WARP DRIVE ENGINEERING

IMPLEMENTING AND OPTIMIZING THE DILITHIUM SIGNATURE SCHEME

AMBER SPRENKELS

Warp Drive Engineering

Implementing and optimizing the Dilithium signature scheme

Amber Sprenkels

Printed by Proefschriftspecialist, Zaandam
Cover design by Marilou Maes

ISBN 978-94-93391-67-3

Warp Drive Engineering

Implementing and optimizing the Dilithium signature scheme

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

dinsdag 3 december 2024
om 10.30 uur precies

door

Amber Daan Sprenkels

geboren op 12 april 1994
te Heesch

Promotor

Prof. dr. Peter Schwabe

Copromotor

Dr. Joppe Bos
NXP Semiconductors, België

Manuscriptcommissie

Prof. dr. Lejla Batina

Dr. Vadim Lyubashevsky
IBM Research Europe, Zwitserland

Prof. dr. ir. Nele Mentens
Leiden Universiteit, Nederland

Prof. dr. Damien Stehlé
École Normale Supérieure de Lyon, Frankrijk

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

Contents

1 Introduction

1.1 Digital signature schemes

When people talk directly to one another (e.g., while in the same room), they can trust that the words they hear from each other are unaltered. If nobody is around to listen in to the conversation, they can also trust that the contents of their conversation remain private, as long as the participants do not disseminate it further.

Unfortunately, many of our communications do not happen directly, but rather indirectly through intermediates or hops; be it a courier, a hard disk,[1] a wire, or a radio wave. Protecting these channels is often practically impossible, as couriers can be bribed; hard disks can be rewritten; and wires and radio waves can be intercepted. Consequently, most channels are by nature not trustworthy.

Throughout history, humans have applied mechanisms of enciphering messages to protect their communications from prying eyes, for example the Caesar substitution cipher, which was used by the Roman emperor Julius Caesar [Sin00]. Additionally, humans have recognized or invented mechanisms that are presumed hard to recreate by illegitimate parties. For instance, in the case of a €20 bill (which is essentially a statement from the European Central Bank that "this sheet of cotton is worth €20"), I trust its legitimacy because of its abundance of watermarks, microprints, special inks and other features that are hard to mimic. Maybe the most common example is the handwritten signature which is often presumed to be hard to mimic by anyone other than the original author. It is an indication from the author that they have agreed to the contents of the signed document.

In the digital world, the physical elements are often abstracted away, and we cannot rely on physical means to protect the confidentiality or authenticity of our communi-

[1] Technically, this data is not in transit but at rest. Still, what is a stored piece of data, if not a message to someone (even oneself) in the future?

cations. Cryptography provides us with a method to protect the confidentiality and authenticity of information by using mathematical operations.

1

One core cryptographic component for protecting the authenticity of a piece of data is a *digital signature scheme* [DH76]. A *digital signature* is a small string of bytes that accompanies a *message* (which is also a string of bytes). Signing identities—like the European Central Bank or the agreeing parties of a contract—are associated with *key pairs*, which consist of a *secret key* and a *public key*. A signing algorithm uses the secret key to generate a digital signature over some message. Later, a verification algorithm uses the message, the signature, and a public key to check whether the signature correctly authenticates the message. A correct signature indicates that the signature has been generated by somebody who knows the secret key (*authentication*), and that the message has been unaltered since (*integrity*).

Cryptographic signature schemes are such a valuable building block that they are—nowadays—used virtually everywhere. For example, Transport Layer Security (used to secure protocols like HTTPS, SMTPS, etc.) uses digital signatures to authenticate the identities of servers (and clients) on the web [IETF18]; cryptographic authenticators use digital signatures to replace (or supplement) passwords, authenticating users to online services [ITU18; W3C21]; messaging protocols use signatures to authenticate the identities of communicating parties [IETF23]; consumer devices use digital signatures to ensure the authentication and integrity of kernel images using secure boot [UEFI22]; certificate transparency logs provide auditability by anyone using signatures [IETF21]; et cetera. It is hard to express the extent to which the modern digital world relies on digital signature schemes. As such, it is paramount that they remain efficient and secure.

1.2 Post-quantum cryptography

Almost all currently deployed signature schemes are based on either the discrete logarithm problem (DLP) [DH76] (or its elliptic-curve variant (ECDLP) [Kob87; Mil86]) or the RSA problem (RSA) [RSA78]. These problems are presumed to be hard to solve for *classical computers*, i.e., the devices that we all know as computers. However, *quantum computers* can theoretically solve these problems efficiently using Shor's algorithm, and consequently completely break all the cryptographic systems based on them [Sho94]. At the time of writing, quantum computers are not yet very

powerful—far from being able to break current deployments of the DLP or the RSA problem [BSI20b; IBM24]. Yet, in recent years we have seen consistent advances in their capabilities. While few field experts expect a cryptographically relevant quantum computer (CRQC) to be constructed in the next 10 years, many expect that one will be built eventually [GRI23].

Consequently, the cryptographic community has shifted a lot of its focus towards developing *post-quantum cryptography*: cryptography that remains secure even in the presence of a CRQC. They rely on problems different than the DLP and the RSA problem. Because of the quantum-computing threat, some government and standardization bodies have started to standardize or recommend specific instances of post-quantum cryptographic schemes, e.g. the German BSI [BSI20a], the French ANSSI [ANSSI22], the (international) IRTF [IRTF18; IRTF19], or the American NSA [NSA22]. However, by far the largest centralized effort for the evaluation of post-quantum schemes was spearheaded by the American National Institute of Standards and Technology (NIST).

In 2016, NIST called for proposals for new post-quantum schemes to replace the existing standards for key establishment (SP 800-56A [NIST18] and SP 800-56B [NIST19b]) and digital signatures (FIPS 186-4 [NIST13]) [NIST16]. After receiving 69 submissions in 2017, NIST narrowed down to 26 schemes advancing to the second round in 2019, and 7 finalists in 2020 [NIST19a; NIST20a]. In the end, NIST standardized 4 schemes: Kyber for key encapsulation, and Dilithium [NIST22a], Falcon [PFHK$^+$22], and SPHINCS$^+$ for digital signatures. SPHINCS$^+$ [BHKN$^+$19]—which is based only on the security of hash functions—is considered too slow for many applications [BKNS20], leaving Dilithium [DKLL$^+$18] and Falcon, which are based on lattice optimization problems. Because of the inapplicability of SPHINCS$^+$, breakthroughs in attacks on lattices could leave implementors of post-quantum crypto with no standardized alternatives. Therefore, NIST called for additional signature schemes based on other hard problems (such as codes, multivariate, isogenies, or MPC-in-the-head) [NIST23a]. At the time of writing, that standardization process is still ongoing.

1.3 Dilithium

1

This thesis focuses on the *Dilithium signature scheme*. It was submitted to the NIST competition in 2017 by the CRYSTALS Team[2] together with its sibling Kyber [ABDK+17; DKLL+17], and published as an article in TCHES [DKLL+18]. Dilithium is a scheme which is based on the *module learning with errors* (MLWE) and *module shortest integer solution* (MSIS) hard problems. These are adaptations of the *learning with errors* (LWE) [Reg05] and *shortest integer solution* (SIS) [Ajt96] hard problems in lattice cryptography. The scheme comes in three parameter sets: Dilithium2, Dilithium3, and Dilithium5—corresponding to three different security levels specified by NIST—ranging from 128 to 256 claimed bits of security. Depending on the parameter set, the public keys are 1.3 to 2.6 kilobytes in size, and the signatures are 2.4 to 4.6 kilobytes in size.

The scheme is built around arithmetic of 256-coefficient polynomials modulo a 23-bit prime number $q = 2^{23} - 2^{13} + 1$. It also includes many calls to deterministic random number generation and hashing operations, which are all based on the SHAKE extensible output function [BDPA13; NIST15a].

The name of the scheme comes from the science fiction series Star Trek. Dilithium is a fictional material that occurs naturally in crystal form. In Star Trek, these dilithium crystals are a critical controlling substance in the *warp drive*: a fictional type of engine at the core of a spaceship that allows for travel faster than the speed of light (warp-speed). These warp drives, and the engineering thereof, are what inspired the title of this thesis.

1.4 Research objective

During the NIST competition, the cryptographic community set out to evaluate all the different schemes. Many evaluation perspectives have been considered, such as a scheme's security properties, their suitability for incorporation into existing protocols, and estimating their performance characteristics. Over the last five years, I have done

[2] CRYSTALS stands for "Cryptographic Suite for Algebraic Lattices", which consists of Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle (`https://web.archive.org/web/20240202143804/https://pq-crystals.org/`).

my best to contribute to this process by evaluating the question: *How do embedded software implementations of Dilithium perform?*

Most of the work presented in this thesis (Chapters 4 to 6) approaches this question by exploring possible optimization strategies to reduce the algorithm's latency or its memory footprint. We also evaluate the impact of adding the Dilithium scheme to an existing embedded system in Chapter 7.

1.5 Organization of this thesis

Chapter 2 covers all of the notation used in this thesis, describes some of the basic concepts in cryptographic engineering, and introduces the platforms that we will be optimizing for. Then Chapter 3 focuses on the preliminaries for the Dilithium signature scheme. Chapters 4 to 8 contain the main technical contributions of this thesis, which will be summarized in the next section. Finally, in Chapter 9, we will tie things together and formulate an outlook toward the future.

1.6 Contributions

All of the work presented in this thesis has been realized in collaboration with one or more co-authors. This section outlines the academic contribution that corresponds to each chapter, with a focus on the parts that I contributed personally. In some places I use the phrase *engineering work*, which I use to describe the entirety of developing software. This not only includes writing code, but also designing the software, setting up build and debugging setups, designing and writing tests, and writing and executing benchmarks. Lastly I would like to remark that, in our publishing culture, author lists are not ordered in order of contribution but alphabetically.

Chapter 4: Fast Dilithium on Cortex-M3 and Cortex-M4

In Chapter 4, we develop and present Dilithium implementations for the Arm Cortex-M3 and Cortex-M4 microcontroller platforms. This chapter is based on the paper that was published in TCHES 2021, Issue 1:

> Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2021). Artifact available at `https://artifacts.iacr.org/tches/2021/a1`, pp. 1–24. ISSN: 2569-2925. DOI: `10.46586/tches.v2021.i1.1-24`.

The original aim of the project was to manufacture fast implementations, as well as to analyze the general memory usage of those implementations. We achieved new speed records for Cortex-M4. Additionally, we wrote the first implementations of Kyber, Dilithium, and NewHope for Cortex-M3. The paper also contained an analysis of different strategies for memory reduction of Dilithium implementation. However, the presentation of that work has been moved to Chapter 6, such that all the work on memory improvements of Dilithium is combined into Chapter 6.

The main contribution of this project is the Cortex-M3 implementation, to which we all contributed an equal amount. The memory reductions (presented in Section 6.2) were devised and implemented predominantly by me. All of us contributed equally to the writing of the paper.

Chapter 5: NTT optimizations on the Cortex-M4

The second "speed-optimization" chapter describes various improvements to implementations of the Kyber and Dilithium number theoretic transforms (NTTs) for Cortex-M4. This chapter is based on the paper presented at ACNS 2022:

> Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. "Faster Kyber and Dilithium on the Cortex-M4." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, June 2022, pp. 853–871. DOI: `10.1007/978-3-031-09234-3_42`. URL: `https://eprint.iacr.org/2022/112`.

The publication presented three main contributions: First, we presented speed records for the Kyber NTT (i.e. a 16-bit NTT) by using improvement techniques that were thus far only used for other post-quantum schemes. Second, we found a faster instruction sequence for packed 16-bit Barrett reduction. Third, we presented a faster method to compute $c\mathbf{s_1}$ and $c\mathbf{s_2}$ in Dilithium, by using a smaller $q' \in \{257, 769\}$ instead

of the Dilithium q. We measured and reported on the speed improvements of the completed implementations.

My primary contribution to that paper was providing the insight that the $c\mathbf{s_1}$ and $c\mathbf{s_2}$ multiplications in Dilithium could be executed using NTTs using smaller moduli (i.e., the third contribution). Although I did contribute to the rest of the project in a qualitative fashion, all of the engineering work on the Kyber implementation was done by my coauthors. Therefore (and because the subject of this thesis is Dilithium and not Kyber), I have removed most of the parts about Kyber, changing the main focus of Chapter 5 to Dilithium.

Chapter 6: Dilithium for memory-constrained devices

Chapters 4 and 5 focus on improving Dilithium implementations in terms of execution speed. They prioritize speed at the cost of memory usage, which disqualifies more memory-constrained chips from running Dilithium. Therefore, in Chapter 6, we analyze and optimize the memory usage of Dilithium. We aim to gain insight into the general memory usage for a Dilithium implementation and to determine the minimal amount of memory needed to run Dilithium.

Section 6.2 is based on Section 5 of the paper published in TCHES 2021 ([GKS21], i.e. the paper that Chapter 4 is based on). In Section 6.2, we analyze different (high-level) time-memory tradeoffs that arise when implementing Dilithium. As mentioned earlier, that analysis was done mostly by me.

The rest of the chapter is based on the paper presented at AfricaCrypt 2022:

> Joppe W. Bos, Joost Renes, and Amber Sprenkels. "Dilithium for Memory Constrained Devices." In: *AFRICACRYPT 2022: 13th.* Ed. by Lejla Batina and Joan Daemen. Vol. 2022. Lecture Notes in Computer Science. Springer, July 2022, pp. 217–235. DOI: 10.1007/978-3-031-17433-9_10. URL: https://eprint.iacr.org/2022/323.

In the paper, we present a new memory-optimized implementation of Dilithium written purely in C. We use multiple different (low-level) memory-optimization strategies, such as the use of alternative NTTs modulo a smaller q', the compression of the polynomials c and $\mathbf{w}$, and carefully hand-crafted allocations of all the variables in the Keygen, Sign, and Verify algorithms. We measure the achieved memory usage and the impact on the execution speed of the algorithms.

This paper is one of two projects that I did during an internship at NXP Semiconductors during 2021–2022. All of the engineering work of this paper has been done solely by me, with the other authors providing technical and organizational guidance. We all contributed equally to the writing of the paper.

1

Chapter 7: Post-quantum secure boot on vehicle network processors

In Chapter 7, we divert from *implementing* Dilithium to *integrating* Dilithium. The previous chapters focus on improving Dilithium implementations, but their evaluation is based on in-vitro setups. For example, their benchmarks run only the signature algorithms (i.e., no operating system or application code), and the chips are configured for consistent measurements rather than real-world deployments. This approach is very suitable for reproducibility and comparability with other schemes and implementations, but it risks missing factors that arise when integrating the scheme into real-world applications. Chapter 7 aims to fill this knowledge gap. It is based on the paper published at ESCAR 2022:

> Joppe W. Bos, Brian Carlson, Joost Renes, Marius Rotaru, Amber Sprenkels, and Geoffrey P. Waters. "Post-quantum secure boot on vehicle network processors." In: *20th escar Europe - The World's Leading Automotive Cyber Security Conference (15. - 16.11.2022)*. Ruhr-Universität Bochum, 2022, pp. 112–125. DOI: 10.13154/294-9372. URL: https://eprint.iacr.org/2022/635

In the paper, we examined the real-world scenario of protecting kernel images using secure boot on the S32G274A vehicle network processor. Originally, the secure boot flow of the S32G274A processor only supported images that are signed by classical (i.e., pre-quantum) digital signatures. We created a fault-protected Dilithium signature verification algorithm and added it to the S32G274A's hardware security engine as an option for secure boot. Afterward, we examined the impact on the installation and boot times of the application firmware image.

This paper is the second project that I did during the internship at NXP. I integrated Dilithium into the S32G274A's hardware security engine in equal collaboration with Joost Renes, and together we acquired and analyzed the results. I wrote about one-third of the paper.

Chapter 8: Dilithium nonce recycling

This chapter came from the idea to experiment with parallelizing Dilithium across its rejection-sampling loop iterations, as—in crypto-engineering folklore—it is often best to parallelize code at the highest possible level. This idea evolved into the observation that, in some scenarios, particular values in Dilithium's rejection sampling could be reused across loop iterations. We identified two concrete optimizations, which when applied together result in a 3% – 6% speedup for Dilithium signing on Cortex-M4, and similar speedups for Cortex-M3 and AVX2. The chapter is based on the unpublished manuscript which is available as:

> Amber Sprenkels and Bas Westerbaan. *Don't throw your nonces out with the bathwater: Speeding up Dilithium by reusing the tail of y.* Cryptology ePrint Archive. 2021. URL: `https://eprint.iacr.org/2020/1158` (visited on Jan. 29, 2024).

The report proposes both Dilithium optimizations and describes the argumentation as to why we believe the modifications do not adversely affect the security of the scheme. We analyze the reduction in the number of primitive operations (e.g., Keccak permutes, (inverse) NTTs, etc.) and integrate the modifications into existing Dilithium implementations for Cortex-M3, Cortex-M4, and AVX2. For both modifications, we measure the reduction of the Dilithium signing latency.

The $\mathbf{z}$-check proposal (Section 8.3.1) was invented by me and the $\mathbf{r_0}$-check proposal (Appendix 8.A) was found in equal collaboration with Bas Westerbaan. I contributed the engineering work of the simulations and implementations that were involved in the performance analysis. The ePrint report was written in equal collaboration with Bas, after which the project was shelved. After the publishing of [BBDD+23], I picked up the project again and with guidance from Yi Lee I updated the security analysis. Section 8.6, which analyzes the applicability of our proposals to HAETAE, is based on discussions with Georg Land. All the changes and additions in Chapter 8 that were done after the submission of the of the ePrint report[3] are written by me.

[3] That is, version `20211216:094108`.

1.7 Artifacts and measurement data

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[4]

The following research software and datasets have been produced during this PhD research:

Speed-optimized round-2 Dilithium on Cortex-M3 and Cortex-M4 (Chapters 4 and 6). The software implementations of Dilithium and the benchmarking measurements that were recorded have been archived at

> Amber Sprenkels. *Speed-optimized round-2 Dilithium on Cortex-M3 and Cortex-M4*. 2024. DOI: 10.5281/zenodo.10706370. URL: https://doi.org/10.5281/zenodo.10706370.

Speed-optimized round-3 Dilithium on Cortex-M4 (Chapter 5). The software implementations of Dilithium and the benchmarking measurements that were recorded have been archived at

> Amber Sprenkels. *Speed-optimized round-3 Dilithium on Cortex-M4*. 2024. DOI: 10.5281/zenodo.10707141. URL: https://doi.org/10.5281/zenodo.10707141.

Memory-optimized round-3 Dilithium in pure C (Chapter 6). At the time of construction, the software implementation became closed-source property of NXP Semiconductors, and is not publicly archived. The benchmarking measurements have been archived at

> Amber Sprenkels. *Memory-optimized round-3 Dilithium in pure C*. 2024. DOI: 10.5281/zenodo.10708284. URL: https://doi.org/10.5281/zenodo.10708284.

[4] https://ru.nl/icis/research-data-management/, last accessed February 22nd, 2024.

Recycling nonces in Dilithium (Chapter 8). The software implementations of Dilithium and the benchmarking measurements that were recorded have been archived at

> Amber Sprenkels. *Dilithium nonce recycling experiments and benchmarks.* 2024. DOI: 10.5281/zenodo.10708819. URL: https://doi.org/10.5281/zenodo.10708819.

1

2 Preliminaries

The Dilithium signature scheme is such a central part of this thesis, that I have decided that it deserves its own chapter (Chapter 3). *This* chapter will set up the mathematical groundwork and conventions that will be used in the rest of this thesis. Unfortunately, the line between elements that are "a part of Dilithium", versus "general cryptographic knowledge" is very subjective. Which cryptographic elements are *basic*, and which ones are *specialized*? In making this distinction, I have loosely followed the principle that any unspecialized cryptographic engineer should already know most of the contents of this chapter as part of their basic repertoire. Even if they have never heard of Dilithium before, I estimate that most readers should be able to skip this chapter; and immediately move on to Chapter 3.

In Sections 2.2 and 2.3, we will first cover some of the necessary mathematical background and notation. Then, in Sections 2.4, 2.5, and 2.6, we will look at some cryptographic engineering fundamentals. In particular, we will look at some different modular reduction methods, and we will provide an overview of the Cortex-M4 and Cortex-M3 architectures, which will become relevant in Chapters 4 through 6.

2.1 Pronouns

As some of the PhD candidates who came before me have done ([Rij19; Vig21; Wig24]), I would like to take a moment to clarify the perspective in which this manuscript is written. By now, you may have noticed that both singular form (*I*) as well as plural form (*we*) are used dynamically throughout this text. This is in contrast to most academic writing, which is usually written by teams of researchers, and uses *we* exclusively.

Likewise in this thesis, *we* is used to refer to us, the researchers who did the work. Sometimes however, it includes you (the reader), as we take you with us as we move through the subject matter. Other times it refers to the cryptographic community in

the broader sense. However, there are times when I cannot speak for my coauthors. In these cases I will have to speak from my own soul, in which case I will use *I*.

2.2 Notation

This thesis follows general mathematical conventions. However, because our math intersects with computer programming, in some places we use some notation that might be unconventional. This section serves as a clarifying reference for the notation that you will find in the rest of the thesis.

Polynomials, vectors, and matrices. Let $\mathbb{Z}$ be the ring of integers, and let $\mathbb{Z}_q$ be the ring of integers modulo q. $\mathbb{Z}_q[X]$ denotes the polynomial ring in the variable X with all coefficients in $\mathbb{Z}_q$. $\mathbb{Z}_q[X]/\big(f(X)\big)$ denotes the quotient ring with all operations modulo q and $f(X)$.

Dilithium is built around matrices and vectors of polynomials in $\mathbb{Z}_q[X]/(X^n+1)$. To allow ourselves to distinguish these types of variables from one another, each of them has a different font. Polynomials are denoted by italic letters (e.g., c); vectors use lowercase variable names in boldface (e.g., $\mathbf{z}$); and matrices will use uppercase variable names in boldface (e.g., $\mathbf{A}$). Polynomial-coefficient indexing is *zero-based*; i.e., the first coefficient of c is c_0. This way, the polynomial coefficient a_k corresponds to X^k. However, vector and matrix indexing is *one-based*, i.e., the first element of $\mathbf{z}$ is $\mathbf{z}_1$.

Regular polynomial and vector multiplication is represented using conventional multiply operators (e.g., $\mathbf{Az}$ and $\mathbf{A} \cdot \mathbf{z}$). However, *pointwise* (or *coefficient-wise*) multiplication uses the $\circ$ operator (e.g., $\hat{c} \circ \hat{\mathbf{s}}$).

Last, the notation $\|a\|_\infty$ is used to describe the *uniform norm* (or *infinity norm*, or *sup norm*) of a. For some polynomial $a = a_0 + \cdots + a_{k-1}X^{k-1}$, the uniform norm $\|a\|_\infty$ is equal to $\max(|a_0|, \ldots, |a_{k-1}|)$. For vectors and matrices, the uniform norm is computed recursively from the elements, i.e., $\|\mathbf{z}\|_\infty = \max(\|\mathbf{z}_1\|_\infty, \ldots, \|\mathbf{z}_n\|_\infty)$.

Modular arithmetic. For integers, $a \bmod q$ and $a \bmod^+ q$ both denote the unique positive representation of a modulo q, such that $0 \leq a < q$. Additionally, $\bmod^\pm$ denotes the *centered* modulo operator, i.e., $a \bmod^\pm q$ is equal to the unique representation of a modulo q, such that $-\frac{q}{2} \leq a < \frac{q}{2}$. For vectors and polynomials, the mod, $\bmod^+$, and $\bmod^\pm$ operations are applied on a coefficient-wise basis of the polynomial(s).

NTT domain. Polynomials that are represented in the *number-theoretic transform NTT domain* (see Section 3.4) have a hat (e.g., $\hat{c}$). Moreover, because of the relation to the fast-Fourier transform algorithm, the NTT domain is sometimes called the "frequency domain". Conversely, the "time domain" (or "regular domain") corresponds to when polynomials are in their untransformed state.

Shell expansion. Dilithium comes in different variants, and we often find ourselves describing multiple of these variants at the same time. In these cases, braces indicate "shell expansion". For example, Dilithium{3,5} expands to "Dilithium3 and Dilithium5".

Algorithms. All algorithm names can be recognized by their sans-serif names (e.g., Dilithium2, NTT^{-1}). For variable names, we use either single letters; or we use names, in which case they also use a sans-serif font family (e.g., sk, pk, seed). We write $a := 42$ to denote variable assignment, and we write $a \xleftarrow{\$} \mathbb{A}$ to denote that the variable a is uniformly sampled from the set $\mathbb{A}$.

Rounding and bit-selection. By convention, $\lfloor x \rceil$ denotes the value of x rounded to the nearest integer with ties towards positive infinity; and $\lceil x \rceil$ and $\lfloor x \rfloor$ denote that x is rounded up and down respectively. By similar convention, all *measurement* data, whenever rounded, uses rounding with ties towards even numbers. Additionally, we use a custom syntax for bit-selection operations, based on [Pla21]. That is, $[x]_k = x \bmod 2^k$ selects the lowest k bits from x; and $[x]^k = \left\lfloor \frac{x}{2^k} \right\rfloor$ is equivalent to arithmetically shifting x by k positions to the right.

Units. When discussing algorithm speeds and sizes, cc denotes (clock) *cycles* (and kcc denotes kilocycles accordingly); KB denotes kilobytes (1000 bytes), and KiB denotes *kibibytes* (1024 bytes).

2.3 Signature schemes

Dilithium is a cryptographic *signature scheme*. The concept of a cryptographic signature scheme was first proposed in the seminal paper by Diffie and Hellman from 1976 [DH76]. The goal of a signature scheme is to provide *authentication* and to

protect the *integrity* of some *message*. In layman's terms: a cryptographic signature guarantees that a message originates from an identified sender (authentication) and that the message has not been modified in transit (integrity).

In this subsection, we will briefly cover the fundamentals of cryptographic signature schemes. For the sake of brevity I have assumed some concepts to be known, for example the security parameter, probabilistic polynomial time algorithms (PPT), etc. For more details and deeper definitions, I would like to direct you to [MF21], or alternatively one of [GB08; KL20].

2.3.1 Security fundamentals

Conceptually, all asymmetric cryptographic constructions are based on some "hard" problem, i.e., problems that cannot be solved in polynomial time.[1] For example, RSA's [RSA78] key-only security is based on the hardness of integer factoring, and elliptic-curve cryptography [Kob87; Mil86] is based on some variant of the elliptic-curve discrete-logarithm problem. We construct our cryptographic algorithms in such a way, that if the algorithm can be broken in probabilistic polynomial time (PPT), then that "breaking algorithm" can be used to solve instances of the underlying problem in probabilistic polynomial time. The reasoning is that we assume that no efficient algorithms can exist that solve the hard problem, and therefore no algorithms can exist that break the scheme. In cryptography, these hypothetical attacking algorithms are called *adversaries* (or *attackers*).

The central instrument that is used to formalize these scenarios, i.e., the breaking of the scheme or the solving of the hard problem, is the *game* (or *experiment*). A game is a mathematical thought experiment that concretely describes a cryptographic problem statement. In the experiment, the *challenger* samples an instance of the problem and challenges the adversary to find a solution to that instance. Afterward, the challenger will check whether the solution that was proposed by the adversary was correct. If the adversary is able to find a correct solution to the instance of the problem, then the adversary *wins* the game.

[1] In practice, this means that the time to solve these problems increases (sub)exponentially as the size of the inputs increases. As long as we make the inputs large enough, (e.g., we try to find really big prime factors), then it becomes practically impossible to solve the problem.

The advantage. For cryptographic schemes, we are interested in quantifying the ability of an adversary to win that game. We call this probability the *advantage* of the adversary. We write $\mathsf{Adv}^{\mathcal{G}}(\mathcal{A})$ to denote the advantage of the adversary $\mathcal{A}$ for some security game $\mathcal{G}_{\mathcal{A}}$. The goal of all security proofs is to show that, for all possible adversaries, the advantage is negligible under some common assumptions. A bound ϵ is *negligible* if $\epsilon(\lambda) \leq \frac{1}{p(\lambda)}$, for all possible polynomials $p(\lambda)$, and a sufficiently large λ.

The security parameter. If the advantage of $\mathcal{A}$ is negligible for some cryptosystem, then λ describes a lower bound for the security of said cryptosystem. This λ, which is a positive number, is called the *security parameter*. As it increases linearly, the advantage of $\mathcal{A}$ is expected to decrease (sub-)exponentially. The security parameter is provided as an argument to security games, adversaries, and primitives (i.e., $\mathcal{G}_{\mathcal{A}}(\lambda)$, $\mathcal{A}(\lambda)$, etc.), but for the sake of clarity it will be omitted.

Game-hopping proofs. The main strategy for arguing for the security of a scheme is by use of a *game-hopping proof*. First, we *assume* that there exists a game (let's call this game $\mathcal{G}^0$) wherein the adversary $\mathcal{A}$ breaks the scheme. Then we make a small tweak to the game, resulting in $\mathcal{G}^1$. We keep doing this until we end up with the game in which the adversary solves the cryptographic problem that was presumed to be hard.

After each game hop from $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$, the original adversary $\mathcal{A}$ might not completely apply to $\mathcal{G}^{i+1}$. $\mathcal{A}$ might now lose for a fraction of instances of $\mathcal{G}^{i+1}$ where it would have won in the case of $\mathcal{G}^i$. These exception cases, where $\mathcal{A}$ wins a $\mathcal{G}^i$ instance, but where $\mathcal{A}$ loses the same $\mathcal{G}^{i+1}$ instance, are called *bad events*. We bound the prevalence of the bad events by bounding the *security loss* $\epsilon_i = \left|\Pr\left[\mathcal{G}^i_{\mathcal{A}} = \mathsf{true}\right] - \Pr\left[\mathcal{G}^{i+1}_{\mathcal{A}} = \mathsf{true}\right]\right|$.

Call the last game (i.e., the "hard problem game") $\mathcal{G}_k$, and let us denote its adversary (i.e., the adversary that solves the hard problem) $\mathcal{B}$. We can now compute the sum of all ϵ_is to get a relation between the advantages of both adversaries:

$$\mathsf{Adv}(\mathcal{A}) \leq \mathsf{Adv}(\mathcal{B}) + \epsilon_0 + \cdots + \epsilon_{k-1} \tag{2.1}$$

If the total security loss is negligible, this proof shows a useful *security reduction* between $\mathcal{G}^0$ and $\mathcal{G}^k$. Informally, it states that algorithm $\mathcal{A}$'s ability to break the cryptographic scheme $\mathcal{G}^0$ is always less than or equal to the ability of the algorithm

$\mathscr{B}$ to break the cryptographic hard problem $\mathscr{G}^k$, plus some negligible bound. As we assume that no PPT algorithm $\mathscr{B}$ can ever exist, we also know that no PPT algorithm $\mathscr{A}$ will ever exist.

Hybrid proofs. We do not know anything about the internal mechanics of $\mathscr{A}$; as it is a hypothetical black box. However, we can look at the inputs that the adversary is provided. In a *hybrid proof*, we bound the statistical distance $\Delta(X;Y)$ of the inputs to the adversary between two games. Then we use the property that

$$\Delta\left(\mathscr{A}(X^i);\mathscr{A}(X^{i+1})\right) \leq \Delta\left(X^i;X^{i+1}\right) \tag{2.2}$$

which holds for any function $\mathscr{A}$. Consequently, we have an upper bound for $\Delta\left(\mathscr{G}^i;\mathscr{G}^{i+1}\right)$, which is equal to $\left|\Pr\left[\mathscr{G}^i_{\mathscr{A}} = \mathsf{true}\right] - \Pr\left[\mathscr{G}^{i+1}_{\mathscr{A}} = \mathsf{true}\right]\right| = \epsilon_i$.

2.3.2 Signature schemes in theory

Definition 2.1 (Signature scheme). *A* signature scheme *is a tuple of three efficient algorithms* $\mathsf{Sig} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *following the format*

$$\begin{aligned} \mathsf{sk}, \mathsf{pk} &\leftarrow \mathsf{KeyGen}(1^\lambda) \\ \sigma &\leftarrow \mathsf{Sign}(\mathsf{sk}, M) \\ \mathsf{ok} &\leftarrow \mathsf{Verify}(\mathsf{pk}, M, \sigma) \end{aligned}$$

where the individual algorithms are described as follows.

Key generation (KeyGen**).** *The* probabilistic *key generation algorithm* KeyGen*, that takes the security parameter* 1^λ*; outputs a* keypair*, i.e., a pair consisting of a secret key (*sk*) and a public key (*pk*).*

Signature generation (Sign**).** *The signature generation algorithm* Sign*, which may be probabilistic, takes a secret key* sk *and a* message *M as input; outputs a* signature σ*.*

Verification (Verify**).** *The deterministic signature verification algorithm* Verify*, which takes a public key* pk*, a message M, and a signature* σ *as input; outputs* true *if the*

signature correctly authenticates that it was generated over M using the secret key corresponding to pk*; and outputs* false *otherwise.*

Definition 2.2 (Correctness). *A signature scheme is* correct *if* all *valid signatures generated by the* Sign *algorithm can be verified to be valid using the* Verify *algorithm, i.e., for all λ, M:* $\Pr[\mathsf{Verify}(\mathsf{pk}, M, \mathsf{Sign}(\mathsf{sk}, M)) = \mathsf{true} \mid (\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)] = 1$.

2.3.3 Security notions

Definition 2.3 (Unforgeability under no-message Attacks (UF-NMA)). *A signature scheme* Sig *is unforgeable under no message attacks if for all* PPT *adversaries $\mathscr{A}$ the advantage* $\mathsf{Adv}^{\mathsf{UF\text{-}NMA}}(\mathscr{A})$ *is negligible, with the game* $\mathscr{G}^{\mathsf{UF\text{-}NMA}}_{\mathsf{Sig},\mathscr{A}}(\lambda)$ *defined as*

Game $\mathscr{G}^{\mathsf{UF\text{-}NMA}}_{\mathsf{Sig},\mathscr{A}}(\lambda)$*:*

1: $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
2: $(M^*, \sigma^*) \xleftarrow{\$} \mathscr{A}(\mathsf{pk})$
3: ***return*** $(\mathsf{Verify}(\mathsf{pk}, M^*, \sigma^*) = \mathsf{true})$

In all *unforgeability* models, the adversary wins when it is able to forge a signature over the public key that was provided. In UF-NMA, the adversary is only provided the public key of the scheme. That is, the adversary *only gets access to the public key*, which is why it is also generally known as *unforgeability under key-only attack* (UF-KO).

Definition 2.4 (Unforgeability under chosen-message attacks (UF-CMA)). *A signature scheme* Sig *is unforgeable under chosen message attacks if for all* PPT *adversaries $\mathscr{A}$ the advantage* $\mathsf{Adv}^{\mathsf{UF\text{-}CMA}}(\mathscr{A})$ *is negligible, with the game* $\mathscr{G}^{\mathsf{UF\text{-}CMA}}_{\mathsf{Sig},\mathscr{A}}(\lambda)$ *defined as*

Game $\mathscr{G}^{\mathsf{UF\text{-}CMA}}_{\mathsf{Sig},\mathscr{A}}(\lambda)$*:*

1: $Q := \{\}$
2: $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
3: $(M^*, \sigma^*) \xleftarrow{\$} \mathscr{A}_{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$
4: ***return*** $(\mathsf{Verify}(\mathsf{pk}, M^*, \sigma^*) = \mathsf{true}$ *and* $M^* \notin Q)$

with a signing oracle $\mathcal{O}(\mathsf{sk}, M)$ *that is defined as*

Signing oracle $\mathcal{O}(\mathsf{sk}, M)$*:*

1: $Q := Q \cup \{M\}$
2: $\sigma \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, M)$
3: ***return*** σ

Definition 2.4 defines the UF-CMA security model. In this security model, we assume that the attacker has access to a *signing oracle* $\mathcal{O}$. When the attacker queries the oracle with a message, the oracle will give back a valid signature for that message. The attacker can keep adaptively querying (a possibly large number of) signatures from the oracle as part of their attack. The adversary wins (and the scheme is considered broken) when the adversary can forge a signature with a non-negligible probability, for any message, as long as they did not ask the oracle to sign that message as part of the attack. This ensures that the adversary is not allowed to just query the oracle for M^* and submit the resulting signature to the challenger; the attacker has to produce a signature over a *new* message.

Definition 2.5 (Strong unforgeability under chosen-message attacks (SUF-CMA**)).** *A signature scheme* Sig *is strong unforgeable under chosen message attacks if for all* PPT *adversaries* $\mathcal{A}$ *the advantage* $\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}(\mathcal{A})$ *is negligible, with the game* $\mathcal{G}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{A}}(\lambda)$ *defined as*

Game $\mathcal{G}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{A}}(\lambda)$*:*

1: $Q := \{\}$
2: $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
3: $(M^*, \sigma^*) \xleftarrow{\$} \mathcal{A}_{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$
4: ***return*** $(\mathsf{Verify}(\mathsf{pk}, M^*, \sigma^*) = \mathsf{true}$ *and* $(M^*, \sigma^*) \notin Q)$

with a signing oracle $\mathcal{O}(\mathsf{sk}, M)$ *that is defined as*

Signing oracle $\mathcal{O}(\mathsf{sk}, M)$*:*

1: $\sigma \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, M)$
2: $Q := Q \cup \{(M, \sigma)\}$
3: ***return*** σ

In the UF-CMA security model, the adversary wins when they can forge a signature for any message that they did not input into the oracle as part of the attack. The SUF-CMA is a stronger security model, which only requires that the adversary cannot submit a *signature* that they received from the oracle [ADR02].

For example, consider the scenario in which the generated signatures are *malleable*: When signatures are malleable, the adversary can query $\sigma \leftarrow \mathcal{O}(M^*)$, and then use σ to find some σ^* which is also valid over M^*. In UF-CMA, the adversary does not win the game, because M^* was input into the signing oracle at some point. However, in the SUF-CMA game, the adversary does win, because only σ was output from the signing oracle before; σ^* is different from σ, so submitting that to the challenger *does* result in a win. If a signature scheme is SUF-CMA secure, then it is also UF-CMA secure.

2.3.4 Signature schemes in practice

Section 2.3.2 describes how signatures are commonly viewed in cryptography. That description is built for efficient reasoning about their security. However, from the perspective of the engineer, it is sometimes more convenient to not reduce the primitive to such an ideal form, as it omits many details that become relevant once you start thinking about real-world aspects.

In practice, there is often a lot of "artistic freedom" and there are many tradeoffs to be considered. After all, a crypto implementation can do anything as long as its outputs correspond to the specification, and as long as it protects against side-channels (see Section 2.6.1). This also applies to the signature scheme's API.

For example, how do we provide the random number generator? After all, *real* randomness does not exist out of nothing; the randomness has to come from somewhere. We could provide the scheme's functions with random seeds, or we could supply an RNG function as one of the inputs. Another scenario in which the common API does not fit is when we are signing or verifying a batch of many signatures in one go. Then we will have to come up with an API that supports *batched* signing/verification. Or, what if some of the inputs or outputs do not entirely fit into memory? In this case we might want to design some kind of API where the input values are streamed in (like in [HRS16] or [GHKK+21]); or we can delegate the compressing of M to a more powerful environment (i.e., implement online/offline signing [EGM96]).

In crypto engineering, it is not only important to know the algorithm, but it is also important to consider the environment (platform, use case, etc.) in which it will be deployed; to utilize its strengths and know its weaknesses. Of all descriptions of signatures schemes, we can then apply the one that suits the environment most.

2.4 Modular integer multiplication

Almost all asymmetric cryptographic schemes are in some way based on arithmetic in $\mathbb{Z}_q$. In the ideal case, q would be a power of 2, because then the modular arithmetic would be trivial to implement on all modern CPUs.

Unfortunately, the moduli used in many cryptographic schemes are not powers of two; and, as such, we cannot use the CPU's native integer multiplier *as is* for modular multiplication. We overcome this complication by using *modular-multiplication* algorithms.

These algorithms take some bounded inputs a, b and a modulus q. They compute $c \equiv a \cdot b \pmod{q}$, such that the range of c is concretely bounded as well.

Modular-multiplication algorithms usually consist of two stages: first, a standard multiplication operation is performed, and second, the result is reduced modulo q. These two steps can be seen as separate operations, and the reduction step can be used on its own for reducing values that have accumulated through operations other than multiplication. However, for most of the implementations in this thesis, every multiplication is followed by a modular reduction step. Therefore in this section, the multiplication step and the reduction step will be presented as two parts of the same algorithm.

Unsigned vs. signed representation. Over the wire and in storage, cryptographic values (public keys, signatures, etc.) are *encoded* into opaque byte strings. In these encodings, cryptographic values are represented in their *standard representation*, which usually use *unsigned* values.

However, internally in cryptographic implementations, we are free to use any kind of representation that fits our needs. Indeed, it is sometimes more efficient to opt for using a *signed* representation instead of an unsigned representation, because unsigned representations are less compatible with subtractions. Every time we compute a subtraction $c \leftarrow a_0 - a_1$ on unsigned values, we have to ensure that a_0 is larger

than or equal to a_1 to prevent an integer overflow from happening. Often the most efficient way to do this is to add a multiple of q to a_0 before subtracting, such that the left operand is guaranteed to be greater than a_1 before the subtraction occurs, i.e., $c \leftarrow (a_0 + kq) - a_1$. Apart from adding an extra addition to the code, this also widens the bounds of the result, which leads to the need for more modular reductions modulo q. In signed operations, this overflow does not occur, and as such, no additional code is needed to prevent them.

Dilithium's q is a Solinas prime [Sol11]. Although its structure has been used to build specialized reductions in hardware ([ZZWY+21]), in software we usually fall back to using general modular reduction algorithms. For this thesis, the two main general modular multiplication methods are Barrett reduction [Bar87] and Montgomery multiplication [Mon85]. Although these modular reduction methods were initially published as algorithms for unsigned values, they have since been adapted by others for use with signed values. Since all the implementations presented in this thesis utilize a signed representation for their internal values, only the signed versions of the modular-multiplication algorithms will be listed.

2.4.1 Barrett reduction

A concrete instantiation of the Barrett reduction algorithm is listed in Algorithm 2.6. Barrett reduction is based on the following strategy:

0. (compute the unreduced multiplication of the operands a and b);
1. approximate $t \leftarrow \left\lfloor \frac{ab}{q} \right\rceil$;
2. output $c \leftarrow ab - tq$.

With this strategy, the output bounds are determined by the quality of the approximation: the lower the approximation error, the tighter the bounds of c. The algorithm listed in Algorithm 2.6 uses the approximation where $t \leftarrow \left\lfloor ab \cdot (q^{-1} \bmod^{\pm} 2^n) \right\rceil^n$, which only needs a single multiplication and an arithmetic shift to the right.

Correctness. Reduction methods are *correct* when, provided correct inputs, they always produce correct outputs. In other words, does the reduction algorithm output $c \equiv ab \pmod q$ for all valid pairs a, b? For Barrett reduction, this is easy to see: tq is a multiple of q, and as such, $c = ab - tq \equiv ab \pmod q$. □

Algorithm 2.6: Signed multiplication with Barrett reduction of $c \leftarrow a \cdot b \pmod{q}$

input: a, b, q, n with $q, 2^n$ pairwise coprime, $q < 2^n$ and $-2^{n-1}q \leq ab < 2^{n-1}q$
output: $c \equiv ab \pmod{q}$ and $-q < c < q$
let: $R = \left\lfloor \frac{2^n}{q} \right\rceil$

1: $T_1 \leftarrow a \cdot b$ ▷ n-bit multiply
2: $t \leftarrow [R \cdot T_1]^n$ ▷ $2n$-bit multiply
3: $T_2 \leftarrow t \cdot q$ ▷ n-bit multiply
4: $c \leftarrow T_1 - T_2$
5: **return** c

Usefulness. Aside from being correct, a reduction algorithm also has to be *useful*. For the algorithm to be useful, we desire for the bounds of its outputs to be reasonably tight. We can determine the algorithm's usefulness by examining the bounds of c.

The bounds of ab are $\left[-2^{n-1}q, 2^{n-1}q\right)$, as required by the algorithm specification. To compute the output bounds, we fill in the assumed bounds for ab and then we simplify from there:

$$
\begin{aligned}
-2^{n-1}q - [-2^{n-1}qR]^n q \leq c &< 2^{n-1}q - [2^{n-1}qR]^n q \\
-q\left(2^{n-1} + [-2^{n-1}qR]^n\right) \leq c &< q\left(2^{n-1} + [-2^{n-1}qR]^n\right) \\
-q\left(2^{n-1} + \left[-2^{n-1}q\left(\frac{2^n}{q} - \frac{1}{2}\right)\right]^n\right) \leq c &< q\left(2^{n-1} + \left[-2^{n-1}q\left(\frac{2^n}{q} - \frac{1}{2}\right)\right]^n\right) \\
-q\left(2^{n-1} + \left[-2^{n-1}\left(2^n - \frac{q}{2}\right)\right]^n\right) \leq c &< q\left(2^{n-1} + \left[-2^{n-1}\left(2^n - \frac{q}{2}\right)\right]^n\right) \\
-q\left(\cancel{2^{n-1}} - \cancel{2^{n-1}} + \left[\frac{q}{2}\right]^n\right) \leq c &< q\left(\cancel{2^{n-1}} - \cancel{2^{n-1}} + \left[\frac{q}{2}\right]^n\right) \\
-q\left[\frac{q}{2}\right]^n \leq c &< q\left[\frac{q}{2}\right]^n \\
-q \leq c &< q
\end{aligned}
$$

This shows that the outputs of the Barrett reduction algorithm are always between $-q$ and q, which makes the reduction algorithm useful. □

Cost. Barrett reduction uses three multiply operations. Two of the multiply operations are on a single word of n bits, which is usually a cheap operation on most platforms. However, one of these operations (Algorithm 2.6, line 2) is a $2n$-bit multiplication, which might be considerably more expensive.

2.4.2 Montgomery multiplication

Another modular multiplication method is *Montgomery multiplication*, invented by Montgomery in 1985 [Mon85]. Originally it was proposed as an algorithm for *unsigned* integer multiplication. In 2018, Seiler proposed the *signed* variant [Sei18] that is listed in Algorithm 2.7.

Algorithm 2.7: Signed Montgomery modular multiplication of $c \leftarrow a \cdot b \pmod{q}$

input: a, b, q, n with $q, 2^n$ pairwise coprime, and $-2^{n-1}q \leq ab < 2^{n-1}q$
output: $c \equiv 2^{-n}ab \pmod{q}$ and $-q < c < q$
let: $R = q^{-1} \bmod {}^{\pm}2^n$

1: $T_1 \leftarrow a \cdot b$ ▷ n-bit multiply
2: $t \leftarrow \left[[T_1]_n \cdot R\right]_n$ ▷ n-bit multiply
3: $T_2 \leftarrow t \cdot q$ ▷ n-bit multiply
4: $c \leftarrow [T_1 - T_2]^n$
5: **return** c

Montgomery reduction takes a different perspective from Barrett reduction, where instead of computing $c \equiv ab$ we compute $c \equiv 2^{-n}ab$. At first thought, you would think that this eliminates the whole purpose of the reduction method, as the final result is off by a factor 2^{-n}. However, this does not matter much, because the 2^{-n} factor is easily eliminated by Montgomery-multiplying the result with $2^{2n} \pmod{q}$, resulting in $(2^{-n}2^{2n})(2^{-n}ab) \equiv ab$. On the other side, this extra factor 2^{-n} adds a lot of freedom which can be used to construct a more efficient reduction algorithm.

Instead of approximating $t \leftarrow \lfloor ab/q \rceil$, the Montgomery reduction algorithm approximates a small factor t such that $ab - tq$ divides 2^n. Then the result is computed as $c \leftarrow (ab - tq)/2^n$, which is where the factor 2^{-n} appears. The formula for t is described by

$$t = [[ab]_n \cdot R]_n$$

where $R \equiv q^{-1} \pmod{2^n}$.

Correctness. For the algorithm to be correct it must hold that $tq \equiv 0 \pmod{q}$ (trivial), and it must be that $ab - tq$ is divisible by 2^n, because otherwise the division would be undefined (or it would add rounding error, depending on your definitions). Fortunately, this follows directly from the value we chose for R, as R was chosen such that $R \cdot q \equiv 1 \pmod{2^n}$:

$$\begin{aligned} ab - tq &= ab - [[ab]_n \cdot R]_n q \\ &= ab - [[ab]_n \cdot Rq]_n \\ &= ab - [[ab]_n \cdot (2^n + 1)]_n \\ &= ab - [ab]_n \text{ which is divisible by } 2^n. \end{aligned}$$

□

Usefulness. The bounds of ab are $\left[-2^{n-1}q, 2^{n-1}q\right)$, as is specified as a requirement of the algorithm inputs. The bounds of tq are also $\left[-2^{n-1}q, 2^{n-1}q\right)$, because t is bounded by -2^{n-1} and 2^{n-1} through the $[\_]_n$ bit selection operation. Therefore, after the last division by 2^n, c is bounded by:

$$\begin{gathered} -\frac{\overbrace{2^{n-1}q}^{ab\text{ bound}} + \overbrace{2^{n-1}q}^{tq\text{ bound}}}{2^n} \leq c < \frac{\overbrace{2^{n-1}q}^{ab\text{ bound}} + \overbrace{2^{n-1}q}^{tq\text{ bound}}}{2^n} \\ -\frac{q+q}{2} \leq c < \frac{q+q}{2} \\ -q \leq c < q \end{gathered}$$

Just like the outputs of the Barrett reduction algorithm, the Montgomery reduction algorithm's outputs are always between $-q$ and q. □

Cost. From both algorithm listings, you can see that, in terms of operation count, the primary difference between Barrett and Montgomery modular multiplication is that Barrett reduction uses one $2n$-bit multiplication aside from two n-bit multiplications, whereas Montgomery only uses three n-bit multiplications. The actual difference in

performance between both algorithms depends on whether the platform provides native $2n$-bit multiplications. If that is the case, then both algorithms are roughly equally fast. However when the platform only supports n-bit multiplications, then the $2n$-bit multiplication will have to be implemented using four n-bit multiplications. In that case, Montgomery reduction is undeniably faster.

2.4.3 Montgomery multiplication with precomputed constants

Often when we are computing modular multiplications, one of the operands is known at compile-time, i.e., it is a *constant*. In this case, we can erase the extra 2^{-n} factor by multiplying the precomputed operand with 2^n before using it as an input to the multiplication algorithm. When we do this we do not have to recover c from the Montgomery domain, because $c \equiv 2^{-n}(2^n ab) \equiv ab$. This optimized version of the Montgomery multiplication algorithm is listed in Algorithm 2.8.

Algorithm 2.8: Signed Montgomery modular multiplication of $c \leftarrow a \cdot b \pmod{q}$ where a is a constant

input: a, b, q, n with $-2^{n-1}q \leq ab < 2^{n-1}q$
output: $c \equiv ab \pmod{q}$ and $-q < c < q$
let: $R = q^{-1} \bmod^{\pm} 2^n$ and $a' = 2^n a \bmod^{\pm} q$

1: $t \leftarrow [a' \cdot b]^n$ ▷ n-bit multiply
2: $T_2 \leftarrow t \cdot q$ ▷ n-bit multiply
3: $c \leftarrow [T_1 - T_2]^n$
4: **return** c

2.5 Cortex-M3 and Cortex-M4

The implementation work in this thesis focuses heavily on the Arm Cortex-M3 [ARM10a; ARM10b] and Arm Cortex-M4 [ARM11; ARM20] microarchitectures. The variety of embedded computing architectures is enormous, but implementation characteristics are hard to compare across different kinds of architectures, as each has their own particular strengths and weaknesses. In order to facilitate apples-to-apples comparisons of post-quantum crypto implementations, NIST requested the evaluation of schemes (see Section 1.2) to be narrowed down to as few platforms as possible.

They chose Cortex-M4, which is a reasonable platform to stick to because it is often included in chips that are considered constrained, but not too constrained as to immediately disqualify all of the "bigger" candidates.[2] However, in the embedded industry, Cortex-M4 is often still considered a relatively powerful architecture. Therefore, we also consider Cortex-M3, a somewhat smaller (and cheaper) alternative to Cortex-M4.

2.5.1 The Armv7E-M Thumb architecture

Arm Cortex-M4 implements the Armv7E-M Thumb instruction set architecture (ISA), and Cortex-M3 implements the Armv7-M Thumb ISA. The ISAs are very similar, with the Armv7E-M architecture being slightly more powerful than the Armv7-M architecture. Both ISAs are in-order, and feature a 3-stage (fetch-decode-execute) pipeline. Each ISA features 16 32-bit registers (`r0`–`r15`), of which 2 are reserved for the program counter (`r15`) and the stack pointer (`r13`), leaving 14 usable general-purpose registers.

32-bit to 64-bit multiplication. Multiplications are the core operation of cryptographic implementations. Aside from instructions for 32-bit to 32-bit multiplication (`mul`, `mla`, `mls`), both the ISAs support instructions for native 32-bit to 64-bit multiplications (`umull`, `smull`, `umlal`, `smlal`). On Cortex-M4, all of these instructions execute in a single cycle, making them very suitable for crypto implementations. However on Cortex-M3, these "big" instructions have an execution time that is dependent on the instruction operands [Gro15], varying from 3 to 7 cycles per instruction. This makes them unusable for computing on secret data because that kind of use would be vulnerable to timing side-channel attacks [GOPT09] (see Section 2.6.1). Therefore on Cortex-M3, we can only use these instructions when working with public values.

SIMD instructions. On top of the powerful 32-bit to 64-bit multiplication instructions, the ARMv7E-M ISA provides SIMD instructions like `smlad` or `uadd16`. These have been shown to achieve significant speedups for NTT-based polynomial multiplication [ABCG20; BKS19] and Toom–Cook-based polynomial multiplication [BKV20; KBSV18; KRS19] on the Cortex-M4.

[2] This happened in the email conversation with the subject "On Recommended Hardware" on the pqc-forum mailing list of 5–6 February 2019.

Inline shifts. A neat feature that the Armv7 architectures implement is that Thumb data-processing instructions can inline shift or rotate their second operand before use. This feature is often referred to as the *barrel shifter*.[3] For example, the instruction `add r0, r0, r1, lsl #2` shifts the contents of `r1` two positions to the left, before adding them to `r0`. The shift operation adds no overhead; all instructions use the same amount of cycles whether the shifter was used or not.

Floating-point unit. The Cortex-M4 architecture features an optional floating-point unit (FPU). The presence of an FPU is often denoted by a suffix F in the architecture name (e.g., Cortex-M4F). Even if our code does not use any floating-point operations, the FPU can still be useful, because it provides 32 additional registers (`s0–s31`). One cannot use these registers directly for general-purpose data processing. However, moving between general-purpose registers and FPU registers is faster than accessing memory (1 cycle instead of 2 cycles). This is why, in Chapter 5, we use the FPU registers to store some of our local variables.

2.5.2 STM32F4 Discovery

As mentioned, the Cortex-M3 and Cortex-M4 architectures are implemented on a plethora of chips and development boards. At the start of the NIST competition, the *pqm4* [PQM4] project ported Cortex-M4 implementations of all the NIST competition candidates to the *STM32F4 Discovery board*. Because of this, the STM32F4 Discovery board [STM20b] has evolved to be the de facto development board for the evaluation of post-quantum crypto schemes on Cortex-M4. As such, it is also the main board used for the evaluations in this thesis. The board features an STM32F407VGT6 microcontroller [STM20a], which has 1 MB of flash space, and 192 KiB of SRAM (of which 64 KiB is faster "core-coupled" memory). Its core runs with a maximum clock speed of 168 MHz. The chip includes the optional Cortex-M4 FPU and includes a hardware true random number generator (TRNG). It does not feature any kind of acceleration for cryptographic primitives.

[3] It is, presumably, implemented using a barrel shifter.

2.5.3 Arduino Due

Aside from Cortex-M4 evaluations, we also evaluate Dilithium's performance on Cortex-M3. For Cortex-M3 we use the *Arduino Due* development board. This board is based on the ATSAM3X8E IC [Atmel15] which contains a Cortex-M3 core. The chip comes with 512 KiB of flash space, and 96 KiB of homogeneous SRAM; and its core's maximum clock frequency is 84 MHz. Just like the STM32F4 chips, it does not feature any acceleration for cryptographic primitives. It does, however, also feature a hardware true random number generator.

2.6 Software & measurements

2.6.1 Side-channel resistance

Our implementations are only considering timing side-channels [Koc96], i.e., we provide *constant-time* code that avoids leaking secret data through the execution time of operations on the platform. Formally, constant-timeness is based on the notion of *computational probabilistic non-interference* [BP02]. In practice, whether code is constant-time is highly dependent on both the platform and the leakage model. In the case of this thesis, all code that is described as constant-time is implemented according to the following rules:

1. All values in the algorithm's execution are public or secret.
2. For all atomic operations, the output is secret if *any* of the inputs is secret, and public if *all* of the inputs are public.
3. Values that are secret are not used as conditions in conditional branching operations.
4. Values that are secret are not used as address operands for operations that interact with memory.
5. Values that are secret are not used as operands for operations from which it is known that their execution time depends on that operand.
6. The Declassify(s) operation takes a secret variable s and designates it as public.

7. Variables are only declassified when they are of no use to an attacker. In other words, for all $\mathscr{B}$ adversaries that get *all* public values as input, and $\mathscr{A}$ is the adversary that breaks the scheme (i.e., that of Definition 2.3, 2.4, or 2.5), values are only declassified when $\mathsf{Adv}(\mathscr{B}) \leq \mathsf{Adv}(\mathscr{A})$ remains satisfied.

For certain use cases one may want to consider to also protect against more powerful attacks like power analysis attacks, e.g., using masking. There exists work in the literature that masks modified versions of Dilithium [MGTF19; PPRS23], and there are (at the time of writing) only few masked Dilithium implementation that conform to the Dilithium specification [ABCH+23; CGTZ23]. More research is still needed to determine the best ways of masking Dilithium in implementations. However, that topic is researched in parallel to our research, and masking techniques have been left outside of the scope of this thesis.

2.6.2 Benchmarks

Cortex-M4. Our Cortex-M4 benchmarking setup is based on pqm4 [PQM4]. As such, we benchmark all our Cortex-M4 implementations on the STM32F407 Discovery board. It was clocked at 24 MHz to eliminate flash wait states when fetching instructions or data from flash. For benchmarking the algorithm latency, we used the SysTick counter clocked from the same clock as the core.

Cortex-M3. The Cortex-M3 speed measurements were done on the Arduino Due board which uses the ATSAM3X8E microcontroller. The ATSAM chip was clocked at 16 MHz, which results in a flash access time with zero wait-states. The algorithm latencies were measured using the internal cycle counter (`DWT->CYCCNT`).

Memory usage. On both architectures, the memory usage was measured by filling the stack memory with dummy values, then executing the algorithm, and afterward measuring the amount of dummy-value bytes that were overwritten during the execution (no static or heap memory was used). By convention, in the memory measurements, space reserved for input and output values (i.e., buffers for keys, messages, and signatures) is not counted.

2

3 Dilithium

3.1 Lattice-based cryptography

As we saw in Section 2.3.1, all cryptographic constructions are based on some hard mathematical problem. The Dilithium signature scheme is based on the MLWE and MSIS hard problems in lattice cryptography. Both problems are "hard", i.e., we presume that the advantage of an adversary solving one of these problems is very small. Their definitions are provided by the Dilithium specification [DKLL+20], but for completeness I will provide them here as well.

Definition 3.1 ($\mathsf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}$). *For integers k and ℓ, and a probability distribution $D : R_q \to [0,1]$, the advantage of an adversary $\mathscr{A}$ of solving the* MLWE *problem over R_q is*

$$\mathsf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathscr{A}) := \Big|\Pr\Big[b = \mathsf{true} \mid \mathbf{A} \xleftarrow{\$} R_q^{k\times\ell}; \mathbf{t} \xleftarrow{\$} R_q^k; b \leftarrow \mathscr{A}(\mathbf{A},\mathbf{t})\Big] - \Pr\Big[b = \mathsf{true} \mid \mathbf{A} \xleftarrow{\$} R_q^{k\times\ell}; \mathbf{s_1} \leftarrow D^\ell; \mathbf{s_2} \leftarrow D^k; b \leftarrow \mathscr{A}(\mathbf{A}, \mathbf{As_1} + \mathbf{s_2})\Big]\Big|. \tag{3.1}$$

Definition 3.1 describes the *Decisional Module Learning With Errors* (MLWE) problem, based on the LWE problem proposed by Regev in 2005 [Reg05]. It declares that if some adversary $\mathscr{A}$ exists which is able to distinguish between MLWE pairs $(\mathbf{A}, \mathbf{t})$ of the form

$$\mathbf{A} \xleftarrow{\$} R_q^{k\times\ell}, \mathbf{t} \leftarrow \mathbf{As_1} + \mathbf{s_2},$$

with $\mathbf{s_1}$ and $\mathbf{s_2}$ sampled from the distribution D; and *random* pairs $(\mathbf{A}, \mathbf{t})$ of the form

$$\mathbf{A} \xleftarrow{\$} R_q^{k\times\ell}, \mathbf{t} \xleftarrow{\$} R_q^k,$$

then its advantage is equal to $\mathsf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathscr{A})$.

Informally speaking, we assume that both kinds of pairs "look the same". From just $\mathbf{A}$ and $\mathbf{t}$ alone, we presume no adversary can make out which $\mathbf{t}$s were generated using the $\mathbf{t} \leftarrow \mathbf{As_1} + \mathbf{s_2}$ formula, and which ones are *really* random. If (contrary to our beliefs), there exists some $\mathscr{A}$ that *can* solve the problem, then its advantage is described by $\mathsf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathscr{A})$.

Definition 3.2 ($\mathsf{Adv}^{\mathsf{MSIS}}_{k,\ell,\gamma}$). *The advantage of an adversary $\mathscr{A}$ of solving the* MSIS *problem over R_q is*

$$\mathsf{Adv}^{\mathsf{MSIS}}_{k,\ell,\gamma}(\mathscr{A}) := \Pr\left[\left[\,\mathbf{I} \mid \mathbf{A}\,\right] \cdot \mathbf{y} = \mathbf{0} \text{ and } 0 < \|\mathbf{y}\|_\infty \leq \gamma \mid \mathbf{A} \xleftarrow{\$} R_q^{k\times\ell}; \mathbf{y} \leftarrow \mathscr{A}(\mathbf{A})\right]. \quad (3.2)$$

Definition 3.2 describes the *Module Shortest Integer Solution* (MSIS) problem, based on the SIS problem from the Ajtai paper of 1996 [Ajt96]. It defines the advantage of an adversary $\mathscr{A}$ that can find some non-zero vector $\mathbf{y}$ which

- satisfies $\left[\,\mathbf{I} \mid \mathbf{A}\,\right] \cdot \mathbf{y} = \mathbf{0}$; and
- is *small*, i.e., $\|\mathbf{y}\|_\infty \leq \gamma$.

The Dilithium specification also describes the SelfTargetMSIS problem. SelfTargetMSIS is particularly relevant for security analysis of Dilithium in the quantum random oracle model (QROM) [BDFL+11]. However, it is not relevant in the classical setting, because, in the regular random oracle model (ROM) [BR93], SelfTargetMSIS can be reduced to MSIS. We will not cover the QROM security of Dilithium, and as such the SelfTargetMSIS problem is outside the scope of this thesis.

3.2 Dilithium simplified

The MLWE problem can be used to construct cryptographic lattice signatures [BG14; GPV08; Lyu12]. We will provide some intuition on how the confidentiality of the secret key and signature soundness follows from MLWE and MSIS. Note that this section does not attempt to *prove* the security and soundness of Dilithium. Multiple papers have been dedicated to the security analysis of the scheme (e.g., [BBDD+23; DFPS23; KLS18; LS15; Lyu09; Lyu12]), and a complete separate book could be written

about it. However, I will do my best to provide a high-level overview of the scheme's security, and to refer to the relevant literature when applicable.

Before diving into the full Dilithium scheme, let us first look at the simplified version that I have listed in Algorithm 3.3. It follows all of the same principles, but it leaves out the details that are irrelevant for security.

In this section, all parameters ($q, \eta, \gamma_1, \gamma_2, \beta$, etc.) will be left undefined. I recognize that this leads to a somewhat abstract or even nebulous description of the scheme. However, their definitions are not yet needed at this point. Just know that they are scalar constants, and their concrete values will be listed in Section 3.3.3.

Algorithm 3.3: Simplified version of Dilithium.

function KeyGen
  $\mathbf{s_1} \leftarrow S_\eta^\ell$
  $\mathbf{s_2} \leftarrow S_\eta^k$
  $\mathbf{A} \xleftarrow{\$} R_q^{k \times \ell}$
  $\mathbf{t} := \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
  return $\mathsf{sk} := (\mathbf{A}, \mathbf{s_1}, \mathbf{s_2}), \mathsf{pk} := (\mathbf{A}, \mathbf{t})$

function $\mathsf{Sign}((\mathbf{A}, \mathbf{s_1}, \mathbf{s_2}) := \mathsf{sk}, M)$
  $\mathbf{y} \xleftarrow{\$} S_{\gamma_1 - 1}^\ell$
  $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$
  $c := \mathsf{H}(M, \mathbf{w_1})$
  $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
  $\mathbf{r_0} := \mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ $\mathbf{z}$-check
    return $\perp$
  if $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
    return $\perp$
  return $\sigma := (c, \mathbf{z})$

function $\mathsf{Verify}((\mathbf{A}, \mathbf{t}) := \mathsf{pk}, (c, \mathbf{z}) := \sigma, M)$
  $\mathbf{w_1'} := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$
  return $[\![c = \mathsf{H}(M \,\|\, \mathbf{w_1'})]\!]$ **and** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$

3.2.1 KeyGen

The key generation routine (KeyGen) generates a uniformly distributed $k \times \ell$ public matrix $\mathbf{A}$ with elements in $R_q = \mathbb{Z}_q[X]/[X^n + 1]$. It also generates two secret vectors $\mathbf{s_1} \xleftarrow{\$} S_\eta^\ell, \mathbf{s_2} \xleftarrow{\$} S_\eta^k$ where S_η is the set of polynomials in R with coefficients in $\{-\eta, \dots, \eta\}$. We define $\mathsf{sk} := (\mathbf{A}, \mathbf{s_1}, \mathbf{s_2})$ as the secret key, and $\mathsf{pk} := (\mathbf{A}, \mathbf{t})$ is the generated public key. $\mathbf{A}$ is included as part of the secret key, because it will be necessary in the generation of signatures (Section 3.2.2). However, only $\mathbf{s_1}$ and $\mathbf{s_2}$ need to remain secret.

We can intuitively see how the secrecy of $\mathbf{s_1}$ and $\mathbf{s_2}$ is provided by the MLWE assumption: if the adversary is provided an oracle that, given $\mathbf{A}$ and $\mathbf{t}$, provides information about $\mathbf{s_1}$ and $\mathbf{s_2}$, then they can use that to solve instances of the MLWE problem.

3.2.2 Sign

In the signature generation algorithm, the HighBits routine is first used. It is defined as follows.

Definition 3.4 (HighBits**,** LowBits**).** HighBits *and* LowBits *uniquely decompose a vector* $\mathbf{x}$ *such that*

$$\alpha \cdot \mathsf{HighBits}(\mathbf{x}, \alpha) + \mathsf{LowBits}(\mathbf{x}, \alpha) = \mathbf{x} \text{ and } \|\mathsf{LowBits}(\mathbf{x}, 2\alpha)\|_\infty < \alpha. \tag{3.3}$$

Using the HighBits function, we compute the signature generation algorithm in multiple stages. First, the signer samples a random *nonce* (or *mask*) $\mathbf{y}$ with all coefficients smaller than γ_1, and from it computes the *commitment* $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$. Then the commitment is, after concatenation with the message, input into a random oracle H, which outputs a *challenge* polynomial c. The signer then computes the *response* $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$. After the response is computed, the signer will do two checks that are necessary to ensure that the scheme is secure.

In the $\mathbf{z}$-check, the signer checks that $\|\mathbf{z}\|_\infty$ is smaller than $\gamma_1 - \beta$, where β is chosen such that $\|c\mathbf{s_1}\|_\infty \leq \beta$. This check ensures that, when output, $\mathbf{z}$ does not leak any information about $\mathbf{s_1}$. In the $\mathbf{r_0}$-check, the signer will check that $\mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2) = \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2)$, and that $\mathbf{r_0}$ does not leak any information about $\mathbf{s_2}$. This check also ensures that the commitment $\mathbf{w_1}$ will be recoverable from $\mathbf{z}$ during the

signature verification. If any of these checks fail, the signature generation is aborted. After all these steps are done, the algorithm outputs the signature $\sigma := (c, \mathbf{z})$.

3.2.3 Verify

To verify σ, the verifier first recovers the commitment by computing $\mathbf{w_1'} := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$. Then the verifier can use $\mathbf{w_1'}$ to compute $c' := \mathsf{H}(\mu \,\|\, \mathbf{w_1'})$. The first check in the verification ensures that $c' = c$. The second check ensures that $\mathbf{z}$ is small, which is required because the MSIS assumption only holds if $\mathbf{z}$ is small (see Definition 3.2).

Correctness. For the correctness of the first check in the signature verification, we first show that $\mathbf{w_1'} = \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2)$:

$$\begin{aligned}
\mathbf{w_1'} &:= \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) \\
&= \mathsf{HighBits}(\mathbf{A}(\mathbf{y} + c\mathbf{s_1}) - c(\mathbf{As_1} + \mathbf{s_2}), 2\gamma_2) \\
&= \mathsf{HighBits}(\mathbf{Ay} + \mathbf{A}c\mathbf{s_1} - c\mathbf{As_1} - c\mathbf{s_2}, 2\gamma_2) \\
&= \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2) =: \mathbf{r_1}
\end{aligned}$$

To show that $\mathbf{r_1} = \mathbf{w_1}$, consider that $\mathbf{r_1} \neq \mathbf{w_1}$: if $\mathbf{r_1} \neq \mathbf{w_1}$, this means that the subtraction of $c\mathbf{s_2}$ from $\mathbf{Ay}$ has lead to a "carry" appearing in $\mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2)$. When this carry appears, the corresponding coefficient (call it x) in $\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2}, 2\gamma_2)$ is "close to overflow", i.e., $|x| \geq \gamma_2 - \beta$. However, it was ensured by the $\mathbf{r_0}$-check this is not that case, and as such $\mathbf{r_1}$ must be equal to $\mathbf{w_1}$. Then, it follows that $\mathsf{H}(M \,\|\, \mathbf{w_1'}) = \mathsf{H}(M \,\|\, \mathbf{w_1}) = c$.

The correctness of the second check in the signature verification follows directly from the fact that all candidate signatures with $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ were rejected during the signing algorithm. □

3.2.4 Security

As already mentioned, the security of Dilithium is complex and I will try to not replicate the complete proof in this chapter. Besides, at the time of writing, some discoveries are still quite recent [BBDD+23; DFPS23], which have impacted the security proof of Dilithium. This section contains an *intuitive* impression of the security analysis of Dilithium (i.e., not a complete one).

Canonical identification schemes (ID). The Dilithium security argument consists of multiple layers. At its core is a common *canonical identification scheme* (ID) [AABN02; GMR85]. ID schemes consist of three steps: First, the *prover* (which holds the secret key) produces and sends a *commitment* to the verifier (which, in the case of Dilithium, is the value $\mathbf{w_1}$). After receipt of the commitment, the verifier sends a random *challenge* back to the prover (c). Finally, the prover computes a *response* ($\mathbf{z}$) from the challenge. If $\mathbf{z}$ satisfies the checks described in Section 3.2.2, the prover sends it to the verifier $\mathbf{z}$; and otherwise it *aborts*. Using the prover's public key, the verifier uses the complete *transcript* $(\mathbf{w_1}, c, \mathbf{z})$ to decide if the proof is accepted.

Fiat–Shamir-with-aborts (FSwA) and the random oracle model ROM. The identification scheme (and its security proof) is transformed from the interactive setting to a non-interactive signature scheme using the Fiat–Shamir-with-aborts (FSwA) transform [Lyu09]. FSwA is based on the Fiat–Shamir (FS) transform [AABN02; FS87], which replaces the generation of c by the verifier with the computation $c := \mathsf{H}(M \| \mathbf{w_1})$, where H is a hash function modelled as a *random oracle* [BR93]. The difference between the FS transform and the FSwA transform is that the FSwA transform deals with the aborting nature of the underlying identification scheme. The FSwA transform achieves this by wrapping the aborting scheme in a rejection-sampling loop, essentially retrying the signature generation algorithm until a good (i.e., non-leaking) signature is found. For Fiat–Shamir-based schemes to be secure in the ROM, they need to meet two core security properties[1]: special soundness, and honest verifier zero-knowledgeness.

Special soundness. The *special soundness* property is the property that provides unforgeability under no-message-attack UF-NMA. Special soundness is usually demonstrated using the *forking lemma* [PS00]. In the Dilithium forking lemma, we first assume that the MLWE problem (Definition 3.1) is hard. The hardness of MLWE implies that all public keys $(\mathbf{A}, \mathbf{t})$ are cannot be distinguished from uniformly distributed keys by the adversary. The forking lemma now uses the reprogrammability ability of the random oracle to show that if we have a signing adversary $\mathscr{A}_{\mathsf{H}}$ that takes *only public data* as input, then we can construct a reduction $\mathscr{B}_{\mathscr{A}_{\mathsf{H}}}$ that can generate two signatures $(\mathbf{w_1}, c, \mathbf{z})$ and $(\mathbf{w_1}, c', \mathbf{z'})$, with $c \neq c'$. We derive solutions to the MSIS

[1] Apart from correctness (see Section 3.2.3), and an adequate min entropy of the commitment.

problem from the values $\mathbf{z} - \mathbf{z}'$ and $c - c'$. Because MSIS was assumed to be hard to solve, this means that the scheme is UF-NMA.

Concretely, where $\mathscr{B}$, $\mathscr{C}$ and $\mathscr{D}$ are adversaries in each applicable game, the following relation holds between the adversaries' advantages:

$$\mathsf{Adv}^{\mathsf{UF\text{-}NMA}}_{\mathsf{SimplifiedDilithium}}(\mathscr{B}) \leq \mathsf{Adv}^{\mathsf{MLWE}}_{k,\ell,S_\eta}(\mathscr{C}) + \mathsf{Adv}^{\mathsf{MSIS}}_{k,\ell+1,\zeta}(\mathscr{D})$$

where $\zeta := \max\{\gamma_1 - \beta, 2\gamma_2 + 1\}$.

Honest verifier zero-knowledgeness. *Honest verifier zero-knowledgeness* (HVZK) indicates that all signatures produced by the scheme are statistically independent of the secret key. This property is important because otherwise, each signature would provide the attacker with some information about the secret key.[2] In HVZK schemes, adversaries that are provided a signing oracle that *knows* sk have no advantage over adversaries that are provided a signing oracle that only takes public information as input. And because there is no advantage, these two scenarios (i.e., games) are equivalent. In this way, HVZK allows a security reduction from UF-CMA to UF-NMA. I.e., if a Fiat–Shamir based scheme is shown to be UF-NMA secure *and* HVZK, then it is also UF-CMA secure.

Concretely, where $\mathscr{A}$ and $\mathscr{B}$ are adversaries against the applicable games, and ϵ is negligible, if the scheme is zero-knowledge, then

$$\mathsf{Adv}^{\mathsf{UF\text{-}CMA}}_{\mathsf{SimplifiedDilithium}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathsf{UF\text{-}NMA}}_{\mathsf{SimplifiedDilithium}}(\mathscr{B}) + \epsilon$$

HVZK is usually demonstrated using a *simulator* $\mathcal{S}$. This simulator is a subroutine that generates simulated signatures that are statistically independent of real signatures in the ROM. The Dilithium simulator is listed in Algorithm 3.5.

The simulated signatures are statistically independent of the "transcript" signatures (output by a signing oracle). Therefore, we can swap out the signing oracle for the simulator in the security game. Because of the ROM, the signature distributions are not perfectly equal, so a security loss is incurred.[3]

[2] That scenario would be workable, but then we would have to either ensure that the secret key is not used after too much information has leaked (like in [BDH11]), or we would have to ensure that (for each signature) the amount of information that is leaked about the secret key is limited (e.g., in [BHHL+15]).

[3] See [BBDD+23, Section 5] for concrete values.

Algorithm 3.5: Simulator for Dilithium signatures.

function $\mathcal{S}_{\mathsf{H}}((\mathbf{A}, \mathbf{t}) := \mathsf{pk}, M)$
  loop
    $(c, \mathbf{z}) \xleftarrow{\$} B_\tau \times S^\ell_{\gamma_1-\beta}$ ▷ $B_\tau \times S^\ell_{\gamma_1-\beta}$ is the set of all challenges and responses
    $\mathbf{r_1} := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$
    $\mathbf{r_0} := \mathsf{LowBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$
    if $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
      continue
    $\mathsf{H}(\mathbf{r_1} \,\|\, M) := c$
    return $\sigma' := (c, \mathbf{z})$

Strongness. At this point, the scheme is UF-CMA secure. However, in the Dilithium specification [LDKL+20, Section 6.2.2], a short proof is included that shows that Dilithium is SUF-CMA by (again) reducing the security to the MSIS assumption.

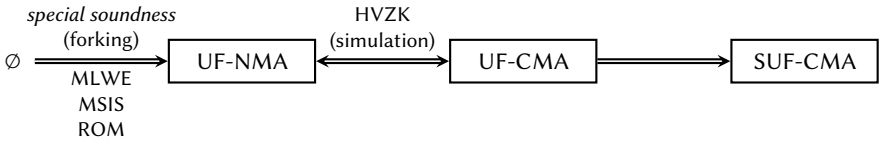


Figure 3.1: Structure of the Dilithium security proof.

Recap. Let us briefly recap the contents of this section. The structure of Dilithium is based on canonical identification schemes. Using the Fiat–Shamir with aborts heuristic, the scheme is transformed into a non-interactive signature scheme. The security reduction of Dilithium consists of three steps (see Figure 3.1): Special soundness is used to show UF-NMA security in the ROM, provided by the MLWE and MSIS assumptions. Honest verifier zero-knowledgeness is used to show that (in the ROM) the UF-NMA scheme is UF-CMA secure. Lastly, the (already assumed) hardness of the MSIS assumption makes Dilithium SUF-CMA secure.

Throughout the security proof, a number of security losses are incurred, which are all negligible in the security parameter. The concrete security loss of the complete scheme has recently been reanalyzed and is summarized in [BBDD+23, Theorem 2].

3.3 Dilithium

The previous section described a simplified version of Dilithium. However, for efficiency reasons, the *real* Dilithium scheme is a bit more complex. In Algorithm 3.17 you will find the main algorithm listing, while the running text contains the additional functions and collections that are used in the scheme. In this section, we cover the version of Dilithium that was submitted to the third round of the NIST competition.

3.3.1 Symbols and subroutines

Definition 3.6 (R_q). *R_q describes the main polynomial ring that is used. R_q is defined as $R_q := \mathbb{Z}_q[X]/(X^n + 1)$.*

Definition 3.7 (S_η). *S_η denotes the set of all polynomials in R_q with coefficients in $[-\eta, +\eta]$.*

Definition 3.8 ($\tilde{S}_{\gamma_1}$). *$\tilde{S}_{\gamma_1}$ denotes the set of all polynomials in R_q with coefficients in $[-\gamma_1, +\gamma_1)$.*

Definition 3.9 (B_τ). *B_τ denotes the set of all polynomials in R_q with exactly τ coefficients in $\{+1, -1\}$, and all the other coefficients 0.*

Definition 3.10 ($\hat{R}_q$). *$\hat{R}_q$ describes the NTT domain of R_q (and will be concretely defined in Equation (3.7)).*

Definition 3.11 (H). H *is a cryptographic hash function that is modeled as a random oracle with an output length of 256 bits. It is instantiated with* SHAKE256.

Definition 3.12 (CRH). CRH *is another cryptographic hash function, required to be collision resistant, which is also instantiated with* SHAKE256. *It is different from* H *in that its output is 384 bits long.*

Definition 3.13 (ExpandA, ExpandS, and ExpandMask). ExpandA, ExpandS, *and* ExpandMask *each pseudorandomly sample polynomials uniformly from a seed.* ExpandA *samples polynomials in $\hat{R}_q$,* ExpandS *samples polynomials in S_η, and* ExpandMask *samples polynomials in $\tilde{S}_{\gamma_1}$.*

Definition 3.14 (SampleInBall**).** *The challenge $\tilde{c}$ that is produced by the hash function on line 20 of algorithm 3.17 is a 256-bit bit-string. To convert this bit-string into an element in B_τ, it is used as a seed for a* SHAKE256 *instance. The destination polynomial c is initialized as $c := 1 \cdot X^0 + \cdots + 1 \cdot X^{\tau-1}$. Then τ bits are squeezed from the* SHAKE256 *instance to randomize the signs of the non-zero coefficients. More random bits are squeezed and used for a Fisher–Yates shuffling algorithm [Dur64] that randomizes the location of the* 1 *coefficients in c. The resulting polynomial is a polynomial that is uniformly distributed in B_τ.*

Definition 3.15 (PopCount**).** $\mathsf{PopCount}(x)$ *returns the* population count *(or* Hamming weight*) of x, i.e., the number of non-zero coefficients.* $\mathsf{PopCount}(\mathbf{x})$ *is computed recursively from the elements of $\mathbf{x}$, i.e.,* $\mathsf{PopCount}(\mathbf{x}) = \sum_{x_i \in \mathbf{x}} \mathsf{PopCount}(x_i)$.

Definition 3.16 (MakeHint **and** UseHint**).** $\mathsf{MakeHint}(z, r, \alpha)$ *returns a hint bit h that is* 1 *if* $\mathsf{HighBits}(r, \alpha) \neq \mathsf{HighBits}(r + z, \alpha)$ *or* 0 *if both terms are equal.* $\mathsf{UseHint}(h, r, \alpha)$ *uses the hint generated by* MakeHint *to recover the original value of r.*

Together, MakeHint *and* UseHint *satisfy*

$$\mathsf{UseHint}(\mathsf{MakeHint}(z, r, 2\gamma_2), r, 2\gamma_2) = \mathsf{HighBits}(r + z, 2\gamma_2).$$

3.3.2 KeyGen, Sign & Verify

The key generation, signing, and verification algorithms are listed in Algorithm 3.17. Even though the real Dilithium algorithm looks a lot more complex, it still follows the same structure as the simplified algorithm that we saw earlier in Algorithm 3.3.

In the new version, the KeyGen algorithm now takes a single random seed ζ as input. From that seed, three other seeds ρ, ς, and K are sampled. ρ denotes the seed which determines the value of $\mathbf{A}$; ς determines the secret key vectors $\mathbf{s_1}$ and $\mathbf{s_2}$; and K is used during the signing algorithm to generate random $\mathbf{y}$s. This tweak (generating the keypair from a single seed) allows us to regenerate the secret key from ζ every time when we are signing messages. This can be useful in cases in which a device's key storage space is limited, as ζ is just 32 bytes in size.

The computation of $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$ goes via the number theoretic transform (NTT). NTT-based multiplications are a very efficient method to multiply the polynomials in R_q, which will be covered in more depth in Section 3.4. After $\mathbf{t}$ is computed, it is split

Algorithm 3.17: Dilithium signature scheme.

function KeyGen$\left(\zeta \in \{0,1\}^{256}\right)$
  $(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} := \mathsf{H}(\zeta)$
  $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\varsigma)$
  $\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$
  $\mathbf{t} := \mathbf{A s_1} + \mathbf{s_2}$ ▷ Compute $\mathbf{A s_1}$ as $\mathsf{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{s_1}))$
  $(\mathbf{t_1}, \mathbf{t_0}) := (\lfloor \mathbf{t} \rceil^d, [\mathbf{t}]_d)$ ▷ Recall that $[\_]_d$ and $\lfloor \_ \rceil^d$ denote bit-selection (Sec. 2.2)
  $\mathsf{tr} \in \{0,1\}^{512} := \mathsf{CRH}(\rho \parallel \mathbf{t_1})$
  return $(\mathsf{sk} := (\rho, K, \mathsf{tr}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}), \mathsf{pk} := (\rho, \mathbf{t_1}))$

function Sign$((\rho, K, \mathsf{tr}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}) := \mathsf{sk}, M)$
  $\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$
  $\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{tr} \parallel M)$
  $\mathsf{rnd} \in \{\text{""}\} \cup \{0,1\}^{256} := \text{""}$ ▷ Or $\mathsf{rnd} \xleftarrow{\$} \{0,1\}^{256}$ (see Section 3.3.4)
  $\rho' \in \{0,1\}^{384} := \mathsf{CRH}(K \parallel \mathsf{rnd} \parallel \mu)$
  $\kappa := 0$ ▷ κ is a 16-bit counter
  loop ▷ Precompute $\hat{\mathbf{s_1}} = \mathsf{NTT}(\mathbf{s_1})$, $\hat{\mathbf{s_2}} = \mathsf{NTT}(\mathbf{s_2})$, $\hat{\mathbf{t_0}} = \mathsf{NTT}(\mathbf{t_0})$
    $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell := \mathsf{ExpandMask}(\rho' \parallel \kappa)$
    $\kappa := \kappa + 1$
    $\mathbf{w} := \mathbf{A y}$ ▷ Compute $\mathbf{w} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{y}))$
    $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{w}, 2\gamma_2)$
    $\tilde{c} \in \{0,1\}^{256} := \mathsf{H}(\mu \parallel \mathbf{w_1})$
    $c \in B_\tau := \mathsf{SampleInBall}(\tilde{c})$ ▷ Precompute $\hat{c} := \mathsf{NTT}(c)$
    $\mathbf{z} := \mathbf{y} + c \cdot \mathbf{s_1}$ ▷ Compute $c\mathbf{s_1} := \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s_1}})$
    $\mathbf{r_0} := \mathsf{LowBits}(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$ ▷ Compute $c\mathbf{s_2} := \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s_2}})$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ $\mathbf{z}$-check
      continue
    if $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
      continue
    $\mathbf{h} := \mathsf{MakeHint}(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$ ▷ Compute $c\mathbf{t_0} := \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t_0}})$
    if $\|c\mathbf{t_0}\|_\infty \geq \gamma_2$ **or** $\mathsf{PopCount}(\mathbf{h}) > \omega$ **then** ▷ Compression check
      continue
    return $(\mathbf{z}, \mathbf{h}, \tilde{c})$

function Verify$((\rho, \mathbf{t_1}) := \mathsf{pk}, M, \sigma)$
  $\hat{\mathbf{A}} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$
  $\mathsf{tr} \in \{0,1\}^{512} := \mathsf{CRH}(\mathsf{pk}) \parallel M$
  $\mu \in \{0,1\}^{384} := \mathsf{CRH}(\mathsf{tr})$
  $c \in B_\tau := \mathsf{SampleInBall}(\tilde{c})$ ▷ Precompute $\hat{c} := \mathsf{NTT}(c)$
  $\mathbf{w_1}' := \mathsf{UseHint}(\mathbf{Az} - c\mathbf{t_1} \cdot 2^d, 2\gamma_2)$ ▷ Compute as $\mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{z}} - \hat{c} \circ \hat{\mathbf{t_1}} \cdot 2^d)$
  return $[\![\tilde{c} = \mathsf{H}(\mu \parallel \mathbf{w_1}')]\!]$ **and** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$

in two, with $\mathbf{t_1}$ containing the "high bits", and $\mathbf{t_0}$ containing the "low bits". This is part of a public-key compression scheme that is implemented in Dilithium, whose details are not really important. The key takeaway is that, even though $\mathbf{t_0}$ is part of the secret key, its exclusion from pk is only done to reduce the public key size. $\mathbf{t_0}$ does not need to remain secret for the security of Dilithium. Lastly, tr is a domain-separation value that is precomputed during the key generation algorithm.

Moving over to the signature generation algorithm, we first precompute a batch of values, such that they do not have to be recomputed during every iteration of the rejection-sampling loop: $\hat{\mathbf{A}}$, μ, ρ', $\hat{\mathbf{s_1}}$, $\hat{\mathbf{s_2}}$, $\hat{\mathbf{t_0}}$. On Line 15, we enter the rejection-sampling loop. Just as in the simplified version of Dilithium, we first generate a nonce vector $\mathbf{y}$. Via the NTT domain, we compute the commitment $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$. $\mu \parallel \mathbf{w_1}$ is hashed into the challenge bitstring $\tilde{c}$. This bitstring is used as a seed for the polynomial representation of the challenge (c) with the SampleInBall function. $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are computed via the NTT domain, and then the response $\mathbf{z}$ is computed. After the main rejection-sampling checks ($\mathbf{z}$-check and $\mathbf{r_0}$-check) pass, the core of the signature generation has finished. From Line 28 and further, $\mathbf{z}, \tilde{c}$ are public and could be encoded into a signature. Lines 28–30 compute some auxiliary information into $\mathbf{h}$ in order to help the verifier to verify the signature with only $\mathbf{t_1}$.

The verification routine again precomputes the values $\hat{\mathbf{A}}$, μ, and tr. It uses the information in $\mathbf{h}$ to reconstruct $\mathbf{w_1}$ without $\mathbf{t_0}$. Then it checks if the hash of $\mu \parallel \mathbf{w_1}$ matches $\tilde{c}$, and it ensures that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$.

3.3.3 Parameter sets

Table 3.1 lists all of the Dilithium parameters based on the NIST competition requirements. In this table Dilithium{2,3,5} targets the NIST security level {2,3,5}. That is, Dilithium2 targets a security level equivalent to finding a collision for a 256-bit hash function; Dilithium3 targets 192-bit key search; and Dilithium5 targets 256-bit key search [NIST16].

3.3.4 Randomized signatures

Although Dilithium signature generation is principally defined as a deterministic algorithm, there exists a non-deterministic variant. In this variant, on Line 13 of Algorithm 3.17, rnd is initialized with a random value, instead of the empty string

Table 3.1: Dilithium parameters

NIST security level	Dilithium2	Dilithium3	Dilithium5
sk size [B]	2528	4000	4864
pk size [B]	1312	1952	2592
σ size [B]	2420	3293	4595
n (ring dimension) [(a)]	256	256	256
q (main modulus) [(a)]	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$
(k, ℓ) (dimensions of $\mathbf{A}$)	$(4, 4)$	$(6, 5)$	$(8, 7)$
η (max coeff value of $\mathbf{s_1}, \mathbf{s_2}$)	2	4	2
τ (pop count of c)	39	49	60
β $(= \tau \cdot \eta)$	78	196	120
γ_1 (max coeff value of $\mathbf{y}$)	2^{17}	2^{19}	2^{19}
γ_2 (max coeff value of $\mathbf{w_0}$ and $\mathbf{r_0}$)	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
d (bit size of $\mathbf{t_0}$) [(a)]	13	13	13
ω (max of hints set)	80	55	75
Pr[¬reject] [(b)]	0.24	0.20	0.26
$\mathbb{E}$[#iterations] [(c)]	4.25	5.09	3.85
x : Pr[#iterations $\geq x$] $\leq 2^{-128}$ [(d)]	332	406	296

[(a)] This value is the same across all parameter sets.
[(b)] Based on [LDKL+22, Equation 5].
[(c)] $\mathbb{E}[\text{iterations}] = \sum_{k=1}^{\infty} k \cdot \Pr[\text{reject}]^{k-1} \cdot \Pr[\neg\text{reject}]$ is the expected (mean of the) number of rejection-sampling loop iterations during the signature generation algorithm.
[(d)] With probability $\left(1 - 2^{128}\right)$, the rejection-sampling loop in the signature generation algorithm will take at most x iterations $\left(\text{computed as } -\frac{128}{\log_2 \Pr[\text{reject}]}\right)$.

(""). This will result in different $\mathbf{y}$ vectors being generated for different signatures over the same message.

In deterministic Dilithium, the Sign function does not need to be provided a strong randomness source (which may be difficult on some embedded platforms), and it makes it easier to test the correctness of generated signatures using established test vectors. On the other hand, when a hardware RNG is present, randomized implementations may far outperform their deterministic equivalents in terms of speed. More so, in embedded implementations, SHAKE256 is sometimes more prone to side-channel

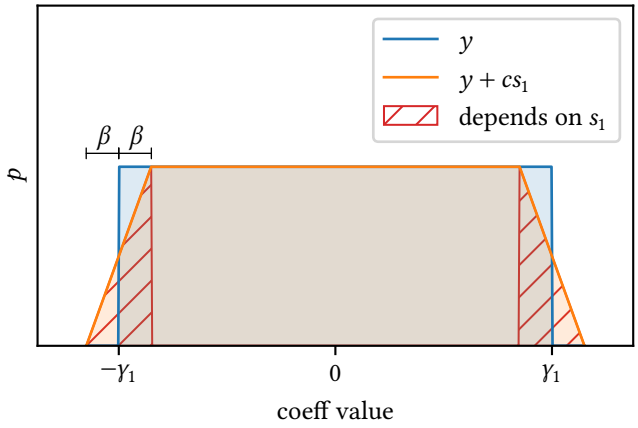


Figure 3.2: The distribution of $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ before the $\mathbf{z}$-check (if $c\mathbf{s_1}$ is assumed to be randomly distributed in S_β^ℓ).

leakage than the internal RNG. In these cases, we might favor randomized signature generation.

3.3.5 Rejection sampling

Dilithium is based on FSwA, and as such, the signature generation algorithm is built around a rejection-sampling loop. Let us zoom in a bit more into the rejection sampling property of Dilithium, as it is not very typical for signature schemes to be built around rejection sampling.

The crux of Dilithium that leads to rejection sampling is the ability to construct simulated transcripts with $\mathcal{S}$. $\mathbf{z}$ vectors *before* the $\mathbf{z}$-check depend on $\mathbf{s_1}$, and $\mathbf{r_0}$ vectors *before* the $\mathbf{r_0}$-check depend on $\mathbf{s_2}$. These dependencies violate the zero-knowledgeness property, as no simulator can be constructed that generates simulated transcripts that are statistically independent from real transcripts.

$\mathbf{z}$-check. In the signature generation algorithm, the $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ addition "blurs" the uniform value $\mathbf{y}$, which is illustrated in Figure 3.2. After this addition (before the check), the distribution of $\mathbf{z}$ is uniform between $-\gamma_1 + \beta$ and $\gamma_1 - \beta$, but in the regions

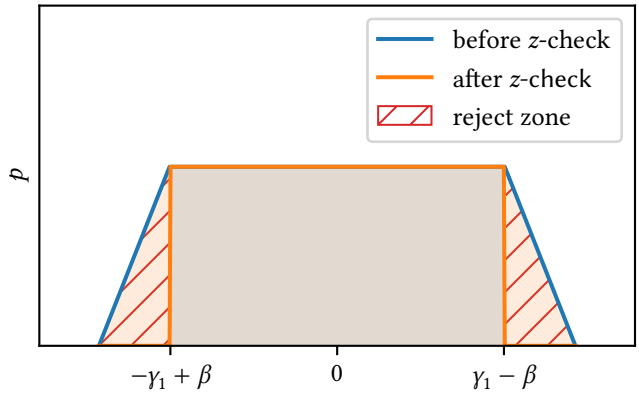


Figure 3.3: Probability distribution of $\mathbf{z}$ before and after the $\mathbf{z}$-check, and the zone where $\mathbf{z}$ coefficients lead to an abort.

$[-\gamma_1-\beta,-\gamma_1+\beta]$ and $[\gamma_1-\beta,\gamma_1+\beta)$ the probability ramps up and down. More importantly, in these regions the probability distribution also *depends* on $c\mathbf{s_1}$

To overcome this, the $\mathbf{z}$-check is added. The $\mathbf{z}$-check cuts off the ramps from both sides of the probability distribution of $\mathbf{z}$. This leaves a $\mathbf{z}$ vector with all coefficients uniformly distributed in $(-\gamma_1+\beta,\gamma_1-\beta)$ as depicted in Figure 3.3 (i.e., $\mathbf{z}$ is uniform in $S^{\ell}_{\gamma_1-\beta}$) which can easily be simulated by $\mathcal{S}$.

$\mathbf{r_0}$-check. The $\mathbf{r_0}$-check has a similar function, as it ensures the zero-knowledgeness of $\mathbf{r_0}$. In Algorithm 3.3, $\mathbf{r} := \mathbf{w} - c\mathbf{s_2}$, which is recovered during as $\mathbf{r} = \mathbf{Az} - c\mathbf{t}$. For valid signatures it holds that $\mathbf{r_1} = \mathbf{w_1}$, therefore the verifier and adversary also know $\mathbf{r} - \mathbf{r_1} = \mathbf{r_0} = \mathsf{LowBits}(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$.

Recall from Section 3.2.3 that the addition of $c\mathbf{s_2}$ did not lead to any "carries" in the high-bits part of $\mathbf{r}$, because otherwise $\mathbf{w_1}$ would not equal $\mathbf{r_1}$. However, this means that if a coefficient of $\mathsf{LowBits}(\mathbf{w}-c\mathbf{s_2}, 2\gamma_2)$ is close to $\pm\gamma_2$, this means that that coefficient in $c\mathbf{s_2}$ is small. As such, coefficients close to $\pm\gamma_2$ depend on $c\mathbf{s_2}$.

Indeed, the shape of the probability distribution of $\mathbf{r_0} \mid (\mathbf{r_1} = \mathbf{w_1})$ is similar to that of $\mathbf{y} + c\mathbf{s_1}$ in Figure 3.2. Conversely, we cut off the same ramps from the edges, similar to the action depicted in Figure 3.3. This leads to a $\mathbf{r_0}$ vector that is uniformly distributed in $S^{k}_{\gamma_2-\beta}$, which is simulatable by $\mathcal{S}$.

Declassify**ing the branch condition bit.** In Section 2.6.1, we mentioned that we do not branch on values that are secret. However, before the checks, both $\mathbf{z}$ and $\mathbf{r_0}$ are secret vectors. In order to reject the signature if one of the vectors exceeds the norm bound, we need to Declassify them first. This is done on a coefficient-wise basis. That is, for each coefficient it is determined (in constant-time) whether they exceed the bound, which will result in a bit $b \in \{\text{true}, \text{false}\}$. This bit is declassified, after which the declassified bit is used as the conditional for the subcheck for this coefficient.

Practical implications. As described, after aborting on an "incorrect" signature, the signing algorithm keeps generating new candidate signatures until one of them passes both checks. The acceptance probability is constant and listed in Table 3.1 under the label Pr[¬reject].

Because of the rejection-sampling loop, the signing algorithm has a probabilistic runtime. Every loop iteration the algorithm might finish, leading to a geometrically distributed number of loop iterations, as illustrated in Figure 3.4. This leads to an algorithm that may, under very unlucky scenarios, take a long time to execute. For example, on average once every 2^{20} ($\approx$ 1 million) executions, Dilithium3-Sign will take at least 64 iterations to execute. This bad "worst-case" performance is a significant weakness of Dilithium, and it might pose a problem for real-time applications.

Furthermore, we cannot use the convention of reporting our software speed benchmarks by median, because that convention is based on the assumption that an algorithm always takes the same time to execute (with all deviations being attributed to noise produced by the platform). Instead, for Dilithium we report speed benchmarks using the average run-time.

3.4 The number theoretic transform

Schoolbook multiplication. Apart from computing Keccak permutations, most of Dilithium's run-time is occupied by multiplications in R_q.[4] The naive method of computing the product of two polynomials a and b is to use the *Schoolbook* method. That is, we compute

[4] In Chapter 4, on Cortex-M4 about 80% of signature verification is Keccak, and of the other 20%, about 70% of functions contain arithmetic in R_q.

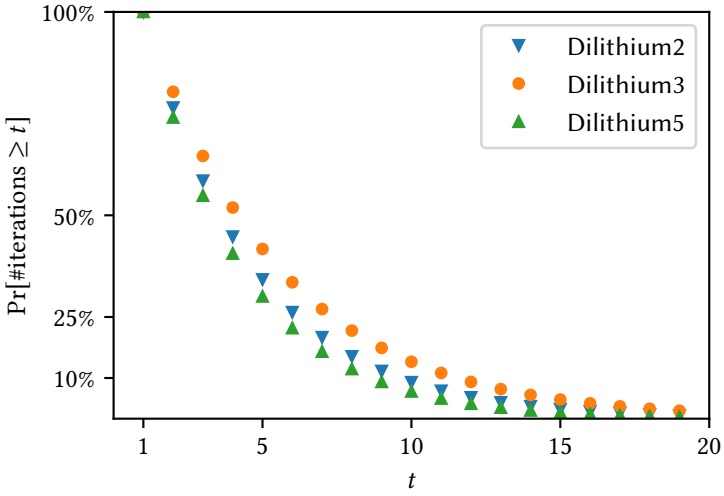


Figure 3.4: Expected percentage of Sign executions that needs at least t rejection-sampling loop iterations.

$$(a \cdot b)_k = \sum_{i+j=k} a_i b_j. \tag{3.4}$$

However, our ring R_q is $\mathbb{Z}_q[X]/(X^n+1)$, so when we take into account the reduction of the product into an $(n-1)$-degree polynomial, we get

$$(a \cdot b)_k = \sum_{i+j\equiv k, i+j<n} a_i b_j - \sum_{i+j\equiv k, i+j\geq n} a_i b_j. \tag{3.5}$$

Using this method, for *every* coefficient in the product, we need to do n multiplications of coefficients. Because we need to do this for n coefficients in the product, this results in a multiplication complexity of $\mathcal{O}(n^2)$. For Dilithium, which has a not-so-small $n = 256$, this results in very slow Schoolbook polynomial multiplications.

In order to make the Dilithium scheme faster, the Dilithium scheme is heavily tuned to enable faster polynomial multiplications through the use of the *number theoretic transform* (NTT). Using the NTT, the polynomial multiplication complexity in Dilithium is reduced to $\mathcal{O}(n \log n)$.

The number theoretic transform is an application of the Fourier transform applied to finite fields [Fid72; Pol71]. It works by choosing a polynomial ring such that multiplication becomes analogous to vector-convolution of the coefficient vectors. Now the convolution theorem applies, which states that where $\hat{a} = \mathsf{NTT}(a)$ and $\hat{b} = \mathsf{NTT}(b)$,

$$a \cdot b = \mathsf{NTT}^{-1}(\hat{a} \circ \hat{b}). \tag{3.6}$$

With this new method NTT-based method of multiplying polynomials, the complexity is dominated by NTT and NTT^{-1}. Fortunately, just as with regular Fourier transforms, we can apply the *fast Fourier transform* (FFT) [CT65] to implement NTT and NTT^{-1} in $\mathcal{O}(n \log n)$. Now the full multiplication is implemented in $\mathcal{O}(n \log n)$.

Splitting R_q using the CRT. The *Chinese remainder theorem* (CRT) allows us to split R_q into two smaller rings. The CRT (generalized to rings) considers a list of k polynomials a_i modulo some other polynomials m_i which are all pairwise coprime. Now let M be the product of these moduli, i.e., $M = m_1 \cdots m_k$. Then every list of polynomials $a_1, \ldots, a_k$ describes *exactly one* polynomial A mod M.

The polynomial modulus in R_q is $M = X^{256} + 1$. We find that M can be factorized into $m_1 = X^{128} - \zeta_4$ and $m_2 = X^{128} + \zeta_4$, where ζ_k is the kth primitive root of unity modulo q. In this situation the CRT states that all polynomials modulo $X^{256} + 1$ can be represented as two polynomials modulo $X^{128} - \zeta_4$ and $X^{128} + \zeta_4$ respectively. Or in other words,

$$\mathbb{Z}_q[X]/(X^{256} + 1) \cong \mathbb{Z}_q[X]/(X^{128} - \zeta_4) \times \mathbb{Z}_q[X]/(X^{128} + \zeta_4).$$

Together, $\mathbb{Z}_q[X]/(X^{128} - \zeta_4)$ and $\mathbb{Z}_q[X]/(X^{128} + \zeta_4)$ form a *residue number system* (RNS) for R_q. As such, the addition and multiplication in R_q can be implemented by doing those same operations on each of the split rings independently. This does not impact the speed of addition, as we needed 256 coefficient additions before splitting, and after splitting R_q we still need 256 coefficient additions. However for multiplication, the amount of coefficient multiplications has been reduced from 256^2 to $2 \cdot 128^2$. Using this RNS, the number of base multiply operations has been reduced by a factor of 2.

Recursing down. From here, we can split up the ring system further by factoring $X^{128} - \zeta_4$ and $X^{128} - \zeta_4$. This works, as

$$X^{128} - \zeta_4 = (X^{64} - \zeta_8)(X^{64} + \zeta_8), \text{ and}$$
$$X^{128} + \zeta_4 = (X^{64} - \zeta_8^3)(X^{64} + \zeta_8^3).$$

A pattern exposes itself to us here. A polynomial $(X^\alpha - \zeta_\beta)$ factors into $(X^{\alpha/2} \mp \zeta_{2\beta})$. And, as $-1 = \zeta_2$, a polynomial $(X^\alpha + \zeta_\beta)$ factors into $(X^{\alpha/2} \mp \zeta_4\zeta_{2\beta})$. So, after another layer of splitting, we get an even more granular factorization of $X^{256} + 1$:

$$(X^{64} - \zeta_8)(X^{64} + \zeta_8)(X^{64} - \zeta_8^3)(X^{64} + \zeta_8^3)$$

We can keep splitting these modulus polynomials until we are down to the last ζ_k value, which is ζ_{512}; the 512*th primitive root of unity*. Because it is the last ζ_k value, we will call this one ζ from here on[5]. In the end, we end up with the factorization of $(X^{256} + 1)$:

$$(X - \zeta)(X + \zeta)(X - \zeta^{129})(X + \zeta^{129}) \cdots (X - \zeta^{127})(X + \zeta^{127})(X - \zeta^{255})(X + \zeta^{255})$$

Correspondingly, we have constructed a ring system of 256 rings (with all moduli of degree 1) that is isomorphic to R_q. We will call this ring system $\hat{R}_q$:

$$\begin{aligned}\hat{R}_q = \quad & \mathbb{Z}_q[X]/(X - \zeta) \times \mathbb{Z}_q[X]/(X + \zeta) \times \mathbb{Z}_q[X]/(X - \zeta^{129}) \times \mathbb{Z}_q[X]/(X + \zeta^{129}) \times \cdots \\ & \cdots \times \mathbb{Z}_q[X]/(X - \zeta^{127}) \times \mathbb{Z}_q[X]/(X + \zeta^{127}) \times \mathbb{Z}_q[X]/(X - \zeta^{255}) \times \mathbb{Z}_q[X]/(X + \zeta^{255}) \end{aligned} \tag{3.7}$$

During each of these splits, we reduce the number of coefficient multiplications in a polynomial multiplication by a factor of 2. After n layers of splitting, the updated complexity of polynomial multiplication is $\mathcal{O}(n)$. This follows from intuition: after n layers, we end up with n polynomials of degree 0. Each of those is multiplied using a single coefficient multiplication.

The forward transform. Building on the construction from the previous paragraphs, we can construct an efficient mapping from R_q (the "time" domain) to $\hat{R}_q$ (the NTT domain). To transform a polynomial $a \in \mathbb{Z}_q[X]/(X^{2y} - z^2)$ one layer towards the NTT domain, we essentially reduce it modulo $\mathbb{Z}_q[X]/(X^y - z)$ and $\mathbb{Z}_q[X]/(X^y + z)$:

[5]The rule for computing ζ_k values from ζ is $\zeta_k = \zeta^{512/k}$.

$$\begin{aligned} a_L &\equiv a \bmod (X^y - z) \\ a_R &\equiv a \bmod (X^y + z) \end{aligned} \tag{3.8}$$

As an example, we take the first split that transforms a polynomial $a \in \mathbb{Z}_q[X]/\left(X^{256}+1\right)$ into two polynomials $(a_L, a_R) \in \mathbb{Z}_q[X]/(X^{128}-\zeta^{128}) \times \mathbb{Z}_q[X]/(X^{128}+\zeta^{128})$.

Let us first look at a_L. We need to take all the "top" coefficients a_i where $i \geq 128$ and reduce them modulo $X^{128} - \zeta^{128}$. Because $X^{128} \equiv \zeta^{128}$, we know that $a_i X^i \equiv a_i \zeta^{128} X^{i-128}$. So to get rid of the top coefficients, we multiply them with ζ^{128} and add them to the coefficient that is 128 spots further down. This results in:

$$a_L = (a_0 + \zeta^{128} a_{128}) + (a_1 + \zeta^{128} a_{129})X + (a_2 + \zeta^{128} a_{130})X^2 + \cdots \tag{3.9}$$

Now let us look at a_R. In this case the reduction polynomial has a positive ζ^{128} term, instead of a negative one. So in this case, because $X^{128} = -\zeta^{128}$, we multiply with $-\zeta^{128}$ instead of ζ^{128}:

$$a_R = (a_0 - \zeta^{128} a_{128}) + (a_1 - \zeta^{128} a_{129})X + (a_2 - \zeta^{128} a_{130})X^2 + \cdots \tag{3.10}$$

Both formulas can be applied in parallel in a coefficient-wise fashion. The benefit is that, for every coefficient, we only need to compute each multiplication $a_i \cdot \zeta^{128}$ only once. Then, after respectively adding and subtracting, both reductions are computed using only 128 coefficient-wise multiplications.

The base operation that maps a pair of (a_i, a_{i+128}) coefficient to $(a_{L,i}, a_{R,i})$ is called a *butterfly operation*. This particular butterfly (that of the *forward* transform), is often called the *Cooley–Tukey* (CT) butterfly [CT65].

The butterfly operation is *the* central element of the fast Fourier algorithm and can be depicted using a "butterfly diagram". It is called a "butterfly" because, with a bit of imagination, you can see a butterfly in its arrows.

The inverse transform. Using the forward transform, we can relatively easily construct the inverse transform. Let me illustrate this for the example that we covered in the previous paragraph (Equations (3.9) and (3.10)). After removing some of the clutter, these equations show the following butterfly operation:

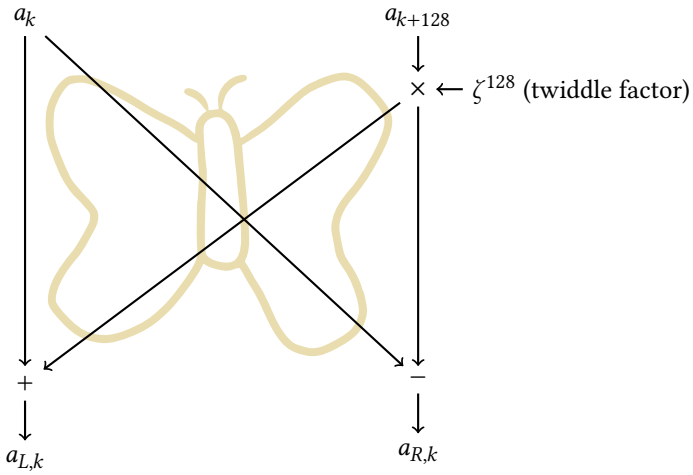


Figure 3.5: Diagram of the Cooley–Tukey butterfly for the first layer.

$$a_{L,0} = a_0 + \zeta^{128} a_{128}$$
$$a_{R,0} = a_0 - \zeta^{128} a_{128}$$

In the inverse butterfly, our inputs are $a_{L,0}$ and $a_{L,0}$; and we need to find a_0 and a_{128}. First, we add the equations together to find a formula for a_0:

$$\begin{aligned} a_{L,0} + a_{R,0} &= (a_0 + \cancel{\zeta^{128} a_{128}}) + (a_0 - \cancel{\zeta^{128} a_{128}}) \\ 2a_0 &= a_{L,0} + a_{R,0} \\ a_0 &= 2^{-1}(a_{L,0} + a_{R,0}) \end{aligned}$$

Now, we take the difference of the equations to find a formula for a_{128}:

$$
\begin{aligned}
a_{L,0}^{(1)} - a_{R,0}^{(1)} &= (\cancel{a_0} + \zeta^{128} a_{128}) - (\cancel{a_0} - \zeta^{128} a_{128}) \\
2\zeta^{128} a_{128} &= a_{L,0}^{(1)} - a_{R,0}^{(1)} \\
a_{128} &= 2^{-1}\zeta^{-128}\left(a_{L,0}^{(1)} - a_{R,0}^{(1)}\right)
\end{aligned}
$$

The reverse butterfly operation looks similar to the forward operation and this one is usually called the Gentleman–Sande (GS) or Sande–Tukey butterfly [GS66]. Its butterfly diagram is listed in Figure 3.6. We observe however that an extra factor 2^{-1} was introduced.[6] That normalization factor 2^{-1} is the same in every layer, so we can accumulate this factor for every layer into a single multiplication with $2^{-\ell}$, where ℓ is the number of NTT layers. Even though it does not matter when the $2^{-\ell}$ normalization factor is factored in, most implementations apply the normalization at the end of the inverse transform.

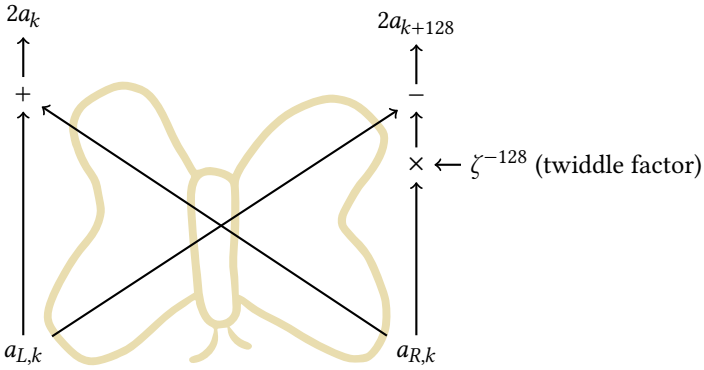


Figure 3.6: Diagram of the Gentleman–Sande butterfly for the last layer.

Polynomial multiplication efficiency. For both the forward, as well as the inverse transform, $\mathcal{O}(n \cdot \ell)$ operations are needed to compute the transformed representation of

[6] This factor does not come out of nowhere. It is similar to the normalization factor that we find in the common formula for the inverse discrete Fourier transform.

a polynomial. Hence, a full polynomial multiplication, i.e., $\mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$ is $\mathcal{O}(n \cdot \ell)$ as well. This is a lot faster than the $\mathcal{O}(n^2)$ complexity for the Schoolbook method, which highlights the importance of the NTT in Dilithium.

The NTT and FFT. The "(inverse) number theoretic transform" refers to the mathematical mapping that we just covered. The "fast Fourier transform" refers to an *implementation* of that mathematical mapping. Indeed, the FFT is not the only algorithm that implements the NTT. One could also express the (inverse) NTT using matrix multiplication, as described in one of [Gre20; Kan22], even though it would not make sense from a performance perspective.

Usually, the FFT algorithm and its inverse are called just called the "FFT" and "inverse FFT" algorithm. However, due to the vast literature on the subject, and the appropriation of the algorithm by the cryptographic community, multiple synonyms exist. For example, the (forward) FFT is also known as the "Cooley–Tukey (CT) FFT", or the "decimation-in-time (dit) FFT ". Conversely, the inverse FFT is also known as the "Gentleman–Sande (GS) FFT", the "Sande–Tukey (ST) FFT", or the "decimation-in-frequency (dif) FFT".[7] Because of the tight relation to signal processing, the untransformed data is said to be in the *time domain* (R_q). The transformed data is said to exist in the *NTT domain* or the *frequency domain* ($\hat{R}_q$).

Incomplete NTTs. We mentioned earlier that a 512th primitive root of unity must exist to execute the NTT modulo q, which is the case for Dilithium. While this is true, it is still possible to execute an NTT for $\ell < \log_2 n$ layers. We call this transformation an *incomplete NTT*. When computing polynomial products with an incomplete NTT, we apply some of the NTT layers, and then use the Schoolbook (or some other) method, to multiply the base polynomials. This idea is used in many implementations, e.g., when the modulus does not support an NTT on $\log_2 n$ layers, or as a performance performance optimization [ABCG20; ACCH+22; CHKS+21].

[7] *Gentleman–Sande* refers to the authors of the paper in which the inverse FFT butterfly was first described [GS66]. That paper however describes that version of the algorithm as the "Sande version", implying that it was invented by Sande alone. Decades later, Cooley published an essay in which he clarified that the inverse algorithm was proposed by Sande while they were following one of Tukey's courses, leading to the *Tukey–Sande* denomination [Coo87]. For consistency, we will only refer to this algorithm as the *Gentleman–Sande* algorithm from this point on.

Twisting NTTs. Multiplying all coefficients a_i of $a \in R_q$ by powers of a $2n$-th root of unity ζ_{2n} is called *twisting* [Ber01]. A twisting operation maps the polynomials from $\mathbb{Z}_q[X]/(X^n + 1)$ to $\mathbb{Z}_q[X]/(X^n - 1)$, or back if the coefficients are divided instead. Twisting during the NTT allows us to switch between kinds of butterfly operations, i.e., we can use GS butterflies to compute the NTT, and we can use CT butterflies to compute the inverse NTT. Twisting NTTs is a common optimization technique for implementations of NTT-based lattice schemes [ADPS16; LN16; Sei18]. In Chapter 5 this is used to optimize the inverse NTT in Kyber and Dilithium.

Further reading. For more information about the number theoretic transform—as it is used in Kyber and Dilithium—I highly recommend consulting the thesis of Kannwischer [Kan22, Sec. 2.2.4, 2.2.5]. If you are looking for a more fundamental source that also covers the Fermat Number Transform (see Chapter 5), I recommend reading Nussbaumer's book on the subject [Nus81, Chapter 8]. For more information about twisting in the NTT, I recommend [Sei18, Section 2.1].

4 Fast Dilithium on Cortex-M3 and Cortex-M4

4.1 Introduction

In the early stages of the NIST competition, there was still a scarcity of insight into the performance of lattice schemes on small microcontrollers. There has already been some efforts to optimize Dilithium for speed on the Cortex-M4, but the previous work has surveyed only a subset of the algorithms (i.e., only signing) or parameter sets (i.e., only for the NIST competition "recommended" security level). Moreover, we feel that the Dilithium speed records could still be broken. Therefore, we see a good reason to write a complete implementation for Cortex-M3 and Cortex-M4, including all algorithms and parameter sets, using multiple different tradeoffs for memory usage, and putting the resulting implementation in the public domain.

While the original work improves *both* the speed, as well as the memory usage of Dilithium on both architectures, this chapter will cover only the improvements to the speed. In Chapter 6, we will cover the memory improvements from this work together with the more novel contribution of [BRS22]. At the time this research was done, the NIST competition had advanced only to round 2. Therefore, unless otherwise specified, all of this chapter's contents refer to version 2 of the Dilithium algorithm [DKLL+19].

Constant time & Cortex-M3. The Cortex-M4 architecture provides various advanced instructions for optimizing cryptographic schemes, which might be one of the reasons why it gets so much attention from the cryptographic community. However as we described in Section 2.5, the Cortex-M3 comes with one "feature" which does appear interesting from an implementation and also from a side channel perspective: Different from the Cortex-M4, it does not have a constant-cycle 32-bit multiplier

producing a 64-bit result, but only a variable-cycle one. Therefore, an implementation of any scheme working on large (secret) integers compiled for the Cortex-M3 is most likely going to leak information about these secret integers via timing side channels. This has been shown to pose a problem for cryptographic schemes in preceding Arm architectures [GOPT09]. It is particularly interesting for Dilithium, because of the large prime modulus $q = 8380417$. If existing implementations for Dilithium are simply compiled for the Cortex-M3, they are very likely to be vulnerable to timing attacks within the polynomial multiplication. In this chapter, we build a safe *constant-time* implementation of Dilithium on the Cortex-M3. That is, the execution time of the algorithm is invariant over all the secret values in the algorithm.

Contribution. The contribution of this chapter is threefold: First, we further optimize the existing Dilithium implementation for the Cortex-M4 by switching to a signed polynomial representation and optimizing more parts of the scheme. Second, we present the first constant-time implementation of Dilithium on the Cortex-M3. Finally, as a by-product, we provide Cortex-M3 implementations of the lattice-based key-encapsulation schemes Kyber and NewHope. This, most notably, consists of constant-time implementations of the NTT and NTT^{-1} operations in those schemes. The original work also contained stack consumptions and speed trade-offs for the signing procedure of Dilithium, which will be covered in Chapter 6.

Code. The implementations of Dilithium, Kyber, and NewHope that are the result of this work are in the public domain and can be obtained by following the instructions described on page 10. The code is published and licensed under a CC0 copyright waiver.

Related Work. Previous speed records for Dilithium on the Cortex-M4 were set by Ravi, Gupta, Chattopadhyay, and Bhasin [RGCB19] and were built upon an implementation by Güneysu, Krausz, Oder, and Speith [GKOS18]. A masked implementation of a modified Dilithium on Cortex-M3 is presented in [MGTF19]. In that paper, Migliore, Gérard, Tibouchi, and Fouque propose to use a power-of-two modulus instead of the original prime modulus to allow for cheaper masking. However, strictly speaking, they do not implement the Dilithium scheme as it was submitted to NIST. There is an extensive line of work for Cortex-M4 implementation of lattice-based key-encapsulation

mechanisms [ABCG20; AJS16; BKS19; BKV20; KBSV18; KRS19]. Similar studies exist on hardware implementations and instruction set extensions [AELN+20; BUC19; MTKS+20]. Other lattice-based signatures have been implemented on the Cortex-M4: Pornin presents a fast constant-time implementation of Falcon on the Cortex-M4 [Por19]; In 2019, [GR19] presented a masked implementation of qTesla; More recently, [WTJB+20] presented a hardware-accelerated implementation of qTesla.

Structure of this chapter. Section 4.2 introduces the lattice-based signature scheme Dilithium and the peculiarities of the Cortex-M3 and Cortex-M4 relevant for this work. In Section 4.3 we present some improvements for the Cortex-M4. Section 4.4 presents the first constant-time implementation of Dilithium on the Cortex-M3. Section 4.5 presents the performance results for both implementations. In Appendix 4.A, we provide performance results for Kyber and NewHope on the Cortex-M3 which are a by-product of this work.

4.2 Preliminaries

Dilithium version 2. The research of this chapter was done on the older version of Dilithium that was submitted to round 2 of the NIST post-quantum competition. Therefore not all of the specification from Section 3.3 applies, as it covered the version of Dilithium that was submitted to the third round of the NIST competition. Fortunately, the structure of the keygen, signing, and verification algorithms is more or less the same.

Most of the updates to the scheme are in the parameter sets. Round-2 Dilithium had the parameter sets Dilithium2, Dilithium3, and Dilithium4. In the third round of the NIST competition, the CRYSTALS team tweaked the scheme and all its parameter sets and renamed the highest security level from Dilithium4 to Dilithium5. For both versions of Dilithium, the parameter sets are listed in Table 4.1.

Functions. As a central building block, Dilithium uses the NTT and NTT^{-1} functions which are used to implement efficient polynomial multiplication of a, b as $\mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$. The details of the Dilithium NTT are described in Section 3.4, and between round-2 and round-3 Dilithium the use of the NTT is identical. In addition, round-2 Dilithium uses a collision-resistant hash-function H directly

Table 4.1: Dilithium round-2 parameters versus Dilitium round-3 parameters

	NIST round 2			NIST round 3		
NIST security level	Dilithium2	Dilithium3	Dilithium4	Dilithium2	Dilithium3	Dilithium5
pk size [B]	1184	1472	1760	1312	1952	2592
sig size [B]	2044	2701	3366	2420	3293	4595
n (ring dimension)	256	256	256	256	256	256
q (main modulus)	$2^{23}-2^{13}+1$	$2^{23}-2^{13}+1$	$2^{23}-2^{13}+1$	$2^{23}-2^{13}+1$	$2^{23}-2^{13}+1$	$2^{23}-2^{13}+1$
(k,ℓ) (dimensions of $\mathbf{A}$)	$(4,3)$	$(5,4)$	$(6,5)$	$(4,4)$	$(6,5)$	$(8,7)$
η (max coeff value of $\mathbf{s_1}, \mathbf{s_2}$)	6	5	3	2	4	2
τ (pop count of c)	60	60	60	39	49	60
β $(= \tau \cdot \eta)$	360	300	180	78	196	120
γ_1 (max coeff value of $\mathbf{y}$)	$(q-1)/16$	$(q-1)/16$	$(q-1)/16$	2^{17}	2^{19}	2^{19}
γ_2 (max coeff value of $\mathbf{w_0}$ and $\mathbf{r_0}$)	$(q-1)/32$	$(q-1)/32$	$(q-1)/32$	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
d (bit size of $\mathbf{t_0}$)	14	14	14	13	13	13
ω (max of hints set)	80	96	120	80	55	75

outputting a challenge polynomial roughly following the method as described in Definition 3.14. Furthermore, round-2 Dilithium defines the seed expansion functions ExpandA and ExpandMask; the rounding functions Power2Round, HighBits, and Decompose and the hint functions MakeHint and UseHint as defined in Section 3.3.1. Even though small differences may be present between the different Dilithium versions, for brevity we omit the details of those functions and refer the reader to the Dilithium specification for round 2 [DKLL+19].

4.3 Improving speed on Cortex-M4

Our Cortex-M4 implementation is based on the Dilithium implementation by Ravi, Gupta, Chattopadhyay, and Bhasin [RGCB19], which includes the NTT and inverse NTT assembly implementation from Güneysu, Krausz, Oder, and Speith [GKOS18].

In Dilithium, the NTT and inverse NTT are computed iteratively and in-place, such that no auxiliary vectors are required to store intermediate results. For computing the NTT, Dilithium uses such an iterative Cooley–Tukey algorithm, which takes its input vector in normal order, and outputs the vector in bit-reversed order. The inverse NTT is implemented using an iterative Gentleman–Sande algorithm, which takes its input vector in bit-reversed order and returns a vector in normal order. Note that this has no effect on the polynomial-multiplication property (using coefficient-wise multiplication), as described in Section 4.2.

In our implementation similarly to previous work, we precompute and store the twiddle factors in flash. The twiddle factors are stored in the Montgomery domain (with modulus $R = 2^{32}$), such that after the multiplication in the FFT butterfly, we can use Montgomery reduction [Mon85; Sei18] to reduce the product modulo q.

After each level of the NTT and inverse NTT, the polynomial coefficients are growing in size due to additions and subtractions. Intuitively, we would apply a modular reduction after each addition/subtraction operation. However, the coefficients in the input polynomial are bounded by $2q$ (which is only 24 bits) and even if we do not reduce mod q after each level, we will not overflow the 32-bit registers in which we store the coefficients. Therefore, we reduce each coefficient mod q only once, at the end of the NTT and inverse NTT. This technique of delaying the reduction is usually referred to as *lazy reduction*.

When implementing the NTT and inverse NTT, we first unroll the outer loop which iterates over the 8 levels of the NTT and inverse NTT. Furthermore, similar to the merging technique in [GOPS13], we can merge two levels of the NTT and inverse NTT on Cortex-M4 ({0,1}, {2,3}, {4,5} and {6,7}). Merging k layers here means that instead of loading two coefficients, one loads the 2^k coefficients which are used together in k consecutive layers. By doing so one can eliminate the load and store operations between the layers. Hence, the number of layers that can be merged is bounded by the available registers. For our implementation, we achieved the best performance by merging two layers, using 4 registers for the polynomial coefficients and 3 registers for the twiddle factors.[1] As a consequence of the merge, the number of store and load instructions is reduced by a factor of 2.

Lastly, the main difference that distinguishes our implementation from the one published in [GKOS18] is changing the polynomial coefficients to signed representation. When unsigned integers are subtracted from each other, it is possible for the result to wrap around zero (when the result would be negative). To prevent this overflow, the subtractions in the reference implementation are accompanied by an addition with a multiple of q, pushing the results back into the positive domain. By switching to the signed representation, the problem of negative overflows is fixed, and we do not need this extra multiple-of-q addition. Therefore, switching to signed representation allows us to eliminate all these additions throughout the code.

Algorithm 4.1: CT butterfly from [GKOS18]

input: `p0`, `p1`, `twiddle`
output: `p0`, `p1`
let: `q=8380417`, `qinv=4236238847`

```
1 umull tmp0, tmp1, p1, twiddle
2 mul   pol1, tmp0, qinv
3 umlal tmp0, tmp1, p1, q
4 add   p1, p0, q, lsl#1
5 sub   p1, p1, tmp1
6 add   p0, p0, tmp1
```

[1] Accordingly, for k layers—if we do not reload or spill any value—we need 2^k registers for the polynomial coefficients and $\frac{1}{2}k(k+1)$ registers for the twiddle factors.

Algorithm 4.2: Our CT butterfly

input: `p0`, `p1`, `twiddle`
output: `p0`, `p1`
let: `q=8380417`, `qinv=4236238847`

```
smull tmp0, tmp1, p1, twiddle
mul   p1, tmp0, qinv
smlal tmp0, tmp1, p1, q
sub   p1, p0, tmp1
add   p0, p0, tmp1
```

Algorithm 4.3: GS butterfly from [GKOS18]

input: `p0`, `p1`, `twiddle`
output: `p0`, `p1`
let: `q=8380417`, `qinv=4236238847`

```
add   tmp0, p0, q, lsl#8
sub   tmp0, tmp0, p1
add   p0, p0, p1
umull tmp1, p1, tmp0, twiddle
mul   tmp0, tmp1, qinv
umlal tmp1, p1, tmp0, q
```

Algorithm 4.4: Our GS butterfly

input: `p0`, `p1`, `twiddle`
output: `p0`, `p1`
let: `q=8380417`, `qinv=4236238847`

```
sub   tmp0, p0, p1
add   p0, p0, p1
smull tmp1, p1, tmp0, twiddle
mul   tmp0, tmp1, qinv
smlal tmp1, p1, tmp0, q
```

This is especially relevant for the NTT and inverse NTT implementations because every butterfly operation has a subtraction. Algorithm 4.2 shows our improvements to the CT butterfly in the NTT by [GKOS18] which is shown in Algorithm 4.1. For the GS butterflies in the inverse NTT, the improvements are listed in Algorithms 4.3 and 4.4.

However, the overflow-mitigating additions were not only present in the NTT, but also in the sampling of $\mathbf{s_1}$, $\mathbf{s_2}$, and $\mathbf{y}$, polynomial subtraction, and unpacking operations throughout the scheme. By switching to signed representation, we did not only improve the performance of the NTT, but also of all the other routines listed above.

Finally, in addition to improving the NTT and inverse NTT, we rewrote the pointwise polynomial multiplication, uniform sampling of polynomials, and polynomial reduction in assembly as these were the most expensive operations besides the already optimized NTT, inverse NTT, and hashing operations using Keccak. We omit the details, as they result straightforwardly from the reference code.

4.4 Fast Constant-Time NTTs on Cortex-M3

Our constant-time Cortex-M3 implementation of Dilithium is based on the Cortex-M4 implementation described in the previous section. To keep this section concise, we only describe the differences here, which are mainly in order to make the implementation constant-time. When compiling the existing implementation [GKOS18] for the Cortex-M3, we identify three functions that make use of the variable-time instructions `umull` and `umlal`: NTT, NTT^{-1}, and pointwise multiplication ($\circ$). These functions are the only ones that involve the multiplication of the 32-bit coefficients of polynomials. When any of them operates on secret data, it will leak information through a timing side channel.

Previous work by [MGTF19] suggests that the reference implementation of Dilithium is constant time. This is however not true for Cortex-M3, because the compiler is in no way prevented from emitting any of the variable-time instructions. In their paper, the authors propose a modified Dilithium with a power-of-two modulus $q = 2^{32}$ to allow for cheaper masking. As a side-effect of this proposed change, multiplications can be done using `mul`, `mls`, and `mla` as those implicitly wrap their results modulo 2^{32}. In that case, implementing Dilithium in constant-time is more straightforward.

Interestingly, many of the operations within Dilithium do not handle secret data, and, hence, do not need to be constant time. Particularly, all operations in the signature verification (Algorithm 3.17, Verify) are only operating on public data and can, therefore, be implemented in variable time. Similarly, in signature generation (Algorithm 3.17, Sign) $\mathsf{NTT}(\mathbf{t_0})$ (line 15), $\mathsf{NTT}(\mathsf{H}(\mu, \mathbf{w_1}))$ (line 20), and $\mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_\mathbf{0})$ (line 28) are not processing secret data as both $\mathbf{t}$ and c are considered public. For the details we refer to the security proof in [LDKL+19, Section 5]. The remaining calls to NTT, NTT^{-1}, and $\circ$ do process secret data. Similarly, all operations in the key generation of Dilithium (Algorithm 3.17, Sign) have secret inputs. In our implementation we provide both a constant-time and variable-time (`leaktime`) implementation implementations of NTT, NTT^{-1}, and $\circ$. Because the variable-time implementations are significantly faster, we prefer using them over the constant-time implementations when we are only dealing with public data.

Note that, in theory, the compiler could introduce `umull`, `umlal`, `smull`, and `smlal` instructions in other parts of the code as well. Since there is no easy way to prevent compilers (`gcc` and `clang`) from emitting those instructions, we instead carefully analyze the assembly generated by the compiler to not contain these instructions in functions that are safe to leak. We add the suffix `_leaktime` to the names of variable-time functions only operating on public data to support this analysis.

The remainder of this section describes the necessary changes to the Cortex-M4 implementation to ensure it executes in constant-time on the Cortex-M3. We describe the details from the bottom up, i.e., we start with the multiplication of coefficients, continue with the changes to the implementations of the Cooley–Tukey and Gentleman–Sande butterfly operations, and finally describe the changes to the NTT, NTT^{-1} and the rest of the scheme.

4.4.1 `smull` and `smlal`

As Dilithium uses a 23-bit modulus q, its polynomials are usually represented as vectors of 32-bit values. Consequently, multiplying coefficients requires multiplication of 32-bit values producing a 64-bit product. Usually, Montgomery multiplication is used, so that the result is promptly reduced back to 32-bits. In our Cortex-M4 implementation the Montgomery multiplication is computed using `smull` and `smlal`, which—as we

discussed in Section 2.5—execute in variable-time on the Cortex-M3. In case the inputs are secret, we cannot use those instructions.

In general, there are two approaches to address this issue: either re-implement `smull` and `smlal` using available constant-time instructions (`mul`, `mla`, `add`) or using a different representation of polynomials that does not require to multiply 32-bit coefficients. We experimented with the latter approach by using multiple smaller 16-bit polynomial multiplications to construct a larger 23-bit polynomial multiplication. The idea is to perform polynomial multiplications in R_q by first splitting up the polynomial into multiple polynomials in $\mathbb{Z}_{q_i}/(X^n+1)$, performing the polynomial multiplication in these smaller rings and then reconstructing the result in R_q using the explicit Chinese remainder theorem [BS07]. A similar approach is used in the AVX2 implementation of NTRUPrime [BCLv19]. For the result to be correct, it needs to hold that $2n \cdot \lfloor q/2 \rfloor^2 < \prod q_i$. For example, one could use the NTT-friendly primes $\{7681, 10753, 11777, 12289\}$. However, this approach turned out to be slower than re-implementing the `smull` and `smlal` instructions using `mul` instructions, and hence we did not use it in our implementation. Nonetheless, we present results for 16-bit NTTs on the Cortex-M3 for the primes 3329 and 12289 which are used in the NIST key-encapsulation candidates Kyber [ABDK+19] and NewHope [PAAB+19] respectively. We report the results for the full schemes in Appendix 4.A.

To re-implement `smull` and `smlal`, we use the schoolbook approach, i.e., we represent the 32-bit inputs in radix 2^{16} and compute the product as sums of 32-bit products. Let $a = 2^{16}a_1 + a_0$ and $b = 2^{16}b_1 + b_0$, with $0 \leq a_0, b_0 < 2^{16}$ and $-2^{15} \leq a_1, b_1 < 2^{15}$, then the product $ab = 2^{32}a_1b_1 + 2^{16}(a_0b_1 + a_1b_0) + a_0b_0$, with $-2^{31} \leq a_ib_j < 2^{31}$. Accordingly, our constant-time assembly implementations for `smull` and `smlal` are illustrated in Algorithm 4.5 and Algorithm 4.6. We denote them by `SBSMULL` and `SBSMLAL` in the following. The four 16-bit halves of the two multiplicands are passed in the registers a_0, a_1, b_0, and b_1; the 64-bit output is placed in c_0 (lower half) and c_1 (upper half). For `smlal`, c_0 and c_1 initially contain the value to be added to the product. On the Cortex-M3, additions and multiplications use 1 cycle, while `mla` uses 2 cycles. As such, the `SBSMULL` macro takes 7 cycles to execute, while `SBSMLAL` takes 9 cycles.

It is important to note that `SBSMULL` (and `SBSMLAL`) are not semantically equivalent to `smull` (and `smlal`). In case the accumulation $(a_0b_1 + a_1b_0)$ in line 4 of Algorithm 4.5 or line 6 of Algorithm 4.6 overflows, the carry bit is lost and the result will not be

Algorithm 4.5: Schoolbook `smull` (`SBSMULL`)

input: $a = \texttt{a0} + \texttt{a1} \cdot 2^{16}$
input: $b = \texttt{b0} + \texttt{b1} \cdot 2^{16}$
output: $c = ab = \texttt{c0} + \texttt{c1} \cdot 2^{32}$

```
1 mul  c0, a0, b0
2 mul  c1, a1, b1
3 mul  tmp, a1, b0
4 mla  tmp, a0, b1, tmp
5 adds c0, c0, tmp, lsl #16
6 adc  c1, c1, tmp, asr #16
```


Algorithm 4.6: Schoolbook `smlal` (`SBSMLAL`)

input: $a = \texttt{a0} + \texttt{a1} \cdot 2^{16}$
input: $b = \texttt{b0} + \texttt{b1} \cdot 2^{16}$
input: $c = \texttt{c0} + \texttt{c1} \cdot 2^{32}$
output: $c' = c + ab = \texttt{c0} + \texttt{c1} \cdot 2^{32}$

```
1 mul  tmp, a0, b0
2 adds c0, c0, tmp
3 mul  tmp, a1, b1
4 adc  c1, c1, tmp
5 mul  tmp, a1, b0
6 mla  tmp, a0, b1, tmp
7 adds c0, c0, tmp, lsl #16
8 adc  c1, c1, tmp, asr #16
```

correct. Hence, our schoolbook multiplication does not support the full 32-bit range of the inputs. In general, we have to consider two cases:

1. One of the factors (say b) is small, e.g., a twiddle factor ($|r| < \frac{q}{2}$) or the constant q. In that case, b_1 is at most $\left\lfloor \frac{q}{2^{16}} \right\rfloor = 127$. In the worst case, both b_0 and a_0 are equal to $2^{16} - 1$. Consequently, for the addition ($a_0 b_1 + a_1 b_0$) not to overflow, a_1 can be at most $\left\lfloor \frac{2^{31}-1-127\cdot(2^{16}-1)}{2^{16}-1} \right\rfloor = 32641$.
2. Both multiplicands can be equally large. This occurs, for example, in the pointwise polynomial multiplication. In that case, both $a_0 b_1$ and $a_1 b_0$ need to be less or equal to $\left\lfloor \frac{2^{31}-1}{2} \right\rfloor = 2^{30} - 1$ and hence, $a_1, b_1 \leq \left\lfloor \frac{2^{30}-1}{2^{16-1}} \right\rfloor = 2^{14}$.

Case 1 applies in the NTT and NTT^{-1}. In the NTT, the coefficient values never exceed $10q$, which is sufficiently small for the multiplication to remain safe. Similarly, in the NTT^{-1} coefficients never exceed $128q < 32641 \cdot 2^{16}$.

Case 2 applies in the pointwise polynomial multiplication. In that case, the input coefficients are bounded by $10q$ which is comfortably below 2^{30}.

4.4.2 Cooley–Tukey and Gentleman–Sande Butterflies

Algorithm 4.7: Constant-time Cooley–Tukey butterfly on the Cortex-M3

input: `p0` (32-bit signed)
input: `p1` $=$ `p1l` $+$ `p1h` $\cdot 2^{16}$ (`p1l` 16-bit unsigned, `p1h` 16-bit signed)
input: twiddle $=$ `tl` $+$ `th` $\cdot 2^{16}$ (`tl` 16-bit unsigned, `th` 16-bit signed)
output: `p0`, `p1` (32-bit signed)
let: `qinv` $= 4236238847$, $q = 8380417 =$ `ql` $+$ `qh` $\cdot 2^{16}$

```
1  SBSMULL tmpl, tmph, p1l, p1h, tl, th  ; (tmpl, tmph) := (p1l, p1h) · twiddle
2  mul p1h, tmpl, qinv
3  ubfx p1l, p1h, #0, #16
4  asr p1h, p1h, #16
5  SBSMLAL tmpl, tmph, p1l, p1h, ql, qh  ; (tmpl, tmph) := (tmpl, tmph) + (p1l, p1h) · q
6  sub p1, p0, tmph
7  add p0, p0, tmph
```

Using constant-time `SBSMULL` and `SBSMLAL` subroutines, we can construct the butterfly operations needed to implement the NTT and NTT^{-1}. Algorithm 4.7 depicts

the modified Cooley–Tukey butterfly operation based on Algorithm 4.2. To be able to use SBSMULL, p_1 and the twiddle factor need to be loaded in half-words, while p_0 can be loaded as a 32-bit word. For the multiplication by q, we require to have the lower and the upper half-word of q separately. Additionally, we need to split up the 32-bit result of the multiplication by $-q^{-1}$ into half-words (lines 3 and 4). In total, the Cooley–Tukey butterfly operation requires 21 cycles on the Cortex-M3, while Algorithm 4.2 only needs 5 cycles on the Cortex-M4.

Algorithm 4.8: Constant-time Gentleman–Sande butterfly on the Cortex-M3

input: p0, p1 (32-bit signed)
input: twiddle = tl + th · 2^{16} (tl 16-bit unsigned, th 16-bit signed)
output: p0, p1 (32-bit signed)
let: qinv = 4236238847, q = 8380417 = ql + qh · 2^{16}

```
1  sub tmp, p0, p1
2  add p0, p0, p1
3  ubfx tmpl, tmp, #0, #16
4  asr tmph, tmp, #16
5  SBSMULL tmp, p1, tmpl, tmph, tl, th   ; (tmp, p1) := (tmpl, tmph) · twiddle
6  mul tmph, tmp, qinv
7  ubfx tmpl, tmph, #0, #16
8  asr tmph, tmph, #16
9  SBSMLAL tmp, p1, tmpl, tmph, ql, qh   ; (tmp, p1) := (tmp, p1) + (tmpl, tmph) · q
```

Similarly, Algorithm 4.8 depicts our constant-time assembly implementation of the Gentleman–Sande butterfly. As the addition and subtraction happens before the multiplication by the twiddle factor, both p_0 and p_1 are loaded as full 32-bit words, while the twiddle factor is again split into two half words. After the subtraction in line 1, we split up the result before we pass it into SBSMULL. To perform the Montgomery reduction, we again need the split up the result of the multiplication by $-q^{-1}$ into halves, before multiplying it by q using SBSMLAL. Each Gentleman–Sande butterfly operation requires 23 cycles on the Cortex-M3 which compares to 5 cycles for Algorithm 4.4 on the Cortex-M4.

4.4.3 NTT, NTT^{-1}, and $\circ$

Using the Cooley–Tukey butterfly from the previous section, we implement the NTT. Similar to in the Cortex-M4 implementation, we pre-compute all the twiddle factors and place them into flash. As our Cooley–Tukey butterfly requires the second coefficient and the twiddle factor in halves, we load those using `ldrh` (for the unsigned lower half-word) and `ldrsh` (for the signed upper half-word). This, however, significantly increases register pressure and hinders the common optimization technique of merging multiple levels of butterfly operations with the purpose of saving store and load instructions. Therefore, we can not use that optimization and need to perform one layer at a time. This also leads to a slightly different ordering of the twiddle factors in memory. The results of the butterfly are returned as 32-bit values and can, hence, be stored back using `str`.

For the NTT^{-1}, we proceed likewise. However, the inputs to the butterfly have to be loaded in full-words using `ldr`. At the end of the NTT^{-1}, each coefficient of the polynomial is multiplied with the constant n^{-1} followed by a Montgomery reduction. We integrate this step into the last level of the NTT^{-1} in order to minimize load and store operations. Furthermore, we observe that n^{-1} in Montgomery domain is 41 978 and, hence, less than 16-bits. Therefore, we do not need a full `SBSMULL` but can use a simpler multiplication routine that multiplies a 32-bit word by the 16-bit constant which requires 2 multiplication instructions and, hence, 2 cycles less.

Besides the NTT and NTT^{-1} we identify one other place where our compiler is introducing `smull` and `smlal` instruction: The pointwise multiplication $\circ$. If either of the multiplicands is secret, the pointwise multiplication must not use the variable time instructions. We guarantee that by rewriting the pointwise multiplication in assembly and making use of the Montgomery multiplication using `SBSMULL` and `SBSMLAL` like in our butterfly operation in Algorithm 4.7 and Algorithm 4.8. In case both inputs are considered public, we use the pointwise multiplication which was presented in Section 4.3.

4.5 Results

This section presents the performance results for our Dilithium implementations. First, we present new speed records for the Dilithium NTT on the Cortex-M4 and first results for the Dilithium, Kyber, and NewHope NTT in Section 4.5.1. We then present results for the full Dilithium scheme on the Cortex-M4 (Section 4.5.2) and on the Cortex-M3 (Section 4.5.3). Finally, we profile our implementations on the Cortex-M4 in Section 4.5.4.

Cortex-M4 setup. We benchmark all our Cortex-M4 implementations on an STM32-F407 Discovery board, which features the STM32F407VG microcontroller. It was clocked at 24 MHz to eliminate flash wait states when fetching instructions or data from flash. For benchmarking the algorithm latency we used the SysTick counter. Our build and benchmarking setup is based on pqm4 [PQM4] and benchmarking our code within pqm4 gives the same performance results. After review of our work, we have merged our code into pqm4, where it has remained until it was superseded by the new work of Chapter 5.

Cortex-M3 setup. The Cortex-M3 measurements were done on an Arduino Due board which uses the ATSAM3X8E microcontroller. The ATSAM chip was clocked at 16 MHz, which results in a flash access time with zero wait-states. The algorithm latencies were measured using the internal cycle counter (CYCCNT).

Compiler, random numbers, stack measurements, and Keccak. For all measurements, we used the GCC compiler, version `10.2.0`. For obtaining random numbers (e.g., ρ and K), we use the hardware random number generators which are available on both cores. The stack usage was measured by filling the memory with sentinel values, executing the algorithm, and measuring the amount of sentinel-value bytes that were overwritten during the execution. In the stack measurements, space reserved for input and output values is not counted. For `SHA3` and `SHAKE`, we use the assembly-optimized implementation of the Keccak permutation from the eXtended Keccak Code Package (XKCP).[2] As it only uses `ARMv7-M` instructions, we use the same implementation on both platforms.

[2] `https://github.com/XKCP/XKCP`

Table 4.2: Performance results for NTT, NTT^{-1}, and $\circ$ of Dilithium, Kyber, and NewHope for the Cortex-M3 and the Cortex-M4 reported in clock cycles. The Cortex-M3 (SAM3X8E) is running at 16MHz, and the Cortex-M4 (STM32F407) is running at 24 MHz.

				NTT	NTT^{-1}	$\circ$
Dilithium[a]	[GKOS18]	constant-time	M4	10 701	11 662	–
	This work	constant-time	M4	8 540	8 923	1 955
	This work	variable-time	M3	19 347	21 006	4 899
	This work	constant-time	M3	33 025	36 609	8 479
Kyber[b]	[ABCG20]	constant-time	M4	6 855	6 983	2 325
	This work	constant-time	M3	10 819	12 994	4 773
NewHope1024[c]	[ABCG20]	constant-time	M4	68 131	51 231	6 229
	This work	constant-time	M3	77 001	93 128	18 722

[a] $n = 256$, $q = 8380417$ (23 bits), 8 layer $\text{NTT}/\text{NTT}^{-1}$
[b] $n = 256$, $q = 3329$ (12 bits), 7 layer $\text{NTT}/\text{NTT}^{-1}$
[c] $n = 1024$, $q = 12289$ (14 bits), 10 layer $\text{NTT}/\text{NTT}^{-1}$

4

4.5.1 NTT performance

In Table 4.2, we list the benchmarking results for the optimized NTT, NTT^{-1}, and pointwise multiplications ($\circ$) implementations in Dilithium, Kyber, and NewHope1024 on the Cortex-M3 and Cortex-M4. For the Cortex-M4, we obtain a speedup of 20% and 23% for the NTT and NTT^{-1} compared to [GKOS18; RGCB19]. This speedup is mainly due to the switch to a signed representation of polynomials. We use this representation throughout our new Dilithium implementations, which saves a number of additions of multiples of q. Additionally, we optimize the pointwise multiplication ($\circ$) which was not optimized in previous implementations.

In the Cortex-M3 results, we first benchmark the implementation also used on the Cortex-M4 which uses `smull` and `smlal`. As `smull` and `smlal`, but also `mla`, need significantly more cycles on the Cortex-M3 (respectively 3–5, 4–7, and 2 on the M3 vs. 1 on the Cortex-M4), the cycle counts for NTT, NTT^{-1}, and $\circ$ increase between 2.3× and 2.5×. Making those constant-time on the Cortex-M3 using `SBSMULL` and `SBSMLAL` from Section 4.4.1 increases the number of cycles by a factor of 1.7.

4.5.2 Cortex-M4 performance

Table 4.3: Performance results on the Cortex-M4 (STM32F407 at 24 MHz). Averaged over 10 000 executions. From [RGCB19], "scenario 1" was used for comparison, as it corresponds to the format that was used in this work.

Algorithm	Params	Work	Speed [kcc]
Keygen	Dilithium2	**This work**	1 315
	Dilithium3	[GKOS18]	2 320
	Dilithium3	**This work**	2 013
	Dilithium4	**This work**	2 837
Sign	Dilithium2	[RGCB19]	4 632
	Dilithium2	**This work**	3 987
	Dilithium3	[GKOS18]	8 348
	Dilithium3	[RGCB19]	7 085
	Dilithium3	**This work**	6 053
	Dilithium4	[RGCB19]	7 061
	Dilithium4	**This work**	6 001
Verify	Dilithium2	**This work**	1 259
	Dilithium3	[GKOS18]	2 342
	Dilithium3	**This work**	1 917
	Dilithium4	**This work**	2 720

Table 4.3 lists the benchmarking results of our Dilithium implementation, together with the cycle counts from the relevant related work. As signing time varies considerably depending on the number of rejections, we performed 10 000 executions and took the average of the resulting cycle counts. Compared to the [GKOS18] implementation, we obtain speedups of 13%, 27%, and 18% for key generation, signing, and verification respectively for Dilithium3. When comparing to the [RGCB19, scenario 1] implementation, the acieved signing speedup is 15%.

4.5.3 Cortex-M3 performance

Table 4.4 presents our results for the Cortex-M3. The only other work implementing (a modified version of) Dilithium on the Cortex-M3 is from Migliore, Gérard, Tibouchi, and Fouque [MGTF19]. However, they do not report cycle counts on the Cortex-M3,

Table 4.4: Performance results on the Cortex-M3 (SAM3X8E at 16 MHz). Averaged over 10000 executions.

Algorithm/ strategy	Params	Speed [kcc]
KeyGen	Dilithium2	1 699
	Dilithium3	2 562
	Dilithium4	3 587
Sign	Dilithium2	7 115
	Dilithium3	10 667
	Dilithium4	10 031
Verify	Dilithium2	1 541
	Dilithium3	2 321
	Dilithium4	3 260

and we were not able to find their source code online. Therefore, we can unfortunately not compare our results to theirs.

4.5.4 Profiling

To identify how much is still left to optimize in our implementation, we profiled the implementations on the Cortex-M4. Table 4.5 contains the profiling results. We see that the run-time of the scheme is mostly dominated by Keccak. The proportion of cycles spent in hashing is up to 85% for key generation, 63% for signing, and 81% for verification, which greatly limits the speedup achievable by further optimizing the arithmetic of the scheme. This result clearly indicates that hardware acceleration for SHA3 will essentially be a prerequisite for getting better-performing Dilithium implementations when using Cortex-M4 or Cortex-M3 cores.

Only about 3.4% to 24.5% of cycles are spent in the NTT and inverse NTT. Another 3.9% to 13.2% of cycles are spent in the other assembly-optimized functions which are pointwise multiplication, uniform sampling, and modular reduction. The time spent in non-optimized C code is consistently relatively small. Hence, optimizing the remaining code is not going to provide a large speedup. When looking at individual functions of the non-optimized code, no function takes more than 3% of the total run-time.

Table 4.5: Profiling results on the Cortex-M4

	Operation	Keygen	Sign	Verify
Dilithium2	Keccak	81.4%	55.4%	76.6%
	NTT	1.9%	7.2%	5.4%
	NTT^{-1}	2.7%	11.7%	2.8%
	other asm	6.2%	9.3%	6.8%
	not opt.	**7.8%**	**16.3%**	**8.4%**
Dilithium3	Keccak	82.8%	63.7%	79.1%
	NTT	1.7%	6.8%	4.4%
	NTT^{-1}	2.2%	8.6%	2.3%
	other asm	6.4%	8.4%	7.0%
	not opt.	**6.9%**	**12.5%**	**7.2%**
Dilithium4	Keccak	84.2%	61.8%	80.9%
	NTT	1.5%	6.2%	3.7%
	NTT^{-1}	1.9%	9.2%	1.9%
	other asm	6.6%	10.0%	7.1%
	not opt.	**5.8%**	**12.9%**	**6.3%**

4.A Kyber and NewHope on Cortex-M3

As a side-product of Section 4.4, we present implementations for the NTT and NTT^{-1} operations for the primes 3329 and 12289. While those did not allow us to speed up our Dilithium implementation further, they can be used to implement the key-encapsulation mechanisms Kyber and NewHope on the Cortex-M3 in constant-time. We report the results for these schemes here. Our implementations of both Kyber and NewHope are based on the implementations by Alkim–Bilgin–Cenk–Gérard [ABCG20]. As those implementations make heavy use of instructions not available on the Cortex-M3 (e.g., SIMD instructions like `uadd16`, or multiplication instructions like `smlabb`), these are not directly functional on the Cortex-M3.

In addition to the NTT and NTT^{-1} implementations, we further port the other assembly routines to Cortex-M3. For Kyber this includes polynomial addition, polynomial subtraction, Barrett reduction, and base multiplication. For NewHope, we use the same approach as [ABCG20], and use the Cooley–Tukey algorithm [CT65] for NTT and the Gentleman–Sande algorithm [GS66] for NTT^{-1}. Beside that, we port the code for polynomial addition, pointwise multiplication, and bit-reversal to

Table 4.6: Kyber and NewHope results on the Cortex-M3 (SAM3X8E at 16 MHz) compared to the fastest Cortex-M4 implementation. Average of 100 executions.

		Platform	Keygen [kcc]	Encaps [kcc]	Decaps [kcc]
Kyber512	[ABCG20]	Cortex-M4	455	586	544
	This work	Cortex-M3	539	682	652
Kyber768	[ABCG20]	Cortex-M4	864	1 033	970
	This work	Cortex-M3	1 012	1 194	1 145
Kyber1024	[ABCG20]	Cortex-M4	1 405	1 606	1 526
	This work	Cortex-M3	1 636	1 853	1 793
NewHope1024-CCA	[ABCG20]	Cortex-M4	1 157	1 675	1 587
	This work	Cortex-M3	1 239	1 921	1 888

Cortex-M3. We present the results for both NewHope and Kyber in Table 4.6. The slow-down compared to the Cortex-M4 implementation is between 7% and 20% and as such it is not as significant as for the Dilithium implementations. However, it does demonstrate the limitations of the Cortex-M3.

5 NTT optimizations on Cortex-M4

5.1 Introduction

At the point of writing, the NIST PQC standardization process is nearing the end of its third round with announcements due in early 2022. Since Chapter 4, Dilithium advanced to the finalists; its specification has been updated to a third version, and its parameter sets have seen considerable modifications. Among the other finalists in the competitions are four other lattice-based schemes including the three key-encapsulation mechanisms (KEMs) Kyber, NTRU, and Saber and the competing digital signature scheme Falcon. As there are only two other finalists (Classic McEliece and Rainbow) that are not lattice-based, which both have excessively large keys, it appears very likely that some of the lattice-based schemes are going to be selected for standardization unless there are cryptanalytic breakthroughs.

It appears that the number-theoretic transforms are the main optimization target of all high-speed implementations of lattice-based crypto for the Cortex-M4. It is either prescribed in the specification of Dilithium, Falcon [PFHK+22], and Kyber [SABD+22], or maintains to be the fastest polynomial multiplication methods in Saber, NTRU [CHKS+21], and NTRU Prime [ACCE+20].

Moreover, as we have seen in Chapter 4, the performance of the NTT greatly determines the performance of a Dilithium implementation. More so, we expect that when acceleration for the Keccak family (i.e., SHA3/SHAKE) becomes mainstream in microcontrollers, most of the computation will be spent computing the (inverse) NTT.

In this work, we leave Cortex-M3, and focus on improving Dilithium on the Cortex-M4. We show that even though implementations have been improving for many years, we can still significantly improve the involved arithmetic.

5.1.1 Contributions

First, in Section 5.4, we observe that in Dilithium we can optimize the computation of $c\mathbf{s_1}$ and $c\mathbf{s_2}$. Since both c and $\mathbf{s_1}$,$\mathbf{s_2}$ have very small absolute values, we can switch to a much smaller modulus q' that allows efficient computation of the product. For Dilithium2 and Dilithium5, we make use of the Fermat prime $q' = 257$, which allows using a particularly fast variant of the NTT called the Fermat number transform (FNT), similar to [LMPR08] for SWIFFT. Furthermore, [LMPR08] implements an FNT on an Intel processor while we implement the FNT on the Cortex-M4 and make use of its barrel shifter. For Dilithium3, the FNT does not work, as its value $\beta = 196$ is too large. We instead use an incomplete NTT with $q' = 769$ which is still much faster than computing the NTT modulo the original Dilithium q. To best of our knowledge, we are the first to propose using a smaller modulus for these multiplications within Dilithium.

The new NTT modulo the smaller $q' = 769$ is very similar to the NTT used in Kyber (which uses $q = 3329$). Therefore in Section 5.3, we can learn from techniques from previous work on the Cortex-M4 optimizing Kyber, Saber, NTRU, and NTRU Prime, and integrate them into both the "big" NTT (modulo the big q that is used in Dilithium), as well as the "small" NTTs (which is used in Kyber and Dilithium3). While the techniques are already known, they have so far not been applied to Dilithium. This includes (1) the use of Cooley–Tukey butterflies for the inverse NTT previously proposed for Saber in [ACCH+22]; (2) the use of floating point registers for caching values in the NTT which was first proposed in the context of NTTs for NTRU Prime in [ACCE+20]. This allows one to merge more layers of the NTT and reduce memory access time for loading twiddle factors; and (3) we make use of the "asymmetric multiplication" proposed in [BHKY+21] which eliminates some redundant computations in the base multiplication of the small NTTs at the cost of extra stack usage.

In Section 5.5, we present the resulting implementation. We measure the performance results using the pqm4 [PQM4] framework and compare them to previous work.

Code. The implementations of Kyber and Dilithium that are the result of this work are in the public domain and can be obtained by following the instructions described on page 10. The code is published and licensed under a CC0 copyright waiver.

5.2 Preliminaries

5.2.1 Fermat Number Transform

The *Fermat number transform* (FNT) is a special case of NTT in that the modulus is a Fermat number $F_t := 2^{2^k} + 1$. It was introduced in [SS71] for large integer multiplications and in [AB74; AB75] for digital convolutions. In this chapter, we implement FNT for negacyclic convolution. For arbitrary F_k as the modulus, cyclic transformations of sizes dividing 2^{k+2} are supported [AB74; AB75]. For computing a negacyclic transformation of size $n = 2^{k+1}$ and $\zeta_{2n} = \sqrt{2}$, the first split becomes

$$\begin{aligned}\mathbb{Z}_{F_k}[X]/(X^n - 2^{2^k}) \quad &\cong \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{k-1}}) \times \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} + 2^{2^{k-1}}) \\ &= \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{k-1}}) \times \mathbb{Z}_{F_t}[X]/(X^{\frac{n}{2}} - 2^{2^{k-1}(1+2)}).\end{aligned}$$

After applying k layers, all of the polynomial rings are of the form $\mathbb{Z}_{F_t}[x]/(X^{\frac{n}{2^k}} - 2^j)$ where j is an odd number. Since $\zeta_{2n}^2 = 2$, we can apply one more split. Furthermore, if F_k is a prime, then we can compute cyclic transformations of sizes up to $2^{2^k} = F_k - 1$ and negacyclic transformations up to 2^{2^k-1}. Since the twiddles in initial k layers are powers of two, we can multiply with the twiddles using shift operations which is much cheaper than explicit multiplications on many platforms. Note that the only known prime Fermat numbers are $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65\,537$. Out of those, only F_3 and F_4 appear promising for the use in Dilithium. They allow to compute 3 or 4 layers using only shifts.

5.3 Improvements to the NTT

5.3.1 FPU registers & improved layer merging

In the first six layers of the Dilithium NTT, each time 7 twiddle factors are required and re-used multiple times throughout the iterations. By using the floating-point registers for caching the twiddle factors, the number of cycles for memory loads are reduced. This technique has been proven to be beneficial in past work [ACCE+20; ACCH+22; CHKS+21]. In our implementations, we load the 7 twiddle factors into 7 floating-point registers once with `vldm` instruction in 8 cycles. Then, in each iteration

the twiddle factors are fetched from the floating-point registers with `vmov` in a single cycle each. This improves what we did in Chapter 4, where many twiddle-factors take two cycles to load. For the last two layers, it is not beneficial to make use of the floating point registers as none of the twiddle factors are re-used.

Aside from reducing the amount of cycles used to load the twiddle factors in the NTT, the usage of the FPU registers reduces the pressure on the Cortex-M4 general-purpose registers. Therefore, we can improve the merging pattern of Chapter 4, where we merged the layers by layers 1–2, 3–4, 5–6, 7–8. Instead we implement the NTT by merging layers 1–3 and 4–6 (while and 7–8 remains unchanged). By changing the number of layer groups from 4 to 3 we reduce the number of coefficient load and store operations with 25%.

5.3.2 Switch to CT-butterflies

In previous implementations of Kyber and Dilithium for the Arm Cortex-M4, the NTT was always implemented using CT butterflies, while the inverse NTT was implemented using GS butterflies, which is a commonly seen pattern for implementations using the NTT in general. Opposed to that, we implement the inverse NTT using CT butterflies to limit the coefficients' growths, as for example suggested in [Sei18, Section 2.1] or implemented for Saber in [ACCH+22]. In the Dilithium NTT, this completely removes the need for any additional intermediate coefficient reductions. Using CT butterflies for the inverse NTT requires to do additional twisting during the computation of the last layer, but the total number of multiplications does generally not increase because multiplications in the same amount can be omitted during the butterfly operations ("light butterflies").

Further, we make use of a technique introduced in [ACCH+22, Appendix D] which computes light butterflies with one less Montgomery reduction.

5.4 Small NTTs for Dilithium

In the signature generation of Dilithium, we recall that the polynomial c consists of τ coefficients that are ± 1s and $256 - \tau$ coefficients that are 0, and all polynomials in $\mathbf{s_1}$ and $\mathbf{s_2}$ consist of coefficients in $[-\eta, \eta]$. The absolute values of the coefficients in $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are bounded by $\tau \cdot \eta = \beta$, and the computation can be regarded as in $\mathbb{Z}_{q'}$ for

$q' > 2\beta$ [CHKS+21, Section 2.4.6]. As far as we know, all previous implementations choose $q' = 8380417$ and employ the NTT defined for Dilithium. However, since only the correct $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are required, there is some freedom for choosing q'. The bound β is {78,196,120} for Dilithium{2,3,5} respectively. Consequently, we choose the Fermat number $q' = F_3 = 257$ for Dilithium2 and Dilithium5, and $q' = 769$ for Dilithium3. Alternatively, one can also re-use the Kyber prime $q' = 3329$ for any of the parameters in case re-using the code is of interest. We have also experimented with the Fermat number $q' = F_4 = 65537$ for Dilithium3. However, this did not result in in a speedup compared to $q' = 769$.

5.4.1 FNT for Dilithium2 and Dilithium5

For $q' = 257 = 2^8 + 1$, we have FNT defined over $\mathbb{Z}_{257}[X]/(X^{256} + 1)$. We implement the forward transformation with 7 layers of CT butterflies. Since the input coefficients for c, $\mathbf{s_1}$, and $\mathbf{s_2}$ are at most in $[-\eta, \eta]$, we only need very few reductions. Recall that a CT butterfly maps (a, b) to $(a + \omega b, a - \omega b)$ (where ω is the twiddle factor), so we can implement it with `mla` and `mls`. Furthermore, we can also take a closer look at the initial layers. Since $-1 \equiv 2^8 \pmod{257}$, the first layer can be written as $\mathbb{Z}_{257}[X]/(X^{256} + 1) \cong \mathbb{Z}_{257}[X]/(X^{128} - 2^4) \times \mathbb{Z}_{257}[X]/(X^{128} + 2^4)$ and the corresponding CT butterfly maps (a, b) to $(a + 2^4 b, a - 2^4 b)$. We denote such computation as `CT_FNT`$(a, b, 4)$. Notice that without loading twiddle factors, we can implement `CT_FNT`$(a, b,$ `logW`$)$ efficiently using the barrel shifter as illustrated in Algorithm 5.2.

Algorithm 5.1: CT butterfly with small ω

Input: $($`a`, `b`$) = (a, b)$
Output: $($`aout`, `bout`$) = (a + \omega b, a - \omega b)$

```
1: mla aout, b, ω, a
2: mls bout, b, ω, a
```

Let FNT $^{-1}$ be the inverse of FNT. We first observe that the inverse of 2^k can be written as $2^{-k} \equiv 2^{16-k} \equiv -2^{8-k} \pmod{2^8 + 1}$. There are two places where we need to multiply by an inverse of a power of two: (i) the inverses corresponded to the butterflies with $\omega = 2^{\texttt{logW}}$ in `CT_FNT`, and (ii) the scaling by 128^{-1} at the end of FNT $^{-1}$. We denote `CT_iFNT`$(a, b,$ `logW`$)$ as the function mapping (a, b) to $(a - 2^{\texttt{logW}} b, a + 2^{\texttt{logW}} b) = (a + 2^{8+\texttt{logW}} b, a - 2^{8+\texttt{logW}} b)$ and implement it with the barrel

shifter as shown in Algorithm 5.3. Clearly, if CT_FNT(a, b, k) computes $(a+2^k b, a-2^k b)$, then CT_iFNT$(a, b, 8-k)$ computes $(a+2^{-k}b, a-2^{-k}b)$ which can be used in FNT $^{-1}$. We compute FNT $^{-1}$ with four layers of GS butterflies followed by three layers of CT butterflies. During the GS butterflies, since the twiddle factors are also very small, we can replace some of the `mul`, `add`, and `sub` with `mla` and `mls`. For CT butterflies, since the twiddle factors are powers of two, we implement them with Algorithm 5.3. Lastly, at the end of CT butterflies, we merge the twisting by powers of two with the multiplication by 128^{-1}.

Algorithm 5.2: CT_FNT$(a, b,$ logW$)$

input: $(\mathsf{a}, \mathsf{b}) = (a, b)$
output: $(\mathsf{a}, \mathsf{b}) = (a + 2^{\mathsf{logW}}b, a - 2^{\mathsf{logW}}b)$

```
1: add a, a, b, lsl #logW
2: sub b, a, b, lsl #(logW+1)
```

Algorithm 5.3: CT_iFNT$(a, b,$ logW$)$

input: $(\mathsf{a}, \mathsf{b}) = (a, b)$
output: $(\mathsf{a}, \mathsf{b}) = (a - 2^{\mathsf{logW}}b, a + 2^{\mathsf{logW}}b)$

```
1: sub a, a, b, lsl #logW
2: add b, a, b, lsl #(logW+1)
```

5.4.2 NTT over 769 for Dilithium3

For Dilithium3, since the maximum absolute value of $c\mathbf{s_1}$ and $c\mathbf{s_2}$ is bounded by $\beta = 4 \cdot 49 = 196$, we cannot use $q' = 257 < 2 \cdot 196$. We therefore choose $q' = 769$, which is the next prime for which a 256th primitive root of unity exists. We use the 16-bit NTT and NTT^{-1} from Kyber ([AHKS22, Section 3.1]), but we remove most of the Barrett reductions.

In the NTT, we do not need any intermediate Barrett reductions. Moreover, since we are using a 16-bit NTT for computing $c\mathbf{s_1}$ and $c\mathbf{s_2}$, we can remove the Barrett reductions at the end and allow elements growing up to $7q'$ in absolute value.

For the NTT^{-1}, replacing with $q' = 769$ allows us to postpone the Barrett reductions by one layer and cut the number of Barrett reductions by half. At the end of NTT^{-1}, we replace the 16-bit Montgomery multiplication with straight multiplication and 32-bit Barrett reduction. By using 32-bit Barrett reduction, the result is within $[-384, 384]$ if the product is in $[-113025697, 113025697]$. Since $\log_2(\frac{113025697}{384}) \approx 18.17$, we derive values in $[-384, 384]$ by applying 32-bit Barrett reduction to the product of any signed 16-bit value and any constant from $[-384, 384]$. The downside for using 32-bit Barrett reduction is a slightly higher register pressure, but overall it is more favorable because we don't need to reduce them again. This is different from the 16-bit NTT

in [ACCH+22]. They implemented the twist with Montgomery multiplication and then reduced the result to $[-384, 384]$ with an additional 32-bit Barrett reduction.

5.4.3 Asymmetric Multiplication

In Dilithium, $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are computed as $\mathsf{NTT}^{-1}(\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_x}))$. During each rejection-sampling loop $\mathbf{s_1}$ and $\mathbf{s_2}$ remain unchanged, so usually their NTT representation is precomputed before entering the rejection-sampling loop.

The new small-q' NTTs from Section 5.4, are incomplete, i.e., 7 instead of 8 layers are computed, and therefore the product of two polynomials inside NTT-domain $\hat{u} = \hat{c} \circ \hat{s}$ consists of 128 2×2 schoolbook multiplications. For computing $\hat{u}_{2i} + \hat{u}_{2i+1}X = (\hat{c}_{2i} + \hat{c}_{2i+1}X)(\hat{s}_{2i} + \hat{s}_{2i+1}X) \bmod (X^2 - \omega_i)$, we have $\hat{u}_{2i} = \hat{c}_{2i}\hat{s}_{2i} + \hat{c}_{2i+1}\hat{s}_{2i+1}\omega_i$ and $\hat{u}_{2i+1} = \hat{c}_{2i}\hat{s}_{2i+1} + \hat{s}_{2i}\hat{c}_{2i+1}$ (where ω_i is the relevant twiddle factor).

The computation of $c\mathbf{s_1}$ and $c\mathbf{s_2}$ visits $\ell + k$ polynomials in $\mathbf{s_1}, \mathbf{s_2}$, which means that the computation of $\hat{c}_{2i+1}\omega_i$ (or the computation of $\hat{s}_{2i+1}\omega_i$) is repeated $\ell + k$ times. This can be avoided by caching the intermediate results of $\hat{c}_{2i+1}\omega_i$ in a separate vector $\hat{c}'$ [BHKY+21, Section 4.2].

5.5 Results

Our benchmarking setup is based on pqm4 [PQM4] and follows the methodology from Section 2.6.2. During the benchmarks, we clock the microcontroller at 24 MHz in order to avoid wait states during memory operations. We compile the code using `arm-none-eabi-gcc` version 10.2.1 with the `-O3` option. Regarding the Keccak implementation, we make use of the code provided in pqm4. For the randomness generation we rely on the microcontroller's hardware rng.

We compare our implementations of Dilithium{2,3} to the code in pqm4 which is based on [GKS21] (i.e., Chapter 4). For Dilithium5, pqm4 does not currently have an implementation due to a lack of stack space. We apply some of the stack optimizations of [GKS21] to our implementations, especially to make Dilithium5 work as well. It is important to note that the parameters of Dilithium were changed at the start of the third round of the NISTPQC competition. The numbers presented here reflect the round 3 versions contained in pqm4. Those are optimizations from the original

papers ported to the third round parameters. The performance results for the full schemes do not match the original publications.

5.5.1 Performance of NTT-related functions

In Table 5.1, we present the cycle counts for the transformations we deploy in our implementation of Dilithium. We achieve a speedup of 5.2% for the Dilithium NTT, and 5.7% for the NTT^{-1}. For the small NTTs the metric we are optimizing is $(k + l) \cdot \mathsf{NTT} + \#\text{reps} \cdot (\mathsf{NTT} + (k + l) \cdot (\text{basemul} + \mathsf{NTT}^{-1}))$. As most of the small NTT are computed outside of the loop, we moved some of the reductions into the NTT resulting in a faster basemul. Note that for $q = 257$ and $q = 769$ the NTT and NTT^{-1} have very close performance, but the basemul differs. This results in the FNT being advantageous for Dilithium2 and Dilithium5. For (basemul + NTT^{-1}), we achieve a speedup of 37.6% for $q = 257$, and 33.1% for $q = 769$ compared to $q = 8380417$ from [GKS21]. We also compare our $q = 769$ implementation to an existing one by [ACCH+22], because theoretically, their 6-layer approach could also be used as well. Since the computation is dominated by (basemul + NTT^{-1}), we find that our 7-layer approach is faster. We also carefully examine the code by [ACCH+22], and find that the last 32-bit Barrett reduction is performed outside the reported NTT^{-1}, so the speedup is more.

Table 5.1: Cycle counts for transformation operations of Kyber and Dilithium. NTT and NTT^{-1} correspond to the schemes default transformations, i.e., $q = 3329$ for Kyber and $q = 8380417$ for Dilithium. The NTT with $q = 257$ is deployed for Dilithium2 and Dilithium5, and the NTT with $q = 769$ is used for Dilithium3.

	prime	implementation	NTT	NTT^{-1}	basemul
Kyber	$q = 3329$	[ABCG20]	6 852	6 979	2 317
		[AHKS22][a]	5 992	5 491	1 613[b]
Dilithium	$q = 8380417$	[GKS21]	8 540	8 923	1 955
		This work	8 093	8 415	1 955
	$q = 257$	**This work**	5 524	5 563	1 225
	$q = 769$	[ACCH+22] (6-layer)	4 852	4 817	2 966
		This work	5 200	5 537	1 740

[a] Result from the published paper corresponding to this chapter.
[b] Asymmetric basemul as used in the IP (enc). As the basemul in the MVP and IP consists of individual function calls, the cycle count is not straightforward to measure.

5.5.2 Performance of the full scheme

Table 5.2 contains the speed performance results for Dilithium. We achieve consistent speedups for all parameter sets. The absolute savings due to our optimizations are clearly seen, particularly in signing. The speedup for signing ranges from 1.5% to 5.6%.

In relative terms, the impact of our optimizations on the scheme seems relatively small compared to the speedups we gain for the polynomial arithmetic. This is due to dominance of the hashing operations as thoroughly analyzed in previous work [PQM4].

Table 5.2: Cycle counts and stack usage for Dilithium. K, S, and V correspond to the key generation, signature generation, and signature verification. Cycle counts are averaged over 10 000 executions.

implementation		Dilithium2		Dilithium3		Dilithium5	
		kcc	stack [kB]	kcc	stack [kB]	kcc	stack [kB]
pqm4, [GKS21]	K	1 602	38	2 835	61	4 836	98
	S	4 336	49	6 721	74	9 037	115
	V	1 579	36	2 700	58	4 718	93
This work	K	1 596	8 508	2 827	9 540	4 829	11 696
	S	4 093	49	6 623	69	8 803	116
	V	1 572	36	2 692	58	4 707	93

5

6 Dilithium for memory-constrained devices

6.1 Introduction

Dilithium signing has two main practical drawbacks for embedded devices. The first one is the variable signing time, which follows a geometric distribution. When using the parameter set targeting NIST security level 3, the probability that the signing time is more than twice the expected average is approximately 14 percent. This is significant and will have a real impact on many performance requirements for various use-cases. The second drawback relates to the memory requirements which are significantly higher for virtually all PQC schemes compared to the classical digital signature counterparts. This can not only attributed to relatively large key and signature sizes, but also heavy use of stack space for the storage of intermediate data. For example, the embedded benchmarking platform *pqm4* [KRSS19; PQM4] (which executes on the Arm Cortex-M4) initially reported memory requirements for Dilithium in the range of 50–100 KiB for the original reference as well as the optimized implementations.

Dilithium has received a significant amount of attention from the cryptographic community. One direction of study comes from an applied cryptographic engineering perspective: how can one realize efficient implementations in practice for a selected target platform. Often the single most important optimization criterion is latency: the algorithm needs to execute as fast as possible, at the possible expense of other important metrics. Examples include the AVX2-based implementations from [DKLL+18] and [FK19]; or the implementation from [RGCB19], which requires up to 175 KiB of memory; or the implementations from Chapters 4 and 5.

Instead, in this chapter we target platforms that have significantly less memory and computational power. Typical examples are platforms which are based on Arm Cortex-

M0(+) cores. Such platforms are typical for a large family of IoT applications. Products in this range include the LPC800 series by NXP (4–16 KiB of SRAM), STM32F0 by ST (4–32 KiB of SRAM), and the XMC1000 by Infineon (16 KiB of SRAM). It is clear that PQC implementations with memory requirements of well over 50 KiB do not fit on these platforms and will not be able to sign nor verify digital post-quantum signatures.

In this chapter we investigate how to approach the challenge of trimming down the memory usage of the Dilithium algorithm. We intend to find out whether it is possible to execute Dilithium on such memory-constrained devices that often have up to 8 KiB of SRAM and, if so, which performance penalty is incurred.

Contribution. First, in Section 6.2, we present various *high-level* memory consumption and speed trade-offs for the signing procedure of Dilithium. Due to the iterative nature of the signing procedure, there exist interesting implementation choices. We implement each trade-off into the implementation presented in [GKS21] and present the resulting stack usages and cycle counts. In the second part, we select the most memory-economical strategy and continue to consider and apply multiple *low-level* tradeoffs, and assess how far we can (reasonably) go with trimming down the memory usage. In particular, in Sections 6.4 and 6.5 we come up with methods to reduce the amount of memory needed to store the $\mathbf{w}$ vector and the amount of memory needed to compute $c \cdot \mathbf{s_1}$, $c \cdot \mathbf{s_2}$, and $c \cdot \mathbf{t_0}$. For all operations (KeyGen, Sign, and Verify), we propose an efficient allocation of the variables used during the algorithm. We present a new pure-C implementation for Dilithium in which the techniques are applied, which is optimized *only* low-memory usage. Then in Section 6.6, we measure the achieved memory usage and the impact on the performance of the algorithm on the Cortex-M4 platform using the pqm4 [PQM4] framework.

6.2 Basic time-memory trade-offs

Depending on the programmer's requirements, there are multiple ways in which we can implement Dilithium signing, each with their own tradeoffs.

For microcontroller implementations of Dilithium the main challenge is that computing $\mathbf{A}$ is expensive since it involves many calls to `SHAKE256` which is relatively slow in software. Also, $\mathbf{A}$ is used multiple times during the signing procedure. Conse-

quently, we either have to store the complete matrix $\mathbf{A}$ in RAM or flash, or incur the cost of having to recompute it during each loop iteration.

In order to explore this time-memory tradeoff, we look at the signing operation using three different strategies. In the first strategy, we refuse to recompute $\mathbf{A}$ during the signing operation and instead store it in flash. The second strategy describes the more traditional implementation of Dilithium, expanding $\mathbf{A}$ once during each signing operation before entering the rejection-sampling loop. The third case describes the situation wherein we are highly constrained in flash and SRAM size, but have ample performance budget. In this strategy, we save the amount of memory needed by computing both $\mathbf{A}$ and $\mathbf{y}$ on the fly.

Although the algorithm's intermediate values can be stored anywhere in the RAM, to keep it simple, we will consider that all variables are stored on the stack.

6.2.1 Strategy 1: A in flash

In Dilithium signing, the values $\mathbf{A}$, $\hat{\mathbf{s}}_1$, $\hat{\mathbf{s}}_2$, and $\hat{\mathbf{t}}_0$ depend only on the Dilithium key pair. Therefore, instead of computing these values during signing, we can compute these values as part of the key generation. We assume that the platform has some kind of non-volatile storage that is large enough (and secure enough[1]) to store these extra values. Then, during the signature generation algorithm, instead of passing in sk (as described in line 9 of Algorithm 3.17), we pass a larger struct that also contains the precomputed values. These precomputed values ($\mathbf{A}$, $\hat{\mathbf{s}}_1$, $\hat{\mathbf{s}}_2$ and $\hat{\mathbf{t}}_0$) add up to $k \cdot \ell + 2k + \ell$ (with k, ℓ as listed in Table 3.1) polynomials that have to be stored extra. In the case of NIST round-3 Dilithium3, this amounts to 47 KiB of extra flash space as each Dilithium polynomial requires 1 KiB when stored uncompressed.

Because these four values are now stored separately, we do not have to compute (and store) them anymore during the signature generation. Thus, this strategy will save a considerable amount of SRAM, in exchange for (relatively cheap) flash space. Furthermore, in the absence of hardware-accelerated SHAKE256, generating $\mathbf{A}$ is a relatively expensive step in the signature-generation process. Having $\mathbf{A}$ stored in flash will speed up the overall performance of generating signatures. Hence, we think that this strategy will be the most favored to be deployed in a real-world small-devices environment.

[1] $\mathbf{A}$, and $\hat{\mathbf{t}}_0$ need to be integrity-protected; $\hat{\mathbf{s}}_1$, and $\hat{\mathbf{s}}_2$ need to remain secret and integrity-protected.

6.2.2 Strategy 2: A in SRAM

When there *is* enough SRAM available on the device, we opt for the "traditional" implementation of the signature generation algorithm. That is, we follow the specification closely and implement signature generation following the general structure of the Sign function Algorithm 3.17. Apart from some space for storing intermediate values, we will need to allocate

- $4k$ polynomial slots for storing $\hat{\mathbf{t}}_0$, $\hat{\mathbf{s}}_2$, $\mathbf{w}$, $\mathbf{w_1}$;
- $(k+3)\ell$ polynomial slots for storing $\mathbf{A}$, $\hat{\mathbf{s}}_1$, $\mathbf{y}$ and $\hat{\mathbf{y}}$; and
- 1 polynomial slot for storing $\hat{c}$.

This adds up to a pretty high lower bound of $k \cdot \ell + 4k + 3\ell + 1$ KiB of necessary stack space, e.g., 70 KiB for Dilithium3.

6.2.3 Strategy 3: streaming A and y

For the last strategy we considered the situation, wherein we optimize stack usage without using extra long-term storage for precomputed values. In the signing implementation, we optimize *exclusively* for stack usage. We only intend to find the lower bound of the needed stack space.

In contrast to the other strategies, we do not store any complete copies of $\mathbf{A}$ and $\mathbf{y}$. Instead, we regenerate every element of $\mathbf{A}$ and $\mathbf{y}$ on the fly when we compute elements of $\mathbf{w}$ (in line 18 of Algorithm 3.17). Because we do not retain $\mathbf{y}$ after this step, we regenerate it again later (in line 23 of Algorithm 3.17). Relative to strategy 2, this saves us $k \cdot \ell$ polynomials of space for $\mathbf{A}$, and another ℓ polynomials for $\mathbf{y}$.

When we look further into stack-optimizing the signing algorithm, we find that the main bottleneck in terms of stack usage is the overlapping lifetimes of $\mathbf{w}$ and $\hat{c}$. In lines 23 and 28 of Algorithm 3.17, the values $\mathbf{r_1}$, $\mathbf{r_0}$ and $\mathbf{h}$ all depend on both $\mathbf{w}$ and $\hat{c}$. However, in lines 20 to 21 we also need the complete value of $\mathbf{w_1}$ (and thus $\mathbf{w}$) to compute $\hat{c}$. Therefore, we conclude that we either have to store $\mathbf{w}$ and $\hat{c}$ both at the same time; *or* we have to recompute every element of $\mathbf{w}$ on the fly when we are computing $\mathbf{r_0}$ in line 23, and when we are constructing the hint $\mathbf{h}$ in line 28.

In order to recompute elements of $\mathbf{w}$, we would have to do the matrix multiplication $\mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{y}))$ all over again, including the complete regenerating of $\mathbf{A}$ and $\mathbf{y}$.

The performance cost of this optimization would be at least a factor 2, so we chose to not do this. Instead we accept that $\mathbf{w}$ and $\hat{c}$ both need to be stored at the same time.

6.2.4 Splitting signature generation in an offline and online phase

To speed up the Dilithium signing process even more, one can choose to split the signature generation in an *offline* and *online* phase, where the offline phase can already be performed before the message to be signed is known. The general idea of using an offline/online phase was introduced in 1989 by Even, Goldreich, and Micali [EGM96], and was first proposed for usage in lattice-based signature schemes in [AYS15]. It has also been used by Ravi, Gupta, Chattopadhyay, and Bhasin in [RGCB19, Section 4.1.2] to optimize the online latency of Dilithium signing.

However for Dilithium, this optimization comes with a significant cost. In their paper, Ravi, Gupta, Chattopadhyay, and Bhasin describe that an additional 260 KiB of space[2] is needed to store the precomputed values for (NIST round-2) Dilithium3, such that there is a 95% probability that at least one of the $\mathbf{y}$ values results in an accepted signature. For the main target of the implementation from Chapter 4 (the ATSAM3X8E), that would mean that more than half its flash space would already be lost to storing these precomputed values. We think that, in the general case, the improved signature-generation latency does not justify this kind of loss in available flash space.

6.2.5 Results

Each of the strategies we incorporate into the NIST round-2 implementation of Dilithium from Chapter 4. Then we evaluate the stack usage for Keygen, Sign, and Verify for each different strategy. For our signing strategy 1, we need to pre-compute $\mathbf{A}$, $\hat{\mathbf{s}}_1$, $\hat{\mathbf{s}}_2$ and $\hat{\mathbf{t}}_0$. We include this pre-computation in the key generation.

In Table 6.1, we show the considerable improvement in stack-space usage over the previous works that implement the NIST round-2 version of Dilithium. We see that signature verification needs only around 10 KiB of storage space (depending on the Dilithium parameters), without incurring a performance hit. Furthermore, when

[2] See [RGCB19, Table 6]. Compute $300 - 34 = 266$ KB $\approxeq$ 260 KiB.

Dilithium is deployed on a device that has enough space to store $\mathbf{A}$—either in SRAM or in flash—we get a reasonable signature-generation latency.

However, in the same tables we see the cost of aggressively optimizing for stack space (i.e., when applying strategy 3). On both platforms, we see really disproportionate cycle counts for signature generation, with a slowdown of 3.3× to 3.9× when comparing to strategy 2. For Dilithium3, the strategy-3 signature generation takes about 24 million cycles on the Cortex-M4. On slow devices (like common 16 MHz microcontrollers), this latency grows into the order of *seconds*.

When comparing to the [RGCB19] implementation, our strategy 1 is similar to their scenario 2, while our strategy 2 corresponds to their scenario 1. For both scenarios, we achieve substantial speedups for all parameter sets ranging from 14% to 20%. Unfortunately, [RGCB19] does not report the memory usage of their implementations, so we cannot compare them. We can however observe that our speedup of strategy 1 compared to strategy 2 is consistent with that of [RGCB19], with speedups of {21%–22%,22%–24%,33%–37%} for Dilithium{2,3,4} respectively.

6.3 Introducing advanced memory optimizations

In Section 6.2 we have looked at some time-memory tradeoffs that can be applied to Dilithium, and how they perform for NIST round-2 Dilithium. Although the analysis provides a good intuition into the general memory footprint of a Dilithium implementation, it is still outdated, it only covers signature generation, and it does not indicate any kind of lower bound. In the rest of this chapter, we will to complete the work. While Section 6.2 only looked at the high-level structure of the Dilithium signing algorithm, we will allow ourselves to step into primitive operations (such as the NTT or pointwise multiplication) and modify them as part of the optimization endeavour. Our goal is to determine if, using these optimizations, Dilithium can be deployed to a device that only a minimal amount of memory. We will apply the advanced memory optimization techniques and try to find that lower bound for the latest version (i.e., NIST round 3) of Dilithium, using a new pure-C implementation of Dilithium. We will measure the memory footprint of the implementation to determine which variants of Dilithium can be executed in 8 KiB of RAM.

Table 6.1: Performance result for NIST round-2 Dilithium on the Cortex-M4 (STM32F407 at 24 MHz). Averaged over 10 000 executions.

Algorithm/ strategy	Params	Work	Speed [kcc]	Stack [B]
KeyGen (1)	Dilithium2	**This work**	2 267	12 836[c]
	Dilithium3	**This work**	3 545	15 916[d]
	Dilithium4	**This work**	5 086	18 980[e]
KeyGen (2 & 3)	Dilithium2	**This work**	1 315	7 916
	Dilithium3	[GKOS18]	2 320	50 488
	Dilithium3	**This work**	2 013	8 940
	Dilithium4	**This work**	2 837	9 964
Sign (1)	Dilithium2	[RGCB19, scen. 2][a]	3 640	–
	Dilithium2	**This work**	3 097	14 452[c]
	Dilithium3	[RGCB19, scen. 2][a]	5 495	–
	Dilithium3	**This work**	4 578	17 660[d]
	Dilithium4	[RGCB19, scen. 2][a]	4 733	–
	Dilithium4	**This work**	3 768	20 860[e]
Sign (2)	Dilithium2	[RGCB19, scen. 1][b]	4 632	–
	Dilithium2	**This work**	3 987	38 300
	Dilithium3	[GKOS18]	8 348	86 568
	Dilithium3	[RGCB19, scen. 1][b]	7 085	–
	Dilithium3	**This work**	6 053	52 756
	Dilithium4	[RGCB19, scen. 1][b]	7 061	–
	Dilithium4	**This work**	6 001	69 276
Sign (3)	Dilithium2	**This work**	13 332	8 924
	Dilithium3	**This work**	23 550	9 948
	Dilithium4	**This work**	22 658	10 972
Verify	Dilithium2	**This work**	1 259	9 004
	Dilithium3	[GKOS18]	2 342	54 800
	Dilithium3	**This work**	1 917	10 068
	Dilithium4	**This work**	2 720	11 084

[a] "Strategy 1" from Section 6.2.1 corresponds to "Scenario 2" in [RGCB19].
[b] "Strategy 2" from Section 6.2.2 corresponds to "Scenario 1" in [RGCB19].
[c] For Dilithium2 using stack strategy 1, an additional 23 632 bytes of flash space are used for storing the precomputed values.
[d] For Dilithium3 using stack strategy 1, an additional 34 896 bytes of flash space are used for storing the precomputed values.
[e] For Dilithium4 using stack strategy 1, an additional 48 208 bytes of flash space are used for storing the precomputed values.

6.4 Signature generation

As we have seen in Table 6.1, the digital signature generation in Dilithium requires a significant amount of memory. For NIST round-2 Dilithium, the smallest implementation uses around 9–11 KiB of memory. Because of the increase of k and ℓ in round 3 of the NIST competition, we expect these number to increase with the next version of Dilithium. The fastest implementation reported on the benchmark results from pqm4[3] even requires approximately $\{49, 80, 116\}$ KiB for Dilithium{2,3,5} respectively. In this section we outline the proposed techniques to reduce the memory requirements of the signing algorithm.

6.4.1 Streaming A and y

In Dilithium's signature generation algorithm the matrix $\mathbf{A}$ requires $k \cdot \ell$ KiB: by far the largest contributor to memory. As we follow "Strategy 3" (Section 6.2.3), we apply the optimization to not generate the entire matrix $\mathbf{A}$ but only generate the elements of $\mathbf{A}$ and $\mathbf{y}$ on-the-fly when they are needed.

Compared to the traditionally structured signing algorithm, the expected memory reduction of this optimization is $k \cdot \ell$ KiB for $\mathbf{A}$, and ℓ KiB for $\mathbf{y}$; in practice this means a saving of $\{20, 35, 63\}$ KiB, for Dilithium{2,3,5} respectively. This optimization comes at a performance price: the matrix $\mathbf{A}$ needs to be regenerated again from ρ on every iteration of the rejection-sampling loop. Moreover, $\mathbf{y}$ needs to be generated twice during each iteration of the rejection-sampling loop; once for computing $\mathbf{w} = \mathbf{A}\mathbf{y}$, and once for computing $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ later on. From the high-level analysis, we expect a slowdown factor of around 3.3–3.9 compared to precomputing $\mathbf{A}$ and $\mathbf{y}$ if we assume that both versions of Dilithium behave similarly.

6.4.2 Compressing w

Another significant contributor to the overall memory requirements is the vector $\mathbf{w}$. This could be resolved if one could compute and use a single element at a time during the signature generation. Unfortunately, this is not possible due to the overlapping lifetimes of $\mathbf{w}$ and c, as identified in [GKS21] (Section 6.2). In line 20 of Algorithm 3.17,

[3] Accessed February 14, 2022, using revision `3bfbbfd3`. The implementation in pqm4 is based on [AHKS22].

c is computed from $\mathbf{w_1} = \mathsf{HighBits}(\mathbf{w})$. On lines 23 and 28, the values $\mathbf{r_0}$ and $\mathbf{h}$ depend on c and the complete vector $\mathbf{w}$. This means that either *all* elements of $\mathbf{w}$ must be retained between computing c and computing $\mathbf{r_0}$ and $\mathbf{h}$, or $\mathbf{w} = \mathbf{Ay}$ must be computed twice during each iteration of the rejection-sampling loop. Recomputing the matrix-vector multiplication in every loop iteration will roughly double the execution time of the signing algorithm; although a viable direction to reduce memory we deemed this performance impact too large. Instead, we explore the other option where all elements of $\mathbf{w}$ have to be alive at the same time at the cost of storing k polynomials.

One polynomial has $n = 256$ coefficients, which are all bounded by $q = 2^{23} - 2^{13} + 1$. In previous works, implementations have always used 32-bit data types for storing these coefficients. Instead, we use a *compressed* representation for storing $\mathbf{w}$. Instead of using 32-bit registers for storing $\mathbf{w}$ coefficients, the approach is to explicitly reduce them modulo q, reducing each coefficient to 24 bits and next pack the 256 24-bit coefficients into a 768-byte array. The (24-bit) compressed coefficients reduce the amount of storage that is used for storing $\mathbf{w}$ from $k \cdot 1024$ bytes to $k \cdot 768$ bytes, which results in a reduction of {1.0,1.5,2.0} KiB for Dilithium{2,3,5}, respectively. Packing and unpacking coefficients of $\mathbf{w}$ adds a little overhead during the matrix-vector multiplication.

It should be noted that one could compress each coefficient into 23 bits instead of 24 bits. This would save an additional 32 bytes per polynomial. However, working with the 23-bit format (packing and unpacking) is significantly more cumbersome and therefore slower compared to the 24-bit format for alignment reasons and the need for more expensive reductions during the computation of $\mathbf{w}$. This explains why we compress to 24 bits for the results presented in Section 6.6.

6.4.3 Compressing $c \cdot \mathbf{s_1}$, $c \cdot \mathbf{s_2}$, and $c \cdot \mathbf{t_0}$

The multiplications of the challenge $c \in B_\tau$ with the polynomials $\mathbf{s_1} \in S_\eta^\ell$, $\mathbf{s_2} \in S_\eta^k$, and $\mathbf{t_0} \in S_{2^d}^k$, are typically computed using NTTs (see lines 22, 23 and 28 of Algorithm 3.17). As the values of $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$ are static throughout a whole signing computation, it is computationally most efficient to pre-compute the NTTs on all these elements, and store $\hat{\mathbf{s}}_\mathbf{1}$, $\hat{\mathbf{s}}_\mathbf{2}$ and $\hat{\mathbf{t}}_\mathbf{0}$ in memory before entering the rejection-sampling loop. Avoiding the storage of these elements reduces the total memory used by $2k + \ell$ KiB; i.e., {12,17,23} KiB for Dilithium{2,3,5}, respectively. Indeed, this would naively imply a

performance loss as the NTTs need to be computed several times (at least once for each aborted signature). However, the routine using (inverse) NTTs on the fly needs at least 1.75 KiB of space: 1 KiB is needed to compute the (inverse) NTT for one operand, while 0.75 KiB is needed to store the other operand in (24-bit) compressed form. We find that, for the computation of $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$, we do not necessarily need to use the regular NTT. For most values involved, there is a lot of structure that can be exploited. In the remainder of this section we discuss three different ideas to compute $c \cdot \mathbf{s_1}, c \cdot \mathbf{s_2}$ and $c \cdot \mathbf{t_0}$ with lower memory requirements than using regular NTTs.

Sparse polynomial multiplication. The most obvious choice for polynomial multiplication is the schoolbook approach. At first glance, using schoolbook multiplication actually requires *more* memory compared to NTT-based multiplication because one cannot do the multiplication in-place. However, when using schoolbook multiplication, one does not need to store the right-hand-side operand polynomials ($\mathbf{s_1}$, $\mathbf{s_2}$, and $\mathbf{t_0}$) completely. We can multiply their coefficients in a streaming fashion, unpacking them "lazily" from the secret key. Apart from using a small buffer, we have now removed the need to store any full element from $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$. Although one still needs 1.0 KiB for the accumulator polynomial, only 68 bytes are required for storing the challenge c; as well as a small buffer of 32 bytes, which is used to unpack polynomial coefficients more efficiently from the secret key. This adds up to 1124 bytes total: a reduction of a factor 1.37 compared to using a regular (32-bit) NTT.

Furthermore, one can reduce the computational as well as the memory complexity by exploiting the regular structure of the challenge c [ABBK+16; WTJB+20]. Recall that the challenge polynomial has exactly τ non-zero coefficients that are either ± 1, where $\tau \in \{39, 49, 60\}$ depending on the Dilithium parameter set. Therefore, when multiplying c with some other polynomial, one really only needs to multiply each coefficient from the right-hand side operand with τ coefficients in c. Skipping the multiplications with the zero-elements is not a security concern (e.g., from a timing leakage perspective) since the challenge value c is public.

Using this property, one can use a data structure for c that allows for fast iteration over all the non-zero coefficients. We use a single 64-bit datatype which indicates for each of the τ non-zero positions whether it is a $+1$ or a -1; and an array of τ bytes which stores positions of the non-zero coefficients. The benefit of storing the indices of all non-zero coefficients, as opposed to storing a bit-string with bits set for each

non-zero coefficient, is the fast iteration over the non-zero indices. If we store a bit for every coefficient in c, we have to do a conditional addition/subtraction of the coefficient in the other operand for *every* coefficient of c, i.e., n times. By only storing the non-zero indices, we only have to do the addition/subtraction τ times and avoid computing any multiplications. Hence, this polynomial multiplication with c can be done using $\tau \cdot n$ additions or subtractions only.

Alternative Number Theoretic Transforms. When computing $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ one can use a different-sized NTT over a smaller prime as described in [AHKS22] (Chapter 5). The idea is that all coefficients of both $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are bounded by $\pm\tau \cdot \eta = \pm\beta$. This allows computing the polynomial product with modulus $q' = 257$ for Dilithium{2,5}, and $q' = 769$ for Dilithium3. Since the coefficients in the product are bounded by $\pm\beta$, they will not overflow when computing them modulo $q' \geq 2\beta$. In [AHKS22], this improves the performance of the NTT-based multiplications because—with $q' = 257$—some of the multiplications with twiddle factors become cheaper. Moreover, [AHKS22] still uses 32-bit registers for all values, which provides so much headroom that it eliminates the need for any intermediate Barrett reductions in both NTT algorithms. However, the small-modulus NTTs also allows one to store all coefficients in 16-bit variables; computing an NTT in half the amount of memory at the cost of reintroducing the intermediate Barrett reductions. When using this technique, the memory requirement of $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ is reduced to 1.0 KiB: 0.5 KiB for the first operand and product, and another 0.5 KiB for the second operand.

Kronecker Substitution. By applying (generalizations of) Kronecker substitution [Har09; Kro82] to $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ one can reduce the polynomial multiplication to the integer multiplications $c(2^\lambda) \cdot \mathbf{s_1}(2^\lambda)$ and $c(2^\lambda) \cdot \mathbf{s_2}(2^\lambda)$ modulo $2^{256\lambda} + 1$. The application of Kronecker substitution to lattice-based cryptography has been studied [AHHP+18; BRV22], but its use for $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ has not been considered yet. In order to retrieve the coefficients of the resulting polynomial, we require that $2^\lambda \geq 2\beta$. This means we can select $\lambda = 8$ for Dilithium{2,5} and $\lambda = 9$ for Dilithium3, transforming the full polynomial multiplication into a single 2048-bit multiplication and a 2304-bit multiplication respectively. This requires 256 or 288 bytes for each of the two inputs and result polynomials: assuming the result can overwrite one of the inputs this means 512 or 576 bytes in total. Additionally one can use the more general

Kronecker+ method [BRV22] to improve the performance further (the optimal setting depending on the platform).

Although Kronecker substitution works perfectly well on the regular central processing unit it is particularly suitable for small systems that typically have dedicated hardware to perform (public-key) cryptographic operations in a timely manner. For RSA or elliptic-curve cryptography (ECC), such co-processors come in the form of large-integer multipliers that are heavily optimized for performing integer (modular) multiplications.

6.4.4 Variable Allocation

After applying the memory optimizations described above we analyzed efficient memory allocation schemes during the Dilithium signature generation algorithm. This showed that one can reuse the 1 KiB memory location that is used for doing computations on non-compressed polynomials. On top of that, we need 128 bytes for storing μ and ρ'; and 68 bytes for storing c, as described earlier ($\tilde{c}$ is stored solely in the output buffer). The complete memory allocation of the signature generation algorithm is listed in Figure 6.1.

When looking at Figure 6.1 one can observe that the memory bottle-neck is shared between multiple subroutines. We see no trivial way to further optimize the allocation of variables in memory. The only time-memory tradeoff that could still be performed is to keep a single element of $\mathbf{w}$ at a time. Following the observation from Section 6.2.3 we dismiss this approach because it requires us to compute all elements of $\mathbf{w}$ *twice* during each iteration of the rejection sampling loop. This would not only require expanding all elements of $\mathbf{A}$ *and* $\mathbf{y}$ twice, one would also need to recompute $\hat{\mathbf{y}} = \mathsf{NTT}(\mathbf{y})$ and $\mathbf{w} = \mathsf{NTT}^{-1}(\hat{\mathbf{w}})$. Because the matrix multiplication is already a dominating factor in the signing algorithm, this optimization would likely result in another slowdown by a factor two. Its gains in terms of memory consumption would be $(k-1) \cdot 768$ bytes, i.e., $\{2.25, 3.75, 5.25\}$ KiB for Dilithium{2,3,5}, so it might be worthwhile if one can compensate for or cope with this performance penalty.

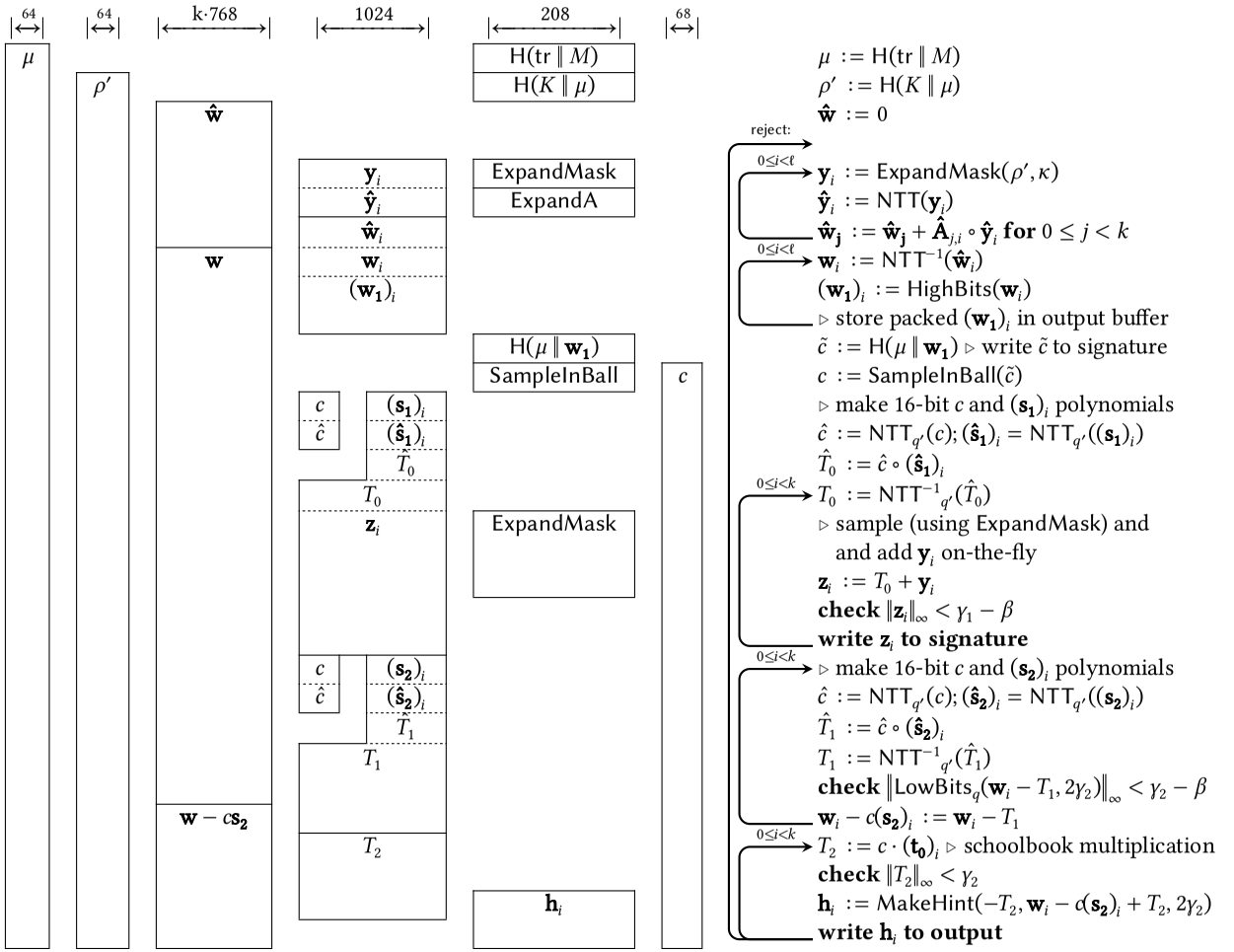


Figure 6.1: (Continued on page 100)

Figure 6.1: Memory allocation of the Dilithium signature generation algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range, except for the loop annotated by reject:, which indicates which code is repeated when a signature in the Sign algorithm is aborted. All temporary values are denoted by a T_i.

6.4.5 Summary of optimizations

In this section we have described a large number of (possible) optimizations that can be applied to optimize the signature generation algorithm. For clarity, let us summarize which optimizations were selected for use in our implementation:

- we generate the elements of $\mathbf{A}$ and $\mathbf{y}$ on-the-fly, as described in Section 6.4.1;
- for storing $\mathbf{w}$, we use the compressed format described in Section 6.4.2;
- for computing $c \cdot \mathbf{t_0}$, we use *sparse polynomial multiplication* (Section 6.4.3);
- for computing $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ we use the adapted *small-modulus* NTTs from Chapter 5 (as described in Section 6.4.3); and
- we use the variable allocation described by Figure 6.1 and Section 6.4.4.

6.5 Dilithium key generation and signature verification

Both the Dilithium key generation and verification algorithms are fundamentally different from the signature algorithm with the most important difference being that there is no rejection-sampling loop. Therefore, there is no performance benefit to precomputing the matrix $\mathbf{A}$ in these algorithms, which already reduces the memory requirement naturally. Moreover, in both KeyGen and Verify there are no polynomials for which it makes sense to precompute the NTT representation to speed-up the algorithms. This makes both algorithms significantly more lightweight in terms of memory compared to the signature generation, even without any further optimizations.

It is common that the key generation algorithm is executed on the same device where one performs the signature generation algorithm. Therefore, we do not attempt to reduce the memory footprint of KeyGen to the maximum extent, but instead try and minimize the memory footprint of max(KeyGen, Sign). In other words, we optimize the memory use of KeyGen, until it is at least as low as the memory use of Sign which we try to optimize as much as possible.

6.5.1 Key Generation

When following the same strategy for computing the multiplication $\mathbf{A} \cdot \mathbf{s_1}$ in the key generation algorithm as in the signing algorithm one can already remove the need for ℓ different memory slots for polynomials. Using this optimization in combination with careful scheduling the other memory (see Figure 6.2) already means that all variables used in KeyGen use less memory than the signature generation algorithm. Hence, there is no reason to sacrifice any performance to optimize the KeyGen algorithm further.

Let us outline some memory improvements for the interested reader who has requirements to reduce the memory even further. One idea comes up from the observation that one can transpose the order in which the multiplication in $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$ is performed. Recall that in the Sign algorithm, the lifetime of c overlaps with the lifetimes of all elements in $\mathbf{w}$ (where $\mathbf{w}$ is the output of the matrix-vector multiplication) which limits the potential to reduce memory. However, in the KeyGen algorithm there is no (equivalent to) c, i.e., there is no variable that causes the lifetimes of the elements in $\mathbf{t}$ to overlap. Hence, the elements in $\mathbf{t}$ do not have to be alive at the same time and can be computed in a streaming fashion. With this optimization one can reduce the memory by $(k-1)$ KiB, saving $\{3.0, 4.5, 6.0\}$ for Dilithium{2,3,5}, respectively.

6.5.2 Signature Verification

In the setting of the Dilithium signature verification algorithm we are interested in minimizing the memory usage as much as possible. There are many embedded applications that only use signature verification, e.g., secure boot implementations or in the case of public-key infrastructures.

The optimizations one can apply to the signature verification algorithm follow the same pattern as those of Sign and KeyGen. In particular, it is possible to verify any signature using only two slots for storing polynomials, of which one is 1.0 KiB and one is 768 bytes, using the optimizations from Section 6.5.1. Apart from the 1.75 KiB for storing two polynomials, one still needs twice the space for storing the SHA-3 state (208 bytes) plus one compressed challenge polynomial c (68 bytes). This sums up to a minimum of 2276 bytes of required memory for such an approach in the Dilithium verification algorithm. In contrast to the KeyGen and Sign algorithms, the memory usage of the Verify algorithm is independent from any of the Dilithium parameters.

6.6 Results & discussion

Our implementation. Using the Dilithium reference implementation[4] as a starting point, we wrote a new implementation for Dilithium, in which we applied the techniques described in Sections 6.4 and 6.5. Because we are only interested in validating the memory reduction techniques and *not* focused on performance we have opted to write a cross-platform implementation in pure C. Correspondingly, our implementation does not include any architecture-specific optimizations. Moreover, our implementation as well as the implementations we compare against are not hardened in any fashion except for the prevention of (cache-)timing attacks.

Our implementation introduces many new internal data types that are optimized for a lower memory footprint; like compressed polynomials (with 24-bit coefficients and 16-bit coefficients) and the compressed challenge. We implemented the $q' \in \{257, 769\}$ NTTs for $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ multiplications, and we implemented the schoolbook multiplication for the $c \cdot \mathbf{t_0}$ and $c \cdot \mathbf{t_1}$ multiplications. We improved the implementation such that parts can be called in a streaming fashion. For example, the matrix-vector multiplication and ExpandA routines have been merged into a single non-buffering function; and almost all packing/unpacking functions have been refactored to allow for (un)packing polynomials in small chunks. Because of the tight memory budget we have removed some local stack allocations from all internal Dilithium routines. Instead, one memory block is allocated on the stack in the root functions (i.e., `di-`

[4] `https://github.com/pq-crystals/dilithium`

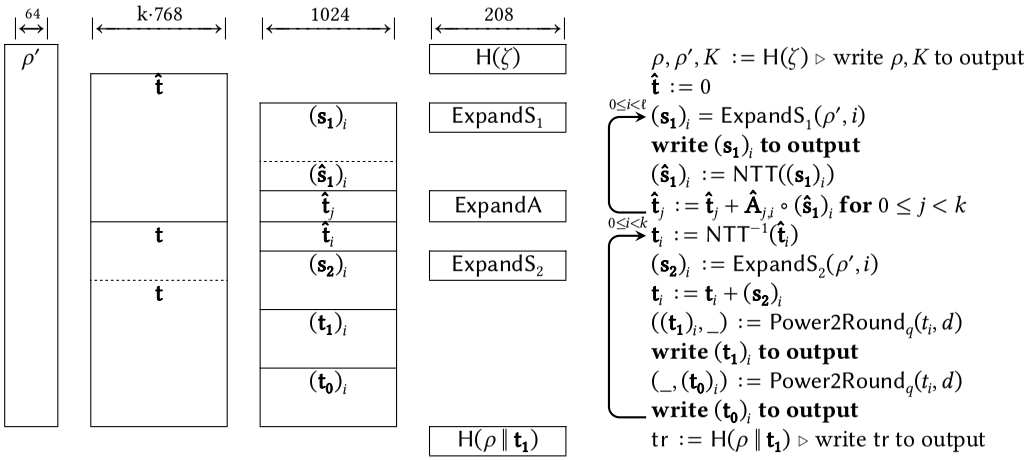


Figure 6.2: Memory allocation of the Dilithium key generation algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range.

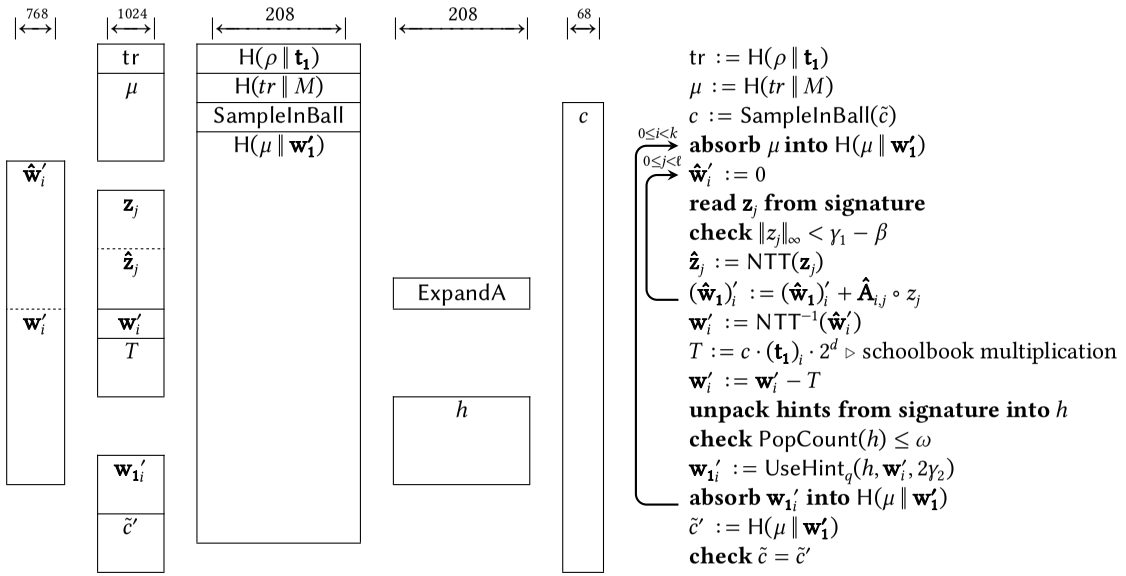


Figure 6.3: Memory allocation of the Dilithium signature verification algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range. All temporary values are denoted by a T_i.

`lithium_keygen`, `dilithium_signature`, and `dilithium_verify`) and passed to the internal functions.

As opposed to the previous works that only support a single Dilithium variant at a time, selected using C preprocessor macros at compile time, our implementation integrates all variants at the same time, and the variant is selected by the user at runtime as in typical in cryptographic software libraries.

Results. We integrated our implementation into a local fork of the benchmarking framework pqm4 [PQM4].[5] We compared the memory footprint and the execution times of our implementation to those of the Dilithium implementation in PQClean [KSSW22], the Dilithium-round-3 updated port of [GKS21] in pqm4, and the recent implementation results from [AHKS22].[6]

It should be noted that all of these implementations have different goals and implementation methods, so evaluating the benchmarking results is not as straightforward as just comparing performance numbers. Firstly, the PQClean implementation has been published as a "clean" implementation of Dilithium. Its main goal is to provide an implementation of Dilithium, written purely in C, that works cross-platform and follows best coding practices. It has been written with performance in mind and ensures a running time independent of secret-key-related material. However, it does not include any platform-optimized assembly code which has the potential to greatly improve the performance. On the other side, there are the pqm4 ([GKS21]) and [AHKS22] implementations. These implementations are specifically hand-crafted for the Arm Cortex-M4 platform and are highly optimized for performance (i.e., reducing the number of required cycles) and large parts of these implementations are written in Armv7 assembly. We also include the "strategy 3" implementation from [GKS21] (i.e., Section 6.2), but unfortunately it is hard to compare directly since that implementation is based on round-2 parameters of Dilithium which are significantly different compared to the latest (round 3) ones. As an indication, the round-2 Dilithium3 memory usage of signature verification and generation using this strategy are in both settings 10 KiB: significantly less compared to previous work but still too large for the embedded devices we target in this chapter.

[5] Commit hash `e47864b3`, forked on 8 Oct 2021.
[6] As of early 2022, this implementation has replaced the port of [GKS21] in pqm4.

Our implementation is designed with a different goal in mind: it is a cross-platform C implementation that optimizes in the first place for memory usage to ensure it can execute on memory-constrained ($\leq$ 8KiB) platforms. It makes a significant amount of sacrifices in terms of performance and does not contain any routines that are specially optimized for the Cortex-M4 (the techniques presented are platform independent). Therefore we expect the pqm4 implementation from [GKS21] (Chapter 4) and the implementation from [AHKS22] (Chapter 5) to outperform this implementation on Cortex-M4: we use a slower approach and a generic implementation. In order to assess the impact of the proposed techniques we remove the optimized assembly implementation from the equation and compare to the generic PQClean implementation. We include the performance figures of the other implementations for the sake of completeness.

An overview of the results is provided in Table 6.2. The testing followed the methodology described in Section 2.6.2. We used the STM32F4 Discovery board, which is based on the STM32F407 microcontroller. Our implementation was benchmarked using the pqm4 framework. To obtain the cycle counts we measured 10 000 executions and computed the average. The results for the pqm4 ([GKS21]) and [AHKS22] implementations are based on the results listed in [AHKS22]. The code was compiled using GCC version 9.2.1,[7] with optimization level `-Os`.

In Table 6.3 we have listed the code sizes for all the implementations that we compare in Table 6.2. We have measured these code sizes using the same settings as for the memory/performance measurements. Because the [GKS21] pqm4 and the [AHKS22] implementations are optimized for speed, we have listed their code sizes for the optimization levels `-O3` and `-Os`. In these metrics, the contribution of symmetric primitives—e.g., the size of the SHAKE code—has been excluded.

Discussion. The memory footprints reported in Table 6.2 for the presented techniques are close to the lower bounds provided earlier. The discrepancy in memory use is around 0.4 KiB of memory for all algorithms. The largest contributor to this additional memory use is the execution of SHAKE. The SHAKE code, which has been unadapted from the Dilithium reference implementation uses around 300 bytes of

[7] `arm-none-eabi-gcc (15:9-2019-q4-0ubuntu1) 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]`

Table 6.2: Memory usage and cycle counts for Dilithium in kibibytes (KiB) and kilocycles (kcc). The [GKS21] (pqm4) and [AHKS22] are implementations that include optimized assembly for Cortex-M4; PQClean [KSSW22] and **This work** are pure-C implementations. K, S, and V correspond to the signing primitives KeyGen, Sign, and Verify respectively. All cycle counts were averaged over 10 000 iterations.

			Dilithium2			Dilithium3			Dilithium5		
	variant		KiB		kcc	KiB		kcc	KiB		kcc
			total	≤ 8		total	≤ 8		total	≤ 8	
with asm	[GKS21] (pqm4)	K	37.1		1 602	59.6		2 835	95.7		4 835
		S	47.9		4 219	72.3		6 742	112.3		8 960
		V	35.2		1 579	56.6		2 700	90.8		4 718
	[AHKS22]	K	37.1		1 598	59.6		2 830	95.7		4 828
		S	47.9		4 083	67.4		6 624	113.3		8 726
		V	35.2		1 572	56.6		2 692	90.8		4 707
C only	PQClean [KSSW22]	K	37.4		2 025	59.4		3 504	–[a]		–[a]
		S	50.7		8 034	77.7		12 987	–[a]		–[a]
		V	35.4		2 223	56.4		3 666	–[a]		–[a]
	This work	K	**4.9**	✓	**2 927**	**6.4**	✓	**5 112**	**7.9**	✓	**8 609**
		S	**5.0**	✓	**18 470**	**6.5**	✓	**36 303**	**8.1**		**44 332**
		V	**2.7**	✓	**4 036**	**2.7**	✓	**7 249**	**2.7**	✓	**12 616**

[a] Implementation disabled because the device does not have enough RAM to support it.

Table 6.3: Code sizes of the implementations from Table 6.2 expressed in bytes. Opt-level denotes the optimization level that was used. Contribution of Keccak and AES code is excluded from all implementations.

implementation	opt-level	Dilithium2	Dilithium3	Dilithium5
[GKS21] (pqm4)	`-O3`	10 564	10 092	–[a]
[AHKS22]	`-O3`	18 448	19 916	18 262
[GKS21] (pqm4)	`-Os`	9 700	9 276	–[a]
[AHKS22]	`-Os`	17 408	19 012	17 234
PQClean	`-Os`	6 986	6 534	–[b]
This work	`-Os`	**10 091**[c]	**10 091**[c]	**10 091**[c]

[a] Not reported by pqm4.
[b] Implementation disabled because the device does not have enough RAM to support it.
[c] Implementation includes support for all Dilithium variants.

stack. The last 100 bytes are found in call-tree information and temporary buffers used during the packing and unpacking of polynomials into bit-arrays.

Table 6.2 clearly shows that the proposed techniques pay off. The states of both Dilithium2 and Dilithium3 for signature generation, verification and key generation easily fit into 8 KiB. It should be noted that none of the other high-speed implementations can execute on devices even with 32 KiB of memory. The amount of headroom arguably allows for plenty of other tasks to run on the device; 3.0 KiB in the case of Dilithium2 and 1.4 KiB for Dilithium3. The memory footprint of Dilithium5 signing *just* exceeds 8 KiB. For Verify, the memory footprint is reduced to 2.7 KiB.

This is of course only half of the story. The memory reduction techniques have a clear impact on the performance of the scheme. When comparing cycle counts to those of the PQClean implementation (which is the implementation most similar to ours), one observes a factor 2.3–2.8 slowdown for Sign and a factor 1.8–2.0 slowdown for Verify. For both algorithms, the difference in performance is due to the overhead from the (24-bit) bit-packing operations in the matrix-vector multiplication, and the slower schoolbook method for multiplying $c\mathbf{t_0}$. For Dilithium3 signing there is some additional overhead, because the $q' = 769$ NTTs are somewhat slower than the $q' = 257$ NTTs in the other variants.

Optimization efforts from [GKS21] and [AHKS22] have lead to a 43%–44% reduction of cycles in Sign compared to the PQClean implementation. Similarly, one can expect

that future performance enhancements will be able to improve the performance of our implementation of the memory reduction techniques as well. Depending on the platform, integrating more optimized assembly implementations for SHAKE, (inverse) NTT, and challenge multiplication could result in significant performance gains. In particular, many of the values in the challenge multiplication are 8 bits, This is suitable for parallel computation using SIMD instructions, which are not used in our C-implementation.

More importantly, many of the memory-constrained devices come equipped with dedicated cryptographic coprocessors for symmetric primitives (such as SHAKE) as well as for big-number arithmetic. When one can make use of these coprocessors, the execution times could be reduced drastically: especially because SHAKE remains a dominating component of the Dilithium execution time as well as the polynomial multiplication [BRV22].

Although the reduction of the run-time state has a big impact on the execution speed of the algorithm, we see from the results in Table 6.3 that this is not the case for the code size. The code for our new implementation is slightly bigger than the PQClean code, but about the same size as the optimized implementations.[8] Moreover, we must take into account that our implementation supports *all* variants of Dilithium at the same time, so a slight increase is actually expected.

6.7 Conclusion

Although there is considerable performance impact when implementing Dilithium in a low-memory environment, we have shown that such low-memory Dilithium implementations are feasible in practice. In particular, we broke the 8 KiB memory barrier for Dilithium2 and Dilithium3. Dilithium5 uses a *little* bit more memory than 8 KiB, but we have shown that there are still time-memory tradeoffs that can be applied, even though these tradeoffs are relatively expensive in terms of performance.

When earlier work (like [RGCB19]) was published, it was not clear whether Dilithium was a scheme that could even be considered for memory-constrained devices. Then [GKS21] showed that the Dilithium algorithms could reasonably fit into 16 KiB of memory. In this chapter, we show that most variants of Dilithium can even fit

[8] It is clear that the handwritten assembly in the [GKS21] and [AHKS22] implementations—which is very aggressively loop-unrolled—comes at a significant cost in code size.

into 8 KiB without a very drastic impact on performance. More so, we reduced the memory footprint for Dilithium verification to below 3 KiB. For memory-constrained devices, storing Dilithium's public keys and signatures has arguably become a bigger challenge than storing its run-time state.

7 Post-quantum secure boot on vehicle network processors

7.1 Introduction

Up to this point in this thesis, we have only focused on the *implementation* of Dilithium. In this chapter, we will look at the *usage* of Dilithium in larger embedded systems. In embedded systems, one typical use for digital signatures is for verifying that the code that runs on a device has not been tampered with. For example, firmware updates are usually authenticated by the software vendor using digital signatures. Some devices can also be configured to verify the firmware's signature again during boot, preventing the malicious modification of the code after the code is installed on the device.

Because of this, these signatures serve as the basis of trust for any other applications running on the system, and are critical for providing safety and security to automotive, edge, industrial, and many other domains. For example, modern cars feature service-oriented gateways that are responsible for transferring data between various vehicle networks, handling over-the-air (OTA) updates, communicating with the cloud, etc. For these kinds of devices, it is critical for the safety and security of a driver that they are securely booted and updated.

7.1.1 Secure boot

The goal of secure boot is to guarantee integrity and authenticity of the software running on a system. Although there are different ways in which to achieve this, ultimately the confidence in a system leads back to a so-called *Root of Trust* (RoT). For example, an RoT can consist of executable code and (hashes of) keys in Read-Only Memory (ROM) that performs various initializations, verifies the authenticity

of the firmware, and finally passes control to the (now authenticated) firmware. The requirements on RoTs are well-documented by various organizations, e.g., see TCG [CG19, Part 1 §9.5.5] or GlobalPlatform [GP18], and its implementation should hold up against strong testing and certification (e.g., ISO 26262) requirements. In particular, in order to prevent any of the ROM code being modified, or executable instructions skipped altogether, the RoT should be protected against physical fault attacks [BDL97; BS97].

Since both security requirements as well as cost of implementation for RoTs are high, their design typically aims to provide the necessary security requirements with minimal footprint. As such, in most modern systems the boot flow is not completed when the ROM code passes on control. Instead, more advanced features are offloaded to a (second-stage) *boot loader*, which is verified by ROM code and made responsible for the remainder of the boot sequence. Of course, this boot loader can in turn verify and advance control to the next stage, creating a *chain of trust*. In complex ecosystems distinct parties can be responsible for the different stages in the chain: while immutable hardware such as ROM needs to be established at manufacturing time by a Tier-1 or Tier-2 supplier, second (or higher) stage boot loaders can rely on memory that is programmed only later in the process by Original Equipment Manufacturers (OEMs), for example. This chain can extend all the way to end users running their Operating System (OS) of choice.

7.1.2 Post-quantum digital signatures for secure boot

As far as we are aware at the time of writing, all widely used and deployed approaches to realize digital signatures in the embedded space are either based on elliptic curves [Kob87; Mil86] or RSA [RSA78]. As such, the well-known threat of the realization of a quantum computer applies here as well: if large-scale quantum computers are to become a reality, Shor's algorithm [Sho94] will be able to recover ECC/RSA private keys in polynomial time. In the context of automotive network processors, such a development would allow an attacker to sign their own firmware updates, and install their own (unauthorized) code on the device. Even though this scenario might seem far away, cars will often operate on the road for multiple decades; as such, their security measures have to be able to sustain decades of attacks. There-

fore, we cannot delay evaluating the impact of using post-quantum cryptography for their secure-boot setups.

7.1.3 Related work

Sanwald, Kaneti, Stöttinger, and Böhner performed a thorough investigation of secure boot in the automotive domain [SKSB20]. Integration of post-quantum secure key exchange and digital signature verification has been studied before. The main investigations have been around hash-based signature schemes, since they have already been standardized by NIST [NIST20b]. They come with some potential disadvantages of requiring to keep a state during signature generation. An impact assessment of hash-based post-quantum secure schemes on secure boot is studied by Kampanakis, Panburana, Curcio, and Shroff [KPCS20]. Hermelink, Pöppelmann, Stöttinger, Wang, and Wan perform an investigation into Authenticated Key Exchange (AKE) combining XMSS and NewHope [HPSW+20], while Feritzmann, Vith, Flórez, and Sepúlveda analyze lattice-based key encapsulation mechanisms (KEMs) for automotive systems [FVFS21]. Also, Kumar, Gupta, Chattopadhyay, Kasper, Krauß and Niederhagen [KGCK+20] investigate how hash-based schemes can be integrated into a secure SoC platform around RISC-V cores and evaluated on an FPGA.

As far as we are aware, the integration of lattice-based schemes into the secure-boot flow has been not investigated before. In this work, we focus on the NIST signature finalist Dilithium. Dilithium is often considered for embedded applications due to its favorable runtime and relatively small size, for example, the embedded implementation from [GKOS18; RGCB19] and the improvements presented in Chapters 4 and 6.

7.1.4 Contribution

We investigate the practical impact of protecting the secure boot flow for vehicle network processors against quantum attacks. This is realized by integrating the Dilithium digital signature scheme into the secure boot process of the S32G platform. As part of this work we created a fault-attack resistant (against single-targeted faults) Dilithium signature verification algorithm, which uses significantly less memory than the state-of-the-art. This implementation was integrated into the S32G HSE secure boot flow (cf. Section 7.3). We have measured the latency of our Dilithium verification algorithm in a regular setting and using *pre-hashing* with SHA256. Our

results (Section 7.3.3) make it clear that the usage of post-quantum cryptography does have an impact on the (one-time) installation time of an application image. However after installation, the S32G uses a reference proof instead of verifying the original signature, which means that the boot time is not affected by the signature scheme. We evaluate these results and find that the impact is fairly minimal: a transition to post-quantum secure boot can be considered practical for this application.

7.1.5 Organization

We begin with a description of the S32G automotive platform and its (secure) boot flow in Section 7.2. In Section 7.2.2 we describe the process and results of integrating Dilithium. Finally, we present our conclusions in Section 7.4.

7.2 S32G vehicle network processors

7.2.1 Platform description

In this work, we use the *S32G vehicle network processor* as the target platform for the impact assessment of integrating post-quantum cryptography in the secure boot flow. This high-end platform is developed by NXP Semiconductors and part of a larger S32 product family which includes the S32R, S32K and S32S and is designed to meet the safety and security requirements in the automotive and industrial domains (i.e., compliance with IEC 61508 [IEC10] and ASIL-D classification in ISO 26262 [ISO18]). Typical uses include service-oriented gateways, domain controllers, vehicle computers and safety processors. The S32G consists of a combination of microcontrollers (MCUs) based on the Arm Cortex-M7, and microprocessors (MPUs) based on Arm Cortex-A53. These are combined with several types of memory (SRAM, DRAM, NOR/NAND Flash) and various hardware accelerators. Most notably, it contains a Hardware Security Engine (HSE) which supports both symmetric and (classical) asymmetric cryptography accelerators, a random number generator, and dedicated secure memory. The HSE is also powered by an Arm Cortex-M7 core and serves both as a secure environment for host applications, as well as being responsible for (part of) the boot flow if secure boot is enabled.

The precise configuration depends on the choice of model: we deploy the S32G274A which contains 3 Arm Cortex-M7 cores, 4 Arm Cortex-A53 cores, and 8 MB of system RAM. Each of the MCUs runs in a delayed lockstep configuration at a maximum frequency of 400 MHz and has 32 KB instruction and data caches. The MPUs are configured as 2 clusters of 2 cores each running at a maximum frequency of 1 GHz. Every core has access to 32 KB L1 instruction and data caches, while each cluster shares another 512 KB of L2 cache. Optionally, the A53 clusters can be configured to also run in a delayed lockstep setting, effectively removing one of the clusters from an application's point of view but increasing the fault tolerance.

7.2.2 Secure boot on the S32G274

The S32G274A provides two modes of startup, a normal start-up sequence (also referred to as "normal boot") and a secure start-up sequence (also referred to as "secure boot"). The secure boot process involves a series of stages. In each stage, a new piece of code is executed after passing all necessary checks for authenticity and optionally decryption of the protected content. In case the authentication fails, the related CPU subsystem remains in reset, potentially rendering the device inoperant, or at least not operating as originally designed for the targeted application.

In the secure startup process, the S32G starts operating a trusted-boot stored Read-Only Memory (ROM) firmware (BootROM), that is responsible for verifying, decrypting, and loading the HSE Security firmware (HSE-FW) into HSE secure memory before handing over the control to it. Once HSE-FW is up and running, it is responsible for initiating the next boot stage, by verifying and optionally decrypting the Application (Bootloader), before starting the Application CPUs. The authenticity check is based on cryptographic primitives. In particular, digital signatures schemes that are supported for authenticating application images are RSA [RSA78] with various padding schemes [PKCS198], ECDSA [SECG00], and EdDSA [BDLS+11; NIST23b].

The HSE requires three essential components for the boot sequence, which need to be installed before enabling secure boot. For example, this can be done by executing an application that performs the installation via the normal boot sequence, or through the serial interface. Firstly, any user keys (i.e., those not already in ROM) that are to be used by the HSE-FW to check the authenticity of the application need to be provisioned.

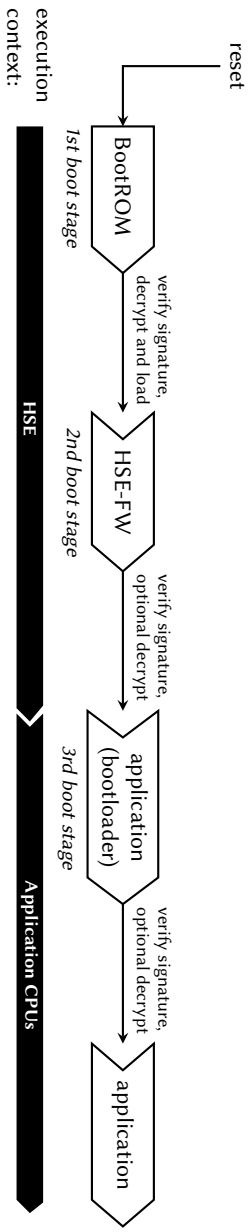


Figure 7.1: Secure boot flow for the S32G274A.

Secondly, the application images need to be installed in non-volatile memory. This is done with the use of Secure Memory Regions (SMRs), which are regions in Non-Volatile Memory (NVM) defined by an address, a length, and an (initial) proof of authenticity (e.g., a digital signature linked to a previously provisioned key). Finally, the user has to specify the Application CPUs for which SMRs require verification before continuing with the boot flow (and which sanctions are applied on failure). The Application (Bootloader) and the Application can be associated with one or more SMRs. The HSE secure boot configuration can be locked by advancing the device lifecycle, which disables any future changes to the configuration.

It should be noted that the above description is a very high-level view: in reality, the S32G274A boot sequence is highly configurable and supports a multitude of options. A particularly interesting one is the ability to use reference proofs of authenticity for SMRs. On initial SMR installation, the HSE-FW will check the initial authenticity proof that was stored in non-volatile memory (e.g., the digital signature). If the initial authenticity proof verifies correctly, the HSE-FW computes a reference authenticity proof that is stored internally in the HSE. As the application has already been authenticated with an initial proof of authenticity, the requirements on the reference proof are lighter. Therefore verification of the reference proof can be much faster than the initial one. During secure boot, the HSE-FW only verifies the SMRs by checking the reference proofs, significantly speeding up the boot procedure. The S32G also supports runtime (periodically or on-demand) attestation, meaning that SMRs can be verified (initial or reference) during the execution.

7.3 S32G274 Post-quantum Secure Boot

In Section 7.2.2 we summarized the S32G platform and its boot flow. In this section, we describe our Dilithium implementation for the HSE core and how this was integrated into the HSE-FW to support its signature verification in the boot flow. Finally, we discuss the installation of the secure memory regions selecting features that are most appropriate for our setting.

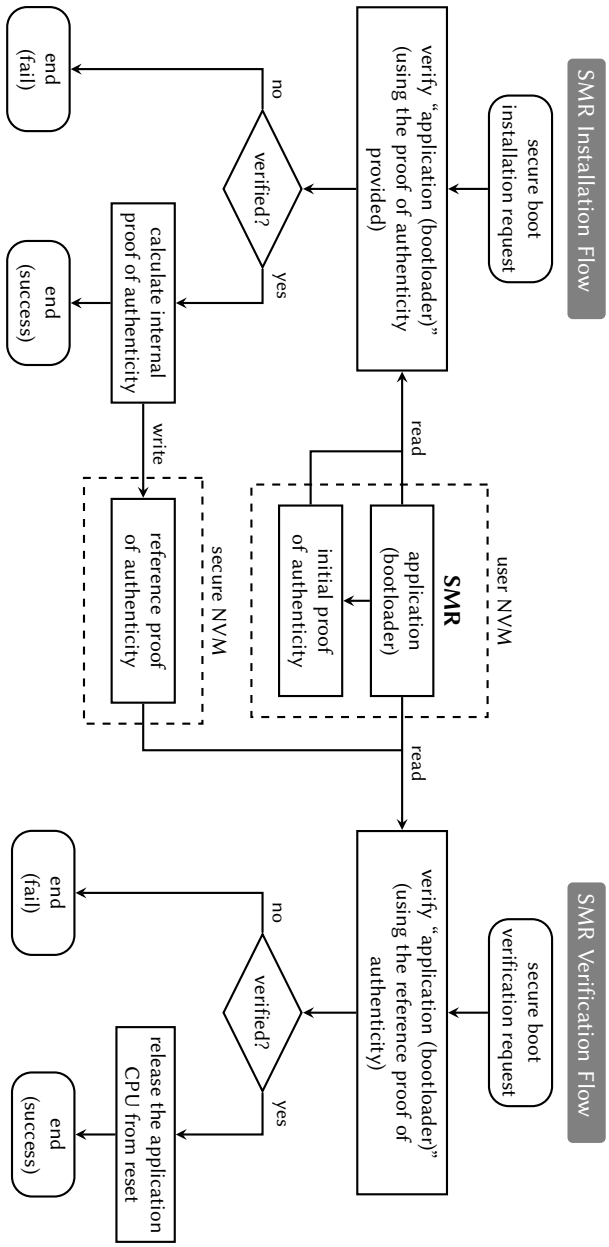


Figure 7.2: Overview of booting using Secure Memory Regions.

7.3.1 Dilithium software

The Dilithium submission to the NIST standardization effort is accompanied by various implementations [LDKL+20]. Some of the parameter sets for Dilithium have also been integrated and optimized for *pqm4*: a testing and benchmarking framework for the Arm Cortex-M4 [PQM4]. The implementations supported in pqm4 provide a good overview of the state-of-the-art performance of the post-quantum algorithms within some constraints related to the Arm Cortex-M4 platform. For example, the total memory available is 112 + 16 KB (SRAM1 and SRAM2). Moreover, it should be noted that these implementations ensure a runtime independent of any secret key material but are *not* protected against active [BDL97; BS97] (faults) or passive [KJJ99] (side-channel) attacks. For critical applications, such as secure boot on vehicle network processors, protection against these advanced attacks is often a minimal requirement.

We implemented the Dilithium algorithms for all parameter sets from scratch and ensured they comply with the proposed specification and pass the Known-Answer-Tests provided in [LDKL+20]. Our main focus is on the signature *verification* since this is the only functionality required in the secure boot flow. For verification the protection against passive attacks is not relevant; a side-channel attack tries to deduce information about the bits of the secret key material used during execution based on, for instance, the observed power consumption of the device. However, no secret key material is used during signature verification. Protection against fault attacks is required since it would be trivial to force acceptance of a wrong signature by introducing a well-targeted fault in the implementation. Our implementation includes countermeasures against single-targeted fault attacks: this is achieved by both adding countermeasures protecting the control flow as well as algorithmic checks to ensure no steps are skipped or memory regions have been altered.

As can be observed from the pqm4 benchmarking framework, the stack consumption of Dilithium is significantly larger compared to the classic public-key counterparts (such as RSA and ECC). The stack requirement for signature verification for Dilithium 3 reported by pqm4 is around 58 KB. Recent work [GKS21] has shown how to reduce this stack to around 10 KB. Our fault attack resistant Dilithium verification code requires less than 3 KB of stack for all parameter sets. This is a huge improvement over previous works: still, it is an order of magnitude larger compared to signature verification based on elliptic curves.

There are two variants of Dilithium specified in the supporting documentation: the main version where symmetric primitives for matrix expansion are instantiated with SHAKE, and a second version where AES is used. The latter was included mostly to demonstrate the efficiency of Dilithium on platforms which do not have support for SHAKE yet or have dedicated hardware support for AES. In this work we only focus on the recommended variant using SHAKE. For the SHAKE implementation we use (a slightly modified version of) the assembly code published in the eXtended Keccak Code Package[1] (XKCP).

7.3.2 Firmware integration

Given a functional Dilithium implementation, the next step is to update the HSE-FW to support its use. This is made easy by the fact that the Dilithium signature verification API (as mandated by NIST) is virtually identical to that of RSA and elliptic-curve-based signature schemes. The main complications arise from the fact that the memory use of Dilithium is higher, both in terms of key and signature size as well as stack. However, keys still easily fit into the key catalog, while 3 KB stack can be handled by the HSE. Hence we observed no significant obstacles in adding Dilithium support to the boot flow.

In order to evaluate and benchmark the integration, we created a simple demo application. For this purpose we require the compiled application images to be accompanied by a Dilithium signature, for which we wrote a stand-alone command-line tool. This tool was written in C, and was built around the avx2 implementation of Dilithium from the CRYSTALS team.[2] Using our signing tool, we pack the compiled application code into a flash image, together with a Dilithium signature and the public key under which the code was signed, and load it together with our demo application image into flash. We also use the demo application as boot loader. On first boot, when secure boot is still disabled, the demo application loads the Dilithium key and signature into the HSE. Following the description in Section 7.2.2, it installs our application code in a Secure Memory Region using the attached digital signature, and enables secure boot on the device. To verify whether the secure boot configuration was effective, we can query the HSE Core Boot status, which contains the info on

[1] https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/ARM/KeccakP-1600-inplace-32bi-armv7m-le-armcc.s

[2] https://github.com/pq-crystals/dilithium

which SMRs were correctly verified during boot. In our development setup we do not advance the lifecycle of the device, as that would brick our development setup.

7.3.3 Performance results

Beyond validating that a functional Dilithium-based secure boot setup is feasible, it is of course interesting to compare its performance to the status quo. When secure boot is enabled, the boot latency is dominated by signature verification. Therefore, it is sufficient to measure Dilithium verification latency, and compare it to the verification latencies of a selection of other signature schemes.

The latency is not only determined by the choice of signature scheme, but also by the length of the application image. All relevant schemes sign and verify in essentially two steps. First, the variable-length message is *pre-hashed* down to a fixed-size digest, possibly including padding, a public key, a commitment, etc. Afterwards the digest is processed to create the final digital signature. Unfortunately, although this step is essentially independent of the signature scheme, the choice of hash function does slightly differ. For example, for ECDSA [ANSI15] a hash function specified in FIPS 180 [NIST15b] should be used (e.g., SHA-256) that is applied only on the message itself, while in EdDSA [JL17] pre-hashing is optional. The Ed25519 instantiation does not pre-hash, reducing the message size implicitly together with a prefix in an application of SHA-512, while Ed25519ph first explicitly reduces the input message using SHA-512. On the other hand, the Dilithium signature scheme signs arbitrary-length messages by hashing them together with the public key, using SHAKE-256. Although the choice of hash function (assuming appropriate length is chosen) is independent of the security of the public-key signature scheme, it can have significant impact on the performance. More concretely, by offering hardware support for SHA-2 and not for (variants of) SHA-3, the S32G274A offers a clear performance benefit for SHA-2. Therefore we investigate two categories of variants: DilithiumX for $X \in \{2, 3, 5\}$ where application images are signed directly with Dilithium, and DilithiumX-ph where a SHA-256 hash over the image is signed instead. To investigate the impact of hashing, we measure the verification of a signature on both a small image (1 KiB) and a larger image (128 KiB).

We distinguish between the *installation* time of an application image where the digital signature of Dilithium is verified (the original proof of authenticity) and *boot* time where only the reference proof is verified. Of course, a user can opt to also

Table 7.1: Latencies of installation (inst.) and boot in milliseconds for supported algorithms on the S32G274A. Key sizes are reported in bytes. The pre-hash (ph) variants of Dilithium first hash the image using SHA-256, and verify the Dilithium signature over the hash.

algorithm	size		1 KiB		128 KiB	
	pk	sig	inst.	boot	inst.	boot
RSA 4K	512	512	2.6	0.0	2.7	0.2
ECDSA-p256	64	64	6.2	0.0	6.4	0.2
Dilithium2	1312	2420	12.1	0.0	158.9	0.2
Dilithium3	1952	3293	17.8	0.0	164.4	0.2
Dilithium5	2592	4595	26.6	0.0	173.3	0.2
Dilithium2-ph	1312	2420	11.1	0.0	11.3	0.2
Dilithium3-ph	1952	3293	16.7	0.0	16.9	0.2
Dilithium5-ph	2592	4595	25.5	0.0	25.7	0.2

verify the proof authenticity on each boot, but the performance impact is large (even in a classical setting), while there are no (significant) security benefits. Because verification of the reference proof does not depend on the choice of digital signature scheme, the boot time is actually not affected by switching to a post-quantum variant. Although the installation for Dilithium is slower than for RSA and elliptic-curve based variants, it is only performed once (or a few times) and its runtime is not as critical. We summarize all of our measurements in Table 7.1.

From our benchmarks, we see that Dilithium verification of small images is 5–10 times slower than RSA 4K and 2–5 times slower than ECDSA-p256, depending on the chosen post-quantum security level. The security level for Dilithium2, Dilithium3 and Dilithium5 is as at least as high as AES-128, AES-192 and AES-256 respectively, for both classical as well as quantum adversaries, which can help guide in choosing the appropriate security level for a use case.

When verifying small images, the Dilithium signature verification completes in less than 30 ms for all variants. Looking at the results for the verification of larger images without pre-hashing, we see latencies up in the hundreds of milliseconds. As mentioned, this is almost completely attributed to the SHAKE-256 hash has that is applied to the image. With additional SHA-256 pre-hashing, the large-image verification latencies are almost equal to the latencies we measure for small images

(actually even lower). It is clear that without hardware support for SHAKE-256, the image verification is dominated by the hashing of the image. In fact, even *with* pre-hashing the dominating cost in Dilithium is the pseudo-random matrix generation using the SHAKE-128 eXtendable Output Function (XOF). Hence, improved latencies for SHAKE variants would significantly help for low-latency signature verification using Dilithium. We do not observe this in the case of RSA 4K and ECDSA-p256, since they hash the image using SHA-256 (for which hardware acceleration is present). However, we re-iterate that the signature verification only impacts *installation* time of the SMR and is irrelevant for the boot time, for which low latency is much more crucial.

7.4 Conclusion

The main challenges that can be expected when migrating from classical signature verification schemes such as RSA or ECC to post-quantum variants such as Dilithium, are an increase in memory (keys, signatures as well as stack) and runtime. The significantly larger public keys and signatures do not cause any real practical problems on our target platform in the setting of vehicle network processors. Moreover, we showed that in this setting of signature verification the amount of stack space required for cryptographic operations needs to be increased only marginally. The performance of Dilithium signature verification is indeed worse than that of ECC/RSA verification. However, as this is only performed during installation time, there is no impact on the boot time itself. We believe a transition to post-quantum secure boot can be considered practical for this application.

8 Dilithium nonce recycling

8.1 Introduction

At its core, the Dilithium signature scheme is not unlike the Schnorr signature algorithm [Sch90], i.e., a zero-knowledge identification scheme which is made non-interactive using the Fiat–Shamir heuristic [FS87]. Such constructions are widely used, for instance in the Ed25519 [BDLS+11] or MEDS [CNPR+23] signature schemes.

In Schnorr, signature generation starts by picking a nonce $\mathbf{y}$ at random. In Dilithium, however, contrary to traditional Schnorr signatures, not every nonce $\mathbf{y}$ will result in a valid signature. For correctness and security, the signature is subjected to several checks. When any of these checks fail, a completely new $\mathbf{y}$ is sampled, and a new candidate signature is generated and scrutinized in turn. Only when a signature passes all the checks, it is output to the user. This construction, where candidate signatures are generated until one of them passes the checks, is called *Fiat–Shamir with Aborts* (FSwA) [Lyu09].

In this chapter we demonstrate that one does not have to resample $\mathbf{y}$ completely. Instead, for one of the four checks, we only need to resample parts of $\mathbf{y}$. This allows one to reuse computations involving the nonce between attempts and leads to a speed-up in signing time in the order of 4–6%, depending on the platform and the Dilithium variant.

Touching nonces in Schnorr signatures and ECDSA is considered to be a dangerous affair. That is because many attacks have been published that have broken schemes or implementations that reused nonces, or where nonce bias could be detected [AFGK+14; ANTT+20; BCP10; BH19; BvSY14]. We recognize this fact and carefully study the security of our proposal: we show that the modified version of Dilithium is as secure as the original.

Contributions. We start this chapter by giving a brief recap of Dilithium and the relevant checks applied during the rejection-sampling loop of the signature generation algorithm. In Section 8.3, we introduce a proposal to slightly optimize the Dilithium signature generation algorithm by reusing some of the nonce material. In Section 8.4, we examine the security of the scheme after (applying the) modification. In Section 8.5, we look at the performance impact of the new construction, by first counting the basic operations and then benchmarking optimized AVX2, Cortex-M4, and Cortex-M3 implementations.

8.2 Dilithium recap

In this section, we will give brief recap of the Dilithium signature scheme [DKLL+20] and will go into more detail about the parts relevant to our optimizations.

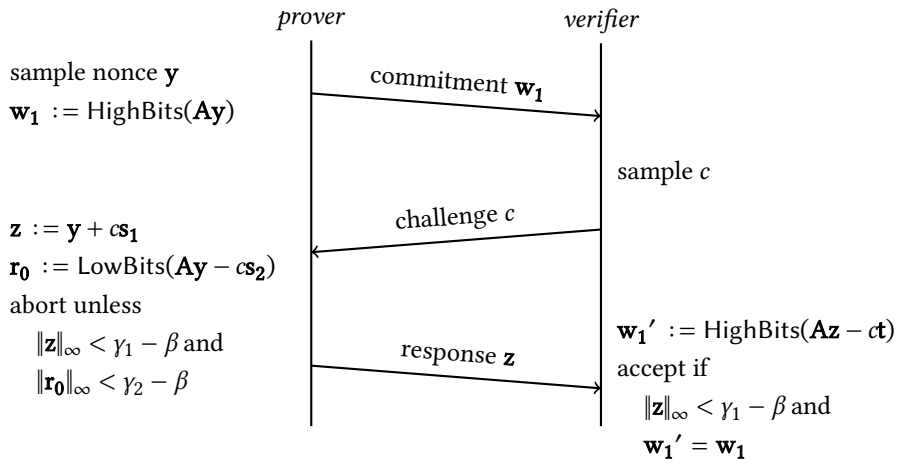
The basic building block of Dilithium are polynomials of degree $n = 256$ with integer coefficients modulo $q = 2^{23} - 2^{13} + 1$ and the rule $X^{256} \equiv -1$ when computing multiplication. Mathematically, these form the ring $R_q := \mathbb{Z}_q[X]/(X^n + 1)$.

The "size" of polynomials plays a crucial role in Dilithium. It is taken to be the size of the largest coefficient, which is its absolute value, so both 1 and $q - 1 = -1$ are considered small. In Dilithium, this size is defined as the infinity norm, i.e., $\|\_\|_\infty$ (see Section 2.2).

The core of the private key are two small vectors over R_q: $\mathbf{s_1} \in R_q^\ell$ and $\mathbf{s_2} \in R_q^k$ sampled uniformly with $\|\mathbf{s_1}\|_\infty, \|\mathbf{s_2}\|_\infty \leq \eta$, where η, k and ℓ depend on the security level. (For NIST level 2, we have $\eta = 2$, $k = 4$, $\ell = 4$, see Section 3.3.3.) The core of the public key is a random $k \times \ell$-matrix $\mathbf{A}$ over R_q together with the vector $\mathbf{t} := \mathbf{As_1} + \mathbf{s_2}$. It is hard to recover $\mathbf{s_1}$ and $\mathbf{s_2}$ from $\mathbf{t}$ and this is known as the *Module Learning With Errors* (MLWE) problem (Definition 3.1).

8.2.1 Underlying identification scheme

Dilithium is based [KLS18] on the following interactive *identification scheme* where a *prover* having access to the private key, demonstrates this fact to a *verifier* that knows the public key, without leaking any information.



The prover generates a random secret nonce[1] $\mathbf{y} \in R_q^\ell$ with all coefficients in $[-\gamma_1, \gamma_1)$ (with $\gamma_1 = 2^{17}$ for security level 2). The prover sends the *commitment* $\mathbf{w_1} = \mathsf{HighBits}(\mathbf{Ay})$ to the verifier, where HighBits and LowBits decompose a vector $\mathbf{x}$ in the following unique way (with $\gamma_2 = \frac{q-1}{88}$ for level 2):

$$\mathsf{HighBits}(\mathbf{x}) \cdot 2\gamma_2 + \mathsf{LowBits}(\mathbf{x}) \;=\; \mathbf{x} \quad \text{and} \quad \|\mathsf{LowBits}(\mathbf{x})\|_\infty \;\leq\; \gamma_2.$$

Note that the prover must only send the higher bits of $\mathbf{w} := \mathbf{Ay}$ for otherwise they would leak $\mathbf{y}$ as $\mathbf{A}$ is likely to be invertible. After receiving $\mathbf{w_1}$, the verifier returns a random *challenge* $c \in R_q$ with τ non-zero coefficients, all either 1 or -1, (with $\tau = 39$ for security level 2). Now the prover computes the *response* $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$. Note that $\|c\mathbf{s_1}\|_\infty$ is not very large, it is at most $\beta := \tau\eta$. Before sending the response, it performs the following two checks on the sizes of $\mathbf{z}$ and $\mathbf{r_0} := \mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2})$,

[1] Nonce as in "number only used once" is misleading: $\mathbf{y}$ is neither a number nor is its single-use the only requirement it should satisfy.

whose importance will become clear later on.

$$\|\mathbf{z}\|_\infty < \gamma_1 - \beta. \qquad (\mathbf{z}\text{-check})$$
$$\|\mathbf{r_0}\|_\infty < \gamma_2 - \beta, \qquad (\mathbf{r_0}\text{-check})$$

If any of these fail, the prover aborts and restarts from the beginning. When eventually receiving a response (after typically around 3 restarts) the verifier accepts whenever $\mathbf{w_1}' := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}) = \mathbf{w_1}$ and $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$.

Without the checks, the scheme wouldn't always work. Indeed, in general

$$\mathbf{w_1}' \equiv \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t}) = \mathsf{HighBits}(\mathbf{Ay} - c\mathbf{s_2}) \neq \mathsf{HighBits}(\mathbf{Ay}) \equiv \mathbf{w_1}$$

as even though $c\mathbf{s_2}$ has small coefficients (also $\leq \beta$) they might still carry into the higher bits and so the verifier won't trust the prover. This problem is solved by making sure that the subtraction in $\mathbf{Ay} - c\mathbf{s_2}$ does not overflow into $\mathbf{r_1}$, which is ensured by the $\mathbf{r_0}$-check [KLS18, Eq. 3]. A different issue is that $\mathbf{r_0}$ and $\mathbf{z}$ might leak information on respectively $\mathbf{s_1}$ and $\mathbf{s_2}$ if they have large coefficients. For instance, if $\mathbf{z}_1 = \gamma_1 + \beta - 1$, then we must have $\mathbf{y}_1 = \gamma_1 - 1$ and $(c\mathbf{s_1})_1 = \beta$. Both checks prevent this kind of leakage.

The Dilithium scheme is *accepting honest-verifier zero-knowledge* (acHVZK): that means we can replicate the distribution of $(c, \mathbf{z})$ in successful sessions without having access to the secret key.[2] Not all $(c, \mathbf{z})$ can occur, but those that do, occur with equal probability. Now, to simulate a session, pick random $(c, \mathbf{z})$ with $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, $\|\mathsf{LowBits}(\mathbf{Az} - c\mathbf{t})\|_\infty < \gamma_2 - \beta$, and c as a verifier would sample it. Every pair $(c, \mathbf{z})$ that occurs in a real session could be generated as such: the first requirement is the $\mathbf{z}$-check and the second the $\mathbf{r_0}$-check because $\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} - c\mathbf{s_2}$. Conversely, given such a simulated pair, set $\mathbf{y} := \mathbf{z} - c\mathbf{s_1}$. This $\mathbf{y}$ could have been picked as $\|\mathbf{y}\|_\infty < \gamma_1$ for $\|c\mathbf{s_1}\|_\infty \leq \beta$. With this nonce, the prover will pick the right response $\mathbf{z}$. With the first two requirements, we also made sure that the prover will pass the $\mathbf{z}$-check and $\mathbf{r_0}$-check. And so in the same way as we prove correctness in a regular run, we see that the verifier will accept. Thus we can indeed simulate the sessions perfectly.

[2] The commitment $\mathbf{w_1}$ is not included as in a successful session it is computed from the challenge and response.

8.2.2 Vanilla Dilithium

As covered in Chapter 3, the identification scheme is turned into a signature scheme using the Fiat–Shamir transform [FS87]. A signature on a message M is given by a pair $(c, \mathbf{z})$ of a challenge c and a response $\mathbf{z}$ of a successful interaction of the identification scheme, where the challenge is not picked randomly by a verifier, but rather computed as $\mathsf{H}(M \,\|\, \mathbf{w_1})$ for a hash function H that ranges over the challenge space B_τ. After applying the Fiat–Shamir transform, we get the non-interactive signature generation algorithm as listed in Algorithm 8.1.

To check a signature, a verifier (like in the identification scheme) first checks $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and then computes $\mathbf{w_1}' := \mathbf{Az} - c\mathbf{t}$, which should be equal to the original commitment $\mathbf{w_1}$. The verifier does not have access to the original commitment (as it was not included in the signature), but can check whether it was correct by recomputing the challenge using the supposed commitment and comparing it against the one included in the signature.

Algorithm 8.1: Simplified vanilla Dilithium

$\mathsf{Sign}_{\text{vanilla}}(sk = (\mathbf{A}, \mathbf{t}, \mathbf{s_1}, \mathbf{s_2}), M)$

1: $\kappa := 0$
2: `sign`: **loop**
3: **for** i **from** 0 **up to** $\ell - 1$ **do**
4: $\mathbf{y}_i := \mathsf{ExpandMask}(\kappa); \kappa := \kappa + 1$
5: $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay})$
6: $c := \mathsf{H}(M \,\|\, \mathbf{w_1})$
7: $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
8: **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ $\mathbf{z}$-check
9: **continue** `sign`
10: **if** $\|\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2}, \gamma_2)\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
11: **continue** `sign`
12: **return** $(c, \mathbf{z})$

The full Dilithium scheme is rather more complex, as it includes tricks to decrease signature and key sizes (such as only publishing the higher bits of $\mathbf{t}$) while increasing performance (by sampling in the NTT domain.) These details, however, do not impact the security of the scheme or our proposal and will direct the curious reader Chapter 3.

8.3 Our proposal

In vanilla Dilithium, to create a signature, we randomly sample a nonce $\mathbf{y}$ and then compute in sequence the commitment $\mathbf{w_1}$, challenge c and response $\mathbf{z}$. Not every $\mathbf{y}$ will lead to a valid identification session as the $\mathbf{z}$-check or $\mathbf{r_0}$-check might fail. In that case, we completely start over again with a new nonce $\mathbf{y}$.

8.3.1 Resample only the prefix of y after failed z-check

Note that by the definition of the norm, the $\mathbf{z}$-check involves the following ℓ subchecks, one for each component of $\mathbf{y}$: $\|(\mathbf{y} + c\mathbf{s_1})_i\|_\infty < \gamma_1 - \beta$. If the first subcheck fails (without having performed the other checks or subchecks), then instead of aborting completely and resampling all elements of $\mathbf{y}$, we propose to resample $\mathbf{y}_1$ but keep $\mathbf{y}_2, \dots, \mathbf{y}_\ell$ for the next iteration. This allows one to reuse the computations of $\mathbf{A}_{ij}\mathbf{y}_j$ for $j \neq 1$, which were required to compute c via $\mathbf{Ay}$. As $\mathbf{Ay}$ changes, the commitment $\mathbf{w_1}$ changes with high probability (cf. [KLS18, Lemma C.1]) and the challenge c will be different after this partial abort.

If $\mathbf{z}$-check fails at $\mathbf{y}_2$ (after $\mathbf{y}_1$ passed) then we cannot reuse $\mathbf{y}_1$, because it will have to pass the check for at least one other challenge c. This will introduce a bias in $\mathbf{y}_1$, although it is unclear to us whether this bias could lead to a practical attack. Instead we propose to resample only $\mathbf{y}_1, \dots, \mathbf{y}_i$ if the first check fails at $\mathbf{y}_i$ (and only having checked $\mathbf{y}_1, \dots, \mathbf{y}_i$).

After resampling, this new $\mathbf{y}$ is computationally indistinguishable from a freshly generated one. Indeed, its only bias is that $\mathbf{y}_{i+1}, \dots, \mathbf{y}_\ell$ has been used to compute the previous challenge c, but we assume that the hash function H behaves as a random oracle.

In Algorithm 8.2 we provide the first modified version of the Dilithium signing procedure. We label the alteration[3] with alt-$\mathbf{z}$ and denote the algorithm with $\mathsf{Dilithium}_{\text{alt-}\mathbf{z}}$. In the new $\mathsf{Sign}_{\text{alt-}\mathbf{z}}$ routine, the variable ξ is introduced to keep track of how many $\mathbf{y}$-elements are to be resampled after a failed $\mathbf{z}$-check.

Note that signatures are compatible between vanilla and modified Dilithium: a signature generated by one will be verified by the other. Indeed, we did not change the verification routine. However, when using deterministic signatures, signing the same

[3] We say *alteration* (abbreviated as alt), instead of *modification* (abbreviated as mod), to prevent confusion with *modulo* (also abbreviated as mod).

Algorithm 8.2: Reusing polynomials in $\mathbf{y}$ after partially checking $\mathbf{z}$.

$\mathsf{Sign}_{\text{alt-}\mathbf{z}}(sk = (\mathbf{A}, \mathbf{t}, \mathbf{s_1}, \mathbf{s}_2), M)$

1: $\kappa := 0; \xi := \ell$
2: `sign`: **loop**
3: **for** i **from** 1 **up to** ξ **do** ▷ Only (re)sample the first ξ elements of $\mathbf{y}$
4: $\mathbf{y}_i := \mathsf{ExpandMask}(\kappa); \kappa := \kappa + 1$
5: $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$
6: $c \in B_\tau := \mathsf{H}(M \| \mathbf{w_1})$
7: $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
8: **for** i **from** 1 **up to** ℓ **do**
9: **if** $\|\mathbf{z}_i\|_\infty \geq \gamma_1 - \beta$ **then** ▷ $\mathbf{z}$-check
10: $\xi := i$
11: **continue** `sign`
12: $\xi := \ell$
13: **if** $\|\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2}, \gamma_2)\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
14: **continue** `sign`
15: **return** $(c, \mathbf{z})$

message using the same secret key will lead to two different signatures on the same message if different implementations are used to sign the same message. Moreover, the modified versions of Dilithium are not specification compliant.

8.3.2 Compatibility with streaming implementations

Some implementations (e.g. [GKS21, Strategy 3]), optimize for memory-constrained environments. To use memory efficiently, they typically compute $\mathbf{w} = \mathbf{Ay}$ one element at a time where each component of $\mathbf{A}$ and $\mathbf{y}$ is generated on the fly.

With our modifications, we are not resampling exactly ℓ elements of $\mathbf{y}$ during each loop iteration. Some polynomials $\mathbf{y}_i$ might have been fixed some loop iterations ago, using an old κ that could have been forgotten. To ensure compatibility with other implementations, the streaming implementation will have to keep track of the κ values that were used to generate the elements of $\mathbf{y}$ that are still in use.

8.4 Security

The security claim of Dilithium is *strong unforgeability under chosen message attacks* (SUF-CMA), of which we have provided a high-level intuition in Section 3.2.4. The original security proof is given in [KLS18] and its references. However, the proof of the *Fiat–Shamir with aborts* (FSwA) heuristic was found to be incomplete. The gap was identified in [BBDD+23; DFPS23] and the proof was closed. Using [BBDD+23], we will show that that proof still applies, even though the value of ϵ (denoting the guessing probability of $\mathbf{w_1}$) might be reduced.

8.4.1 Adapting the ROM proof of [BBDD+23]

[BBDD+23] contains both a proof in the random oracle model (ROM) and a proof in the quantum random oracle model (QROM). In this chapter, we will use the existing ROM proof to analyze the security of our proposals, and leave the QROM analysis for future work. To reduce the security of (vanilla) Dilithium from UF-CMA to UF-NMA in the ROM, [BBDD+23] uses a hybrid proof with three main steps: In the first hybrid step, they replace the signing oracle $\mathsf{Sign}(M)$ with the $\mathsf{Prog}(M)$ oracle. $\mathsf{Prog}(M)$, instead of querying a challenge $c := \mathsf{H}(M, \mathbf{w_1})$ from the random oracle H, samples c uniformly and programs $\mathsf{H}(M, \mathbf{w_1}) := c$ accordingly. In the second step, $\mathsf{Prog}(M)$ is replaced with $\mathsf{Trans}(M)$, which hoists the random-oracle programming out of the rejection sampling loop and programs the random-oracle once when an accepting signature is found. Then, in the third step, $\mathsf{Trans}(M)$ is replaced with $\mathsf{ZKSim}(M)$, in which the signature $(\mathbf{w_1}, c, \mathbf{z})$ is generated by the simulator (instead of a signing routine).

For the modified version of Dilithium, we will have to adjust the hybrid steps to incorporate the nonce-reusing aspect of the scheme. In Figure 8.1, we list the adversary's oracle for each hybrid step. In these algorithms, $\mathsf{Comm}(\mathsf{sk})$ denotes the commitment-generating part of the Dilithium algorithm. The modified-Dilithium variant $\mathsf{Comm}^{\text{alt-}\mathbf{z}}(\mathsf{sk}, \mathbf{y}, \xi)$ also takes the already-present vector $\mathbf{y}$ as input, plus ξ which is the number of $\mathbf{y}$ elements that should be regenerated. $\mathsf{Resp}(\mathbf{w_1}, c, \mathbf{y})$ denotes the computation that generates $\mathbf{z}$ from $\mathbf{w_1}$, c, and $\mathbf{y}$. $\mathsf{Resp}(\mathbf{w_1}, c, \mathbf{y})$ includes the rejection-sampling checks, and returns $\perp$ (indicating failure) if any of the checks fail. The modified-Dilithium variant $\mathsf{Resp}^{\text{alt-}\mathbf{z}}(\mathbf{w_1}, c, \mathbf{y})$ additionally returns an up-

$\underline{\mathsf{Sign}^{\text{alt-}\mathbf{z}}(M)\text{:}}$

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: **repeat**
4: $\quad (\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\text{alt-}\mathbf{z}}(\mathsf{sk}, \mathbf{y}, \xi)$
5: $\quad c := \mathsf{H}(\mathbf{w_1}, M)$
6: $\quad (\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\text{alt-}\mathbf{z}}(\mathbf{w_1}, c, \mathbf{y})$
7: **until** $\mathbf{z} \neq \bot$
8: **return** $(\mathbf{w_1}, \mathbf{z})$

$\underline{\mathsf{Prog}^{\text{alt-}\mathbf{z}}(M)\text{:}}$

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: **repeat**
4: $\quad (\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\text{alt-}\mathbf{z}}(\mathsf{sk}, \mathbf{y}, \xi)$
5: $\quad \mathsf{H}(\mathbf{w_1}, M) := c \leftarrow C$
6: $\quad (\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\text{alt-}\mathbf{z}}(\mathbf{w_1}, c, \mathbf{y})$
7: **until** $\mathbf{z} \neq \bot$
8: **return** $(\mathbf{w_1}, \mathbf{z})$

$\underline{\mathsf{Trans}^{\text{alt-}\mathbf{z}}(M)\text{:}}$

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: **repeat**
4: $\quad (\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\text{alt-}\mathbf{z}}(\mathsf{sk}, \mathbf{y}, \xi)$
5: $\quad c \leftarrow C$
6: $\quad (\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\text{alt-}\mathbf{z}}(\mathbf{w_1}, c, \mathbf{y})$
7: **until** $\mathbf{z} \neq \bot$
8: $\mathsf{H}(\mathbf{w_1}, M) := c$
9: **return** $(\mathbf{w_1}, \mathbf{z})$

$\underline{\mathsf{Sim}^{\text{alt-}\mathbf{z}}(M)\text{:}}$

1: $(\mathbf{w_1}, c, \mathbf{z}) \leftarrow \mathsf{ZKSim}^{\text{alt-}\mathbf{z}}(\mathsf{pk})$
2: $\mathsf{H}(\mathbf{w_1}, M) := c$
3: **return** $(\mathbf{w_1}, \mathbf{z})$

Figure 8.1: The oracles $\mathsf{Sign}^{\text{alt-}\mathbf{z}}$, $\mathsf{Prog}^{\text{alt-}\mathbf{z}}$, $\mathsf{Trans}^{\text{alt-}\mathbf{z}}$, and $\mathsf{Sim}^{\text{alt-}\mathbf{z}}$, which are used during the security analysis.

dated value for ξ, which corresponds to how many elements of $\mathbf{y}$ are discarded and regenerated during the next iteration.

In Section 8.4.2 we will quantify the security loss between the $\mathsf{Sign}^{\text{alt-}\mathbf{z}}(M)$ game and the $\mathsf{Trans}^{\text{alt-}\mathbf{z}}(M)$ game, based on the reasoning from [BBDD+23]. Then, in Section 8.4.3 we list the modified transcript simulator, demonstrating that the signature generation of $\mathsf{Trans}^{\text{alt-}\mathbf{z}}(M)$ is zero-knowledge. At that point, we have reduced the construction from UF-CMA to UF-NMA. The rest of the security is unaffected in the modified version of Dilithium.

8.4.2 From $\mathsf{Sign}^{\mathbf{alt\text{-}z}}(M)$ to $\mathsf{Trans}^{\mathbf{alt\text{-}z}}(M)$

We copy and tweak the proof from [BBDD+23] to determine the new bounds for $\Delta^{\mathsf{Sign}^{\mathbf{alt\text{-}z}} \to \mathsf{Prog}^{\mathbf{alt\text{-}z}}}_{p,\epsilon_0,\epsilon}(q_S, q_H)$ and $\Delta^{\mathsf{Prog}^{\mathbf{alt\text{-}z}} \to \mathsf{Trans}^{\mathbf{alt\text{-}z}}}_{p,\epsilon_0,\epsilon}(q_S, q_H)$. Consider a collection of hybrid oracles Hyb^k which program the random oracle (RO) during the first k iterations, and make regular RO calls during the other iterations. This way we transform from $\mathsf{Sign}^{\mathbf{alt\text{-}z}} = \mathsf{Hyb}^0$ into $\mathsf{Prog}^{\mathbf{alt\text{-}z}} = \mathsf{Hyb}^\infty$ by each time replacing the oracle by an oracle in which the RO is programmed *one* additional time. Likewise, we will replace $\mathsf{Sign}^{\mathbf{alt\text{-}z}}$ with $\mathsf{Prog}^{\mathbf{alt\text{-}z}}$ in one of the adversary's queries at a time.

$\underline{\mathsf{Hyb}^k(M)\text{:}}$

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: $i := 0$
4: **repeat**
5: $\quad(\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\mathbf{alt\text{-}z}}(\mathsf{sk}, \mathbf{y}, \xi)$
6: $\quad$**if** $i < k$ **then**
7: $\quad\quad\mathsf{H}(\mathbf{w_1}, M) := c \leftarrow C$
8: $\quad$**else**
9: $\quad\quad c := \mathsf{H}(\mathbf{w_1}, M)$
10: $\quad(\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\mathbf{alt\text{-}z}}(\mathbf{w_1}, c, \mathbf{y})$
11: $\quad i := i + 1$
12: **until** $\mathbf{z} \neq \bot$
13: **return** $(\mathbf{w_1}, \mathbf{z})$

Figure 8.2: Hybrid signing oracle in which the first k iterations, the random oracle is programmed, and the random oracle is called in all subsequent iterations. When increasing k, Hyb^k is gradually transformed from $\mathsf{Hyb}^0 = \mathsf{Sign}^{\mathbf{alt\text{-}z}}$ to $\mathsf{Hyb}^\infty = \mathsf{Prog}^{\mathbf{alt\text{-}z}}$.

So, in total, we will have $q_S \cdot \kappa$ steps, where q_S is the number of queries that the adversary is allowed to make, and κ is the number of iterations after which the signature generation is aborted in its entirety. Consider the hybrid step (i, j) in which the adversary's queries 0 to $i-1$ are answered by $\mathsf{Prog}^{\mathbf{alt\text{-}z}}$; before the step, query i is answered by Hyb^j; after the step, query i is answered by Hyb^{j+1}; and the adversary's queries $i+1$ to $q_S - 1$ are answered by $\mathsf{Sign}^{\mathbf{alt\text{-}z}}$.

Both scenarios behave identically, unless iteration j is reached during query i, and $(\mathbf{w_1}, M)$ is already in the domain of the RO. We can bound the probability of this bad

event occurring by

$$\delta_{i,j,\epsilon_j} := p^j \epsilon_j \left(\frac{i}{1-p} + q_H + j \right),$$

where

- p is the rejection probability;
- ϵ_j is a minimum bound on the guessing probability of $\mathbf{w_1}$ during iteration j of the rejection sampling loop;
- $\frac{i}{1-p}$ oracle queries are included in the domain of the RO due to i previous queries to Prog;
- q_H oracle queries are included in the RO domain because of the adversary's RO-query allowance; and
- j queries are included because at iteration $j+1$, the Hyb^{j+1} will have made j queries to the RO during the same signature's generation.

The value of δ_{i,j,ϵ_j} is the same as is reported by [BBDD+23], except for one aspect: In [BBDD+23], the guessing probability of $\mathbf{w_1}$ is a single constant, whereas with our proposals the guessing probability of $\mathbf{w_1}$ is not constant across loop iterations of the signature-generation oracles.

With the proposal, the first iteration of the rejection-sampling loop generates $\mathbf{y}$ and $\mathbf{w_1}$ identical to vanilla Dilithium. As such, the guessing probability of the first commitment (i.e., ϵ_0) is equal to the guessing probability in vanilla Dilithium (ϵ in [BBDD+23]). However, subsequent iterations may reuse the tail of $\mathbf{y}$. This results in a correlation between the different commitment values, which increases the guessing probability of ϵ_j where $j > 0$. We will get back to that in Section 8.4.4. For now, we will just consider ϵ_0 as the guessing probability of $\mathbf{w_1}$ in iteration $j = 0$, and ϵ is the guessing probability of $\mathbf{w_1}$ for iterations $j > 0$.

We take a sum over all the steps in this game hop, i.e., q_S steps for the number of queries, and κ steps for the number of Hyb^k hybrids. This results in a total loss of

$$\sum_{i=0}^{q_S-1} \left(p^\kappa + \sum_{j=0}^{\kappa-1} \delta_{i,j,\epsilon_j} \right) = \sum_{i=0}^{q_S-1} \left(p^\kappa + \left(\delta_{i,0,\epsilon_0} - \delta_{i,0,\epsilon} \right) + \sum_{j=0}^{\kappa-1} \delta_{i,j,\epsilon} \right).$$

Taking the limit of $\kappa \to \infty$ results in[4]

$$\begin{aligned}\Delta_{p,\epsilon_0,\epsilon}^{\mathsf{Sign}^{\text{alt-}\mathbf{z}} \to \mathsf{Prog}^{\text{alt-}\mathbf{z}}} &= \lim_{\kappa\to\infty}\left(\sum_{i=0}^{q_S-1}\left(p^\kappa + \delta_{i,0,\epsilon_0} - \delta_{i,0,\epsilon} + \sum_{j=0}^{\kappa-1}\delta_{i,j,\epsilon}\right)\right)\\ &\leq q_S(\epsilon_0-\epsilon)\left(\frac{q_S-1}{2(1-p)}+q_H\right) + q_S\epsilon\frac{q_H}{1-p} + q_S\epsilon\frac{q_S+1}{2(1-p)^2}. \qquad (8.1)\end{aligned}$$

We continue with the step from $\mathsf{Prog}^{\text{alt-}\mathbf{z}}$ to $\mathsf{Trans}^{\text{alt-}\mathbf{z}}$, still closely following the proof from [BBDD+23]. The new hybrid oracle Hyb_2 (the updated version of Hyb_2 from [BBDD+23]) is listed in Figure 8.3.

$\underline{\mathsf{Hyb}_2^k(M)\text{:}}$

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: $i := 0$
4: **repeat**
5: $\quad(\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\text{alt-}\mathbf{z}}(\mathsf{sk}, \mathbf{y}, \xi)$
6: $\quad c \leftarrow C$
7: $\quad$**if** $i \geq k$ **then**
8: $\quad\quad \mathsf{H}(\mathbf{w_1}, M) := c \leftarrow C$
9: $\quad(\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\text{alt-}\mathbf{z}}(\mathbf{w_1}, c, \mathbf{y})$
10: $\quad i := i + 1$
11: **until** $\mathbf{z} \neq \bot$
12: **if** $i < k$ **then**
13: $\quad \mathsf{H}(\mathbf{w_1}, M) := c$
14: **return** $(\mathbf{w_1}, \mathbf{z})$

Figure 8.3: Hybrid signing oracle in which, during the first k iterations, the random oracle is programmed only once, and the random oracle is programmed every time during all subsequent iterations. When increasing k, Hyb^k is gradually transformed from $\mathsf{Hyb}^0 = \mathsf{Prog}^{\text{alt-}\mathbf{z}}$ to $\mathsf{Hyb}^\infty = \mathsf{Trans}^{\text{alt-}\mathbf{z}}$.

Again, both scenarios behave almost identically. However, when iteration j is reached and the candidate signature is rejected during that iteration, H has one more $(\mathbf{w_1}, M)$ pair in its domain. The bad event occurs when the adversary manages to query H for that same $(\mathbf{w_1}, M)$ pair. The probability of this bad event occurring is

[4] Derivation is listed in Appendix 8.B.

bounded by

$$\delta'_{i,j,\epsilon_j} = p^j q_H \epsilon_j.$$

After taking the sum over all the steps and taking the limit of $\kappa \to \infty$ this results in

$$\begin{aligned}
\Delta^{\mathsf{Sign}^{\mathsf{alt}} \to \mathsf{Prog}^{\mathsf{alt}}}_{p,\epsilon_0,\epsilon} &= \lim_{\kappa\to\infty} \sum_{i=0}^{q_S-1} \left(p^\kappa + \sum_{j=0}^{\kappa-1} \delta'_{i,j,\epsilon_j} \right) \\
&= \sum_{i=0}^{q_S-1} \left(\delta'_{i,0,\epsilon_0} - \delta'_{i,0,\epsilon} + \sum_{j=0}^{\infty} \delta'_{i,j,\epsilon} \right) \\
&= \sum_{i=0}^{q_S-1} \left((\epsilon_0 - \epsilon) p^0 q_H + \sum_{j=0}^{\infty} p^j q_H \epsilon \right) \\
&= \sum_{i=0}^{q_S-1} \left(\frac{(\epsilon_0 - \epsilon)(1-p) q_H}{1-p} + \epsilon \frac{q_H}{1-p} \right) \\
&= q_S q_H \frac{\epsilon_0 + p(\epsilon - \epsilon_0)}{1-p}.
\end{aligned} \tag{8.2}$$

8.4.3 Zero-knowledgeness of $\mathsf{Trans}^{\mathbf{alt\text{-}z}}(M)$

In $\mathsf{Trans}^{\mathsf{alt}\text{-}\mathbf{z}}(M)$, $\mathbf{z}$ is computed element-wise, as $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$ as always. At this point, c is uniformly sampled from the challenge space B_τ, instead of being provided by the random oracle. Consequently, c is now completely independent of $\mathbf{y}$. Therefore, there is no statistical dependence left between the elements of $\mathbf{z}$, and after the $\mathbf{z}$-check, $\mathbf{z}$ is completely uniform.

As such, we can replace the computation of $\mathbf{z}$ with $\mathbf{z} \xleftarrow{\$} S_{\gamma_1-\beta}$, computing $\mathbf{r_0}$ as $\mathbf{r_0} := \mathbf{Az} - c\mathbf{t}$. This leads to the ZKSim(pk) algorithm, as listed in Figure 8.4, which perfectly simulates the transcripts of $\mathsf{Trans}^{\mathsf{alt}\text{-}\mathbf{z}}(M)$. It is the same simulator as that of vanilla Dilithium.

8.4.4 Min-entropy of $\mathbf{w_1}$

As we saw in the previous sections, the distribution of $\mathbf{y}$ is slightly different from the distribution of $\mathbf{y}$ in vanilla Dilithium. This resulted in a separation from the guessing probability of $\mathbf{w_1}$ for the first iteration ϵ_0 and the guessing probability for every subsequent iteration ϵ in the security analysis. Each guessing probability is

ZKSim($(\mathbf{A}, \mathbf{t}) := \mathsf{pk}$):

repeat
  $c \xleftarrow{\$} B_\tau$
  $\mathbf{z} \xleftarrow{\$} S^{\ell-1}_{\gamma_1-\beta}$
  $\mathbf{r_1} := \mathsf{HighBits}(\mathbf{Az} - c\mathbf{t})$
  $\mathbf{r_0} := \mathsf{LowBits}(\mathbf{Az} - c\mathbf{t})$
until $\|\mathbf{r_0}\|_\infty < \gamma_2 - \beta$
return $(\mathbf{r_1}, c, \mathbf{z})$

Figure 8.4: Simulator for $\mathsf{Trans}^{\text{alt-}\mathbf{z}}$ transcripts.

related to the min-entropy of the value by $H_\infty(X) = -\log_2 x$ where x is the guessing probability of X.

For vanilla Dilithium it is shown that, with overwhelming probability, $\mathbf{A}$ is generated such that the min-entropy of $\mathbf{w_1}$ is at least 117 bits [KLS18, Lemma C.1].[5] Even though the appendix of [BBDD+23] has included a much more elaborate analysis of the min-entropy of the commitment, for simplicity's sake we base our reasoning on the original Dilithium paper. We adapt their proof to our situation.

Recall $\mathbf{w_1} = \mathsf{HighBits}(\mathbf{w})$ and $\mathbf{w} = \mathbf{Ay}$. For brevity, write $w_{11} := (\mathbf{w_1})_1$. Let W be the set of those w with $\mathsf{HighBits}(w) = w_{11}$. By definition of HighBits, the size of W is at most $(2\gamma_2 + 1)^n$. Note $\mathbf{w_1} = \sum_j \mathbf{A}_{j,1} y_j$. Assume for now that there is an invertible element $\mathbf{A}_{i,1}$ in the first column of $\mathbf{A}$. Then

$$Y := \left\{\mathbf{y}_1 ; \sum_{0\le j\le \ell} \mathbf{A}_{j1}\mathbf{y}_j = \mathbf{w_1}\right\} = \mathbf{A}_{i1}^{-1}\Big(W - \sum_{j\neq i} \mathbf{A}_{j1}\mathbf{y}_j\Big). \tag{8.3}$$

Hence Y has the same number of elements of W. Crucially in our modification of Dilithium, the distribution of $\mathbf{y}_1$ is still uniform, and so the chance we get one that leads to w_{11} is

$$\Pr\left[\mathbf{y}_1 \in Y\right] = \frac{\#Y}{\#\tilde{S}_{\gamma_1}} \le \left(\frac{2\gamma_2 + 1}{2\gamma_1}\right)^n. \tag{8.4}$$

[5] For the original version of Dilithium as published in [KLS18] the min-entropy is with a high probability ($\ge 1 - 2^{-179}$ for Dilithium2) of at least 255 bits. With the updated parameters of Dilithium *round 3*, the min-entropy is with an overwhelming probability ($\ge 1 - 2^{-239}$ for Dilithium2) of at least 117 bits.

Thus, if there is an invertible element in the first column of $\mathbf{A}$, then the min-entropy of the resulting commitment $\mathbf{w_1}$ is at least

$$-\log_2 \max_{\mathbf{y}_1 \in \tilde{S}_{\gamma_1}} \Pr\left[\mathbf{y}_1 \in Y\right] = -n \log_2 \frac{2\gamma_2 + 1}{2\gamma_1} \geq 117, \tag{8.5}$$

resulting in a guessing probability of $\epsilon \leq 2^{-117}$. Therefore, we conclude that as long as the assumption on $\mathbf{A}$ holds, the guessing probability of $\mathbf{w_1}$ is not reduced. In other words $\epsilon = \epsilon_0$.

A min-entropy of 117 seems low, especially because the min-entropy needs to be much higher than the scheme's claimed security level in bits. However in [KLS18] it is noted that the bounded number of bits is probably far from the real min-entropy. As the range of $\mathsf{HighBits}(\mathbf{A}\,\cdot)$ is very large, upwards of 2^{17960}, and heuristically close to uniform, it is very likely that the min-entropy is much larger. Additionally, there is another result [KLS18, Lemma 4.7] which shows that for smaller γ_1 and γ_2 the min-entropy (which heuristically should then be smaller) is upwards of 1000 bits, without needing an invertible element in $\mathbf{A}$. Therefore, even if none of the elements in $\mathbf{A}$ are invertible, it seems unlikely that the min-entropy of $\mathbf{w_1}$ is ever dangerously small in (the modified) Dilithium. Moreover, [BBDD+23, Appendix A] includes a more thorough analysis of the min-entropy which again leads to an adequately low guessing probability. At this point it is unclear to us whether that analysis is also applicable to our modifications.

The assumption that the first column of $\mathbf{A}$ contains an invertible element is different from vanilla Dilithium, where the invertible element can exist in *any* column of $\mathbf{A}$. The probability that some uniformly sampled polynomial is invertible is $\left(1 - \frac{1}{q}\right)^n \geq 1 - \frac{n}{q}$. Thus, the chance that none of the polynomials in the first column of $\mathbf{A}$ is invertible is at most $\left(\frac{n}{q}\right)^k$. This probability is the highest for the Dilithium2 parameter set, where $k = 4$ and this probability is approximately 2^{-60}. Technically, there is no single unequivocal way to judge whether this probability is too high, because the UF-CMA security model does not allow the adversary to trigger key-generation events and we anticipate that the real probability is much lower. However, from a practical perspective, even for random-chance events, 2^{-60} is just too high. Fortunately, we can easily work around the issue by always regenerating the ith element of $\mathbf{y}$ instead of the 1st element, where i corresponds to a column of $\mathbf{A}$ that contains at least one invertible

element.[6] The overhead of this approach will be minimal, and the probability bound will be completely restored (to $2^{-\ell \cdot 60}$). A less complicated fix could be to increase the minimum number of $\mathbf{y}$ elements that are to be sampled freshly. For example, when we require that there are always at least 2 freshly generated $\mathbf{y}$ elements, the probability is reduced to 2^{-120}. This still leads to considerable speed-ups for Dilithium. Still, we encourage further research towards finding better bounds both for the min-entropy of $\mathbf{w_1}$ as well as the unlikeliness of $\mathbf{A}$.

8.5 Performance

8.5.1 Operations saved

By not resampling the complete vector $\mathbf{y}$ every time a $\mathbf{z}$-check fails, we save a bit of computation time, that was originally spent generating $\mathbf{y}$ and computing $\mathbf{w} := \mathbf{Ay}$.

Using a Sage script, we estimate the potential performance improvement by simulating the rejection-sampling loop up to the second check. Note that this does *not* include the expansion of $\mathbf{A}$.[7]

The simulations count the number of $\mathbf{y}$ elements that have been sampled, and count the number of calls to `KeccakF1600_StatePermute` (the SHA3/SHAKE primitive) and `NTT`. For completeness, we also include calls to the inverse NTT ($\texttt{NTT}^{-1}$), even though the number should be the same for both scenarios.[8] The results are listed in Table 8.1.

For every mode of Dilithium, we reduce the number of $\mathbf{y}$-component generations: up to 20% of the total number of generated polynomials, in the case of Dilithium3. This saving is reflected in the total number of `KeccakF1600_StatePermute` calls (14% less) and the number of computed `NTT`s (16% less.) There is (as expected) no change in the number of computed $\texttt{NTT}^{-1}$s.

The theoretical counts provide a useful high-level intuition of the speedup that our optimizations provide. However, as performance of these primitives (and their subtle interaction) varies per platform, we continue with measurements on actual implementations on various platforms.

[6] We also have to make sure that the corresponding element of $\mathbf{z}$ is checked *first* during the $\mathbf{z}$-check.

[7] To decrease the size of the public key, Dilithium does not store $\mathbf{A}$ in the public key, but rather a seed from which $\mathbf{A}$ can be reconstructed.

[8] The slight differences in Table 8.1 are due to the stochastic nature of the simulation.

Table 8.1: Average number of sampled $\mathbf{y}$ elements, calls to `KeccakF1600_StatePermute`, `NTT`, and $\mathtt{NTT}^{-1}$ in the Dilithium rejection-sampling loop; using unmodified Dilithium signing, and using the modification proposed in this chapter. Percentages indicate the relative number of calls for each operation (lower is better). Averages were computed over 100 000 runs.

		baseline	alt-$\mathbf{z}$
Dilithium2	$\mathbf{y}$ elems	17.30 (100%)	13.93 (80%)
	`KeccakF`	95.16 (100%)	81.65 (86%)
	`NTT`	21.63 (100%)	18.25 (84%)
	$\mathtt{NTT}^{-1}$	51.90 (100%)	51.90 (100%)
Dilithium3	$\mathbf{y}$ elems	25.49 (100%)	21.38 (84%)
	`KeccakF`	158.05 (100%)	137.73 (87%)
	`NTT`	30.59 (100%)	26.52 (87%)
	$\mathtt{NTT}^{-1}$	86.67 (100%)	87.33 (101%)
Dilithium5	$\mathbf{y}$ elems	27.18 (100%)	22.99 (85%)
	`KeccakF`	182.47 (100%)	161.71 (89%)
	`NTT`	31.06 (100%)	26.89 (87%)
	$\mathtt{NTT}^{-1}$	89.29 (100%)	89.60 (100%)

8.5.2 Optimized implementation

We have implemented the altered Dilithium signature scheme in optimized implementations for x64 with AVX2, Cortex-M4, and Cortex-M3, and benchmarked their performance.

AVX2. For AVX2, we base our modified implementation on the round-3 code package from the CRYSTALS team [DKLL+18].[9] Because of the relative abundance of RAM on x64 platforms, we can easily cache all of the accumulated values in $\mathbf{w}$. That is, we keep in memory all the values $\mathbf{A}_{ij}\mathbf{y}_j$ for $i = 1, \ldots k$ and $j = 1, \ldots, \ell$.

In all implementations (i.e., including the baseline) we apply aggressive lane stuffing. I.e., we always generate four elements of $\mathbf{y}$ in parallel during the execution of ExpandMask, and if we do not need that many $\mathbf{y}$ elements at that point; we precompute elements for use in the next iteration of the rejection-sampling loop.

Cortex-M{4,3}. For the Cortex-M4 platform, we use the STM23F407 Discovery board, which is based on the STM32F407VG microcontroller; for Cortex-M3, we use the Arduino Due, which features an ATSAM3X8E microcontroller. We have ported the reference implementation to each platform, and then applied the optimizations described in [GKS21, Sec. 4].

In the context of post-quantum signature schemes, both of these boards have a relatively low amount of SRAM. This makes it impossible to cache all components of $\mathbf{w}$, for which we would need another $k \times \ell$ KiB of SRAM. Instead, on the Cortex-M platforms, we cache only the value

$$\mathbf{w}' = \mathbf{A}(0, \mathbf{y}_2, \mathbf{y}_3, \ldots).$$

Storing this extra $\mathbf{w}'$-vector only needs an extra k KiB of SRAM space.

Benchmarking setup. We benchmark the AVX2 implementation of Dilithium using the benchmarking tool provided in the NIST submission code package. For the AVX2 implementation, 100 000 iterations were run on an Intel Core i7-4770 (Haswell) processor and its average recorded. On the x64 processor, all measurements were

[9] Available for download at `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions`.

done with Turbo Boost disabled, all Hyper-Threading cores shut down, and with the CPU clocked at the maximum nominal frequency. The Arm Cortex-M4 and M3 implementations were benchmarked on an STM32F407VG and an ATSAM3X8E respectively. The STM32F407 chip was clocked at 24 MHz and the flash wait states were set to zero; the algorithm latencies were measured using the SysTick counter. The ATSAM3X8E was clocked at 16 MHz and its wait states were also set to zero; the measurements used the internal `CYCCNT` cycle counter. On Cortex-M4, the measurements were averaged over 10 000 samples; on Cortex-M3, the measurements were averaged over 1 000 samples.

Table 8.2: Average latencies of Dilithium signature generation on AVX2, Cortex-M4, and Cortex-M3. Cycle counts are listed in kilocycles and *include* the computation of $\mathbf{A}$. Percentages report the total speedup of applying both proposals compared to the baseline (vanilla) Dilithium. Note that these results cannot be compared with [GKS21], because the parameters of Dilithium have been updated for round 3 of the NIST competition (and so our baseline is an update of [GKS21, Strategy 2]).

		baseline	alt-$\mathbf{z}$	
Dilithium2	AVX2	367	345	(6%)
	Cortex-M4	4 458	4 168	(6%)
	Cortex-M3	7 591	7 275	(4%)
Dilithium3	AVX2	564	532	(6%)
	Cortex-M4	7 137	6 889	(3%)
	Cortex-M3	12 316	12 015	(2%)
Dilithium5	AVX2	691	661	(4%)
	Cortex-M4	9 447	9 079	(4%)
	Cortex-M3	–[a]	–[a]	

[a] Not enough SRAM available to store Dilithium5 state.

Results. The results of our improved version of Dilithium are listed in Table 8.2. We observe performance speedups ranging from 2% for Dilithium3 on Cortex-M3, up to 6% for multiple configurations.

It should be stressed that all of these benchmarks include the *setup* stage of Dilithium. This is the conventional method of measuring Dilithium's performance. That is, the measurements include the expansion of the matrix $\mathbf{A}$ and the initial NTTs of $\mathbf{s_1}$, $\mathbf{s_2}$

and $\mathbf{t_0}$. In cases where the generation of the matrix is relatively fast (e.g., when the platform has hardware-acceleration for SHAKE), the *setup stage* will be shorter, resulting in a greater relative speedup.

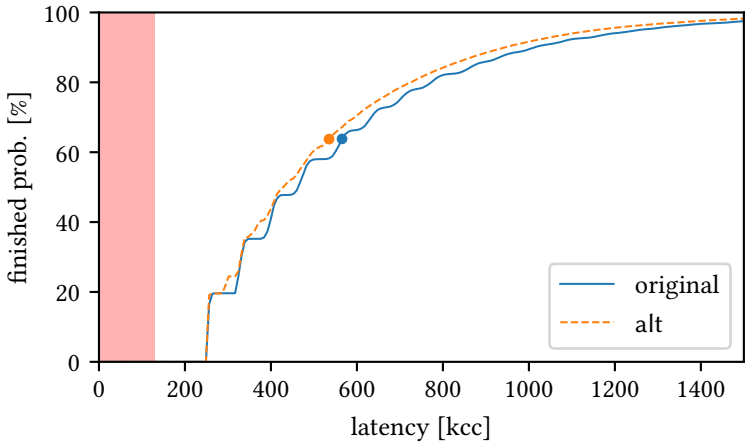


Figure 8.5: Probability of Dilithium3 signature generation on AVX2 to complete after a latency of x cycles. The *setup* stage is illustrated by the red box that runs from 0 to 130 kcc. The average latency is marked with a dot.

Moreover, it has been argued by [RGCB19] and [GKS21] that this setup stage often does not need to be computed during signature generation, but that it can be considered as part of the key generation instead. To provide you with an intuition, we have plotted the portion of Dilithium3 signature generations that finishes after x cycles in Figure 8.5. The figure shows that when the setup stage is precomputed, the relative speedup is 7% instead of 6%.

If we do not precompute the setup stage, the effect of an improved performance in the rejection-sampling loop is still better for the worse-case runs of the signature generation algorithm, because the latency of the setup stage is amortized. Indeed, if we look at the 90% percentile, the speedup of our improved algorithm is 13%; and at the 99% percentile, the speedup is 16%.

8.6 HAETAE

One of the submissions to the NIST competition for additional post-quantum digital signatures [NIST22b] is HAETAE [CCDG+23]. Like Dilithium, HAETAE is a Fiat–Shamir-with-aborts scheme. Perhaps our proposals could also be applied to that scheme.

However, after closer inspection we find that our proposals cannot be applied to HAETAE. Instead of the infinity norm, HAETAE uses the ℓ_2 norm (or Euclidean norm) in its rejection-sampling checks. In HAETAE, the ℓ_2 norm of a vector $\mathbf{v}$ is computed as $\|\mathbf{v}\|_2 = \sqrt{\mathbf{v}_{1,0} + \cdots + \mathbf{v}_{k,n-1}}$, which cannot easily be split into a number of subchecks. One has to look at *all* the elements of $\mathbf{v}$ before one can conclude that $\mathbf{v}$ would lead to an abort. However, after having inspected all the elements of $\mathbf{v}$, all the elements of $\mathbf{v}$ will have been significantly biased with the result of the check. This bias will remain when reusing the values in the next iterations, which will result in non-uniform $\lfloor \mathbf{z} \rceil$ output values. We see no way to overcome this as long as the norms in HAETAE are ℓ_2.

8.7 Conclusion

We propose an optimization to the Dilithium signature scheme, where we reuse parts of the nonce vector $\mathbf{y}$ when an abort occurs on one of the rejection-sampling checks. Our security analysis shows that the there is no additional security loss. In turn, the modifications lead to a slight speedup for the Dilithium signature generation algorithm, which ranges from 2% on Cortex-M3 for Dilithium3 to 6% for multiple configurations.

Still, our security analysis depends on the Dilithium matrix $\mathbf{A}$ containing an invertible polynomial in its first column.[10] Even though we can mitigate this reliance using extra logic, that logic would lead to very error-prone bookkeeping in the signature generation algorithm, which is not ideal. Therefore, we leave it to the community to decide which use cases allow for the implementation of these optimizations. In any case, we encourage additional scrutiny of our proposals, as well as analysis on

[10] One could consider modifying the key generation to only output keys with $\mathbf{A}$ matrices that have at least one invertible element in the first column. However, this changes the distribution of $\mathbf{A}$, which might be in violation with the MLWE assumption.

better bounds of the min-entropy of $\mathbf{w_1}$, and the prevalence of insecure $\mathbf{A}$ matrices in Dilithium.

Before our contribution, the iterations of the rejection-sampling loop of any Fiat–Shamir with aborts scheme have always been completely independent. As far as we know, our work is the first attempt at reusing values across iterations of the FSwA rejection-sampling loop; and even though this chapter focuses entirely on Dilithium, we would like to clarify that these kinds of nonce-reusal optimizations could be applied elsewhere, if the rejection condition can be divided into subchecks. Larger lattice-based zero-knowledge proofs, or schemes that sample their nonces from Gaussian distributions in particular stand to benefit as sampling from those distributions is expensive (e.g. [LNS20; Lyu12]).

8.A Resampling only $\mathbf{y}_1$ after failed $\mathbf{r_0}$-check

In this section we propose a second possible optimization to the Dilithium scheme, where we only resample the first element of $\mathbf{y}$ any time the $\mathbf{r_0}$-check leads to a reject. We have not been able to completely show the security of this modification, however we have also not encountered any clear reasons why it should be insecure.

8.A.1 $\text{Sign}^{\text{alt-}\mathbf{r_0}}$

Heuristically, as the $\mathbf{r_0}$-check only looks at the lower bits and $\mathbf{A}$ (being uniform) mixes all components of $\mathbf{y}$, resampling just $\mathbf{y}_1$ should give a new independent chance for $\mathbf{r_0}$-check to pass. Thus, a second proposal, is to resample only $\mathbf{y}_1$ when the $\mathbf{r_0}$-check fails, and to perform the $\mathbf{r_0}$-check before the $\mathbf{z}$-check. Contrary to the other proposal, the order of the checks is important. If we were to perform the $\mathbf{z}$-check first, then it is likely that we will have picked a $\mathbf{y}$ whose tail has passed the $\mathbf{z}$-checks multiple times, with different challenges c. This will bias $\mathbf{y}$ to have smaller values in its tail.[11]

After swapping the checks and modifying Algorithm 8.2, such that only $\mathbf{y}_1$ is resampled when $\mathbf{r_0}$-check fails, we get Algorithm 8.3. We will call the new alteration of the algorithm alt-$\mathbf{r_0}$. It can be applied independent of the other modification

[11] Another way of looking at this is that we have to discard all vector elements that we checked during the $\mathbf{z}$-check. We can only reuse the polynomials that we have not looked at yet. Conversely, for the $\mathbf{r_0}$-check, if any of the polynomials in $\mathbf{r_0}$ exceeds the bounds, we have to regenerate at least one polynomial in $\mathbf{y}$.

Algorithm 8.3: Proposal 2: Reuse $\mathbf{y}_2, \ldots, \mathbf{y}_\ell$ after aborting on $\mathbf{r_0}$-check.

$\mathsf{Sign}^{\text{alt-}\mathbf{r_0}}(sk = (\mathbf{A}, \mathbf{t}, \mathbf{s_1}, \mathbf{s}_2), M)$

1: $\kappa := 0; \xi := \ell$
2: `sign`: **loop**
3: **for** i **from** 1 **up to** ξ **do** ▷ Only (re)sample the first ξ elements of $\mathbf{y}$
4: $\mathbf{y}_i := \mathsf{ExpandMask}(\kappa); \kappa := \kappa + 1$
5: $\mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay}, 2\gamma_2)$
6: $c \in B_\tau := \mathsf{H}(M \| \mathbf{w_1})$
7: $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
8: **if** $\|\mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2}, \gamma_2)\|_\infty \geq \gamma_2 - \beta$ **then** ▷ $\mathbf{r_0}$-check
9: $\xi := 1$
10: **continue** `sign`
11: **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ $\mathbf{z}$-check
12: $\xi := \ell$
13: **continue** `sign`
14: **return** $(c, \mathbf{z})$

described in Section 8.3; or it can be combined, compounding the speed improvements of both proposals.

8.A.2 Security

The security reduction, from $\mathsf{Sign}^{\text{alt-}\mathbf{r_0}}$ to $\mathsf{Trans}^{\text{alt-}\mathbf{r_0}}$ is the same as in Section 8.4.2. It leads to the transcript generator listed in Figure 8.6. Unfortunately, we see no way to perfectly simulate the transcripts generated by $\mathsf{Trans}^{\text{alt-}\mathbf{r_0}}$, because the order of the checks is swapped in alt-$\mathbf{r_0}$.

In vanilla Dilithium, reordering the checks is allowed because the action that is taken is the same regardless of the check (i.e., all elements of $\mathbf{y}$ are resampled). This leads to $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$ being completely uniform after the $\mathbf{z}$-check, and as such it can be generated as $\mathbf{z} \xleftarrow{\$} S^\ell_{\gamma_1 - \beta}$ in the simulator.

However, in $\mathsf{Trans}^{\text{alt-}\mathbf{r_0}}$, the actions are different, as in the case of the $\mathbf{r_0}$-check ξ is set to 1, while in the case of $\mathbf{z}$-check ξ is set to ℓ. So swapping the checks changes the distribution of $\mathbf{z}$, and simulating it as $\mathbf{z} \xleftarrow{\$} S^\ell_{\gamma_1 - \beta}$ becomes invalid.

We hypothesize that we can use the hybrid Hyb^k_3, listed in Figure 8.7, to step towards a version of the scheme where a reject of the $\mathbf{r_0}$-check leads to all elements

$\underline{\mathsf{Trans}^{\text{alt-}\mathbf{r_0}}(M)}$:

1: $\mathbf{y} := \perp$
2: $\xi := \ell$
3: done $:=$ false
4: **repeat**
5: $\quad \mathbf{y}_{1\ldots\xi} \leftarrow Y$
6: $\quad \mathbf{w_1} := \mathsf{HighBits}(\mathbf{Ay})$
7: $\quad c \leftarrow C$
8: $\quad \mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
9: $\quad \mathbf{r_0} := \mathsf{LowBits}(\mathbf{Ay} - c\mathbf{s_2})$
10: $\quad$ **if** $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then**
11: $\quad\quad \xi = 1$
12: $\quad\quad$ **continue**
13: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then**
14: $\quad\quad \xi = \ell$
15: $\quad\quad$ **continue**
16: $\quad$ done $:=$ true
17: **until not** done
18: $\mathsf{H}(\mathbf{w_1}, M) := c$
19: **return** $(\mathbf{w_1}, \mathbf{z})$

Figure 8.6: Transcript generator for the alt-$\mathbf{r_0}$-modified scheme. Lines 5 to 6 describe $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}$ and Lines 8 to 16 describe $\mathsf{Resp}^{\text{alt-}\mathbf{r_0}}$.

$\mathsf{Hyb}_3^k(M)$:

1: $\mathbf{y} := \bot$
2: $\xi := \ell$
3: $i := 0$
4: **repeat**
5: $\quad(\mathbf{w_1}, \mathbf{y}) \leftarrow \mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \mathbf{y}, \xi)$
6: $\quad c \leftarrow C$
7: $\quad$**if** $i \geq k$ **then**
8: $\quad\quad(\mathbf{z}, \xi) \leftarrow \mathsf{Resp}^{\text{alt-}\mathbf{r_0}}(\mathbf{w_1}, c, \mathbf{y})$
9: $\quad$**else**
10: $\quad\quad\mathbf{z} \leftarrow \mathsf{Resp}(\mathbf{w_1}, c, \mathbf{y})$
11: $\quad i := i + 1$
12: **until** $\mathbf{z} \neq \bot$
13: $\mathsf{H}(\mathbf{w_1}, M) := c$
14: **return** $(\mathbf{w_1}, \mathbf{z})$

Figure 8.7: Hybrid signing oracle which, in the first k iterations, always sets ξ to ℓ when the $\mathbf{z}$-check leads to an abort; and which in all subsequent iterations sets ξ to the index of the element in $\mathbf{z}$ that lead to an abort during the $\mathbf{z}$-check. When increasing k, Hyb_3^k is gradually transformed from $\mathsf{Hyb}_3^0 = \mathsf{Trans}^{\text{alt-}\mathbf{r_0}}$ to $\mathsf{Hyb}_3^\infty = \mathsf{Trans}$.

of $\mathbf{y}$ being resampled. Then, as both checks will lead to $\xi := \ell$, we can swap the checks back to their original order, and the regular Dilithium simulator will apply (Figure 8.4). In the rest of this section follows a heuristic argument that quantifies the security loss of the hybrid step.

We replace $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}$ by Com and $\mathsf{Resp}^{\text{alt-}\mathbf{r_0}}$ by Resp respectively. Com always regenerates all elements of $\mathbf{y}$, while $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}$ takes ξ as an argument. I.e., Com(sk) is equal to $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \bot, \xi := \ell)$. $\mathsf{Resp}^{\text{alt-}\mathbf{r_0}}$ does differ from Resp: In $\mathsf{Resp}^{\text{alt-}\mathbf{r_0}}$, if the $\mathbf{r_0}$-check fails, then it will return $\xi := 1$. Resp is the same as in vanilla Dilithium, i.e., if the $\mathbf{r_0}$-check fails, then it will return $\xi := \ell$ instead of $\xi := 1$. The behavior around the $\mathbf{z}$-check remains the same, i.e., if the $\mathbf{z}$-check leads to an abort the function returns $\xi := \ell$.

The execution of Hyb_3^k and Hyb_3^{k+1} is equal when any iteration up to iteration k leads to a success. Only in the next iteration $k + 1$ does the change of ξ impact the execution, because it may lead to a different value of $\mathbf{y}$. This difference may propagate to any of the subsequent iterations, leading to a change of output when the

algorithm finally succeeds. As such, the distance of the outputs of the two hybrids will never be greater than the distance between the different $\mathbf{y}$s during iteration $k+1$. In other words, we need to bound the distance between $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \mathbf{y}, \xi = 1)$ and $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \mathbf{y}, \xi = \ell)$.

For convenience, define $\mathbf{v}_{\mathsf{tail}} = (\mathbf{v}_2, \ldots, \mathbf{v}_\ell)$ for any kind of $\mathbf{v}$, Y the output distribution of $\mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \mathbf{y}, \ell)$, and Y' the output distribution of $\mathsf{Com}(\mathsf{sk}) = \mathsf{Com}^{\text{alt-}\mathbf{r_0}}(\mathsf{sk}, \mathbf{y}, 1)$. We aim to find a bound for the statistical distance

$$\Delta(Y; Y') = \Delta(Y_{\mathsf{tail}}; Y'_{\mathsf{tail}}) = \frac{1}{2} \sum_{\mathbf{y}_{\mathsf{tail}} \in \tilde{S}^{\ell-1}_{\gamma_1}} \left| \Pr[Y_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}}] - \Pr[Y'_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}}] \right|.$$

Y_{tail} is equal to uniformly sampling from $\tilde{S}^{\ell-1}_{\gamma_1}$, i.e., $\Pr[Y_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}}] = 1/\#\tilde{S}^{\ell-1}_{\gamma_1}$. On the other hand, Y'_{tail} only contains remaining $\mathbf{y}$ tails from the previous iteration. If $\xi = 1$, then the previous iteration will have aborted because of a rejecting $\mathbf{r_0}$-check. In that case, we struck a $\mathbf{y}_1$ value that when joined with $\mathbf{y}_{\mathsf{tail}}$ (together with some $\mathbf{A}$, c and $\mathbf{s_2}$) led to a rejecting $\mathbf{r_0}$. I.e.,

$$\begin{aligned}
\Pr[Y'_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}}] &= \sum_{\mathbf{y}_1 \in \tilde{S}_{\gamma_1}} \Pr\left[Y_1 = \mathbf{y}_1 \textbf{ and } Y_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}} \;\middle|\; \|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta\right] \\
&= \frac{1}{\#\tilde{S}^{\ell}_{\gamma_1}} \sum_{\mathbf{y}_1 \in \tilde{S}_{\gamma_1}} \frac{\Pr\left[\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta \mid Y_1 = \mathbf{y}_1 \textbf{ and } Y_{\mathsf{tail}} = \mathbf{y}_{\mathsf{tail}}\right]}{\Pr\left[\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta\right]}.
\end{aligned}$$

In the literature, it has been heuristically assumed that the low order bits of $\mathbf{r}$ are uniformly distributed modulo $2\gamma_2$. If that is the case, the $\mathbf{r_0}$-check rejection probability is indeed independent of anything else, and the ratio inside the sum is equal to 1. However, absurd examples like when $\mathbf{A} = \mathbf{0}$ indicate that this cannot be the case. The rejection probability depends directly on $\mathbf{A}$. Fortunately, we can determine the range of this probability for "reasonable" values of $\mathbf{A}$. We will return to what "reasonable" values of $\mathbf{A}$ could look like.

Consider the value $\mathbf{r}_{\mathsf{proto}} = \mathbf{A}_{\mathsf{tail}}\mathbf{y}_{\mathsf{tail}} - c\mathbf{s_2}$, which corresponds to the case that $\mathbf{y}_1 = 0$. $\left\|\mathsf{LowBits}(\mathbf{r}_{\mathsf{proto}})\right\|_\infty$ might exceed $\gamma_2 - \beta$, or not; we do not know. However, we can look at every other value for $\mathbf{y}_1$ and complete the expression. If $\left\|\mathsf{LowBits}(\mathbf{r}_{\mathsf{proto}} + \mathbf{A}_1\mathbf{y}_1)\right\|_\infty$ exceeds $\gamma_2 - \beta$, then *any* of the coefficients in $\mathbf{r}_{\mathsf{proto}} + \mathbf{A}_1\mathbf{y}_1 \bmod 2\gamma_2$ must be in the

range $[\gamma_2 - \beta, \gamma_2 + \beta]$. This can be arranged when *any* of the $k \cdot n$ coefficients in $\mathbf{A}_1$ is set accordingly. This probability is equal to

$$p' = \left(1 - \frac{\beta + 1}{\gamma_2}\right)^{n \cdot k}, \tag{8.6}$$

which is, unsurprisingly, equal to the heuristically computed probability that a random $\mathbf{r_0}$ is rejected [DKLL+20, Equation 5].

However, recall that we are computing the probability that a randomly generated matrix $\mathbf{A}$ satisfies the property that $\mathbf{r_0}$ leads to a reject for some $\mathbf{r}_{\text{proto}}$ and $\mathbf{y}_1$. For a number of m rejected $\mathbf{r_0}$, generated from a number of $\#\tilde{S}_{\gamma_1} - 1$ (i.e., excluding $\mathbf{y}_1 = 0$) different possible $\mathbf{y}_1$, the probability that $\mathbf{A}$ satisfies that number of rejects follows the binomial distribution $m = B(\#\tilde{S}_{\gamma_1} - 1, p')$. One could see this distribution as (a bound of) the likeliness of $\mathbf{A}$ given if m out of $\#\tilde{S}_{\gamma_1} - 1$ of the possible $\mathbf{y}_1$ would lead to a reject. We look at the confidence interval where at most $\{2^{-128}, 2^{-192}, 2^{-256}\}$ of the distribution's area falls outside the interval for Dilithium{2,3,5} respectively. This gives us a lower and upper bound for the total number of $\mathbf{y}_1$ values for "reasonable" values of $\mathbf{A}$ that lead to a $\mathbf{r_0}$-check reject, which we will call m_{lo} and m_{hi}. We plug this into the expression for $\Pr[Y'_{\text{tail}} = \mathbf{y}_{\text{tail}}]$ and obtain

$$\begin{aligned}
\frac{1}{\#\tilde{S}^{\ell}_{\gamma_1}} \sum_{\mathbf{y}_1 \in \tilde{S}_{\gamma_1}} \frac{m_{\text{lo}}}{m_{\text{hi}}} &\leq \Pr[Y'_{\text{tail}} = \mathbf{y}_{\text{tail}}] \leq \frac{1}{\#\tilde{S}^{\ell}_{\gamma_1}} \sum_{\mathbf{y}_1 \in \tilde{S}_{\gamma_1}} \frac{m_{\text{hi}}}{m_{\text{lo}}} \\
\frac{1}{\#\tilde{S}^{\ell-1}_{\gamma_1}} \cdot \frac{m_{\text{lo}}}{m_{\text{hi}}} &\leq \Pr[Y'_{\text{tail}} = \mathbf{y}_{\text{tail}}] \leq \frac{1}{\#\tilde{S}^{\ell-1}_{\gamma_1}} \cdot \frac{m_{\text{hi}}}{m_{\text{lo}}}.
\end{aligned}$$

This leads to

$$\Delta(Y; Y') \leq \sum_{\mathbf{y}_{\text{tail}} \in \tilde{S}^{\ell-1}_{\gamma_1}} \frac{1}{\#\tilde{S}^{\ell-1}_{\gamma_1}} \left(1 - \frac{m_{\text{lo}}}{m_{\text{hi}}}\right) = 1 - \frac{m_{\text{lo}}}{m_{\text{hi}}},$$

which, for Dilithium{2,3,5}, is approximately $\{2^{-2289}, 2^{-2544}, 2^{-2543}\}$.

In wrapping up this hybrid step, we multiply this distance with the probability that we reach iteration $k + 1$, which is p^k. Summing over all signing-oracle queries and taking the limit of κ to infinity results in

$$\Delta^{\mathsf{Trans}^{\text{alt-}\mathbf{r_0}} \to \mathsf{Trans}} \leq \lim_{\kappa \to \infty} \left(\sum_{i=0}^{q_S - 1} \left(p^{\kappa} + \sum_{j=0}^{\kappa - 1} p^j \Delta(Y; Y') \right) \right) = q_S \frac{\Delta(Y; Y')}{1 - p}. \tag{8.7}$$

What are "reasonable" values of $\mathbf{A}$? The security argument of that we just described is based on the idea that $\mathbf{A}$ should have a "reasonable" value. This means that we assume that, for all possible $\mathbf{y}_{\mathsf{tail}}$, the value of $\mathbf{A}$ is such that the probability that the $\mathbf{r_0}$-check leads to an abort is close to the probability that the $\mathbf{r_0}$-check leads to an abort if all of $\mathbf{y}$ was freshly sampled. Examples of *bad* values of $\mathbf{A}$ are when a polynomial in the first column of $\mathbf{A}$ is equal to 0, or when many of the coefficients in those polynomials $\mathbf{A}$ are close to a small factor of $2\gamma_2$. Ideally, we would like to fix the (reasonable) value of $\mathbf{A}$ ahead of time (i.e., during KeyGen) and then show that this value leads to a good rejection probability for all $\mathbf{y}_{\mathsf{tail}}$. However, we were not able to find a good description of the class of "reasonable" values of $\mathbf{A}$, and were therefore forced to rely on a more ad-hoc argument—expressing $\mathbf{A}$ in terms of $\mathbf{y}_{\mathsf{tail}}$ instead of the other way around. Still, because the probability space of $\mathbf{y}_1$ is so incredibly big (with more than 4000 bits of entropy for all variants), this results in a very small (even in cryptographic terms) statistical distance ($\leq 2^{-2289}$). As such, we expect that the security loss of this hybrid step is minimal.

8.A.3 Performance

The proposal described in this appendix adds another performance improvement to the Dilithium scheme; this one more substantial than the first. The combination of both alterations, which we denote by alt, results in the largest improvement in signing speed.

In Table 8.3, we list the updated number of primitive operations, and in Table 8.4 we list the updated speedups for optimized implementations. In Table 8.4, we observe new speedups ranging from 15% for Dilithium2 on Cortex-M3, up to 23% for Dilithium3 with AVX2, when using the combination of both alterations to the scheme.

Table 8.3: Extension of Table 8.1: Average number of sampled $\mathbf{y}$ elements, calls to `KeccakF1600_StatePermute`, `NTT`, and $\texttt{NTT}^{-1}$ in the Dilithium rejection-sampling loop; using unmodified Dilithium signing, and using the modifications proposed in this chapter. Percentages indicate the relative number of calls for each operation (lower is better). Averages were computed over 100 000 runs.

		baseline	alt-$\mathbf{z}$	alt-$\mathbf{r_0}$	alt
Dilithium2	$\mathbf{y}$ elems	17.30 (100%)	13.93 (80%)	9.87 (57%)	8.42 (49%)
	KeccakF	95.16 (100%)	81.65 (86%)	65.33 (69%)	59.47 (63%)
	NTT	21.63 (100%)	18.25 (84%)	14.18 (66%)	12.72 (59%)
	$\texttt{NTT}^{-1}$	51.90 (100%)	51.90 (100%)	51.67 (100%)	51.61 (99%)
Dilithium3	$\mathbf{y}$ elems	25.49 (100%)	21.38 (84%)	11.59 (45%)	10.24 (40%)
	KeccakF	158.05 (100%)	137.73 (87%)	88.72 (56%)	81.91 (52%)
	NTT	30.59 (100%)	26.52 (87%)	16.72 (55%)	15.36 (50%)
	$\texttt{NTT}^{-1}$	86.67 (100%)	87.33 (101%)	87.13 (101%)	87.06 (100%)
Dilithium5	$\mathbf{y}$ elems	27.18 (100%)	22.99 (85%)	12.95 (48%)	11.32 (42%)
	KeccakF	182.47 (100%)	161.71 (89%)	111.37 (61%)	103.37 (57%)
	NTT	31.06 (100%)	26.89 (87%)	16.84 (54%)	15.22 (49%)
	$\texttt{NTT}^{-1}$	89.29 (100%)	89.60 (100%)	89.32 (100%)	89.68 (100%)

Table 8.4: Extension of Table 8.2. Average latencies of Dilithium signature generation on AVX2, Cortex-M4, and Cortex-M3. Cycle counts are listed in kilocycles and *include* the computation of $\mathbf{A}$. Percentages report the total speedup of applying both proposals compared to the baseline (vanilla) Dilithium. Note that these results cannot be compared with [GKS21], because the parameters of Dilithium have been updated for round 3 of the NIST competition (and so our baseline is an update of [GKS21, Strategy 2]).

		baseline	alt-$\mathbf{z}$	alt-$\mathbf{r_0}$	alt	
Dilithium2	AVX2	367	345	318	309	(16%)
	Cortex-M4	4 458	4 168	3 810	3 698	(17%)
	Cortex-M3	7 591	7 275	6 595	6 472	(15%)
Dilithium3	AVX2	564	532	445	434	(23%)
	Cortex-M4	7 137	6 889	5 873	5 736	(20%)
	Cortex-M3	12 316	12 015	10 177	10 016	(19%)
Dilithium5	AVX2	691	661	561	549	(20%)
	Cortex-M4	9 447	9 079	7 899	7 766	(18%)
	Cortex-M3	–[a]	–[a]	–[a]	–[a]	

[a] Not enough SRAM available to store Dilithium5 state.

8.B Derivation of Equation (8.1)

$$
\begin{aligned}
\Delta_{p,\epsilon_0,\epsilon}^{\mathsf{Sign}^{\mathsf{alt}}\to\mathsf{Prog}^{\mathsf{alt}}} &= \lim_{\kappa\to\infty}\left(\sum_{i=0}^{q_S-1}\left(p^{\kappa}+\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\sum_{j=0}^{\kappa-1}\delta_{i,j,\epsilon}\right)\right)\\
&= \lim_{\kappa\to\infty}\left(q_S p^{\kappa}+\sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\sum_{j=0}^{\kappa-1}\delta_{i,j,\epsilon}\right)\right)\\
&= \sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\sum_{j=0}^{\infty}\delta_{i,j,\epsilon}\right)\\
&= \sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\sum_{j=0}^{\infty}\left(p^j\epsilon\left(\frac{i}{1-p}+q_H+j\right)\right)\right)\\
&= \sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\sum_{j=0}^{\infty}\epsilon\left(\frac{i}{1-p}+q_H\right)p^j+\sum_{j=0}^{\infty}\epsilon j p^j\right)\\
&= \sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\epsilon\frac{\left(\frac{i}{1-p}+q_H\right)}{1-p}+\epsilon\frac{p}{(1-p)^2}\right)\\
&= \sum_{i=0}^{q_S-1}\left(\delta_{i,0,\epsilon_0}-\delta_{i,0,\epsilon}+\epsilon\frac{q_H}{1-p}+\epsilon\frac{i+p}{(1-p)^2}\right)\\
&= \sum_{i=0}^{q_S-1}\left(p^0(\epsilon_0-\epsilon)\left(\frac{i}{1-p}+q_H+0\right)+\epsilon\frac{q_H}{1-p}+\epsilon\frac{i+p}{(1-p)^2}\right)\\
&= \sum_{i=0}^{q_S-1}\left((\epsilon_0-\epsilon)\left(\frac{i}{1-p}+q_H\right)+\epsilon\frac{q_H}{1-p}+\epsilon\frac{i+p}{(1-p)^2}\right)\\
&= q_S\left((\epsilon_0-\epsilon)\left(\frac{\frac{q_S-1}{2}}{1-p}+q_H\right)+\epsilon\frac{q_H}{1-p}+\epsilon\frac{\frac{q_S-1}{2}+p}{(1-p)^2}\right)\\
&= q_S(\epsilon_0-\epsilon)\left(\frac{q_S-1}{2(1-p)}+q_H\right)+q_S\epsilon\frac{q_H}{1-p}+q_S\epsilon\frac{q_S+2p-1}{2(1-p)^2}\\
&\le q_S(\epsilon_0-\epsilon)\left(\frac{q_S-1}{2(1-p)}+q_H\right)+q_S\epsilon\frac{q_H}{1-p}+q_S\epsilon\frac{q_S+1}{2(1-p)^2}
\end{aligned}
$$

9 Conclusion

Speed. In this thesis, we explored methods for polynomial multiplication that are faster than the original method proposed for Dilithium. The first implementations of Dilithium [GKOS18; RGCB19]—including ours ([GKS21])—followed the strategy of the Dilithium team, using Cooley–Tukey butterflies for the forward NTT and Gentleman–Sande butterflies for the inverse NTT. We found that we could improve the speed of the inverse NTT, by using Cooley–Tukey instead of Gentleman–Sande butterflies, and adopting the technique mentioned in [ACCH+22, Appendix D].

From the start, Dilithium was designed to use polynomials modulo $q = 8380417$, to enable very fast polynomial multiplications using the NTT. However, we found that—for some polynomial multiplications in the signing algorithm—we are not bound to the original Dilithium q. In Chapter 4, we explored the idea of using a multi-moduli NTT to compute the 32-bit polynomial multiplications in smaller 16-bit chunks. Our experiments did not show a significant performance increase in the general case, i.e., when using this method to optimize (unbounded) polynomial multiplications in R_q. However, in Chapter 5, we found that—because of the tighter bounds—$c\mathbf{s_1}$ and $c\mathbf{s_2}$ could be computed modulo $q' \in \{257, 769\}$, which led to polynomial multiplications that are 38% faster for $q' = 257$ and 33% faster for $q' = 769$, compared to $q = 8380417$. Moreover, others have found speedups when applying the multi-moduli idea to the computation of $c\mathbf{t_0}$ (using $q'_0 = 769$ and $q'_1 = 3329$) on Cortex-M3 [HAZD+24].

One recurring theme throughout this thesis is that we see a large part of Dilithium's computation time is spent in SHAKE. Even with the recent work of [HAZD+24], SHAKE still takes up 59%–84% of Dilithium's computation time, depending on the algorithm and parameter set. As such, platforms with hardware acceleration for SHAKE have a clear advantage over platforms that don't have any hardware acceleration for SHAKE.

Recent improvements in speed. Recently it was found that 16-bit NTTs could be improved by using Plantard reduction [Pla21] (instead of Montgomery reduction) for the internal twiddle-factor multiplications [HZZL+22]. This improvement is also applicable to Dilithium and has been applied in [HAZD+24]. [HAZD+24] also improved the SHAKE implementation from the eXtended Keccak Code Package (XKCP),[1] and with their improvements, they currently[2] hold the speed records for Dilithium on Cortex-M3 and Cortex-M4.

Memory usage. In Chapter 6, we built a Dilithium implementation that was optimized for memory usage rather than for speed, as many devices only have a very limited amount of SRAM. In this implementation, we were able to use the 16-bit NTTs to reduce the memory usage of the $c\mathbf{s_1}$ and $c\mathbf{s_2}$ polynomial multiplications from 2 KiB bytes to 1 KiB. We found that, in the signing algorithm, the generation of both the matrix $\mathbf{A}$ as well as the vector $\mathbf{y}$ can be streamed for a slowdown factor of about 3.3–3.9.

Before our work, it was not clear whether Dilithium was a scheme that could reasonably fit into 16 KiB of memory. In our memory-optimized implementation, we were able to fit Dilithium2 and Dilithium3 in 8 KiB of SRAM, with Dilithium5 slightly above at 8.1 KiB. We even managed to reduce the memory usage of signature verification to only 3 KiB of memory. These figures show that Dilithium is a practical scheme for use on memory-constrained devices. In [NIST22a, Section 2.2.2], NIST recognized our findings and used them in their decision to standardize Dilithium.

Deployment. In Chapter 7, we added support for verification with Dilithium in the hardware security engine (HSE) of the S32G274A vehicle network processor. The S32G274A HSE does not have hardware acceleration for SHAKE. This would not be a problem for Dilithium verification were it not for the hashing of the boot image. For a boot image of 128 KiB, most of the verification is spent compressing the message, leading to a relatively slow image verification. The S32G274A provides a mechanism to overcome this, by verifying the Dilithium signature ("initial proof of authenticity") when the image is installed, and constructing an optimized "reference proof of authenticity" for use during boot. However, not all chips provide such a

[1] `https://github.com/XKCP/XKCP`
[2] As of February 2024.

mechanism. In that case, the only alternative is to pre-hash the image using a hash for which hardware acceleration *is* present on the chip (e.g., SHA2), and then verify the signature over the hash instead. When using pre-hashed Dilithium variants, implementations must domain-separate the pre-hashed variant from other Dilithium variants, and bind the signer's public key to the signed image. There exists a risk that these extra measures will be forgotten or improperly implemented, which could lead to weaknesses in protocols. Hence, we should not overlook that the proper fix is to add hardware acceleration for SHAKE to the chip. Unfortunately, given the lifetime of many common chip families, it will realistically take years and maybe even decades until we can expect SHAKE acceleration to be as mainstream as acceleration for SHA2.

Outlook. The NIST competition has attracted the attention of many researchers to the evaluation criteria that NIST stipulated. As such, there has been a lot of research into the implementation of fast and small post-quantum implementations, both in software as well as in hardware. There have also been plenty of projects dedicated to the analysis of side-channels and fault-tolerance in Dilithium (e.g., [ABCH+23; BVCM+23; CGTZ23; CKAM+21; EFGT16; FDK20; HLKL+21; IMSS+22; Jen24; KAA21; MGTF19; MUTS22; RCDB23; RJHC+18][3]), leading to good understanding of the weak spots in the algorithm. Even so, there still only a handful of public side-channel and fault protected implementations of Dilithium. I believe it would benefit the community to focus on constructing more protected implementations, rather than exploiting weaknesses in unprotected implementations.

Aside from side-channel protected implementations, there is more work to do evaluating the candidates from the NIST "Additional Signatures" standardization process [NIST23a]. Although the evaluation of the new NIST schemes is important, I hope that we will also find time to increase our understanding of other post-quantum authentication schemes that are more advanced than regular digital signatures, like zero-knowledge proofs [LNP22], designated verifier signatures [BFGJ+22; HKKP22; JSI96], and (linkable) ring signatures [BKP20; LW05; RST01].

[3] Alongside these attacks, there is also work that attacks the NTT in other lattice schemes (like Kyber and NTRU). Many other attacks are also applicable to the usage of the NTT in Dilithium.

Bibliography

[AABN02] Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. "From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security." In: *Advances in Cryptology – EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Springer, Apr. 2002, pp. 418–433. DOI: 10.1007/3-540-46035-7_28 (cit. on p. 38).

[AB74] Ramesh C. Agarwal and C. Sidney Burrus. "Fast convolution using Fermat number transforms with applications to digital filtering." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22.2 (1974), pp. 87–97. ISSN: 0096-3518. DOI: 10.1109/TASSP.1974.1162555 (cit. on p. 79).

[AB75] Ramesh C. Agarwal and C. Sidney Burrus. "Number theoretic transforms to implement fast digital convolution." In: *Proceedings of the IEEE* 63.4 (1975), pp. 550–560. ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9791 (cit. on p. 79).

[ABBK+16] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. "An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation." In: *AFRICACRYPT 16: 8th International Conference on Cryptology in Africa*. Ed. by David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi. Vol. 9646. Lecture Notes in Computer Science. Springer, Apr. 2016, pp. 44–60. DOI: 10.1007/978-3-319-31517-1_3 (cit. on p. 96).

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. "Cortex-M4 optimizations for {R,M}LWE schemes." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020), pp. 336–357. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.336-357. URL: https://tches.iacr.org/index.php/TCHES/article/view/8593/8160 (cit. on pp. 28, 55, 59, 72, 75, 76, 84).

[ABCH+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. "Protecting Dilithium against Leakage Revisited Sensitivity Analysis and Improved Implementations." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (2023), pp. 58–79. ISSN: 2569-2925. DOI: 10.46586/tches.v2023.i4.58-79 (cit. on pp. 31, 159).

[ABDK+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*. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIST16]. 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions (cit. on p. 4).

[ABDK+19] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS–Kyber*. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIST16]. 2019. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions (cit. on p. 66).

[ACCE+20] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. "Polynomial Multiplication in NTRU Prime: Comparison of Optimization Strategies on Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2020), pp. 217–238. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i1.217-238. URL: https://tches.iacr.org/index.php/TCHES/article/view/8733 (cit. on pp. 77–79).

[ACCH+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. "Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2022), pp. 127–151. ISSN: 2569-2925. DOI: 10.46586/tches.v2022.i1.127-151. URL: https://tches.iacr.org/index.php/TCHES/article/view/9292 (cit. on pp. 55, 78–80, 83, 84, 157).

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange - A New Hope." In: *USENIX Security 2016: 25th USENIX*

Security Symposium. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, Aug. 2016, pp. 327–343 (cit. on p. 56).

[ADR02] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the Security of Joint Signature and Encryption." In: *Advances in Cryptology – EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Springer, Apr. 2002, pp. 83–107. DOI: 10.1007/3-540-46035-7_6 (cit. on p. 21).

[AELN+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. "ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020), pp. 219–242. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.219-242. URL: https://tches.iacr.org/index.php/TCHES/article/view/8589 (cit. on p. 59).

[AFGK+14] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. "GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias." In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, Dec. 2014, pp. 262–281. DOI: 10.1007/978-3-662-45611-8_14 (cit. on p. 125).

[AHHP+18] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. "Implementing RLWE-based Schemes Using an RSA Co-Processor." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019 (2018), pp. 169–208. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i1.169-208. URL: https://tches.iacr.org/index.php/TCHES/article/view/7338 (cit. on p. 97).

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. "Faster Kyber and Dilithium on the Cortex-M4." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, June 2022, pp. 853–871. DOI: 10.1007/978-3-031-09234-3_42. URL: https://eprint.iacr.org/2022/112 (cit. on pp. 6, 82, 84, 94, 97, 105–109, 200).

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. "A new hope on ARM Cortex-M." In: *Security, Privacy, and Advanced Cryptography Engineering: 6th International Conference, SPACE 2016*. Ed. by Claude Carlet, M. Anwar Hasan, and Vishal Saraswat. Vol. 10076. Lecture Notes in Computer Science. Springer,

2016, pp. 332–349. DOI: 10.1007/978-3-319-49445-6_19. URL: https://eprint.iacr.org/2016/758 (cit. on p. 59).

[Ajt96] Miklós Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)." In: *28th Annual ACM Symposium on Theory of Computing*. ACM Press, May 1996, pp. 99–108. DOI: 10.1145/237814.237838 (cit. on pp. 4, 34).

[ANSI15] *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. American National Standards Institute (ANSI), X9.62-1998. Nov. 2015 (cit. on p. 121).

[ANSSI22] *NSSI views on the Post-Quantum Cryptography transition*. 2022. URL: https://cyber.gouv.fr/en/publications/anssi-views-post-quantum-cryptography-transition (visited on Apr. 9, 2024) (cit. on p. 3).

[ANTT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. "LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage." In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 225–242. DOI: 10.1145/3372297.3417268. URL: https://eprint.iacr.org/2020/615 (cit. on p. 125).

[ARM10a] ARM. *Cortex-M3 Devices Generic User Guide*. 2010. URL: https://developer.arm.com/documentation/dui0552/latest (visited on May 15, 2023) (cit. on p. 27).

[ARM10b] ARM. *Cortex-M3 Technical Reference Manual r2p0*. 2010. URL: https://developer.arm.com/documentation/ddi0337/latest (visited on May 15, 2023) (cit. on p. 27).

[ARM11] ARM. *Cortex-M4 Devices Generic User Guide*. 2011. URL: https://developer.arm.com/documentation/dui0553/latest (visited on May 15, 2023) (cit. on p. 27).

[ARM20] ARM. *Cortex-M4 Technical Reference Manual r0p1*. 2020. URL: https://developer.arm.com/documentation/100166/0001 (visited on May 15, 2023) (cit. on p. 27).

[Atmel15] Atmel. *SAM3X / SAM3A Series Datasheet*. 2015. URL: https://www.microchip.com/en-us/product/ATsam3x8e (visited on May 23, 2023) (cit. on p. 30).

[AYS15] Aydin Aysu, Bilgiday Yuce, and Patrick Schaumont. "The Future of Real-Time Security: Latency-Optimized Lattice-Based Digital Signatures." In: *ACM Transactions on Embedded Computing Systems* 14.3 (2015). ISSN: 1539-9087. DOI: 10.1145/2724714 (cit. on p. 91).

[Bar87] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor." In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, Aug. 1987, pp. 311–323. DOI: 10.1007/3-540-47721-7_24 (cit. on p. 23).

[BBDD+23] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. "Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium." In: *Advances in Cryptology – CRYPTO 2023, Part V*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14085. Lecture Notes in Computer Science. Springer, Aug. 2023, pp. 358–389. DOI: 10.1007/978-3-031-38554-4_12 (cit. on pp. 9, 34, 37, 39, 40, 132–136, 138, 139).

[BCLv19] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. *NTRU Prime*. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIST16]. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. 2019 (cit. on p. 66).

[BCP10] Ben "bushing" Byer, Hector Martin "marcan" Cantero, and Sven Peter. *Console Hacking 2010*. 27th Chaos Communication Congress – 27C3, https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html. 2010 (cit. on p. 125).

[BCRR+22] Joppe W. Bos, Brian Carlson, Joost Renes, Marius Rotaru, Amber Sprenkels, and Geoffrey P. Waters. "Post-quantum secure boot on vehicle network processors." In: *20th escar Europe - The World's Leading Automotive Cyber Security Conference (15. - 16.11.2022)*. Ruhr-Universität Bochum, 2022, pp. 112–125. DOI: 10.13154/294-9372. URL: https://eprint.iacr.org/2022/635 (cit. on pp. 8, 199).

[BDFL+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. Lecture Notes in Computer Science. Springer, Dec. 2011, pp. 41–69. DOI: 10.1007/978-3-642-25385-0_3 (cit. on p. 34).

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*.

Ed. by Bo-Yin Yang. Springer, Nov. 2011, pp. 117–129. DOI: 10.1007/978-3-642-25405-5_8 (cit. on p. 39).

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *Advances in Cryptology – EUROCRYPT'97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, May 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4 (cit. on pp. 112, 119).

[BDLS+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures." In: *CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9. URL: https://link.myspringer.com/chapter/10.1007/978-3-642-23951-9_9 (cit. on pp. 115, 125).

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak." In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, May 2013, pp. 313–314. DOI: 10.1007/978-3-642-38348-9_19 (cit. on p. 4).

[Ber01] Daniel J. Bernstein. *Multidigit Multiplication For Mathematicians*. 2001. URL: https://cr.yp.to/papers/m3.pdf (cit. on p. 56).

[BFGJ+22] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. "Post-quantum Asynchronous Deniable Key Exchange and the Signal Handshake." In: *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe. Vol. 13178. Lecture Notes in Computer Science. Springer, Mar. 2022, pp. 3–34. DOI: 10.1007/978-3-030-97131-1_1 (cit. on p. 159).

[BG14] Shi Bai and Steven D. Galbraith. "An Improved Compression Technique for Signatures Based on Learning with Errors." In: *Topics in Cryptology – CT-RSA 2014*. Ed. by Josh Benaloh. Springer, 2014, pp. 28–47. DOI: 10.1007/978-3-319-04852-9_2. URL: https://eprint.iacr.org/2013/838 (cit. on p. 34).

[BH19] Joachim Breitner and Nadia Heninger. "Biased Nonce Sense: Lattice Attacks Against Weak ECDSA Signatures in Cryptocurrencies." In: *FC 2019: 23rd International Conference on Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Vol. 11598. Lecture Notes in Computer Science. Springer, Feb. 2019, pp. 3–20. DOI: 10.1007/978-3-030-32101-7_1 (cit. on p. 125).

[BHHL+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, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, Apr. 2015, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15 (cit. on p. 39).

[BHKN+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS+ Signature Framework." In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, Nov. 2019, pp. 2129–2146. DOI: 10.1145/3319535.3363229 (cit. on p. 3).

[BHKY+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. *Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1*. Cryptology ePrint Archive, Report 2021/986. https://ia.cr/2021/986. 2021 (cit. on pp. 78, 83).

[BKNS20] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. "Post-Quantum TLS on Embedded Systems: Integrating and Evaluating Kyber and SPHINCS+ with mbedTLS." In: *ASIACCS 20: 15th ACM Symposium on Information, Computer and Communications Security*. Ed. by Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese. ACM Press, Oct. 2020, pp. 841–852. DOI: 10.1145/3320269.3384725. URL: https://eprint.iacr.org/2020/308 (cit. on p. 3).

[BKP20] Ward Beullens, Shuichi Katsumata, and Federico Pintore. "Calamari and Falafl: Logarithmic (Linkable) Ring Signatures from Isogenies and Lattices." In: *Advances in Cryptology – ASIACRYPT 2020, Part II*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. Lecture Notes in Computer Science. Springer, Dec. 2020, pp. 464–492. DOI: 10.1007/978-3-030-64834-3_16 (cit. on p. 159).

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4." In: *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*. Ed. by Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi. Vol. 11627. Lecture Notes in Computer Science. Springer, July 2019, pp. 209–228. DOI: 10.1007/978-3-030-23696-0_11. URL: https://eprint.iacr.org/2019/489.pdf (cit. on pp. 28, 59).

[BKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. "Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (2020), pp. 222–244. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i2.222-244. URL: https://tches.iacr.org/index.php/TCHES/article/view/8550 (cit. on pp. 28, 59).

[BP02] Michael Backes and Birgit Pfitzmann. "Computational Probabilistic Noninterference." In: *ESORICS 2002: 7th European Symposium on Research in Computer Security*. Ed. by Dieter Gollmann, Günter Karjoth, and Michael Waidner. Vol. 2502. Lecture Notes in Computer Science. Springer, Oct. 2002, pp. 1–23. DOI: 10.1007/3-540-45853-0_1 (cit. on p. 30).

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." In: *ACM CCS 93: 1st Conference on Computer and Communications Security*. Ed. by Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby. ACM Press, Nov. 1993, pp. 62–73. DOI: 10.1145/168588.168596 (cit. on pp. 34, 38).

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. "Dilithium for Memory Constrained Devices." In: *AFRICACRYPT 2022: 13th*. Ed. by Lejla Batina and Joan Daemen. Vol. 2022. Lecture Notes in Computer Science. Springer, July 2022, pp. 217–235. DOI: 10.1007/978-3-031-17433-9_10. URL: https://eprint.iacr.org/2022/323 (cit. on pp. 7, 57, 199).

[BRV22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. "Post-Quantum Cryptography with Contemporary Co-Processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer." In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 3683–3697. URL: https://eprint.iacr.org/2020/1303 (cit. on pp. 97, 98, 109).

[BS07] Daniel J. Bernstein and Jonathan P. Sorenson. "Modular exponentiation via the explicit Chinese remainder theorem." In: *Mathematics of Computation* 76.257 (2007), pp. 443–454. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-06-01849-7, archived at https://web.archive.org/web/20221101000000*/https://cr.yp.to/antiforgery/meecrt-20030815.pdf on Nov. 1, 2022 (cit. on p. 66).

[BS97] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems." In: *Advances in Cryptology – CRYPTO'97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, Aug. 1997, pp. 513–525. DOI: 10.1007/BFb0052259 (cit. on pp. 112, 119).

[BSI20a] *Migration to Post Quantum Cryptography*. Recommendation. Federal Office for Information Security (BSI), 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.html (visited on Jan. 23, 2024) (cit. on p. 3).

[BSI20b] *Status of quantum computer development (version 1.2)*. Tech. rep. Federal Office for Information Security (BSI), 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/P283_QC_Studie-V_1_2.html (visited on Jan. 23, 2024) (cit. on p. 3).

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.4 (2019), pp. 17–61. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i4.17-61. URL: https://tches.iacr.org/index.php/TCHES/article/view/8344 (cit. on p. 59).

[BVCM+23] Alexandre Berzati, Andersson Calle Viera, Maya Chartouny, Steven Madec, Damien Vergnaud, and David Vigilant. "Exploiting Intermediate Value Leakage in Dilithium: A Template-Based Approach." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (2023), pp. 188–210. ISSN: 2569-2925. DOI: 10.46586/tches.v2023.i4.188-210 (cit. on p. 159).

[BvSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. ""Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way." In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, Sept. 2014, pp. 75–92. DOI: 10.1007/978-3-662-44709-3_5. URL: https://eprint.iacr.org/2014/161 (cit. on p. 125).

[CCDG+23] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Marc Möller, Damien Stehlé, and MinJune Yi. *HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures*. Cryptology ePrint Archive, Paper 2023/624. 2023. URL: https://eprint.iacr.org/2023/624 (cit. on p. 145).

[CG19] David Challener and Kenneth Goldman. *Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59*. 2019. URL: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/ (visited on Dec. 13, 2022) (cit. on p. 112).

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. "Improved Gadgets for the High-Order Masking of Dilithium." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (2023), pp. 110–145. ISSN: 2569-2925. DOI: 10.46586/tches.v2023.i4.110-145 (cit. on pp. 31, 159).

[CHKS+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. "NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021), pp. 159–188. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.159-188. URL: https://tches.iacr.org/index.php/TCHES/article/view/8791 (cit. on pp. 55, 77, 79, 81).

[CJLM+20] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Amber Sprenkels, and Benoît Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?" In: *CANS 20: 19th International Conference on Cryptology and Network Security*. Ed. by Stephan Krenn, Haya Shulman, and Serge Vaudenay. Vol. 12579. Lecture Notes in Computer Science. Springer, Dec. 2020, pp. 526–545. DOI: 10.1007/978-3-030-65411-5_26. URL: https://eprint.iacr.org/2020/836 (cit. on p. 200).

[CKAM+21] Zhaohui Chen, Emre Karabulut, Aydin Aysu, Yuan Ma, and Jiwu Jing. "An Efficient Non-Profiled Side-Channel Attack on the CRYSTALS-Dilithium Post-Quantum Signature." In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 583–590. DOI: 10.1109/ICCD53106.2021.00094 (cit. on p. 159).

[CNPR+23] Tung Chou, Ruben Niederhagen, Edoardo Persichetti, Tovohery Hajatiana Randrianarisoa, Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. "Take Your MEDS: Digital Signatures from Matrix Code Equivalence." In: *AFRICACRYPT 23: 14th International Conference on Cryptology in Africa*. Ed. by Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne. Vol. 14064. Lecture Notes in Computer Science. Springer, July 2023, pp. 28–52. DOI: 10.1007/978-3-031-37679-5_2. URL: https://eprint.iacr.org/2022/1559 (cit. on p. 125).

[Coo87] James W. Cooley. "The re-discovery of the fast Fourier transform algorithm." In: *Mikrochimica Acta* 93.1-6 (1987), pp. 33–45. ISSN: 0026-3672. DOI: 10.1007/BF01201681 (cit. on p. 55).

[CT65] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series." In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1965-0178586-1 (cit. on pp. 50, 52, 75).

[data24a] Amber Sprenkels. *Dilithium nonce recycling experiments and benchmarks*. 2024. DOI: 10.5281/zenodo.10708819. URL: https://doi.org/10.5281/zenodo.10708819 (cit. on p. 11).

[data24b] Amber Sprenkels. *Memory-optimized round-3 Dilithium in pure C*. 2024. DOI: 10.5281/zenodo.10708284. URL: https://doi.org/10.5281/zenodo.10708284 (cit. on p. 10).

[data24c] Amber Sprenkels. *Speed-optimized round-2 Dilithium on Cortex-M3 and Cortex-M4*. 2024. DOI: 10.5281/zenodo.10706370. URL: https://doi.org/10.5281/zenodo.10706370 (cit. on p. 10).

[data24d] Amber Sprenkels. *Speed-optimized round-3 Dilithium on Cortex-M4*. 2024. DOI: 10.5281/zenodo.10707141. URL: https://doi.org/10.5281/zenodo.10707141 (cit. on p. 10).

[DFPS23] Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. "A Detailed Analysis of Fiat-Shamir with Aborts." In: *Advances in Cryptology – CRYPTO 2023, Part V*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14085. Lecture Notes in Computer Science. Springer, Aug. 2023, pp. 327–357. DOI: 10.1007/978-3-031-38554-4_11. URL: https://eprint.iacr.org/2023/245 (cit. on pp. 34, 37, 132).

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography." In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638 (cit. on pp. 2, 15).

[DKLL+17] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium – Submission to round 1 of the NIST post-quantum project*. 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions (visited on Jan. 29, 2024) (cit. on p. 4).

[DKLL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018), pp. 238–268. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.238-268. URL: https://tches.iacr.org/index.php/TCHES/article/view/839 (cit. on pp. 3, 4, 87, 142).

[DKLL+19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium – Submission to round 2 of the NIST post-quantum project.* 2019. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions (visited on Sept. 25, 2023) (cit. on pp. 57, 61).

[DKLL+20] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-Dilithium – Submission to round 3 of the NIST post-quantum project.* 2020. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions (visited on Sept. 25, 2023) (cit. on pp. 33, 126, 151).

[Dur64] Richard Durstenfeld. "Algorithm 235: Random Permutation." In: *Commun. ACM* 7.7 (July 1964), p. 420. ISSN: 0001-0782. DOI: 10.1145/364520.364540. URL: https://doi.org/10.1145/364520.364540 (cit. on p. 42).

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. "Loop-Abort Faults on Lattice-Based Fiat-Shamir and Hash-and-Sign Signatures." In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Springer, Aug. 2016, pp. 140–158. DOI: 10.1007/978-3-319-69453-5_8 (cit. on p. 159).

[EGM96] Shimon Even, Oded Goldreich, and Silvio Micali. "On-Line/Off-Line Digital Signatures." In: *Journal of Cryptology* 9.1 (1996), pp. 35–67. ISSN: 1432-1378. DOI: 10.1007/BF02254791 (cit. on pp. 21, 91).

[FDK20] Apostolos P. Fournaris, Charis Dimopoulos, and Odysseas Koufopavlou. "Profiling Dilithium Digital Signature Traces for Correlation Differential Side Channel Attacks." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Ed. by Alex Orailoglu, Matthias Jung, and Marc Reichenbach. Springer,

2020, pp. 281–294. ISBN: 978-3-030-60939-9. DOI: 10.1007/978-3-030-60939-9_19 (cit. on p. 159).

[Fid72] Charles M. Fiduccia. "Polynomial Evaluation via the Division Algorithm: The Fast Fourier Transform Revisited." In: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*. ACM, 1972, pp. 88–93. DOI: 10.1145/800152.804900. URL: https://dl.acm.org/doi/10.1145/800152.804900 (cit. on p. 50).

[FK19] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Available at https://github.com/cloudflare/circl. v1.1.0 Accessed Feb 2022. June 2019 (cit. on p. 87).

[FS87] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, Aug. 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12 (cit. on pp. 38, 125, 129).

[FVFS21] Tim Fritzmann, Jonas Vith, Daniel Flórez, and Johanna Sepúlveda. "Post-quantum cryptography for automotive systems." In: *Microprocessors and Microsystems* 87 (2021), p. 104379. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2021.104379 (cit. on p. 113).

[GB08] Shafi Goldwasser and Mihir Bellare. *Lecture Notes on Cryptography*. 2008. URL: https://cseweb.ucsd.edu/~mihir/papers/gb.pdf (cit. on p. 16).

[GHKK+21] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. "Verifying Post-Quantum Signatures in 8 kB of RAM." In: *Post-Quantum Cryptography*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Springer, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12. URL: https://eprint.iacr.org/2021/662 (cit. on p. 21).

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. "Evaluation of Lattice-Based Signature Schemes in Embedded Systems." In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2018, pp. 385–388. DOI: 10.1109/ICECS.2018.8617969 (cit. on pp. 58, 61–64, 72, 73, 93, 113, 157).

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2021). Artifact available at https://artifacts.iacr.org/tches/2021/a1, pp. 1–24. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i1.1-24 (cit. on pp. 6, 7, 83–85, 88, 94, 105–109, 119, 131, 142–144, 154, 157, 200).

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)." In: *17th Annual ACM Symposium on Theory of Computing*. ACM Press, May 1985, pp. 291–304. DOI: 10.1145/22145.22178 (cit. on p. 38).

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. "Software Speed Records for Lattice-Based Signatures." In: *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013*. Ed. by Philippe Gaborit. Springer, June 2013, pp. 67–82. DOI: 10.1007/978-3-642-38616-9_5 (cit. on p. 62).

[GOPT09] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. "Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications." In: *ICISC 09: 12th International Conference on Information Security and Cryptology*. Vol. 5984. Lecture Notes in Computer Science. Springer, 2009, pp. 176–192. DOI: 10.1007/978-3-642-14423-3_13. URL: https://eprint.iacr.org/2009/538.pdf (cit. on pp. 28, 58).

[GP18] GlobalPlatform Technology. *Root of Trust Definitions and Requirements Version 1.1 (GP_REQ_025)*. 2018. URL: https://globalplatform.org/specs-library/root-of-trust-definitions-and-requirements-v1-1-gp-req_025/ (visited on Dec. 13, 2022) (cit. on p. 112).

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In: *40th Annual ACM Symposium on Theory of Computing*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 197–206. DOI: 10.1145/1374376.1374407 (cit. on p. 34).

[GR19] François Gérard and Mélissa Rossi. "An Efficient and Provable Masked Implementation of qTESLA." In: *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*. Ed. by Sonia Belaïd and Tim Güneysu. Vol. 11833. Lecture Notes in Computer Science. Springer, 2019, pp. 74–91. DOI: 10.1007/978-3-030-42068-0_5. URL: https://eprint.iacr.org/2019/606 (cit. on p. 59).

[Gre20] Denisa O. C. Greconici. "Kyber on RISC-V." MA thesis. Radboud University Nijmegen, 2020 (cit. on p. 55).

[GRI23] Michele Mosca and Marco Piani. *2023 Quantum Threat Timeline Report*. Tech. rep. Global Risk Institute, 2023. URL: https://globalriskinstitute.org/publication/2023-quantum-threat-timeline-report/ (visited on Jan. 23, 2024) (cit. on p. 3).

[Gro15] Wouter de Groot. "A Performance Study of X25519 on Cortex-M3 and M4." https://pure.tue.nl/ws/portalfiles/portal/47038543. MA thesis. Technische Universiteit Eindhoven, 2015 (cit. on p. 28).

[GS66] W. M. Gentleman and G. Sande. "Fast Fourier Transforms: for fun and profit." In: *FJCC 1966*. AFIPS 1966 (Fall). ACM, 1966, pp. 563–578. DOI: 10.1145/1464291.1464352. URL: https://dl.acm.org/doi/10.1145/1464291.1464352 (visited on Aug. 1, 2023) (cit. on pp. 54, 55, 75).

[Har09] David Harvey. "Faster polynomial multiplication via multipoint Kronecker substitution." In: *Journal of Symbolic Computation* 44.10 (2009), pp. 1502–1510. ISSN: 0747-7171. DOI: 10.1016/j.jsc.2009.05.004 (cit. on p. 97).

[HAZD+24] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. "Revisiting Keccak and Dilithium Implementations on ARMv7-M." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024 (2 2024). to appear. ISSN: 2569-2925 (cit. on pp. 157, 158).

[HKKP22] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-quantum, State Leakage Secure, and Deniable." In: *Journal of Cryptology* 35.3 (July 2022), p. 17. ISSN: 0933-2790. DOI: 10.1007/s00145-022-09427-1 (cit. on p. 159).

[HL23] Helena Handschuh and Anna Lysyanskaya, eds. *Advances in Cryptology – CRYPTO 2023, Part V*. Vol. 14085. Lecture Notes in Computer Science. Springer, Aug. 2023.

[HLKL+21] Jaeseung Han, Taeho Lee, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Jihoon Cho, Dong-Guk Han, and Bo-Yeon Sim. "Single-Trace Attack on NIST Round 3 Candidate Dilithium Using Machine Learning-Based Profiling." In: *IEEE Access* 9 (2021), pp. 166283–166292. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3135600 (cit. on p. 159).

[HPSW+20] Julius Hermelink, Thomas Pöppelmann, Marc Stöttinger, Yi Wang, and Yong Wan. "Quantum safe authenticated key exchange protocol for automotive application." In: *18th escar Europe : The World's Leading Automotive Cyber Security Conference (Konferenzveröffentlichung)*. Ruhr-Universität Bochum, 2020. DOI: 10.13154/294-7549 (cit. on p. 113).

[HRS16] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS." In: *Public-Key Cryptography – PKC 2016*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Springer, 2016, pp. 446–470. DOI: 10.1007/978-3-662-49384-7_17. URL: https://eprint.iacr.org/2015 (cit. on p. 21).

[HZZL+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. "Improved Plantard Arithmetic for Lattice-based Cryptography." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.4 (2022), pp. 614–636. ISSN: 2569-2925. DOI: 10.46586/tches.v2022.i4.614-636 (cit. on p. 158).

[IBM24] *Development & Innovation Roadmap*. 2024. URL: https://www.ibm.com/roadmaps/quantum/ (visited on Apr. 9, 2024) (cit. on p. 3).

[IEC10] International Electrotechnical Commission (IEC). *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC 61508. 2010 (cit. on p. 114).

[IETF18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments RFC 8446. Internet Engineering Task Force, 2018. URL: https://datatracker.ietf.org/doc/rfc8446/ (visited on Jan. 22, 2024) (cit. on p. 2).

[IETF21] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. *Certificate Transparency Version 2.0*. Request for Comments RFC 9162. Internet Engineering Task Force, 2021. URL: https://datatracker.ietf.org/doc/rfc9162/ (visited on Jan. 22, 2024) (cit. on p. 2).

[IETF23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol*. Request for Comments RFC 9420. Internet Engineering Task Force, 2023. URL: https://datatracker.ietf.org/doc/rfc9420/ (visited on Jan. 22, 2024) (cit. on p. 2).

[IMSS+22] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. "Signature Correction Attack on Dilithium Signature Scheme." In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 647–663. DOI: 10.1109/EuroSP53844.2022.00046 (cit. on p. 159).

[IRTF18] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391. URL: https://www.rfc-editor.org/info/rfc8391 (visited on Jan. 23, 2024) (cit. on p. 3).

[IRTF19] David McGrew, Michael Curcio, and Scott Fluhrer. *Leighton-Micali Hash-Based Signatures*. RFC 8554. Apr. 2019. DOI: 10.17487/RFC8554. URL: https://www.rfc-editor.org/info/rfc8554 (visited on Jan. 23, 2024) (cit. on p. 3).

[ISO18] International publisher for Standardization (ISO). *Road vehicles - Functional safety*. ISO 26262. 2018 (cit. on p. 114).

[ITU18] *Client to authenticator protocol/Universal 2-factor framework*. Recommendation X.1278. International Telecommunication Union, 2018. URL: https://www.itu.int/rec/T-REC-X.1278-201811-I/en (visited on Jan. 22, 2024) (cit. on p. 2).

[Jen24] Sönke Jendral. *A Single Trace Fault Injection Attack on Hedged CRYSTALS-Dilithium*. Cryptology ePrint Archive, Paper 2024/238. 2024. URL: https://eprint.iacr.org/2024/238 (cit. on p. 159).

[JL17] S. Josefsson and I. Luisvaara. *RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)*. Internet Research Task Force (IRTF), Jan. 2017 (cit. on p. 121).

[JSI96] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. "Designated Verifier Proofs and Their Applications." In: *Advances in Cryptology – EUROCRYPT'96*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Springer, May 1996, pp. 143–154. DOI: 10.1007/3-540-68339-9_13 (cit. on p. 159).

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. "Single-Trace Side-Channel Attacks on ω-Small Polynomial Sampling: With Applications to NTRU, NTRU Prime, and CRYSTALS-DILITHIUM." In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 35–45. DOI: 10.1109/HOST49136.2021.9702284 (cit. on p. 159).

[Kan22] Matthias J. Kannwischer. "Polynomial Multiplication for Post-Quantum Cryptograpy." PhD thesis. Radboud University, 2022. URL: https://hdl.handle.net/2066/247905 (cit. on pp. 55, 56).

[KBSV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. "Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 243–266. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.243-266. URL: https://tches.iacr.org/index.php/TCHES/article/view/7275 (cit. on pp. 28, 59).

[KGCK+20] Vinay BY Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kasper, Christoph Krauß, and Ruben Niederhagen. "Post-quantum secure boot." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1582–1585. DOI: 10.23919/DATE48585.2020.9116252 (cit. on p. 113).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, Aug. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25 (cit. on p. 119).

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 3rd edition. Chapman and Hall/CRC., 2020. ISBN: 978-1-35113-303-6. DOI: 10.1201/9781351133036 (cit. on p. 16).

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. "A Concrete Treatment of Fiat-Shamir Signatures in the Quantum Random-Oracle Model." In: *EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Lecture Notes in Computer Science. Springer, 2018, pp. 552–586. DOI: 10.1007/978-3-319-78372-7_18. URL: https://eprint.iacr.org/2017/916 (cit. on pp. 34, 127, 128, 130, 132, 138, 139).

[Knu02] Lars R. Knudsen, ed. *Advances in Cryptology – EUROCRYPT 2002*. Vol. 2332. Lecture Notes in Computer Science. Springer, Apr. 2002.

[Kob87] Neal Koblitz. "Elliptic curve cryptosystems." In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. ISSN: 1088-6842. DOI: 10.1090/S0025-5718-1987-0866109-5 (cit. on pp. 2, 16, 112).

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, Aug. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9 (cit. on p. 30).

[KPCS20] Panos Kampanakis, Peter Panburana, Michael Curcio, and Chirag Shroff. "Post-Quantum Hash-Based Signatures for Secure Boot." In: *Silicon Valley Cybersecurity Conference*. Springer, 2020, pp. 71–86. DOI: 10.1007/978-3-030-72725-3 (cit. on p. 113).

[Kro82] L. Kronecker. "Grundzüge einer arithmetischen Theorie der algebraischen Grössen." In: *Band 92*. Ed. by A. L. Crelle, C. W. Borchardt, and Schellbach. De Gruyter, 1882, pp. 1–122. ISBN: 978-3-1123-4240-4. DOI: 10.1515/9783112342404-001 (cit. on p. 97).

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. "Faster multiplication in $\mathbf{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates." In: *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. Lecture Notes in Computer Science. Springer, 2019, pp. 281–301. DOI: 10.1007/978-3-030-21568-2_14. URL: https://eprint.iacr.org/2018/1018.pdf (cit. on pp. 28, 59).

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. Cryptology ePrint Archive, Report 2019/844. https://eprint.iacr.org/2019/844. 2019 (cit. on p. 87).

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Improving Software Quality in Cryptography Standardization Projects." In: *ieee European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010. URL: https://eprint.iacr.org/2022/337 (cit. on pp. 105, 107).

[LDKL+19] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions. National Institute of Standards and Technology, 2019 (cit. on p. 65).

[LDKL+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-

3-submissions. National Institute of Standards and Technology, 2020 (cit. on pp. 40, 119).

[LDKL+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 45).

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. "SWIFFT: A Modest Proposal for FFT Hashing." In: *Fast Software Encryption*. Ed. by Kaisa Nyberg. Springer, 2008, pp. 54–72. ISBN: 978-3-540-71039-4. DOI: 10.1007/978-3-540-71039-4_4 (cit. on p. 78).

[LN16] Patrick Longa and Michael Naehrig. "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography." In: *CANS 2016: Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Vol. 10052. Springer, 2016, pp. 124–139. DOI: 10.1007/978-3-319-48965-0_8. URL: https://eprint.iacr.org/2016 (cit. on p. 56).

[LNP22] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General." In: *Advances in Cryptology – CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Springer, Aug. 2022, pp. 71–101. DOI: 10.1007/978-3-031-15979-4_3 (cit. on p. 159).

[LNS20] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. "Practical Lattice-Based Zero-Knowledge Proofs for Integer Relations." In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1051–1070. DOI: 10.1145/3372297.3417894 (cit. on p. 146).

[LOKV20] Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, eds. *ACM CCS 2020: 27th Conference on Computer and Communications Security*. ACM Press, Nov. 2020.

[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. ISSN: 0925-1022. DOI: 10.1007/s10623-014-9938-4. URL: https://eprint.iacr.org/2012/090 (visited on June 6, 2023) (cit. on p. 34).

[LW05] Joseph K. Liu and Duncan S. Wong. "Linkable Ring Signatures: Security Models and New Schemes." In: *Computational Science and Its Applications – ICCSA 2005*. Ed. by Osvaldo Gervasi, Marina L. Gavrilova, Vipin Kumar, Antonio Laganà, Heow Pueh Lee, Youngsong Mun, David Taniar, and Chih Jeng Kenneth Tan. Springer, 2005, pp. 614–623. DOI: 10.1007/11424826_65 (cit. on p. 159).

[Lyu09] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures." In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Springer, Dec. 2009, pp. 598–616. DOI: 10.1007/978-3-642-10366-7_35. URL: https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf (cit. on pp. 34, 38, 125).

[Lyu12] Vadim Lyubashevsky. "Lattice Signatures without Trapdoors." In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, Apr. 2012, pp. 738–755. DOI: 10.1007/978-3-642-29011-4_43 (cit. on pp. 34, 146).

[MF21] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles*. Ed. by David Basin and Kenny Paterson. Springer, 2021. ISBN: 978-3-030-63286-1. DOI: 10.1007/978-3-030-63287-8 (cit. on p. 16).

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. "Masking Dilithium - Efficient Implementation and Side-Channel Evaluation." In: *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. Lecture Notes in Computer Science. Springer, June 2019, pp. 344–362. DOI: 10.1007/978-3-030-21568-2_17. URL: https://eprint.iacr.org/2019/394 (cit. on pp. 31, 58, 64, 73, 159).

[Mil86] Victor S. Miller. "Use of Elliptic Curves in Cryptography." In: *Advances in Cryptology – CRYPTO*. Ed. by Hugh C. Williams. Vol. 218. Springer, 1986, pp. 417–426. ISBN: 978-3-540-39799-1. DOI: 10.1007/3-540-39799-X_31 (cit. on pp. 2, 16, 112).

[Mon85] Peter L. Montgomery. "Modular multiplication without trial division." In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1985-0777282-X. URL: https://www.ams.org/mcom/1985-44-170/S0025-5718-1985-0777282-X/ (visited on Feb. 14, 2023) (cit. on pp. 23, 25, 61).

[MTKS+20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. "Compact domain-specific co-processor for accelerating module lattice-based KEM." In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218727 (cit. on p. 59).

[MUTS22] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. *Profiling Side-Channel Attacks on Dilithium: A Small Bit-Fiddling Leak Breaks It All*. Cryptology ePrint Archive, Report 2022/106. https://eprint.iacr.org/2022/106. 2022 (cit. on p. 159).

[NIST13] *FIPS186-4: Digital Signature Standard (DSS)*. Tech. rep. National Institute of Standards and Technology (NIST), 2013. DOI: 10.6028/NIST.FIPS.186-4. URL: https://doi.org/10.6028/NIST.FIPS.186-4 (visited on Jan. 23, 2024) (cit. on p. 3).

[NIST15a] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202 (cit. on p. 4).

[NIST15b] *Secure Hash Standard*. National Institute of Standards and Technology, NIST FIPS PUB 180-4, U.S. Department of Commerce. Aug. 2015 (cit. on p. 121).

[NIST16] National Institute of Standards and Technology. *Post-Quantum Cryptography – Post-Quantum Cryptography Standardization*. 2016. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization (visited on Sept. 18, 2023) (cit. on pp. 3, 44, 162, 165, 184).

[NIST18] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. *NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*. Tech. rep. National Institute of Standards and Technology (NIST), 2018. DOI: 10.6028/NIST.SP.800-56Ar3. URL: https://doi.org/10.6028/NIST.SP.800-56Ar3 (visited on Jan. 23, 2024) (cit. on p. 3).

[NIST19a] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. NIST IR 8240. National Institute of Standards and Technology, 2019. DOI: 10.6028/NIST.IR.8240.

URL: https://csrc.nist.gov/pubs/ir/8240/final (visited on Jan. 25, 2024) (cit. on p. 3).

[NIST19b] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. *NIST SP 800-56B Rev. 2: Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography*. Tech. rep. National Institute of Standards and Technology (NIST), 2019. DOI: 10.6028/NIST.SP.800-56Br2. URL: https://doi.org/10.6028/NIST.SP.800-56Br2 (visited on Jan. 23, 2024) (cit. on p. 3).

[NIST20a] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. NIST IR 8309. National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.IR.8309. URL: https://csrc.nist.gov/pubs/ir/8309/final (visited on Jan. 25, 2024) (cit. on p. 3).

[NIST20b] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. *Recommendation for Stateful Hash-Based Signature Schemes*. SP 800-208. National Institute of Standards and Technology, 2020 (cit. on p. 113).

[NIST22a] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. NIST IR 8314. National Institute of Standards and Technology, 2022. DOI: 10.6028/NIST.IR.8413-upd1. URL: https://csrc.nist.gov/pubs/ir/8413/upd1/final (visited on Jan. 25, 2024) (cit. on pp. 3, 158).

[NIST22b] National Institute of Standards and Technology. *Post-Quantum Cryptography: Digital Signature Schemes*. 2022. URL: https://csrc.nist.gov/Projects/pqc-dig-sig/standardization (visited on Dec. 18, 2023) (cit. on p. 145).

[NIST23a] *Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process*. 2023. URL: https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals (visited on Jan. 25, 2024) (cit. on pp. 3, 159).

[NIST23b] Lily Chen, Dustin Moody, Andrew Regenscheid, and Angela Robinson. *FIPS 186-5: Digital Signature Standard (DSS)*. 2023. DOI: https://doi.org/10.

6028/NIST.FIPS.186-5. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=935202 (visited on Feb. 27, 2024) (cit. on p. 115).

[NSA22] *Announcing the Commercial National Security Algorithm Suite 2.0.* Cybersecurity Advisory. National Security Agency (NSA), 2022. URL: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF (visited on Jan. 23, 2024) (cit. on p. 3).

[Nus81] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Ed. by King Sun Fu, Thomas S. Huang, and Manfred R. Schroeder. Vol. 2. mySpringer Series in Information Sciences. Springer, 1981. ISBN: 978-3-662-00553-8. DOI: 10.1007/978-3-662-00551-4. URL: http://link.myspringer.com/10.1007/978-3-662-00551-4 (visited on Aug. 28, 2023) (cit. on p. 56).

[Odl87] Andrew M. Odlyzko, ed. *Advances in Cryptology – CRYPTO'86.* Vol. 263. Lecture Notes in Computer Science. Springer, Aug. 1987.

[PAAB+19] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. *NewHope.* Submission to the NIST Post-Quantum Cryptography Standardization Project [NIST16]. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. 2019 (cit. on p. 66).

[PFHK+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON.* Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on pp. 3, 77).

[PKCS198] *PKCS #1: RSA Cryptography Standard.* RSA Data Security, Inc. Version 2.0. 1998, archived at https://web.archive.org/web/20160410165357/http://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf on Apr. 10, 2016 (cit. on p. 115).

[Pla21] Thomas Plantard. "Efficient Word Size Modular Arithmetic." In: *IEEE Transactions on Emerging Topics in Computing* 9.3 (2021), pp. 1506–1518. ISSN: 2168-6750. DOI: 10.1109/TETC.2021.3073475 (cit. on pp. 15, 158).

[Pol71] J. M. Pollard. "The fast Fourier transform in a finite field." In: *Mathematics of Computation* 25.114 (1971), pp. 365–374. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1971-0301966-0 (cit. on p. 50).

[Por19] Thomas Pornin. *New Efficient, Constant-Time Implementations of Falcon*. Cryptology ePrint Archive, Report 2019/893. 2019. URL: https://eprint.iacr.org/2019/893 (cit. on p. 59).

[PPRS23] Rafaël del Pino, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. "High-Order Masking of Lattice Signatures in Quasilinear Time." In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1168–1185. DOI: 10.1109/SP46215.2023.10179342 (cit. on p. 31).

[PQM4] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. URL: https://github.com/mupq/pqm4 (visited on May 30, 2023) (cit. on pp. 29, 31, 71, 78, 83, 85, 87, 88, 105, 119).

[PS00] David Pointcheval and Jacques Stern. "Security Arguments for Digital Signatures and Blind Signatures." In: *Journal of Cryptology* 13.3 (June 2000), pp. 361–396. ISSN: 0933-2790. DOI: 10.1007/s001450010003 (cit. on p. 38).

[RCDB23] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. "Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results." In: *ACM Trans. Embed. Comput. Syst.* (2023). ISSN: 1539-9087. DOI: 10.1145/3603170. URL: https://eprint.iacr.org/2022/737 (cit. on p. 159).

[Reg05] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography." In: *37th Annual ACM Symposium on Theory of Computing*. Ed. by Harold N. Gabow and Ronald Fagin. ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603 (cit. on pp. 4, 33).

[RGCB19] Prasanna Ravi, Sourav Sen Gupta, Anupam Chattopadhyay, and Shivam Bhasin. "Improving Speed of Dilithium's Signing Procedure." In: *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*. Ed. by Sonia Belaïd and Tim Güneysu. Vol. 11833. Lecture Notes in Computer Science. Springer, 2019, pp. 57–73. DOI: 10.1007/978-3-030-42068-0_4. URL: https://eprint.iacr.org/2019/420.pdf (cit. on pp. 58, 61, 72, 73, 87, 91–93, 109, 113, 144, 157).

[Rij19] Joost Rijneveld. "Practical Post-Quantum Cryptography." PhD thesis. Radboud University, 2019. ISBN: 978-9-463-32568-4. URL: https://hdl.handle.net/2066/208551 (cit. on p. 13).

[RJHC+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. *Side-channel Assisted Existential Forgery Attack on Dilithium - A NIST PQC candidate*. Cryptology ePrint Archive, Report 2018/821. https://eprint.iacr.org/2018/821. 2018 (cit. on p. 159).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342 (cit. on pp. 2, 16, 112, 115).

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret." In: *Advances in Cryptology – ASIACRYPT 2001*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Springer, Dec. 2001, pp. 552–565. DOI: 10.1007/3-540-45682-1_32 (cit. on p. 159).

[SABD+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 77).

[Sch90] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards." In: *Advances in Cryptology – CRYPTO'89*. Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. Springer, Aug. 1990, pp. 239–252. DOI: 10.1007/0-387-34805-0_22 (cit. on p. 125).

[SECG00] *Certicom Research, Standards for Efficient Cryptography Group (SECG) — SEC 1: Elliptic Curve Cryptography*. Version 1.0. 2000. URL: http://www.secg.org/secg_docs.htm (cit. on p. 115).

[Sei18] Gregor Seiler. *Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography*. Cryptology ePrint Archive, Report 2018/039. https://eprint.iacr.org/2018/039. 2018 (cit. on pp. 25, 56, 61, 80).

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *35th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700 (cit. on pp. 2, 112).

[Sin00] Simon Singh. *The code book: the science of secrecy from ancient Egypt to quantum cryptography*. 1st edition. Anchor Books, 2000. ISBN: 978-1-85702-879-9 (cit. on p. 1).

[SKSB20] Steffen Sanwald, Liron Kaneti, Marc Stöttinger, and Martin Böhner. "Secure boot revisited: challenges for secure implementations in the automotive domain." In: *SAE International Journal of Transportation Cybersecurity and Privacy* 2.11-02-02-0008 (2020), pp. 69–81. ISSN: 2572-1046. DOI: 10.4271/11-02-02-0008 (cit. on p. 113).

[Sol11] Jerome A. Solinas. "Generalized Mersenne Prime." In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: mySpringer US, 2011, pp. 509–510. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_32. URL: https://doi.org/10.1007/978-1-4419-5906-5_32 (cit. on p. 23).

[SS19] Peter Schwabe and Amber Sprenkels. "The Complete Cost of Cofactor $h = 1$." In: *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*. Ed. by Feng Hao, Sushmita Ruj, and Sourav Sen Gupta. Vol. 11898. Lecture Notes in Computer Science. Springer, Dec. 2019, pp. 375–397. DOI: 10.1007/978-3-030-35423-7_19. URL: https://eprint.iacr.org/2019/1166 (cit. on p. 200).

[SS71] Arnold Schönhage and Volker Strassen. "Schnelle Multiplikation großer Zahlen." In: *Computing* 7.3-4 (1971), pp. 281–292. ISSN: 0010-485X. DOI: 10.1007/BF02242355 (cit. on p. 79).

[STM20a] STMicroelectronics. *STM32F407VG*. 2020. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f407vg.html (visited on May 22, 2023) (cit. on p. 29).

[STM20b] STMicroelectronics. *STM32F4DISCOVERY*. 2020. URL: https://www.st.com/en/evaluation-tools/stm32f4discovery.html (visited on May 22, 2023) (cit. on p. 29).

[SW21] Amber Sprenkels and Bas Westerbaan. *Don't throw your nonces out with the bathwater: Speeding up Dilithium by reusing the tail of y*. Cryptology ePrint Archive. 2021. URL: https://eprint.iacr.org/2020/1158 (visited on Jan. 29, 2024) (cit. on p. 9).

[UEFI22] *Unified Extensible Firmware Interface version (Release 2.10)*. Specification. UEFI Forum, Inc., 2022. URL: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf (visited on Jan. 22, 2024) (cit. on p. 2).

[Vig21] Benoît Viguier. "A Panorama on Classical Cryptography." PhD thesis. Radboud University, 2021. ISBN: 978-9-463-32806-7. URL: https://hdl.handle.net/2066/241047 (cit. on p. 13).

[W3C21] Emil Lundberg, Akshay Kumar, J.C. Jones, Michael Jones, and Jeff Hodges. *Web Authentication: An API for accessing Public Key Credentials - Level 2*. W3C Recommendation. https://www.w3.org/TR/2021/REC-webauthn-2-20210408/. W3C, 2021 (cit. on p. 2).

[Wig24] Thom Wiggers. "Post-Quantum TLS." PhD thesis. Radboud University, 2024. ISBN: 978-9-464-73330-3. URL: https://hdl.handle.net/2066/300702 (cit. on p. 13).

[WTJB+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. "Parameterized Hardware Accelerators for Lattice-Based Cryptography." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020), pp. 269–306. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.269-306. URL: https://tches.iacr.org/index.php/TCHES/article/view/8591 (cit. on pp. 59, 96).

[ZZWY+21] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. "A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2021), pp. 270–295. ISSN: 2569-2925. DOI: 10.46586/tches.v2022.i1.270-295. URL: https://tches.iacr.org/index.php/TCHES/article/view/9297 (cit. on p. 23).

Summary

Digital signature schemes are one of the core building blocks in modern cryptography. They protect data against any kind of unauthorized modification. Unfortunately, most digital signature schemes that are currently in use will be broken with the advent of cryptographically relevant quantum computers. In order to remain secure in their presence, we must research new post-quantum digital signature schemes. One of these schemes is the *Dilithium* signature scheme. In this thesis, we evaluate whether Dilithium is suitable for implementation and deployment on embedded platforms.

This question is evaluated from different perspectives. In Chapters 4, 5 and 8, we look at the speed of the scheme. That is, we evaluate how long it takes to run Dilithium's algorithms, and how their run times can be further optimized. In Chapter 6, we look at its memory usage. In Chapter 7, we implement the scheme on the S32G274A vehicle network processor and add it as an option from image verification in the chip's secure-boot mechanism.

Chapter 4. In this chapter, we present implementations of the Dilithium scheme for the two types of microcontrollers Arm Cortex-M3 and Cortex-M4. On Cortex-M3, one of the big challenges is the availability of appropriate instructions for multiplication, as the most powerful multiply instructions have timings that depend on their data. This leaks information about the numbers being multiplied, which makes the instructions unsuitable for secret cryptographic computations. To solve this issue, we propose two new routines for integer multiplication on Cortex-M3 that are based on smaller instructions that all execute in constant-time.

Chapter 5. During Dilithium's signature generation algorithm, some polynomials are always "small", i.e. all of their coefficients are always between $-\beta$ and β (where $\beta \in \{78, 196, 120\}$). This allows us to compute their polynomial products modulo a smaller value of $q' \in \{257, 769\}$ instead of the larger Dilithium modulus $q = 8380417$.

With this in mind, we build new optimized NTT and polynomial multiplication algorithms using the smaller moduli and integrate them into the signing algorithm for Arm Cortex-M4.

Chapter 6. Most microcontroller implementations of Dilithium use 50 to 100 KiB of RAM, depending on the Dilithium variant that is used. However, many chips only have up to 8 to 16 KiB of RAM, preventing Dilithium from being used on these devices. In this chapter, we implement the Dilithium algorithm, but optimize for memory usage rather than speed. This results in the first implementation of Dilithium for which the recommended parameter set requires less than 7 KiB of memory for key and signature generation and less than 3 KiB of memory for signature verification.

Chapter 7. In this chapter, we investigate the practical impact of migrating the secure boot flow on a vehicle network processor towards post-quantum cryptography. We create a fault-attack-resistant (against single-targeted fault) implementation of Dilithium signature verification, which we incorporate into the secure boot flow of the Hardware Security Engine of the S32G274A.

Chapter 8. The Dilithium signature generation is built around a rejection-sampling loop: First a *nonce* (number-only-used-once) vector is generated, then a candidate signature is generated, and finally the algorithm checks whether the candidate signature does not leak any information about the secret key. If the candidate signature is safe to be output, the signing algorithm returns that value; otherwise, all the intermediate results are discarded, and the process is restarted using a new nonce vector. This routine is repeated until a safe signature is found. We propose a modification to the Dilithium algorithm where—when a candidate signature is deemed unsafe—we reuse some of the nonce material that is still safe for usage in subsequent iterations of the rejection-sampling loop, instead of completely discarding all the nonce material. With our modification, only part of the nonce vector needs to be regenerated, which slightly improves the speed of the Dilithium signing algorithm.

Samenvatting

Digitale handtekeningsystemen zijn een veelgebruikte bouwsteen in de cryptografie. Ze beveiligen digitale informatie (data) tegen ongeautoriseerde wijzigingen. Wanneer cryptografisch relevante kwantumcomputers in de toekomst gerealiseerd worden, zullen deze de meest gebruikte digitale handtekeningsystemen eenvoudig kunnen kraken. Daarom moet er onderzoek worden gedaan naar nieuwe *post-kwantum* digitale handtekeningsystemen die nog steeds veilig zijn wanneer dit soort kwantumcomputers daadwerkelijk bestaan. Een van de opties is het handtekeningsysteem *Dilithium*. In dit proefschrift beoordelen we of Dilithium geschikt is voor implementatie en gebruik in ingebedde systemen.

We benaderen deze vraag vanuit verschillende invalshoeken. In Hoofdstukken 4, 5 en 8 kijken we naar de snelheid van het systeem. Oftewel, we bekijken de rekentijd van Dilithium, en we zoeken optimalisaties om de duur verder te verkorten. In Hoofdstuk 6 kijken we naar de hoeveelheid geheugen die het systeem gebruikt; en in Hoofdstuk 7 gebruiken we het systeem als onderdeel van het secure-boot mechanisme van de S32G274A voertuignetwerkprocessor.

Hoofdstuk 4. In dit hoofdstuk presenteren we implementaties van Dilithium voor twee categorieën van microcontrollers, namelijk de Arm Cortex-M3 en Cortex-M4. Een belangrijk obstakel op Cortex-M3 is de afwezigheid van snelle instructies voor vermenigvuldiging, omdat de meest voor de hand liggende instructies een uitvoertijd hebben die afhankelijk is van de data waarop ze uitgevoerd worden. Dit lekt informatie over de getallen die vermenigvuldigd worden, wat de instructies ongeschikt maakt voor cryptografische berekeningen op geheime data. Daarom gebruiken we twee nieuwe procedures voor de vermenigvuldiging van getallen op Cortex-M3, die intern gebruik maken van instructies die altijd dezelfde uitvoertijd hebben.

Hoofdstuk 5. In het algoritme wat handtekeningen genereert zijn er een aantal polynomen die altijd "klein" zijn; oftewel, al hun coëfficiënten hebben een waarde tussen $-\beta$ en β (met $\beta \in \{78, 196, 120\}$). Dit kunnen we gebruiken om het vermenigvuldigen van deze polynomen te versnellen door kleinere moduli $q' \in \{257, 769\}$ te gebruiken in plaats van de grotere Dilithium-modulus $q = 8380417$. We maken nieuwe geoptimaliseerde NTT-algoritmes gespecialiseerd voor deze kleinere moduli en gebruiken ze in het genereren van handtekeningen op Cortex-M4.

Hoofdstuk 6. De meeste implementaties van Dilithium voor microcontrollers gebruiken 50 tot 100 KiB geheugen, afhankelijk van de variant van Dilithium die gebruikt wordt. Veel chips hebben echter maar 8 tot 16 KiB geheugen beschikbaar, waardoor er geen implementaties zijn die op deze chips uitgevoerd kunnen worden. In dit hoofdstuk implementeren we het Dilithium-algoritme opnieuw, maar dit keer optimaliseren we voor een vermindering in geheugengebruik in plaats van uitvoertijd. Dit leidt tot de ontwikkeling van de eerste implementatie van Dilithium waarvan de standaardvariant minder dan 7 KiB geheugen gebruikt voor het genereren van sleutels en handtekeningen, en minder dan 3 KiB geheugen gebruikt voor het controleren van handtekeningen.

Hoofdstuk 7. In dit hoofdstuk kijken we naar de impact die de migratie naar post-kwantum cryptografie heeft op de secure-boot implementaties van voertuignetwerkprocessors. Hiervoor maken we een glitch-bestendige implementatie voor de controle van Dilithium-handtekeningen, en integreren we deze in de secure-boot implementatie van de Hardware Security Engine van de S32G274A chip.

Hoofdstuk 8. Het Dilithium-handtekeningenalgoritme doet meestal meerdere pogingen tot het genereren van een handtekening, voordat er succesvol een handtekening gemaakt kan worden. Er wordt eerst een uniform-willekeurige *nonce-vector* gegenereerd, waarna er met die nonce-vector een voorlopige handtekening wordt gemaakt. Tenslotte controleert het algoritme of de voorlopige handtekening geen informatie over de geheime sleutel bevat. Als dit wél het geval is, dan worden alle tussentijdse variabelen gewist en start het proces opnieuw met een nieuwe nonce-vector. Deze procedure wordt dan herhaald totdat er een geschikte handtekening is gevonden. In dit hoofdstuk stellen we een aanpassing aan het algoritme voor, die – wanneer

een voorlopige handtekening ongeschikt blijkt – een deel van de nonce-vector hergebruikt in opvolgende pogingen van het algoritme, in plaats van de nonce-vector altijd volledig te wissen. Met onze aanpassing hoeft soms alleen een deel van de nonce-vector opnieuw gegenereerd te worden, wat leidt tot een snellere versie van het handtekeningenalgoritme.

Acknowledgements

I feel like I cannot really claim that I wrote this dissertation by myself, as I have had an unmeasurable amount of support from the people in my life during the five years that led to this point. This thesis would not have been written without you. I hope I have made you proud. <3

First, I would like to thank the Radboud University, NXP Semiconductors, and MPI-SP for hosting me during my PhD. All have been excellent places to work, and I will miss the atmosphere of their offices. I will miss my desk in the room formerly called PQHQ; the desk that has been my second home for five years.

Furthermore, I would like to acknowledge the Horizon 2020 research and innovation program of the ERC (Starting Grant 805031) and NXP Semiconductors, as all of the research bundled within this thesis has been funded by one of these sources.

To my supervisors:

Peter, I feel immensely lucky to have crossed your path. You have been an amazing supervisor. From the start you have submerged me in a deluge of enthusiasm, confidence, research ideas, and connections. Your authenticity and faithfulness know no limits. For every challenge, be it intellectual, emotional, or interpersonal, I could always rely on you to support me and to guide me towards the next challenge. I am grateful for your pragmatic attitude, your encouragement to visit conferences and academic friends, and your recognition of engineering as a core part of cryptographic research. I am grateful for the confidence that you have had in me, even in moments when my own supply of confidence was depleted. May your food be excellent, and your beer be cold.

Joppe, with confidence I can say that my time at NXP has been the most productive time during all of my PhD. Most of all this has been because of your amazing ability to lead and manage me. Naturally, I am very grateful that you were willing to extend our professional relationship beyond my internship at NXP by becoming my second

supervisor. In your role, you were always quick to recognize and ward off distractions to my work, and to steer my focus (back) towards the goal of a project. Thank you for teaching me how to bring projects (like writing a PhD thesis) to a finished state.

During these five years, I have had the pleasure to meet and work with so many people. This is the part where I will have to concede that I will not be able to call out everybody by name. Thank you all nonetheless.

First and foremost, thank you to my collaborators, especially **Amin**, **Bas**, **Benoît**, **Denisa**, **Fabio**, **Joost**, **Joppe**, **Matthias**, **Vincent**, and **Yi**. I feel privileged to have been able to work together with bright minds like you. Thank you for complementing the gaps in my abilities, as I have done my best to complement yours. I am proud of the work we produced together.

Thank you to **Azade**, **Benoît**, **Erik**, **Jan**, **Joppe**, **Marrit**, **Matthias**, **Peter**, **Pol**, **Thom**, and **Vincent** for proofreading parts of this thesis and preventing many errors from surviving until the final version. Thanks to the **anonymous reviewers** of my papers for your helpful comments. Thanks to the reading committee, **prof. dr. Lejla Batina**, **dr. Vadim Lyubashevsky**, **prof. dr. ir. Nele Mentens**, **prof. dr. Damien Stehlé**, **prof. dr. Bo-Yin Yang**, for offering your time to review my manuscript.

Ronny, thank you for being such a stable bright presence in the digital security department. Thank you **Irma**, **Janet**, **Shanley**, **Simone**, for keeping it running.

Thanks to my (ex-)colleagues, **Anna**, **Azade**, **Benoît**, **Christine**, **Denisa**, **Joost**, **Joost**, **Krijn**, **Marloes**, **Matthias**, **Pol**, **Thom**; for sharing shots of Fireball in the late evenings; for taking early morning dives in the Adriatic Sea with me; for the KLM wine we drank when the world was ending. Thank you for our discussions and enlightenment; for the camaraderie that we enjoyed together; and for being the role models that you are.

A huge thanks goes out to the cryptographic community. Thank you for providing a welcoming space for me to meet people, and to learn about your different cryptographic challenges and solutions. Thank you to all the **friends that I have met along the way**. The actual list of names is too long to list here, and attempting to list them all would be ill-advised—as I would surely forget some of them. Thank you for our talks; for looking after me in unsafe situations; for our late-night parties on the Croatian beach or in underground Praguan karaoke bars; for that time I had a rough incident at RWC'23, and we skipped the next session to get ice cream together.

A special shout-out goes out **Florine** and **Julius**. You have emotionally supported me more than you probably realize.

I am furthermore very grateful for the opportunity to travel as much as I have. Thanks to everybody that has helped facilitate this, as well as everybody that participated with me. In particular, thank you **Bo-Yin Yang** for inviting me to Taipei twice.

To, **andżela** and **S1mba**, I am grateful that fate led us together during Covid times. Thanks for the hundreds of fun games of CS over the years. Let's keep rushing B and do not stop.

Then there are all of my friends at #RU. Although I usually describe us as *a chat group of friends* to people in the out-group, this does not even remotely convey how much time we spend together. I feel connected to you in a way that is truly unique. I like climbing and bouldering and going on weekends together; I like our movie nights, our pub evenings, our camping trips, and all the other things we do. It does not matter what we do specifically, because I really just like *hanging out* with you. I am grateful for our friendship and I hope that it may be everlasting.

Special thanks go out to all the folks in #academia. Having your support during my PhD felt like having a cheat code. Thank you for your all the invaluable information; and for allowing me to vent, and then responding with effective pep talks.

My close friends, **Abel**, **Annelies***, **Annemiek***, **Bart**, **Erik**, **Gerdriaan**, **Hanne***, **Jille**, **Joost**, **Judith**, **Linda***, **Loeka**, **Loeka**, **Mara***, **Margot**, **Margot**, **Marrit***, **Pax**, **Pol**, **Rian**, **Rik***, **Sjors***, and **Yorick**; I love you guys. (To the stars* among you, thank you for protecting me and supporting me during a ride on a rollercoaster of self-discovery.) Thank you for your company, and for your support in times of tough decisions. You mean the world to me.

Lastly, thanks to my family, **Frank**, **Marianne**, **Rik**, **Bram**, **Koen**, **Sanne**, **Annemiek**, and **Carla**, for your persistent love and support.

And to **Marrit**: I love you. Let's have many more adventures together.

Amber Sprenkels
Nijmegen, June 2024

About the author

Amber Sprenkels was born on the 12th of April 1994 in Heesch, The Netherlands. After graduating from high school she studied Chemistry at the Faculty of Science of Radboud University in Nijmegen, from which she received her Bachelor's degree in 2016. Thereafter, she switched to computing science and started specializing in cybersecurity and cryptographic engineering. Her Master's thesis, entitled *ECC implementation on Sandy Bridge: The cost of cofactor $h = 1$*, was supervised by Peter Schwabe. She received her Master's degree in Computing Science in 2019.

After obtaining her Master's degree, she continued working with Schwabe at Radboud University, pursuing a PhD on the implementation of post-quantum cryptography. During her PhD, she has interned as a cryptographic engineer with NXP Semiconductors in 2021–2022, supervised by dr. Joppe Bos. This thesis presents a selection of her work from 2019 to 2023.

List of publications

The following list is a list of academic publications that Amber has coauthored (in reverse-chronological order):

- Joppe W. Bos, Brian Carlson, Joost Renes, Marius Rotaru, Amber Sprenkels, and Geoffrey P. Waters. "Post-quantum secure boot on vehicle network processors." In: *20th escar Europe - The World's Leading Automotive Cyber Security Conference (15. - 16.11.2022)*. Ruhr-Universität Bochum, 2022, pp. 112–125. DOI: 10.13154/294-9372. URL: https://eprint.iacr.org/2022/635

- Joppe W. Bos, Joost Renes, and Amber Sprenkels. "Dilithium for Memory Constrained Devices." In: *AFRICACRYPT 2022: 13th*. Ed. by Lejla Batina and Joan Daemen. Vol. 2022. Lecture Notes in Computer Science. Springer, July

2022, pp. 217–235. DOI: 10.1007/978-3-031-17433-9_10. URL: https://eprint.iacr.org/2022/323

- Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. "Faster Kyber and Dilithium on the Cortex-M4." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, June 2022, pp. 853–871. DOI: 10.1007/978-3-031-09234-3_42. URL: https://eprint.iacr.org/2022/112

- Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2021). Artifact available at https://artifacts.iacr.org/tches/2021/a1, pp. 1–24. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i1.1-24

- Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Amber Sprenkels, and Benoît Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?." In: *CANS 20: 19th International Conference on Cryptology and Network Security*. Ed. by Stephan Krenn, Haya Shulman, and Serge Vaudenay. Vol. 12579. Lecture Notes in Computer Science. Springer, Dec. 2020, pp. 526–545. DOI: 10.1007/978-3-030-65411-5_26. URL: https://eprint.iacr.org/2020/836

- Peter Schwabe and Amber Sprenkels. "The Complete Cost of Cofactor $h = 1$." In: *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*. Ed. by Feng Hao, Sushmita Ruj, and Sourav Sen Gupta. Vol. 11898. Lecture Notes in Computer Science. Springer, Dec. 2019, pp. 375–397. DOI: 10.1007/978-3-030-35423-7_19. URL: https://eprint.iacr.org/2019/1166


9 789493 391673