8.3.0, using the optimization flag -O3.

---

[12] `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm`

5.4 Multiplication in $\mathbb{Z}_{2^m}[x]$

Rather than implementing multiplication in $\mathcal{R}_q = \mathbb{Z}_q[x]/f$ for some $f \in \mathbb{Z}_q[x]$, we focus on non-reduced multiplication in $\mathbb{Z}_{2^m}[x]$. This is identical across all schemes we investigate — the reduction is done outside. At little extra cost, this greatly generalizes the usability of our results without introducing additional complexity.

In this section, we describe the way we break down such a multiplication for a specific number of coefficients n, modulo a specific q. This is done using combinations of Toom-Cook's and Karatsuba's multiplication algorithms. For a given n and q, there are multiple possible approaches; we explore the entire space and select the optimum for each parameter set. We use Python scripts that generate optimized assembly functions for all combinations, for arbitrary-degree polynomials (with degree below 1024). These scripts are parameterized by the degree, the variant of Toom's method (Toom-3, Toom-4, both Toom-4 and Toom-3 or no Toom layer at all), and the threshold at which to switch from recursive Karatsuba to schoolbook multiplication. We analyze these results in Section 5.5.1.

5.4.1 Revisiting Karatsuba and Toom-Cook

The multiplication algorithms by Karatsuba [KO63] and Toom [Too63; Coo66] were briefly introduced in Section 5.2.1. We now revisit specific instances in the context of multiplications in $\mathbb{Z}_{2^m}[x]$. For this work, we make use of Toom-3 and Toom-4, and recursive application of degree-2 Karatsuba. While Toom-4 is more efficient than Toom-3 in the asymptotics (which, in turn, outperforms Karatsuba), in practice the additional additions and subtractions make it non-trivial to reason about the cut-off points. In particular, the expensive and complex memory-access patterns make this hard to analyze.

Toom-Cook

It is important to note that there is a loss in precision when using Toom's method, as it involves division over the integers. While divisions by three and five can be replaced by multiplications by their inverses modulo 2^{16}, i.e., 43691 and 52429, this is not possible for divisions by powers of two. Consequently, for our choice of evaluation points, Toom-3 loses one bit of precision, and Toom-4 loses three bits. Since our Karatsuba and schoolbook implementations operate in $\mathbb{Z}_{2^{16}}[x]$, this imposes constraints on the values of q for which our implementations can be used; Toom-3 can be used for $q \leq 2^{15}$, Toom-4 can be used for $q \leq 2^{13}$. These losses accumulate, and a combination of both is only possible if $q \leq 2^{12}$. This also rules out higher-order Toom methods. While switching to 32-bit arithmetic would allow using higher-order Toom, this slows down Karatsuba and the schoolbook multiplications significantly by increasing load-store overhead and ruling out DSP instructions.

We reiterate that the runtime is significantly impacted by the additions and subtractions, as well as the increased and more intricate memory-access patterns. This makes it not immediately obvious in general which degree of Toom-Cook is the fastest for a given n. We first evaluate whether to decompose using a layer of Toom-4, Toom-3, both Toom-4 and Toom-3, or no Toom at all. We then repeatedly apply Karatsuba's method to break down the multiplications further. Through trial and error, this continues up to the threshold at which the 'naive' schoolbook method becomes the fastest approach.

Karatsuba

The call to the topmost Karatsuba layer is a function call, but from that point on, we recursively inline the separate layers. Upon reaching the threshold at which the schoolbook approach takes precedence, we jump to the schoolbook multiplication as an explicit subroutine. This provides a trade-off that keeps code size reasonable and is flexible to implement and experiment with, but does imply that the register allocation between the final Karatsuba layer and the underlying schoolbook is disjoint. It may be worthwhile to further optimize this for specific n. Note that we only apply Karatsuba's method at degree-2, and also do not combine operations across recursive calls. See [WP06] for details on a more general approach.[13]

As we perform several nested layers of Karatsuba multiplication, it is important to carefully manage memory usage. We do not go for a largely in-place approach, as is done in [KMR+18], but instead allocate stack space for the sums of the high and low limbs, relying on the input and output buffers for all other terms. This leads to effective memory usage without paying with performance.

---

[13] The approach by Weimerskirch and Paar provides a middle ground between Karatsuba and Toom-Cook. While allowing for a wider range of splits than traditional Karatsuba and a more efficient way of dealing with the newly introduced additions, it does come at the cost of more small-sized multiplications than similarly-sized Toom-Cook instances. A key advantage, though, is the fact that this approach does not introduce divisions that lead to a loss of precision. This could be relevant in particular for multiplications where both n and q are large.

Assembly-level optimizations

For both Toom and Karatsuba, most operations involve adding and subtracting polynomials of moderate size from a given address. We stress the importance of careful pipelining, loading and storing 16-bit coefficients pairwise into full-word registers, and using `uadd16` and `usub16` arithmetic operations. We rely on offset-based instructions for memory operations, in particular for the more intricate memory-access patterns in Toom and Karatsuba. This leads to a slight increase in code size compared to using `ldm` and `stm`, (and some bookkeeping for polynomials exceeding the maximal offset of 4095 bytes), but ensures that addresses can be computed during code generation.

For ease of implementation, our code generator for Toom is restricted to dimensions that divide without remainder. For Karatsuba, we do not restrict the dimensions at all: the implementation can work on unbalanced splits, and thus polynomials of unequal length. In order not to waste any memory or cycles here (e.g., by applying common refinement approaches), the Python script becomes a rather complex composition of conditionals. Rather than trying to combine pairs of 16-bit additions into `uadd16` operations on the fly, we run a post-processing step over the scheduled instructions to do so.

Similarly, instead of considering alignment to 32-bit word boundaries during code generation, we use a post-processing step. After compilation, we disassemble the resulting binary and expand Thumb instructions in the cases where they cause misalignment. This allows us to use the smaller Thumb instructions where possible, but avoids paying the overhead of misalignment. In particular, this is important when an odd number of Thumb instructions is followed by a large block of 32-bit instructions. The alignment post-processing is done using a Python script that is included with our code generation software, and may be of independent interest.

5.4.2 Small schoolbook multiplications

We carefully investigate several approaches to perform the small-degree schoolbook multiplications that underlie Karatsuba and Toom-Cook, varying the approaches and implementing distinct generation routines for different n.

For each approach, we keep the polynomial in packed representation, loading coefficients into the 32-bit registers in pairs. The ARMv7E-M instruction set provides multiplication instructions that efficiently operate on data in this format.

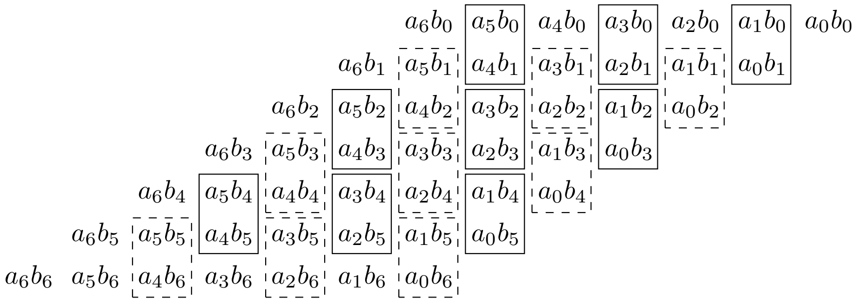

a_6b_0 a_5b_0 a_4b_0 a_3b_0 a_2b_0 a_1b_0 a_0b_0
a_6b_1 a_5b_1 a_4b_1 a_3b_1 a_2b_1 a_1b_1 a_0b_1
a_6b_2 a_5b_2 a_4b_2 a_3b_2 a_2b_2 a_1b_2 a_0b_2
a_6b_3 a_5b_3 a_4b_3 a_3b_3 a_2b_3 a_1b_3 a_0b_3
a_6b_4 a_5b_4 a_4b_4 a_3b_4 a_2b_4 a_1b_4 a_0b_4
a_6b_5 a_5b_5 a_4b_5 a_3b_5 a_2b_5 a_1b_5 a_0b_5
a_6b_6 a_5b_6 a_4b_6 a_3b_6 a_2b_6 a_1b_6 a_0b_6

Figure 5.2: Pairing coefficients to reduce the number of multiplications using `smladx` / `smlad`. Dashed boxes represent multiplications involving repacked b.

It includes the discussed parallel multiplications, but also instructions that operate only on specific halfwords.

For $n \leq 10$, all input coefficients can be kept in registers simultaneously, with several registers remaining to keep the pointers to the source and destination polynomials around. We first compute all coefficients of terms with odd exponents, before using `pkh` instructions to repack one of the input polynomials and computing the remaining coefficients. This ensures that the vast majority of the multiplications can be computed using the two-way parallel multiply-accumulate dual instructions. See Figure 5.2 for an illustration of this — here, b is repacked to create the dashed pairs. This is somewhat similar to the approach used in [KMR+18], but ends up needing less repacking and memory interaction.

For $n = 11$ and $n = 12$, we spill the source pointers to the stack after loading the complete polynomials. At these dimensions the registers are used to their full potential, and by using the DSP instructions we end up needing only 78 multiplication instructions: 66 combined multiplications, 12 single multiplications, and not a single dedicated addition instruction. This offsets the extra cost of the six packing instructions considerably. For $n = 13$ and $n = 14$, not all coefficients fit in registers at the same time. This leads to spills for the middle columns (i.e., the computation of coefficients around x^n, which are affected by all input coefficients). Even when using the Python abstraction layer, manual register allocation becomes somewhat tedious in the cases that involve many spills to the stack. We remedy this by using bare-bone register-allocation functions akin to the scripts in [HRS+17a].

For larger n, the above strategy leads to an excessive amount of register spills. Instead, we compose the multiplication of a grid of smaller instances. For 15 ≤

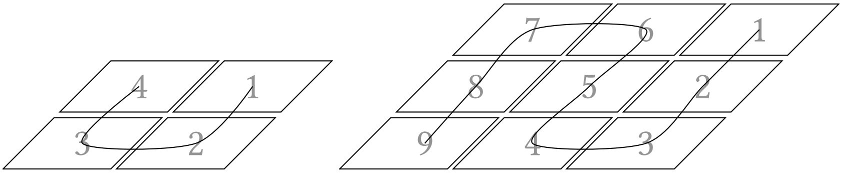


Figure 5.3: Decomposing larger schoolbook multiplications

$n \leq 24$, we compose the multiplication out of four smaller multiplications, for $25 \leq n \leq 36$, we use a grid of nine multiplications, et cetera. Note that we use at most $n = 12$ for the building blocks, given the extra overhead of the register spills for $n \in \{13, 14\}$. We further remark that it is important to carefully schedule the (re)loading and repacking of input polynomials. We illustrate this in Figure 5.3.

The approach described above works trivially when n is divisible by $\lceil \frac{n}{12} \rceil$, but leads to a less symmetric pattern for other dimensions. We plug these holes by starting from a value of n that divides evenly, and either adding a layer 'around' the parallelogram or nullifying the superfluous operations by post-processing.

Figure 5.4 and Table 5.3 illustrate the performance of these routines.

5.5 Measuring multiplication performance

In this section we present benchmark results for polynomial multiplication, and for key generation, encapsulation, and decapsulation of the round-1 versions of Kindi, NTRUEncrypt, NTRU-HRSS, RLizard, and Saber. We attempt to select parameter sets that target NIST security category 3, but have to make an exception for NTRU-HRSS and Kindi. The NTRU-HRSS submission only provides a single parameter set, rated at category 1. For Kindi, the reference implementations of the category 3 parameter sets require more than 128 KiB of RAM, and thus do not trivially fit our platform. We use Kindi-256-3-4-2 instead, which targets security category 1. See Section 2.2 for details on the NIST security categories.

All cycle counts presented in this section were obtained by using an adapted version of the `pqm4` benchmarking framework [KRS+18], which uses the built-in 24-bit hardware timer. We also obtain memory-usage measurements using the measurement code from `pqm4`. This is done by writing a canary to the entire stack space, running the scheme under test and subsequently checking how much of the canary was overwritten.

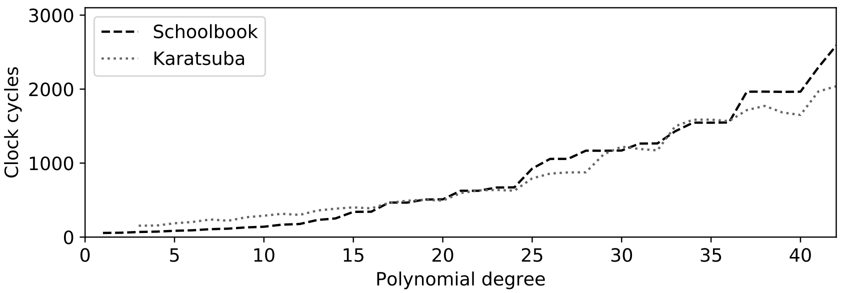


Figure 5.4: Runtime of generated optimized polynomial multiplication for small n. For $n < 20$ our hand-optimized schoolbook multiplications are clearly superior, for $n > 36$ first applying at least one layer of Karatsuba is faster.

5.5.1 Isolated multiplications

We first present performance results for polynomial multiplication as a building block. We obtain benchmarks for the multiplication for all possible $n < 1024$, iterating over the different alternatives to find the optimal decomposition.

Figure 5.4 shows the runtime of our hand-optimized schoolbook implementations and the generated optimized Karatsuba code for small n. For the Karatsuba benchmarks, we provide measurements at the optimal optimal ‘schoolbook threshold’. To illustrate this, consider $n = 32$. Here, we can either apply one layer of Karatsuba and then use the schoolbook method for $n = 16$ or, alternatively, use two layers of Karatsuba and use schoolbook multiplications at $n = 8$. The former variant is faster in this scenario, which leads to a schoolbook threshold of 16. The graph shows that directly applying the schoolbook method is optimal for $n < 20$, and that Karatsuba outperforms schoolbook for $n > 36$. For values in between these bounds, the plot is inconclusive. To some extent, this can be attributed to the amount of hand-optimization that went into some of the more relevant schoolbook implementations. More importantly, it is strongly influenced by register pressure: there is a large performance hit in the step from $n = 14$ to $n = 15$, which then propagates to dimensions that break down to these schoolbook multiplications after applying Karatsuba. For dimensions relevant in the decomposition of the multiplications used in the schemes, we found that the cross-over point is at $n = 22$, i.e., for values $n > 22$ one should use an additional layer of Karatsuba.

Table 5.3: Benchmarks for small schoolbook multiplication routines. The cycle counts include an overhead of approximately 50 cycles for benchmarking.

n	cycles	n	cycles	n	cycles	n	cycles
1	56	13	232	25	926	37	1 965
2	59	14	252	26	1 057	38	1 966
3	69	15	341	27	1 057	39	1 963
4	74	16	343	28	1 168	40	1 965
5	85	17	467	29	1 167	41	2 294
6	92	18	466	30	1 170	42	2 588
7	107	19	508	31	1 264	43	2 595
8	114	20	510	32	1 266	44	2 594
9	131	21	626	33	1 431	45	2 824
10	140	22	626	34	1 547	46	2 825
11	168	23	670	35	1 546	47	2 822
12	177	24	672	36	1 549	48	2 824

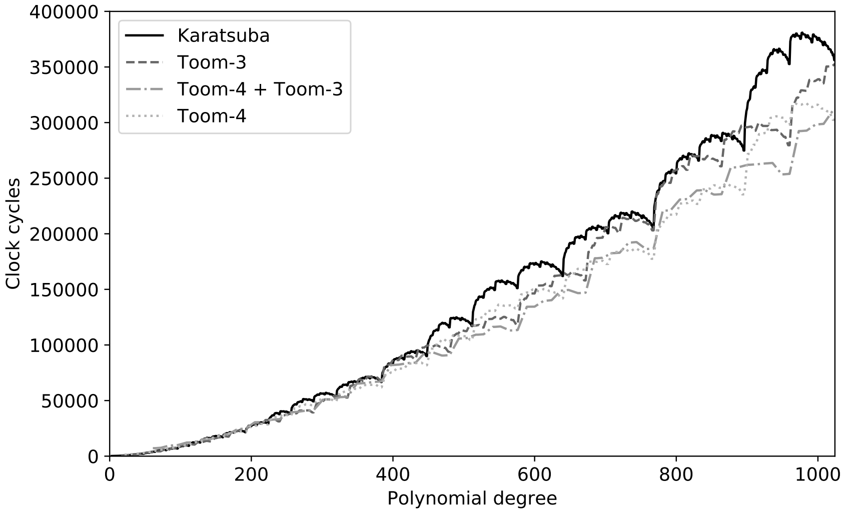


Figure 5.5: Runtime of different decomposition variants for multiplications of polynomials of large degree.

Figure 5.5 shows the performance of the different multiplication approaches for larger n. While the general trend is visible, we still observes a jagged line. We speculate that the main cause for this is similar to the irregularities in Figure 5.4: the variance in the increasing cost of the schoolbooks is magnified as n grows larger and specific schoolbook sizes are repeated in the decomposition of large multiplications. Because of the difference in decomposition between Toom-3 and Toom-4, this favors each method for different ranges of n, resulting in alternating optimality. The specific decomposition choices also strongly affect the memory-access patterns and, by extension, data alignment, sometimes resulting in a large performance penalty. In practice, comparing benchmarks for specific n seems to be the only way to come to conclusive results. We observe that the lines are not even monotonically increasing — it is, of course, trivial to pad a smaller-degree polynomial and use a larger multiplication routine to benefit of a more efficient decomposition. Through careful post-processing, one can then eliminate many redundant instructions while retaining the apparently beneficial decompositions.

As Figure 5.5 does not allow us to identity which method performs best for clear bounds on n, we focus on individual n as relevant for the five cryptographic schemes we intend to cover. This restricts n to $\{256, 701, 743, 1024\}$. We report the cycle counts for each method in Table 5.4, as well as the additional stack usage. All cycle counts are for multiplication *excluding* subsequent reduction needed to obtain an n-coefficient polynomial; the additional cost of reduction differs depending on the specific choice of ring. While there is some performance benefit to performing the reduction inline, the main improvement is in memory usage. For the Toom variants, this would allow for in-place recomposition of the output polynomial, reducing stack usage by roughly $2n$ coefficients. We leave this for future work.

For the rather small $n = 256$ (Saber, Kindi), we already see that Toom-4 followed by two layers of Karatsuba is slightly faster than directly applying Karatsuba. As the difference is small, one might decide to not use a Toom layer at all, at the benefit of a much simpler implementation and considerably reduced stack usage. Toom-4 is not suitable for Kindi ($n = 256, q = 2^{14}$), as q is too large. Again the impact is marginal, though, as Karatsuba is only a few percent slower at this dimension, also outperforming Toom-3 by a small difference. For larger $n \in \{701, 743, 1024\}$ (NTRU-HRSS, NTRUEncrypt,RLizard) applying Toom-4 is most efficient. The second layer ends up in the same range as the smaller n, where it is again a close competition between applying Toom-3 or directly switching to recursive Karatsuba.

Table 5.4: Benchmarks for polynomial multiplication, excluding reduction. The fastest approach is marked with a ▷ symbol. The ‘Toom-4’ and ‘Toom-4 + Toom-3’ approaches are not applicable to all parameter sets, as q may be too large.

		approach	threshold	cycles	stack
Saber ($n = 256$, $q = 2^{13}$)		Karatsuba only	16	38 000	2 020
		Toom-3	11	39 043	3 480
	▷	Toom-4	16	36 274	3 800
Kindi-256-3-4-2 ($n = 256$, $q = 2^{14}$)	▷	Karatsuba only	16	38 000	2 020
		Toom-3	11	39 043	3 480
NTRU-HRSS ($n = 701$, $q = 2^{13}$)		Karatsuba only	11	202 889	5 676
		Toom-3	15	205 947	9 384
	▷	Toom-4	11	172 882	10 596
NTRU-KEM-743 ($n = 743$, $q = 2^{11}$)		Karatsuba only	12	217 130	6 012
		Toom-3	16	211 588	9 920
	▷	Toom-4	12	186 639	11 208
		Toom-4 + Toom-3	16	192 503	12 152
RLizard-1024 ($n = 1024$, $q = 2^{11}$)		Karatsuba only	16	356 046	8 188
		Toom-3	11	352 770	13 756
	▷	Toom-4	16	302 504	15 344
		Toom-4 + Toom-3	11	310 712	16 816

5.5.2 Encapsulation and decapsulation

We now present performance results for RLizard, Saber, Kindi, NTRUEncrypt, and NTRU-HRSS. All the software described in this section started from the reference implementations submitted to round 1 of NIST's Post-Quantum Cryptography Standardization project, but went considerably further than just replacing the multiplication routines with the optimized routines described in Section 5.4. For Saber, we considered starting from the already optimized implementation by Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede [KMR+18], but achieved marginally better performance by starting from the reference code. We start by describing the changes that apply to the reference implementations; some of these changes might be more generally advisable as updates to reference software. Where relevant for round 2 of NIST's Post-Quantum Cryptography Standardization project, we have brought the suggestions to the attention of the respective submitters.

Memory allocations

The reference implementations of Kindi, RLizard, and NTRUEncrypt make use of dynamic memory allocation on the heap. The RLizard implementation does not free all the allocated memory, which results in memory leaks; also it misinterprets the NIST API and assumes that the public key is always stored right behind the secret key. This may result in reads from uninitialized (or even unallocated) memory. Luckily none of the implementations *require* dynamically allocated memory; the sum of all allocated memory is reasonably small and known at compile time. We eliminated all dynamic memory allocations and only rely on the stack to store temporary data, significantly improving performance.

Hashing

Each of the five schemes makes use of variants of SHA-3, SHAKE [NIST15b], or SHA-512 [NIST15a]. We attempt to eliminate the influence of difference in implementation quality for the symmetric primitives by making use of the same implementations for all schemes. For SHA-3 and SHAKE we use the optimized assembly implementation available in `pqm4` [KRS+18], which makes use of the optimized Keccak-permutation from the Keccak Code Package [DHP+]. For SHA-512, we use a reference C implementation from SUPERCOP [BL] that proved to be hard to outperform with hand-written assembly.

Table 5.5: Benchmarks for the five schemes targeted in this work.

	implementation	clock cycles		stack usage	
Saber	NIST reference	*K:*	6 530k	*K:*	12 616
		E:	8 684k	*E:*	14 896
		D:	10 581k	*D:*	15 992
	[KMR+18]	*K:*	1 147k	*K:*	13 883
		E:	1 444k	*E:*	16 667
		D:	1 543k	*D:*	17 763
	This work	*K:*	895k	*K:*	13 248
		E:	1 161k	*E:*	15 528
		D:	1 204k	*D:*	16 624
Kindi-256-3-4-2	NIST reference	*K:*	21 794k	*K:*	59 864
		E:	28 176k	*E:*	71 000
		D:	37 129k	*D:*	84 096
	This work	*K:*	969k	*K:*	44 264
		E:	1 320k	*E:*	55 392
		D:	1 517k	*D:*	64 376
NTRU-HRSS	NIST reference	*K:*	205 156k	*K:*	10 020
		E:	5 166k	*E:*	8 956
		D:	15 067k	*D:*	10 204
	This work	*K:*	145 963k	*K:*	23 396
		E:	404k	*E:*	19 492
		D:	819k	*D:*	22 140
NTRU-KEM-743	NIST reference	*K:*	59 815k	*K:*	14 148
		E:	7 540k	*E:*	13 372
		D:	14 229k	*D:*	18 036
	This work	*K:*	5 198k	*K:*	25 320
		E:	1 601k	*E:*	23 808
		D:	1 881k	*D:*	28 472
RLizard-1024	NIST reference	*K:*	26 423k	*K:*	4 272
		E:	32 156k	*E:*	10 532
		D:	53 181k	*D:*	12 636
	This work	*K:*	525k	*K:*	27 720
		E:	1 345k	*E:*	33 328
		D:	1 716k	*D:*	35 448

Table 5.6: Benchmarks on the Cortex-M4 for other KEMs submitted to NIST's Post-Quantum Cryptography Standardization project.

	implementation	clock cycles		stack usage	
R5ND_1PKEb	[SBGM+18]	*K:*	658k	*K:*	?
		E:	984k	*E:*	?
		D:	1 265k	*D:*	?
R5ND_3PKEb	[SBGM+18]	*K:*	1 032k	*K:*	?
		E:	1 510k	*E:*	?
		D:	1 913k	*D:*	?
NewHopeCCA1024	[KRS+18; AJS16]	*K:*	1 244k	*K:*	11 152
		E:	1 963k	*E:*	17 448
		D:	1 979k	*D:*	19 648
Kyber768	[KRS+18]	*K:*	1 200k	*K:*	10 544
		E:	1 446k	*E:*	13 720
		D:	1 477k	*D:*	14 880

Comparison to reference code

Table 5.5 lists benchmarks for the optimized implementations as well as for the reference implementations with the modifications described above. We dramatically increase the performance for all schemes covered by this work; the improvements go up to a factor of 49 for the key generation of RLizard-1024. Since both Karatsuba and Toom-Cook require storing additional intermediate polynomials on the stack, we increase stack usage for all schemes except Kindi-256-3-4-2. The reference implementations of Kindi-256-3-4-2 already contained optimized polynomial multiplication methods, which did not take stack usage into account.

Comparison to previous results

To the best of our knowledge, at the time of writing, Saber was the only scheme of the five schemes targeted in this work that had been optimized for the ARM Cortex-M family in previous work [KMR+18]. Our optimized implementation outperforms this CHES 2018 implementation by 22% for key generation, 20% for encapsulation, and 22% for decapsulation. Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede report 65 459 clock cycles for their optimized 256-coefficient polynomial multiplication, but we should note that their polynomial multiplication includes the reduction. Including the reduction, our multiplication requires 38 215 clock cycles, which is an improvement of 42%. On a more granular level, they claim 587 cycles for 16-coefficient schoolbook multiplication, while we require only 343 cycles (see Table 5.3; this includes approximately 50 cycles for benchmarking).

Several other NIST candidates have been evaluated on Cortex-M4 platforms. We list their performance results in Table 5.6 for comparison. Most recently, record-setting results were published for Round5 on Cortex-M4 [SBGM+18]. The fastest scheme described in our work, targeting NIST security category 1, NTRU-HRSS, is 59% faster for encapsulation and 35% faster for decapsulation compared to the corresponding CCA variant of Round5 at the same security level. The key generation of NTRU-HRSS is considerably slower, but its inversions have not been optimized yet. The fastest scheme described here that targets NIST security category 3, Saber, is 13% faster for key generation, 23% faster for encapsulation, and 37% faster for decapsulation. We also make note of the optimized implementations of NewHopeCCA1024 [KRS+18; AJS16] and Kyber768 [KRS+18]. Both implementations are outperformed by NTRU-HRSS and Saber.

Side-channel resistance

While side-channel resistance was not a focus of this work, we ensure that our polynomial multiplication is protected against timing attacks. More specifically, in the multiplication routines we avoid all data flow from secrets into branch conditions and into memory addresses. The special multiplication routine in [SBGM+18] is less conservative and *does* use secret-dependent lookup indices with a reference to [ARMc] saying that the Cortex-M4 does not have internal data caches. However, it is not clear to us that really all Cortex-M4 cores do not have any data cache; [ARMc] states that the *"Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory. However, it is possible for a SoC design to integrate a system level cache."* Also, it *is* clear that some ARMv7E-M processors (for example, the ARM Cortex-M7) have data caches. Our multiplication code is also protected against timing attacks on those devices.

Key-generation performance

The focus of this work is to improve performance of encapsulation and decapsulation. All key-encapsulation mechanisms considered in this work are CCA-secure, so the impact of a poor key-generation performance can in principle be minimized by caching ephemeral keys for some time. Such caching of ephemeral keys makes software more complex and in some cases also requires changes to higher level protocols; we therefore believe that key-generation performance, also for CCA-secure KEMs, remains an important target of optimization. The key generation of RLizard, Saber, and Kindi is rather straight-forwardly optimized by integrating our fast multiplication. The key generation of NTRUEncrypt and NTRU-HRSS also requires inversions, which we did not optimize in this work. We believe that further research into efficient inversions for those two schemes will significantly improve their key-generation performance on this platform.

5.5.3 Profiling optimized implementations

The speedup achieved by optimizing polynomial multiplication clearly shows that it vastly dominates the runtime of reference implementations. Having replaced this core arithmetic operation with highly optimized assembly, we analyze how much time the optimized implementations still spend in non-optimized code. This reflects how much performance could still be gained by hand-optimizing scheme-

Table 5.7: Time spent in multiplication, hashing, and sampling randomness.

scheme		total	polymul		hashing		random
Saber	*K:*	895k	327k	(37%)	475k	(53%)	2.0k
	E:	1 161k	435k	(38%)	615k	(53%)	0.6k
	D:	1 204k	544k	(45%)	500k	(42%)	0
Kindi-256-3-4-2	*K:*	969k	342k	(35%)	409k	(42%)	1.2k
	E:	1 320k	456k	(35%)	604k	(46%)	0.6k
	D:	1 517k	570k	(38%)	603k	(40%)	0
NTRU-HRSS	*K:*	145 963k	1 556k	(1%)	80k	(<1%)	0.6k
	E:	404k	173k	(43%)	107k	(26%)	0.6k
	D:	819k	519k	(63%)	67k	(8%)	0
NTRU-KEM-743	*K:*	5 198k	1 680k	(32%)	0		85k
	E:	1 601k	187k	(12%)	1 171k	(73%)	46k
	D:	1 881k	373k	(20%)	1 172k	(63%)	0
RLizard-1024	*K:*	525k	303k	(58%)	0		123k
	E:	1 345k	605k	(45%)	628k	(47%)	2.2k
	D:	1 716k	908k	(53%)	628k	(36%)	0

specific procedures. To this end, we measure the clock cycles spent in polynomial multiplication, hashing, and random number generation. Table 5.7 shows that still a considerable portion of encapsulation and decapsulation is spent in polynomial multiplication, but cycles consumed by hashing and randomness generation become much more prominent. For encapsulation, hashing (SHA-3 and SHA-2) dominates the runtime of Kindi-256-3-4-2, NTRU-KEM-743, and Saber. We have replaced these primitives with the fastest implementations available, but all schemes still spend a substantial number of clock cycles computing hashes. This is partly due to the Fujisaki-Okamoto transformation used to achieve CCA security, but also because of sampling of random numbers. Kindi-256-3-4-2, NTRU-HRSS, and Saber do not make use of `randombytes` extensively, but sample a small seed and then expand this using SHAKE. RLizard-1024 and NTRU-KEM-743 directly sample their randomness using `randombytes`. As we implement `randombytes` using the hardware random number generator on the STM32F407, this is more efficient than using SHAKE to expand a seed — this is, of course, highly platform-specific. There are, however, important caveats to consider when only using the hardware random number generator. In particular, it is unclear what the cryptographic properties of such an RNG are, and how this affects the security of the various schemes, especially considering that most reveal randomness as part of the CCA transform.

Outlook

Even though it has been almost fifteen years since the first workshops on post-quantum cryptography, the field is still very much in its infancy and exploratory research is very much ongoing. Depending on the specific subfield, this varies from theoretical novelty to deployment-focused experiments and standardization. Perhaps the most important direction for future work lies in cryptanalysis — in particular when accounting for non-generic adversaries with access to a quantum computer. As the main focus of this thesis is primarily on engineering efforts, we instead highlight practical considerations within the discussed subfields.

Hash-based signatures have a long legacy, but only really caught popular interest in recent years. As post-quantum cryptography is becoming more relevant, their practical use is a real consideration. While the constructions are well-understood and quite carefully tweaked, and there seems to be little purely academic work remaining, there is a lot to be done in terms of deployment. It would be very valuable to evaluate concrete deployment experiences with regards to stateful variants (i.e., XMSS with different tree traversal and state management strategies), as well as actual use-case driven parameter choices for the SPHINCS$^+$ framework. While [BHK+19] provides a starting point towards more flexible parameter selection, real-world use may present more specific bottlenecks (e.g., because of properties of communication protocols) that allow for more informed trade-offs.

Even when SPHINCS$^+$ is standardized as-is, individual users might find it worthwhile to make adjustments. In particular, this can include parameter sets that target a smaller number of total signatures (and thus reduce the required hypertree height), but also more structural changes such as the dismissed ‘larger top tree’ and ‘temporarily-stateful batch signing’ constructions. We refer to the latest version of the specification [BDE+17] (in particular Section 8.2) for some discussion on these omissions, but welcome a more thorough experimental exploration.

It will be interesting to see what the effect of broader availability of hash functions in hardware will be, especially when Keccak constructions become more commonplace. On a related (but effectively orthogonal) note, and perhaps only tangentially related to this work, hash-based signatures present an interesting application for efficient short-length hash functions. Haraka is an important starting point in this line of work, but has arguably not yet seen sufficient use or cryptanalysis to warrant precedence over SHA-256 or SHAKE256.

We should, however, be careful to strike a balance between standard-wide improvements and local, deployment-specific optimizations. Consider, e.g., that SHA-512/256 is likely preferable over SHA-256 on some 64-bit platforms, but not on smaller processors. With the current specification proposing 36 parameter sets across four dimensions, the framework is highly susceptible to a proliferation of incompatible implementations and deployments.

The other discussed digital signature variant is perhaps a bit further away from immediate deployment. In parallel with this work, a paper was publicized [KZ19] describing a forgery attack against MQDSS, considerably reducing the claimed security of the proposed instances. While accompanied by a security proof, this shows the necessity for tightness: the margin in the proof allows for attacks like these to be overlooked. In general, there is ample space for future work in the area of tight proofs for signature schemes in the ROM and QROM, as well as attacks such these that allow us to assess the degree to which such proof gaps are artificial or a real concern. Recent work on proofs for the Fiat-Shamir transform in the QROM takes important steps in this direction [KLS18; LZ19; DFM+19].

Off on a tangent: IDS-based $\mathcal{MQ}$ signatures on small devices are a missing data point. The arithmetic pairs well with vector units, but may be costly without.

Arguably the most active subfield by any metric is that of lattice-based primitives. Their efficiency successfully attracted several notable real-world experiments. From an implementor's perspective, the recent work on applying the number-theoretic transform to NTRU [LS19] is highly relevant, but one could argue that, moving forward, performance is no longer the main concern — however easy it may be to measure and compare. Instead, one would hope for broader consensus on security considerations and how to objectively assess different structure and parameter choices. This includes the actual, computational relation to lattice problems, but, perhaps more concretely, a better understanding of side-channel leakage and its effects on reducing the difficulty of solving lattice problems, real-world effects of decryption failures, and secure random sampling.

Bibliography

[AAB+17] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. *NewHope: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. 2017. URL: `https://newhopecrypto.org` (cit. on p. 191).

[ABB+15] Erdem Alkim, Nina Bindel, Johannes Buchmann, and Özgür Dagdelen. *TESLA: Tightly-Secure Efficient Signatures from Standard Lattices.* IACR Cryptology ePrint Archive, Report 2015/755. 2015. URL: `https://eprint.iacr.org/2015/755/20161117:055833` (cit. on pp. 146, 163).

[ABB+17] Erdem Alkim, Nina Bindel, Johannes Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. "Revisiting TESLA in the quantum random oracle model." In: *Post-Quantum Cryptography – PQCrypto 2017.* Ed. by Tanja Lange and Tsuyoshi Takagi. Vol. 10346. LNCS. Springer, 2017, pp. 143–162. URL: `https://eprint.iacr.org/2015/755` (cit. on p. 163).

[ABD+17] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS–Kyber: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. 2017. URL: `https://pq-crystals.org/kyber` (cit. on p. 191).

[ADP+16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum key exchange – a new hope." In: *Proceedings of the 25th USENIX Security Symposium.* Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016. URL: `https://eprint.iacr.org/2015/1092` (cit. on pp. 64, 171, 174, 189–191).

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. *Gravity-SPHINCS*. Submission to the NIST Post-Quantum Cryptography Standardization project. 2017. URL: `https://sphincs.org` (cit. on pp. 88, 102, 111).

[AE18] Jean-Philippe Aumasson and Guillaume Endignoux. “Improving stateless hash-based signatures.” In: *Topics in Cryptology – CT-RSA 2018*. Ed. by Nigel P. Smart. Vol. 10808. LNCS. Springer, 2018, pp. 219–242. URL: `https://eprint.iacr.org/2017/933` (cit. on pp. 88, 102).

[AHM+08] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. “SHA-3 proposal BLAKE.” In: *Submission to the NIST hash function competition* (2008). URL: `https://131002.net/blake/blake.pdf` (cit. on pp. 90, 93, 100).

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. “A new hope on ARM Cortex-M.” In: *Security, Privacy, and Advanced Cryptography Engineering – SPACE 2016*. Ed. by Claude Carlet, Anwar Hasan, and Vishal Saraswat. Vol. 10076. LNCS. Springer, 2016, pp. 332–349. URL: `https://eprint.iacr.org/2016/758` (cit. on pp. 210, 211).

[AMP10] Elena Andreeva, Bart Mennink, and Bart Preneel. “Security Reductions of the Second Round SHA-3 Candidates.” In: *Information Security – ISC 2010*. Ed. by Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilić. Vol. 6531. LNCS. Springer, 2010, pp. 39–53. URL: `https://eprint.iacr.org/2010/381` (cit. on p. 27).

[APR19] Pol Van Aubel, Erik Poll, and Joost Rijneveld. “Non-Repudiation and End-to-End Security for EV-charging.” In: *Innovative Smart Grid Technologies Europe 2019. To appear.* IEEE, 2019 (cit. on p. 259).

[ARMa] ARM Limited. *Cortex-M0 Processor – ARM*. URL: `http://www.arm.com/products/processors/cortex-m/cortex-m0.php` (cit. on pp. 41, 92).

[ARMb] ARM Limited. *Cortex-M3 Technical Reference Manual. Document ID: ARM DDI 0337E*. 2005. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e` (cit. on p. 38).

[ARMc] ARM Limited. *ARM Cortex-M Programming Guide to Memory Barrier Instructions.* 2012. URL: `https://static.docs.arm.com/dai0321/a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf` (cit. on p. 212).

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. "Quantum attacks on classical proof systems: The hardness of quantum rewinding." In: *Annual Symposium on Foundations of Computer Science – FOCS 2014.* IEEE. 2014, pp. 474–483. URL: `https://eprint.iacr.org/2014/296` (cit. on p. 136).

[Ban17] Rachid El Bansarkhani. *KINDI: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. 2017. URL: `http://kindi-kem.de` (cit. on pp. 191, 195).

[Bar86] Paul Barrett. "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor." In: *Advances in Cryptology – CRYPTO '86.* Vol. 263. LNCS. Springer. 1986, pp. 311–323 (cit. on p. 181).

[BBH13] Christoph Busold, Johannes Buchmann, and Andreas Hülsing. "Forward secure signatures on smart cards." In: *Selected Areas in Cryptography – SAC 2012.* Ed. by Lars R. Knudsen and Huapeng Wu. Vol. 7707. LNCS. Springer. 2013, pp. 66–80. URL: `https://eprint.iacr.org/2018/924` (cit. on pp. 61, 72, 78, 91, 92).

[BCC+10] 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: *Cryptographic Hardware and Embedded Systems – CHES 2010.* Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. LNCS. Springer, 2010, pp. 203–218. URL: `https://eprint.iacr.org/2010/313` (cit. on p. 157).

[BCC+14] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. "How to manipulate curve standards: a white paper for the black hat." In: *Security Standardization Research – SSR*

2015. Vol. 9497. LNCS. Springer, 2014, pp. 109–139. URL: `https://eprint.iacr.org/2014/571` (cit. on p. 130).

[BCD+16] Joppe Bos, Craig Costello, Leo 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: *Conference on Computer and Communications Security – CCS '16*. Ed. by Christopher Kruegel, Andrew Myers, and Shai Halevi. ACM, 2016, pp. 1006–1018. URL: `https://eprint.iacr.org/2016/659` (cit. on pp. 171, 189, 190).

[BCL+17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. "NTRU Prime." In: *Selected Areas in Cryptography – SAC 2017*. Ed. by Jan Camenisch and Carlisle Adams. Vol. 10719. LNCS. Springer, 2017, pp. 235–260. URL: `https://eprint.iacr.org/2016/461/20160513:121102` (cit. on pp. 171, 172, 174, 176, 180, 190).

[BCN+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: *Symposium on Security and Privacy – S&P 2015*. Ed. by Lujo Bauer and Vitaly Shmatikov. IEEE, 2015, pp. 553–570. URL: `https://eprint.iacr.org/2014/599` (cit. on pp. 171, 190).

[BDE+11] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. "On the Security of the Winternitz One-Time Signature Scheme." In: *Progress in Cryptology – AFRICACRYPT 2018*. Ed. by Abderrahmane Nitaj and David Pointcheval. Vol. 6737. LNCS. Springer, 2011, pp. 363–378. URL: `https://eprint.iacr.org/2011/191` (cit. on pp. 63, 102).

[BDE+17] 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, and Peter Schwabe. *SPHINCS*$^+$. Submission to the NIST Post-Quantum Cryptography Standardization project. 2017. URL: `https://sphincs.org` (cit. on pp. 17, 22, 43, 50, 72, 102, 215, 261).

[BDF+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. "Random Oracles in a Quantum World." In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. LNCS. Springer, 2011, pp. 41–69. URL: `https://eprint.iacr.org/2010/428` (cit. on p. 176).

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Post-Quantum Cryptography – PQCrypto 2011*. Ed. by Bo-Yin Yang. Vol. 7071. LNCS. Springer, 2011, pp. 117–129. URL: `https://eprint.iacr.org/2011/484` (cit. on pp. 44, 61, 63, 65, 102).

[BDK+07] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. "Merkle Signatures with Virtually Unlimited Signature Capacity." In: *Applied Cryptography and Network Security – ACNS 2017*. Ed. by Jonathan Katz and Moti Yung. Vol. 4521. LNCS. Springer, 2007, pp. 31–45 (cit. on pp. 91, 92).

[BDK+17] 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: *European Symposium on Security and Privacy – EuroS&P 2018*. IEEE, 2017, pp. 353–367. URL: `https://eprint.iacr.org/2017/634` (cit. on p. 190).

[BDL+11] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. LNCS. Springer, 2011, pp. 124–142. URL: `https://eprint.iacr.org/2011/368` (cit. on pp. 63, 64).

[BDP+11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The* KECCAK *reference*. 2011. URL: `http://keccak.noekeon.org` (cit. on pp. 33, 138, 159, 168).

[BDP+18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keccak Code Package*. `https://github.com/`

gvanas/KeccakCodePackage, Retrieved on May 7, 2018. 2018 (cit. on p. 138).

[BDS08] Johannes Buchmann, Erik Dahmen, and Michael Schneider. "Merkle tree traversal revisited." In: *Post-Quantum Cryptography – PQCrypto 2008*. Ed. by Johannes Buchmann and Jintai Ding. Vol. 5299. LNCS. Springer, 2008, pp. 63–78. URL: https://www.cdc.informatik.tu-darmstadt.de/reports/reports/AuthPath.pdf (cit. on pp. 61, 62, 67, 70).

[BDS09] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. "Hash-based digital signature schemes." In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Springer, 2009, pp. 35–93 (cit. on p. 21).

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. The State of the Art of Stream Ciphers – SASC 2008. 2008. URL: http://cr.yp.to/papers.html\#chacha (cit. on pp. 90, 93).

[Ber09] Daniel J. Bernstein. "Batch binary edwards." In: *Advances in Cryptology – CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. LNCS. Springer. 2009, pp. 317–336. URL: https://www.iacr.org/archive/crypto2009/56770315/56770315.pdf (cit. on p. 182).

[BFP09] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. "Hybrid approach for solving multivariate systems over finite fields." In: *Journal of Mathematical Cryptology* 3.3 (2009), pp. 177–197. URL: http://www-polsys.lip6.fr/~jcf/Papers/JMC2.pdf (cit. on p. 137).

[BFP12] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. "Solving polynomial systems over finite fields: improved analysis of the hybrid approach." In: *International Symposium on Symbolic and Algebraic Computation – ISSAC 2012*. Ed. by Joris van der Hoeven and Mark van Hoeij. ACM, 2012, pp. 67–74. URL: https://hal.inria.fr/hal-00776070 (cit. on pp. 137, 157).

[BFS+13] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. "On the complexity of solving quadratic Boolean systems." In: *Journal of Complexity* 29.1 (2013), pp. 53–75. URL: www-polsys.lip6.fr/~jcf/Papers/BFSS12.pdf (cit. on pp. 157, 168).

[BFS15] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. "On the complexity of the F5 Gröbner basis algorithm." In: *Journal of Symbolic Computation* 70 (2015), pp. 49–70. URL: `https://hal.inria.fr/hal-01064519` (cit. on pp. 137, 157).

[BGD+06] Johannes Buchmann, L. C. Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. "CMSS - An Improved Merkle Signature Scheme." In: *Progress in Cryptology – INDOCRYPT 2006*. Ed. by Rana Barua and Tanja Lange. Vol. 4329. LNCS. Springer, 2006, pp. 349–363. URL: `https://eprint.iacr.org/2006/320` (cit. on p. 60).

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. "On the hardness of learning with rounding over small modulus." In: *Theory of Cryptography*. Ed. by Eyal Kushilevitz and Tal Malkin. Vol. 9562. LNCS. Springer, 2016, pp. 209–224. URL: `https://eprint.iacr.org/2015/769` (cit. on p. 193).

[BGML+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. "Round5: Compact and Fast Post-Quantum Public-Key Encryption." In: *Post-Quantum Cryptography – PQCrypto 2019*. Ed. by Jintai Ding and Rainer Steinwandt. LNCS. Springer, 2018, pp. 83–102. URL: `https://eprint.iacr.org/2018/725` (cit. on p. 191).

[BH17] Leon Groot Bruinderink and Andreas Hülsing. ""Oops, I did it again" – Security of One-Time Signatures under Two-Message Attacks." In: *Selected Areas in Cryptography – SAC 2017*. Ed. by Carlisle Adams and Jan Camenisch. LNCS. Springer, 2017, pp. 299–322. URL: `https://eprint.iacr.org/2016/1042` (cit. on p. 44).

[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: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Marc Fischlin and Elisabeth Oswald. Vol. 9056. LNCS. Springer, 2015, pp. 368–397. URL: `https://eprint.iacr.`

`org/2014/795` (cit. on pp. 16, 44, 71, 85–87, 90, 93, 94, 99, 102, 104, 114, 131, 146, 163).

[BHK15] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. "Subtleties in the definition of IND-CCA: When and how should challenge decryption be disallowed?" In: *Journal of Cryptology* 28.1 (2015), pp. 29–48. URL: `https://eprint.iacr.org/2009/418` (cit. on p. 30).

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS$^+$ signature framework." In: *Conference on Computer and Communications Security – CCS '19*. Ed. by XiaoFeng Wang and Jonathan Katz. *To appear*. ACM, 2019. URL: `https://eprint.iacr.org/2019/1086` (cit. on pp. 17, 43, 92, 102–104, 106, 111, 146, 215, 259).

[BL] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. URL: `http://bench.cr.yp.to` (cit. on pp. 40, 189, 208).

[BL06] Johannes Buchmann and Christoph Ludwig. "Practical Lattice Basis Sampling Reduction." In: *Algorithmic Number Theory – ANTS-VII*. Ed. by Florian Hess, Sebastian Pauli, and Michael Pohst. LNCS. Springer, 2006, pp. 222–237. URL: `https://eprint.iacr.org/2005/072` (cit. on p. 171).

[BM99] Mihir Bellare and Sara K. Miner. "A forward-secure digital signature scheme." In: *Advances in Cryptology – CRYPTO '99*. Ed. by Michael Wiener. Vol. 1666. LNCS. Springer, 1999, pp. 431–448. URL: `https://cseweb.ucsd.edu/~mihir/papers/fsig.pdf` (cit. on p. 52).

[BP18] Daniel J. Bernstein and Edoardo Persichetti. *Towards KEM Unification*. IACR Cryptology ePrint Archive, Report 2018/526. 2018. URL: `https://eprint.iacr.org/2018/526` (cit. on p. 179).

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. "Pseudorandom functions and lattices." In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. LNCS. Springer, 2012, pp. 719–737. URL: `https://eprint.iacr.org/2011/401` (cit. on pp. 193, 194).

[Bra16] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography.* Posting on the Google Security Blog. 2016. URL: `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html` (cit. on p. 171).

[BSI] *Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token.* Tech. rep. TR-03110. Version 2.20. German Federal Office for Information Security (BSI), 2015. URL: `https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html` (cit. on pp. 73, 80).

[BY18] Daniel J. Bernstein and Bo-Yin Yang. "Asymptotically faster quantum algorithms to solve multivariate quadratic equations." In: *Post-Quantum Cryptography – PQCrypto 2018.* Ed. by Tanja Lange and Rainer Steinwandt. Vol. 10786. LNCS. Springer. 2018, pp. 487–506. URL: `https://eprint.iacr.org/2017/1206` (cit. on pp. 137, 158).

[BY19] Daniel J Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.3 (2019), pp. 340–398. URL: `https://eprint.iacr.org/2019/266` (cit. on p. 186).

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. "SSE implementation of multivariate PKCs on modern x86 CPUs." In: *Cryptographic Hardware and Embedded Systems – CHES 2009.* Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. LNCS. Springer, 2009, pp. 33–48. URL: `https://www.iacr.org/archive/ches2009/57470031/57470031.pdf` (cit. on pp. 138, 140, 142).

[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." In: *Conference on Computer and Communications Security – CCS '17.* Ed. by David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 1825–1842. URL: `https://eprint.iacr.org/2017/279` (cit. on pp. 146, 154, 163).

[CDG+19] Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. *The Picnic Signature Scheme*. Submission to round 2 of the NIST Post-Quantum Cryptography Standardization project. 2019. URL: `https://github.com/microsoft/Picnic/blob/master/spec/design-v2.0.pdf` (cit. on pp. 102, 111).

[CDH+19] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. *NTRU: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2019. URL: `https://ntru.org` (cit. on pp. 19, 171, 191, 261).

[CDM+05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. "Merkle-Damgård revisited: How to construct a hash function." In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer. 2005, pp. 430–448. URL: `https://iacr.org/archive/crypto2005/36210424/36210424.pdf` (cit. on p. 27).

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. "Proofs of partial knowledge and simplified design of witness hiding protocols." In: *Advances in Cryptology – CRYPTO '95*. Ed. by Yvo G. Desmedt. Vol. 839. LNCS. Springer. 1994, pp. 174–187 (cit. on p. 117).

[CGP] Nicolas Courtois, Louis Goubin, and Jacques Patarin. *SFLASH, a fast asymmetric signature scheme for low-cost smartcards - Primitive specification and supporting documentation*. URL: `http://www.minrank.org/sflash-b-v2.pdf` (cit. on p. 113).

[CHK+17] Jung Hee Cheon, Kyoohyung Han, Jinsu Kim, Changmin Lee, and Yongha Son. "A Practical Post-Quantum Public-Key Cryptosystem Based on spLWE." In: *Information Security and Cryptology – ICISC 2016*. Ed. by Seokhie Hong and Jong Hwan Park. Vol. 10157. LNCS. Springer, 2017, pp. 51–74. URL: `https://eprint.iacr.org/2016/1055` (cit. on pp. 171, 189, 190).

[CHR+16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "From 5-Pass $\mathcal{MQ}$-Based Identification to $\mathcal{MQ}$-Based Signatures." In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. LNCS. Springer, 2016, pp. 135–165. URL: `https://eprint.iacr.org/2016/708` (cit. on pp. 18, 22, 113, 114, 124, 130, 132, 135, 137, 138, 142, 143, 146, 154, 156, 163, 260).

[CHR+17] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. *MQDSS*. Submission to NIST's Post-Quantum Cryptography Standardization project. 2017. URL: `http://mqdss.org` (cit. on pp. 18, 113, 261).

[CHR+18] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "SOFIA: MQ-based signatures in the QROM." In: *Public Key Cryptography – PKC 2018*. Ed. by Michel Abdalla and Ricardo Dahab. Vol. 10770. LNCS. Springer, 2018, pp. 3–33. URL: `https://eprint.iacr.org/2017/680` (cit. on pp. 18, 22, 113, 114, 148, 155, 156, 158, 260).

[CKL+18] 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." In: *Security and Cryptography for Networks – SCN 2018*. Ed. by Dario Catalano and Roberto De Prisco. Vol. 11035. LNCS. Springer, 2018, pp. 160–177. URL: `https://eprint.iacr.org/2016/1126` (cit. on pp. 171, 190).

[CKP+00] Nicolas Courtois, Er Klimov, Jacques Patarin, and Adi Shamir. "Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations." In: *Advances in Cryptology – EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, 2000, pp. 392–407. URL: `www.iacr.org/archive/eurocrypt2000/1807/18070398-new.pdf` (cit. on pp. 137, 157, 168).

[CMF19] Michael Curcio, David McGrew, and Scott Fluhrer. *Leighton-Micali Hash-Based Signatures*. Request for Comments 8554. IETF, 2019. URL: `https://tools.ietf.org/html/rfc8554` (cit. on pp. 33, 108).

[Com90] Paul G. Comba. "Exponentiation cryptosystems on the IBM PC." In: *IBM systems journal* 29.4 (1990), pp. 526–538 (cit. on p. 181).

[Con03] Consortium for Efficient Embedded Security. *EESS #1: Implementation Aspects of NTRUEncrypt and NTRUSign v. 2.0.* 2003. URL: `http://grouper.ieee.org/groups/1363/lattPK/submissions/EESS1v2.pdf` (cit. on pp. 171, 174).

[Coo66] Stephen Cook. "On the Minimum Computation Time of Functions." PhD thesis. Harvard University, 1966 (cit. on pp. 182, 199).

[Cop06] Jack B. Copeland, ed. *Colossus: The Secrets of Bletchley Park's Codebreaking Computers.* Oxford University Press, 2006. ISBN: 978-0-19-284055-4 (cit. on p. 13).

[Cou01] Nicolas T. Courtois. "Efficient zero-knowledge authentication based on a linear algebra problem MinRank." In: *Advances in Cryptology – ASIACRYPT 2001.* Ed. by Colin Boyd. Vol. 2248. LNCS. Springer, 2001, pp. 402–421. URL: `https://eprint.iacr.org/2001/058` (cit. on pp. 113, 120).

[CPL+17] Jung Hee Cheon, Sangjoon Park, Joohee Lee, Duhyeong Kim, Yongsoo Song, Seungwan Hong, Dongwoo Kim, Jinsu Kim, Seong-Min Hong, Aaram Yun, Jeongsu Kim, Haeryong Park, Eunyoung Choi, Kimoon kim, Jun-Sub Kim, and Jieun Lee. *Lizard: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`. 2017 (cit. on pp. 191, 193).

[CVE10] Pierre-Louis Cayrel, Pascal Véron, and Sidi Mohamed El Yousfi Alaoui. "A zero-knowledge identification scheme based on the q-ary syndrome decoding problem." In: *Selected Areas in Cryptography – SAC 2010.* Ed. by Alex Biryukov, Guang Gong, and Douglas R. Stinson. Vol. 6544. LNCS. Springer, 2010, pp. 171–186. URL: `https://hal.inria.fr/hal-00674249` (cit. on p. 124).

[Dam90] Ivan B. Damgård. "A design principle for hash functions." In: *Advances in Cryptology – CRYPTO '89.* Ed. by Gilles Brassard. Vol. 435. LNCS. Springer, 1990, pp. 416–427 (cit. on pp. 68, 109).

[Den03] Alexander W. Dent. "A Designer's Guide to KEMs." In: *Cryptography and Coding*. Ed. by Kenneth G. Paterson. Vol. 2898. LNCS. Springer, 2003, pp. 133–151. URL: `http://www.cogentcryptography.com/papers/designer.pdf` (cit. on pp. 173, 176).

[DFG13] Özgür Dagdelen, Marc Fischlin, and Tommaso Gagliardoni. "The Fiat-Shamir Transformation in a Quantum World." In: *Advances in Cryptology - ASIACRYPT 2013.* Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. LNCS. Springer, 2013, pp. 62–81. URL: `https://eprint.iacr.org/2013/245` (cit. on pp. 136, 144).

[DFM+19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. *Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model.* Cryptology ePrint Archive, Report 2019/190. 2019. URL: `https://eprint.iacr.org/2019/190` (cit. on pp. 117, 136, 144, 216).

[DFS+07] Vivien Dubois, Pierre-Alain Fouque, Adi Shamir, and Jacques Stern. "Practical cryptanalysis of SFLASH." In: *Advances in Cryptology – CRYPTO 2007.* Ed. by Alfred Menezes. Vol. 4622. LNCS. Springer, 2007, pp. 1–12. URL: `https://eprint.iacr.org/2007/141` (cit. on p. 113).

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography." In: *Transactions on Information Theory* 22.6 (1976), pp. 644–654. URL: `http://www-ee.stanford.edu/\%7Ehellman/publications/24.pdf` (cit. on p. 13).

[DHP+] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *eXtended Keccak Code Package.* URL: `https://github.com/XKCP/XKCP` (cit. on p. 208).

[DHY+06] Jintai Ding, Lei Hu, Bo-Yin Yang, and Jiun-Ming Chen. *Note on Design Criteria for Rainbow-Type Multivariates.* IACR Cryptology ePrint Archive, Report 2006/307. 2006. URL: `https://eprint.iacr.org/2006/307` (cit. on pp. 113, 120).

[Die04] Claus Diem. "The XL-Algorithm and a Conjecture from Commutative Algebra." In: *Advances in Cryptology – ASIACRYPT 2004.* Ed. by Pil Joong Lee. Vol. 3329. LNCS. Springer, 2004, pp. 323–337. URL: `https://www.iacr.org/archive/asiacrypt2004/33290320/33290320.pdf` (cit. on pp. 137, 157).

[DKR+17] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. *SABER: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`. 2017 (cit. on pp. 191, 194).

[DKR+18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM." In: *Progress in Cryptology – AFRICACRYPT 2018.* Ed. by Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi. Vol. 10831. LNCS. Springer, 2018, pp. 282–305. URL: `https://eprint.iacr.org/2018/230` (cit. on p. 194).

[DOT+08] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. "Digital Signatures Out of Second-Preimage Resistant Hash Functions." In: *Post-Quantum Cryptography – PQCrypto 2008.* Ed. by Johannes Buchmann and Jintai Ding. Vol. 5299. LNCS. Springer, 2008, pp. 109–123 (cit. on pp. 63, 102).

[DR99] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael." In: (1999). URL: `https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf` (cit. on p. 33).

[DS05] Jintai Ding and Dieter Schmidt. "Rainbow, a New Multivariable Polynomial Signature Scheme." In: *Applied Cryptography and Network Security.* Ed. by John Ioannidis, Angelos D. Keromytis, and Moti Yung. Vol. 3531. LNCS. Springer, 2005, pp. 164–175 (cit. on p. 114).

[EDV+12] Sidi Mohamed El Yousfi Alaoui, Özgür Dagdelen, Pascal Véron, David Galindo, and Pierre-Louis Cayrel. "Extended Security Arguments for Signature Schemes." In: *Progress in Cryptology – AFRICACRYPT 2012.* Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. LNCS. Springer, 2012, pp. 19–34. URL: `https://eprint.iacr.org/2012/068` (cit. on pp. 124–127).

[Eur17] Eurosmart. *Digital Security Industry To Pass The 10 Billion Mark In 2018 For Worldwide Shipments Of Secure Elements*. Press Release. 2017. URL: `http://www.eurosmart.com/news-publications/press-release/296` (cit. on p. 72).

[EVMY14] Thomas Eisenbarth, Ingo Von Maurich, and Xin Ye. "Faster Hash-based Signatures with Bounded Leakage." In: *Selected Areas in Cryptography – SAC 2013*. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisoněk. Vol. 8282. LNCS. Springer, 2014, pp. 223–243. URL: `http://users.wpi.edu/~teisenbarth/pdf/SignatureswithBoundedLeakageSAC.pdf` (cit. on pp. 91, 92).

[Fau02] Jean-Charles Faugère. "A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)." In: *International Symposium on Symbolic and Algebraic Computation – ISSAC 2002*. ACM, 2002, pp. 75–83. URL: `http://www-polsys.lip6.fr/~jcf/Papers/F02a.pdf` (cit. on pp. 137, 157, 168).

[Fau99] Jean-Charles Faugère. "A new efficient algorithm for computing Gröbner bases (F4)." In: *Journal of Pure and Applied Algebra* 139 (1999), pp. 61–88. URL: `http://www-polsys.lip6.fr/~jcf/Papers/F99a.pdf` (cit. on pp. 137, 157, 168).

[FGP+15] Jean-Charles Faugère, Danilo Gligoroski, Ludovic Perret, Simona Samardjiska, and Enrico Thomae. "A Polynomial-Time Key-Recovery Attack on MQQ Cryptosystems." In: *Public-Key Cryptography – PKC 2015*. Ed. by Jonathan Katz. Vol. 9020. LNCS. Springer, 2015, pp. 150–174. URL: `https://eprint.iacr.org/2014/811` (cit. on p. 113).

[Fis05] Marc Fischlin. "Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors." In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer, 2005, pp. 152–168. URL: `https://www.iacr.org/archive/crypto2005/36210148/36210148.pdf` (cit. on p. 147).

[FLP08] Jean-Charles Faugère, Françoise Levy-dit-Vehel, and Ludovic Perret. "Cryptanalysis of MinRank." In: *Advances in Cryptology – CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, 2008, pp. 280–

296. URL: `https://iacr.org/archive/crypto2008/51570280/51570280.pdf` (cit. on pp. 113, 120).

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes." In: *Advances in Cryptology – CRYPTO '99*. Ed. by Michael Wiener. Vol. 1666. LNCS. Springer, 1999, pp. 537–554 (cit. on pp. 173, 176).

[For18] Java Card Forum. *About the JCF*. accessed 2018-03-1. 2018. URL: `https://javacardforum.com` (cit. on p. 73).

[FS86] Amos Fiat and Adi Shamir. "How to prove yourself: Practical solutions to identification and signature problems." In: *Advances in Cryptology – CRYPTO '86*. Vol. 263. LNCS. Springer. 1986, pp. 186–194 (cit. on p. 117).

[Für09] Martin Fürer. "Faster integer multiplication." In: *Journal on Computing* 39.3 (2009), pp. 979–1005 (cit. on p. 182).

[Gar05] LC Coronado Garcıa. *On the security and the efficiency of the Merkle signature scheme*. IACR Cryptology ePrint Archive, Report 2005/192. 2005. URL: `https://eprint.iacr.org/2005/192` (cit. on p. 63).

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979 (cit. on pp. 113, 119).

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical lattice-based cryptography: a signature scheme for embedded systems." In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. LNCS. Springer, 2012, pp. 530–547. URL: `https://www.iacr.org/archive/ches2012/74280529/74280529.pdf` (cit. on p. 92).

[GM84] Shafi Goldwasser and Silvio Micali. "Probabilistic encryption." In: *Journal of computer and system sciences* 28.2 (1984), pp. 270–299 (cit. on p. 30).

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. “A digital signature scheme secure against adaptive chosen-message attacks.” In: *Journal on Computing* 17.2 (1988), pp. 281–308. URL: `https://people.csail.mit.edu/silvio/Selected\%20Scientific\%20Papers/Digital\%20Signatures/A_Digital_Signature_Scheme_Secure_Against_Adaptive_Chosen-Message_Attack.pdf` (cit. on p. 28).

[GMZB+17] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, and Jose-Luis Torre-Arce. *Round2: Algorithm Specification and Supporting Documentation.* Submission to NIST’s Post-Quantum Cryptography Standardization project. 2017. URL: `https://www.onboardsecurity.com/nist-post-quantum-crypto-submission` (cit. on p. 191).

[GØJ+11] Danilo Gligoroski, Rune S. Ødegård, Rune E. Jensen, Ludovic Perret, Jean-Charles Faugère, Svein Johan Knapskog, and Smile Markovski. “MQQ-SIG - An Ultra-Fast and Provably CMA Resistant Digital Signature Scheme.” In: *Trusted Systems – INTRUST 2011.* Ed. by Liqun Chen, Moti Yung, and Liehuang Zhu. Vol. 7222. LNCS. Springer, 2011, pp. 184–203. URL: `https://hal.inria.fr/hal-00778083` (cit. on p. 113).

[Gol04] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge, UK: Cambridge University Press, 2004 (cit. on pp. 60, 85).

[Gol87] Oded Goldreich. “Two remarks concerning the Goldwasser-Micali-Rivest signature scheme.” In: *Advances in Cryptology – CRYPTO ‘86.* Ed. by Andrew M. Odlyzko. Vol. 263. LNCS. Springer, 1987, pp. 104–110. URL: `http://theory.csail.mit.edu/ftp-data/pub/people/oded/gmr.ps` (cit. on pp. 60, 85).

[GPW+04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. “Comparing elliptic curve cryptography and RSA on 8-bit CPUs.” In: *Cryptographic Hardware and Embedded Systems – CHES 2004.* Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. LNCS. Springer. 2004, pp. 119–132. URL: `www.iacr.org/archive/ches2004/31560117/31560117.pdf` (cit. on p. 181).

[Gro15] Wouter de Groot. “A Performance Study of X25519 on Cortex-M3 and M4.” MA thesis. Technische Universiteit Eindhoven, 2015. URL: `https://research.tue.nl/en/studentTheses/a-performance-study-of-x25519-on-cortex-m3-and-m4` (cit. on p. 38).

[Gro96] Lov K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search.” In: *Symposium on Theory of Computing – STOC ‘96*. ACM, 1996, pp. 212–219. URL: `https://arxiv.org/pdf/quant-ph/9605043v3.pdf` (cit. on pp. 24, 137, 157).

[HBG+15] Andreas Hülsing, D. Butin, S. Gazdag, and A. Mohaisen. *XMSS: Extended Hash-Based Signatures*. Crypto Forum Research Group Internet-Draft. 2015. URL: `https://tools.ietf.org/html/draft-irtf-cfrg-xmss-hash-based-signatures-01` (cit. on p. 99).

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. Request for Comments 8391. IETF, 2018. URL: `https://tools.ietf.org/html/rfc8391` (cit. on pp. 17, 33, 44, 50, 52, 65–67, 70, 72, 75, 78, 82, 261).

[HG07] Nick Howgrave-Graham. “A hybrid lattice-reduction and meet-in-the-middle attack against NTRU.” In: *Advances in Cryptology – CRYPTO 2007*. Ed. by Alfred Menezes. Vol. 4622. LNCS. Springer, 2007, pp. 150–169. URL: `http://www.iacr.org/archive/crypto2007/46220150/46220150.pdf` (cit. on pp. 171, 174).

[HGSS+03] Nick Howgrave-Graham, Joseph H. Silverman, Ari Singer, and William Whyte. *NAEP: Provable Security in the Presence of Decryption Failures*. IACR Cryptology ePrint Archive, Report 2003/172. 2003. URL: `https://eprint.iacr.org/2003/172` (cit. on pp. 171, 173).

[HGSW05] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. “Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3.” In: *Topics in Cryptology – CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. LNCS. Springer, 2005, pp. 118–135. URL: `https://eprint.iacr.org/2005/045` (cit. on pp. 171, 174).

[HHHG+09] Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. “Choosing NTRUEncrypt Parameters in Light of Combined Lattice Reduction and MITM Approaches.” In: *Applied Cryptography and Network Security – ACNS 2009*. Ed. by Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud. Vol. 5536. LNCS. Springer, 2009, pp. 437–455. URL: `https://eprint.iacr.org/2005/045` (cit. on pp. 171, 174).

[HHL16] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. “Even faster integer multiplication.” In: *Journal of Complexity* 36 (2016), pp. 1–30 (cit. on p. 182).

[HK06] Shai Halevi and Hugo Krawczyk. “Strengthening digital signatures via randomized hashing.” In: *Advances in Cryptology – CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Springer, 2006, pp. 41–59. URL: `https://www.iacr.org/archive/crypto2006/41170039/41170039.pdf` (cit. on p. 130).

[HPS00] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *Public key cryptosystem method and apparatus*. United States Patent 6081597. Application filed August 19, 1997, `http://www.freepatentsonline.com/6081597.html`. 2000 (cit. on p. 172).

[HPS+17] Jeff Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. “Choosing Parameters for NTRUEncrypt.” In: *Topics in Cryptology – CTA-RSA 2017*. Ed. by Helena Handschuh. Vol. 10159. LNCS. Springer, 2017, pp. 3–18. URL: `https://eprint.iacr.org/2015/708` (cit. on pp. 174, 190, 194).

[HPS96] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *NTRU: A New High Speed Public Key Cryptosystem*. Preliminary draft CRYPTO ‘96 rump session. `http://web.securityinnovation.com/hubfs/files/ntru-orig.pdf`. 1996 (cit. on pp. 171, 174, 179).

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. “NTRU: A ring-based public key cryptosystem.” In: *Algorithmic Number Theory – ANTS-III*. Ed. by Joe P. Buhler. Vol. 1423. LNCS. Springer, 1998, pp. 267–288 (cit. on pp. 171–174, 179, 194).

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. “Optimal Parameters for XMSS^{MT}.” In: *Security Engineering and Intelligence Informatics*. Ed. by Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu. Vol. 8128. LNCS. Springer, 2013, pp. 194–208. URL: `https://eprint.iacr.org/2017/966` (cit. on pp. 63, 65, 72).

[HRS16a] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. “ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM.” In: *Public Key Cryptography – PKC 2016*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9614. LNCS. Springer, 2016, pp. 446–470. URL: `https://eprint.iacr.org/2015/1042` (cit. on pp. 16, 21, 22, 43, 92, 260).

[HRS16b] Andreas Hülsing, Joost Rijneveld, and Fang Song. “Mitigating Multi-Target Attacks in Hash-based Signatures.” In: *Public Key Cryptography – PKC 2016*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9614. LNCS. Springer, 2016, pp. 387–416. URL: `https://eprint.iacr.org/2015/1256` (cit. on pp. 16, 21, 43, 63, 65, 67, 70, 72, 90, 102, 108, 260).

[HRS+17a] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. “High-speed key encapsulation from NTRU.” In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, 2017, pp. 232–252. URL: `https://eprint.iacr.org/2017/667` (cit. on pp. 19, 22, 171–176, 180, 202, 260).

[HRS+17b] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. *NTRU-KEM-HRSS: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2017. URL: `https://ntru-hrss.org` (cit. on pp. 19, 171, 180, 191, 261).

[HS06] Jeffrey Hoffstein and Joseph H. Silverman. *Speed enhanced cryptographic method and apparatus*. United States Patent 7031468. Application filed August 24, 2001, `http://www.freepatentsonline.com/7031468.html`. 2006 (cit. on p. 172).

[Hül13a] Andreas Hülsing. "Practical Forward Secure Signatures using Minimal Security Assumptions." PhD thesis. TU Darmstadt, 2013. URL: `http://tuprints.ulb.tu-darmstadt.de/3651` (cit. on pp. 52, 67, 69).

[Hül13b] Andreas Hülsing. "W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes." In: *Progress in Cryptology – AFRICACRYPT 2013*. Ed. by Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien. Vol. 7918. LNCS. Springer, 2013, pp. 173–188. URL: `https://eprint.iacr.org/2017/965` (cit. on pp. 63, 102).

[HW11] Michael Hutter and Erich Wenger. "Fast multi-precision multiplication for public-key cryptography on embedded microprocessors." In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. LNCS. Springer. 2011, pp. 459–474. URL: `https://www.iacr.org/archive/ches2011/69170459/69170459.pdf` (cit. on p. 181).

[ICAO14] *Supplemental Access Control for Machine Readable Travel Documents*. Tech. rep. Version 1.1. International Civil Aviation Organization (ICAO), 2014 (cit. on pp. 73, 80).

[IEE09] IEEE. *IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices*. Std 1363.1-2008. 2009 (cit. on p. 171).

[IT88] Toshiya Itoh and Shigeo Tsujii. "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases." In: *Information and computation* 78.3 (1988), pp. 171–177. URL: `https://sciencedirect.com/science/article/pii/0890540188900247` (cit. on p. 184).

[JV17] Antoine Joux and Vanessa Vitse. "A crossbred algorithm for solving Boolean polynomial systems." In: *Number-Theoretic Methods in Cryptology – NuTMiC 2017*. Ed. by Jerzy Kaczorowski, Josef Pieprzyk, and Jacek Pomykała. Vol. 10737. LNCS. Springer, 2017, pp. 3–21. URL: `https://eprint.iacr.org/2017/372` (cit. on p. 157).

[Kah96] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. ISBN: 978-0-68-483130-5 (cit. on p. 13).

[KGB+18] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. "Differential Power Analysis of XMSS and SPHINCS." In: *Constructive Side-Channel Analysis and Secure Design – COSADE 2018*. Ed. by Junfeng Fan Benedikt Gierlichs. Vol. 10815. LNCS. Springer, 2018, pp. 168–188. URL: `https://eprint.iacr.org/2018/673` (cit. on p. 85).

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014 (cit. on p. 30).

[KLM+17] Stefan Kölbl, Martin Lauridsen, Florian Mendel, and Christian Rechberger. "Haraka v2 – Efficient Short-Input Hashing for Post-Quantum Applications." In: *IACR Transactions on Symmetric Cryptology* 2016.2 (2017), pp. 1–29. URL: `https://eprint.iacr.org/2016/098` (cit. on p. 106).

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. "A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model." In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, 2018, pp. 552–586. URL: `https://eprint.iacr.org/2017/916` (cit. on pp. 136, 144, 216).

[KMR+18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. "Saber on ARM." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 243–266. URL: `https://eprint.iacr.org/2018/682` (cit. on pp. 191, 200, 202, 208, 209, 211).

[KO63] Anatolii Karatsuba and Yuri Ofman. "Multiplication of multidigit numbers on automata." In: *Soviet Physics Doklady* 7 (1963). Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. Scanned version on `http://cr.yp.to/bib/1963/karatsuba.html`, pp. 595–596 (cit. on pp. 181, 199).

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. "Unbalanced Oil and Vinegar Signature Schemes." In: *Advances in Cryptology – EUROCRYPT '99*. Ed. by Jacques Stern. Vol. 1592. LNCS. Springer, 1999, pp. 206–222. URL: `http://www.goubin.fr/papers/OILLONG.PDF` (cit. on p. 114).

[KRS+18] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *PQM4: Post-quantum crypto library for the ARM Cortex-M4.* 2018. URL: `https://github.com/mupq/pqm4` (cit. on pp. 35, 41, 92, 203, 208, 210, 211).

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. "Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates." In: *Applied Cryptography and Network Security – ACNS 2019.* Ed. by Robert H. Deng, Valérie Gauthier, Martín Ochoa, and Moti Yung. Vol. 11464. LNCS. Springer, 2019, pp. 281–301. URL: `https://eprint.iacr.org/2018/1018` (cit. on pp. 20, 22, 35, 171, 173, 260).

[KRS+19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4.* Second NIST PQC Standardization Conference. 2019 (cit. on pp. 22, 41, 92, 260).

[KS98a] Burt Kaliski and Jessica Staddon. *PKCS #1: RSA Cryptography Specifications.* Request for Comments 2437. IETF, 1998. URL: `https://tools.ietf.org/html/rfc2437` (cit. on p. 109).

[KS98b] Aviad Kipnis and Adi Shamir. "Cryptanalysis of the Oil & Vinegar Signature Scheme." In: *Advances in Cryptology – CRYPTO '98.* Ed. by Hugo Krawczyk. 1998. URL: `http://antoanthongtin.vn/Portals/0/UploadImages/kiennt2/KyYe/DuLieuNuocNgoai/3.Advances\%20in\%20cryptology-Crypto\%201998-LNCS\%201462/14620257.PDF` (cit. on p. 113).

[KZ19] Daniel Kales and Greg Zaverucha. "Forgery Attacks on MQDSSv2.0." In: (2019). URL: `https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/LlHhfwg73eQ/omM6TWwlEwAJ` (cit. on p. 216).

[Lam79] Leslie Lamport. *Constructing digital signatures from a one way function.* Technical Report SRI-CSL-98. SRI International Computer Science Laboratory, 1979 (cit. on pp. 43, 45).

[Lan18] Adam Langley. *CECPQ2.* 2018. URL: `https://www.imperialviolet.org/2018/12/12/cecpq2.html` (cit. on p. 171).

[Lei18] Dominik Leichtle. “Post-Quantum Signatures from Identification Schemes.” MA thesis. Universität Stuttgart, 2018. URL: `https://research.tue.nl/en/studentTheses/post-quantum-signatures-from-identification-schemes` (cit. on pp. 143, 155, 159).

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On Ideal Lattices and Learning with Errors over Rings.” In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. LNCS. Springer, 2010, pp. 1–23. URL: `https://eprint.iacr.org/2012/230` (cit. on p. 172).

[LPR+18] Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. “Is Java Card ready for hash-based signatures?” In: *Advances in Information and Computer Security – IWSEC 2018*. Ed. by Atsuo Inomata and Kan Yasuda. Vol. 11049. LNCS. Springer, 2018, pp. 127–142. URL: `https://eprint.iacr.org/2018/611` (cit. on pp. 16, 21, 43, 260).

[LS15] Adeline Langlois and Damien Stehlé. “Worst-case to average-case reductions for module lattices.” In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599. URL: `https://eprint.iacr.org/2012/090` (cit. on p. 195).

[LS19] Vadim Lyubashevsky and Gregor Seiler. *NTTRU: Truly Fast NTRU Using NTT*. IACR Cryptology ePrint Archive, Report 2019/040. 2019. URL: `https://eprint.iacr.org/2019/040` (cit. on pp. 180, 216).

[Lud03] Christoph Ludwig. “A Faster Lattice Reduction Method Using Quantum Search.” In: *Algorithms and Computation – ISAAC 2003*. Ed. by Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono. Vol. 2906. LNCS. Springer, 2003, pp. 199–208. URL: `https://www.cdc.informatik.tu-darmstadt.de/reports/TR/TI-03-03.QSamplingPaper.pdf` (cit. on p. 171).

[LZ19] Qipeng Liu and Mark Zhandry. *Revisiting Post-Quantum Fiat-Shamir*. IACR Cryptology ePrint Archive, Report 2019/262. 2019. URL: `https://eprint.iacr.org/2019/262` (cit. on pp. 136, 144, 216).

[Mal63] Colin L. Mallows. “Patience sorting.” In: *SIAM review* 5.4 (1963), p. 375 (cit. on p. 185).

[May99] Alexander May. *Cryptanalysis of NTRU.* 1999. URL: `https://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/paper/cryptanalysisofntru.ps` (cit. on p. 171).

[Mer79] Ralph Charles Merkle. "Secrecy, authentication, and public key systems." PhD thesis. Stanford University, 1979 (cit. on pp. 68, 109).

[Mer90] Ralph Merkle. "A Certified Digital Signature." In: *Advances in Cryptology – CRYPTO '89.* Ed. by Gilles Brassard. Vol. 435. LNCS. Springer, 1990, pp. 218–238. URL: `www.merkle.com/papers/Certified1979.pdf` (cit. on pp. 13, 45, 47, 48, 52–54, 57).

[MM+85] Stephen Matyas, Carl Meyer, and Jonathan Oseas. "Generating strong one-way functions with cryptographic algorithm." In: *IBM Technical Disclosure Bulletin* 27 (1985), pp. 5658–5659 (cit. on p. 78).

[Moo18] Dustin Moody. *Let's Get Ready to Rumble – The NIST PQC "Competition".* Presentation at PQCrypto 2018. 2018. URL: `https://csrc.nist.gov/Presentations/2018/Let-s-Get-Ready-to-Rumble-The-NIST-PQC-Competiti` (cit. on p. 33).

[MRH04] Ueli Maurer, Renato Renner, and Clemens Holenstein. "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology." In: *Theory of Cryptography – TCC 2004.* Vol. 2951. LNCS. Springer. 2004, pp. 21–39. URL: `https://eprint.iacr.org/2003/161` (cit. on p. 27).

[NIST01] *FIPS PUB 197: Advanced Encryption Standard (AES).* National Institute of Standards and Technology, 2001. URL: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf` (cit. on p. 33).

[NIST15a] *FIPS PUB 180-4: Secure Hash Standard.* National Institute of Standards and Technology, 2015. URL: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf` (cit. on pp. 33, 106, 208).

[NIST15b] *FIPS PUB 202 – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* National Institute of Standards and Technology, 2015. URL: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf` (cit. on pp. 26, 33, 106, 208).

[NIST16] *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process.* National Institute of Standards and Technology, 2016. URL: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf` (cit. on p. 33).

[NIST19] *Round 2 Submissions.* National Institute of Standards and Technology, 2019. URL: `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions` (cit. on p. 33).

[NSA15] *Commercial National Security Algorithm Suite.* National Security Agency, 2015. URL: `https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm` (cit. on p. 33).

[NSW05] Dalit Naor, Amir Shenhav, and Avishai Wool. *One-time signatures revisited: Have they become practical?* IACR Cryptology ePrint Archive. 2005. URL: `https://eprint.iacr.org/2005/442` (cit. on p. 62).

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. “Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices.” In: *Design Automation Conference – DAC 2014.* ACM, 2014, pp. 1–6. URL: `https://www.sha.rub.de/media/attachments/files/2014/06/bliss_arm.pdf` (cit. on p. 92).

[Pat96] Jacques Patarin. “Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): Two new families of asymmetric algorithms.” In: *Advances in Cryptology – EUROCRYPT ‘96.* Ed. by Ueli Maurer. Vol. 1070. LNCS. Springer, 1996, pp. 33–48. URL: `http://www.minrank.org/hfe.pdf` (cit. on pp. 113, 120).

[Pat97] Jacques Patarin. “The Oil and Vinegar signature scheme.” In: *Dagstuhl Workshop on Cryptography.* 1997 (cit. on p. 113).

[PCG01] Jacques Patarin, Nicolas Courtois, and Louis Goubin. “QUARTZ, 128-Bit Long Digital Signatures.” In: *Topics in Cryptology – CT-RSA 2001.* Ed. by David Naccache. Vol. 2020. LNCS. Springer, 2001, pp. 282–297. URL: `http://www.goubin.fr/papers/rsa2001b.pdf` (cit. on p. 113).

[PCY+15] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. “Design Principles for HFEv- Based Multivariate Signature Schemes.” In: *Advances in Cryptology – ASIACRYPT 2015.* Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452. LNCS. Springer, 2015, pp. 311–334. URL: `https://www.iacr.org/archive/asiacrypt2015/94520213/94520213.pdf` (cit. on p. 114).

[PKCS11] Susan Gleeson and Chris Zimman, eds. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40.* OASIS Standard. 2015. URL: `http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/errata01/os/pkcs11-base-v2.40-errata01-os-complete.html` (cit. on p. 74).

[PLP16] Rafael del Pino, Vadim Lyubashevsky, and David Pointcheval. “The Whole is Less Than the Sum of Its Parts: Constructing More Efficient Lattice-Based AKEs.” In: *Security and Cryptography for Networks – SCN 2016.* Ed. by Vassilis Zikas and Roberto De Prisco. Vol. 9841. LNCS. Springer, 2016, pp. 273–291. URL: `https://eprint.iacr.org/2016/435` (cit. on pp. 171, 174).

[PO95] Bart Preneel and Paul C. van Oorschot. “MDx-MAC and Building Fast MACs from Hash Functions.” In: *Advances in Cryptology – CRYPTO ‘95.* Ed. by Don Coppersmith. Vol. 963. LNCS. Springer, 1995, pp. 1–14. URL: `https://people.scs.carleton.ca/~paulv/papers/Crypto95.pdf` (cit. on p. 110).

[PP03] David Pointcheval and Guillaume Poupard. “A New NP-Complete Problem and Public-Key Identification.” In: *Designs, Codes and Cryptography* 28.1 (2003), pp. 5–31 (cit. on p. 124).

[PS96] David Pointcheval and Jacques Stern. “Security Proofs for Signature Schemes.” In: *Advances in Cryptology – EUROCRYPT ‘96.* Ed. by Ueli Maurer. Vol. 1070. LNCS. Springer, 1996, pp. 387–398. URL: `https://www.di.ens.fr/~pointche/Documents/Papers/1996_eurocrypt.pdf` (cit. on pp. 117, 130, 144, 147).

[RED+08] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. “Fast hash-based signatures on constrained devices.” In: *Smart Card Research and Advanced Applications – CARDIS 2008.* Ed. by Gilles Grimaud and François-Xavier Standaert. Vol. 5189. LNCS. Springer, 2008, pp. 104–117. URL: https://www-old.cdc.informatik.tu-darmstadt.de/reports/reports/REDBP08.pdf (cit. on pp. 72, 91).

[Rom90] John Rompel. "One-way functions are necessary and sufficient for secure signatures." In: *Symposium on Theory of Computing – STOC '90*. ACM, 1990, pp. 387–394. URL: https://www.cs.princeton.edu/courses/archive/spr08/cos598D/Rompel.pdf (cit. on p. 43).

[RR02] Leonid Reyzin and Natan Reyzin. "Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying." In: *Information Security and Privacy – ACISP 2002*. Ed. by Lynn Batten and Jennifer Seberry. Vol. 2384. LNCS. Springer, 2002, pp. 144–153. URL: https://eprint.iacr.org/2002/014 (cit. on p. 87).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126 (cit. on p. 13).

[Saa17a] Markku-Juhani O. Saarinen. *HILA5: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. 2017. URL: https://mjos.fi/hila5 (cit. on p. 191).

[Saa17b] Markku-Juhani O. Saarinen. "Ring-LWE Ciphertext Compression and Error Correction: Tools for Lightweight Post-Quantum Cryptography." In: *IoT Privacy, Trust, and Security – IoTPTS 2017*. ACM, 2017, pp. 15–22. URL: https://eprint.iacr.org/2016/1058 (cit. on p. 171).

[Sak07] Halvor Sakshaugh. "Security analysis of the NTRUEncrypt public key encryption scheme." MA thesis. Norwegian University of Science and Technology, 2007. URL: https://brage.bibsys.no/xmlui/handle/11250/258846 (cit. on pp. 173, 176).

[Sal05] David Salomon. *Coding for Data and Computer Communications.* Springer, 2005. ISBN: 978-0-38-721245-6 (cit. on p. 13).

[SAL+17] Nigel P. Smart, Martin R. Albrecht, Yehuda Lindell, Emmanuela Orsini, Valery Osheter, Kenny Paterson, and Guy Peer. *LIMA: Algorithm Specification and Supporting Documentation.* Submission to

NIST's Post-Quantum Cryptography Standardization project. 2017. URL: `https://lima-pq.github.io` (cit. on p. 191).

[SBGM+18] Markku-Juhani O. Saarinen, Sauvik Bhattacharya, Oscar Garcia-Morchon, Ronald Rietman, Ludo Tolhuizen, and Zhenfei Zhang. "Shorter Messages and Faster Post-Quantum Encryption with Round5 on Cortex M." In: *Smart Card Research and Advanced Applications – CARDIS 2018*. Ed. by Begül Bilgin and Jean-Bernard Fischer. Vol. 11389. LNCS. Springer, 2018, pp. 95–110. URL: `https://eprint.iacr.org/2018/723/20181013:085018` (cit. on pp. 191, 210–212).

[Sch03] Claus-Peter Schnorr. "Lattice Reduction by Random Sampling and Birthday Methods." In: *Symposium on Theoretical Aspects of Computer Science – STACS 2003*. Ed. by Helmut Alt and Michel Habib. Vol. 2607. LNCS. Springer, 2003, pp. 145–156. URL: `https://www.math.uni-frankfurt.de/~dmst/research/papers/ABRStacs.pdf` (cit. on p. 171).

[Sec17] Safran Identity & Security. *The impact of Java Card technology yesterday and tomorrow: Safran Identity & Security celebrates 20 years with the Java Card Forum*. Press Release. accessed 2018-03-12. 2017. URL: `https://www.morpho.com/en/media/impact-java-card-technology-yesterday-and-tomorrow-safran-identity-security-celebrates-20-years-java-card-forum-20170302` (cit. on p. 73).

[Sec17] Security Innovation. *Security Innovation Makes NTRUEncrypt Patent-Free*. 2017. URL: `https://www.securityinnovation.com/company/news-and-events/press-releases/security-innovation-makes-ntruencrypt-patent-free` (cit. on p. 172).

[Sho97] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *Journal on Computing* 26.5 (1997), 1484–1509 (cit. on p. 14).

[Sil99] Joseph H. Silverman. *Almost inverses and fast NTRU key creation*. Tech. rep. #014. Version 1. `https://assets.onboardsecurity.com/static/downloads/NTRU/resources/NTRUTech014.pdf`. NTRU Cryptosystems, 1999 (cit. on pp. 184, 186).

[Sin99] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Doubleday, 1999. ISBN: 978-1-85-702879-9 (cit. on p. 13).

[SOO+95] Richard Schroeppel, Hilarie Orman, Sean O'Malley, and Oliver Spatscheck. "Fast key exchange with elliptic curve systems." In: *Advances in Cryptology – CRYPTO '95*. Ed. by Don Coppersmith. Vol. 963. LNCS. Springer, 1995, pp. 43–56. URL: `https://www.cs.arizona.edu/sites/default/files/TR95-03.pdf` (cit. on p. 186).

[SS17] Peter Schwabe and Ko Stoffelen. "All the AES you need on Cortex-M3 and M4." In: *Selected Areas in Cryptology – SAC 2016*. Ed. by Roberto Avanzi and Howard Heys. Vol. 10532. LNCS. Springer, 2017, pp. 180–194. URL: `https://eprint.iacr.org/2016/714` (cit. on p. 198).

[SS71] Arnold Schönhage and Volker Strassen. "Schnelle multiplikation grosser zahlen." In: *Computing* 7.3 (1971), pp. 281–292 (cit. on p. 182).

[SSH11] Koichi Sakumoto, Taizo Shirai, and Harunaga Hiwatari. "Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials." In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Springer, 2011, pp. 706–723. URL: `https://www.iacr.org/archive/crypto2011/68410703/68410703.pdf` (cit. on pp. 114, 120, 121, 124, 127, 130, 133, 136, 145–148, 155, 156, 165, 166).

[Sta05] Martijn Stam. "A Key Encapsulation Mechanism for NTRU." In: *Cryptography and Coding*. Ed. by Nigel P. Smart. Vol. 3796. LNCS. Springer, 2005, pp. 410–427 (cit. on pp. 173, 176).

[Ste93] Jacques Stern. "A new identification scheme based on syndrome decoding." In: *Advances in Cryptology – CRYPTO '93*. Ed. by Douglas R. Stinson. Vol. 773. LNCS. Springer, 1993, pp. 13–21. URL: `https://www.di.ens.fr/~stern/data/St47.pdf` (cit. on p. 124).

[Ste96] Jacques Stern. "A new paradigm for public key identification." In: *Transactions on Information Theory* 42.6 (1996), pp. 1757–1768. URL: `https://www.di.ens.fr/users/stern/data/St55b.pdf` (cit. on p. 133).

[SXY18] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. "Tightly-secure key-encapsulation mechanism in the quantum random oracle model." In: *Advances in Cryptology – EUROCRYPT 2018.* Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer. 2018, pp. 520–551. URL: `https://eprint.iacr.org/2017/1005` (cit. on p. 179).

[TGT+10] Shigeo Tsujii, Masahito Gotaishi, Kohtaro Tadaki, and Ryou Fujita. "Proposal of a Signature Scheme Based on STS Trapdoor." In: *Post-Quantum Cryptography – PQCrypto 2010.* Ed. by Nicolas Sendrier. Vol. 6061. LNCS. Springer, 2010, pp. 201–217. URL: `https://eprint.iacr.org/2010/118` (cit. on p. 113).

[Tho13] Enrico Thomae. "About the Security of Multivariate Quadratic Public Key Schemes." PhD thesis. Ruhr-University Bochum, Germany, 2013. URL: `https://www.iacr.org/phds/116_EnricoThomae_AboutSecurityMultivariateQuadr.pdf` (cit. on pp. 113, 120).

[Too63] Andrei L. Toom. "The complexity of a scheme of functional elements realizing the multiplication of integers." In: *Soviet Mathematics Doklady* 3 (1963), pp. 714–716. URL: `www.de.ufpe.br/~toom/my-articles/engmat/MULT-E.PDF` (cit. on pp. 182, 199).

[TU15] Ehsan Ebrahimi Targhi and Dominique Unruh. *Quantum Security of the Fujisaki-Okamoto and OAEP Transforms.* IACR Cryptology ePrint Archive, Report 2015/1210. 2015. URL: `https://eprint.iacr.org/2015/1210` (cit. on pp. 176, 177).

[TW12] Enrico Thomae and Christopher Wolf. "Cryptanalysis of Enhanced TTS, STS and All Its Variants, or: Why Cross-Terms Are Important." In: *Progress in Cryptology – AFRICACRYPT 2012.* Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. LNCS. Springer, 2012, pp. 188–202 (cit. on p. 113).

[Unr15] Dominique Unruh. "Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model." In: *Advances in Cryptology – EUROCRYPT 2015.* Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. LNCS. Springer, 2015, pp. 755–784. URL: `https://eprint.iacr.org/2014/587` (cit. on pp. 18, 114, 115, 145, 147, 154, 177).

[Unr17] Dominique Unruh. “Post-Quantum Security of Fiat-Shamir.” In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. LNCS. Springer. 2017, pp. 65–95. URL: `https://eprint.iacr.org/2017/398` (cit. on pp. 117, 136).

[WHC+13] Patrick Weiden, Andreas Hülsing, Daniel Cabarcas, and Johannes Buchmann. *Instantiating Treeless Signature Schemes*. IACR Cryptology ePrint Archive, Report 2013/065. 2013. URL: `https://eprint.iacr.org/2013/065` (cit. on p. 138).

[Win84] Robert S. Winternitz. “A secure one-way hash function built from DES.” In: *Symposium on Security and Privacy – S&P 1984*. Ed. by Dorothy E. Denning and Jonathan K. Millen. IEEE, 1984, pp. 88–90 (cit. on p. 78).

[WP06] André Weimerskirch and Christof Paar. *Generalizations of the Karatsuba Algorithm for Efficient Implementations*. IACR Cryptology ePrint Archive, Report 2006/224. 2006. URL: `https://eprint.iacr.org/2003/172` (cit. on p. 200).

[WS16] Bas Westerbaan and Peter Schwabe. “Solving binary $\mathcal{MQ}$ with Grover’s algorithm.” In: *Security, Privacy, and Advanced Cryptography Engineering – SPACE 2016*. Ed. by Claude Carlet, Anwar Hasan, and Vishal Saraswat. Vol. 10076. LNCS. Springer, 2016, pp. 303–322. URL: `https://eprint.iacr.org/2019/151` (cit. on p. 157).

[Wun16] Thomas Wunderer. *Revisiting the Hybrid Attack: Improved Analysis and Refined Security Estimates*. IACR Cryptology ePrint Archive, Report 2016/733. 2016. URL: `https://eprint.iacr.org/2016/733` (cit. on p. 171).

[YC04] Bo-Yin Yang and Jiun-Ming Chen. “Theoretical Analysis of XL over Small Fields.” In: *Information Security and Privacy – ACISP 2004*. Ed. by Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan. Vol. 3108. LNCS. Springer, 2004, pp. 277–288. URL: `http://www.iis.sinica.edu.tw/papers/byyang/2386-F.pdf` (cit. on pp. 157, 168).

[YC05a] Bo-Yin Yang and Jiun-Ming Chen. “All in the XL Family: Theory and Practice.” In: *Information Security and Cryptology – ICISC 2004*. Ed. by Choon sik Park and Seongtaek Chee. Springer, 2005, pp. 67–86.

URL: `http://by.iis.sinica.edu.tw/by-publ/recent/xxl.pdf` (cit. on pp. 137, 157).

[YC05b] Bo-Yin Yang and Jiun-Ming Chen. "Building Secure Tame-like Multivariate Public-Key Cryptosystems: The New TTS." In: *Information Security and Privacy – ACISP 2005.* Ed. by Colin Boyd and Juan Manuel González Nieto. Vol. 3574. LNCS. Springer, 2005, pp. 518–531. URL: `http://www.iis.sinica.edu.tw/papers/byyang/2381-F.pdf` (cit. on p. 113).

[YCC04a] Bo-Yin Yang, Jiun-Ming Chen, and Yen-Hung Chen. "TTS: High-Speed Signatures on a Low-Cost Smart Card." In: *Cryptographic Hardware and Embedded Systems – CHES 2004.* Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. LNCS. Springer, 2004, pp. 371–385. URL: `https://www.iacr.org/archive/ches2004/31560371/31560371.pdf` (cit. on p. 113).

[YCC04b] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. "On Asymptotic Security Estimates in XL and Gröbner Bases-Related Algebraic Cryptanalysis." In: *Information and Communications Security – ICICS 2004.* Ed. by Javier Lopez, Sihan Qing, and Eiji Okamoto. Vol. 3269. LNCS. Springer, 2004, pp. 401–413. URL: `http://www.iis.sinica.edu.tw/papers/byyang/2384-F.pdf` (cit. on p. 168).

[YCC+06] Bo-Yin Yang, Chen-Mou Cheng, Bor-Rong Chen, and Jiun-Ming Chen. "Implementing Minimized Multivariate PKC on Low-Resource Embedded Systems." In: *Security in Pervasive Computing – SPC 2006.* Ed. by John A. Clark, Richard F. Paige, Fiona A. C. Polack, and Phillip J. Brooke. Vol. 3934. LNCS. Springer, 2006, pp. 73–88. URL: `http://precision.moscito.org/by-publ/recent/39340073.pdf` (cit. on p. 92).

[YCY13] Jenny Yuan-Chun Yeh, Chen-Mou Cheng, and Bo-Yin Yang. "Operating Degrees for XL vs. F4/F5 for Generic $\mathcal{MQ}$ with Number of Equations Linear in That of Variables." In: *Number Theory and Cryptography: Papers in Honor of Johannes Buchmann on the Occasion of His 60th Birthday.* Ed. by Marc Fischlin and Stefan Katzenbeisser. Springer, 2013, pp. 19–33. URL: `http://www.iis.sinica.edu.tw/papers/byyang/17377-F.pdf` (cit. on p. 137).

[ZCH+17] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. *NTRUEncrypt: Algorithm Specification and Supporting Documentation.* Submission to NIST's Post-Quantum Cryptography Standardization project. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`. 2017 (cit. on pp. 191, 194).

Symbols and acronyms

Here, we provide an alphabetized list of symbols, acronyms and other formalisms. Note that symbols and notation may overlap between chapters (or even within a chapter) with distinct meaning. When this is the case, it will be clarified explicitly. Names of schemes (such as AES, WOTS and NTRU-HRSS) are not included. An underscore indicates that the symbol includes a subscript or superscript.

$[\ldots]_q, [\ldots]_3$	Computation in $\mathcal{R}_q$ and $\mathcal{S}_3$, respectively
$\circledast$	Multiplication in $\mathcal{R}_q$
$\mathcal{A}$	Algorithm representing an adversary
α	In Section 3.7, input length of a tweakable hash function; in Chapter 4, a challenge in $\mathbb{F}_q$
$\alpha^{(\_)}$	A challenge in $\mathbb{F}_q$
AVX2	Advanced Vector Extensions 2
BDS_k	BDS trade-off parameter
BMI	Bit Manipulation Instruction Sets
c	Checksum in one-time signature schemes
$c_{\_}, c_{\_}^{(\_)}$	Commitment in identification schemes
c	Ciphertext (see Definition 2.1.10)
$\mathcal{C}$	Algorithm representing a challenger
$\mathsf{ch}, \mathsf{ch}^{(\_)}$	Challenge
ChS	Challenge space
CLMUL	Carry-less Multiplication
$Com, Com_{\_}$	Commitment function
$\mathsf{com}, \mathsf{com}^{(\_)}$	Commitment string
$\mathsf{cr}_{\_}^{(\_)}$	Blinded response to a challenge
d	Number of layers in a hypertree
DSP	Digital Signal Processing extensions
$\mathbf{e}_0^{(\_)}, \mathbf{e}_1^{(\_)}$	Challenge-dependent shares of $\mathbf{F}(\mathbf{r}_0^{(\_)})$

EU-CMA	Existential unforgeability under adaptive chosen message attacks (see Definition 2.1.8)
$\mathbb{F}_{\_}, \mathbb{F}_q$	Finite field (of order q)
$\mathbb{F}_q^n$	Vector of elements of a finite field of order q
$\mathcal{F}$	Pseudorandom function to generate bitmasks
F	One-way function
$\mathbf{F}$	System of multivariate quadratic equations
f	Secret key polynomial in $\mathcal{S}_p$; from Section 5.3 onwards, the quotient of $\mathcal{R}_q$
f_q^{-1}	Inverse of f with respect to $\mathcal{R}_q$
f_3^{-1}	Inverse of f with respect to $\mathcal{S}_3$
FTS	Few-time signature scheme
$\mathbf{G}$	Polar form of a system of multivariate quadratic equations
$G_s, G_{\mathrm{sk}}, G_{\mathcal{S}_F}, G_{\mathrm{rte}}$	Pseudorandom generators
H	Hash function used to combine two tree nodes
$H_1, H_2, \mathcal{H}$	Cryptographic hash functions
$\mathrm{H}_r, \mathrm{H}_{\mathrm{ss}}, \mathrm{H}_{\mathrm{qrom}}$	Cryptographic hash functions
h	Public key polynomial in $\mathcal{R}_q$
h_1, h_2	Hash digests of partial identification scheme transcripts
h	Height of a binary tree; total height of a hypertree
$I_{\_}, B_{\_}$	Indices of blinded responses in Unruh's transform
IDS	Identification scheme
IETF	Internet Engineering Task Force
IND-CPA	Indistinguishability under adaptive chosen-ciphertext attacks (see Definition 2.1.12)
k	Security parameter; see Section 2.1.1
κ	Number of trees in FORS; soundness error in an identification scheme
KEM	Key-encapsulation mechanism (see Definition 2.1.10)
ℓ_1, ℓ_2, ℓ	Length of (part of) a Winternitz signature
ℓ	In Section 5.3, the dimension of a modular lattice
m	Message; in Chapter 4, number of equations in a system of equations
m	In Chapter 4, a message
md	Message digest

$\mathbb{N}$	The set of natural numbers
n	In Section 3.1, message length; from Section 3.3 onwards, length of a hash digest; in Chapter 4, number of variables in a multivariate equation; in Chapter 5, degree of a polynomial
NIST	United States National Institute of Standards and Technology
OTS	One-time signature scheme
P	Public parameter input into a tweakable hash function
$\mathcal{P}, \mathcal{P}_{\_}$	Algorithm representing a prover
Φ_d	The d^{th} cyclotomic polynomial
p	Number of signatures in a many-time signature scheme (see Definition 3.2.1); in Chapter 5, integer co-prime to q
$p_{\_}, p_{\_}^{(\_)}$	Public values in a one-time or few-time signature scheme
pk	Public key
$pk_{\_}$	Public key in a many-time signature scheme
PKE	Public-key encryption scheme (see Definition 2.1.9)
PRF	Pseudorandom function (see Definition 2.1.6)
PRG	Pseudorandom generator (see Definition 2.1.4)
PRP	Pseudorandom permutation (see Definition 2.1.3)
q	In Chapter 4, order of finite field $\mathbb{F}_q$; in Chapter 5, integer co-prime to p
QROM	Quantum random oracle model
R	Index randomization value
$\mathcal{R}_p, \mathcal{R}_q$	Rings $\mathbb{Z}_p[x]/(f)$ and $\mathbb{Z}_q[x]/(f)$, with $f = x^n - 1$ up until Section 5.3
$\mathbf{r}_0^{(\_)}, \mathbf{r}_1^{(\_)}$	Vectors of elements of $\mathbb{F}_q$, combining to $\mathbf{s}$
r	In Chapter 4, number of rounds in a transformed identification scheme; In Chapter 5, random polynomial
$r_{\_}$	Root of a tree in FORS
$\mathrm{resp}, \mathrm{resp}_{\_}^{(\_)}$	Response to a challenge in an identification scheme
RNG	Random number generator
ROM	Random oracle model
$\mathcal{S}$	Algorithm representing a simulator
$\mathcal{S}_{\mathrm{sk}}, \mathcal{S}_F, \mathcal{S}_\rho, \mathcal{S}_{\mathrm{rte}}$	Random seeds
$\mathcal{S}_p, \mathcal{S}_q$	Rings $\mathbb{Z}[x]/(p, \Phi_n)$ and $\mathbb{Z}[x]/(q, \Phi_n)$, for fixed n

$\mathbf{s}$	Secret input vector to a system of multivariate equations
$s_{\_}, s_{\_}^{(\_)}$	Secret values in a one-time or few-time signature scheme
$\sigma, \sigma_{\_}$	(Part of) a signature
SIMD	Single instruction, multiple data
sk	Secret key
$sk_{\_}$	Evolving secret key in a many-time signature scheme
ss	Shared secret (see Definition 2.1.10)
T	Tweak input into a tweakable hash function
Θ	Average-case complexity
$\mathbf{t}_0^{(\_)}, \mathbf{t}_1^{(\_)}$	Challenge-dependent shares of $\mathbf{r}_0^{(\_)}$
t	Number of revealed values in HORST; height of a tree in FORS; number of transcripts per round in Unruh's transform
trans	Transcript of an identification scheme
$\mathcal{V}$	Algorithm representing a verifier
$\mathbf{v}$	Public output vector of a system of multivariate equations
w	Winternitz parameter
$\mathbf{x}, \mathbf{y}$	Generic vector of elements of $\mathbb{F}_q$
$\mathbf{x}_{high}, \mathbf{x}_{low}$	Low and high halves of $\mathbf{x}$
x	In Section 3.5.2, HORST layer included in the signature
XOF, $\text{XOF}_{\_}$	Extendable output function (see Definition 2.1.5)
$\mathbb{Z}$	The set of integers

Summary

In anticipation of a universal quantum computer that is able to break current-day cryptography, this thesis describes efforts towards practical post-quantum primitives. To this end, we construct efficient digital signature schemes and key exchange protocols and provide highly optimized software implementations.

Chapter 1 and 2 introduce the problem and give context, both in terms of cryptographic definitions and with regards to engineering. In particular, we introduce NIST's Post-Quantum Cryptography Standardization project: an ongoing standardization effort for which much of the work in this thesis has direct relevance. The main body of this work is divided in three chapters, each describing contributions to a separate subfield of post-quantum cryptography.

Chapter 3: Hash-based signatures

This is the first of two chapters on digital signatures. With constructions dating back to the seventies, rigorous security proofs, and very few assumptions with regard to the security of the underlying building blocks, hash-based signature schemes are generally considered to be the most robust choice and likely to hold up for decades to come. This comes at the cost of efficiency and usability: their considerably larger signatures, higher computational costs, and non-standard interfaces may be a costly replacement for current-day signature schemes.

In this chapter, we describe various constructions leading up to XMSS and SPHINCS$^+$, the state-of-the-art in hash-based signature schemes. We discuss scheme design and describe implementation aspects, focusing in particular on embedded platforms. Besides reference implementations, we present an implementation of XMSS$^{\text{MT}}$ on the Java Card smart card platform and adapt the SPHINCS implementation to run on a board with less memory available than the size of a single signature. We then present the SPHINCS$^+$ framework as submitted to NIST, and evaluate several instances optimized for speed and signature size.

Chapter 4: $\mathcal{MQ}$-based signatures

The second class of digital signatures addressed in this thesis finds its basis in multivariate quadratic equations. While $\mathcal{MQ}$-based signatures are among the smallest possible, the security of their constructions is not always well-understood.

In this chapter we take a non-standard approach to designing signature schemes based on the $\mathcal{MQ}$ problem, and present MQDSS and SOFIA. Both schemes are based on a transformed identification scheme and come with (non-tight) security proofs, using variants of the Fiat-Shamir transform and Unruh's transform, respectively. We instantiate the schemes and describe highly optimized implementations of MQDSS-31-64 and SOFIA-4-128, making extensive use of Intel's AVX2 vector extensions to evaluate the $\mathcal{MQ}$ function for carefully chosen parameters. MQDSS is a contender in round 2 of NIST's standardization project.

Chapter 5: Lattice-based KEMs

The final chapter considers lattice-based key-encapsulation mechanisms (KEMs). KEMs that rely on the hardness of lattice problems are among the most efficient proposals for a post-quantum key exchange, each with their own unique structures and choices. As key exchange is arguably the most urgent concern, it is no surprise that experiments with deployment in this category are well on their way.

In the first half of this chapter, we describe a carefully tweaked variant of the NTRU scheme: NTRU-HRSS. We present a highly optimized AVX2 implementation, focusing in detail on the optimization of its specific multiplication and inversion operations. NTRU-HRSS is part of the round 2 NTRU submission to NIST's standardization project. In the remainder of this chapter we consider multiplication operations for lattice-based KEMs more generally, and present an automated design-space exploration that combines various instances of Karatsuba and Toom-Cook multiplication. This leads to record-setting code-generation routines for multiplication of polynomials in $\mathbb{Z}_{2^m}[x]$ on the ARM Cortex-M4.

Samenvatting

In afwachting van een algemeen toepasbare quantumcomputer die in staat is om moderne cryptografie te breken, draagt dit proefschrift bij aan de ontwikkeling van praktisch bruikbare post-quantumcryptografie. Met dit doel in gedachten ontwerpen we efficiënte systemen voor digitale handtekeningen en sleuteluitwisselingsprotocollen, en voorzien we in tot in detail geoptimaliseerde software.

Hoofdstuk 1 en 2 introduceren het probleem en schetsen de context, zowel op het gebied van cryptografie als ook qua softwareontwikkeling. In het bijzonder bespreken we NISTs 'Post-Quantum Cryptography Standardization project:' een lopend standaardisatieproject waarvoor een groot deel van dit proefschrift direct relevant is. De kern van dit proefschrift is verdeeld in drie hoofdstukken die elk ingaan op bijdragen aan een deelgebied van de post-quantumcryptografie.

Hoofdstuk 3: Handtekeningen op basis van hashfuncties

Dit is het eerste van twee hoofdstukken over digitale handtekeningen. Vanwege constructies die teruggaan tot de jaren zeventig, zorgvuldige bewijzen en bescheiden aannames met betrekking tot de veiligheid van de onderliggende bouwstenen worden handtekeningen op basis van hashfuncties gezien als de meest robuuste keuze; één die nog vele jaren meegaat. Dit gaat wel ten koste van efficiëntie en gebruiksgemak: de grote handtekeningen, rekenkosten en ongebruikelijke interfaces maken het een dure onderneming om huidige handtekeningen te vervangen.

In dit hoofdstuk beschrijven we de diverse constructies die uiteindelijk leiden tot XMSS en SPHINCS$^+$, de state-of-the-art wat betreft handtekeningen op basis van hashfuncties. We bespreken implementatieaspecten en het ontwerp, en gaan in het bijzonder in op geïntegreerde systemen. Naast referentiecode geven we een implementatie van XMSS^{MT} voor de Java Card, en passen we de code van SPHINCS aan zodat het draait op een 'computer' met minder geheugen dan de grootte van een enkele handtekening. Verder beschrijven we SPHINCS$^+$ zoals ingediend naar NIST en bespreken we de vele mogelijke variaties in snelheid en grootte.

Hoofdstuk 4: Handtekeningen op basis van $\mathcal{MQ}$

De tweede soort handtekeningen die we behandelen, is gebaseerd op meerdimensionale kwadratische vergelijkingen ($\mathcal{MQ}$). Dit type handtekeningen is veruit het meest compact, maar de betrouwbaarheid wordt niet altijd even goed doorgrond.

In dit hoofdstuk hanteren we een atypische aanpak voor handtekeningen gebaseerd op het $\mathcal{MQ}$-probleem, wat leidt tot MQDSS en SOFIA. Beide systemen zijn gebaseerd op een getransformeerd identificatieprotocol en worden vergezeld door een (niet-strikt) bewijs van veiligheid. Hiervoor maken we gebruik van varianten van respectievelijk de transformatie van Fiat-Shamir en van Unruh. We geven specifieke varianten van de constructies en beschrijven zorgvuldig geoptimaliseerde implementaties van MQDSS-31-64 en SOFIA-4-128, waarbij we uitvoerig gebruik maken van Intels AVX2 vector-uitbreidingen om de $\mathcal{MQ}$-functie te berekenen. MQDSS is een kandidaat in de tweede ronde van NISTs standaardisatieproject.

Hoofdstuk 5: Sleutelinkapseling op basis van roosters

Het laatste hoofdstuk gaat in op mechanismes voor sleutelinkapseling (KEMs) op basis van roosters. KEMs gebaseerd op wiskundige problemen gerelateerd aan roosters zijn de meest efficiënte kandidaten voor post-quantum-sleuteluitwisseling, elk met hun eigen unieke structuren en ontwerpkeuzes. Sleuteluitwisseling lijkt het meest urgente probleem, dus het is niet verrassend dat praktijkgerichte experimenten in deze categorie al in volle gang zijn.

In het eerste deel van dit hoofdstuk beschrijven we een zorgvuldig afgestemde variant van NTRU: NTRU-HRSS. We geven een nauwlettend geoptimaliseerde AVX2-implementatie en gaan in detail in op de optimalisatie van de specifieke vermenigvuldigingen en inversies. NTRU-HRSS is onderdeel van NTRU, een kandidaat in de tweede ronde van NISTs standaardisatieproject. In de rest van dit hoofdstuk gaan we in algemenere zin in op vermenigvuldigingen voor KEMs gebaseerd op roosters, en beschrijven we een geautomatiseerde zoektocht waarin we combinaties maken van diverse varianten van Karatsuba- en Toom-Cook-vermenigvuldiging. Door middel van codegeneratie leidt dit tot recordbrekende routines voor vermenigvuldiging van polynomen in $\mathbb{Z}_{2^m}[x]$ op de ARM Cortex-M4.

About the author

Joost was born on December 11, 1992, in Culemborg, The Netherlands, where he graduated cum laude with a bilingual vwo degree from ORS Lek en Linge in 2010.

Joost completed his bachelor degree in Computing Science cum laude in 2013 with a thesis entitled *"How the Dutch broke the Japanese Blue Code in the late 1930s"* under the supervision of Bart Jacobs. He then enrolled in the Kerckhoffs' programme, a joint master in computer security offered by Radboud University, Eindhoven University of Technology and University of Twente, graduating cum laude at Radboud University in 2015. His master thesis, *"Implementing SPHINCS with restricted memory"*, was supervised by Peter Schwabe and Andreas Hülsing.

In 2015, Joost started as a Ph.D. student in the Digital Security group at Radboud University, under the supervision of Peter Schwabe. Working on practical software implementations of post-quantum cryptography, his position was funded as part of the Horizon 2020 EU PQCRYPTO project. This thesis is the result of that work.

Over the summer of 2019, Joost was an intern in the cryptographic engineering group at Apple in Cupertino, California, supervised by Yannick Sierra.

Academic publications

The following is a list of academic publications that Joost coauthored (in reverse-chronological order). This includes both peer-reviewed work and preprints — the latter are marked accordingly. Authors are ordered alphabetically.

9. Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS$^+$ signature framework." In: *Conference on Computer and Communications Security – CCS '19. To appear.* ACM, 2019.

8. Pol Van Aubel, Erik Poll, and Joost Rijneveld. "Non-Repudiation and End-to-End Security for EV-charging." In: *Innovative Smart Grid Technologies Europe 2019. To appear.* IEEE, 2019.

7. Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. “Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates.” In: *Applied Cryptography and Network Security – ACNS 2019*. Vol. 11464. LNCS. Springer, 2019.

6. Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. “Is Java Card ready for hash-based signatures?” In: *Advances in Information and Computer Security – IWSEC 2018*. Vol. 11049. LNCS. Springer, 2018.

5. Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. “SOFIA: MQ-based signatures in the QROM.” in: *Public Key Cryptography – PKC 2018*. Vol. 10770. LNCS. Springer, 2018.

4. Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. “High-speed key encapsulation from NTRU.” in: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Vol. 10529. LNCS. Springer, 2017.

3. Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. “From 5-Pass $\mathcal{MQ}$-Based Identification to $\mathcal{MQ}$-Based Signatures.” In: *Advances in Cryptology – ASIACRYPT 2016*. Vol. 10032. LNCS. Springer, 2016.

2. Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. “ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM.” in: *Public Key Cryptography – PKC 2016*. Vol. 9614. LNCS. Springer, 2016.

1. Andreas Hülsing, Joost Rijneveld, and Fang Song. “Mitigating Multi-Target Attacks in Hash-based Signatures.” In: *Public Key Cryptography – PKC 2016*. Vol. 9614. LNCS. Springer, 2016.

Technical publications

The following is a list of technical publications that Joost coauthored (in reverse-chronological order).

5. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. Second NIST PQC Standardization Conference. 2019.

4. Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. Request for Comments 8391. IETF, 2018.

3. Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. *NTRU-KEM-HRSS: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2017.

   superseded by

   Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. *NTRU: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project. 2019.

2. 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, and Peter Schwabe. *SPHINCS*$^+$. Submission to the NIST Post-Quantum Cryptography Standardization project. 2017.

1. Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. *MQDSS*. Submission to NIST's Post-Quantum Cryptography Standardization project. 2017.