Polynomial Multiplication for Post-Quantum Cryptography

Matthias J. Kannwischer

Cover design: @WinloxArts with Jonalyn Madriaga and Felix Wilhelm
Printed by GVO drukkers & vormgevers

This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE).

Polynomial Multiplication for Post-Quantum Cryptography

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

maandag 4 april 2022
om 12:30 uur precies

door

Matthias Julius Kannwischer

geboren op 20 april 1993
te Tübingen, Duitsland

Promotoren:

prof. dr. Peter Schwabe

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

Manuscriptcommissie:

prof. dr. Lejla Batina (voorzitter)

prof. dr. Daniel J. Bernstein
University of Illinois Chicago, Verenigde Staten en
Ruhr-Universität Bochum, Duitsland

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

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

dr. Joppe W. Bos
NXP Semiconductors, België

Acknowledgements

First and foremost, I want to thank Peter Schwabe for supervising me over the last few years. From the moment we met in Tenerife, you supported me in my research and always made time for me when I asked for advice. I admire your positive attitude to virtually everything which helped me a lot to get through the frustrating phases of my PhD. I am very grateful that you allowed and encouraged me to attend numerous academically valuable conferences, workshops, and schools in many touristically valuable places. Thank you, Peter!

I also want to thank my second supervisor Bo-Yin Yang. You intended to invite me for a 6-week research visit to Taipei in March 2020. Now, two years later, I am still here. Thank you for jumping through all the hoops to allow me to stay and come back! I am also honored that you agreed to be my second supervisor and I enjoyed the many technical and non-technical discussions we had. And Sukiyaki! Also, thanks to your assistant Sinman who helped me with a lot of Mandarin paperwork and supplied me with food during my first quarantine.

I also want to thank my previous supervisor at the University of Surrey, Liqun Chen. You introduced me to cryptography when I was working at HP Labs in 2014, and encouraged me to pursue a PhD. Without you, I would not be where I am today!

I thank the members of my reading committee Lejla Batina, Daniel J. Bernstein, Joppe W. Bos, Tim Güneysu, and Ingrid Verbauwhede for taking the time to read this thesis and providing me valuable feedback.

Even though there is one name on this book, this is the result of collaborations with numerous people. It was great fun to be able to work with so many different people from around the world. Thank you to my co-authors Albrecht, Amin, Andreas, Andrew, Aymeric, Bo-Yin, Cheng-Jhih, Daan, Denis, Denisa, Dieter, Elisabeth, Fabio, Gregor, Hanno, Hervé, Hiroshi, Jacques, Jintai, Jiun-Peng, Johannes, Joost, Juliane, Ko, Leon, Marc, Marvin, Michael, Ming-Shing, Peter, Peter, Robert, Ruben, Shang-Yi, Tanja, Thom, Tung, Vincent, and Yu-Jia.

I thank everyone who has proofread parts of this thesis beforehand. Thanks Amin, Daan, Fabio, Jonathan, Krijn, Lorenz, Pedro, and Thom.

I am grateful for having had the opportunity to teach and supervise various students over the years. I enjoyed it very much and you taught me much more than I could ever teach you.

Having worked in four institutes in four countries over the course of my PhD, I have met many great people and made a lot of friends. Thank you for a great time!

Thank you, Dan and Jorden. You introduced me to the PhD student life and British customs. I hope we will go to Greggs soon. Also thanks to François, Ioana, Johnni, and Linda for the great (although brief) time in Guildford.

Thank you to the many great people in Nijmegen. Thanks for coffee breaks, crypto dinners, vrjidag middag borrels, barbeques, and movie nights. In particular, I want to thank Joost, Ko, and Pedro. I appreciated all your help when starting in Nijmegen.

Thanks, Pedro, for still advising me today about the problems of life and Montgomery multiplications.

I moved flats many times in the last years, but only once did I move flats while physically being on the other end of the world. Thank you, Loes, Joost, Veelasha, and Peter for supporting this crazy endeavor.

For my time in Bochum, I want to thank Irmgard for helping me a great lot with paperwork. Thank you, Nils, for generously offering to supply me with food during yet another involuntary quarantine.

I thank Dor, Lorenz, and Jonathan for sharing an office with me in Taipei and exploring this exceptionally beautiful country. Thank you, Dan and Tanja, for the many nice hot pot dinners.

Thanks, Fabio, for your support and helping me to accept that even my code sometimes contains bugs.

I want to thank my friends at home. Even though my decision to move abroad meant I could not see you very often, it is great to not be forgotten. Thank you for thinking of me, staying in touch, and for making me feel welcome when I was visiting home. In particular, I want to thank Anna, Felix, Martin, Max, Michael, Paul, Sabrina, and Tim.

I thank my family for always supporting me and trying very hard to understand what I am doing in my research. Thanks especially to my parents, Bärbel and Helmut, for always helping and encouraging me! Thank you also to Christian and Thomas for numerous skiing trips and inspiring advice. Thank you, Winfried, for exciting me about science. Thank you, grandma, Karin, Keith, Sabine, too.

Jonalyn, I thank you for making the last two years very special and fun. Thank you, for your patience, support, love, and all the delicious food!

Thank you all.

Matthias Kannwischer
Taipei, January 2022

Contents

Chapter 1

Introduction

Protecting means of communication has been a human need since ancient times. With the advancement of public-key cryptography in the 1970s [DH76], secure communication without a shared secret key became feasible for the first time. With the rise of the internet, this technology became widespread and the vast majority of individuals, corporations, and governments rely on public-key cryptography daily.

At the core of any secure communication, we are interested in protecting the confidentiality, integrity, and authenticity of data. Confidentiality refers to the inability of unauthorized parties to read the contents of the exchanged messages. The integrity of messages implies that the receiver can be sure that the message has not been manipulated by a third party. Lastly, authenticity guarantees that a message was transmitted by the claimed sender. Cryptography offers a way to guarantee these security goals using cryptographic schemes. Most commonly, we use encryption for achieving confidentiality, and digital signatures or message-authentication codes (MAC) for achieving integrity and authenticity. We distinguish two categories of cryptographic schemes: symmetric (or secret-key) cryptography and asymmetric (or public-key) schemes. While secret-key cryptography requires both the sender and receiver to share a cryptographic secret (e.g., a long random password) that must not be known by any adversary, public-key cryptography works fundamentally different: One party generates a key pair consisting of a secret key and a public key. The public key is shared with the world (e.g., posted on a website), while the secret key must remain private. In the case of public-key encryption (PKE), it then suffices to be in possession of the public key to encrypt messages which can then only be decrypted by the owner of the secret key. For digital signatures, a message is signed using the secret key and the signature can be verified using the corresponding public key.

However, public-key cryptography is much more costly than secret-key cryptography both in terms of computation time and data that needs to

be transmitted. Hence, it is customary to combine public-key cryptography with a secret-key primitive, i.e., encrypt a small symmetric key (e.g., 256 bit) using PKE which is then used to key a symmetric cipher. A public-key scheme that can only be used for encrypting such a short key is referred to as a key-encapsulation mechanism (KEM). A KEM, however, is more general than a fixed-size PKE as it is not necessarily the case that the key can be chosen. It can also be derived as a part of the KEM.

Unfortunately, the advancement of quantum computers threatens the security of public-key cryptography. This is mostly due to Shor's algorithm [Sho94] which was published in 1994. Shor's algorithm allows to factor a large integer into its prime factors on a quantum computer in polynomial time while the best-known classical algorithms require super-polynomial time. Also, Shor's algorithm allows computing discrete logarithms in polynomial time. This has devastating consequences as common public-key schemes rely on the hardness of factoring or computing discrete logarithms: RSA [RSA78] relies on factoring, DH [DH76] relies on the discrete-logarithm problem (DLP), elliptic-curve cryptography (ECC) [Mil85] relies on the elliptic-curve DLP. Once a large-scale quantum computer is available, cryptographic schemes building upon these problems can be broken and it is possible to compute the secret keys from the public keys. Hence, all encrypted messages can be decrypted, and all digital signatures are useless as one can simply forge signatures. Even worse, it is possible to record encrypted messages now and decrypt them using a future quantum computer. Even though large-scale quantum computers are not existing today, recent progress in their development suggests that they might exist in the next decades [MP21].

This motivates the need to migrate away from these schemes based on factoring or the DLP to schemes that resist attacks by both classical computers and also quantum computers. This type of cryptography is called post-quantum cryptography and started attracting interest in the cryptographic research community in the 2000s which resulted in proposals over the last two decades. Cryptography that cannot resist quantum attacks is called classical cryptography in the following. Note that the use of post-quantum cryptography does not require a quantum computer, i.e., it can be deployed on classical computers.

Aside from Shor's algorithm, there is one other noteworthy quantum algorithm that needs to be taken into account: Grover's algorithm [Gro96]. It allows searching a set of size n in $\sqrt{n}$ steps, a task that would require n steps on a classical computer. This also applies to cryptography: It can be used to search a cryptographic key by enumerating all possible keys (i.e., brute-forcing). This mostly impacts symmetric cryptography as the best-known attacks are often close to brute-forcing. Luckily, this square-root speed-up can be countered by doubling the key sizes. For example, one could use a key length of 256 bits rather than 128 bits when using the Advanced

family	KEM/PKE	signature
Code	18	3
Hash	0	2
Isogeny	1	0
Lattice	21	5
MQ	3	7
Other	6	3
Total	49	20

Table 1.1: 69 *complete and proper* submission to the NISTPQC [NIS16]

Encryption Standard (AES) [DR02] to achieve comparable post-quantum security. Hence, the term post-quantum cryptography commonly refers to quantum-resistant public-key cryptography.

Due to the emerging threat of quantum computers, the US National Institute for Standards and Technology (NIST) has announced in 2016 [NIS16] that it plans to replace their standards based on the DLP and factoring with post-quantum alternatives. Specifically, this affects their standards for key establishment (NISTSP800-56 [Nat18, Nat19a]), and digital signatures (FIPS186-4 [Nat13]). The replacements are to be determined in a public competition for which a call for proposals was published in 2016 with a deadline for submissions in late 2017. We refer to the competition as NISTPQC in the following. Since key-establishment protocols can be built from either public-key encryption algorithms or key-encapsulation algorithms, and, hence, NIST decided to accept submissions for both. In contrast to the previous AES and SHA-3 competitions, NIST does not plan to select a single winner for each category, but expects to select multiple schemes. Therefore, it is sometimes referred to as the NISTPQC project (or *not-a-competition*) rather than a competition. Similar to previous NIST competitions, NIST aims to standardize multiple parameter sets providing different levels of security. To enable this, NIST defined security levels one to five providing at least the classical and post-quantum security as AES128, SHA256, AES192, SHA384, and AES256, respectively. Most submissions only provide parameter sets that target security levels one, three, and five. NISTPQC consists of multiple rounds of evaluation and at the end of each round, NIST selects a subset of proposals for the next round and publishes a report justifying their selection. NIST received 82 submissions of which it deemed 69 as *complete and proper* [NIS16]. Among those submissions were all five major families of schemes that are conjectured to resist quantum attacks: code-based cryptography, hash-based cryptography, isogeny-based cryptography, lattice-based cryptography, and multivariate-based cryptography. See Table 1.1 for an overview distribution of the submissions. While they all come with their advantages and disadvantages, all have promising instantiations which may

	KEM/PKE	signature
Code	**BIKE**, **Classic McEliece**, **HQC**, LEDAcrypt, NTS-KEM, ROLLO, RQC	–
Hash	–	**SPHINCS$^+$**
Isogeny	**SIKE**	–
Lattice	**FrodoKEM**, **Kyber**, LAC, NewHope, **NTRU**, **NTRU Prime**, Round5, **Saber**, Three Bears	**Dilithium**, **Falcon**, qTesla
MQ	–	**GeMSS**, LUOV, MQDSS, **Rainbow**
Other	–	**Picnic**

Table 1.2: Second-round and third-round NISTPQC candidates. Third round candidates are **bold**. Third round finalists are additionally **underlined**.

be useful for certain use cases. The code-based schemes in the NIST competition are mostly key-establishment schemes. They are characterized by large public keys and small ciphertexts. Hash-based cryptography can only be used to construct digital signatures, but not key establishment. It uses small public keys but results in large signatures. Isogeny-based cryptography only has a single candidate in the NIST competition: SIKE [JAC+17], which is a key-encapsulation mechanism. It has small public keys and ciphertexts but is significantly slower than other schemes. Lattice-based cryptography is the most prominent and arguably the promising family as it has reasonable ciphertext, signature, and public-key sizes, and also has a good performance. Last but not least, there are schemes based on the hardness of solving multivariate quadratic (MQ) equations. In the NIST competition, MQ schemes are mostly signature schemes. They are characterized by small signatures, but large public keys. There were also nine submissions that do not fit any of these five families. Sadly, none of the submissions comes without any disadvantages when compared to classical schemes like ECC: They either have much larger keys, larger ciphertexts, larger signatures, or are much slower.

In 2019, NIST published their first-round report [Nat19b] and announced the start of a second round with 26 candidate schemes advancing of which 17 are key-establishment schemes and nine are signature schemes. The second-round candidates are shown in Table 1.2. It is notable that all code-based signatures and all multivariate key-establishment schemes have been eliminated. Half of the candidates are based on lattices and still represent the most popular family of schemes. Note that only one scheme in the *other* family has survived the first round: Picnic [ZCD+17]. It is built using sym-

metric cryptography and zero-knowledge proofs. All other schemes have either been broken or been eliminated due to a lack of confidence in their security.

Another year later in mid-2020, NIST announced the third round [Nat20] with 15 schemes advancing. NIST distinguishes between seven finalists and eight alternate schemes. NIST states that finalists are likely ready for standardization at the end of the third round in case they are selected, while alternative schemes will require more research and further rounds of evaluation. It also states that due to similarities of the proposals it will standardize at most one of Kyber, NTRU, and Saber and will standardize at most one of Dilithium or Falcon, but not both.

NIST states that besides the security of the schemes and their key and message sizes, an important factor for selecting the future PQC standard is good performance [Nat19b]. Especially beyond the first round, performance is one of the most crucial factors unless significant cryptographic advances threaten the security of some schemes. It is essential that the selected schemes perform well on a wide range of platforms from smart cards and microcontrollers to high-end servers in a data center. To enable fair comparisons of implementations, NIST initially recommended that submission teams focus on 64-bit Intel processors.[1] At the end of the first round, NIST announced that in addition to Intel processors, it would like the community to focus on Arm Cortex-M4 microcontrollers and Artix-7 FPGAs.[2]

In this thesis, I study how to implement selected NISTPQC candidates on the Arm Cortex-M4 to achieve the best performance. Compared to Intel processors, working on microcontrollers presents some challenges for PQC:

- Most importantly, memory is much more limited on microcontrollers, often not exceeding 100 kB. Many NISTPQC submissions require significant changes to the implementations to even be functional on such devices. Some schemes even have keys that are larger than the available memory which often hinders their use.

- The instruction sets of microcontrollers are much more limited than those of high-end Intel CPUs. While Intel implementations often exploit data-level parallelism using vector instructions, microcontrollers offer no or very limited ways to exploit data-level parallelism.

- Cryptographic hardware accelerators (e.g., for AES or SHA-2) are much less common in microcontrollers. While virtually all high-end processors now have specialized instructions for symmetric cryptography, these are usually non-existent on small devices. This results in

[1] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-process

[2] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ

a big slow-down for PQC schemes relying on symmetric cryptography as a building block which is the case for most schemes.

- Microcontrollers run at a much lower frequency than high-end CPUs, which results in some schemes taking seconds or even minutes. For many use cases, this can be prohibitively slow.

- Energy consumption is a much larger concern for microcontrollers as they are often embedded into battery-powered systems.

- Library support is much more limited for microcontrollers. Even if libraries are available, they are often not heavily optimized for the target architecture. This usually requires replacing all libraries with custom optimized code.

Overarching of the work presented in this thesis is the `pqm4` [KPR+] project which was started in early 2018 by Rijneveld, Stoffelen, Schwabe, and myself. `pqm4` is a unified testing and benchmarking framework for NISTPQC schemes on the Arm Cortex-M4. Initially, the goal was to make as many schemes as possible work while making sure that the implementations are compatible with the reference implementations running on a high-end CPU. The unified benchmarking framework was used by many optimization papers to obtain fair performance metrics including speed, memory consumption, and code size. To ensure consistency and accessibility, `pqm4` collects all optimized (open-source) implementations and posts the performance benchmarks online.

Research Data Management. 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. The research datasets produced during this PhD research packaged into a single archive are available at `https://doi.org/10.5281/zenodo.5555735`. For more details, see Appendix A.

1.1 Contributions

The content in the main body of this thesis is the result of collaboration and publication with multiple co-authors. As usual in the field of cryptography, all publications list authors in alphabetical order as it is usually not possible to identify one main author. Nonetheless, my contributions to these publications vary and the following section outline which parts are my contributions.

Part I: Multiplication for NTT-friendly Rings

In the first content part of this thesis, we study implementations of post-quantum KEMs and signature schemes which have been specifically designed to benefit from using number-theoretic transforms (NTTs) for polynomial arithmetic. The part is based on two publications published at Africacrypt 2019 and TCHES 2021.

> Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology – Africacrypt 2019*, LNCS, pages 209–228. Springer, 2019. `https://eprint.iacr.org/2019/489`

This work, contained in Chapter 3, studies Cortex-M4 implementations of the Kyber KEM both with and without the changes that have been introduced in the second round of the NIST competition. The paper discusses implementations optimized for speed and for stack consumption. I wrote all the assembly code involved in the polynomial arithmetic. The stack optimized implementations are based on a Cortex-M0 implementation by Leon Botros, that I ported to the M4 implementation together with him. The writing of the paper was a joint effort of all authors.

> Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. `https://eprint.iacr.org/2020/1278`

The second publication is presented in Chapter 4. It presents new speed-records for the signature scheme Dilithium on the Cortex-M4, and additionally presents the first implementations for Dilithium, Kyber, and NewHope on the Cortex-M3. We provide several time-memory trade-offs for the signature generation of Dilithium. The Cortex-M3 implementation of the Dilithium NTT is the core contribution to which all authors contributed equally. Additionally, I wrote the faster Cortex-M4 NTT as well as the Cortex-M3 implementations of Kyber and NewHope. Paper writing was distributed equally between authors.

Part II: Multiplication for NTT-unfriendly Rings

While some schemes are specifically designed to benefit from NTT multiplication, other lattice-based schemes are built using different polynomial rings which allow various ways of implementing polynomial arithmetic. The most prominent schemes in that realm are Saber and NTRU which to date are still candidates for the NISTPQC standardization. Implementations of those were studied in two papers published at ACNS 2019 and TCHES 2021.

> Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security – ACNS 2019*, LNCS, pages 281–301. Springer, 2019. `https://eprint.iacr.org/2018/1018`

Chapter 5 covers the earlier work which describes how five first-round candidates (including Saber and NTRU) relying on polynomial multiplication in $\mathbb{Z}_{2^m}[x]$ can be efficiently implemented on the Cortex-M4 using Karatsuba multiplication and Toom–Cook multiplication. As the polynomial rings used in these five schemes vary significantly, the main contribution of this work is a code generator supporting arbitrary polynomial degrees and allowing a number of different combination of multiplication methods. The code generation was written in close collaboration with Joost Rijneveld with the help of the blackboard in our beautiful corner office in Nijmegen. The paper was written by all three authors. We have received the ACNS 2019 best student paper award for this paper.

> 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. `https://eprint.iacr.org/2020/1397`

The second publication presents how polynomial multiplication can be implemented even more efficiently by using the NTT. The details are covered in Chapter 6. We showcase the superiority of NTT implementations compared to Toom–Cook and Karatsuba multiplication for the three KEMs NTRU, Saber, and LAC and target the Cortex-M4 and AVX2. While the former two are currently still under consideration by NIST, the later scheme has been chosen as a winner of the Chinese Association for Cryptologic Research (CACR) post-quantum competition. The Cortex-M4 code is a result of a summer internship by Chi-Ming Marvin Chung, Vincent Hwang, and Cheng-Jhih Shih at Academia Sinica, Taiwan that was supervised by Bo-Yin Yang and me. Vincent Hwang and I got these implementations into shape for publication. The AVX2 implementations are by Gregor Seiler. The paper was written by Vincent Hwang, Gregor Seiler, Bo-Yin Yang, and me. We have received the TCHES 2021 best artifact award for the code submitted alongside this paper.

Other publications

I have chosen not to include various other publications in this thesis, as it allows me to spend more words on connecting the selected work and elaborate

on the common preliminaries in greater depth. The peer-reviewed academic publications not appearing in this thesis in reverse chronological order are:

- 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2021. `https://eprint.iacr.org/2021/986`

- 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2021. `https://eprint.iacr.org/2021/995`

- 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 – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. `https://eprint.iacr.org/2021/662`

- Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):650–675, 2021. `https://eprint.iacr.org/2021/532`

- Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with dummy-operations against fault injection attacks. In *Workshop on Fault Detection and Tolerance in Cryptography*, pages 57–65, 2020. `https://eprint.iacr.org/2020/1005`

- Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):243–268, 2021. `https://eprint.iacr.org/2020/371`

- Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2018*, pages 168–188. Springer, 2018. `https://eprint.iacr.org/2018/673`

Beyond these formally published publications, the following papers have been accepted to conferences and workshops without formal proceedings:

- Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NISTPQC on ARM

Cortex-M4. In *Second NIST PQC Standardization Conference*, 2019. https://eprint.iacr.org/2019/844

- Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. In *Kangacrypt*, 2018. https://eprint.iacr.org/2018/674

In addition to these papers, I am involved in the NISTPQC signature finalist Rainbow:

- Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://www.pqcrainbow.org/

Chapter 2

Preliminaries

This chapter presents the preliminaries that are common among the following chapters of this thesis. Section 2.1 introduces the basic terminology of key-establishment and digital signature schemes. Section 2.2 presents polynomial-multiplication techniques with a focus on fast software implementations. Section 2.3 introduces algorithms for fast modular arithmetic, which is used as a building block of polynomial multiplication. Lastly, Section 2.4 presents the primary optimization platforms throughout this thesis: the Arm Cortex-M4 and Arm Cortex-M3.

2.1 Cryptographic Schemes

The NISTPQC competition seeks post-quantum replacements for the NIST standards for key-establishment schemes and digital signature schemes. The following sections introduce the terminology that is commonly used when describing those schemes. It discusses the security properties that are generally desired for such schemes. While these definitions are not limited to post-quantum cryptography, a scheme is called post-quantum in case the security properties still hold in case the adversary has a large-scale quantum computer.

2.1.1 Key-Establishment Schemes

The objective of key establishment is for two (or more) parties to agree on a shared secret key. Cryptographically this can be achieved in multiple different ways including either a PKE, a KEM, or a non-interactive key exchange (NIKE). Up until recently, post-quantum schemes were limited to either PKE or KEM. In 2018, after the deadline of the NIST project has passed, also post-quantum NIKE based on isogenies has been proposed [CLM+18].

However, as this came long after NIST called for proposals, NIST only allowed PKE and KEM submissions. As shown in the following, it is easy to construct a KEM from a PKE and vice versa.

A PKE scheme consists of three algorithms: `KeyGen`, `Encrypt`, and `Decrypt`:

`KeyGen()` takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

`Encrypt(pk, m)` takes as inputs a public key `pk` and a message `m`, which is usually a bitstring. It outputs the encrypted message as ciphertext `c`.

`Decrypt(sk, c)` takes as inputs a secret key `sk` and a ciphertext `c`. It outputs the decrypted message `m'`.

A PKE scheme is correct if `Decrypt(sk,Encrypt(pk,m)) = m` for any key pair (`pk`, `sk`) produced by `KeyGen`() and any message `m`. It is called a deterministic PKE (DPKE) if `Encrypt` outputs the same ciphertext if called with the same inputs multiple times. Any non-deterministic PKE can be turned into a DPKE by slightly changing the API and making the randomness an explicit argument, e.g., by adding an argument `seed` that is used to derive all randomness pseudo-randomly.

A key-encapsulation mechanism (KEM) consists of three algorithms: `KeyGen`, `Encaps`, and `Decaps`:

`KeyGen()` takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

`Encaps(pk)` takes as input a public key `pk` and returns a ciphertext `c` and a session key `ss`.

`Decaps(sk, c)` takes as input a secret key `sk` and a ciphertext `c`. It outputs a session key `ss'`.

A KEM is correct if given $\texttt{c}, \texttt{ss} \leftarrow \texttt{Encaps(pk)}$, `Decaps(sk,c) = ss` for any key pair (`pk`, `sk`) produced by `KeyGen`(). To formalize the security of cryptographic schemes one usually describes the capabilities of the most powerful attack that a cryptographic scheme can resist. The most common security notion of PKE schemes and KEMs is indistinguishability under chosen-plaintext attacks (IND-CPA; CPA for short) and indistinguishability under chosen-ciphertext attacks (IND-CCA; CCA for short). Informally, these notations can be explained as follows. Indistinguishability refers to the inability of an attacker to reliably distinguish which of two ciphertexts corresponds to a given message. In the CPA setting, the adversary can encrypt as many messages as needed to obtain information from the ciphertexts that help to distinguish the given setting. This implies that a DPKE can never be CPA secure as the adversary can simply encrypt the given message. In

the CCA setting, the adversary is more powerful: He can actively choose ciphertexts (different from the target ciphertexts) and ask a decryption oracle to decrypt those. The information obtained from those queries must not help him to distinguish the ciphertexts. CPA attacks are also called passive attacks, while CCA attacks are called active attacks referring to the adversary actively tampering with some ciphertexts to obtain information about the secret key.

In practice, in most settings, the desired security notion is CCA security as it can often not be guaranteed that the adversary is unable to mount active attacks. For example, consider a public web server responding to requests. An adversary can send crafted ciphertexts to the web server that it will decrypt; the behavior of the web server often can be used to obtain some information about the secret key involved. Hence, it is usually only acceptable to use a CPA-secure scheme if each public key is only used once.

Luckily, there exist constructions that transform CPA-secure schemes into CCA-secure schemes. The most common one is the Fujisaki–Okamoto (FO) [FO99] transform which allows transforming a CPA-secure DPKE into a CCA-secure KEM. This approach is used by a large number of NISTPQC schemes including most of the ones covered in this thesis. Informally, the FO transform allows the decrypting party to verify that the ciphertext was honestly generated or if it was crafted. In case a dishonest ciphertext is detected, a random key is returned such that no information about the secret key is revealed. For a more formal description of the construction refer to [HHK17].

2.1.2 Digital Signature Schemes

A digital signature scheme consists of three algorithms: `KeyGen`, `Sign`, and `Open`:

`KeyGen()` takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

`Sign(sk, m)` takes as inputs a secret key `sk` and a message `m` which is usually a bitstring of arbitrary length. It outputs a signed message `sm` commonly consisting of the concatenation of the message itself and a signature.

`Open(pk, sm)` takes as inputs a public key `pk` and a signed message `sm`. If the signature is valid under `pk`, it returns the message `m`. Otherwise, it returns an error.

A digital signature scheme is called correct if $\texttt{Open}(\texttt{pk}, \texttt{Sign}(\texttt{sk}, \texttt{m})) = \texttt{m}$ for any (`pk`, `sk`) produced by `KeyGen`() and any message `m`. There is an alternative definition of signature schemes where `Sign` returns a signature rather than a signed message, which is then verified using an algorithm

`Verify`. In that case, it is left to the user of the cryptographic scheme to decide what to do in case a signature is invalid. This can have catastrophic consequences as users (e.g., implementers using a cryptographic library) will often ignore such failure and proceed as usual. In the vast majority of cases, the sensible action to do is to discard the message and trigger some kind of error handling as the message cannot be trusted. This is the behavior enforced by the `Sign`/`Open` definition of signature schemes. NIST is following the `Sign`/`Open` definitions and requires the submitted software to adhere to the APIs reflecting them.

For a digital signature scheme to be secure, we require that it is impossible to produce a valid signed message `sm` without the knowledge of the secret key `sk`. The most common security notion for signature schemes is existential unforgeability under chosen message attacks (EU-CMA). Informally, it implies that an adversary is incapable to produce a signature for any message (existential forgery) even if he is allowed to obtain signatures for other messages, i.e., has access to a signing oracle.

2.2 Polynomial Multiplication for Computer Scientists

For thousands of years, humankind has studied how to efficiently multiply, either in anticipation of intriguing applications like cryptography or for its mere mathematical beauty. Hence, we can now choose from a plethora of multiplication methods that may or may not be useful for the problem at hand. Polynomial multiplication is a core building block for cryptography of all kinds. In particular, the vast majority of post-quantum schemes use polynomial multiplication. A closely related task is the multiplication of large integers which is used in yet another large number of cryptographic schemes.

Even more, multiplication methods are constantly adapted and optimized for a certain instance, such that any given implementation of polynomial multiplication contains dozens of tricks that have been discovered in decades or even centuries of research. This makes it incredibly hard to understand a given implementation without reading a huge number of papers. This is amplified by the fact that most tricks are simply part of all modern implementations, but rarely fully explained or attributed to the corresponding publications.

To date, I am not aware of any good comprehensive introduction to all polynomial multiplication methods used in state-of-the-art implementations of post-quantum cryptographic schemes. The renowned work by Bernstein [Ber01] is an excellent survey of numerous tricks and concisely presents them for a reader with experience in the underlying mathematical concepts. However, coming from a computer science background and lacking the nec-

essary mathematical background, it appears to be hardly decipherable. As I struggled for many years to conceive the tricks and I have seen many students struggle with it in the same way, I have decided to try to bridge this gap and write an introduction to polynomial multiplication for computer scientists.

In the following sections, I present every single way of multiplying polynomials that I have encountered in actual implementations of post-quantum cryptographic schemes over the years. Alongside the descriptions in this thesis, each section comes with sample implementations in Python and C. I aimed for this chapter to be brief, self-contained, and easy to follow. In parts, I skip over a lot background and formal definitions and instead provide examples that I hope provide a good intuition and serve as a starting point. For a more formal treatment, I recommend reading [vzGG13], [Nus82], and of course [Ber01].

The code is available at `https://github.com/mkannwischer/polymul` and can be freely used under a CC0 copyright waiver. All source code related to this thesis is also available in a single archive. See Appendix A.

Notation. Let $\mathbb{Z}$ be the ring of integers, and $\mathbb{Z}_q$ be the integer ring containing $\{0, ..., q-1\}$ with q being a positive integer and arithmetic being performed modulo q. We write $\mathbb{Z}[x]$ to denote the polynomial ring in the variable x with integer coefficients, and $\mathbb{Z}_q[x]$ to denote the polynomial ring with coefficients in $\mathbb{Z}_q$. Given a polynomial a in some polynomial ring, we write a_i to denote the coefficient corresponding to x^i, i.e, $a = \sum_{i=0}^{n-1} a_i x^i$. As these polynomial rings have an infinite number of elements, they are not particularly useful for cryptography. Hence, one uses a finite polynomial ring instead by computing modulo a certain polynomial $f(x)$ with degree n, such that polynomials remain at degree at most $n-1$, i.e., n coefficients. We write $\mathbb{Z}_q[x]/(f(x))$ to denote the ring with all operations modulo $f(x)$ and q. Sometimes we require to split a polynomial a into multiple parts by setting $y = x^k$ for some $k < n$. To denote the part i, we write $a^{(i)}$, such that $a = \sum_{i=0}^{\lceil n/k \rceil} a^{(i)} y^i$.

Example 1: Consider the polynomial ring $\mathbb{Z}_2[x]/(x^2+1)$. This polynomial ring has four elements: $\{0, 1, x, x+1\}$. We can now multiply two polynomials, e.g., $x \cdot (x+1) = x^2 + x \equiv x + 1 \pmod{q, x^2+1}$ as $x^2 \equiv -1 \pmod{q, x^2+1}$.

For cryptographic purposes, one uses polynomial rings with many more elements. In the following we are mostly considering rings that are used in cryptographic schemes based upon structured lattices, but all of the algorithms find application far beyond. A common choice for q is either a power of two or a prime, commonly below 32 or 16 bits, such that one or two coefficients fit neatly into a processor word on all popular platforms. A common choice for n is between 256 and 1024. For some polynomial multiplication

algorithms, n is ideally a power of two as well, but that is not always the case in cryptographic schemes. A very common choice for $f(x)$ is $x^n + 1$ or $x^n - 1$ as the reduction is simple to implement and can often be done on the fly.

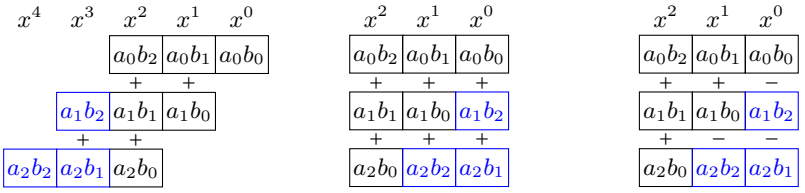
Application: Kyber [ABD+17] uses the ring $\mathbb{Z}_{3329}[x]/(x^{256} + 1)$, Saber [DKRV17] uses $\mathbb{Z}_{8192}[x]/(x^{256} + 1)$, NTRU [ZCH+19] uses (among others) $\mathbb{Z}_{8192}[x]/(x^{701} - 1)$, and Dilithium [LDK+17] uses $\mathbb{Z}_{8380417}[x]/(x^{256} + 1)$.

Coefficient Multiplication. Every polynomial-multiplication algorithm requires multiplying elements in $\mathbb{Z}_q$. Since q is chosen such that coefficients fit into registers, multiplication can in most cases use the available multiplication instructions which multiply mod 2^k with $k \in 16, 32, 64$. If q is a power of two, one can simply use the instruction $2^k > q$ and obtain the standard representative $(0, ..., q-1)$ using a logical AND with $q-1$. For intermediate values, the standard representative is usually not required, and one can omit the reductions. However, if q is not a power of two, the multiplication needs to be performed mod $2^k > q^2$ and needs to be followed by an explicit reduction modulo q to bring coefficients back to a single word. Furthermore, additions and subtractions of coefficients cause them to grow and one needs to be careful to reduce them before they overflow the word size. For reductions after multiplications, one commonly uses Montgomery reductions [Mon85], while for reductions after additions, one can use Montgomery reductions, Barrett reductions [Bar86], or specialized reductions for special primes, e.g., Solinas primes [Sol99]. For describing the polynomial-multiplication algorithms, we assume that modular multiplications in $\mathbb{Z}_q$ can be performed efficiently and cover the concrete algorithms separately in Section 2.3.

Convolution. In literature about polynomial multiplication one often also reads about convolution, positively wrapped (or cyclic) convolution, and negatively wrapped (or negacyclic) convolution. In general, convolution (written as $*$) of two functions $f(x)$ and $g(x)$ is defined as [Nus82, Sec 2.2.4]

$$[f * g](x) = \int f(\tau)g(x - \tau)\, d\tau.$$

However, if f and g are polynomials in $\mathbb{Z}[x]$ (or $\mathbb{Z}_q[x]$), the convolution of f and g is equivalent to polynomial multiplication. Similarly, cyclic convolution is equivalent to multiplication in $\mathbb{Z}[x]/(x^n - 1)$ (or $\mathbb{Z}_q[x]/(x^n - 1)$), and negacyclic convolution is equivalent to multiplication in $\mathbb{Z}[x]/(x^n + 1)$ (or $\mathbb{Z}_q[x]/(x^n + 1)$). These terms are often used interchangeably.



(a) $\mathbb{Z}[x]$ (b) $\mathbb{Z}[x]/(x^3-1)$ (c) $\mathbb{Z}[x]/(x^3+1)$

Figure 2.1: Schoolbook multiplication of polynomials $a = a_2x^2 + a_1x + a_0$ and $b = b_2x^2 + b_1x + b_0$

2.2.1 Schoolbook Multiplication

Before diving into the different algorithms available for polynomial multiplication, it makes sense to revisit the problem at hand and its straightforward solution: Given two n-coefficient polynomials a, b in some polynomial ring, we want to compute the product $a \cdot b$. In case the polynomial ring is $\mathbb{Z}[x]$ or $\mathbb{Z}_q[x]$, the multiplication is defined as

$$a \cdot b = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j \cdot x^{i+j}.$$

In case the polynomial ring is $\mathbb{Z}[x]/(x^n - 1)$, the multiplication becomes

$$a \cdot b \equiv \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} a_i \cdot b_j \cdot x^{i+j} + \sum_{j=1}^{n-1} \sum_{i=n-j}^{n-1} a_i \cdot b_j \cdot x^{i+j-n} \pmod{x^n - 1}.$$

Similarly, for $\mathbb{Z}[x]/(x^n + 1)$ the multiplication is defined as

$$a \cdot b \equiv \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} a_i \cdot b_j \cdot x^{i+j} - \sum_{j=1}^{n-1} \sum_{i=n-j}^{n-1} a_i \cdot b_j \cdot x^{i+j-n} \pmod{x^n + 1}.$$

As we usually work with polynomial rings over $\mathbb{Z}_q$ rather than over $\mathbb{Z}$, the intermediate addition and multiplication of coefficients can be performed modulo q.

Example 2: Let $a = x^2 + 2x + 3$, $b = x^2 + x$.
In $\mathbb{Z}[x]$, the product is $x^4 + 3x^3 + 5x^2 + 3x$.
In $\mathbb{Z}[x]/(x^3 - 1)$, the product is $(5x^2 + 3x) + (x + 3) = 5x^2 + 4x + 3$.
In $\mathbb{Z}[x]/(x^3 + 1)$, the product is $(5x^2 + 3x) - (x + 3) = 5x^2 + 2x - 3$.

These three multiplications are illustrated for 3-coefficient polynomials in Figure 2.1. When implementing schoolbook multiplication, there exist two

different approaches: Either one fixes a coefficient a_i and iterates through all coefficients of b. This corresponds to computing one row in Figure 2.1 and is called operand scanning. Alternatively, one uses product scanning, where one column in Figure 2.1 is computed at a time.

Note that the number of multiplications of coefficients required for all these algorithms is n^2, while the number of additions is $(n-1)^2$. For small n it is often the case that schoolbook multiplication is actually the fastest approach available as most other approaches work by breaking down a larger multiplication into multiple smaller ones which incur some overhead that may outweigh the gain. The actual cut-off point for each method not only depends on the polynomial ring, but also on the available multipliers and adders on the target platform and their respective performance characteristics. Hence, it is important to understand the optimal approach to implement schoolbook multiplication.

Application: Schoolbook multiplication of small polynomials is often used as a building block in other multiplications methods. Chapter 5 describes extensive optimization on the Arm Cortex-M4 of schoolbook multiplications for polynomials with eight to 16 coefficients applicable, for example, to Saber [DKRV17] and NTRU [ZCH+19].

2.2.2 Karatsuba Multiplication

Karatsuba's multiplication method [KO63] allows breaking down a large polynomial multiplication (n-coefficient multiplicands) into three smaller polynomial multiplications with $n/2$-coefficient multiplicands. The following example illustrates Karatsuba's idea.

Example 3: Consider the most straightforward example with 2-coefficient inputs $a = a_1x + a_0$ and $b = b_1x + b_0$. The schoolbook method from the previous section would compute the product $a \cdot b$ as

$$a \cdot b = (a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_0b_1 + a_1b_0)x + a_0b_0.$$

This requires four coefficient multiplications and one coefficient addition. The Karatsuba algorithm exploits the fact that

$$(a_0b_1 + a_1b_0) = (a_0 + a_1)(b_0 + b_1) - a_1b_1 - a_0b_0.$$

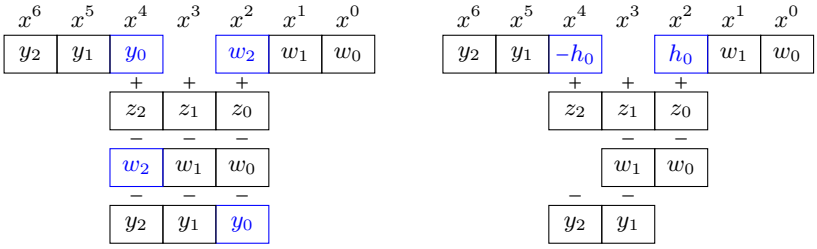
It hence computes the product as

$$a \cdot b = (a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + ((a_0 + a_1)(b_0 + b_1) - a_1b_1 - a_0b_0)x + a_0b_0.$$

This now requires five coefficient products. However, a_0b_0 and a_1b_1 are used twice, consequently, we only need to compute three coefficient multiplications. Despite the savings in multiplications, we now also require more additions. In total, four additions are needed.

This approach can be easily generalized to arbitrary-degree polynomials. Consider $a = \sum_{i=0}^{n-1} a_i x^i$, $b = \sum_{i=0}^{n-1} b_i x^i$. We now set $t = x^{\lfloor \frac{n}{2} \rfloor}$, and rewrite $a = a^{(1)}t + a^{(0)}, b = b^{(1)}t + b^{(0)}$ where $a^{(0)}$ and $b^{(0)}$ are $\lfloor \frac{n}{2} \rfloor$-coefficient polynomials consisting of the lower half of the coefficients of a and b. Conversely, $a^{(1)}$ and $b^{(1)}$ are $\lceil \frac{n}{2} \rceil$-coefficient polynomials consisting of the upper half of the coefficients of a and b. We can then use Karatsuba, to compute $a \cdot b$ using $a^{(0)} \cdot b^{(0)}$, $a^{(1)} \cdot b^{(1)}$, and $(a^{(0)} + a^{(1)}) \cdot (b^{(0)} + b^{(1)})$. Note that the smaller multiplications are now $\lfloor \frac{n}{2} \rfloor$- or $\lceil \frac{n}{2} \rceil$-coefficient polynomial multiplications. We illustrate the Karatsuba trick for $n = 4$ in Figure 2.2a.

Recursive Karatsuba. For large-degree inputs, this approach can be applied recursively. We can break down an n-coefficient multiplication into three $n/2$-coefficient multiplications each of which gets broken down into three $n/4$-coefficient multiplications. This is called applying multiple layers of Karatsuba. Each additional layer introduces more additions while saving some multiplications. It is likely that it is not optimal to do this all the way down to 1-coefficient polynomials at which polynomial multiplication becomes trivial, but rather one wants to stop earlier and perform a schoolbook multiplication of small-degree polynomials.



(a) Karatsuba (b) Refined Karatsuba

Figure 2.2: Karatsuba multiplication and refined Karatsuba multiplication for polynomials $a = a_0 + a_1x + a_2x^2 + a_3x^3, b = b_0 + b_1x + b_2x^2 + b_3x^3$. Let $w = (a_0 + a_1x)(b_0 + b_1x), y = (a_2 + a_3x)(b_2 + b_3x), z = ((a_0 + a_2) + (a_1 + a_3)x)((b_0 + b_2) + (b_1 + b_3)x)$. For refined Karatsuba, let $h = w_2 - y_0$.

Example 4: A common approach for multiplying 256-coefficient polynomials is to use four layers of Karatsuba and then switch to schoolbook multiplication. The schoolbook multiplications handle polynomials with $256/16 = 16$ coefficients and we need a total of $3^4 = 81$ of them.

Refined Karatsuba. Looking at Figure 2.2a, one can see that the term $x_2 - y_0$ appears twice in our product. Once for coefficient $c^{(2)}$, and once for coefficient $c^{(4)}$ (as $y_0 - x_2$). We can exploit this fact and simply compute the subtraction once as $h = x_2 - y_0$. The result is what is referred to as refined Karatsuba [Ber01] and is shown in Figure 2.2b. Assuming inputs of n coefficients, this trick can save $(n/2) \cdot 2 - 1 = n - 1$ subtractions. Note, however, that this will only work if the negation of h can be computed for free. This can be achieved by computing $z_2 - h$ rather than $-h + z_2$.

Application: Chapter 5 describes how to use recursive refined Karatsuba to implement efficient multiplications of polynomials of degree 16 to about 256 on the Arm Cortex-M4. Below degree 16 schoolbook multiplication is superior, above 256 Toom–Cook outperforms Karatsuba.

2.2.3 Toom–Cook Multiplication

The idea of splitting up input polynomials into smaller polynomials as done by Karatsuba's multiplication algorithm can be generalized to split into a larger number of polynomials. One such generalization is Toom–Cook multiplication [Too63, Coo66]. N-way Toom–Cook (or Toom-N for short) is splitting polynomials with n coefficients into N parts of n/N coefficients each. The underlying smaller polynomial multiplications will then process polynomials of n/N coefficients.

For example, Toom-3 splits inputs into three parts of $n/3$ coefficients. This is done in the following way: We substitute $y = x^{n/3}$ and write the polynomial a as $y^2a^{(2)} + ya^{(1)} + a^{(0)}$ and proceed similarly for the input b. The goal is now to compute the product

$$c = a \cdot b = y^4c^{(4)} + y^3c^{(3)} + y^2c^{(2)} + yc^{(1)} + c^{(0)}.$$

Toom–Cook does so by evaluating the polynomial a and b at certain values of y, then multiplying the smaller n/N polynomials, such that c can be recovered from the smaller products using interpolation. For Toom-3, this requires five values for y. A common choice is $y = \{0, 1, -1, -2, \infty\}$ with $a(\infty) = a^{(2)}$ Evaluating a and b yields

$$\begin{bmatrix} a(0) \\ a(1) \\ a(-1) \\ a(-2) \\ a(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a^{(0)} \\ a^{(1)} \\ a^{(2)} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b(0) \\ b(1) \\ b(-1) \\ b(-2) \\ b(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b^{(0)} \\ b^{(1)} \\ b^{(2)} \end{bmatrix}.$$

By *pointwise* multiplication, we obtain five points of $c = a \cdot b$ which is sufficient to recover the polynomial $y^4c^{(4)} + y^3c^{(3)} + y^2c^{(2)} + yc^{(1)} + c^{(0)}$.

Given that

$$\begin{bmatrix} a(0) \cdot b(0) \\ a(1) \cdot b(1) \\ a(-1) \cdot b(-1) \\ a(-2) \cdot b(-2) \\ a(\infty) \cdot b(\infty) \end{bmatrix} = \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(-2) \\ c(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \end{bmatrix}$$

we can obtain $c^{(0)}, c^{1)}, c^{(2)}, c^{(3)}$, and $c^{(4)}$ by inverting the matrix as

$$\begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1/2 & 1/3 & -1 & 1/6 & -2 \\ -1 & 1/2 & 1/2 & 0 & -1 \\ -1/2 & 1/6 & 1/2 & -1/6 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(-2) \\ c(\infty) \end{bmatrix}.$$

This immediately reveals a disadvantage of Toom multiplication. In the case of Toom-3, one needs to be able to divide by 3 and 2. As we are commonly working in a finite ring $\mathbb{Z}_q$, this presents a challenge. In general, there are two ways this can be achieved: If working in $\mathbb{Z}_q$ with q co-prime to 2 and 3, one can simply multiply by the inverses of 2 and 3. However, if q is not co-prime to either 2 or 3, this is no longer possible. To counter this issue, we instead need to do all computations modulo a larger q', such that we can be sure that whenever we need to do divisions, the remainder will always

Algorithm 1 Toom-3 evaluation and interpolation sequence

Input: $a = a^{(0)} + a^{(1)}y + a^{(2)}y^2, \quad b = b^{(0)} + b^{(1)}y + b^{(2)}y^2$
Output: $c = a \cdot b = c^{(0)} + c^{(1)}y + c^{(2)}y^2 + c^{(3)}y^3 + c^{(4)}y^4$

1: $a(0) \leftarrow a^{(0)} \quad b(0) \leftarrow b^{(0)} \quad \triangleright$ Evaluate a and b at $y_i = \{0, 1, -1, -2, \infty\}$
2: $t \leftarrow a^{(0)} + a^{(2)} \quad a(1) \leftarrow t + a^{(1)} \quad a(-1) \leftarrow t - a^{(1)}$
3: $t \leftarrow b^{(0)} + b^{(2)} \quad b(1) \leftarrow t + b^{(1)} \quad b(-1) \leftarrow t - b^{(1)}$
4: $a(-2) \leftarrow a^{(0)} - 2a^{(1)} + 4a^{(2)} \quad b(-2) \leftarrow b^{(0)} - 2b^{(1)} + 4b^{(2)}$
5: $a(\infty) \leftarrow a^{(2)} \quad b(\infty) \leftarrow b^{(2)}$
6: ... $\quad \triangleright$ Perform small mults to obtain $c(0)$, $c(\infty)$, $c(1)$, $c(-1)$, $c(-2)$
7: $c^{(0)} \leftarrow c(0) \quad c^{(4)} \leftarrow c(\infty)$ $\quad \triangleright$ Interpolate c
8: $t_1 \leftarrow (c(-2) - c(1))/3$ $\quad \triangleright\ -c^{(1)} + c^{(2)} - 3c^{(3)} + 5c^{(4)}$
9: $t_2 \leftarrow (c(1) - c(-1))/2$ $\quad \triangleright\ c^{(1)} + c^{(3)}$
10: $t_3 \leftarrow c(-1) - c(0)$ $\quad \triangleright\ -c^{(1)} + c^{(2)} - c^{(3)} + c^{(4)}$
11: $c^{(3)} \leftarrow (t_3 - t_1)/2 + 2c^{(4)}$
12: $c^{(2)} \leftarrow t_3 + t_2 - c^{(4)}$
13: $c^{(1)} \leftarrow t_2 - c^{(3)}$

be 0. If $2 \mid q$, we need $q' \geq 2q$; if $3 \mid q$, we need $q' \geq 3q$; if $6 \mid q$, we need $q' \geq 6q$. Note that in the common case where q is a power of two, the inverse of 2 does not exist, and consequently, we need $q' = 2q$, i.e., an additional bit is needed. In the literature, this is sometimes referenced to Toom–Cook losing bits of precision. In practice, this places constraints on the moduli that can be supported by our multiplication routine. For example, if we would like to use 16-bit coefficients, our Toom-3 implementation only supports $q \leq 2^{15}$.

Note that the performance of Toom–Cook is in large parts determined by the efficiency of the evaluation and interpolation stages. A straightforward implementation of each row of the Toom evaluation matrix and interpolation matrix will not yield a competitive implementation. Algorithm 1 outlines a more efficient evaluation and interpolation sequence.

Toom-4. Similarly, the Toom–Cook can be used to split into more and smaller parts. For example, Toom-4 uses

$$a = y^3a^{(3)} + y^2a^{(2)} + ya^{(1)} + a^{(0)}.$$

It is not hard to see that this requires to evaluate both arguments at seven points as the resulting product contains terms up to y^6:

$$a \cdot b = c = y^6a^{(6)} + y^5a^{(5)} + y^4a^{(4)} + y^3a^{(3)} + +y^2a^{(2)} + ya^{(1)} + c^{(0)}.$$

Common evaluation points for Toom-4 are $y = \{0, 1, -1, 2, -2, 3, \infty\}$, i.e.,

$$\begin{bmatrix} a(0) \\ a(1) \\ a(-1) \\ a(2) \\ a(-2) \\ a(3) \\ a(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 1 & 3 & 9 & 27 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a^{(0)} \\ a^{(1)} \\ a^{(2)} \\ a^{(3)} \end{bmatrix}.$$

After multiplying $a(y_i)$ with $b(y_i)$, we have

$$\begin{bmatrix} a(0) \cdot b(0) \\ a(1) \cdot b(1) \\ a(-1) \cdot b(-1) \\ a(2) \cdot b(2) \\ a(-2) \cdot b(-2) \\ a(3) \cdot b(3) \\ a(\infty) \cdot b(\infty) \end{bmatrix} = \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(2) \\ c(-2) \\ c(3) \\ c(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 \\ 0 & 0 & 0 & 0 & & 0 & 1 \end{bmatrix} \begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \\ c^{(5)} \\ c^{(6)} \end{bmatrix}.$$

By inverting the matrix, we obtain the following for interpolation:

$$\begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \\ c^{(5)} \\ c^{(6)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/3 & 1 & -1/2 & -1/4 & 1/20 & 1/30 & -12 \\ -5/4 & 2/3 & 2/3 & -1/24 & -1/24 & 0 & 4 \\ 5/12 & -7/12 & -1/24 & 7/24 & -1/24 & -1/24 & 15 \\ 1/4 & -1/6 & -1/6 & 1/24 & 1/24 & 0 & -5 \\ -1/12 & 1/12 & 1/24 & -1/24 & -1/120 & 1/120 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(2) \\ c(-2) \\ c(3) \\ c(\infty) \end{bmatrix},$$

which requires a much longer evaluation and interpolation sequence. One which proved to yield good performance results in practice is shown in Algorithm 2. Note that we need to divide by $120 = 5 \cdot 3 \cdot 2^3$ and, consequently, either q needs to be co-prime to 120, such that we can multiply by the inverses, or that we need to perform the smaller multiplications modulo $120q$ such that no wrap-around happens. If q is a power of two, it suffices to use $q' = 8 \cdot q$, i.e., use three extra bits.

Picking evaluation points. In previous sections, we have picked points y at which to evaluate the polynomials. Those points seemingly came out of thin air. Indeed, it is possible to pick other points and it is not immediately clear that these points are optimal. It appears natural to always use 0 and ∞ as those are cheap to evaluate at and interpolate from. The other points are arbitrary, and one picks the ones that give the best performance. The points presented in this section are the ones most commonly used in the

Algorithm 2 Toom-4 interpolation sequence

Input: $a = a^{(0)} + a^{(1)}y + a^{(2)}y^2 + a^{(3)}y^3, \quad b = b^{(0)} + b^{(1)}y + b^{(2)}y^2 + b^{(3)}y^3$
Output: $c = a \cdot b = c^{(0)} + c^{(1)}y + c^{(2)}y^2 + c^{(3)}y^3 + c^{(4)}y^4 + c^{(5)}y^5 + c^{(6)}y^6$

1: ... ▷ Evaluate a and b at $y = \{0, 1, -1, 2, -2, 3, \infty\}$
2: ... ▷ Small multiplications → $c(0)$, $c(\infty)$, $c(1)$, $c(-1)$, $c(2)$, $c(-2)$, $c(3)$
3: $c^{(0)} \leftarrow c(0) \quad c^{(6)} \leftarrow c(\infty)$ ▷ Interpolate c
4: $t_0 \leftarrow (c(1) + c(-1))/2 - c^{(0)} - c^{(6)}$ ▷ $c^{(2)} + c^{(4)}$
5: $t_1 \leftarrow (c(2) + c(-2) - 2c^{(0)} - 128c^{(6)})/8$ ▷ $c^{(2)} + 4c^{(4)}$
6: $c^{(4)} \leftarrow (t_1 - t_0)/3 \quad c^{(2)} \leftarrow (t_0 - c^{(4)})$
7: $t_0 \leftarrow (c(1) - c(-1))/2$ ▷ $c^{(1)} + c^{(3)} + c^{(5)}$
8: $t_1 \leftarrow ((c(2) - c(-2))/4 - t_0)/3$ ▷ $c^{(3)} + 5c^{(5)}$
9: $t_2 \leftarrow (c(3) - c^{(0)} - 9c^{(2)} - 81c^{(4)} - 729c^{(6)})/3$ ▷ $c^{(1)} + 9c^{(3)} + 81c^{(5)}$
10: $t_2 \leftarrow (t_2 - t_0)/8 - t_1$ ▷ $5c^{(5)}$
11: $c^{(5)} \leftarrow t_2/5 \quad c^{(3)} \leftarrow t_1 - t_2 \quad c^{(1)} \leftarrow t_0 - c^{(3)} - c^{(5)}$

literature in the context of lattice-based cryptography, but other choices are possible.

Finding the best combination of Toom-N and Karatsuba. Since Toom–Cook, as well as Karatsuba, can be applied recursively, there are a large number of combinations that can be constructed. The actual optimal choice depends on the target platform and polynomials to be multiplied. The modulus q introduces the largest restrictions. In case it is prime, many more combinations are possible, while for even moduli, higher order Toom–Cook and many combinations are not possible due to the requirements of using a larger q in the smaller multiplications. For example, if smaller multiplications are done using 16-bit multiplications, q must be at most 2^{15} ($2 \cdot q \leq 2^{16}$) for Toom-3, and at most 2^{13} ($8 \cdot q \leq 2^{16}$) for Toom-4. Applying a combination of Toom-4 and Toom-3 would require $8 \cdot 2 \cdot q = 16 \cdot q \leq 2^{16}$, i.e., $q \leq 2^{12}$.

Application: In Chapter 5, Toom-3 and Toom-4 are used in combination with Karatsuba to multiply polynomials of degree 256 and above on the Arm Cortex-M4.

2.2.4 Number-Theoretic Transform (NTT)

The number-theoretic transform can be seen as a discrete Fourier transform in a finite ring. The general idea is to perform (polynomial) multiplications by transforming both factors to a different domain in which multiplications are cheap. In the case of discrete Fourier transforms, this domain is called the frequency domain, while for NTTs, we refer to it as the *NTT domain*. Elements in the *NTT domain* are commonly identified by a hat, e.g., $\hat{a}$. The

product is then transformed back to the *normal domain* which is also called the *time domain* using the inverse NTT (NTT^{-1}).

Given two polynomials a, b, one computes their product as

$$c = a \cdot b = \text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$$

with $\circ$ referring to the multiplication in the NTT domain. In the most straightforward instantiation of NTTs, $\circ$ simply means coefficient-wise multiplication modulo q.

For simplicity, we assume that q is a prime number as this makes the explanation easier. However, composite-modulus NTTs are possible and are being used. For understanding NTTs, there is an essential mathematical entity that we need to understand: primitive roots of unity in $\mathbb{Z}_q$ [Nus82, Sec. 2.1.3]. An element $a \in \mathbb{Z}_q$ is a k-th (with $k \in \mathbb{N}$) root of unity if $a^k \equiv 1 \mod q$. It is a primitive root of unity, if there is no $k' < k$, s.t. $a^{k'} \equiv 1 \mod q$. A primitive k-th root of unity exists only if $k \mid (q-1)$ (or more generally, it only exists if k divides the order of $\mathbb{Z}_q^*$.) Throughout this section, ω_k denotes a primitive k-th root of unity.

Example 5: 1 and $-1 \equiv q-1 \mod q$ are 2-nd roots of unity. However, only -1 is a primitive 2-nd root of unity.
For $\mathbb{Z}_7$, there are six 6-th roots of unity ($\{1,2,3,4,5,6\}$), but only $\{3,5\}$ are primitive 6-th roots of unity.

When computing an NTT, one is evaluating a polynomial at powers of a primitive root of unity ω_k: $\omega_k^0, \omega_k^1, \ldots, \omega_k^{n-1}$.

The NTT [Nus82, Sec. 8.1] is defined as

$$\hat{a} = \sum_{i=0}^{n-1} \hat{a}_i x^i \quad \text{with} \quad \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega_k^{i \cdot j}.$$

Transforming $\hat{a}$ to the normal domain is achieved by computing NTT^{-1} as

$$a = \sum_{i=0}^{n-1} a_i x^i \quad \text{with} \quad a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega_k^{-i \cdot j}.$$

Example 6: From the definition of the NTT, it is not immediately obvious why it works. Consider the following example: We want to multiply $a = a_1x + a_0$ by $b = b_1x + b_0$ with coefficients in $\mathbb{Z}_7$. We know that the result will be of degree at most 2, therefore, we have to evaluate at 3 points. To compute the NTT, we require a 3-rd root of unity, e.g., $\omega_3 = 2$. We now evaluate $a(x)$ and $b(x)$ at $x = \{\omega_3^0, \omega_3^1, \omega_3^2\}$:

$$\hat{a} = \begin{bmatrix} a(\omega_3^0) \\ a(\omega_3^1) \\ a(\omega_3^2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_3 \\ 1 & \omega_3^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} b(\omega_3^0) \\ b(\omega_3^1) \\ b(\omega_3^2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_3 \\ 1 & \omega_3^2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}.$$

A pointwise multiplication yields

$$\hat{a} \circ \hat{b} = \begin{bmatrix} c(\omega_3^0) \\ c(\omega_3^1) \\ c(\omega_3^2) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_0b_1 + a_1b_0 + a_1b_1 \\ a_0b_0 + \omega_3 a_0b_1 + \omega_3 a_1b_0 + \omega_3^2 a_1b_1 \\ a_0b_0 + \omega_3^2 a_0b_1 + \omega_3^2 a_1b_0 + \omega_3 a_1b_1 \end{bmatrix}.$$

Computing NTT^{-1} corresponds to the interpolation of $c(x)$ from $c(\omega_3^0)$, $c(\omega_3^1)$, and $c(\omega_3^2)$. For that we require the inverse of ω_3 which in our example is $\omega_3^{-1} = 4$.
It now holds that,

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = 3^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3^{-1} & \omega_3^{-2} \\ 1 & \omega_3^{-2} & \omega_3^{-1} \end{bmatrix} \begin{bmatrix} c(\omega_3^0) \\ c(\omega_3^1) \\ c(\omega_3^2) \end{bmatrix} = \begin{bmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{bmatrix}.$$

It is important to note here that this all works out because $1 + \omega_3 + \omega_3^2 \equiv 0 \mod q$. It is not hard to see that this can be generalized to any size of polynomials as long as the appropriate primitive root of unit exists.

In a normal polynomial multiplication for n-coefficient factors, it is sufficient to evaluate the polynomials at $2n - 1$ points. However, in real implementations, this is never how it is done because working with polynomials of odd length is very disadvantageous for fast implementations. Hence, one would use $2n$ points or round up to the closest power of two.

Finding a root of unity. In the previous examples, we have pulled the primitive root of unity out of thin air. The easiest way to find one is to try all field elements and check if they are indeed a k-th primitive root of unity.

Example 7: Considering $\mathbb{Z}_{17}$, we can check the multiplicative order of each element, and find that 16 is a primitive 2-nd root of unity, $\{4, 13\}$ are 4-th roots of unity, $\{2, 8, 9, 15\}$ are 8-th roots of unity, and $\{3, 5, 6, 7, 10, 11, 12, 14\}$ are 16-th roots of unity. If one needs a 16-th root of unity, any of the eight possible values will work and result in a correct NTT. Usually, the root is arbitrarily picked as the smallest available.

Cyclic NTT

One big advantage of NTT-based multiplication is that it naturally supports convolutions. In the case of cyclic convolutions (modulo $x^n - 1$), this is possible using a cyclic NTT which is exactly the NTT that was introduced in the previous section. However, one only needs to evaluate at n roots of unity. Why this works can be easily seen in the following example. For a more formal proof of the correctness, see [Nus82, Theorem 8.1].

Example 8: Consider the polynomial ring $\mathbb{Z}_q[x]/(x^2 - 1)$, and assume there is a 2-nd root of unity ω_2, i.e., $\omega_2^2 \equiv 1 \mod q$. The polynomials $a = a_1x + b_0$, $b = b_1x + b_0$, can be transformed to the NTT domain by evaluating the polynomials at ω_2^0, ω_2^1:

$$\hat{a} = \begin{bmatrix} a(\omega_2^0) \\ a(\omega_2^1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} b(\omega_2^0) \\ b(\omega_2^1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}.$$

A pointwise multiplication yields

$$\hat{a} \circ \hat{b} = \begin{bmatrix} c(\omega_2^0) \\ c(\omega_2^1) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_0b_1 + a_1b_0 + a_1b_1 \\ a_0b_0 + \omega_2 a_0b_1 + \omega_2 a_1b_0 + a_1b_1 \end{bmatrix}.$$

When we apply the NTT^{-1} to that pointwise product we obtain

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = 2^{-1} \begin{bmatrix} 1 & 1 \\ 1 & \omega_2^{-1} \end{bmatrix} \begin{bmatrix} c(1) \\ c(\omega_2) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_1b_1 \\ a_0b_1 + a_1b_0 \end{bmatrix},$$

which coincides with polynomial multiplication in $\mathbb{Z}_q[x]/(x^n - 1)$.

Negacyclic NTT.

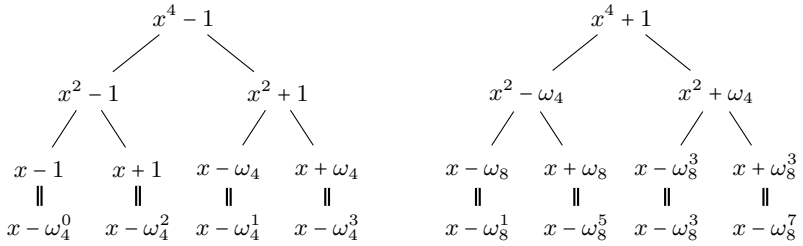
Similarly, we would like to multiply polynomials in $\mathbb{Z}_q[x]/(x^n + 1)$. This can be achieved using the negacyclic NTT [SS71]. Given a $2n$-th root of unity ω_{2n} (i.e., $\omega_{2n}^2 = \omega_n$), it is defined as

$$\hat{a} = \sum_{i=0}^{n-1} \hat{a}_i x^i \quad \text{with} \quad \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega_{2n}^j \omega_n^{i \cdot j}.$$

Its inverse is defined as

$$a = \sum_{i=0}^{n-1} a_i x^i \quad \text{with} \quad a_i = n^{-1} \omega_{2n}^{-i} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-i \cdot j}.$$

Note that the negacyclic NTT is the same as multiplying each input coefficient a_i by ω_{2n}^i and then applying the NTT as defined in the previous section. The multiplication by the powers of roots of unity is called twisting [Ber01]. The inverse also works similarly, but the output is twisted back by multiplying by powers of ω_{2n}^{-1}.



(a) Cyclic NTT for $\mathbb{Z}_q[x]/(x^n - 1)$ (b) Negacyclic NTT for $\mathbb{Z}_q[x]/(x^n + 1)$

Figure 2.3: Splitting polynomial rings using the NTT.

Example 9: Let our input polynomials be $a = \sum_{i=0}^{n-1} a_i x^i$ and $b = \sum_{i=0}^{n-1} b_i x^i$. We twist both polynomials by multiplying by powers of ω_{2n} to obtain $a' = \sum_{i=0}^{n-1} a_i \omega_{2n}^i x^i$ and $b' = \sum_{i=0}^{n-1} b_i \omega_{2n}^i x^i$. Now, we perform a regular polynomial multiplication (e.g., using schoolbook multiplication) to obtain $c' = \sum_{i=0}^{2n-1} c'_i x^i \omega_{2n}^i$. Since $\omega_{2n}^n \equiv -1 \mod q$, we can rewrite this as $c' = \sum_{i=0}^{n-1} c'_i x^i \omega_{2n}^i - \sum_{i=n}^{2n-1} c'_i x^i \omega_{2n}^{i-n}$. When this is convoluted modulo $x^n - 1$ and twisted back (i.e., the powers of ω_{2n} being removed), we obtain the correct product of a and b modulo $x^n + 1$. Note that the regular polynomial multiplication and reduction modulo $x^n - 1$ can be replaced by an implementation using a cyclic NTT.

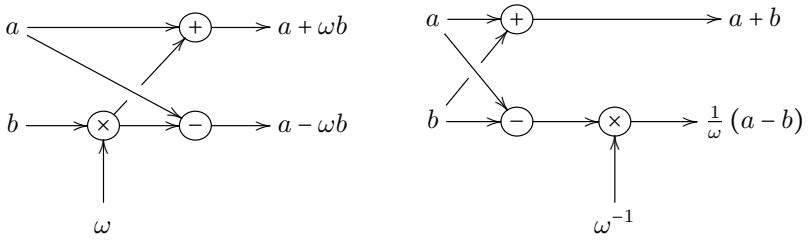
Changing perspective: Chinese Remainder Theorem (CRT)

There is another way to think about what an NTT is, which may appear more natural or elegant: The polynomial $x^{2n} - 1$ can be split into two factors: $x^n - 1$ and $x^n + 1$. That means that a polynomial a in $\mathbb{Z}_q[x]/(x^{2n} - 1)$ can be split into two polynomials, a' in $\mathbb{Z}_q[x]/(x^n - 1)$ and a'' in $\mathbb{Z}_q[x]/(x^n + 1)$ by simply reducing modulo $x^n - 1$ and modulo $x^n + 1$. From the two smaller polynomials, one can recover the original polynomial using the Chinese Remainder Theorem (CRT), i.e.,

$$a = \sum_{i=0}^{n-1} \frac{1}{2}(a'_i + a''_i)x^i + \sum_{i=0}^{n-1} \frac{1}{2}(a'_i - a''_i)x^{n+i}.$$

Example 10: Let $a = a_0 + a_1 x + a_2 x^2 + a_3 x^3$, we can then compute $a' = (a_0 + a_2) + (a_1 + a_3)x$ and $a'' = (a_0 - a_2) + (a_1 - a_3)x$. To recover the original a, we compute $a = \frac{1}{2}\big((a'_0 + a''_0) + (a'_1 + a''_1)x + (a'_0 - a''_0)x^2 + (a'_1 - a''_1)x^3\big)$.

For even n, one can apply this trick again for $x^n - 1$ since $x^n - 1 = (x^{n/2} - 1)(x^{n/2} + 1)$. However, we can also do the same for $x^n + 1$ in case there is an appropriate root of unity $\omega_4 = \sqrt{-1}$, s.t. $x^n + 1 = (x^{n/2} - \omega_4)(x^{n/2} + \omega_4)$.



(a) Cooley–Tukey Butterfly [CT65] (b) Gentleman–Sande Butterfly [GS66]

Figure 2.4: The "Butterflies" of Fast Fourier Transforms

Applying this trick recursively, it is easy to see that, we can split $x^n - 1$ into n linear parts in case there exists an n-th root of unity ω_n:

$$x^n - 1 = \prod_{i=0}^{n-1}(x - \omega_n^i).$$

This splitting is illustrated in Figure 2.3a.

Similarly, for $x^n + 1$ we can fully split into linear terms in case a $2n$-th root of unity ω_{2n} exists as illustrated in Figure 2.3b:

$$x^n + 1 = \prod_{i=0}^{n-1}(x - \omega_{2n}\omega_n^i) = \prod_{i=0}^{n-1}(x - \omega_{2n}^{2i+1}).$$

Application: Chapter 3, Chapter 4, and Chapter 6 cover how to efficiently implement NTTs for Kyber, Dilithium, Saber, NTRU, and LAC. On the Cortex-M4, NTTs are superior to Toom and Karatsuba multiplication for these schemes.

2.2.5 Algorithms for Computing NTTs

Using NTTs for fast arithmetic is only beneficial if the transformations themselves can be implemented efficiently. From the previous section, it is not obvious that these can be implemented efficiently. A straightforward implementation of the transformation requires n^2 multiplications and is, therefore, not at all better than just using the schoolbook method for the multiplication itself. However, there are much faster algorithms that allow computing the NTT in quasi-linear time $\mathcal{O}(n \log n)$. These algorithms are called Fast Fourier Transform (FFT) algorithms. The two most prominent ones are by Cooley–Tukey (CT) [CT65] and Gentleman–Sande (GS) [GS66]. The CT FFT algorithm is also referred to as decimation in time (DIT) FFT, while the GS FFT algorithm is sometimes called decimation in frequency (DIF) FFT. It is common practice to implement the forward NTT using the CT

Algorithm 3 CT FFT implementing forward NTT

Input: Polynomial $a \in \mathbb{Z}_q/(x^{2^k} \pm 1)$; time domain, normal order
Input: Root of unity: ω_{2^k} for $\mathbb{Z}_q/(x^{2^k} - 1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k} + 1)$
Output: $\hat{a}$; NTT domain, bit-reversed order

1: **for** $l = k-1; l \geq 0; l \leftarrow l-1$ **do**
2: **for** $i = 0; i < 2^{k-1-l}; i \leftarrow i+1$ **do**
3: $\psi \leftarrow \begin{cases} \omega_{2^k}^{\mathtt{brv}_{k-1}(i)} \text{ for } \mathbb{Z}_q/(x^{2^k} - 1) \\ \omega_{2^{k+1}}^{\mathtt{brv}_k(2^{k-1-l}+i)} \text{ for } \mathbb{Z}_q/(x^{2^k} + 1) \end{cases}$
4: **for** $j = i \cdot 2^{l+1}; j < i \cdot 2^{l+1} + 2^l; j \leftarrow j+1$ **do**
5: $t_0 \leftarrow a_j$
6: $t_1 \leftarrow \psi \cdot a_{j+2^l}$
7: $a_j \leftarrow t_0 + t_1$
8: $a_{j+2^l} \leftarrow t_0 - t_1$
9: **end for**
10: **end for**
11: **end for**

FFT, and the NTT^{-1} using the GS FFT. However, both transforms can be implemented using either of the two, and it highly depends on the target platform and parameters which one is best.

FFT algorithms are characterized by their *butterfly* operations. Figure 2.4a and Figure 2.4b show the *butterfly* of the CT and GS FFT algorithm respectively. The CT does straightforwardly implement the split of $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ into $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$. The multiplicand c is a constant and is called the twiddle factor. Similarly, the Gentleman–Sande butterfly computes the CRT of elements in $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$ yielding an element in $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ using the twiddle factor c^{-1}. We can also express this as a ring isomorphism $\phi : \mathbb{Z}_q[x]/(f(x)g(x)) \cong \mathbb{Z}_q[x]/(f(x)) \times \mathbb{Z}_q[x]/(g(x))$, $\phi(h) = (h \bmod f, h \bmod g)$. When $f(x) = x^n - c$ and $g(x) = x^n + c$, ϕ naturally becomes

$$\phi\left(\sum_{i=0}^{2n-1} h_i x^i\right) = \left(\sum_{i=0}^{n-1} \underbrace{(h_i + c h_{n+i})}_{CT} x^i, \sum_{i=0}^{n-1} \underbrace{(h_i - c h_{n+i})}_{CT} x^i\right).$$

The inverse computation can be expressed as ϕ^{-1}:

$$\phi^{-1}\left(\left(\sum_{i=0}^{n-1} h'_i x^i\right), \left(\sum_{i=0}^{n-1} h''_i x^i\right)\right) = \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{(h'_i + h''_i)}_{GS} x^i + \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{\frac{1}{c} (h'_i - h''_i)}_{GS} x^{n+i}.$$

Algorithm 4 GS FFT implementing NTT^{-1}

Input: $\hat{a}$; NTT domain, bit-reversed order
Input: Root of unity: ω_{2^k} for $\mathbb{Z}_q/(x^{2^k}-1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k}+1)$
Output: Polynomial $a \in \mathbb{Z}_q/(x^{2^k} \pm 1)$; time domain, normal order

1: **for** $l = 0; l < k; l \leftarrow l+1$ **do**
2: **for** $i = 0; i < 2^{k-1-l}; i \leftarrow i+1$ **do**
3: $\psi \leftarrow \begin{cases} \omega_{2^k}^{-\texttt{brv}_{k-1}(i)} \text{ for } \mathbb{Z}_q/(x^{2^k}-1) \\ \omega_{2^k}^{-\texttt{brv}_k(2^{k-1-l}+i)} \text{ for } \mathbb{Z}_q/(x^{2^k}+1) \end{cases}$
4: **for** $j = i \cdot 2^{l+1}; j < i \cdot 2^{l+1} + 2^l; j \leftarrow j+1$ **do**
5: $t_0 \leftarrow a_j + a_{j+2^l}$
6: $t_1 \leftarrow a_j - a_{j+2^l}$
7: $a_j \leftarrow t_0$
8: $a_{j+2^l} \leftarrow \psi \cdot t_1$
9: **end for**
10: **end for**
11: **end for**
12: **for** $i = 0; i < 2^k; i \leftarrow i+1$ **do**
13: $a_i \leftarrow n^{-1} a_i$ ▷ Cancel out factor of 2 for each butterfly
14: **end for**

Algorithm 3 and Algorithm 4 show how these butterflies can be applied iteratively to larger polynomials to obtain a full NTT and NTT^{-1} in-place. FFT algorithms have the peculiar property that their outputs are in a different order than one might expect: They are in so-called *bit-reversed* order. Bit-reversing an array of length 2^k means interpreting the index of each element as a binary string of length k and reversing the binary string. For an index i with i_j denoting the j-th bit this can be expressed as

$$i = \sum_{j=0}^{k-1} i_j 2^j, \quad \texttt{brv}_k(i) = \sum_{j=0}^{k-1} i_{k-j} 2^j.$$

The reversed index becomes the new index of each element. The operation can be reversed by applying the same transformation again.

Example 11: Given an array of length eight in normal order: $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$.
We write the indices in binary: $[a_{000}, a_{001}, a_{010}, a_{011}, a_{100}, a_{101}, a_{110}, a_{111}]$.
Then, reverse the binary digits: $[a_{000}, a_{100}, a_{010}, a_{110}, a_{001}, a_{101}, a_{011}, a_{111}]$.
This gives us the array in bit-reversed order: $[a_0, a_4, a_2, a_5, a_1, a_5, a_3, a_7]$.

Since bit-reversing is a costly operation in software, one tries to avoid explicitly performing it. Hence, whenever possible one will keep values in

```
# computing twiddles for a cyclic NTT Z_q[x](x^n - 1)
twiddlesNtt    = brv([pow(root, i, q) for i in range(n//2)])
twiddlesInvNtt = brv([pow(root, -i, q) for i in range(n//2)])

# computing twiddles twiddles for a negacyclic NTT Z_q[x](x^n + 1)
twiddlesNtt = brv([pow(root, i, q) for i in range(n)])[1:n]
twiddlesInvNtt = brv([pow(root, -(i+1), q) for i in range(n)])[0:n-1]
```

Listing 2.1: Python code for generating twiddle factors

the NTT domain in bit-reversed order. Since the NTT^{-1} restores the normal order, this has no impact if polynomial multiplication is implemented. Since multiplication and addition in NTT are element-wise operations, the order does not matter.

Looking at Algorithm 3 and Algorithm 4 one also notices that one requires the twiddle factors with the exponents being bit-reversed as well. Note that this makes it impractical to compute the twiddle factors on the fly. Virtually all fast software implementations resolve this by pre-computing the needed powers of the root of unity and storing them in memory. Those can easily be derived from the tree representations of an FFT, e.g., in Figure 2.3. Splitting $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ into $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$ uses the twiddle factor c, while the inverse uses c^{-1}. By traversing the tree breadth-first, we obtain [1, 1, w_4] in the cyclic case, and [ω_8^2, ω_8, ω_8^3] in the negacyclic case.

The Python code in Listing 2.1 generalizes this manual derivation given a function `brv` which reorders an array in bit-reversed order.

Note that in the cyclic case, the twiddles repeat, i.e., each layer uses the first 2^{k-1-l} twiddle factors from the pre-computed array, while in the negacyclic case, each twiddle is only used once due to the required twisting.

Discrete Fourier Transform (DFT) matrices. Another way of thinking about the FFT is by thinking about DFT matrices. A DFT matrix is defined as

$$W_n = \left(\omega_n^{jk}\right)_{j,k=0,\ldots,n-1}$$

Given this DFT matrix, a cyclic NTT transformation is multiplying the coefficient vector of the polynomial by the DFT matrix. The NTT^{-1} consists of multiplying by the inverse of the DFT matrix.

Example 12: Let $n = 4$. The corresponding DFT matrix using a 4-th root of unity ω_4 is

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix}.$$

The cyclic NTT of a polynomial $a(x) \in \mathbb{Z}_q[x]/(x^4 - 1)$ can be written as

$$\hat{a} = W \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

This can be decomposed into four Cooley–Tukey butterflies, i.e.,

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & \omega_4 \\ 0 & 0 & 1 & -\omega_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix}.$$

Note that the second and third row are swapped, i.e., the output is in bit-reversed order.

The negacyclic NTT can be expressed similarly by considering it as twisting followed by a cyclic NTT. We write $W'_n = W_n \cdot \mathtt{diag}(w_{2n}^i)_{i=0,\ldots,n-1}$, where $\mathtt{diag}$ is a diagonal matrix containing the elements $\{w_{2n}^0, \ldots, w_{2n}^{n-1}\}$.

Example 13: Let $n = 4$. The negacyclic NTT of a polynomial $a(x) \in \mathbb{Z}_q[x]/(x^4 + 1)$ with a $2n$-th root of unity ω_{2n} ($\omega_{2n}^2 = \omega_4$) is

$$\hat{a} = W_4 \cdot \mathtt{diag}(w_{2n}^i)_{i=0,\ldots,n-1} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & \omega_8 & \omega_8^2 & \omega_8^3 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

When again accepting that outputs are in bit-reversed order, we can use 4 CT butterflies to implement the negacyclic NTT:

$$\begin{bmatrix} 1 & \omega_8 & 0 & 0 \\ 1 & -\omega_8 & 0 & 0 \\ 0 & 0 & 1 & \omega_8^3 \\ 0 & 0 & 1 & -\omega_8^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & \omega_8^2 & 0 \\ 0 & 1 & 0 & \omega_8^2 \\ 1 & 0 & -\omega_8^2 & 0 \\ 0 & 1 & 0 & -\omega_8^2 \end{bmatrix} = \begin{bmatrix} 1 & \omega_8 & \omega_8^2 & \omega_8^3 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 \end{bmatrix}.$$

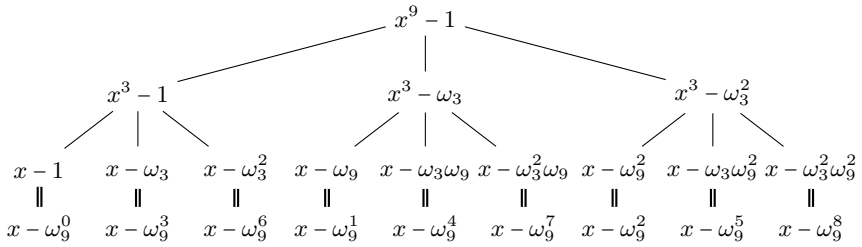


Figure 2.5: Radix-3 FFT for $\mathbb{Z}[x]/(x^9-1)$

2.2.6 Radix-3 FFT and Mixed-Radix FFT

The previous section exclusively covered FFTs working on polynomial rings of the form $\mathbb{Z}[x]/(x^{2^k}\pm 1)$. These are called radix-2 FFTs as each NTT layer splits into two parts. However, we can also use FFTs for a different radix. Most commonly used are radix-3 and radix-5 FFTs. They can be best studied by considering a single layer (i.e., split) of the FFT.

A radix-3 FFT splits the ring $\mathbb{Z}[x]/(x^{3^{k+1}}-c^3)$ as

$$\begin{aligned}&\mathbb{Z}[x]/(x^{3^{k+1}}-c^3)\\ \to\ &\mathbb{Z}[x]/(x^{3^k}-c)\times\mathbb{Z}[x]/(x^{3^k}-\omega_3 c)\times\mathbb{Z}[x]/(x^{3^k}-\omega_3^2 c).\end{aligned}$$

A radix-3 Cooley–Tukey-style butterfly splitting $\mathbb{Z}[x]/(x^{3^k}\pm c^3)$ for $0\le i<3^k$ can be implemented as

$$\begin{aligned}\hat{a}_i &= a_i &+&\quad c a_{i+3^k} &+&\quad c^2 a_{i+2\cdot 3^k},\\ \hat{a}_{i+3^k} &= a_i &+&\quad \omega_3 c a_{i+3^k} &+&\quad \omega_3^2 c^2 a_{i+2\cdot 3^k},\\ \hat{a}_{i+2\cdot 3^k} &= a_i &+&\quad \omega_3^2 c a_{i+3^k} &+&\quad \omega_3 c^2 a_{i+2\cdot 3^k}.\end{aligned}$$

The inverse radix-3 butterfly using a Gentleman–Sande-style is

$$\begin{aligned}a_i &= 3^{-1}\quad (a_i &+&\quad a_{i+3^k} &+&\quad a_{i+2\cdot 3^k}),\\ a_{i+3^k} &= 3^{-1}c^{-1}(a_i &+&\quad \omega_3^2 a_{i+3^k} &+&\quad \omega_3 a_{i+2\cdot 3^k}),\\ a_{i+2\cdot 3^k} &= 3^{-1}c^{-2}(a_i &+&\quad \omega_3 a_{i+3^k} &+&\quad \omega_3^2 a_{i+2\cdot 3^k}).\end{aligned}$$

Example 14: Of course, the radix-3 FFT trick can be applied recursively. An FFT splitting $\mathbb{Z}_q[x]/(x^9-1)$ into linear factors can be seen in Figure 2.5. The twiddle factors (c, c^2) for the first layer are $(1,1)$. For the second layer they are $(1,1)$, (ω_9, ω_9^2), and (ω_9^2, ω_9) from left to right respectively. One may notice that the outputs are once again in a different order than they should be. However, this time the order is not bit-reversed, but the equivalent in base 3. As usual, we do not need to be concerned about the order if we are implementing polynomial multiplication as pointwise multiplication can simply operate on base-3-reversed inputs and the inverse FFT will restore the normal order. Base-3-reversing works similarly as bit-reversing: One represents each index in base 3 and reverts the order of the digits, i.e.,

$$(0,1,2,3,4,5,6,7,8) \to (00,01,02,10,11,12,20,21,22)$$
$$\to (00,10,20,01,11,21,02,12,22) \to (0,3,6,1,4,7,2,5,8).$$

Radix-5 FFT. The above can be easily extended to FFTs for an arbitrary radix. For post-quantum cryptography so far only radix-5 FFTs were used. One layer of a radix-5 FFT splits $\mathbb{Z}_q[x]/(x^{5^{k+1}}-c^5)$ into 5 elements in $\mathbb{Z}_q[x]/(x^{5^k}-{w_5}^i c)$ for $i = \{0,1,2,3,4\}$.

Mixed-radix FFTs. The FFTs for different radices can be combined. One could, for example, perform one layer of radix-2 butterflies followed by a layer of radix-3 butterflies to obtain a 6-FFT. This can be very useful for schemes that have not been specifically designed for the use of the NTT and, hence, often do not nicely split using radix-2 FFTs.

Application: Chapter 6 covers how to efficiently implement `ntruhps4096821` [ZCH+19] which requires polynomial multiplication in $\mathbb{Z}_{2048}/(x^{701}-1)$. This can be implemented using a cyclic NTT for $\mathbb{Z}_{3365569}/(x^{1536}-1)$, which can be efficiently implemented by using 9-layer radix-2 FFTs followed by a 1-layer radix-3 FFT.

2.2.7 Incomplete NTT

A standard radix-2 FTT is splitting a ring (e.g., $\mathbb{Z}_q[x]/(x^n-1)$) down into smaller rings down to having elements in $\mathbb{Z}_q[x]/(x-\omega_n^i)$ $(0 \le i < n)$, each consisting of only one coefficient. In one step of the FFT algorithm, a polynomial in $\mathbb{Z}_q[x]/(x^{2^k}-c^2)$ is split into one element in $\mathbb{Z}_q[x]/(x^{2^{k-1}}-c)$ and one element in $\mathbb{Z}_q[x]/(x^{2^{k-1}}+c)$. Those then get split into $\mathbb{Z}_q[x]/(x^{2^{k-2}}-\sqrt{c}), \mathbb{Z}_q[x]/(x^{2^{k-2}}+\sqrt{c}), \mathbb{Z}_q[x]/(x^{2^{k-1}}+\sqrt{-c}), \mathbb{Z}_q[x]/(x^{2^{k-1}}+\sqrt{-c})$ and so forth.

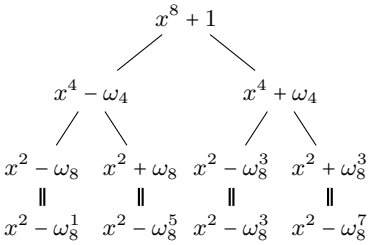


Figure 2.6: Incomplete FFT for $\mathbb{Z}[x]/(x^8+1)$

In this case, it is natural to consider to stop earlier, i.e., instead of doing $\log_2 n$ splits down to linear terms in $\mathbb{Z}_q[x]/(x-\omega_n^i)$ $(0 \le i < n)$, we could do $\log_2 n - 1$ splits and end up with elements in $\mathbb{Z}_q[x]/(x^2-\omega_{n/2}^i)$. This is referred to as an incomplete NTT in the literature.

It results in some clear advantages, the most prominent being that it results in fewer restrictions regarding the prime q. In the cyclic NTT example above, only a $n/2$-th root of unity is needed, while in the case of negacyclic NTTs, an n-th root of unity suffices. This gives much more choices for q. For example, Kyber [ABD+17] used $q = 7681$ in the initial submission together with a negacyclic complete NTT for $\mathbb{Z}_q[x]/(x^{256}+1)$. In the second round of the NISTPQC competition q was changed to 3329. Since no 512-th root of unity modulo 3329 exists, this was only made possible by switching to an incomplete (7-layer) NTT.

Note that the multiplication in the NTT domain is affected by this change as well. The coefficient-wise multiplication becomes multiplication in $\mathbb{Z}_q[x]/(x^2-\omega_n^i)$. This multiplication is usually referred to as base multiplication. Two elements $a, b \in \mathbb{Z}_q[x]/(x^2-\omega_n^i)$ $(a = a_1x + a_0, b = b_1x + b_0)$ can be multiplied by

$$c = (a_1x + a_0)(b_1x + b_0) = (a_0b_0 + a_1b_1\omega_n^i) + (a_0b_1 + a_1b_0)x,$$

where ω_n^i is different for each of the $n/2$ multiplications.

Why it is worth it. Besides leaving more freedom for the choice of parameters, polynomial multiplication using incomplete NTT is often faster than complete NTT. This can be easily seen by counting operations. One butterfly costs one multiplication and two additions/subtractions, i.e., $n/2$ multiplications and n additions for one layer of NTT. Since this is required for both operands and also the inverse transformation of the result, we end up needing $3 \cdot n/2$ multiplications and $3n$ additions per layer of NTTs. However, we also need to consider the cost of base multiplication: If polynomials are of degree 0, the base multiplication (pointwise multiplication) costs 1 multiplication and 0 additions. For 2-coefficient polynomials, this increases to five multiplications and two additions. Hence, by stopping one layer early,

we pay additional $4n/2$ multiplications and $2n/2$ additions, but save $3 \cdot n/2$ multiplications and $3n$ additions which results in a net saving of additions at the cost of additional multiplications. For platforms where multiplications are as cheap as additions, this results in a net speed-up.

How incomplete should it be? The optimal degree of incompleteness needs to be determined for each target platform separately. For example, on the Cortex-M4 it often makes sense [CHK+21] to stop two layers early as a 4×4 schoolbook implementation can be implemented rather efficiently using the available multiply-and-accumulate instructions. However, in case the NTT is being built into the specification of the cryptographic scheme (as it is the case for Kyber [ABD+17], and Dilithium [LDK+17]), the implementer is not given any choice.

Application: Kyber (since the second round of the NIST competition) is using incomplete NTTs to implement polynomial multiplication in $\mathbb{Z}_{3329}[x]/(x^{256}+1)$. Chapter 3 describes how to implement it efficiently on the Cortex-M4. Our implementations of Saber, NTRU, and LAC covered in Chapter 5 also make use of incomplete NTTs.

2.2.8 Good's Trick

Another trick that can be useful for polynomials that are not suitable for radix-2 FFTs, was proposed by Good [Goo51] which is referred to as *Good's trick* or *prime-factor FFT* in the literature. In the context of polynomials, it allows computing an FFT of a polynomial in $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ with p_0 and p_1 being co-prime. A common choice is p_1 being a power of two and p_0 being a small prime (e.g., 3 or 5).

Good's trick maps $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ to $\mathbb{Z}_q[y]/(y^{p_0}-1)[z]/(z^{p_1}-1)$ by setting $x = yz$, i.e., we present a polynomial in $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ as p_0 polynomials in $\mathbb{Z}_q[z]/(z^{p_1}-1)$.

The mapping (usually called Good's permutation) between the two isomorphic rings corresponds to a reshuffling of the coefficients, such that $x^i = y^{i_0}z^{i_1}$. This can be seen as transforming a 1-dimensional array of p_0p_1 coefficients into a 2-dimensional array with dimensions $p_0 \times p_1$.

The forward Good's permutation uses

$$i_0 = i \mod p_0 \text{ and } i_1 = i \mod p_1.$$

To compute the inverse, we use the fact that p_0 and p_1 are co-prime and apply the CRT to obtain i from (i_0, i_1):

$$i = (p_1^{-1} \bmod p_0)p_1i_0 + (p_0^{-1} \bmod p_1)p_0i_1.$$

Example 15: Let $p_0 = 3$ and $p_1 = 2$. Given a polynomial $a = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$, applying Good's permutation results in $a = (a_0 + a_3z) + (a_4 + a_1z)y + (a_2 + a_5z)y$.

When using Good's trick for polynomial multiplication of two polynomials a and b, one proceeds as follows

1. Apply Good's permutation to both a and b
2. Compute p_1-NTT for each of the p_0 components of both a and b
3. Compute p_0-NTT for each of the p_1 components of both a and b
4. Perform pointwise multiplication of the coefficients
5. Compute p_0-NTT^{-1}
6. Compute p_1-NTT^{-1}
7. Apply inverse Good's permutation

Steps (3) to (5) can alternatively be replaced by a schoolbook multiplication $\bmod (y^{p_0} - 1)$ which often turns out to be faster. To achieve competitive performance one will merge Good's permutation with the first layer of the NTT computation and, similarly, merge the inverse permutation with the last layer of the NTT^{-1} computation.

Application: Good's trick can be beneficial for polynomial multiplication of NTRU [ZCH+19]. We make use of it in Chapter 6.

2.3 Modular-Reduction Algorithms and Short Multiplications

Thus far, we have assumed arithmetic in $\mathbb{Z}_q$. However, depending on the choice of q, this presents a challenge as well. In PQC software implementations one will store coefficients as either 16-bit or 32-bit integers. Hence, the examples in this section are limited to these choices.

In the easiest case, q is a power of two, and reductions are naturally supported by CPUs as multiplications are implicitly modulo 2^{16} or 2^{32}. To obtain a value between 0 and $q-1$ one uses a logical `AND` with $q-1$. However, for schemes that do not have power-of-two q, we need a different way for reductions. The rest of this section will introduce ways of performing modular reduction for any odd q.

Application: Saber [DKRV17] and NTRU [ZCH+19] use a power-of-two modulus and, hence, allow cheap modular reduction (see Chapter 5) unless one decides to switch to a prime modulus for performance (see Chapter 6). Dilithium [LDK+17] (Chapter 4) and Kyber [ABD+17] (Chapter 3) use a prime modulus and, hence, special care needs to be taken for modular reduction.

Notation. For a reduction to the interval from 0 to $q-1$ we write $a \mod q$ and call the result the canonical unsigned representative of a. Similarly, a reduction to the interval from $\left\lfloor -\frac{q-1}{2} \right\rfloor$ to $\left\lfloor \frac{q-1}{2} \right\rfloor$ is written as $z \bmod^{\pm} q$ and we call the results the canonical signed representative of a.

Signed vs. unsigned representation. Common reduction algorithms exist in multiple variants depending on inputs and results being in signed or unsigned representation. For cryptographic schemes, the outputs (e.g., keys, ciphertexts) are commonly in unsigned representation, but it is often beneficial to use signed representation for intermediate values. For example, when subtracting unsigned values, one needs to take care that the result is non-negative by adding an appropriate multiple of q. This can be avoided by accepting outputs to be signed. Additionally, signed representation allows a particularly fast Montgomery reduction which is introduced in Section 2.3.2. As all implementations in this thesis are using signed representation, we will restrict to signed reductions in the following unless stated differently. Note that one must not forget to correct the final results to their unsigned representation. This can be achieved by conditionally adding q depending on the sign of the value.

Conditional subtraction and addition. In case the absolute value of a is small, the most straightforward reduction is subtracting (adding) q in case $a > \lfloor q/2 \rfloor$ ($a < \lfloor q/2 \rfloor$). This gives the canonical signed representative in case

Algorithm 5 Barrett reduction [Bar86]

Input: q modulus, $R = 2^n > q$
Input: $a \in \mathbb{Z}$, $|a| \leq \frac{R}{2}$, to be reduced
Output: $a' \equiv a \pmod{q}$, $|a'| < \frac{3}{2}q$.

1: $t \leftarrow \left\lfloor a \left\lfloor \frac{R}{q} \right\rceil / R \right\rceil$
2: **return** $a - qt$

$|a| < 3/2q$. However, when handling secret data one needs to ensure that this operation is implemented in constant time. If the absolute value of a is larger than $3/2q$, one can apply the add/sub multiple times. However, we will usually resort to faster reduction techniques discussed as the rest of this section. Especially after multiplications, this approach is prohibitively slow.

Lazy reduction. Intermediate values are not required to be the canonical representative. In case q is smaller than the word size, one may perform multiple arithmetic operations without reducing as one can be sure that the values will not overflow the datatype used. This is referred to as lazy reduction as one avoids all redundant reductions. This mostly works for additions and subtractions, but sometimes also for multiplications.

Range analysis. To make the best use of lazy reduction, one has to carefully keep track of possible values at each step of the computation. This is called range analysis. For larger computations, for example, an FFT (Section 2.2.5), by carefully keeping track of possible ranges, one can eliminate many reductions.

2.3.1 Barrett Reduction

For a value a, the canonical signed representative is $a \bmod^{\pm} q = a - q \cdot \left\lfloor \frac{a}{q} \right\rceil$. Barrett reduction [Bar86] computes an approximation of $\left\lfloor \frac{a}{q} \right\rceil$ to obtain a value close to $a \bmod^{\pm} q$ (or exactly $a \bmod^{\pm} q$ for some parameters). The approximation used is $\left\lfloor a \left\lfloor \frac{R}{q} \right\rceil / R \right\rceil$ with $R = 2^k > q$ fixed, such that $a \bmod^{\pm} q \approx a - q \cdot \left\lfloor a \left\lfloor \frac{R}{q} \right\rceil / R \right\rceil$. The value $\left\lfloor \frac{R}{q} \right\rceil$ can be pre-computed, and hence a reduction consists of a multiplication by $\left\lfloor \frac{R}{q} \right\rceil$, a bit shift with rounding, a multiplication by q, and a subtraction as shown in Algorithm 5.

In general, the reduction result is guaranteed to be within $\left[-\frac{3}{2}q, \frac{3}{2}q\right]$. However, one usually finds much tighter bounds for concrete parameters.

Example 16: Let $q = 17$, $R = 256$, and hence $\left\lfloor \frac{R}{q} \right\rceil = 15$. Given 72, Barrett reduction computes $a - q\left\lfloor a\left\lfloor \frac{R}{q} \right\rceil / R \right\rceil = 72 - 17\lfloor 72 \cdot 15/256 \rceil = 72 - 17 \cdot 4 = 4$. Given 78, Barrett reduction computes $78 - 17 \cdot 5 = -7$. Trying other values $|z| \leq \frac{R}{2}$, one finds that the output ranges from -9 to 8 and is, hence, very close to, but not quite the canonical signed representative.

Example 17: Assume the Kyber modulus $q = 3329$. Let $R = 2^{26}$, and, hence, $\left\lfloor \frac{R}{q} \right\rceil = 20159$.
Given, e.g., $a = 13370$, the Barrett reduction computes $13370 - 4 \cdot 3329 = 54$. The output range for signed 16-bit values is $[-1664, 1664]$, i.e., Barrett reduction yields the canonical signed representative.

2.3.2 Montgomery Reduction and Montgomery Multiplication

Similar to Barrett reduction, Montgomery's approach is to replace expensive division by odd q with a cheap division by a power of two. When reducing a value a ($< qR$) which coincidentally is a multiple of $R = 2^k > q$, we can simply store a/R which reduces the size of a by k bits and is cheaply computed. Montgomery's idea is to make sure that a is a multiple of R by introducing a correction step, i.e., we want to find a value t, such that, $a - tq$ is divisible by R. Montgomery computes t as $aq^{-1} \mod R$, such that, $a - aq^{-1}q \mod R = 0$. This results in Montgomery reduction as shown in Algorithm 6. In Montgomery's paper, this is followed by a conditional subtraction of q to obtain a value between 0 and $q-1$. However, the final conditional subtraction can be left out if the result does not need to be the standard representative.

The above description differs from Montgomery's [Mon85] original descriptions and from what is commonly described in the broader literature in the following way: Montgomery proposes a way to multiply by first transforming all operands by multiplying them by $R \mod q$. For example, let a, b be numbers to be multiplied. We first compute $aR \mod q$ and $bR \mod q$. These values are then called to be in the *Montgomery domain* or the *Montgomery space*. After multiplication, we obtain $abRR \mod q$. To obtain a result in the Montgomery domain, we need to remove the additional R-factor which we achieve by applying the Montgomery reduction. This will also reduce the size of the representation. This way of multiplying numbers is referred to as Montgomery multiplication.

However, what is encountered in real implementations of PQC looks slightly different: Only one of the two multiplicands is transformed into the Montgomery domain, such that after one multiplication one obtains $a(bR \mod q)$ which is transformed to $ab \mod q$ using one Montgomery reduction.

Algorithm 6 Montgomery reduction [Mon85]

Input: q modulus, $R = 2^n > q$
Input: $-q^{-1} \mod R$
Input: $a \in \mathbb{Z}, a < qR$
Output: $t \equiv aR^{-1} \pmod{q}, 0 \leq t < 2q$

1: $t \leftarrow a(-q^{-1}) \mod R$
2: $t \leftarrow (a + tq)/R$
3: **return** t

Algorithm 7 Signed Montgomery reduction [Sei18]

Input: q modulus, $R = 2^n > q$
Input: $q^{-1} \bmod^{\pm} R$
Input: $a \in \mathbb{Z}, a < qR$
Output: $t \equiv aR^{-1} \pmod{q}, |t| \leq q$

1: $t \leftarrow aq^{-1} \mod^{\pm} R$
2: $t \leftarrow (tq)/R$
3: $t \leftarrow \lfloor a/R \rfloor - t$
4: **return** t

Example 18: Assume we are operating in $\mathbb{Z}_{3329}$ and use $R = 2^{16}$. Hence, $-q^{-1} \mod R = 3327$. Now assume we want to multiply two numbers a and b, e.g., $a = 1234$ and $b = 17$.
In the schoolbook approach, we would first compute $a' = 1234 \cdot R \bmod^{\pm} q = 27$ and $b' = 17 \cdot R \bmod^{\pm} q = 2226$. After multiplication $a'b' = 60102$, we apply the Montgomery reduction to obtain $t = 60102 \cdot 3327 \mod R = 9018$ and consequently, $(a'b' + t \cdot q)/R = (60102 + 9018 \cdot 3329) = 459$ which equals $abR \bmod^{\pm} q$. To obtain the result in the normal domain, we can apply the Montgomery reduction again, i.e., $t = 19765$ and $(459 + 19765 \cdot 3329)/R = 1004$ which is the result we were looking for.
However, as explained above it suffices to transform one multiplicand into the Montgomery domain, e.g., $b' = 17 \cdot R \mod q = 2226$. After multiplication we have $ab' = 2746884$ and apply the Montgomery reduction, i.e., $t = 18940$ and $(ab' + tq) = (2746884 + 18940 \cdot 3329)/R = 1004$.

Another trick was introduced by Seiler [Sei18] which results in a signed variant of the Montgomery reduction and is shown in Algorithm 7. By switching to signed arithmetic, it is sufficient to compute the upper half of tq which is often cheaper than computing the full product. Note that one also needs to use q^{-1} rather than $-q^{-1}$ and consequently a subtraction rather than addition in the last line.

Example 19: Revisiting the previous example for $\mathbb{Z}_{3329}$, $R = 2^{16}$, and $q^{-1} = -3327$, $a = 1234$, and $b = 17$. When using the second approach of only transforming the second multiplicand, we get $b' = 17R \bmod^{\pm} 3329 = -1103$ and $ab' = -1361102$. We first compute $t = ab'(-q^{-1}) \bmod^{\pm} R = -20174$. Instead of computing tq, it is now sufficient to compute the upper half $(tq)/R = -1024$. The last step computes $\lfloor -1361102/R \rceil + 1024 = -20 + 1024 = 1004$

Pre-computation. In lattice-based cryptography, in particular, when using the NTT, it is commonly the case that one of the multiplicands is a constant value (e.g., a twiddle factor) that is pre-computed. When using Montgomery multiplication, it is useful to transform those constants into the Montgomery domain, i.e., store $aR \mod q$ rather than a. In that case, a Montgomery multiplication yields the product in the normal domain directly and no transformations are needed. This trick was first proposed in [ADPS16] for lattice-based cryptography and since then can be found in any fast implementation using NTTs.

2.4 Arm Cortex-M3 and Arm Cortex-M4

The main target architectures throughout this thesis are the Arm Cortex-M3 and the Arm Cortex-M4 implementing the 32-bit `Armv7-M` and `Armv7E-M` instruction set architectures (ISA) respectively. This section introduces the features of both architectures which are most relevant when optimizing cryptographic implementations and in particular polynomial multiplication.

NIST has stated that performance will play an important role in the evaluation of PQC schemes beyond the first round.[1] While most submission teams included optimized Intel implementations (Haswell, usually AVX2) in their first- and second-round submission packages, implementations for microcontrollers were rare and were only added later if at all. As a primary microcontroller optimization target, NIST recommends the use of the Cortex-M4 architecture with all options included. Consequently, it has received the most attention thus far. However, it can be considered a rather high-end microcontroller and is particularly powerful in terms of available multiplication instructions and their performance. This dramatically favors cryptographic schemes heavily relying on multiplications, e.g., in polynomial multiplication. Hence, it is also sensible to evaluate performance on less powerful microcontrollers. The immediate candidate is the Cortex-M3 which remains widely used and is significantly cheaper than the Cortex-M4. It implements the `Armv7-M` ISA and as such is very similar to the Cortex-M4, but is lacking various extensions in the `Armv7E-M` ISA.

[1] `https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/BjLtcwXALbA/Bjj_77pzCAAJ`

We first introduce the features of the `Armv7-M` ISA which are implemented by both the Cortex-M3 and Cortex-M4:

Registers. `Armv7-M` ISA features 16 32-bit registers `r0-r15` of which 14 are general purpose and usable by the developer; the other two are used for program counter (`r15`) and stack pointer (`r13`). `r14` is the link register that contains the return address for subroutine calls. After pushing its value to the stack, one can freely use `r14` for computation as well. According to the Arm ABI [ARM20] `r0-r3` are used for passing arguments to subroutines. If more arguments are needed those go to the stack. The return value is written to `r0`. As such `r0-r3` are generally caller-saved registers, while `r4-r11` are callee-saved registers, i.e., the callee has to preserve them to the stack in case they are needed. `r12` is a scratch register and is not required to be preserved.

Pipeline. Processors implementing the `Armv7-M` ISA implement a very simple pipeline. For example, the Cortex-M3 and Cortex-M4 have a simple 3-stage pipeline consisting of fetch, decode, and execute stages. This means that for all instructions, the result is ready after execution finishes, i.e., can immediately be used by the subsequent instruction. This dramatically simplifies writing optimal code for this architecture as dependencies between instructions do not introduce additional latency. One does not need to distinguish between throughput and latency of instructions.

Barrel shifter. A distinctive feature of the Arm architecture is the barrel shifter which is also called the flexible second operand. It allows to shift or rotate the second operand in most data-processing instructions without adding any latency. For example, `add r0, r1, r2, lsl#2` shifts `r2` by two to the left, adds it to `r1`, and stores the result into `r0`. This often proves useful for implementing cryptography.

Load and store instructions. There are a plethora of load and store instructions available. `ldr` and `str` are used to load or store one word (32-bit). While `str` takes one cycle (since there is a store buffer), `ldr` usually takes two cycles. However, when performing n independent `ldr` instructions consecutively, these pipeline together and usually take $n+1$ cycles. There are also instructions for different sizes. `ldrb`/`strb` handle bytes, `ldrh`/`strh` handle half-words, and `ldrd`/`strd` handle double-words. For signed data types, there are special load instructions `ldrsb` and `ldrsh` which perform sign extension to 32 bits. There also exist instructions for loading more than two words: `ldm` and `stm` which require $n+1$ cycles for loading/storing n words.

The indexing for both load and store is rather flexible:

- `ldr r0, [r1]` loads one word from address in `r1` into `r0`.

- `ldr r0, [r1,#4]` loads from address `r1+4`.
- `ldr r0, [r1,#4]!` loads from address `r1+4` and increments `r1` by four (called pre-indexed).
- `ldr r0, [r1],#4` loads from address `r1` and increments `r1` by four (called post-indexed).
- `ldr r0, [r1, r2]` loads from address `r1+r2`.
- `ldr r0, [r1, r2, lsl#1]` loads from address `r1+2·r2`.

Immediates. Standard data-processing instructions can also be used with a constant as a second operand. `mov`'s are limited to 16-bit immediates `0x0000XYZW` while immediates for other instructions are limited to an 8-bit value `0xXY` shifted by some number of bits, or the special patterns `0x00XY00XY`, `0xXY00XY00`, and `0xXYXYXYXY`.

Flags. Instructions do not set flags (e.g., the carry flag) by default. If one needs the flags, most data-processing instructions have a variant that sets the flags, e.g., `subs` instead of `sub`. The flags can then be used in other instructions, e.g., `adc` (add with carry) or branch instructions like `bne` (branch if not equal).

Instruction and data alignment. A subset of the `Armv7-M` instruction set consists of the 16-bit Thumb instructions, such as simple arithmetic and memory operations with register parameters. All other instructions are encoded in 32 bits. Using 16-bit instructions has an obvious benefit for code size, but comes at the cost of introducing misalignment: instruction fetching is significantly more expensive when instruction offsets are not aligned to multiples of four bytes. To combat this, Thumb instructions can be expanded to full-word width using the `.w` suffix.

Similar problems occur when accessing data: `ldr`/`str` require an extra cycle when the addresses are unaligned, while `ldrd`/`strd` and `ldm`/`stm` do not support unaligned addresses at all.

2.4.1 Arm Cortex-M4

In addition to the `Armv7-M` ISA features, the `Armv7E-M` ISA (and the Cortex-M4 implementing it) comes with additional functionality that is particularly useful for implementing post-quantum cryptography:

Single-cycle multiplications. Table 2.1 presents an overview of the multiplication instructions available on the Cortex-M4. Notably, each of them takes only a single cycle, i.e., the same time as simple data-processing instructions like additions. This presents a vast advantage when implementing polynomial multiplications since multiplication of integers up to 32 bits can be executed in a single cycle.

Table 2.1: Overview of (a subset of) multiplication instructions in `Armv7-M` and `Armv7E-M` and their respective cycle counts on Cortex-M4 and Cortex-M3. `X`, `Y` $\in \{$`T`, `B`$\}$

	instruction	Cortex-M4 cycles	Cortex-M3 cycles
32-bit result	`mul`	1	1
	`mla`	1	2
	`mls`	1	2
64-bit result	`umull`	1	3–5
	`smull`	1	3–5
	`umlal`	1	4–7
	`smlal`	1	4–7
DSP	`smuad`	1	–
	`smlad`	1	–
	`smulXY`	1	–
	`smulXY`	1	–
	`smulwX`	1	–
	`smlawX`	1	–

DSP instructions. The Cortex-M4 offers support for digital signal processing (DSP) instructions which are also known as single-instruction multiple-data (SIMD) instructions. DSP instructions allow computing on multiple data units (e.g., half-words) packed into 32-bit registers. They can be seen as very basic vector instructions and all execute in a single cycle. A subset of them is shown in Table 2.1 as well. For example, `smlad r0, r1, r2, r3` interprets `r1` and `r2` as packed 16-bit values ($\texttt{rX}_L$ and $\texttt{rX}_H$), and computes $\texttt{r0}_L \cdot \texttt{r1}_L + \texttt{r0}_H \cdot \texttt{r1}_H + \texttt{r3}$, i.e., computing two 16-bit multiplications and two 32-bit additions in a single cycle. There are also DSP instructions that are not multiplications, e.g., `uadd16`/`usub16` performs addition/subtraction on packed 16-bit values.

Floating-point unit. The Arm Cortex-M4 optionally supports a floating-point unit (FPU). Note that NIST specifically asked for schemes to be evaluated on Cortex-M4 with all optional features. In case it is implemented, it comes with 32 single-precision (i.e., 32-bit) floating-point registers. While floating-point computations are not common in cryptographic software, one can leverage the FPU differently: Since moving data between the FPU and the main core only requires a single cycle per word it is cheaper than accessing memory which requires $n+1$ cycles for n words. Hence, one can spill intermediate results into the FPU rather than memory to save some memory accesses.

For all experiments in this thesis, we used the STM32F407 discovery board [STM21] which has an Arm Cortex-M4. We also chose this STM32F407 for the `pqm4` framework. It comes with 192 kB of SRAM. However, only 128 kB of it is contiguous and the remaining 64 kB is core-coupled memory (CCM) in a different address range. Hence, in practice one is usually limited to 128 kB. Another caveat is that the 128 kB is segmented into two blocks of 112 kB (SRAM1) and 16 kB (SRAM2). While they are mapped consecutively in the address space, their performance characteristics are different. SRAM2 appears to be slightly slower than SRAM1. Consequently, for best performance, one only uses SRAM1 unless 112 kB SRAM is insufficient. The discovery board also comes with 1 MB of flash memory which is used for storing code and constant data. It is also possible to write to flash memory in case the SRAM is insufficient, but it incurs a vast performance penalty and requires more care as only full sectors can be written at once. For the work presented in this thesis, we do not write to flash.

Access to flash is cached in a 1024-byte instruction cache and a 128-byte data cache and one can optionally enable instruction prefetching. The STM32F407 runs at a maximum frequency of 168 MHz. However, the flash memory is much slower than this. Consequently, whenever a cache miss occurs, one has to wait for up to seven cycles for fetching instructions or data.

However, this highly depends on the memory controller and the timings of this discovery board are not representative for what will be used in a practical deployment. For example, one can use ROM to hold the code and data which is much faster (and cheaper) than flash memory. To account for this in the benchmarks, it is common practice to downclock the core to 24 MHz which is the maximum frequency that ensures that access to flash never causes stalls.

2.4.2 Arm Cortex-M3

The predecessor of the Arm Cortex-M4 is the Arm Cortex-M3. It implements the `Armv7-M` ISA and as such is very similar to the Cortex-M4, but is lacking various extensions in the `Armv7E-M` ISA. Most notably, it does not have DSP instructions or a floating-point unit. Table 2.1 gives an overview of multiplication instructions missing on the Arm Cortex-M3.

Additionally, there is an issue arising when using the `umull`, `smull`, `umlal`, and `smlal` instructions. While they are single-cycle (i.e., constant time) on the Cortex-M4, they are not single-cycle on the Cortex-M3. Instead, `umull` and `smull` take three to five cycles to execute, and `umlal` and `smlal` take four to seven cycles. As there is no authoritative information available on the early-termination conditions for these variable-time instructions, it appears dangerous to use these instructions in code that needs to be constant time. The early-termination conditions have been reverse-engineered

by de Groot [dG15], who showed that there appear to be four properties that cause an early termination: (1) Arguments being zero; (2) arguments being smaller than 16-bits; (3) top-heavy arguments (i.e., zero in the least significant 16-bits); or (4) arguments being a power of two.

Previous work by Großschädl, Oswald, Page, and Tunstall [GOPT09] evaluated early-terminating multiplication instructions on `Armv3` microcontrollers. They propose a constant-time multiplication algorithm that still uses the variable-time multiplication instructions, but avoids any shortcuts from being taken. Unfortunately, the newer `Armv7-M` ISA appears to have vastly more sophisticated shortcuts and it appears unlikely that all shortcuts can be avoided at a reasonable cost. In addition, the shortcuts identified by de Groot [dG15] are not actually confirmed by Arm. It is, hence, possible that not all shortcuts are known or that the shortcuts do not apply to all Cortex-M3 chips. Therefore, when writing constant-time code those instructions should be avoided, which means only the multiplication instructions `mul` and `mla` can be used which only compute the lower 32 bits of the 64-bit product. This presents a challenge for schemes with schemes requiring 32-bit multiplications, e.g., Dilithium. This issue is addressed in Chapter 4.

As the benchmarking platform, we use 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.

Part I

Multiplication for NTT-friendly Rings

Chapter 3

Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4

This chapter is based on work published in

> Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology – Africacrypt 2019*, LNCS, pages 209–228. Springer, 2019. `https://eprint.iacr.org/2019/489`

In this chapter, we explore implementations of Kyber [ABD+17] on the Arm Cortex-M4. At the time of publication, Kyber had advanced to the second round of the NISTPQC competition. This chapter describes the implementation of round-one and round-two Kyber. Note that this chapter presents the results as published in the paper. Later, work by Alkim, Bilgin, Cenk, and Gérard [ABCG20] improved upon the implementation presented here. In round three, the Kyber team introduced small parameter tweaks to the noise distribution, ciphertext compression, and sampling of the public matrix. The fastest round-three Kyber can be found in `pqm4`.[1]

Contribution. The main contribution of this work is to present improved optimization techniques for the NTTs in Kyber. In comparison to the performance presented in [AJS16], our NTT is more than a factor of 1.8 faster (when applying the same scaling to accommodate for the different dimensions that was also used in [AJS16]). Most of the techniques we present

[1] `https://github.com/mupq/pqm4/tree/master/crypto_kem/kyber768`

also apply to the NewHope parameters targeted in [AJS16], but some of the speedups we achieve are specific to the smaller value of $q = 7681$ (NewHope uses $q = 12289$). We also optimize the other performance-critical routines in Kyber and describe how to reduce RAM usage in Kyber without significantly sacrificing performance. As a result, we present the software, that at the same time has the smallest RAM footprint across all NISTPQC KEM candidates that have been optimized for the Cortex-M4, and has the lowest cycle count for the sum of key generation, encapsulation, and decapsulation.

Kyber v2. While this work was in submission, the Kyber team published various round-two tweaks including the change of q from 7681 to 3329 which requires changing the NTT. All the optimizations presented in this work still apply to Kyber v2. We have updated our software to support the new parameter sets and present the performance results for both versions.

Availability of software. We place all the software described in this chapter into the public domain. It is available at `https://github.com/mupq/nttm4`. All source code related to this thesis is also available in a single archive. See Appendix A. The implementations using the round-two parameter sets have also been merged into `pqm4` [KPR+].

Organization of this chapter. Section 3.1 gives the necessary background on the key-encapsulation scheme Kyber. Section 3.2 presents the speed optimizations we applied to the NTT which yield a significantly faster implementation of Kyber. Section 3.3 describes how the fast implementation of Kyber can be gradually modified to use less stack space with minor and moderate computational overhead. Finally, Section 3.4 presents the performance results for our implementations and compares them to previous implementations of Kyber and other round-two candidates in the NIST post-quantum competition.

3.1 Preliminaries

In this section, we establish notation, briefly recall Kyber and the NTT used within Kyber.

Notation. We refer to polynomials by regular-font lower-case letters (a), vectors of polynomials by bold lower-case letters ($\mathbf{a}$) and matrices of polynomials by bold upper-case letters ($\mathbf{A}$). For a polynomial a we use $\hat{a}$ to denote the representation of a in the NTT domain and similarly $\hat{\mathbf{a}}$ and $\hat{\mathbf{A}}$ are the results of the element-wise application of the NTT to the entries of $\mathbf{a}$ and $\mathbf{A}$. (Random) bitstrings are referred to by the lower-case Greek letters ρ, σ, and μ. We abstract away from seed expansion to polynomials following a uniform or centered binomial distribution by just calling `SampleUniform` or `SampleCBD`. Let q be prime and let $\mathbb{Z}_q$ denote the field $\mathbb{Z}/q\mathbb{Z}$. We define polynomial rings of the form $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ over this field where n is

a power of two. We denote by $\circ$ the coefficient-wise multiplication of two polynomials in the NTT domain with the natural extension to vectors and matrices. Similarly, let $c = \mathbf{a} \circ \mathbf{b} \in \mathcal{R}_q$ be the inner product of $\mathbf{a} \in \mathcal{R}_q^k$ and $\mathbf{b} \in \mathcal{R}_q^k$.

3.1.1 Kyber v1

Kyber [BDK+18, ABD+17], which is part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), is built on the hardness of the Module-LWE (MLWE) problem. Different from Ring-LWE, MLWE uses a matrix of polynomials in $\mathcal{R}_q$ as the public information $\hat{\mathbf{A}}$, whereas $\mathbf{s}$ and $\mathbf{e}$ become vectors of polynomials. For Kyber, $\hat{\mathbf{A}}$ is a square $k \times k$ matrix and $\mathbf{s}$ and $\mathbf{e}$ are k-dimensional vectors. MLWE, therefore, presents a generalization of the Ring-LWE and the standard LWE problem. While this might have benefits in terms of security [BDK+18], it is also an advantage for implementations: One can change the security level by changing the dimension of the matrix, i.e., by changing k. Kyber uses the prime $q = 7681 = 2^{13} - 2^9 + 1$ and $\mathcal{R}_q = \mathbb{Z}_{7681}[x]/(x^{256} + 1)$ for all security levels. Since $\mathcal{R}_q$ remains the same for all security levels it is possible to optimize all security levels of Kyber by optimizing arithmetic in $\mathcal{R}_q$. Kyber specifies three security levels: Kyber-512, Kyber-768, and Kyber-1024 which use $k = 2, 3, 4$, respectively. Besides k, the security levels only differ in the parameter of the centered binomial distribution of the secret and error polynomials which is $\eta = 5, 4, 3$ respectively.

Kyber uses a two stage-construction to obtain a CCA-secure KEM: First, build a CPA-secure encryption scheme, which is called Kyber.CPA and then use a variant of the Fujisaki-Okamoto transform [FO99] to build the CCA-secure KEM. Algorithms 8, 9, and 10 illustrate key-generation, encryption, and decryption of the CPA-secure encryption scheme. For the details of the CCA transform, we refer the reader to [ABD+17, Alg. 7–9] for the pseudocode description. Since the public matrix $\mathbf{A}$ is sampled from a uniform distribution and since the number-theoretic transform of uniform randomness is again uniformly distributed, the NTT of $\mathbf{A}$ is omitted and $\hat{\mathbf{A}}$ is instead sampled directly in the NTT domain. However, this is not possible for the secrets and errors, since those need to be small in the normal domain.

Aside from symmetric cryptography used for randomness generation and hashing (in particular in the CCA transform), the main cost in Kyber is arithmetic in $\mathcal{R}_q$ and even more specifically multiplications. The main cost of these multiplications is the (forward and inverse) NTT. The number of NTT operations depends on the parameter k and is $2k, 3k + 1$, and $k + 1$ for Kyber.CPA key generation, encryption, and decryption, respectively. Decapsulation of the CCA-secure KEM includes both Kyber.CPA encryption and Kyber.CPA decryption and thus requires $4k + 2$ NTTs.

Algorithm 8 CPA KeyGen (v1)

Output: public key $pk = (\rho, \mathbf{t}')$
Output: secret key $sk = \hat{\mathbf{s}}$

$\rho, \sigma \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$
$\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
$\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\sigma)$
$\hat{\mathbf{s}} \leftarrow \mathrm{NTT}(\mathbf{s})$
$\mathbf{t} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}) + \mathbf{e}$
return $pk = (\rho, \texttt{Compress}(\mathbf{t})), sk = \hat{\mathbf{s}}$

Algorithm 9 CPA Encryption (v1)

Input: public key $pk = (\rho, \mathbf{t}')$
Input: message $m \in \mathcal{R}_q$
Input: randomness $\mu \in \{0,1\}^{256}$
Output: ciphertext $(\mathbf{u}', v')$

$\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
$\mathbf{r}, \mathbf{e_1} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\mu)$
$e_2 \in \mathcal{R}_q \leftarrow \texttt{SampleCBD}(\mu)$
$\hat{\mathbf{r}} \leftarrow \mathrm{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
$\mathbf{t} \leftarrow \texttt{Decompress}(\mathbf{t}')$
$v \leftarrow \mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{t})^T \circ \hat{\mathbf{r}}) + e_2 + m$
return $(\texttt{Compress}(\mathbf{u}), \texttt{Compress}(v))$

Algorithm 10 CPA Decryption (v1)

Input: secret key $sk = \hat{\mathbf{s}}$
Input: compressed ciphertext $(\mathbf{u}', v')$
Output: message $m \in \mathcal{R}_q$

$\mathbf{u} \leftarrow \texttt{Decompress}(\mathbf{u}')$
$v \leftarrow \texttt{Decompress}(v')$
return $m \leftarrow v - \mathrm{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathrm{NTT}(\mathbf{u}))$

Algorithm 11 CPA KeyGen (v2)

Output: public key $pk = (\rho, \hat{\mathbf{t}})$
Output: secret key $sk = \hat{\mathbf{s}}$
1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$
2: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
3: $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\sigma)$
4: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$
5: **return** $pk = (\rho, \hat{\mathbf{t}}), sk = \hat{\mathbf{s}}$

Algorithm 12 CPA Encryption (v2)

Input: public key $pk = (\rho, \hat{\mathbf{t}})$
Input: message $m \in \mathcal{R}_q$
Input: randomness $\mu \in \{0,1\}^{256}$
Output: ciphertext $(\mathbf{u}', v')$
1: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
2: $\mathbf{r}, \mathbf{e_1} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\mu)$
3: $e_2 \in \mathcal{R}_q \leftarrow \texttt{SampleCBD}(\mu)$
4: $\hat{\mathbf{r}} \leftarrow \mathrm{NTT}(\mathbf{r})$
5: $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
6: $v \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + m$
7: **return** $(\texttt{Compress}(\mathbf{u}), \texttt{Compress}(v))$

The Number Theoretic Transform. First-round Kyber uses a complete (8 layer) negacyclic NTT for $q = 7681$ and $n = 256$ as described in Section 2.2.4. It is usually implemented using Cooley–Tukey butterflies for NTT and Gentleman–Sande for NTT^{-1} as described in Section 2.2.5.

3.1.2 Kyber v2

In the process of writing the paper underlying this chapter, the second round of NIST began and the Kyber team published an updated Kyber specification [ABD+17]. We will in the following refer to this updated version as Kyber v2.

The main design decision for the second round of the NIST competition was to remove the compression of the public key. To compensate for the increased bandwidth requirement, the Kyber team decided to reduce the value of q from 7681 to 3329, a choice that was enabled by the observation from [LS19] that this value of q also supports very fast NTT-based multiplication of polynomials. Another consequence of the decision to not compress public keys is that public keys can now be transmitted in the NTT domain, which saves an NTT operation in encryption (and in the re-encryption dur-

ing decapsulation of the CCA-secure KEM). Finally, the smaller value of q also requires smaller noise to achieve the same security level. This is why the parameter η of the centered binomial distribution changed to $\eta = 2$ for all security levels; note that this change is hidden by our high-level view of `SampleCBD`. The resulting key-generation and encryption algorithms are given in Algorithm 11 and Algorithm 12; decapsulation is the same as for the round-one version in this high-level perspective.

From a computational point of view, the most interesting aspect of the changes is the change of the definition of the NTT. In the first-round version of Kyber, q was chosen such that $\mathbb{Z}_q$ contains 512-th roots of unity. As a consequence, the negacyclic NTT of elements of $\mathcal{R}_q$ is a vector of 256 degree-zero polynomials (i.e., scalars). In the second-round version of Kyber, q is chosen such that $\mathbb{Z}_q$ contains 256-th roots of unity, but not 512-th roots of unity. As a consequence, an incomplete (7 layer) NTT has to be used as introduced in Section 2.2.7.

3.1.3 Arm Cortex-M4

Our target platform is the Arm Cortex-M4, which NIST recommended as the reference platform for evaluation of post-quantum candidates on microcontrollers. For a detailed introduction of the Arm Cortex-M4, we refer to Section 2.4.1.

3.2 Optimizing for Speed

In this section, we describe the optimizations we apply to speed up the computation of Kyber on the Arm Cortex-M4. Optimizations targeting the reduction of RAM usage will be presented in Section 3.3. The starting point of our optimization efforts is the optimized implementation for the Cortex-M4 by the Kyber authors [ABD+], which is the same as the C reference implementation except for a hand-optimized NTT operation and which is included in the `pqm4` framework [KPR+].

3.2.1 Link-Time Optimization

While experimenting with the Kyber implementation from [ABD+], we realized that its performance is heavily penalized in `pqm4` because a number of small functions (in particular modular reductions) are implemented in different files than where they are used. Since `pqm4` compiles all source files separately to object files, the compiler cannot inline those functions, which creates a large overhead from function calls. A simple, but not very elegant solution would be to place all source code in one large file and this indeed results in a speedup of about 5%.

A similar behavior can be achieved by adding the link-time optimization compiler flag `-flto`, which adds additional information in object files to allow optimization when those are linked together. Since `-flto` consistently improves performance for implementations of Kyber, we use it throughout our experiments.

We tried adding `-flto` to `pqm4` [KPR+] as a default option. However, the benchmarks show that not all schemes benefit from `-flto`. Some schemes get significantly slower, while others have an up to 60% increase in stack consumption. Therefore, `-flto` was not turned on by default in `pqm4`.

3.2.2 Speeding up the NTT

In the following, we describe our optimization strategy for the NTT, which includes a combination of known techniques with new micro-architecture-specific improvements.

Representation of polynomials. Polynomials in $\mathcal{R}_q$ have 256 coefficients in $\mathbb{Z}_q$, where q is the 13-bit prime 7681 (or 3329 for Kyber v2). It is natural to represent polynomials as an array of length 256 of 16-bit integers. Inspired by [Sei18] and unlike the implementation by the Kyber authors or the optimized NewHope implementation described in [AJS16], we use an array of *signed* 16-bit integers to represent elements of $\mathcal{R}_q$. We will later discuss the effect of this choice on modular reductions; one immediate advantage of using signed representation is that during subtractions in $\mathbb{Z}_q$ we do not have to worry about underflows. Compared to using unsigned integers we thus trivially save an addition of a multiple of q before subtractions.

Merging NTT layers. Similar to, e.g., [GOPS13] and [AJS16], we merge several layers of the NTT transformation, i.e., we load four coefficients into registers at once, perform four butterfly operations on them, and store them back. This drastically reduces the number of loads and stores. However, it turns out that merging three layers of the NTT as proposed in [AJS16] is not optimal, since there are not enough registers to fit the constants required in the Montgomery and Barrett reductions (see below). In [AJS16] this is solved by reloading the constants for each butterfly, but the cost for these loads is larger than the savings from fewer loads and stores of coefficients. We instead merge only two layers which allows us to still keep all constants in registers and still save 50% of load and store operations.

Pre-computation of twiddle factors. Like most speed-optimized NTT implementations before, we pre-compute all powers of the root of unity and store those in flash. For more efficient modular reduction after multiplication by the twiddle factors, we follow an approach first introduced in [ADPS16] and store twiddle factors in Montgomery representation [Mon85]. More specifically, our optimizations are largely inspired by the refined approach described in [Sei18] and we use the same Montgomery factor $\beta = 2^{16}$.

Algorithm 13 Original unsigned Montgomery reduction [ABD+]; using Montgomery factor $\beta = 2^{18}$.

Input: a (32 bit)
Output: reduced a (16 bit)

1: `mul` t, a, q^{-1}
2: `and` t, #0x3ffff
3: `mla` a, t, q, a ▷ $a \leftarrow a + t \cdot q$
4: `lsr` a, #18

Algorithm 14 Signed Montgomery reduction (this work, adapted from [Sei18]); using Montgomery factor $\beta = 2^{16}$.

Input: a (32 bit)
Output: reduced a (16 bit)

1: `smulbb` t, a, q^{-1} ▷ $t \leftarrow (a \mod \beta) \cdot q^{-1}$
2: `smulbb` t, t, q ▷ $t \leftarrow (t \mod \beta) \cdot q$
3: `usub16` a, a, t ▷ $a_{top} \leftarrow \left\lfloor \frac{a}{2^{16}} \right\rfloor - \left\lfloor \frac{t}{2^{16}} \right\rfloor$

We then reorder the twiddle factors in our table such that they can be picked up sequentially in the NTT computation; increasing the pointer to the twiddle factors after each load is free in `Armv7E-M`. Since we need three twiddle factors per two (merged) layers, we pack two of them into one register, which saves one load operation and one register. The twiddle factors are only used in multiplications with 16-bit coefficients which allows using `smulbb` and `smulbt` to multiply by the upper or the lower twiddle factor inside that register.

Montgomery reductions. After the multiplication in each butterfly, we need to reduce the 32-bit product to 16 bits. This is done using a signed Montgomery reduction tailored to q. It turns out that the signed Montgomery reduction as proposed in [Sei18] can be implemented in three clock cycles (Algorithm 14) on the Arm Cortex-M4 and as such is one clock cycle faster than the unsigned Montgomery reduction in [ABD+] (Algorithm 13).[2]

Unrolling. As usual, we fully unroll the outer loop of the NTT iterating over the NTT levels. Additionally, to save an additional register, we unroll one of the inner loops as well. Depending on the current level, we unroll the loop with the least iterations to minimize the code-size increase. While this is also saving a small number of cycles, the performance gains by having an additional register are much more significant.

Packing. Since q is well below 16-bits, polynomials are usually stored as `int16_t` arrays. Since our target platform is a 32-bit architecture it seems

[2] Alkim et al. [ABCG20, Algorithm 9] further improved this using `smulbb`, `smlabb` with $-q^{-1}$ as the constant.

wasteful to only load one 16-bit coefficient into 32-bit registers. Loading and storing two coefficients at once saves half of the load and store operations. However, the available vector instructions in `Armv7E-M` are quite limited. For example, there is no dedicated instruction performing two 16-bit multiplications yielding two 32-bit results. Still, some operations can be performed in parallel. Therefore, we implement *double* butterflies, i.e., butterflies which operate on packed arguments and return a packed result. By doing this, we can, for example, perform two additions and subtractions in one clock cycle using `uadd16` and `usub16`. Unfortunately, some operations (e.g., the Barrett reduction) are more than twice as expensive to implement on packed arguments. Nonetheless, we achieve a speed-up in every butterfly by using packing.

Instruction alignment. Since some instructions available in `Armv7E-M` are 16-bit Thumb instructions, it is possible that a single Thumb instruction unaligns many following 32-bit Arm instructions which results in a vast performance penalty. Therefore, we make sure our code is as aligned as possible. This can be done by aligning the start of the function using `.align 2` (`.align n` aligns to 2^n bytes) and padding each sole Thumb instruction to 32-bit using the `.w` suffix.

Recent improvements proposed in [LS19]. Very recent work proposed yet another more efficient NTT in AVX2 [LS19] which can also be adapted to Kyber. The major speed-up that [LS19] achieved over [Sei18] in the NTT stems from further optimizing the Montgomery reduction. Lyubashevsky and Seiler save an additional multiplication by avoiding the multiplication by q^{-1} and instead multiplying each of the pre-computed twiddle factors by q^{-1}. This is possible since each product of a polynomial coefficient a_i by a twiddle factor is implemented through two separate multiplication instructions, one computing the low half and one computing the high half of the product. Since the low half of the product is multiplied by $q^{-1} \mod \beta$ inside the Montgomery reduction, one can pre-compute the product of q^{-1} and the corresponding twiddle factor and use this constant for the low product. This saves another multiplication instruction in the Montgomery reduction but requires storing twice as many pre-computed twiddles.

Unfortunately, this does not carry over to our Cortex-M4 implementation since the low and high products are not computed separately, but in a single instruction. Doing these multiplications separately with different constants would be possible, but would require an additional clock cycle and, thus, would not save anything.

3.2.3 Optimizing Matrix-Vector Multiplication

Besides the NTT, another fairly expensive operation in Kyber is the matrix-vector multiplication in line 5 of Algorithm 8 and line 5 of Algorithm 9. We

also optimize this operation in C. Since this optimization depends on the stack-reduction strategy, we describe it in Section 3.3.

3.2.4 Optimized Keccak

As we will see in Subsection 3.4.3, even before our optimization of the NTT and matrix-vector multiplication, most of the cycles of the Kyber computation are spent in hashing and pseudorandom-number generation, which both boil down to the Keccak permutation [BDPA11]. For all derivatives of Keccak inside Kyber (i.e., SHA3-256, SHA3-512, SHAKE-128, and SHAKE-256) we use the highly optimized code from the eXtended Keccak Code Package [DHP+], which is also included in the `pqm4` framework.

3.2.5 Kyber v2

Various changes in the updated Kyber specification have an impact on performance, but all the optimizations presented above still apply with minor modifications: The smaller q allows being lazier with Barrett reductions in the NTT and NTT^{-1} which improves performance. Additionally, both the NTT and NTT^{-1} only require seven instead of eight layers of butterfly operations which saves roughly 1/8 of the cycles. However, the multiplication of polynomials in the NTT domain is no longer pointwise and consequently becomes more expensive.

3.3 Decreasing Stack Usage

In addition to being fast, NTT-based multiplication provides the additional benefit of being entirely in-place; no additional stack space is needed. This presents a major advantage compared to, for example, recent implementations of $\mathbb{Z}_{2^k}[x]/(f(x))$ (Chapter 5) which uses a combination of Toom-Cook [Too63, Coo66] and Karatsuba's [KO63] algorithm which comes with a rather large memory footprint. The existing implementation of the NTT in Kyber is already in-place and the changes we applied to them did not change this. Therefore, we also optimized the C code implementing the remainder of the scheme to use less stack space, making this implementation of Kyber particularly suitable for memory-constrained devices. We analyzed which stack-space requirements can be eliminated at no or very little computational cost, i.e., without re-computations.

Changes to Kyber.CCAKEM. Kyber uses an FO-transformation to transform a CPA-secure PKE into a CCA-secure KEM. The reference implementation of decapsulation does so by first decrypting the ciphertext and then re-encrypting the obtained plaintext. This produces a ciphertext which is then compared to the original. Only if they are equal, the shared secret key is returned. We eliminate this additional ciphertext on the stack by inlining the comparison into CPA encryption in a constant-time manner. This function is only used for re-encrypting and does not return a ciphertext, but rather a boolean value that indicates if the ciphertexts were equal. The actual re-encrypted ciphertext is computed and compared byte per byte. This not only saves a considerable amount of stack space but also slightly improves the speed.

Changes to Kyber.CPAPKE. The remaining changes were made in the C code of Kyber's CPA key generation (Algorithm 8), encryption (Algorithm 9), and decryption (Algorithm 10), where we reduced the number of polynomials that are kept in memory at the same time. In the reference implementation of key generation and encryption, first, the public matrix $\hat{\mathbf{A}}$ of $k \times k$ polynomials is sampled directly in the NTT domain and stored in memory. Then, vectors of noise polynomials are sampled from a centered binomial distribution. Finally, all computations are performed. We optimize this by merging the sampling and the computations, i.e., we sample the required arguments on the fly where possible.

Generating and multiplying $\hat{\mathbf{A}}$. Since a polynomial in Kyber has 256 coefficients each represented by 16 bits, storing one polynomial consumes 512 bytes of memory. Because the size of the matrix $\hat{\mathbf{A}}$ grows quadratically with k, its k^2 polynomials account for the majority of Kyber's stack usage. However, the matrix $\hat{\mathbf{A}}$ is only required once for matrix-vector pointwise multiplication and accumulation (see, e.g., line 4 of Algorithm 8).

The memory footprint can be reduced using an approach that reduces the storage requirements of $\hat{\mathbf{A}}$ to only the state of the extendable output function for one polynomial of $\hat{\mathbf{A}}$ at a time, allowing us to generate a small number of coefficients for multiplication.

In this approach, the polynomials of output vectors $\mathbf{t}$ and $\mathbf{u}$ are serialized one at a time. The vector operands $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$ are used k times in the matrix-vector multiplication. Therefore, we decided to keep those in memory throughout the computation. Only maintaining one polynomial of those in memory would require re-sampling and transforming them to the NTT domain k times which would introduce a significant performance penalty.

For key generation, we require $k + 1$ polynomials, for encryption, we require $k + 1$ polynomials, and for decryption, we only use 3 polynomials regardless of k, but since decapsulation calls both CPA encryption and decryption, the stack usage is determined by encryption.

Adding noise. The noise polynomials $\mathbf{e}$, $\mathbf{e_1}$, and e_2 are only used once and are sampled from a centered binomial distribution using an extendable output function (XOF). We sample the coefficients of those polynomials on-the-fly without having to store the entire polynomials.

Kyber v2. Our stack optimizations are mostly unaffected by the algorithmic tweaks made by the Kyber team in round two. However, in key generation (Algorithm 11), the noise vector $\mathbf{e}$ needs to be in the NTT domain. Since the NTT transformation requires the entire polynomial $\mathbf{e}$ in memory; the on-the-fly sampling is no longer possible. Therefore, key generation requires an additional polynomial, i.e., $k + 2$ in total.

3.4 Results

For our experiments, we use the STM32F407 together with an extended version of the pqm4 [KPR+] benchmarking framework. Particularly all cycles counts and stack measurements are those reported by pqm4, i.e., running the core at a low frequency of 24 MHz to not be impacted by memory wait states due to a slow memory controller. This allows comparing those numbers to boards different from the STM32F407. We extend pqm4 to also report cycles spent in hashing. Similar to pqm4 we use arm-none-eabi-gcc at version 8.2.0 and set the optimization option to -O3.

In this section, we present our results for Kyber. We start by benchmarking the NTT and polynomial multiplication in isolation and then report results for key generation, encapsulation, and decapsulation for all parameter sets of Kyber. All numbers reported in this section refer to the CCA-secure Kyber. Note that the results reported in this chapter are those reported in the original paper. Subsequent work by Alkim, Bilgin, Cenk, and Gérard [ABCG20] improved upon it. We refer to pqm4 for up-to-date results.

Table 3.1: Cycle counts for NTT, NTT^{-1}, and the full polynomial multiplication ($\text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$). We outperform the current speed record by more than a factor of two for NTT and NTT^{-1}. The parameter changes in Kyber v2 further speed up the polynomial multiplication.

	implementation	NTT	NTT^{-1}	`polymul`
Kyber v1	[ABD+]	21 855	23 622	
	This work	9 452	10 373	32 576
Kyber v2	This work	7 725	9 347	27 873

3.4.1 NTT and Polynomial Multiplication

Table 3.1 presents our new speed records for the computation of the NTT. Our optimized Kyber v1 NTT and NTT^{-1} are more than a factor of two faster than the previous speed records [ABD+].

Combining NTT and NTT^{-1} to perform a full polynomial multiplication in $\mathcal{R}_q$, i.e., computing $\text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$ requires 32 576 clock cycles. In Kyber v2 only seven out of eight layers of the NTT are computed, which reduces the run-time to roughly 7/8 of the cycles. Computing $\text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$ is considerably (14%) faster.

The fastest multiplication in $\mathbb{Z}_{2^{13}}[x]/(x^{256}+1)$ using Toom–Cook [Too63, Coo66] and Karatsuba [KO63] described in Chapter 5 requires 38 215 clock cycles.[3] We outperform this by 27%. More importantly, Toom–Cook and Karatsuba multiplication require a significant amount of additional memory for intermediate values. For $\mathbb{Z}_{2^{13}}[x]/(x^{256}+1)$, the Toom–Cook requires 3 800 bytes of intermediate values which excludes the non-reduced result polynomial of 1 022 bytes[4]. Our polynomial multiplication is in place.

In comparison to the performance presented in [AJS16], our NTT is more than a factor of 1.8 faster (when applying the same scaling to accommodate for the different dimensions that was also used in [AJS16]). Most of the techniques we present also apply to the NewHope parameters targeted in [AJS16], but some of the speedup we achieve is specific to the smaller value of q (NewHope uses $q = 12289$).

3.4.2 Kyber.CCA

Table 3.2 presents the cycle counts for all our implementations in comparison to the existing speed records [ABD+]. By just turning on `-flto`, we achieve speedups of 4 – 7% mainly due to in-lining modular reductions. The speedups achieved by applying our speed optimizations are 14 – 23% and, thus, go

[3] Note that this implementation was later outperformed by using NTTs as described in Chapter 6.

[4] $2n - 1$ coefficients of two bytes each

Table 3.2: Cycle counts for all three security levels of Kyber compared to [ABD+]. Link time optimization does benefit Kyber consistently, but our optimizations go far beyond. Kyber v2 is even faster, mainly due to algorithmic changes.

scheme	impl.	KeyGen cycles	Encaps cycles	Decaps cycles
Kyber-512 (v1)	[ABD+]	666k	904k	934k
	`lto`[a]	637k	866k	881k
	This work	575k	763k	730k
Kyber-512 (v2)	This work	499k	634k	597k
Kyber-768 (v1)	[ABD+]	1 098k	1 384k	1 417k
	`lto`[a]	1 048k	1 325k	1 339k
	This work	946k	1 167k	1 117k
Kyber-768 (v2)	This work	947k	1 113k	1 059k
Kyber-1024 (v1)	[ABD+]	1 730k	2 083k	2 134k
	`lto`[a]	1 630k	1 970k	1 994k
	This work	1 483k	1 753k	1 698k
Kyber-1024 (v2)	This work	1 525k	1 732k	1 653k

[a] Only adding the compiler flag `-flto`.

far beyond what the compiler achieves. Our implementations of the round-one variants of Kyber achieve the lowest cycle counts reported.

As a result of the optimizations described in Section 3.3, we were able to reduce the stack usage of all Kyber variants significantly (see Table 3.3). Prior to our optimizations $k^2 + 3k$, $k^2 + 4k + 3$, and $2k + 2$ polynomials were used by key generation, encryption, and decryption respectively. Our optimizations were able to reduce this to $k + 1$ for all. Therefore, we notice a more considerable reduction for the higher security levels of Kyber.

Kyber v2. With our optimizations applied to the round-two versions of Kyber, the cycle counts are comparable to round one if not faster. Similarly, stack size reductions are very comparable with the reductions made in round one. The exception is the key generation procedure which uses $k + 2$ polynomials instead of $k + 1$ as described in Section 3.3.

3.4.3 Profiling

Table 3.4 contains the profiling information of our implementations for all parameter sets of Kyber v1 and Kyber v2. We observe the following:

Dominance of hashing. Note that in the reference implementation already 54% to 69% of execution time is spent in highly hand-optimized assembly implementation of the Keccak permutation. This limits the speed-ups to be obtained since there is nothing or very little to be gained for this large fraction

Table 3.3: Stack usage for all three security levels of Kyber comparing our optimized implementations to [ABD+]. For our stack-optimized implementation we notice a significant decrease of stack usage across all variants. The stack use of key generation of version 2 is roughly one polynomial (512 bytes) larger than in version 1. This is due to the choice of Kyber's authors to represent the public key in the NTT domain.

scheme	**impl.**	KeyGen bytes	Encaps bytes	Decaps bytes
Kyber-512 (v1)	[ABD+]	6 448	9 112	9 920
	This work	2 632	2 672	2 736
Kyber-512 (v2)	This work	3 136	2 720	2 744
Kyber-768 (v1)	[ABD+]	10 544	13 720	14 880
	This work	3 072	3 120	3 176
Kyber-768 (v2)	This work	3 648	3 232	3 248
Kyber-1024 (v1)	[ABD+]	15 664	19 352	20 864
	This work	3 520	3 568	3 624
Kyber-1024 (v2)	This work	4 160	3 752	3 776

of the execution time. Our implementations spend the same time in hashing as the previous implementation, but this accounts for 64% to 81% of the total cycle counts. This confirms what previous work concluded [SBGM+18]: Post-quantum key-encapsulation schemes are vastly dominated by hashing, and having a hardware-accelerated Keccak permutation would speed up the majority of schemes significantly. Kyber v2 spends vastly less time in Keccak which is due to the change of the parameters q and η. Both allow for a more efficient sampling routine that uses less SHAKE output.

NTT. Prior to our optimizations 10% to 24% were spent in the NTT and NTT^{-1}. We speed up those parts of the code by more than a factor of two and, consequently, they only account for 5% to 14% of the cycles in our optimized implementations.

3.4.4 Comparison to other PQC Schemes

Compared to other implementations of NISTPQC KEM candidates on the Arm Cortex-M4 (Table 3.5), our Kyber implementation has both the smallest memory footprint and lowest cycle count for the sum of key generation, encapsulation, and decapsulation. Both our stack-optimized implementations of Kyber-768 outperform all other implementations by large margins in terms of stack usage. We also note a performance gap between the fastest implementation of Saber, reported in [KRS19] (Chapter 5), and the stack-optimized implementation [KBMSRV18], whereas our implementations do not suffer any slow-down due to our stack optimizations.

Table 3.4: Profiling of Kyber before and after applying all our optimizations. The run-time is vastly dominated by hashing. The cycles spent in NTT reduced notably. Only a small portion of the run-time is still spent in non-optimized code.

	impl.		**total**	**Keccak**	**NTT**	**NTT^{-1}**
Kyber-512 (v1)	[ABD+]	**K:**	666k	68%	7%	7%
		E:	904k	66%	10%	8%
		D:	934k	54%	14%	10%
	This work	**K:**	575k	79%	3%	4%
		E:	763k	78%	5%	4%
		D:	730k	69%	8%	6%
Kyber-512 (v2)	This work	**K:**	499k	71%	6%	0%
		E:	634k	74%	2%	4%
		D:	597k	64%	5%	6%
Kyber-768 (v1)	[ABD+]	**K:**	1 098k	69%	6%	6%
		E:	1 384k	67%	9%	7%
		D:	1 417k	56%	14%	8%
	This work	**K:**	946k	80%	3%	3%
		E:	1 167k	79%	5%	4%
		D:	1 117k	71%	8%	5%
Kyber-768 (v2)	This work	**K:**	947k	72%	5%	0%
		E:	1 113k	75%	2%	3%
		D:	1 059k	67%	4%	4%
Kyber-1024 (v1)	[ABD+]	**K:**	1 730k	69%	5%	5%
		E:	2 083k	67%	8%	6%
		D:	2 134k	59%	12%	7%
	This work	**K:**	1 483k	81%	3%	3%
		E:	1 753k	80%	4%	3%
		D:	1 698k	74%	7%	4%
Kyber-1024 (v2)	This work	**K:**	1 525k	73%	4%	0%
		E:	1 732k	75%	2%	3%
		D:	1 653k	69%	4%	3%

Table 3.5: Performance results of Kyber-768 in comparison to other round-two candidates of NISTPQC optimized for the Cortex-M4. Prior to this work the fasted scheme in terms of encapsulation was NTRU-HRSS, whereas key generation is (still) fastest for Saber. The best memory footprints were achieved by R5ND_3PKEb and the memory optimized variant of Saber. Note that Saber, R5ND_3PKEb, and NTRU-KEM-743 are claiming NIST security level 3, whereas NTRU-HRSS claims NIST security level 1.

scheme	**impl.**	**runtime** cycles		**stack usage** bytes	
Kyber-768 (v1)	This work	**K:**	946k	**K:**	3 072
		E:	1 167k	**E:**	3 120
		D:	1 117k	**D:**	3 176
Kyber-768 (v2)	This work	**K:**	947k	**K:**	3 648
		E:	1 113k	**E:**	3 232
		D:	1 059k	**D:**	3 248
Frodo-AES128	[BFM+18]	**K:**	41 681k	**K:**	31 116
		E:	45 758k	**E:**	51 444
		D:	46 720k	**D:**	61 820
Frodo-cSHAKE128	[BFM+18]	**K:**	81 300k	**K:**	26 272
		E:	86 255k	**E:**	41 472
		D:	87 212k	**D:**	51 848
Saber	[KRS19][a]	**K:**	902k	**K:**	13 248
		E:	1 173k	**E:**	15 528
		D:	1 217k	**D:**	16 624
	[KBMSRV18][b]	**K:**	1 165k	**K:**	6 931
		E:	1 530k	**E:**	7 019
		D:	1 635k	**D:**	8 115
R5ND_3PKEb	[SBGM+18][c]	**K:**	1 032k	**K:**	6 796
		E:	1 510k	**E:**	8 908
		D:	1 913k	**D:**	4 296
NewHopeCCA1024	[KPR+][a,d]	**K:**	1 221k	**K:**	11 152
		E:	1 902k	**E:**	17 448
		D:	1 926k	**D:**	19 648
NTRU-HRSS	[KRS19][a]	**K:**	145 986k	**K:**	23 396
		E:	406k	**E:**	19 492
		D:	827k	**D:**	22 140
NTRU-KEM-743	[KRS19][a]	**K:**	5 203k	**K:**	25 320
		E:	1 603k	**E:**	23 808
		D:	1 884k	**D:**	28 472

[a] Re-benchmarked in SRAM1 (see beginning of Section 3.4)
[b] Optimized for stack consumption
[c] Since R5ND_3PKEb does not report any stack usage, we report the numbers from `https://github.com/mupq/pqm4/pull/16`
[d] NTT assembly implementation from [AJS16] with reference implementation in `pqm4` [KPR+]

Chapter 4

Compact Dilithium Implementations on Cortex-M3 and Cortex-M4

This chapter is based on work published in

> Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. `https://eprint.iacr.org/2020/1278`

It presents new speed records for the digital signature scheme and NIST-PQC finalist Dilithium on Arm Cortex-M3 and Arm Cortex-M4. At the time of publishing, Dilithium had just advanced to the third round of the NIST-PQC competition, and, hence, the parameter changes in the third round were not yet known. The core implementation techniques introduced here directly carry over to the new parameter sets. However, the performance results presented here are for the second-round parameters. Updated implementations have been merged into `pqm4`.[1]

Together with Falcon [PFH+17], Dilithium [LDK+17] is one of the two remaining lattice-based signature schemes in the NISTPQC competition in round three. In round two, there was an additional lattice-based signature scheme called qTesla [BAA+17]. However, qTesla was not selected to advance to the third round.

Dilithium and qTesla are conceptually very similar; they are both Fiat-Shamir-with-abort schemes [Lyu09] based on {R,M}LWE and {R,M}SIS.

[1] `https://github.com/mupq/pqm4/pull/178`

However, Dilithium has significantly smaller keys, smaller signatures, and better performance. For example, qTESLA-p-I public keys are 14 880 bytes and signatures are 2 592 bytes, while Dilithium2 has 1 184-byte public keys and 2 044 byte signatures albeit providing the same level of (claimed) security. Note that qTesla initially also proposed *heuristic* parameter sets which achieved sizes and performance closer to Dilithium, but the qTesla team withdrew those parameter sets because Lyubashevsky and Schwabe presented a complete break allowing universal forgeries.[2]

Falcon, on the other hand, is a *hash-and-sign* signature scheme based on NTRU lattices. It has smaller public key and signature sizes than Dilithium while being competitive in terms of computational performance.

Dilithium, together with the finalist key-encapsulation mechanism Kyber [ABD+17], forms the Cryptographic Suite for Algebraic Lattices (CRYSTALS). Both Dilithium and Kyber use structured lattices to allow fast arithmetic and compact key, signature, and ciphertext sizes. Both make use of the polynomial ring $\mathbb{Z}_q[x]/(x^{256}+1)$ which enables efficient polynomial multiplication using the number theoretic transform (NTT). However, Dilithium is using a 23-bit prime modulus, while Kyber is using a 12-bit prime modulus which means that implementations of polynomial arithmetic differ significantly between Kyber and Dilithium.

While there is a vast literature on the implementation of lattice-based key-encapsulation schemes, the coverage of lattice-based signatures is still limited and more research is needed. We advance the field by presenting optimized implementations of Dilithium for the Arm Cortex-M3 and Arm Cortex-M4. In this work, aside from optimizing for speed, we also optimize for stack usage.

The Cortex-M4 has been declared the main microcontroller optimization target for the post-quantum competition by NIST. Hence, the majority of schemes in the third round have an optimized implementation for that architecture. However, its *smaller brother*, the Cortex-M3, is also still widely deployed.

The Cortex-M4 provides various advanced instructions for optimizing cryptographic schemes which might be one of the reasons why it has received much attention from the cryptographic community.

However, 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]. This

[2] https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/HHnavSx4f5Q/fRsujb9ACgAJ

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 fourfold: 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. Third, we present various stack 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. Finally, as a by-product, we provide Cortex-M3 implementations of the lattice-based key-encapsulation schemes Kyber and NewHope. These, most notably, consist of constant-time implementations of the NTT and NTT^{-1} operations in those schemes.

Code. The implementations of Dilithium, Kyber, and NewHope that are the result of this work are in the public domain and available at `https://github.com/dilithium-cortexm/dilithium-cortexm`. All source code related to this thesis is also available in a single archive. See Appendix A.

Related Work. Previous speed-records for Dilithium on the Cortex-M4 were set by Ravi, Gupta, Chattopadhyay, and Bhasin [RGCB19] and were built upon work by Güneysu, Krausz, Oder, and Speith [GKOS18]. A masked implementation of a modified Dilithium on Cortex-M3 is presented in [MGTF19]. 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 [AJS16, ABCG20, KBMSRV18, BMKV20]; see also Chapter 3, Chapter 5, and Chapter 6. Similar studies exist on hardware implementations and instruction set extensions [BMTK+20, BUC19, AEL+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, [WTJ+20] presented a hardware-accelerated implementation of qTesla.

Structure of this chapter. Section 4.1 introduces the lattice-based signature scheme Dilithium. In Section 4.2 we present some improvements for the Cortex-M4. Section 4.3 presents the first constant-time implementation of

Table 4.1: Dilithium parameter sets

Name	NIST level	(k, ℓ)	η	β	ω	$\lvert pk \rvert$	$\lvert sig \rvert$
`Dilithium2`	1	$(4, 3)$	6	325	80	1184	2044
`Dilithium3`	2	$(5, 4)$	5	275	96	1472	2701
`Dilithium4`	3	$(6, 5)$	3	175	120	1760	3366

Dilithium on the Cortex-M3. Section 4.4 presents various trade-offs in terms of stack consumption and speed of Dilithium implementations. Section 4.5 presents the performance results for both implementations. In Section 4.6, we provide performance results for Kyber and NewHope on the Cortex-M3 which are a by-product of this work.

4.1 Preliminaries

4.1.1 Dilithium

Dilithium [DKL+18, LDK+17] is a digital signature scheme based on the hardness of the MLWE and the MSIS lattice problems. It is one of the three digital signature finalists of the NIST Post-Quantum Competition [NIS16]. Note that with the advance to the third round the Dilithium submitters have introduced tweaks to the scheme. The following description is based on the second-round specification.

Parameters. The Dilithium signature scheme consists of four different parameter sets called `Dilithium1`, `Dilithium2`, `Dilithium3`, and `Dilithium4` of which the latter three target NIST security levels one to three respectively. We omit `Dilithium1` in the following as it falls short of the lowest NIST security level.[3] Dilithium operates in the polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$; denoted by R_q in the following. Across all parameter sets, the modulus is fixed at $q = 2^{23} - 2^{13} + 1 = 8380417$ and all polynomials have $n = 256$ coefficients.

Furthermore, for all parameter sets the bound γ_1 is set to $(q-1)/16 = 523776$ and $\gamma_2 = \gamma_1/2 = 261888$. For each parameter set, the remaining parameters and the resulting public key and signature sizes are given in Table 4.1. The parameters consist of the matrix dimension (k, ℓ), the sampling bounds of the secret η, and the rejection thresholds β and ω. The Dilithium signature-generation algorithm uses rejection sampling to find a signature that can be both correctly verified and does not leak information about the secret key. Due to this iterative nature, the runtime of Dilithium varies significantly between multiple signature generations. Note, however, that the rejection probability does not depend on the secret key, and consequently,

[3] `Dilithium1` has also been eliminated from the third-round submission.

Algorithm 15 Dilithium key generation

Output: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Output: Public key $pk = (\rho, \mathbf{t}_1)$
1: $\rho \leftarrow \{0,1\}^{256}$
2: $K \leftarrow \{0,1\}^{256}$
3: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
4: $\hat{\mathbf{A}} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
5: $\mathbf{t} := \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) := \texttt{Power2Round}(\mathbf{t})$
7: $tr \in \{0,1\}^{384} := \mathcal{H}(\rho \| \mathbf{t}_1)$

the variable run-time caused by rejection sampling does not leak any secret information [LDK+17, Section 3.3].

Notation. We follow the notation of the Dilithium specification [LDK+17] and denote polynomials by lower-case Latin letters like c, vectors of polynomials by bold lower-case letters like $\mathbf{t}$, and matrices by bold upper case letters ($\mathbf{A}$). Polynomials, vectors, and matrices that have been transformed to the NTT domain are identified by their hat, e.g., $\hat{c}$, $\hat{\mathbf{a}}$ and $\hat{\mathbf{A}}$. The operator $\circ$ describes coefficient-wise multiplication. The operator $\|$ denotes concatenation of two inputs that are implicitly converted to a byte-string. $\|a\|_\infty$ refers to the maximum absolute coefficient of the polynomial a and is similarly defined for vectors. When sampling a from a certain distribution S, we write $a \leftarrow S$. S_η is the uniform distribution ranging from $-\eta$ to $+\eta$.

Functions. As a building block, Dilithium uses the NTT and NTT^{-1} function which are used to implement efficient polynomial multiplication of a, b as $\mathrm{NTT}^{-1}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$. The details of the Dilithium NTT are described later in this section. In addition, Dilithium uses a collision-resistant hash-function $\mathcal{H}$ with 384-bit output length and a cryptographic hash-function $\mathcal{H}_B$ outputting a polynomial that has exactly 60 coefficients set to ± 1 while the remaining 196 coefficients are zero. The hash functions $\mathcal{H}$ and $\mathcal{H}_B$ are implemented using the extendable-output function (XOF) `SHAKE256`. Furthermore, the Dilithium specification defines the seed-expansion functions `ExpandA` and `ExpandMask`; the rounding functions `Power2Round`, `HighBits`, and `Decompose` and the hint functions `MakeHint` and `UseHint`. To keep the algorithm description brief, we omit the details of those functions and refer the reader to the Dilithium specification.

Scheme Specification. Algorithm 15, Algorithm 16, and Algorithm 17 specify Dilithium key generation, signature generation, and signature verification. The descriptions are consistent with the ones from Figure 4 in the Dilithium specification [LDK+17], but we omit details about rounding that are not relevant to this work.

Algorithm 16 Dilithium signature generation

Input: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Input: Message $M \in \{0,1\}^*$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$

$\hat{\mathbf{A}} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathcal{H}(tr \| M)$
$\kappa := 0; (\mathbf{z}, \mathbf{h}) = \perp$
$\rho' \in \{0,1\}^{384} := \mathcal{H}(K \| \mu)$
$\hat{\mathbf{s}}_1 := \mathrm{NTT}(\mathbf{s_1}); \hat{\mathbf{s}}_2 := \mathrm{NTT}(\mathbf{s_2}); \hat{\mathbf{t}}_0 := \mathrm{NTT}(\mathbf{t}_0)$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
    $\mathbf{y} \in S^{\ell}_{\gamma_1 - 1} := \texttt{ExpandMask}(\rho', \kappa)$
    $\mathbf{w} := \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{y}))$
    $\mathbf{w}_1 := \texttt{HighBits}(\mathbf{w})$
    $c := \mathcal{H}_B(\mu \| \mathbf{w}_1)$
    $\hat{c} := \mathrm{NTT}(c)$
    $\mathbf{z} := \mathbf{y} + \mathrm{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$
    $(\mathbf{r}_1, \mathbf{r}_0) := \texttt{Decompose}(\mathbf{w} - \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2))$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **or** $\mathbf{r}_1 \neq \mathbf{w}_1$ **then**
        $(\mathbf{z}, \mathbf{h}) = \perp$
    else
        $\mathbf{h} := \texttt{MakeHint}\left(-\mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0), \mathbf{w} - \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2) + \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)\right)$
        if $\|\mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)\|_\infty \geq \gamma_2$ **or** # 1's in $\mathbf{h} > \omega$ **then**
            $(\mathbf{z}, \mathbf{h}) = \perp$
        end if
    end if
    $\kappa := \kappa + 1$
end while

Number Theoretic Transform. At the core of the Dilithium scheme construction and parameter choices is the NTT which allows efficient polynomial multiplication. Dilithium uses a negacyclic NTT with $n = 256$ and $q = 8380417$, which allows a complete NTT as described in Section 2.2.4. We implement it using Cooley–Tukey butterflies for NTT and Gentleman–Sande for NTT^{-1} as described in Section 2.2.5.

4.1.2 Target Platforms: Cortex-M3 and Cortex-M4

We target the Cortex-M4 and Cortex-M3 platforms previously described in Section 2.4.1 and Section 2.4.2.

Algorithm 17 Dilithium verification

Input: Public key $pk = (\rho, \mathbf{t}_1)$
Input: Message $M \in \{0,1\}^*$
Input: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
Output: `Valid` or `Invalid`

$\hat{\mathbf{A}} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{384} := \mathcal{H}(\mathcal{H}(\rho \| \mathbf{t}_1) \| M)$
$\mathbf{w}_1' := \texttt{UseHint}(\mathbf{h}, \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(2^d \cdot \mathbf{t}_1)))$
if $c = \mathcal{H}_B(\mu \| w_1')$ **and** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** # 1's in $\mathbf{h} \leq \omega$ **then**
    return `Valid`
else
    return `Invalid`
end if

4.2 Improving Cortex-M4 Performance

Our Cortex-M4 implementation is based on the Dilithium implementation by Ravi, Gupta, Chattopadhyay, and Bhasin [RGCB19], which includes the NTT and NTT^{-1} assembly implementation of Güneysu, Krausz, Oder, and Speith [GKOS18].

In Dilithium, the NTT and NTT^{-1} 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 NTT^{-1} 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 4.1.

In our implementation, similarly to previous work, we pre-compute 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] to reduce the product modulo q.

After each level of the NTT and NTT^{-1}, 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 NTT^{-1}. This technique of delaying the reduction is usually referred to as *lazy reduction*.

When implementing the NTT and NTT^{-1}, we first unroll the outer loop

Algorithm 18 CT butterfly from [GKOS18]

Input: `p0, p1, twiddle`
Input: `q=8380417, qinv=4236238847`
Output: `p0, p1`

```
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
```

Algorithm 19 Our CT butterfly

Input: `p0, p1, twiddle`
Input: `q=8380417, qinv=4236238847`
Output: `p0, p1`

```
1: smull tmp0, tmp1, p1, twiddle
2: mul p1, tmp0, qinv
3: smlal tmp0, tmp1, p1, q
4: sub p1, p0, tmp1
5: add p0, p0, tmp1
```

which iterates over the eight levels of the NTT and NTT^{-1}. Furthermore, similar to the merging technique in [GOPS13], we can merge two levels of the NTT and NTT^{-1} 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. As a consequence, the number of store and load instructions is reduced by a factor of 2.

Lastly, the main difference which 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 wraparound, the subtractions in the reference implementation are accompanied by addition of a multiple of q, pushing the results back into the positive domain. By switching to the signed representation, the problem of negative wraparounds is fixed, and we do not need this extra multiple-of-q addition. This allows us to eliminate all these additions throughout the code.

Algorithm 20 GS butterfly in [GKOS18]

Input: `p0, p1, twiddle`
Input: `q=8380417, qinv=4236238847`
Output: `p0, p1`

```
1: add tmp0, p0, q, lsl#8
2: sub tmp0, tmp0, p1
3: add p0, p0, p1
4: umull tmp1, p1, tmp0, twiddle
5: mul tmp0, tmp1, qinv
6: umlal tmp1, p1, tmp0, q
```

Algorithm 21 Our GS butterfly

Input: `p0, p1, twiddle`
Input: `q=8380417, qinv=4236238847`
Output: `p0, p1`

```
1: sub tmp0, p0, p1
2: add p0, p0, p1
3: smull tmp1, p1, tmp0, twiddle
4: mul tmp0, tmp1, qinv
5: smlal tmp1, p1, tmp0, q
```

This is especially relevant for the NTT and NTT^{-1} implementations because every butterfly operation has a subtraction. Algorithm 19 shows our improvements to the CT butterfly in the NTT by [GKOS18] which is shown in Algorithm 18. For the GS butterflies in the NTT^{-1}, the improvements are listed in Algorithms 20 and 21.

However, the wraparound-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 NTT^{-1}, 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, NTT^{-1}, and hashing operations using `Keccak`. We omit the details, as they result straightforwardly from the reference code.

4.3 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 untrue 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 17) are only operating on public data and can, therefore, be implemented in variable time. Similarly, in signature generation (Algorithm 16) $\text{NTT}(t_0)$ (line 5), $\text{NTT}(\mathcal{H}_B(\mu, \mathbf{w}_1))$ (line 10), and $\text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$ (line 16 and 17) are not processing secret data as both $\mathbf{t}$ and c are considered public. For the details, we refer to the security proof in [LDK+17, 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 15) 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 `smull` 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 not 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.

4.3.1 `smull` and `smlal`

As Dilithium uses a 23-bit modulus q, its polynomials are commonly 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 already discussed—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 $\mathbb{Z}_{q_i}[x]/(x^n+1)$, 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 NTRU Prime [BCLv17]. 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 [ABD+17] and NewHope [AAB+17] respectively. We report the results for the full schemes in Section 4.6.

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 22 and Algorithm 23. 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 `SBSMLAL`, c_0 and c_1 initially contain the value to be added to the

Algorithm 22 Schoolbook `smull` (`SBSMULL`)

Input: $a = a_0 + 2^{16}a_1, b = b_0 + 2^{16}b_1$
Output: $c = ab = c_0 + 2^{32}c_1$

```
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 23 Schoolbook `smlal` (`SBSMLAL`)

Input: $a = a_0 + 2^{16}a_1, b = b_0 + 2^{16}b_1, c = c_0 + 2^{16}c_1$
Output: $c = c + ab = c_0 + 2^{32}c_1$

```
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
```

product. On the Cortex-M3, additions and multiplications use one cycle, while `mla` uses two cycles. As such, the `SBSMULL` macro takes seven cycles to execute, while `SBSMLAL` takes 9 cycles.

Algorithm 24 Constant-time Cooley–Tukey butterfly on the M3

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

```
1: SBSMULL tmpl, tmph, p1l, p1h, tl, th
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) += (p1l,p1h)· q
6: sub p1, p0, tmph
7: add p0, p0, tmph
```

It is important to note that `SBSMULL` (`SBSMLAL`) is not semantically equivalent to `smull` (`smlal`). In case the accumulation $(a_0b_1 + a_1b_0)$ in line 7 of Algorithm 22 or line 11 of Algorithm 23 overflows, the carry bit is lost and the result will not be 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 ($< q$) 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 $(a_0b_1 + a_1b_0)$ 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_0b_1 and a_1b_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}.

4.3.2 Cooley–Tukey and Gentleman–Sande Butterflies

Using constant-time `SBSMULL` and `SBSMLAL` subroutines, we can construct the butterfly operations needed to implement the NTT and NTT^{-1}. Algorithm 24 depicts the modified Cooley–Tukey butterfly operation based on Algorithm 19. 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

Algorithm 25 Constant-time Gentleman–Sande butterfly on the M3

Input: `p0, p1` (32-bit signed), twiddle = `tl`+2^{16}`th` (`tl` unsigned, `th` signed)
Input: $q = 8380417 =$ `ql` + 2^{16}`qh`, `qinv = 4236238847`
Output: `p0, p1` (32-bit signed)

```
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) += (tmpl,tmph)·q
```

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 7 and 8). In total, the Cooley–Tukey butterfly operation requires 21 cycles on the Cortex-M3, while Algorithm 19 only needs five cycles on the Cortex-M4.

Similarly, Algorithm 25 depicts our constant-time assembly implementation of the Gentleman–Sande butterfly. As the addition and subtraction happen 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 2, 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 five cycles for Algorithm 21 on the Cortex-M4.

4.3.3 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 upper signed 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 a 32-bit value and can, hence, be stored back using `str`.

For the NTT^{-1}, we proceed analogously. 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 two multiplication instructions and, hence, two 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 24 and Algorithm 25. In case both inputs are considered public, we simply use the pointwise multiplication which was presented in Section 4.2 section.

4.4 Time-Memory Trade-Offs

Depending on the programmer's requirements, there are multiple ways in which we can implement Dilithium signing, each with its 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. Consequently, 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 implement 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 run-time 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 (i.e. in SRAM/CCM for the STM32F407 and SRAM1/SRAM2 for the ATSAM3X8E), we see no real benefit in doing that. Therefore, to keep it simple, we will store the variables on the stack.

4.4.1 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 the 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[4]) to store these extra values. Then, during the signature generation algorithm, instead of passing in sk (as described in line 1 of Algorithm 16), we pass a larger struct that also contains the pre-computed values. These pre-computed values ($\mathbf{A}$, $\hat{\mathbf{s}}_1$, $\hat{\mathbf{s}}_2$ and $\hat{\mathbf{t}}_0$) add up to $k \cdot l + 2k + l$ polynomials that have to be stored extra. In the case of `Dilithium3`, this amounts to 34 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.

4.4.2 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 Algorithm 16. Apart from some space for storing intermediate values, we will need to allocate

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

This adds up to a pretty high lower bound of $k \cdot l + 4k + 3l + 1$ KiB of necessary stack space, e.g., 53 KiB for `Dilithium3`.

4.4.3 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 pre-computed values. In the

[4] $\mathbf{A}$, 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.

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 8 of Algorithm 16). Because we do not retain $\mathbf{y}$ after this step, we regenerate it again in line 12 of Algorithm 16). Relative to strategy 2, this saves us $k \cdot l$ 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 13 and 17 of Algorithm 16, the values $\mathbf{r}_1$, $\mathbf{r}_0$ and $\mathbf{h}$ all depend on both $\mathbf{w}$ and $\hat{c}$. However, in line 11 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 $\mathbf{r}_1$ and $\mathbf{r}_0$ in line 13, and when we are constructing the hint $\mathbf{h}$ in line 17.

In order to recompute elements of $\mathbf{w}$, we would have to do the matrix multiplication $\mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{y}))$ 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.

4.4.4 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 into 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 [EGM90], and was first proposed for usage in lattice-based signature schemes in [AYS15]. It has also been used last year 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[5] is needed to store the pre-computed values for `Dilithium3`, such that there is a 95% probability that at least one of the y values results in a good signature. For our main target (the ATSAM3X8E), that would mean that more than half its flash space would already be lost to storing these pre-computed values. We think that, in the general case, the improved signature-generation latency does not justify this kind of loss in available flash space.

[5] See [RGCB19, Table 6]. Compute 300 – 34 = 266 KB ≈̂ 260 KiB.

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 the 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 4.5.5.

Cortex-M4 setup. We benchmark all our Cortex-M4 implementations on an STM32F407 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 are based on `pqm4` [KPR+] and benchmarking our code within `pqm4` gives the same performance results. Our code has been merged into `pqm4`.[6]

Cortex-M3 setup. The Cortex-M3 measurements were done on an Arduino Due board that 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`. On both platforms, 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) [DHP+]. As it only uses `Armv7-M` instructions, we use the same implementation on both platforms.

Side-channel Protection. In our implementations we are only considering timing side-channels, i.e., we provide constant-time code that avoids leaking secret data through variable time instructions, secret-dependent branching, and secret-dependent memory addresses. For certain use-cases one may want to consider also protecting against more powerful attacks like power analysis attacks, e.g., using masking. There exists work in the literature for masking Dilithium by Migliore, Gérard, Tibouchi, and Fouque [MGTF19] which presents a protected implementation modified Dilithium. There is more work required for implementing a fully masked Dilithium that is adhering to the

[6] `https://github.com/mupq/pqm4/pull/163`

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. For the Cortex-M3, we report cycles for constant-time (CT) code and variable-time code.

		CT		NTT	NTT^{-1}	$\circ$
Dilithium [a]	[GKOS18]	✓	M4	10 701	11 662	–
	This work	✓	M4	8 540	8 923	1 955
	This work	–	M3	19 347	21 006	4 899
	This work	✓	M3	33 025	36 609	8 479
Kyber [b]	[ABCG20]	✓	M4	6 855	6 983	2 325
	This work	✓	M3	10 819	12 994	4 773
NewHopeCCA1024 [c]	[ABCG20]	✓	M4	68 131	51 231	6 229
	This work	✓	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 $\text{NTT}/\text{NTT}^{-1}$

specification submitted to NIST. However, this work is outside of the scope of this chapter and we leave it for future work.

4.5.1 NTT Performance

In Table 4.2, we list the benchmarking results on the Cortex-M4 and Cortex-M3 for the optimized NTT, NTT^{-1}, and pointwise multiplication ($\circ$) of Dilithium, Kyber, and NewHopeCCA1024. For the Cortex-M4, we obtain a speedup of 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 several 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-M4 (respectively 3 – 5, 4 – 7, and two on the M3 vs. one on the Cortex-M4), the cycle counts for NTT, NTT^{-1}, and $\circ$ increase between 2.3× and 2.5×. Making that constant time on the Cortex-M3 using `SBSMULL` and `SBSMLAL` from Section 4.3.1 increases the number of cycles by a factor of 1.7×.

Table 4.3: Performance results on the Cortex-M4 (STM32F407 at 24 MHz). Averaged over 10 000 executions.

Algorithm/ strategy	Param	Work	Speed [kcc]	Stack [B]
KeyGen (1)	2	**This work**	2 267	7 916[c]
	3	**This work**	3 545	8 940[d]
	4	**This work**	5 086	9 964[e]
KeyGen (2 & 3)	2	**This work**	1 315	7 916
	3	[GKOS18]	2 320	50 488
	3	**This work**	2 013	8 940
	4	**This work**	2 837	9 964
Sign (1)	2	[RGCB19, scen. 2][a]	3 640	–
	2	**This work**	3 097	14 428[c]
	3	[RGCB19, scen. 2][a]	5 495	–
	3	**This work**	4 578	17 628[d]
	4	[RGCB19, scen. 2][a]	4 733	–
	4	**This work**	3 768	20 828[e]
Sign (2)	2	[RGCB19, scen. 1][b]	4 632	–
	2	**This work**	3 987	38 300
	3	[GKOS18]	8 348	86 568
	3	[RGCB19, scen. 1][b]	7 085	–
	3	**This work**	6 053	52 756
	4	[RGCB19, scen. 1][b]	7 061	–
	4	**This work**	6 001	69 276
Sign (3)	2	**This work**	13 332	8 924
	3	**This work**	23 550	9 948
	4	**This work**	22 658	10 972
Verify	2	**This work**	1 259	9 004
	3	[GKOS18]	2 342	54 800
	3	**This work**	1 917	10 028
	4	**This work**	2 720	11 052

[a] "Strategy 1" from Section 4.4.1 corresponds to "Scenario 2" in [RGCB19].

[b] "Strategy 2" from Section 4.4.2 corresponds to "Scenario 1" in [RGCB19].

[c] For `Dilithium2` using stack strategy 1, additional 23 632 bytes of flash space are used for storing the pre-computed values.

[d] For `Dilithium3` using stack strategy 1, additional 34 896 bytes of flash space are used for storing the pre-computed values.

[e] For `Dilithium4` using stack strategy 1, additional 48 208 bytes of flash space are used for storing the pre-computed values.

4.5.2 Cortex-M4 Performance

Table 4.3 lists the benchmarking results of our Dilithium implementation, together with the cycle counts from the relevant related work. As the signing time varies considerably depending on the number of rejections, we performed 10 000 executions and took the average of the resulting cycle counts.

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. Compared to the [GKOS18] implementation, which is comparable to our signing strategy 1, we obtain speedups of 13%, 27%, and 18% for key generation, signing, and verification respectively. We also drastically decrease the stack consumption.

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%.

Our strategy 3 implementation which is solely optimized for memory footprint, achieves by far the worst performance in terms of speed.

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

4.5.4 Stack Usage

Up to this point, we have mainly discussed the improvements in Dilithium's speed. However, as already mentioned, it is also important to be economic in the usage of stack space.

In Tables 4.3 and 4.4, we show the considerable improvement in stack-space usage over the previous works. 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 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. On both platforms, we see really disproportionate cycle counts for signature generation, for example with `Dilithium3` signature generation takes about 33 million cycles on the Cortex-M3. On slow devices (like our 16 MHz Arduino Due), this latency grows into the order of *seconds*.

4.5.5 Profiling

To identify how much is still left to optimize in our implementations, we profiled the implementations on the Cortex-M4. Table 4.5 contains the profile for all our Dilithium implementations. 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, 77% for signing, and 81% for verification, which greatly limits the speedup achievable by further optimizing the arithmetic of the scheme.

Only about 3.4% to 24.5% of cycles are spent in the NTT and NTT^{-1}. 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.

4.6 Kyber and NewHope on Cortex-M3

As a side-product of Section 4.3, 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, and 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}. Besides that, we port the code for polynomial addition, pointwise multiplication, and bit-reversal to 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.

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

Algorithm/ strategy	Params	Speed [kcc]	Stack [B]
KeyGen (1)	`Dilithium2`	2 945	12 631[a]
	`Dilithium3`	4 503	15 703[b]
	`Dilithium4`	6 380	18 783[c]
KeyGen (2 & 3)	`Dilithium2`	1 699	7 983
	`Dilithium3`	2 562	9 007
	`Dilithium4`	3 587	10 031
Sign (1)	`Dilithium2`	5 822	14 869[a]
	`Dilithium3`	8 730	18 083[b]
	`Dilithium4`	7 398	21 273[c]
Sign (2)	`Dilithium2`	7 115	39 503
	`Dilithium3`	10 667	53 959
	`Dilithium4`	10 031	70 463
Sign (3)	`Dilithium2`	18 932	9 463
	`Dilithium3`	33 229	10 495
	`Dilithium4`	31 180	11 511
Verify	`Dilithium2`	1 541	8 944
	`Dilithium3`	2 321	9 967
	`Dilithium4`	3 260	10 999

[a] For `Dilithium2` using stack strategy 1, an additional 23 632 bytes of flash space are used for storing the precomputed values.

[b] For `Dilithium3` using stack strategy 1, an additional 34 896 bytes of flash space are used for storing the precomputed values.

[c] For `Dilithium4` using stack strategy 1, an additional 48 208 bytes of flash space are used for storing the precomputed values.

Table 4.5: Dilithium profiling results on the Cortex-M4

Param	Operation	KeyGen		Sign			Verify
		(1)	(2 & 3)	(1)	(2)	(3)	
2	Keccak	81%	81%	41%	55%	75%	77%
	NTT	5%	2%	5%	7%	7%	5%
	NTT^{-1}	2%	3%	20%	12%	4%	3%
	other asm	5%	6%	10%	9%	4%	7%
	not opt.	**7%**	**8%**	**24%**	**16%**	**10%**	**8%**
3	Keccak	83%	83%	50%	64%	77%	79%
	NTT	4%	2%	6%	7%	7%	4%
	NTT^{-1}	1%	2%	14%	9%	3%	2%
	other asm	5%	6%	12%	8%	4%	7%
	not opt.	**6%**	**7%**	**18%**	**12%**	**9%**	**7%**
4	Keccak	85%	84%	51%	62%	76%	81%
	NTT	4%	2%	6%	6%	7%	4%
	NTT^{-1}	1%	2%	13%	9%	4%	2%
	other asm	5%	7%	13%	10%	4%	7%
	not opt.	**5%**	**6%**	**17%**	**13%**	**10%**	**6%**

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.

			KeyGen [kcc]	Encaps [kcc]	Decaps [kcc]
Kyber512	[ABCG20]	M4	455	586	544
	This work	M3	539	682	652
Kyber768	[ABCG20]	M4	864	1 033	970
	This work	M3	1 012	1 194	1 145
Kyber1024	[ABCG20]	M4	1 405	1 606	1 526
	This work	M3	1 636	1 853	1 793
NewHopeCCA1024	[ABCG20]	M4	1 157	1 675	1 587
	This work	M3	1 239	1 921	1 888

Part II

Multiplication for NTT-unfriendly Rings

Chapter 5

Toom–Cook and Karatsuba Multiplication for $\mathbb{Z}_{2^m}[x]$

This chapter is based on work published in

> Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security – ACNS 2019*, LNCS, pages 281–301. Springer, 2019. `https://eprint.iacr.org/2018/1018`

This work first appeared online during the first round of the NISTPQC competition. In this chapter, we look at KEMs based on structured lattices that were not designed for using NTT-based polynomial multiplication, i.e., use polynomial rings $\mathcal{R}_q$ that are not immediately suitable for benefiting from the NTT. In particular, we look at schemes using a power-of-two modulus $q = 2^m$ which includes six of the 22 lattice-based first-round NISTPQC KEM candidates. Specifically, these schemes are Round2 [GMZB+17], Saber [DKRV17], NTRU-HRSS [HRSS17b], NTRUEncrypt [ZCHW17], Kindi [Ban17], and RLizard [CPL+17]. After the first round, Round2 merged with Hila5 [Saa17] into Round5 [BBF+19] and the Round5 team presented optimized software for the Arm Cortex-M4 processor in [SBGM+18]; the multiplication in Round5 has more structure, allowing for a specialized high-speed routine.

We optimize the other five schemes relying on arithmetic in $\mathcal{R}_q$ with a power-of-two q on the same platform. Note that Saber has previously been optimized on the Arm Cortex-M4 [KBMSRV18] as well; our polynomial multiplication implementation outperforms their results by 42% which

improves the overall performance of key generation by 22%, encapsulation by 20%, and decapsulation by 22%. For the other four schemes, the only software that was readily available for the Cortex-M4 was the reference implementation and, unsurprisingly, our carefully optimized code significantly outperforms these implementations. For example, our optimized implementations of RLizard-1024 and Kindi-256-3-4-2 encapsulation and decapsulation are more than a factor of 20 faster. Our implementation of NTRU-HRSS encapsulation and decapsulation solidly outperforms the optimized Round5 software presented in [SBGM+18].

We achieve our results by systematically exploring different combinations of Toom-3, Toom-4, and Karatsuba decomposition [Too63, Coo66, KO63] of multiplication in $\mathcal{R}_q$, and by carefully hand-optimizing multiplication of low-degree polynomial multiplication at the bottom of the Toom/Karatsuba decomposition. The exploration of the different approaches is automated through a set of Python scripts that generate optimized assembly given the parameters $q = 2^k$ for $k \leq 16$ and $n \leq 1024$. These Python scripts may be of independent interest for a similar design-space exploration on different architectures.

Organization of this chapter. In Section 5.1 we briefly recall the five NIST candidates that we optimize in this chapter. In Section 5.2 we first detail our approach to explore different Toom and Karatsuba decomposition strategies for multiplication in $\mathcal{R}_q$ and then explain how we hand-optimized schoolbook multiplications of low-degree polynomials. Finally, Section 5.3 presents performance results for stand-alone multiplication in $\mathcal{R}_q$ for the different parameter sets, and for the five NIST candidates.

Availability of the software. We have released all software presented in this chapter, including the Python scripts used for design-space exploration, into the public domain. The software is available at `https://github.com/mupq/polymul-z2mx-m4` and the implementations have been integrated into the `pqm4` framework [KPR+]. All source code related to this thesis is also available in a single archive. See Appendix A.

Second and third round of NISTPQC. Since this work first appeared online, NIST announced the second-round candidates of the post-quantum competition. While Kindi and RLizard are no longer under consideration by NIST, Saber, NTRU-HRSS, and NTRUEncrypt made it to the second round. NTRU-HRSS and NTRUEncrypt were merged into the new scheme NTRU. The optimizations presented in this chapter carry over directly to the second-round schemes. In 2020, NTRU and Saber moved to the third round.

5.1 Preliminaries

In this section, we briefly review the five NIST candidates that we optimize in this chapter. Readers interested in the multiplication routine outside the context of NIST submissions are encouraged to skip ahead.

5.1.1 Cryptosystems Optimized in this Chapter

Notation. The full specification of each of the five CCA-secure KEMs would take several pages, so for the sake of brevity, we leave out various details. In this section, we highlight the relevant aspects.

In particular, all five schemes build a CCA-secure KEM from an encryption scheme; for all but NTRUEncrypt, this encryption scheme is only passively secure. In our descriptions, we focus only on the encryption schemes underlying the KEM and highlight the multiplications in $\mathcal{R}_q$—the main target of our optimization effort—by denoting those multiplications with $\circledast$. In general, we denote scalar multiplications with $\cdot$ and polynomial multiplications with $*$.

Similarly, we do not go into any detail with respect to the sampling of random bit strings, polynomials, or matrices, and simply denote all of these functions as $\mathsf{Sample}_{\mathcal{R}}$, where $\mathcal{R}$ is the set from which the elements are drawn. While we specify a set to which the sampled elements belong, we leave the distribution according to which they are sampled unspecified. Where deterministic sampling from a specific seed is relevant, $\mathsf{Sample}_{\mathcal{R}}$ is parameterized with this seed.

Finally, many schemes make use of rounding coefficients of polynomials. We denote any such rounding operation by $\lfloor \dots \rceil$, specify the domain in which the result lives, but again omit the details of how the rounding operation is defined.

RLizard. RLizard is part of the Lizard submission to NIST [CPL+17]. It is a cryptosystem based on the Ring-Learning-with-Errors (Ring-LWE) and Ring-Learning-with-Rounding (Ring-LWR) problems. These problems are closely related, and efficient reductions exist [BPR12, BGM+16]. The submission motivates the choice for the Learning-with-Rounding problem by stressing its deterministic encryption routine and reduced ciphertext size compared to Learning-with-Errors. RLizard.KEM is a CCA-secure KEM that is constructed by applying Dent's variant of the FO transform [FO99, Den03] to the RLizard CPA-secure PKE scheme, which is summarized in Algorithms 26, 27, and 28.

The main structure underlying RLizard is the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, but coefficients of the ciphertext are ultimately reduced to $\mathcal{R}_p$, where $p < q$. We consider the parameter set where $n = 1024$, $q = 2048$ and $p = 512$. In the submission the derived KEM is referred to as `RING_CATEGORY3_N1024`

Algorithm 26 RLizard.KeyGen ()

1: $a, s, e \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $b \leftarrow -a \circledast s + e \in \mathcal{R}_q$
3: **return** $(\mathsf{pk} = (a, b), \mathsf{sk} = s)$

Algorithm 27 RLizard.Enc $(m, (a, b))$

1: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $c_1' \leftarrow a \circledast r \in \mathcal{R}_q$
3: $c_2' \leftarrow b \circledast r \in \mathcal{R}_q$
4: $c_1 \leftarrow \lfloor (p/q) \cdot c_1' \rceil \in \mathcal{R}_p$
5: $c_2 \leftarrow \lfloor (p/q) \cdot ((q/2) \cdot m + c_2') \rceil \in \mathcal{R}_p$
6: **return** (c_1, c_2)

Algorithm 28 RLizard.Dec $((c_1, c_2), s)$

1: $m' \leftarrow \lfloor (2/p) \cdot (c_2 + c_1 \circledast s) \rceil \in \mathcal{R}_2$
2: **return** m'

– for clarity, we denote it as RLizard-1024 from this point onwards. All multiplications in RLizard fit the structure that we target in this work.

NTRU-HRSS. The NTRU-HRSS scheme [HRSS17a] is based on the 'classic' NTRU cryptosystem [HPS98]. It starts from the CPA-secure NTRU encryption scheme, and, like RLizard, applies Dent's variant of the FO transform [FO99, Den03] to construct a CCA-secure KEM. By restricting the parameter space compared to traditional NTRU, the scheme is simplified and avoids implementation pitfalls such as decryption failures and fixed-weight sampling. We look at the concrete instance as submitted to NIST [HRSS17b], i.e., fix the parameters to $p = 3$, $q = 8192$ and $n = 701$. NTRU-HRSS relies on arithmetic in a number of different rings. Glossing over the technicalities (see Sections 2 and 3 of [HRSS17a]), we reuse the notation to define $\Phi_d = 1 + x^1 + x^2 + \cdots + x^{d-1}$, and then define $\mathcal{R}_p = \mathbb{Z}[x]_p/\Phi_n$, $\mathcal{R}_q' = \mathbb{Z}[x]_q/\Phi_n$ and $\mathcal{R}_q = \mathbb{Z}[x]_q/(x^n - 1)$, but abstract away the transitions between rings.

Algorithms 29, 30, and 31 show that the scheme requires several multiplications and inversions. For this chapter, we focus on multiplications in $\mathcal{R}_q'$ and $\mathcal{R}_q$. However, the same routine can be used to perform the multiplication in $\mathcal{R}_p$. Furthermore, as the inversion in $\mathcal{R}_q'$ can be performed using multiplications [HRSS17a], this benefits from the same optimization.

NTRUEncrypt. The NTRUEncrypt scheme [ZCHW17] is also based on the standard NTRU construction [HPS98], but chooses parameters based on a recent revisiting [HGSW05]. The NIST submission of NTRUEncrypt [ZCHW17]

Algorithm 29 NTRU-HRSS.KeyGen ()

$f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_p}$
$f_p^{-1} \leftarrow f^{-1} \in \mathcal{R}_p$
$f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}'_q$ ▷ Uses mult. in $\mathcal{R}_q$
$h \leftarrow \Phi_1 * g \circledast f_q^{-1} \in \mathcal{R}_q$
return $(\mathsf{pk} = p \cdot h, \mathsf{sk} = (f, f_p^{-1}))$

Algorithm 30 NTRU-HRSS.Enc $(m, (p \cdot h))$

$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
$c \leftarrow h' \circledast r + m \in \mathcal{R}_q$
return c

Algorithm 31 NTRU-HRSS.Dec $\left(c, (f, f_p^{-1})\right)$

$v \leftarrow c \circledast f \in \mathcal{R}_q$
$m' \leftarrow v \circledast f_p^{-1} \in \mathcal{R}_p$
return m

presents several instantiations, but we limit ourselves to the instances where $q = 2^k$. We look at the parameter set NTRU-KEM-743, where $p = 3$, $q = 2048$, and $n = 743$; the arithmetic takes place in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$, but coefficients are also reduced modulo p when moving to $\mathcal{R}_p$. The optimizations in this work also carry over to the smaller NTRU-KEM-443 parameter set, but not to NTRU-KEM-1024 (which uses a prime q). As before, the relevant multiplication occurs when the noise polynomial r is multiplied with the public key h, but we also utilize our multiplication routine for the other multiplication in Dec. See the algorithmic descriptions in Algorithm 32, 33, and 34.

Saber. Like Lizard and RLizard, Saber [DKRV17] also relies on the Learning-with-Rounding problem. Rather than directly targeting LWR or the ring variant, it positions itself in the middle-ground formed by the Module-LWR problem. The submission conforms to the common pattern of proposing a PKE scheme and then applying an FO variant [HHK17] to obtain a CCA-secure KEM. We give descriptions of the PKE scheme in Algorithm 35, 36, and 37. Like RLizard, Saber operates in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, and in the smaller $\mathcal{R}_p$. Because of the Module-LWR structure, however, n is fixed to 256 for all parameter sets. Instead of varying the dimension of the polynomial, Saber variants use matrices of varying sizes with entries in the polynomial ring (denoted $\mathcal{R}^{\ell \times k}$). With the fixed $q = 8192$, this ensures that an optimized routine for multiplication in $\mathcal{R}_q$ directly applies to the smaller

Algorithm 32 NTRUEncrypt.KeyGen ()

1: $f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $h \leftarrow (p \cdot g)/(p \cdot f + 1) \mod q$
3: **return** $(\mathsf{pk} = h, \mathsf{sk} = (f, h))$

Algorithm 33 NTRUEncrypt.Enc (m, h)

1: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
2: $t \leftarrow r \circledast h$
3: $m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
4: $m' \leftarrow m - m_{mask} \mod p$
5: $c \leftarrow t + m'$
6: **return** c

Algorithm 34 NTRUEncrypt.Dec $(c, (f, h))$

1: $m' \leftarrow f \circledast c \mod p$
2: $t \leftarrow c - m$
3: $m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
4: $m \leftarrow m' + m_{mask} \mod p$
5: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
6: **if** $p \cdot r \circledast h = t$ **then**
7: **return** m
8: **else**
9: **return** $\perp$
10: **end if**

Algorithm 35 Saber.KeyGen ()

$\rho \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho)$
$s \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$b \leftarrow \lfloor A \circledast s + h \rceil \in \mathcal{R}_p^{\ell}$
return $(\mathsf{pk} = (\rho, b), \mathsf{sk} = s)$

Algorithm 36 Saber.Enc $(m, (\rho, b))$

$A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{l \times l}}(\rho)$
$s' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$b' \leftarrow \lfloor A \circledast s' + h \rceil \in \mathcal{R}_p^{\ell}$
$v' \leftarrow b \circledast \lfloor s' \rceil \in \mathcal{R}_p$
$c_m \leftarrow \lfloor v' + (p/2) \cdot m \rceil \in \mathcal{R}_{2t}$
return (c_m, b')

Algorithm 37 Saber.Dec $((c_m, b'), s)$

$v \leftarrow b' \circledast \lfloor s \rceil \in \mathcal{R}_p$
$m' \leftarrow \lfloor v - (p/(2t)) \cdot c_m + h \rceil \in \mathcal{R}_2$
return m'

Lightsaber and the larger Firesaber instances as well. Other parameters p and t are powers of two smaller than q; for the Saber instance[1], $p = 1024$ and $t = 8$. The vector h is a fixed constant in $\mathcal{R}_q^\ell$.

Note that some of the multiplications in Saber are in $\mathcal{R}_q$ and some are in $\mathcal{R}_p$; in our software both use the same routine. As we will explain in Section 5.2, the smaller value of p would in principle allow us to explore a larger design space for multiplications in $\mathcal{R}_p$; However, for the small value of $n = 256$ there is nothing to be gained in the additional multiplication approaches.

Kindi. In the same vein as Saber, Kindi [Ban17] is based on a matrix of polynomials, relating it to the Module-LWE problem. However, it relies on a trapdoor construction which is somewhat more intricate than the standard approach. It constructs a CPA-secure PKE that is already close to a key-encapsulation mechanism. Kindi operates in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ with $q = 2^k$, the more general $\mathcal{R}_b = \mathbb{Z}_b[x]/(x^n+1)$ for some integer b, and in the polynomial ring with integer coefficients $\mathcal{R} = \mathbb{Z}[x]/(x^n+1)$. The relevant arithmetic primarily happens in $\mathcal{R}_q$, though, meaning that the performance of Kindi still considerably improves as a consequence of this work. We consider the parameter set Kindi-256-3-4-2, where $n = 256$ and $q = 2^{14}$.

In Algorithms 38, 39, and 40, we list the PKE. Here, $g \in \mathcal{R}_q$ is a constant, $\ell = 3, p = 4$, and $[p] \in \mathcal{R}_q$ is constant with all coefficients equal to p. We omit public key compression and message encoding for ease of exposition. To obtain a CCA-secure KEM, a slightly simplified version of the modular FO variant [HHK17] is used: as Kindi exhibits a KEM-like structure and already includes re-encryption in Dec, this results in merely adding hash-function calls.

5.1.2 Arm Cortex-M4

Our target platform is the Arm Cortex-M4, which NIST recommended as the reference platform for evaluation of post-quantum candidates on microcontrollers. For a detailed introduction of the Arm Cortex-M4 refer to Section 2.4.1.

5.2 Multiplication in $\mathbb{Z}_{2^m}[x]$

As discussed in the previous sections, we focus on multiplication in $\mathcal{R}_q$, where $q = 2^m$. In particular, we approach this by looking at the non-reduced multiplication in $\mathbb{Z}_{2^m}[x]$, as this is identical across all schemes we investigate. The reduction is done outside of our optimized polynomial multiplication.

[1] Note that both the scheme and the category three parameter set are called Saber.

Algorithm 38 Kindi.KeyGen()

$\mu \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
$\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$\mathsf{r}, \mathsf{r}' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
$\mathsf{b} \leftarrow \mathsf{A} \circledast \mathsf{r} + \mathsf{r}'$
return $(\mathsf{pk} = (\mathsf{b}, \mu), \mathsf{sk} = (\mathsf{r}, \mathsf{b}, \mu))$

Algorithm 39 Kindi.Enc $(m, (\mathsf{b}, \mu))$

$s_1 \leftarrow \mathsf{Sample}_{\mathcal{R}_2}$
$\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$\mathsf{p} \leftarrow b + \mathsf{g}$
$\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
$(s_2, \ldots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
$\mathsf{s} \leftarrow (s_1 + 2 \cdot \bar{s_1} - [p], s_2 - [p], \ldots, s_\ell - [p]) \in \mathcal{R}_q^{\ell}$
$\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
$\mathsf{u} \leftarrow \bar{u} \oplus m$
$\mathsf{e} \leftarrow (\mathsf{u}_1 - [p], \ldots, \mathsf{u}_\ell - [p]) \in \mathcal{R}_q^{\ell}$
$e_{\ell+1} \leftarrow u_{\ell+1} - [p]$
$(\mathsf{c}, c_{\ell+1}) \leftarrow (\mathsf{A} \circledast \mathsf{s} + \mathsf{e}, \mathsf{p} \circledast \mathsf{s} + \mathsf{g} \cdot [p] + \mathsf{e}) \in \mathcal{R}_q^{\ell+1}$
return $(\mathsf{c}, c_{\ell+1})$

Algorithm 40 Kindi.Dec $(\mathsf{r}, \mathsf{b}, \mu, (\mathsf{c}, c_{\ell+1}))$

$\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
$\mathsf{p} \leftarrow \mathsf{b} + \mathsf{g}$
$\mathsf{v} \leftarrow c_{\ell+1} - \mathsf{c} \circledast \mathsf{r}$
$s_1 \leftarrow (\lfloor \mathsf{v}_1 / 2^{\log q - 1} \rceil, \ldots, \lfloor \mathsf{v}_n / 2^{\log q - 1} \rceil) \in \mathcal{R}_2$
$\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
$(s_2, \ldots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
$\mathsf{s} \leftarrow (s_1 + 2 \cdot \bar{s_1} - [p], s_2 - [p], \ldots, s_\ell - [p])$
$\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
$(\mathsf{e}, e_{\ell+1}) \leftarrow (\mathsf{c} - \mathsf{A} \circledast \mathsf{s}, c_{\ell+1} - \mathsf{p} \circledast \mathsf{s}) \in \mathcal{R}_q^{\ell+1}$
$\mathsf{u} \leftarrow (\mathsf{e}_1 + [p], \ldots \mathsf{e}_\ell + [p])$
$u_{\ell+1} \leftarrow e_{\ell+1} + [p]$
$m \leftarrow \mathsf{u} \oplus \bar{u}$
return m

Here, we describe the way we break down such a multiplication for a specific number of coefficients n, modulo a specific q. This is done using combinations of Toom-Cook's and Karatsuba's multiplication algorithms. For a given n and q, there are multiple possible approaches; we explore the entire space and select the optimum for each parameter set. We use Python scripts that generate optimized assembly functions for all combinations, for arbitrary-degree polynomials (with degrees below 1024). These scripts are parameterized by the degree, the Toom method (see the next subsection; Toom-3, Toom-4, both Toom-4 and Toom-3 or no Toom layer at all), and the threshold at which to switch from Karatsuba to schoolbook multiplication. See Section 5.3.1 for a detailed analysis of these results.

5.2.1 Toom/Karatsuba Strategies

The naive schoolbook approach to multiply two polynomials with n coefficients results in n^2 multiplications in $\mathbb{Z}_q$. Using well-known algorithms by Karatsuba [KO63] (Section 2.2.2) and Toom-Cook [Too63, Coo66] (Section 2.2.3), it is possible to trade some of these multiplications for additions and subtractions.

Toom-Cook. It is important to note that there is a loss in precision when using Toom's method, as it involves division over the integers. While divisions by three and five can be replaced by multiplications by their inverses modulo 2^{16}, i.e., 43691 and 52429, this is not possible for divisions by powers of two. Consequently, Toom-3 loses one bit of precision, and Toom-4 loses three bits. Since our Karatsuba and schoolbook implementations operate in $\mathbb{Z}_{2^{16}}[x]$, this imposes constraints on the values of q for which our implementations can be used; Toom-3 can be used for $q \leq 2^{15}$, Toom-4 can be used for $q \leq 2^{13}$. These losses accumulate, and a combination of both is only possible if $q \leq 2^{12}$. This also rules out higher-order Toom methods. While switching to 32-bit arithmetic would allow using higher-order Toom, this slows down Karatsuba and the schoolbooks significantly by increasing load-store overhead and ruling out DSP instructions.

While asymptotically Toom-4 is more efficient than Toom-3 and Karatsuba, in practice the additions and subtractions also impact the run-time. The increased number of memory accesses following a more intricate pattern also significantly influence performance. Thus, for a given n it is not immediately obvious in general which approach is the fastest. We first evaluate whether to decompose using a layer of Toom-4, Toom-3, both Toom-4 and Toom-3, or no Toom at all. We then repeatedly apply Karatsuba's method to break down the multiplications, up to the threshold at which it becomes inefficient and the "naive" schoolbook method becomes the fastest approach.

Karatsuba. The call to the topmost Karatsuba layer is a function call, but from that point on, we recursively inline the separate layers. Upon reaching

the threshold at which the schoolbook approach takes precedence, we jump to the schoolbook multiplication through a function call.

This provides a trade-off that keeps code size reasonable and is flexible to implement and experiment with, but does imply that the register allocation between the final Karatsuba layer and the underlying schoolbook is disjoint; it may prove worthwhile to look into this for specific n rather than in a general approach.

As we perform several nested layers of Karatsuba multiplication, it is important to carefully manage memory usage. We do not go for a completely in-place approach (as is done in [KBMSRV18]), but instead allocate stack space for the sums of the high and low limbs, relying on the input and output buffers for all other terms. This leads to effective memory usage without reducing performance.

Assembly-level optimizations. For both Toom and Karatsuba, the typical operations require adding and subtracting polynomials of moderate size from a given address. We stress the importance of careful pipelining, loading, and storing 16-bit coefficients pairwise into full-word registers, and using `uadd16` and `usub16` arithmetic operations. We rely on offset-based instructions for memory operations, in particular for the more intricate memory access patterns in Toom and Karatsuba. This leads to a slight increase in code size compared to using `ldm` and `stm`, (and some bookkeeping for polynomials exceeding the maximal offset of 4095 bytes), but ensures that addresses are computed during code generation.

For ease of implementation, our code generator for Toom is restricted to dimensions that divide without remainder. For Karatsuba, we do not restrict the dimensions at all: the implementation can work on unbalanced splits, and thus polynomials of unequal length. In order not to waste any memory or cycles here (e.g., by applying common refinement approaches), the Python script becomes a rather complex composition of conditionals; rather than trying to combine pairs of 16-bit additions into `uadd16` operations on the fly, we run a post-processing step over the scheduled instructions to do so.

Rather than considering alignment to 32-bit word boundaries during code generation, we use a post-processing step. After compilation, we disassemble the resulting binary and expand Thumb instructions in cases where they cause misalignment. This allows using the smaller Thumb instructions where possible but avoids paying the overhead of misalignment. In particular, this is important when an odd number of Thumb instructions is followed by a large block of 32-bit instructions. The alignment post-processing is done using a Python script that is included in our software package and may be of independent interest.

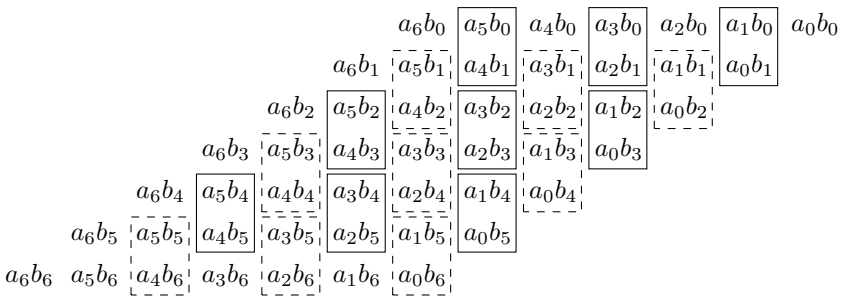


Figure 5.1: Pairing coefficients to reduce the number of multiplications using `smladx` / `smlad`. Dashed boxes represent multiplications involving repacked b.

5.2.2 Small Schoolbook Multiplications

We investigate several approaches to perform the small-degree schoolbook multiplications that underlie Karatsuba and Toom-Cook, varying the approaches and implementing distinct generation routines for different n. For each approach, we keep the polynomial in packed representation, loading all coefficients into the 32-bit registers in pairs. The `Armv7E-M` instruction set provides multiplication instructions that efficiently operate on data in this format: parallel multiplications, but also instructions that operate on a specific half-word.

For $n \leq 10$, all input coefficients can be kept in registers simultaneously, with registers remaining to keep the pointers to the source and destination polynomials around. We first compute all coefficients of terms with odd exponents, before using `pkh` instructions to repack one of the input polynomials and computing the remaining coefficients. This ensures that the vast majority of the multiplications can be computed using the two-way parallel multiply-accumulate dual instructions. See Figure 5.1 for an illustration of this; here, b is repacked to create the dashed pairs. This is somewhat similar to the approach used in [KBMSRV18], but ends up needing less repacking and memory interaction.

For $n \in \{11, 12\}$, we spill the source pointers to the stack after loading the complete polynomials. At these dimensions, the registers are used to their full potential, and by using the DSP instructions we end up needing only 78 multiplications; 66 combined multiplications, 12 single multiplications, and not a single dedicated addition instruction. This offsets the extra cost of the six packing instructions considerably. For $n \in \{13, 14\}$, not all coefficients fit in registers at the same time, leading to spills for the middle columns (i.e., the computation of coefficients around x^n, which are affected by all input coefficients). Even when using the Python abstraction layer, manual register allocation becomes somewhat tedious in cases that involve many spills to the

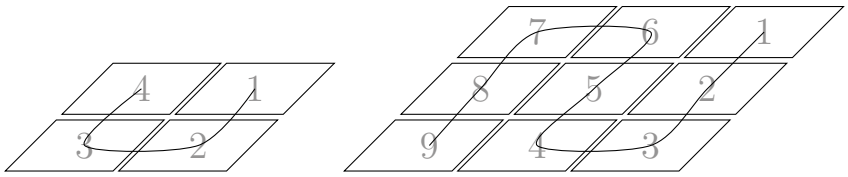


Figure 5.2: Decomposing larger schoolbook multiplications

stack. To remedy this, we use bare-bone register-allocation functions akin to the scripts in [HRSS17a].

For larger n, the above strategy leads to an excessive amount of register spills. Instead, we compose the multiplication of a grid of smaller instances. For $15 \leq n \leq 24$, we compose the multiplication out of four smaller multiplications, for $25 \leq n \leq 36$, we use a grid of nine multiplications, etc. Note that we use at most $n = 12$ for the building blocks, given the extra overhead of the register spills for $n \in \{13, 14\}$. We further remark that it is important to carefully schedule the (re)loading and repacking of input polynomials. We illustrate this in Figure 5.2.

The approach described above works trivially when n is divisible by $\lceil \frac{n}{12} \rceil$, but leads to a less symmetric pattern for other dimensions. We plug these holes by starting from an n that divides even, and either adding a layer 'around' the parallelogram or nullifying the superfluous operations in a post-processing step.

Figure 5.3 shows the performance of these routines; see Table 5.1 for more details.

5.3 Results and Discussion

In this section, we present benchmark results for polynomial multiplication, and for key generation, encapsulation, and decapsulation of the five NIST post-quantum candidates Kindi, NTRUEncrypt, NTRU-HRSS, RLizard, and Saber. For each of the schemes, we have tried to select the parameter set which targets NIST security category three. However, NTRU-HRSS only provides a category one parameter set, hence we use this. Furthermore, the reference implementations for the category three parameter sets of Kindi require more than 128 KiB of RAM and consequently do not trivially fit our platform (STM32F4DISCOVERY). We use Kindi-256-3-4-2 instead, which targets security category 1.

All cycle counts presented in this section were obtained by using an adapted version of the `pqm4` benchmarking framework [KPR+], which uses the built-in 24-bit SysTick timer. Stack measurements were also obtained using the method implemented in `pqm4`, i.e., by writing dummy values to the

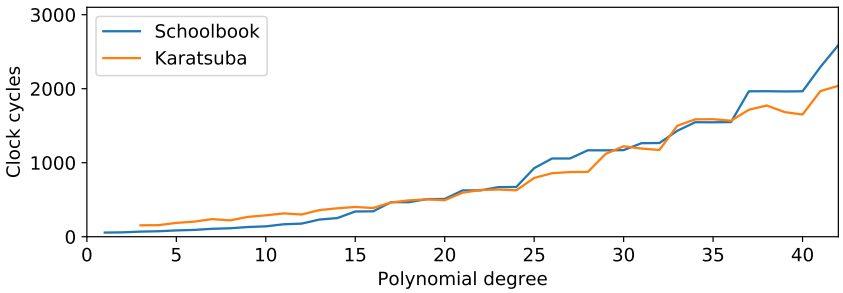


Figure 5.3: Runtime of generated optimized polynomial multiplication for small n. For $n < 20$ our hand-optimized schoolbook multiplications are clearly superior, for $n > 36$ first applying at least one layer of Karatsuba is faster.

entire memory available for the stack, running the scheme under test, and subsequently checking how much of the dummy values were overwritten.

5.3.1 Multiplication Results

We first present results for polynomial multiplication as a building block. We report benchmarks for the multiplication for all possible $n < 1024$, using different approaches to evaluate which strategy is optimal.

Figure 5.3 shows the run-time of our hand-optimized schoolbook implementations and the generated optimized Karatsuba code for small n. For the Karatsuba benchmarks, we have selected the optimal schoolbook threshold, e.g., for $n = 32$ one could either apply one layer of Karatsuba and then use the schoolbook method for $n = 16$ or, alternatively, use two layers of Karatsuba and use schoolbook multiplications for $n = 8$. The former variant is faster in this scenario, which leads to a schoolbook threshold of 16. For each n, we simply iterated over all schoolbook thresholds and selected the fastest variant. The graph shows that directly applying the schoolbook method is superior for $n < 20$, and for $n > 36$ Karatsuba outperforms schoolbook. However, for values in between, the plot is inconclusive. A large cause of this is the amount of hand-optimization that went into some of our schoolbook implementations, but it is also strongly determined by register pressure: there is a large performance hit in the step from $n = 14$ to $n = 15$, which then propagates to dimensions that break down to these schoolbook multiplications using Karatsuba. For cryptographically relevant values we found that the cross-over point is at $n = 22$, i.e., for values $n > 22$ one should use an additional layer of Karatsuba.

Figure 5.4 shows the performance of the different multiplication ap-

proaches for larger n. While that general trend is visible, one still observes a jagged line. We speculate that the main cause for this is similar to the irregularities in Figure 5.3: the variance in the increasing cost of the schoolbooks is magnified as n grows larger and specific schoolbook sizes are repeated in the decomposition of large multiplications. Because of the difference in decomposition between Toom-3 and Toom-4, this favors each method for different ranges for n, resulting in alternating optimality. Another factor that is impacted by specific decomposition is the resulting memory access pattern, and, by extension, data alignment, resulting in a large performance penalty. In practice, comparing benchmarks for specific n seems to be the only way to come to conclusive results. In particular, we observe that the lines are not even monotonically increasing; note that it is trivially possible to pad a smaller-degree polynomial and use a larger multiplication routine to benefit from a more efficient decomposition.

As Figure 5.4 does not allow us to identify which method performs best for clear bounds on n, we instead focus on individual n as relevant for the five cryptographic schemes we intend to cover. This restricts n to $\{256, 701, 743, 1024\}$. In Table 5.2, we report the cycle counts alongside the required additional stack space for each of the multiplication methods. All cycle counts are for polynomial multiplication *excluding* subsequent reduction required to obtain an n-coefficient polynomial; additional cost for reduction differs depending on the specific choice of ring. While there is some performance benefit in performing the reduction inline, the main gain is in stack usage. For the Toom variants, this allows for in-place recomposition, reducing stack usage by roughly $2n$ coefficients. This is not trivial for Karatsuba, though, introducing some additional complexity. We leave this for future work.

For the rather small $n = 256$ (Saber, Kindi), we already see that Toom-4 (followed by two layers of Karatsuba) is slightly faster than directly applying Karatsuba. As the difference is small, however, one might decide to not use a Toom layer at all, at the benefit of a much simpler implementation and considerably reduced stack usage. Toom-4 is not suitable for Kindi ($n = 256, q = 2^{14}$), as q is too large. Again the impact is marginal, though, as Karatsuba is only a few percent slower at this dimension, also performing just above Toom-3. For larger $n \in \{701, 743, 1024\}$ (NTRU-HRSS, NTRU-Encrypt,RLizard) applying Toom-4 is most efficient. The second layer ends up in the same range of small n, where it is a close competition between applying Toom-3 or directly switching to recursive Karatsuba.

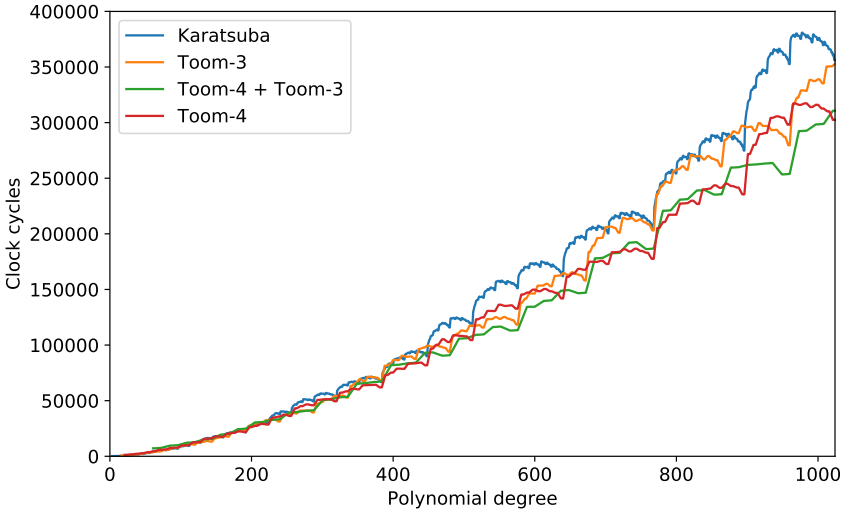


Figure 5.4: Runtime of different decomposition variants for large-degree multiplications.

Table 5.1: Benchmarks for small schoolbook multiplication routines. The cycle counts include an overhead of approximately 50 cycles for benchmarking.

n	**cycles**	**n**	**cycles**	**n**	**cycles**	**n**	**cycles**
1	56	13	232	25	926	37	1 965
2	59	14	252	26	1 057	38	1 966
3	69	15	341	27	1 057	39	1 963
4	74	16	343	28	1 168	40	1 965
5	85	17	467	29	1 167	41	2 294
6	92	18	466	30	1 170	42	2 588
7	107	19	508	31	1 264	43	2 595
8	114	20	510	32	1 266	44	2 594
9	131	21	626	33	1 431	45	2 824
10	140	22	626	34	1 547	46	2 825
11	168	23	670	35	1 546	47	2 822
12	177	24	672	36	1 549	48	2 824

Table 5.2: Benchmarks for polynomial multiplication excluding reduction. Fastest approach is highlighted in **bold**. The 'Toom-4 + Toom-3' and 'Toom-4' approaches are not applicable to all parameter sets, as q may be too large.

	approach	threshold	cycles	stack
Saber ($n = 256$, $q = 2^{13}$)	Karatsuba only	16	38 000	2 020
	Toom-3	11	39 043	3 480
	Toom-4	**16**	**36 274**	**3 800**
Kindi-256-3-4-2 ($n = 256$, $q = 2^{14}$)	**Karatsuba only**	**16**	**38 000**	**2 020**
	Toom-3	11	39 043	3 480
NTRU-HRSS ($n = 701$, $q = 2^{13}$)	Karatsuba only	11	202 889	5 676
	Toom-3	15	205 947	9 384
	Toom-4	**11**	**172 882**	**10 596**
NTRU-KEM-743 ($n = 743$, $q = 2^{11}$)	Karatsuba only	12	217 130	6 012
	Toom-3	16	211 588	9 920
	Toom-4	**12**	**186 639**	**11 208**
	Toom-4 + Toom-3	16	192 503	12 152
RLizard-1024 ($n = 1024$, $q = 2^{11}$)	Karatsuba only	16	356 046	8 188
	Toom-3	11	352 770	13 756
	Toom-4	**16**	**302 504**	**15 344**
	Toom-4 + Toom-3	11	310 712	16 816

5.3.2 Encapsulation and Decapsulation Results

In this section, we present our performance results for RLizard, Saber, Kindi, NTRUEncrypt, and NTRU-HRSS. All the software presented in this section started from the reference implementations submitted to NIST but went considerably further than just replacing the multiplication routines with the optimized routines described in Section 5.2. For Saber, we considered starting from the already optimized implementation by Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede [KBMSRV18], but achieved marginally better performance starting from the reference code. We start by describing the changes that apply to the reference implementations; some of these changes might be more generally advisable as updates to reference software.

Memory allocations. The reference implementations of Kindi, RLizard, and NTRUEncrypt make use of dynamic memory allocation on the heap. The RLizard implementation does not free all the allocated memory, which results in memory leaks; also it misinterprets the NIST API and assumes that the public key is always stored right behind the secret key. This may result in reads from uninitialized (or even unallocated) memory. Luckily none of the implementations *require* dynamically allocated memory; the size of all allocated memory is reasonably small and known at compile time. We eliminated all dynamic memory allocations and our software thus only relies on the stack to store temporary data. Our benchmarks show that this significantly improves performance.

Hashing. The five NIST candidates we optimize in this chapter make use of variants of SHA-3 and SHAKE [NIS15b] and of SHA-512 [NIS15a]. For SHA-3 and SHAKE we use the optimized assembly implementation from `pqm4` [KPR+], which makes use of the optimized Keccak-permutation from the Keccak Code Package [DHP+]. For SHA-512, we use a C implementation from SUPERCOP [BL].

Comparison to reference code. Table 5.3 presents performance results for the optimized implementations as well as the reference implementations with the modifications described above. For all schemes targeted in this chapter, we dramatically increase the performance; the improvements go up to a factor of 49 for the key generation of RLizard-1024. Since both Karatsuba and Toom-Cook require storing additional intermediate polynomials on the stack, we increase stack usage for all schemes except Kindi-256-3-4-2. The reference implementations of Kindi-256-3-4-2 already contained optimized polynomial multiplication methods, which were implemented in a stack-inefficient manner.

Side-channel resistance. While side-channel resistance was not a focus of this work, we ensured that our polynomial multiplication is protected against timing attacks. More specifically, in the multiplication routines, we avoid all data flow from secrets into branch conditions and into memory

Table 5.3: Benchmarks for reference implementations and optimized implementations using fastest multiplication approach. Reporting run-time (cycle count) and stack usage (bytes) for key generation (K), encapsulation (E), and decapsulation (D).

	implementation	**clock cycles**		**stack usage** [bytes]	
Saber	Reference	**K:**	6 530k	**K:**	12 616
		E:	8 684k	**E:**	14 896
		D:	10 581k	**D:**	15 992
	[KBMSRV18]	**K:**	1 147k	**K:**	13 883
		E:	1 444k	**E:**	16 667
		D:	1 543k	**D:**	17 763
	This work	**K:**	895k	**K:**	13 248
		E:	1 161k	**E:**	15 528
		D:	1 204k	**D:**	16 624
Kindi-256-3-4-2	Reference	**K:**	21 794k	**K:**	59 864
		E:	28 176k	**E:**	71 000
		D:	37 129k	**D:**	84 096
	This work	**K:**	969k	**K:**	44 264
		E:	1 320k	**E:**	55 392
		D:	1 517k	**D:**	64 376
NTRU-HRSS	Reference	**K:**	205 156k	**K:**	10 020
		E:	5 166k	**E:**	8 956
		D:	15 067k	**D:**	10 204
	This work	**K:**	145 963k	**K:**	23 396
		E:	404k	**E:**	19 492
		D:	819k	**D:**	22 140
NTRU-KEM-743	Reference	**K:**	59 815k	**K:**	14 148
		E:	7 540k	**E:**	13 372
		D:	14 229k	**D:**	18 036
	This work	**K:**	5 198k	**K:**	25 320
		E:	1 601k	**E:**	23 808
		D:	1 881k	**D:**	28 472
RLizard-1024	Reference	**K:**	26 423k	**K:**	4 272
		E:	32 156k	**E:**	10 532
		D:	53 181k	**D:**	12 636
	This work	**K:**	525k	**K:**	27 720
		E:	1 345k	**E:**	33 328
		D:	1 716k	**D:**	35 448

Table 5.4: Benchmarks on the Cortex-M4 for other KEMs submitted to NISTPQC project.

	implementation	clock cycles		stack usage	
R5ND_1PKEb	[SBGM+18]	**K:**	658k	**K:**	?
		E:	984k	**E:**	?
		D:	1 265k	**D:**	?
R5ND_3PKEb	[SBGM+18]	**K:**	1 032k	**K:**	?
		E:	1 510k	**E:**	?
		D:	1 913k	**D:**	?
NewHopeCCA1024	[KPR+, AJS16]	**K:**	1 244k	**K:**	11 152
		E:	1 963k	**E:**	17 448
		D:	1 979k	**D:**	19 648
Kyber768	[KPR+]	**K:**	1 200k	**K:**	10 544
		E:	1 446k	**E:**	13 720
		D:	1 477k	**D:**	14 880

addresses. The special multiplication routine in [SBGM+18] is less conservative and *does* use secret-dependent lookup indices with a reference to [ARM12] saying that the Cortex-M4 does not have internal data caches. However, it is not clear to us that really all Cortex-M4 cores do not have any data cache; [ARM12] states that the *"Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory. However, it is possible for a SoC design to integrate a system level cache."* Also, it *is* clear that some `Armv7E-M` processors (for example, the Arm Cortex-M7) have data caches and our multiplication code is timing-attack protected also on those devices.

Key-generation performance. The focus of this chapter is to improve performance of encapsulation and decapsulation. All KEMs considered in this chapter are CCA-secure, so the impact of a poor key-generation performance can in principle be minimized by caching ephemeral keys for some time. Such caching of ephemeral keys makes software more complex and in some cases also requires changes to higher-level protocols; we, therefore, believe that key-generation performance, also for CCA-secure KEMs, remains an important target of optimization. The key generation of RLizard, Saber, and Kindi is rather straightforwardly optimized by integrating our fast multiplication. The key generation of NTRUEncrypt and NTRU-HRSS also requires inversions, which we did not optimize in this chapter; we believe that further research into efficient inversions for those two schemes will significantly improve their key-generation performance.

Comparison to previous results. To the best of our knowledge, Saber is

the only scheme of those considered in this chapter that has been optimized for the Arm Cortex-M family in previous work [KBMSRV18]. Table 5.4 contains the performance result on the same platform as ours. Our optimized implementation outperforms the CHES 2018 implementation by 22% for key generation, 20% for encapsulation, and 22% for decapsulation. Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede report 65 459 clock cycles for their optimized 256-coefficient polynomial multiplication, but we note that their polynomial multiplication includes the reduction. Including the reduction, our multiplication requires 38 215 clock cycles, which is 42% faster. On a more granular level, they claim 587 cycles for 16-coefficient schoolbook multiplication, while we require only 343 cycles (see Table 5.1; this includes approximately 50 cycles of benchmarking overhead).

Several other NIST candidates have been evaluated on the Cortex-M4 family. We also list the performance results in Table 5.4 for comparison. Most recently, record-setting results were published for Round5[2] on Cortex-M4 [SBGM+18]. The fastest scheme described in our work, targeting NIST security category 1, NTRU-HRSS, is 59% faster for encapsulation and 35% faster for decapsulation compared to the corresponding CCA variant of Round5 at the same security level. The key generation of NTRU-HRSS is considerably slower, but its inversion is not optimized yet. The fastest scheme implementation described here that targets NIST security category 3, Saber, is 13% faster for key generation, 23% faster for encapsulation, and 37% faster for decapsulation There are also optimized implementations for NewHopeCCA1024 [KPR+, AJS16] and Kyber768 [KPR+]. Both implementations are outperformed by NTRU-HRSS and Saber.

5.3.3 Profiling of Optimized Implementations

The speed-up achieved by optimizing polynomial multiplication clearly shows that it vastly dominates the runtime of reference implementations. Having replaced this core arithmetic operation with highly optimized assembly, we analyze how much time the optimized implementations still spend in non-optimized code to capture how much performance could still be gained by hand-optimizing scheme-specific procedures. We achieve this by measuring the clock cycles spent in polynomial multiplication, hashing, and random number generation. Table 5.5 shows that still a considerable proportion of encapsulation and decapsulation is spent in polynomial multiplication. However, cycles consumed by hashing and randomness generation become more prominent. In the following, we briefly discuss these results and emphasize how one could further speed up those schemes.

Hashing. For encapsulation, hashing (SHA-3 and SHA-2) dominates the run-time of Kindi-256-3-4-2, NTRU-KEM-743, and Saber. We have replaced

[2] R5ND_{1,3,5}PKEb are the CCA-variants of Round5, whereas R5ND_{1,3,5}KEMb are CPA-secure.

Table 5.5: Time spent in multiplication, hashing, and sampling randomness.

scheme		total	polymul		hashing		random
Saber	*K:*	895k	327k	(37%)	475k	(53%)	2.0k
	E:	1 161k	435k	(38%)	615k	(53%)	0.6k
	D:	1 204k	544k	(45%)	500k	(42%)	0
Kindi-256-3-4-2	*K:*	969k	342k	(35%)	409k	(42%)	1.2k
	E:	1 320k	456k	(35%)	604k	(46%)	0.6k
	D:	1 517k	570k	(38%)	603k	(40%)	0
NTRU-HRSS	*K:*	145 963k	1 556k	(1%)	80k	(<1%)	0.6k
	E:	404k	173k	(43%)	107k	(26%)	0.6k
	D:	819k	519k	(63%)	67k	(8%)	0
NTRU-KEM-743	*K:*	5 198k	1 680k	(32%)	0		85k
	E:	1 601k	187k	(12%)	1 171k	(73%)	46k
	D:	1 881k	373k	(20%)	1 172k	(63%)	0
RLizard-1024	*K:*	525k	303k	(58%)	0		123k
	E:	1 345k	605k	(45%)	628k	(47%)	2.2k
	D:	1 716k	908k	(53%)	628k	(36%)	0

these primitives with the fastest implementations available. Still, all schemes spend a substantial number of clock cycles computing hashes. This is partly due to the Fujisaki-Okamoto transformation required to achieve CCA security. Further hash function calls are required to sample pseudo-random numbers from a seed, which most schemes implement using the SHAKE XOF. Having a hardware accelerator for these hash functions would highly improve the performance of all of the examined schemes. While Arm Cortex-M4 platforms with SHA-2 hardware support exist, there are (at the time of writing) none available which have SHA-3 hardware support.

Randomness generation. Kindi-256-3-4-2, NTRU-HRSS, and Saber do not make use of `randombytes` extensively, but sample a small seed and then expand this using SHAKE. RLizard-1024 and NTRU-KEM-743 directly sample their randomness using `randombytes`. As we implement `randombytes` using the hardware RNG on the STM32F4Discovery, it is more efficient than using SHAKE to expand a seed. There are, however, important caveats to consider when only using the hardware number generator. It is unclear what the cryptographic properties of such an RNG are, and how this affects the security of the various schemes, in particular since most reveal randomness as part of the CCA transform.

Chapter 6

NTT Multiplication for NTT-unfriendly Rings

This chapter is based on work published in

> 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. `https://eprint.iacr.org/2020/1397`

It studies the application of NTT-based multiplication to lattice-based KEMs using polynomial rings which were not specifically chosen to support NTT multiplication. As such this work targets some schemes that were also covered in Chapter 5, but results in much faster implementations.

There seems to be a common conception that schemes that were not specifically designed to benefit from NTT-based multiplication by using an *NTT-friendly* ring cannot be efficiently implemented using NTTs and, hence, one has to fall back to other multiplication algorithms like Karatsuba multiplication [KO63] or Toom–Cook multiplication [Too63, Coo66]. Among the NISTPQC finalists, this applies to two schemes: Saber [DKRV17] and NTRU [ZCH+19]. Both use a power-of-two modulus which is inherently incompatible with straightforward NTTs. Previous implementations of Saber and NTRU use a combination of Toom-4 and Karatsuba to implement efficient polynomial arithmetic. However, as we show in this work, it is still possible to use NTTs to implement their underlying polynomial arithmetic and obtain superior performance compared to the state-of-the-art implementations both on the Arm Cortex-M4 and AVX2.

Leaving the performance aspect aside, it is also interesting to be able to implement all lattice-based schemes with the same algorithms from an ease of implementation point of view. Furthermore, this way all schemes can benefit from potential future hardware support for computing NTTs. Because of these reasons we think that even a small increase in runtime may be acceptable when using NTT-based multiplication instead of other methods.

The Chinese Association for Cryptologic Research (CACR) sponsored a competition similar to that of NIST between 2018 and 2019 [CAC19]. All three First Class Award winners were small lattice-based systems. Two of them, styled Aigis-ENC and Aigis-Sign, resemble Kyber and Dilithium in their design (see [ZYF+20], where the authors detail their deviations from Kyber and Dilithium). The other, LAC [LLJ+17], has a very small prime modulus ($q = 251$) which is not suited to NTTs, and the designers suggest a sparse multiplication technique instead. We show that NTTs can be used to obtain performance superior to all previous implementations.

Contribution. We show how NTTs can be used to obtain efficient polynomial arithmetic in finite fields modulo a power of two. We present new implementations of Saber, LAC, and NTRU targeting the Arm Cortex-M4 and AVX2 which are faster than any implementations described in the literature for the majority of parameter sets. Only for `ntruhps2048509` we were unable to obtain a speed-up on AVX2. Interestingly, our two platforms require different multiplication strategies due to limitations of the available multiplication instructions.

Code. Our implementations of Saber, LAC, and NTRU are Open Source and are available at `https://github.com/ntt-polymul/ntt-polymul`. All source code related to this thesis is also available in a single archive. See Appendix A.

Related Work. Concurrent work [FSS20] presents a Saber implementation of a similar NTT-based approach targeting a RISC-V core with a tightly coupled hardware accelerator but did not obtain better performance than their Toom–Cook implementation.

Structure of this Chapter. Section 6.1 describes Saber, LAC, and NTRU and the background of the techniques required to implement polynomial arithmetic using NTTs for each. Section 6.2 presents the implementation details on the Cortex-M4. Section 6.3 presents the implementation details for AVX2 on Skylake. In Section 6.4, we present the performance results for Saber, LAC, and NTRU on our target platforms.

Algorithm 41 LAC Key Generation

Output: $pk = (\text{seed}_a, b), sk = (s)$

1: $\text{seed}_a \leftarrow \texttt{Sample}_U()$
2: $a \in R_q \leftarrow \texttt{Expand}(\text{seed}_a)$
3: $s, e \in R_q^{(h)} \leftarrow \texttt{Sample}_B()$
4: $b \leftarrow as + e$

Algorithm 42 LAC CPA Encryption

Input: $m, pk = (\text{seed}_a, b)$
Output: $ct = (c_1, c_2)$

1: $a \in R_q \leftarrow \texttt{Expand}(\text{seed}_a)$
2: $\hat{m} = \mathsf{ECCEnc}(m)$
3: $r, e_1 \in R_q \leftarrow \texttt{Sample}_B()$
4: $e_2 \in R_q \leftarrow \texttt{Sample}_{B'}()$
5: $c_1 \leftarrow ar + e_1$
6: $c_2 \leftarrow (br)_{l_v} + e_2 + \left\lfloor \frac{q}{2} \right\rceil \hat{m}$

6.1 Preliminaries

This section is organized as follows: First, we introduce the cryptographic scheme LAC (Section 6.1.1); for Saber and NTRU refer to the previous chapter (Section 5.1). Second, Section 6.1.2 introduces the NTT techniques that can be used to implement polynomial arithmetic.

6.1.1 LAC

LAC [LLJ+17] is a lattice-based key-encapsulation mechanism based on the Ring Learning with Errors (RLWE) problem. The polynomial ring used in LAC is $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $q = 251$ and $n = 512$ for `LAC-128` and $n = 1024$ for `LAC-192` and `LAC-256`. Like most other lattice-based schemes, LAC constructs a CCA-secure KEM from a CPA-secure PKE.

Algorithm 41, Algorithm 42, and Algorithm 43 depict the CPA-secure key generation, encryption, and decryption, respectively. $\texttt{Sample}_U$ refers to sampling from a uniform distribution and $\texttt{Sample}_B$ refers to sampling from a fixed-weight ternary distribution. $\texttt{Sample}_{B'}$ refers to sampling from a ternary distribution. `Expand` expands a seed to a uniform matrix of polynomials. $(\cdot)_{l_v}$ means to take the first l_v coefficients of a polynomial as a vector. We omit the CCA variants for brevity and refer the reader to the specification for the corresponding CCA transformation. LAC's major operations are multiplications $(as, ar, br, c_1 s)$.

Algorithm 43 LAC CPA Decryption

Input: $ct = (c_1, c_2), sk = (s)$
Output: $m = \mathsf{ECCDec}(\hat{m})$

1: $\tilde{m} \leftarrow c_2 - (c_1 s)_{l_v}$
2: $\hat{m} \leftarrow \mathtt{Round}(\tilde{m})$

Table 6.1: LAC Parameters

name	n	l_v	B	B'
LAC-128	512	511	$\left(\frac{1}{4}; \frac{1}{2}; \frac{1}{4}\right)^n$	$\left(\frac{1}{4}; \frac{1}{2}; \frac{1}{4}\right)^{l_v}$
LAC-192	1024	511	$\left(\frac{1}{8}; \frac{3}{4}; \frac{1}{8}\right)^n$	$\left(\frac{1}{8}; \frac{3}{4}; \frac{1}{8}\right)^{l_v}$
LAC-256	1024	1023	$\left(\frac{1}{4}; \frac{1}{2}; \frac{1}{4}\right)^n$	$\left(\frac{1}{4}; \frac{1}{2}; \frac{1}{4}\right)^{l_v}$

Parameters. LAC specifies the three parameter sets LAC-128, LAC-192, and LAC-256 targeting the NIST security levels one, three, and five, respectively. The parameters are summarized in Table 6.1.

CCA Transform. To achieve CCA security, LAC is using a variant of the FO transform due to Hofheinz, Hövelmanns, and Kiltz [HHK17], similar to Saber. For technical details on the FO transform, refer to the specification [LLJ+17].

6.1.2 FFT-based Polynomial Multiplications and NTT

In NTRU, LAC, and Saber, we need to multiply in the following rings: $\mathbb{Z}_{8192}[x]/\left(x^{256}+1\right)$, $\mathbb{Z}_{2048}[x]/\left(x^{509}-1\right)$, $\mathbb{Z}_{251}[x]/\left(x^{512}+1\right)$, $\mathbb{Z}_{2048}[x]/\left(x^{677}-1\right)$, $\mathbb{Z}_{8192}[x]/\left(x^{701}-1\right)$, $\mathbb{Z}_{4096}[x]/\left(x^{821}-1\right)$, and $\mathbb{Z}_{251}[x]/\left(x^{1024}+1\right)$.

In Saber, we actually need more: a matrix-vector product and an inner product based on that ring multiplication. Below, we describe tools to construct those multiplications.

We make use of incomplete NTTs (see Section 2.2.7) and Good's trick (Section 2.2.8). The NTTs are implemented using radix-2 FFTs (see Section 2.2.5) and mixed-radix FFTs (see Section 2.2.6). However, the above rings are not immediately suitable for NTT-based polynomial multiplication. We instead propose to lift ring elements to a different ring that is suitable for FFTs as described in the following section.

NTTs with Modulus not of the form $2^k p' + 1$. Suppose we have a convolution modulo $x^n - 1$ modulo q, where $n \nmid (q-1)$. We can express the polynomials with coefficients in $[-\frac{q}{2}; \frac{q}{2})$ and compute the convolution as a polynomial with integer coefficients. The absolute magnitude of the resulting coefficients would be at most $nq^2/4$. Therefore, if we find a prime

$p > nq^2/2$ such that $n \mid (p-1)$, and compute the multiplication mod p (which we can compute using NTTs of length n mod p), then the result must be correct as a polynomial with integer coefficients, and then we can recover our correct result modulo q.

The procedure is quite similar if it is a different kind of convolution or another product. In the case of our applications (Saber and NTRU), one of the multiplicands is usually "small" so that we can use an even smaller prime.

Multiple Moduli and the Explicit CRT. Again, suppose we have a convolution modulo $x^n - 1$ modulo q, where $n \nmid (q-1)$. A different possibility is to take various NTT-friendly primes p_i whose *product* P is sufficiently large (usually $> nq^2/2$). Clearly computing the multiplication mod P must return the correct product as a polynomial with integer coefficients. This we can do by computing the product modulo each p_i using NTTs. There are at least two methods to put the pieces together modulo P, from which we can compute our correct results. One is via the Explicit Chinese Remainder Theorem [BS07]. The other is the following approach:

Theorem 1 *Let $p_i > 0$ be odd, pairwise co-prime ($\gcd(p_i, p_j) = 1$ for $1 \le i < j \le s$). An* explicit *solution u of $u \equiv u_i \pmod{p_i}$, $i = 1 \ldots s$, where $|u_i| < p_i/2$, where $|u| < P/2 = \prod_{i=1}^{s} p_i/2$, is given by*

$$\begin{cases} y_1 &= u_1 \\ y_2 &= y_1 + ((u_2 - y_1)m_2 \bmod^{\pm} p_2)\, p_1 \\ y_3 &= y_2 + ((u_3 - y_2)m_3 \bmod^{\pm} p_3)\, p_1 p_2 \\ \vdots & \vdots \\ u = y_s &= y_{s-1} + ((u_s - y_{s-1})m_s \bmod^{\pm} p_s)\, p_1 \cdots p_{s-1} \end{cases}$$

where each $m_i := (p_1 \cdots p_{i-1})^{-1} \bmod^{\pm} p_i$.

The theorem is also true for a non-centered mod and is faster than [BS07] for small s.

6.2 NTTs on the Cortex-M4

On Cortex-M4, we commonly compute three layers of radix-2 NTTs at a time similarly as [ACC+21b]. We provide details specific to the schemes in the following subsections.

6.2.1 Saber

For Saber, we replace the polynomial multiplications in the subroutines `InnerProd` and `MatrixVectorMul` using the negacyclic NTT trick to eliminate *all* Toom-4 multiplications in Saber. In the interest of brevity, we

only detail `MatrixVectorMul` (which takes most of the time) that multiplies an $l \times l$ matrix with an $l \times 1$ vector, where each component is an element of $\mathbb{Z}_q[x]/(x^{256}+1)$. The design of Saber provides additional incentives to use NTTs because the matrix-to-vector product is turned into a matrix-to-vector point-multiplication in the NTT domain. More concretely, we do not merely save the difference in cycles between Toom-4 and NTT-based degree-255 polynomial multiplications, because to compute the l^2 multiplications in `MatrixVectorMul`, we only need to compute $l^2 + l$ NTTs and l NTTs^{-1} instead of $2l^2$ NTTs and l^2 NTTs^{-1} as normally might be expected.

Our NTT-based `MatrixVectorMul`, therefore, proceeds as follows: compute the size-256 (incomplete) negacyclic NTT for each component in the matrix and the vector, multiply the matrix by the vector, accumulate the result to a vector, and then compute the NTTs^{-1} for each component.

Choosing the best incomplete NTT. When using incomplete NTTs we need to choose the point at which we stop doing NTT butterfly operations and simply multiply the polynomials using schoolbook multiplication. The obvious choices are eight layers of NTTs, seven layers of NTTs followed by 2×2 schoolbooks, six layers of NTTs followed by 4×4 schoolbooks, and five layers of NTTs followed by 8×8 schoolbooks. First, we compare the behavior of incomplete NTTs. On a Cortex-M4, among the 14 general-purpose registers, we need one register for loading coefficients, one register for loading the twiddle factors ζ, two registers for constants used in Montgomery multiplication, and two registers as temporary storage for Montgomery multiplication in the schoolbook multiplication.

There are only eight remaining registers that can be used to compute at most three layers of NTTs without incurring overhead.

Computing five layers of NTTs would not achieve the economical use of registers, since we can compute an additional layer without spilling the registers, i.e., with minimal cost. Computing seven layers of NTTs would involve a lot of `vmov`s because of the lack of registers. For Saber, we achieve the best performance when doing six layers of NTTs. This can be explained by comparing 4×4 schoolbook multiplication and size-4 NTTs. For simplicity, we will focus on an l-dimensional matrix-to-vector product in which each component is a four-coefficient polynomial. A 4×4 schoolbook multiplication requires seven `smull`s, 12 `smlal`s, and seven Montgomery reductions. For accumulation, each l-dimensional row-column inner product requires $4l - 4$ `add`s and $4l - 4$ `vmov`s for temporary storage. Therefore, $41l^2 - 8l$ cycles are required for the 4×4 schoolbook approach. To use a size-4 NTT trick, we calculate the size-4 NTT of each component, multiply components by components with point-multiplication, accumulate to a vector, and finally, compute the size-4 NTT^{-1} of each component. Each size-4 NTT requires 4 Montgomery multiplications and four add-sub pairs. Since only 14 registers are available, we need to `vmov` $4 \cdot (2 \cdot (l-1) + l) \cdot l = 12l^2 - 8l$ times to store intermediate values for accumulation. If the NTT trick is adopted, the

matrix-to-vector product would require $20l^2 + 40l$ cycles for $l^2 + l$ NTTs and l NTTs^{-1}, l^2 Montgomery multiplications, $12l^2 - 8l$ `vmov`s, and $4l^2 - 4l$ `add`s, resulting in $39l^2 + 28l$ cycles. We have $41l^2 - 8l < 39l^2 + 28l$ for $l < 18$.

Hence, we compute incomplete (six layer) NTTs followed by 4×4 schoolbook as it gives the best performance for Saber. To compute $a_0 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_2 \pmod{q'}$ using Montgomery reductions, we only need one `smull`, two `smlal`, and one Montgomery reduction instead of computing three multiplications, each followed by a Montgomery reduction, adding the results together, and then reducing modulo q' again. Furthermore, this idea also applies where each $a_i \cdot b_i$ is a degree-3 schoolbook.

Better Accumulation For Schoolbook Multiplication. There is an even better approach to matrix-to-vector products utilizing the commutativity of instructions. All `add`s and some Montgomery reductions can be removed at the cost of some additional `vmov`s. Consider one inner product $\mathbf{h} = \sum_{i=0}^{3} \mathbf{p}_i \star \mathbf{q}_i$ where $\star$ is multiplication $\pmod{z^4 - \zeta}$. Now $[z^0]\mathbf{h} = \sum_i \big(\mathbf{p}_{i0}\mathbf{q}_{i0} + \zeta(\mathbf{p}_{i1}\mathbf{q}_{i3} + \mathbf{p}_{i2}\mathbf{q}_{i2} + \mathbf{p}_{i3}\mathbf{q}_{i1})\big)$ is its constant term.[1] So we can compute the 64-bit value of $[z^0]\mathbf{h}$ and then reduce it to 32-bit using Montgomery reduction, wherein all the `add`s can be absorbed (changing some `smull` into `smlal`). To summarize, we save $4l^2 - 4l$ `add`s and $4l^2 - 4l$ Montgomery reductions (each of which takes two cycles) at the cost of $8l^2 - 8l$ `vmov`s and, therefore, the cycle count becomes $37l^2 - 4l$, which is smaller than $39l^2 + 28l$ for all l. We find that the above approach is hard to beat regardless of l.

Our Optimized Negacyclic NTT Trick. Since incomplete size-256 negacyclic NTTs are computed, we choose prime $q' = 25166081 = 196610 \cdot 128 + 1$ for Saber and Firesaber, and prime $q' = 20972417 = 163847 \cdot 128 + 1$ for Lightsaber. We compute NTTs with six layers of radix-2 NTTs (CT butterflies), where the first three layers are merged and the following three layers are merged, then compute schoolbook-and-accumulate with the above strategy, and finally compute incomplete size-256 negacyclic NTTs^{-1} using GS butterflies with the same 3-layer merge.

6.2.2 NTRU

In this section, we go into implementation details for polynomial multiplication in NTRU on Cortex-M4. We are targeting the first two `poly_Rq_mul`s and the first `poly_Sq_mul` in key generation, the `poly_Rq_mul` in encryption, and the first `poly_Rq_mul` in decryption. While implementing polynomial multiplication for each parameter set, we optimized the code in various aspects. Some ideas work for all parameter sets, and some are only suitable for a particular one. The core ideas are simple: manipulate registers wisely, compute small convolutions with schoolbook, and change the domain only

[1] Combinatorially it is customary to write $[x^i]f$ for the coefficient of x^i in f.

(a) NTT tricks for NTRU parameter sets.

Parameter sets	$\mathrm{NTT}_{\mathtt{N}}$	q'	Strategy
ntruhps4096821	$1728 = 9 \cdot 64 \cdot 3$	3365569	Mixed-radix (CT+GS)
ntruhrss701	$1536 = 512 \cdot 3$	5747201	Good's (CT+CT)
ntruhps2048677	$1536 = 512 \cdot 3$	1389569	Good's (CT+CT)
ntruhps2048509	$1024 = 256 \cdot 4$	1043969	Radix-2 (CT+GS)

(b) Layers of NTTs for each set of parameter.

	NTT	baseMul	NTT^{-1}
ntruhps4096821	2-layer-radix-3 $+2 \times$ 3-layer-radix-2	3×3	$2 \times$ 3-layer-radix-2 +2-layer-radix-3
ntruhrss701 ntruhps2048677	$3 \times$ 3-layer-radix-2	3×3	$3 \times$ 3-layer-radix-2
ntruhps2048509	$2 \times$ 4-layer-radix-2	4×4	$2 \times$ 3-layer-radix-2

Table 6.2: Overview of NTTs for NTRU on Cortex-M4

when needed. We summarize the tricks used for each parameter set in Table 6.2.

Layers of NTT. As usual, several layers of NTTs are computed at a time to avoid load-stores and to use the registers economically. On Cortex-M4, since only 14 general-purpose registers are available, we compute three layers of radix-2 NTTs (and two layers of radix-3 NTTs) at a time. For ntruhps2048509, we employ a seemingly strange alternative, computing *four* layers of radix-2 NTTs at a time, to set up a better foundation for polynomial multiplication. This results in a slightly faster implementation for ntruhps2048509 compared to the Toom-4 approach.

Tricks for commutative operations. Recall that when computing an NTT, we must cancel out the scaling factor $\mathrm{NTT}_{\mathtt{N}}$. We can halve the number of Montgomery-multiplications by $\mathrm{NTT}_{\mathtt{N}}^{-1}\mathtt{R}^2 \bmod^{\pm} q$ by first reducing modulo the polynomial modulus and then performing the multiplication. The same idea also applies to the operations of reducing the coefficient from $\mathbb{Z}_{q'}$ to $\mathbb{Z}_q$ and packing two coefficients into one register. Because they commute, we pack two coefficients *and then* perform an `and` with $(q-1)\|(q-1)$.

ntruhps4096821. Algorithm 44 depicts the NTT for ntruhps4096821. We compute incomplete mixed-radix size-1728 NTTs for each polynomial by splitting down to $x_{i,j}^3 - \zeta_{i,j}$, multiplying degree-2 polynomials with schoolbook, deriving incomplete mixed-radix NTTs^{-1}, and then reducing the coefficient ring to $\mathbb{Z}_q$. For incomplete size-1728 NTTs, we first compute size-9 NTTs with two radix-3 NTTs for each 9-set distanced apart by 192 units. Next, for each consecutive 192 coefficients, we compute size-64 NTTs with six layers of radix-2 NTTs for each 64-set distanced apart by three units,

Algorithm 44 Incomplete mixed-radix size-1728 NTT for `ntruhps4096821`

Representing $\begin{cases} \texttt{src1}[i] \text{ with } \texttt{ntt1}[i/192]\,[(i \bmod 192)/3]\,[(i \bmod 3)] \\ \texttt{src2}[i] \text{ with } \texttt{ntt2}[i/192]\,[(i \bmod 192)/3]\,[(i \bmod 3)] \end{cases}$.

1: For each j, k, compute $\begin{cases} \mathrm{NTT}_9(\texttt{ntt1[0-8]}\,[j]\,[k]) \\ \mathrm{NTT}_9(\texttt{ntt2[0-8]}\,[j]\,[k]) \end{cases}$.

2: For each i, k, compute $\begin{cases} \mathrm{NTT}_{64:\zeta_{i,0}}(\texttt{ntt1}[i]\,\texttt{[0-63]}\,[k]) \\ \mathrm{NTT}_{64:\zeta_{i,0}}(\texttt{ntt2}[i]\,\texttt{[0-63]}\,[k]) \end{cases}$.

3: For each i, j, compute $\texttt{nttout}[i]\,[j]\,\texttt{[0-2]} = \texttt{ntt1}[i]\,[j]\,\texttt{[0-2]} \star \texttt{ntt2}[i]\,[j]\,\texttt{[0-2]} \bmod (x^3 - \zeta_{i,j})$

4: For each i, k, compute $\mathrm{NTT}^{-1}_{64:\zeta_{i,0}}(\texttt{nttout}[i]\,\texttt{[0-63]}\,[k])$.

5: For each j, k, compute $\mathrm{NTT}_9^{-1}(\texttt{nttout[0-8]}\,[j]\,[k])$.

6: Compute $\texttt{des[0-1727]} = \texttt{final\_stage}(\texttt{nttout[0-8]}\,\texttt{[0-63]}\,\texttt{[0-2]})$.

leaving degree-2 polynomials. Among 9 sets of 192-coefficient, standard size-64 NTTs are computed for the first 192-coefficient and twisted size-64 NTTs are computed for the rest. The incomplete size-1728 NTT^{-1} is computed in a reversed manner. For the final stage, we employ all the ideas mentioned in the previous subheading – taking quotient before Montgomery-multiplying $(\mathtt{R})^2\mathrm{NTT}_{\mathtt{N}}^{-1} \bmod q'$ and pack two coefficients before the `and`. For merging layers, the two layers of radix-3 NTTs are merged, the first three layers of radix-2 NTTs are merged, the following three layers of radix-2 NTTs are merged, and the NTTs^{-1} are merged in the same manner.

`ntruhrss701` and `ntruhps2048677`. Algorithm 45 shows the NTT used for `ntruhrss701` and `ntruhps2048677`. We use Good's trick for both. Our approach is almost the same as [ACC+21b], with a slightly faster final stage. This is because $\pmod{2^k}$ and $\pmod{(x^n - 1)}$ are cheaper. We employ Good's permutation of size 3×2^9 for the size-1536 NTT. The algorithm goes in the following order: compute three size-512 NTTs (CT butterflies), each for 512 contiguous entries, compute 3×3 convolutions, where coefficients are distanced apart by 512 units, invert size-512 NTTs (CT butterflies), and a final stage. This last stage consists of: inverting Good's permutation, taking the remainder $\bmod(x^n - 1)$, Montgomery-multiplication by $(\mathtt{R})^2\mathrm{NTT}_{\mathtt{N}}^{-1} \bmod q'$, packing two coefficients into one register, and reducing to coefficient ring $\mathbb{Z}_q$. We implement the NTTs^{-1} using CT butterflies because we need fewer reductions to avoid overflows. As mentioned above, we do $\bmod(x^n - 1)$ first so we save half the Montgomery-multiplications by $(\mathtt{R})^2\mathrm{NTT}_{\mathtt{N}}^{-1} \bmod q'$.

`ntruhps2048509`. For `ntruhps2048509`, we merge our NTT layers differently to provide a better framework for polynomial multiplication. See Algorithm 46 for the details. We perform two sets of *four-layer* NTTs

Algorithm 45 Good's trick of size-1536 NTT for `ntruhrss701` and `ntruhps2048677`

1: Compute $\begin{cases} \texttt{ntt1[0-2][0-511]} = (\mathrm{NTT}_{512:0-2}^{\otimes 3} \circ \mathrm{Good}_{3\times 512})(\texttt{src1[0-1535]}) \\ \texttt{ntt2[0-2][0-511]} = (\mathrm{NTT}_{512:0-2}^{\otimes 3} \circ \mathrm{Good}_{3\times 512})(\texttt{src2[0-1535]}) \end{cases}$

2: For each i, compute $\begin{cases} \mathrm{NTT}_{512:3-8}(\texttt{ntt1}[i]\texttt{[0-511]}) \\ \mathrm{NTT}_{512:3-8}(\texttt{ntt2}[i]\texttt{[0-511]}) \end{cases}$

3: For each j, compute $\texttt{nttout[0-2]}[j] =$
$$\texttt{ntt1[0-2]}[j] \star \texttt{ntt2[0-2]}[j] \bmod (\omega^3 - 1)$$

4: For each i, compute $\mathrm{NTT}_{512}^{-1}(\texttt{nttout}[i]\texttt{[0-511]})$.

5: Compute $\texttt{des[0-1535]} = \texttt{final\_stage}(\texttt{nttout[0-2][0-511]})$.

Algorithm 46 Incomplete size-1024 NTT for `ntruhps2048509`

Representing $\begin{cases} \texttt{src1}[i] \text{ with } \texttt{ntt1}[i/4]\texttt{[}i \bmod 4\texttt{]} \\ \texttt{src2}[i] \text{ with } \texttt{ntt2}[i/4]\texttt{[}i \bmod 4\texttt{]} \end{cases}$

1: For each j, compute $\begin{cases} \mathrm{NTT}_{256}(\texttt{ntt1[0-255]}[j]) \\ \mathrm{NTT}_{256}(\texttt{ntt2[0-255]}[j]) \end{cases}$

2: For each i, compute $\texttt{nttout}[i]\texttt{[0-3]} =$
$$\texttt{ntt1}[i]\texttt{[0-3]} \star \texttt{ntt2}[i]\texttt{[0-3]} \bmod (x^4 - \zeta_i)$$

3: For each j, compute $\mathrm{NTT}_{256:7-2}^{-1}(\texttt{nttout[0-255]}[j])$.

4: Compute $\texttt{des[0-1023]} = \texttt{final\_stage}(\texttt{nttout[0-255][0-3]})$.

(CT butterflies) for incomplete size-1024 NTTs, perform each 4-coefficient (modulo a degree-3 polynomial) multiplication with schoolbook, do two sets of 3-layer NTTs^{-1} (GS butterflies), and a final stage. Here GS butterflies make for an easier final stage comprising the following operations: two layers of NTTs^{-1}, taking mod$(x^n - 1)$, Montgomery-multiplication by $(\mathtt{R})^2\mathrm{NTT}_{\mathtt{N}}^{-1} \bmod q'$, packing two coefficients into one register, and reducing to coefficient ring $\mathbb{Z}_q$. This approach saves one layer of load-stores.

6.2.3 LAC

For `LAC-128`, `LAC-192`, and `LAC-256`, we focus on big-by-small polynomial multiplications where the "small" polynomials have coefficients in $\{0, \pm 1\}$.

NTT trick for LAC. We employ the negacyclic NTT trick on the rings $Z_q[x]/(x^{512}+1)$, $Z_q[x]/(x^{1024}+1)$, $Z_q[x]/(x^{1024}+1)$ for `LAC-128`, `LAC-192`, and `LAC-256`, respectively. Our approach for `LAC-128` proceeds as follows: compute the negacyclic size-512 NTTs of polynomials, do point-by-point multiplications, and finally, compute the size-512 NTT^{-1}. Our approach for `LAC-192` and `LAC-256` proceeds as follows: derive incomplete negacyclic size-1024 NTT, compute 2×2 schoolbooks, and invert the NTT.

(a) NTT tricks for LAC parameter sets.

Parameter sets	NTT_{N}	q'	Strategy
LAC-128	512	133121	Complete NTT (CT+GS)
LAC-192 LAC-256	1024	270337	Incomplete NTT (CT+GS)

(b) Layers of NTTs for each set of parameter.

	NTT	baseMul	NTT^{-1}
LAC-128	3×3-layer-radix-2	1×1	3×3-layer-radix-2
LAC-192 LAC-256	3×3-layer-radix-2	2×2	3×3-layer-radix-2

Table 6.3: Overview of NTTs for LAC on Cortex-M4

6.3 Vectorized NTT on AVX2

For fast NTT-based polynomial multiplication on current x86 processors from Intel and AMD, it is necessary to use a vectorized implementation of the NTT. These processors support the AVX2 instruction set, offering a large number of instructions that operate on 16 vector registers, each of length 256 bit.

6.3.1 Fast Mulmods

The first obstacle towards fast vectorization of the NTT is the problem of efficiently multiplying many coefficients modulo a small prime q. The standard way to compute modular products is to first compute the double-length products over $\mathbb{Z}$, and then reduce these intermediate results modulo q. In a vectorized implementation, in order to achieve the highest possible throughput, one wants to pack as many coefficients as possible in a vector register. But double-length intermediate products mean it is only possible to achieve half the density compared to packing only mod-q reduced integers. This effectively reduces the speed of the implementation by a factor of two. Note that this is not a problem when computing products modulo a power of two as in other polynomial multiplication implementations for Saber or NTRU that directly operate over the respective polynomial rings. There the arithmetic in modern CPUs automatically takes care of the modular reduction. To overcome this obstacle we use the modified Montgomery reduction algorithm from [Sei18] together with the improvement from [LS19]. Here the modular multiplications are computed from separate intermediate low and high half-products. When using the AVX2 instruction set, this approach is most efficient for 16-bit primes q. The reason is that there is a specific high-half-only product instruction `vpmulhw` for packed 16-bit integers that

Algorithm 47 Multiplication modulo 16-bit q

Require: $-2^{15} \leq a < 2^{15}$, $\frac{q-1}{2} \leq b \leq \frac{q-1}{2}$, $b' = bq^{-1} \bmod 2^{16}$
Ensure: $r \equiv 2^{16}ab \pmod{q}$
1: $t_1 \leftarrow \left\lfloor \frac{ab}{2^{16}} \right\rfloor$ ▷ signed high product
2: $t_0 \leftarrow ab' \bmod 2^{16}$ ▷ signed low product
3: $t_0 \leftarrow \left\lfloor \frac{t_0 q}{2^{16}} \right\rfloor$ ▷ signed high product
4: $r \leftarrow (t_1 - t_0) \bmod 2^{16}$

does not have an equivalent instruction for packed 32-bit integers. Therefore, unlike on the Cortex-M4, we use NTTs modulo 16-bit primes q on AVX2. Then we need to use a multi-modular approach and compute the polynomial products modulo two such primes so that we are able to correctly lift the results to $\mathbb{Z}$ with the help of the Chinese remainder theorem (Theorem 1). The additional polynomial product modulo a second prime involving three NTT computations and a base product computation does not result in reduced speed, because this loss of a factor of two is completely compensated for by twice the throughput from packing 16-bit integers instead of 32-bit integers.

We state the modular multiplication algorithm in Algorithm 47. As input it gets a 16-bit integer a, and a mod-q reduced integer b together with the pre-computed $b' = bq^{-1} \bmod 2^{16}$. The algorithm then outputs a representative modulo q for the scaled product $ab2^{16} \bmod q$. The second multiplicand b is always a fixed constant in the NTT and hence b and the corresponding element b' can easily be pre-computed. The scaling factor 2^{16} is handled as usual by pre-computing b and b' with an additional factor of 2^{-16}.

6.3.2 Choice of Transforms

We considered several choices of transforms. For Saber with its NTT-friendly polynomial modulus $x^{256} + 1$, we compute the negacyclic length-256 transforms modulo $x^{256} + 1$ as we do on the Cortex-M4. For performing only a single polynomial multiplication it is usually advantageous to use an incomplete NTT. However, for the Saber matrix-vector product the vector of polynomials only needs to be transformed once and the inner products can be computed in the NTT domain. Hence, a complete NTT is preferable. In the case of `ntruhps2048677` and `ntruhrss701` we compute an incomplete NTT modulo $x^{1536} - 1$ where we do 9 radix-2 splittings down to factors of degree 3. Since the input polynomials have degrees less than 768, the first splitting is for free. For `ntruhps2048509` and `ntruhps4096821` the same approach that we use on the Cortex-M4 should also give good results on Skylake. In particular, a length-1728 NTT with two radix-3 splittings, followed by 6 radix-2 splittings, down to polynomials of degree less than 3. For LAC with its polynomial moduli $x^{512} + 1$ and $x^{1024} + 1$, we compute incom-

plete negacyclic length-512 and length-1024 NTTs, respectively, each with eight layers, coming down to factors of degree 2 and 4.

We chose the prime moduli 7681 and 10753 for the NTTs of length 256, 512, 1024, and 1536. Their product is slightly longer than 26 bits, which is enough for all our applications. In the case of Saber, the absolute value of the polynomial coefficients when computing the matrix-vector product over $\mathbb{Z}$ is bounded by 2^{24}, which is below 2^{25}. In NTRU, the maximum absolute value is attained in `ntruhrss701`, where the coefficients are bounded by $2^{24.04}$ in all products of a uniform polynomial with a short polynomial. Next, as 7680 and 10752 are divisible by $1536 = 3 \cdot 2^9$, both of these moduli support complete transforms modulo $x^{1536} - 1$, which is all that we need for Saber and the NTRU parameter sets except `ntruhps4096821`. For LAC, the coefficients are even smaller so this is no problem.

For implementing the length-1728 NTT that we need in the remaining NTRU parameter set `ntruhps4096821`, the two 16-bit primes 3457 and 8641 are used. Their product is sufficiently large, they support complete length-1728 NTTs and they are even slightly smaller than the primes described above, which is good for modular reductions.

So, algebraically, for Saber, we compute the map

$$\mathbb{Z}_q[x]/(x^{256} + 1) \to \mathbb{Z}_q[x]/(x - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x - \zeta_{255}),$$

where ζ_i denote all the primitive 512-th roots of unity in $\mathbb{Z}_q$.

For `ntruhrss701` and `ntruhps2048677`, we compute

$$\mathbb{Z}_q[x]/(x^{1536} - 1) \to \mathbb{Z}_q[x]/(x^3 - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x^3 - \zeta_{511}),$$

where ζ_i denote all 512-th roots of unity.

For `ntruhps2048509`, we compute

$$\mathbb{Z}_q[x]/(x^{1024} - 1) \to \mathbb{Z}_q[x]/(x^2 - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x^2 - \zeta_{511}),$$

with ζ_i again ranging over all 512-th roots of unity.

Then, for `ntruhps4096821`, we compute

$$\mathbb{Z}_q[x]/(x^{1728} - 1) \to \mathbb{Z}_q[x]/(x^3 - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x^3 - \zeta_{575}),$$

where ζ_i denote all 576-th roots of unity. Finally, for LAC, we do

$$\mathbb{Z}_q[x]/(x^{512} + 1) \to \mathbb{Z}_q[x]/(x^2 - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x^2 - \zeta_{255}), \text{ and}$$
$$\mathbb{Z}_q[x]/(x^{1024} + 1) \to \mathbb{Z}_q[x]/(x^4 - \zeta_0) \times \cdots \times \mathbb{Z}_q[x]/(x^2 - \zeta_{255}),$$

where ζ_i denote all the primitive 512-th roots of unity.

6.3.3 Register Allocation

Intel's Skylake and later microarchitectures have a throughput of 2 vector multiplications per clock cycle with a latency of 5 cycles [Fog20]. The addition and subtraction instructions have a throughput of 3 instructions per cycle since they can go to a third execution port that is not able to execute multiplications. Their latency is 1 cycle. Hence, the subtraction instruction in Algorithm 47 ideally does not compete with the multiplication instructions for execution resources, and the maximum theoretical throughput is 2/3 vector mulmod operations per cycle, or 32/3 scalar modular multiplications per cycle. However, the critical path of a vector mulmod consists of two multiplication instructions and a subtraction and thus has a latency of 11 cycles. In order for the code to not be completely latency-bounded and get near the maximum throughput, it is important that there are always many independent mulmods that can be computed in parallel. In principle, the out-of-order execution capability allows the CPU to find independent mulmods. But in practice, the code will not come from the small μop cache and the instruction fetch from the L1 instruction cache is limited to 16 bytes per cycle, which translates to only less than about three vector instructions per cycle on average. So the code is likely to bottleneck on the front-end of the pipeline and the instruction decoding will not be able to run sufficiently far ahead for the CPU to be able to find independent instructions if they are far apart in the code. Hence it is important to schedule the instructions so that as many independent mulmods as possible are as close as possible. We achieve this by filling as many vector registers as possible with polynomial coefficients to operate on under the constraint that we also need auxiliary registers for constants and scratch registers for intermediate results. Then we can compute several NTT layers while loading coefficients only once, and, after only a few layers, arrive at polynomials that we can completely load into the registers. We also experimented with more refined approaches to scheduling where we implemented several parallel mulmods in an interleaved fashion so that we could schedule the addition and subtraction instructions in a way that they do not steal execution resources from the multiplication instructions. The downside of this approach is that by interleaving mulmod operations one needs more scratch registers so that one can either only operate on fewer polynomial coefficients at a time or needs to temporarily store away some of the coefficients. In the end, we found that not doing this and letting the register-renaming capability of the CPU take care of allocating scratch registers from the register file leads to superior results. In the power-of-two NTTs we always have 8 vector registers with a total of 128 polynomial coefficients loaded whereas in the NTT for NTRU (whose length 1536 is divisible by 3) we have always 12 registers with 192 coefficients loaded.

6.3.4 Range Analysis

For the two primes $q = 7681$ and $q = 10753$ that we use, it is not possible to compute all the layers of the NTT using straightforward radix-2 steps without performing additional modular reductions. We assume that the input polynomials we want to transform have coefficients less than 4096 in absolute value. This is true for all our applications without first reducing the polynomials modulo q. Now, by [Sei18, Lemma 2], the output coefficients of Algorithm 47 lie in the interval $[-q, q]$. So, using this approximation, we find for the forward negacyclic NTT with Cooley-Tukey butterflies that the coefficients grow by at most q in absolute value in each layer of the NTT. It then follows that we can only perform 2 layers without additional reductions. Instead, we use a more refined range analysis where for each layer and a given input range we compute the maximum range of the modular products. This then determines the range of the output coefficients, which form the inputs for the next layer. With this analysis, we find that we can compute three layers of radix-2 splittings without additional reductions, both in the cyclic and in the negacyclic NTT. After these three layers, we twist all the factors into rings of the form $\mathbb{Z}_q[x]/(x^n - 1)$. The advantage of twisting the factors instead of merely reducing coefficients is that this results in fewer modular multiplications in subsequent layers. Moreover, the mulmods as in Algorithm 47 are even slightly more efficient than, for example, Barrett reductions as they have the same throughput but shorter dependency chains. Concretely, splitting rings of the form $\mathbb{Z}_q[x]/(x^n - 1)$ does not need any mulmod. But for later factors of this form, we do in fact sometimes multiply coefficients by 1 (in the Montgomery domain) in order to reduce them using the Montgomery reduction. We then recursively compute the following maps with $16n$ mulmods, where $\zeta \in \mathbb{Z}_q$ is a primitive 8-th root of unity,

$$
\begin{aligned}
&\mathbb{Z}_q[x]/(x^{8n} - 1) \\
\to\ &\mathbb{Z}_q[x]/(x^{4n} - 1) \times \mathbb{Z}_q[x]/(x^{4n} + 1) \\
\to\ &\mathbb{Z}_q[x]/(x^{2n} - 1) \times \mathbb{Z}_q[x]/(x^{2n} + 1) \times \mathbb{Z}_q[x]/(x^{2n} - \zeta^2) \\
&\times \mathbb{Z}_q[x]/(x^n + \zeta^2) \\
\to\ &\mathbb{Z}_q[x]/(x^n - 1) \times \mathbb{Z}_q[x]/(x^n + 1) \times \mathbb{Z}_q[x]/(x^n - \zeta^2) \\
&\times \mathbb{Z}_q[x]/(x^n + \zeta^2) \times \mathbb{Z}_q[x]/(x^n - \zeta) \times \mathbb{Z}_q[x]/(x^n + \zeta) \\
&\times \mathbb{Z}_q[x]/(x^n - \zeta^3) \times \mathbb{Z}_q[x]/(x^n + \zeta^3) \\
\to\ &\mathbb{Z}_q[x]/(x^n - 1) \times \cdots \times \mathbb{Z}_q[x]/(x^n - 1)
\end{aligned}
$$

6.4 Results

In this section, we describe the benchmarking results for our Saber, NTRU, and LAC implementations. First, we describe our benchmarking setup for

the Cortex-M4 and Skylake and then we report our results for Saber, NTRU, and LAC in Sections 6.4.1, 6.4.2, and 6.4.3.

Benchmarking setup for the Cortex-M4. Our benchmarking setup is based on the `pqm4` [KPR+] benchmarking framework. We target the `STM32F407-DISCOVERY` board which has a `STM32F407VG` core. We clock it at 24 MHz with no flash wait states to obtain similar cycle counts as the ones reported in `pqm4`. For obtaining randomness, we use the hardware random number generator. As both NTRU and Saber make use of SHA-3 and SHAKE, we make use of the optimized assembly implementations of `Keccak` from the XKCP [DHP+] which is also contained in `pqm4`. LAC relies on AES and SHA-2 which we source from [SS17] and SUPERCOP [BL] respectively. All cycle counts in the following were obtained from implementations compiled with gcc and `-O3` (`arm-none-eabi-gcc`, Version 10.2.0).

Benchmarking setup for Skylake. The cycle counts for AVX2 were obtained on a Intel Core i7-6600U (Skylake) processor with a base frequency of 2.6 GHz. As usual, we disable TurboBoost and hyperthreading. We compile our implementations with gcc version 7.5.0 and use the compiler flags `-O3`, `-fomit-frame-pointer`, `-march=native`, `-mtune=native`. All cycle counts are the median cycle counts of 10 000 executions.

6.4.1 Saber Results

Table 6.4 contains the performance results for the polynomial arithmetic speed-ups in Saber. We report the results for matrix-vector multiplication $A \cdot s$ as used in key generation and encryption and vector-vector inner multiplication $b^T \cdot s$ as used in encryption and decryption separately. The dimension of the matrix is $l \times l$ and the dimensions of the vectors are $l \times 1$. The dimension $l = 2, 3, 4$ correspond to parameter sets Lightsaber, Saber, and Firesaber.

On Cortex-M4, we obtain cost reduction between 58% and 61% for $A \cdot s$ and between 42% and 44% for $b^T \cdot s$. The cost reduction on Skylake range from 25% to 39% for $A \cdot s$ and from 47% to 60% for $b^T \cdot s$.

Table 6.4: Saber performance results in clock cycles for core arithmetic operations on Cortex-M4 and Skylake. The inner-product computation in our AVX2 implementation for Saber does not contain the cost of computing the NTT of one of the input vectors. In encryption the NTT of the secret vector is already computed for the matrix-vector product. For decryption the secret vector can be stored in NTT form in the secret key, which does not need to be compatible with other implementations.

`MatrixVectorMul`						
	Cortex-M4			Skylake (AVX2)		
	[BMKV20]	**Our Work**		[BMKV20]	**Our Work**	
$l = 2$	159k	66k	(– 58%)	7 002	5 215	(–25%)
$l = 3$	317k	125k	(– 61%)	14 145	9 579	(–32%)
$l = 4$	528k	205k	(– 61%)	24 342	14 959	(–39%)
`InnerProduct`[a]						
	Cortex-M4			Skylake (AVX2)		
	[BMKV20]	**Our Work**		[BMKV20]	**Our Work**	
$l = 2$	73k	41k	(– 44%)	4 016	2 125	(–47%)
$l = 3$	99k	57k	(– 42%)	5 977	2 706	(–55%)
$l = 4$	126k	73k	(– 42%)	8 040	3 278	(–60%)

[a] [BMKV20] report cycles on a different platform with a slightly newer Kabylake processor. We have re-benchmarked their code on our Skylake platform.

Table 6.5: Performance results in clock cycles for Lightsaber, Saber, and Firesaber

			Cortex-M4		Skylake (AVX2)	
			[BMKV20][a]	**Our Work**	[BMKV20]	**Our Work**
Lightsaber	CPA	**K**	383k	294k (−23%)	49 132	47 068 (−4%)
		E	448k	330k (−26%)	46 311	42 971 (−7%)
		D	93k	58k (−38%)	7 842	5 887 (−2%)
	CCA	**K**	466k	360k (−23%)	61 325	59 831 (−2%)
		E	653k	513k (−21%)	75 876	72 473 (−4%)
		D	678k	498k (−27%)	70 228	64 859 (−8%)
Saber	CPA	**K**	738k	554k (−25%)	86 502	81 579 (−6%)
		E	830k	606k (−27%)	84 852	77 666 (−8%)
		D	128k	79k (−38%)	10 909	7 870 (−28%)
	CCA	**K**	853k	658k (−23%)	104 832	99 715 (−5%)
		E	1 103k	864k (−22%)	125 835	118 446 (−6%)
		D	1 127k	835k (−26%)	118 553	107 264 (−10%)
Firesaber	CPA	**K**	1 191k	879k (−26%)	135 986	126 476 (−7%)
		E	1 312k	947k (−28%)	136 075	123 753 (−10%)
		D	162k	101k (−38%)	14 474	10 184 (−30%)
	CCA	**K**	1 340k	1 008k (−25%)	157 915	148 729 (−6%)
		E	1 642k	1 255k (−24%)	184 322	171 993 (−7%)
		D	1 679k	1 227k (−27%)	177 864	159 950 (−10%)

[a] [BMKV20] only reports cycle counts for the CCA-secure Saber. The CPA-secure cycle counts are our own benchmarks.

Table 6.6: NTRU performance results in clock cycles for polynomial multiplication on Cortex-M4 and Skylake

	Cortex-M4			Skylake (AVX2)		
n	[KRS19][a]	**Our Work**		[ZCH+19]	**Our Work**	
509	104k	101k	(− 3%)	6 643	8 540	(+29%)
677	175k	156k	(− 11%)	11 103	10 373	(−7%)
701	173k	156k	(− 10%)	11 242	10 373	(−8%)
821	230k	199k	(− 13%)	15 507	13 247	(−15%)

[a] [KRS19] (Chapter 5) only reports cycle counts for $n = 701$, but the code generator has been used to generate Toom–Cook polynomial multiplication code to speed-up the other NTRU parameter sets. See `https://github.com/mupq/pqm4/pull/86`

Table 6.5 shows the performance of Lightsaber, Saber, and Firesaber on the Cortex-M4 when our fast `MatrixVectorMul` and `InnerProduct` are plugged into them. In addition to the full CCA-secure KEM schemes, we also report cycle counts for the underlying CPA-Secure PKE. While those are not explicitly exposed in the Saber specification, all our optimizations were inside of the CPA primitives and, hence, the overhead of the CCA transformation did not change. Moreover, some schemes use considerably more expensive CCA transforms than others. For example, Saber and Kyber include very costly public key and ciphertext hashes in their CCA transforms that could be omitted in a different choice of transform.

On Cortex-M4, we achieve significant cost reductions of consistently more than 20%. For CPA-secure decryption, we get the most notable cost reduction of 38%.

6.4.2 NTRU Results

Table 6.6 shows the results for polynomial multiplication for NTRU for the four different polynomial degrees used in `ntruhps2048509`, `ntruhps2048677`, `ntruhrss701`, and `ntruhps4096821`. On the Cortex-M4, for the smallest polynomial size $n = 509$, our implementation using NTTs is performing only slightly better than the Toom-4 implementation [KRS19]. For the larger sizes, the cost reduction on the Cortex-M4 is more pronounced with 10% or more. On AVX2, $n = 509$ is the only polynomial size for which we were not able to obtain a speed-up using NTTs. All other parameter sets have a small cost reduction of 7% to 15%. The reason why we did not achieve a speed-up for $n = 509$ is partly that we chose a different vector layout and shuffling strategy in the length-1024 NTT compared to the other NTTs. The advantage of the different vector layout is that it is easier to pre-compute the constant vectors that, additionally, need less space. But it requires more loads. In principle, the loads do not compete with the arithmetic because

they go to separate execution ports and can be dispatched in parallel. Unfortunately, it turned out that this does incur a penalty, most likely because the code is bottlenecking on the front-end. We leave it as future work to optimize the length-1024 NTTs as well as the other NTTs.

Table 6.7 and Table 6.8 report the results for the full NTRU schemes on Cortex-M4 and Skylake respectively. As we only optimize polynomial multiplication in this chapter and key generation is dominated by polynomial inversion, we do not see a big difference in cycle counts across all parameter sets and platforms. On the Cortex-M4, encapsulation is 1% to 6% faster while decapsulation is 2% to 4% faster. For the underlying CPA-secure PKE, we achieve higher speed-ups with 2% to 13% fewer cycles which comes as no surprise as we did not modify the CCA transformation.

6.4.3 LAC Results

Table 6.9 summarizes the speed of the polynomial multiplication in LAC. We can see that our code is faster than that of [LLZ+18] by a factor of 10× on the Cortex-M4 and a factor of 3× to 7× on Skylake.

Table 6.10 summarizes the results for the full LAC schemes. For `LAC-128` we see a 3× up speedup on the Cortex-M4 while there is a more modest 20–50% speedup for AVX2. For `LAC-192` and `LAC-256` there is a roughly 4× speedup for the Cortex-M4 and roughly a 2× speedup for Skylake.

Table 6.7: Performance results in clock cycles for NTRU on Cortex-M4

			Cortex-M4	
			[KRS19][a]	**Our Work**
ntruhps2048509	CPA	**K**	79 639k	79 617k (±0%)
		E	160k	152k (−5%)
		D	441k	434k (−2%)
	CCA	**K**	79 682k	79 660k (±0%)
		E	572k	564k (−1%)
		D	545k	538k (−1%)
ntruhps2048677	CPA	**K**	143 759k	143 671k (±0%)
		E	251k	224k (−11%)
		D	702k	676k (−4%)
	CCA	**K**	143 808k	143 725k (±0%)
		E	849k	821k (−3%)
		D	845k	818k (−3%)
ntruhrss701	CPA	**K**	153 794k	154 377k (±0%)
		E	299k	274k (−8%)
		D	740k	716k (−3%)
	CCA	**K**	154 477k	154 403k (±0%)
		E	403k	377k (−6%)
		D	896k	871k (−3%)
ntruhps4096821	CPA	**K**	208 892k	208 771k (±0%)
		E	327k	285k (−13%)
		D	906k	862k (−5%)
	CCA	**K**	208 953k	207 495k (−1%)
		E	1 069k	1 027k (−4%)
		D	1 075k	1 030k (−4%)

[a][KRS19] (Chapter 5) only reports cycle counts for the CCA-secure `ntruhrss701` from the first round of the NIST competition. Cycle counts in this table are our own benchmarks of the second-round code contained in `pqm4` [KPR+].

Table 6.8: Performance results in clock cycles for NTRU on Skylake (AVX2)

			Skylake (AVX2)	
			[ZCH+19]	**Our Work**
ntruhps2048509	CPA	**K**	155 306	164 952 (+6%)
		E	10 183	12 052 (+18%)
		D	27 314	31 340 (+15%)
	CCA	**K**	208 653	218 887 (+5%)
		E	71 018	73 176 (+3%)
		D	38 950	42 953 (+10%)
ntruhps2048677	CPA	**K**	264 398	264 276 (±0%)
		E	15 794	15 821 (±0%)
		D	43 352	42 515 (−2%)
	CCA	**K**	332 906	333 278 (±0%)
		E	96 293	95 953 (±0%)
		D	59 169	58 406 (−1%)
ntruhrss701	CPA	**K**	265 341	264 501 (±0%)
		E	19 096	18 507 (−3%)
		D	45 130	43 770 (−3%)
	CCA	**K**	299 066	298 505 (±0%)
		E	56 616	56 084 (−1%)
		D	62 503	61 199 (−2%)
ntruhps4096821	CPA	**K**	375 171	367 911 (−2%)
		E	18 914	16 917 (−11%)
		D	55 573	52 204 (−6%)
	CCA	**K**	458 614	451 664 (−2%)
		E	114 986	113 935 (−1%)
		D	74 182	70 917 (−4%)

[a] [KRS19] (Chapter 5) only reports cycle counts for the CCA-secure `ntruhrss701` from the first round of the NIST competition. Cycle counts in this table are our own benchmarks of the second-round code contained in `pqm4` [KPR+].

Table 6.9: LAC polynomial multiplication clock cycles on Cortex-M4 and Skylake

	Cortex-M4			Skylake (AVX2)		
	[LLZ+18]	**Our Work**		[LLZ+18]	**Our Work**	
LAC-128	638k	65k	(−90%)	14 691	4 552	(−69%)
LAC-192	1 274k	131k	(−90%)	73 955	10 119	(−86%)
LAC-256	1 701k	132k	(−92%)	73 955	10 119	(−86%)

Table 6.10: Performance results in clock cycles for LAC

			Cortex-M4		Skylake (AVX2)	
			[LLZ+18]	**Our Work**	[LLZ+18]	**Our Work**
LAC-128	CPA	**K**	850k	282k (−67%)	42 841	30 959 (−28%)
		E	1 424k	444k (−69%)	60 797	41 485 (−32%)
		D	528k	113k (−79%)	26 880	14 512 (−46%)
	CCA	**K**	850k	282k (−67%)	53 000	42 167 (−20%)
		E	1 430k	450k (−69%)	76 418	59 252 (−22%)
		D	1 960k	565k (−71%)	86 209	55 880 (−35%)
LAC-192	CPA	**K**	1 506k	373k (−75%)	90 742	36 248 (−60%)
		E	2 417k	601k (−75%)	111 839	52 055 (−53%)
		D	899k	210k (−77%)	66 349	12 508 (−81%)
	CCA	**K**	1 507k	373k (−75%)	96 270	41 713 (−57%)
		E	2 427k	610k (−75%)	128 342	67 732 (−47%)
		D	3 329k	824k (−75%)	189 660	74 393 (−61%)
LAC-256	CPA	**K**	2 019k	459k (−77%)	125 380	60 242 (−52%)
		E	3 623k	739k (−80%)	171 038	77 268 (−55%)
		D	1 690k	359k (−79%)	87 588	23 558 (−73%)
	CCA	**K**	2 020k	459k (−77%)	143 568	76 917 (−46%)
		E	3 633k	748k (−79%)	202 346	106 836 (−47%)
		D	5 327k	1 111k (−79%)	262 901	104 897 (−60%)

Chapter 7

Recent Developments and Outlook

This chapter describes some recent developments and future directions related to the research presented in this thesis. With the NISTPQC competition nearing the end of the third round, NIST will likely publish PQC standards in the next few years. Beyond the NISTPQC competition, stateful hash-based signatures have recently been standardized as informational RFCs [HBG+18, MCF19]. This means that PQC will soon be deployed to many applications. As of now, lattice-based cryptography appears to be the most promising family of post-quantum cryptography as it has reasonable sizes and achieves very good performance. This thesis contributed to a better understanding of how state-of-the-art lattice-based cryptography can be best implemented on microcontrollers. For the schemes covered in this thesis, it appears that NTTs result in the fastest implementations even if schemes are not designed with NTTs in mind. Run-time of lattice-based cryptography appears to present a very minor challenge, even on microcontrollers, while RAM consumption can be a larger obstacle. Even though considerable progress in the area of efficient and secure PQC implementation has been made in recent years, still many problems remain unsolved and more research is needed. In the following, I outline what I consider the most relevant problems for PQC implementations pointing out recent work addressing them and challenges that remain unsolved.

More primitives on Cortex-M4. While in this thesis the focus was on lattice-based constructions, other schemes are also promising for some uses cases and have received significantly less attention from the implementation community. In our recent work [CKY21], we study how Rainbow [DCK+19] can be implemented on the Cortex-M4. Due to the large public key of 158 kB, the development board used in this thesis (STM32F4DISCOVERY) cannot

even accommodate the public key. Hence, we use the larger Silicon Labs' Giant Gecko (EFM32GG11B) board which has 512 kB of RAM. The coverage of other schemes also improved over the last years. For example, there is recent work describing Cortex-M4 implementations of BIKE [CCK21], Classic McEliece [CC21], and SIKE [SAJA20, AAK21]. As some of these implementations are the first ones to target the Cortex-M4, there may be more progress to be made.

Smaller processors. Even though NIST has designated the Cortex-M4 as the main microcontroller optimization target, PQC will need to be deployed to smaller devices as well. Hence, we need to understand how different PQC schemes perform. Chapter 4 provided one step in that direction by also studying implementations on the Cortex-M3. The most significant observation is that the Cortex-M4 has much more powerful multiplication instructions than its predecessors. This benefits schemes that inherently rely on multiplications rather than other operations. Beyond the Cortex-M3, one may want to look into the Arm Cortex-M0, RISC-V, or AVR microcontrollers. Those are likely even more restricted in terms of RAM and flash which may present a challenge for PQC implementations.

Larger processors. Another direction to go forward would be to investigate larger processors, for example, those providing more powerful vector instructions. One interesting direction is to study the successors of the Cortex-M4 in the Arm M-Series implementing the M-Profile Vector Extension (MVE), for example, the Arm Cortex-M55. Implementations of Saber using MVE have recently been studied in [BMK+21]. Apart from the M-Series, the Arm A-Series presents another attractive target architecture as it is commonly used, e.g., in smartphones. It implements the Neon vector instruction set. Our recent paper [BHK+21] studies how Kyber, Saber, and Dilithium can be efficiently implemented using Neon on the Arm Cortex-A72 implementing `Armv8-A`. As future Arm A cores (≥`Armv8.4-A`) provide SHA-3 hardware support, they will be very suitable for post-quantum cryptography. Another feature introduced in `Armv8.4-A` are instructions supporting data-independent timing (DIT).[1] The programmer can turn on DIT at run time by setting a flag. When DIT is turned off, instructions may have data-dependent runtime on some cores implementing `Armv8.4-A`. If DIT is turned on, all instructions within the architecturally defined scope of DIT are guaranteed to have an execution time that's independent of their input data. This may introduce a performance penalty as short-cuts in the circuit can not longer be taken. This allows implementing cryptographic code safely while not unnecessarily slowing down application code processing uncritical data. As of `Armv8.4-A` not all instructions are currently covered by DIT, however, this still presents a new way to thwart timing attacks on a microarchitectural level.

[1] https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT–Data-Independent-Timing

Non-interactive key exchange (NIKE). All constructions underlying the KEMs in the NISTPQC competition have a significant drawback: They do not allow to construct a NIKE. However, NIKEs are often used in todays protocols as they are easily constructed from the (EC-)DLP. Replacing those with one of the NISTPQC standards is not going to be possible without significantly changing the protocols. The first post-quantum NIKE is CSIDH [CLM+18] which was proposed in 2018, i.e., after the NISTPQC competition had already started. It is based on supersingular isogeny graphs. While CSIDH has very small public key and ciphertext sizes, it is, unfortunately, prohibitively slow especially on embedded devices. Studying how the performance of CSIDH can be improved, or proposing completely new efficient NIKEs is a promising area of future research.

Side-channel attacks. Timing attacks and power analysis attacks both proved to be very effective in attacking implementations of cryptography. As timing attacks are mountable remotely, they are relevant for virtually all deployments. Hence, the vast majority of cryptographic implementations including the ones described in this thesis aim to be resistant against timing attacks. In theory, timing-attack countermeasures are well understood, however, in practice, they may be implemented insecurely as, for example, shown in [GJN20] for the NISTPQC scheme FrodoKEM. Power analysis attacks are limited to a scenario where an adversary may be able to observe power leakage. However, if that is the case, they are much harder to protect against than timing attacks. The most common countermeasure against (differential) power analysis attacks is masking which has been recently studied for some NISTPQC schemes [MGTF19, BGR+21, FBR+21, ABE+21]. However, recent work on single-trace attacks of PQC [HHP+21, PP19, PPM17] suggests that masking may not be enough. Our work on single-trace attacks on Keccak [KPP21] also applies although it did not exclusively cover post-quantum schemes.

Fault attacks. In case the attacker is able to actively tamper with the device to induce faults in the cryptographic computation, fault attacks may be possible. There already exist a number of studies about the fault attack susceptibility of NISTPQC schemes [PP21, GBP18, AOTZ20, TFMP21]. One of our papers [GKPM18] studied practical fault attacks on SPHINCS (the predecessor of the NISTPQC submission SPHINCS+) and has shown that a single fault in the large hyper-tree computation allows the adversary to create forgeries. In another paper [CKM+20], we studied how to protect the isogeny-based scheme CSIDH [CLM+18] against fault attacks. More work is required to better understand protection of PQC from fault attacks.

Formal verification of implementations. Implementation mistakes can have devastating consequences for the security of cryptographic schemes. Hence, it is essential to ensure that cryptographic implementations are correct, i.e., implemented according to the specification. Testing and checking

that implementations produce the same test vectors as the reference implementations, in the way `pqm4` does, is able to catch many bugs. However, it cannot provide us any correctness guarantees. Therefore, it is still possible that bugs remain that only occur very rarely or on specially crafted inputs. Random testing is unsuitable to find such mistakes. Formal methods can provide much stronger guarantees about the correctness of implementations. While this is an active area of research, thus far, I am not aware of any published verified implementations of any NISTPQC scheme.

Protocols. The entire NISTPQC competition focuses on selecting primitives for key establishment and signatures. However, these primitives will then be embedded into many different types of protocols. It is essential to understand what the implications of switching to post-quantum schemes will be for the protocols. While for some protocols, it may be possible to simply drop in a post-quantum replacement, others may have to be modified or even redesigned entirely. Various research papers have studied the arguably most important protocol on the internet: TLS [BCNS15, PST20, SKD20]. While current proposals require signatures, this will come at a rather large cost in the post-quantum world. It may be preferable to modify TLS to rely on KEMs only, as, e.g., done in KEMTLS [SSW20, SSW21]. As TLS is commonly used on a wide variety of devices, it will be crucial to understand the impact of changing to post-quantum TLS.

Migration. Arguably the biggest challenge of PQC is going to be migration. As there are billions of devices running cryptography vulnerable to quantum attacks, every single one of them will need to be updated or replaced. As many of them have not been designed with cryptographic agility — the ability to replace outdated cryptography – in mind, this will be especially challenging. From an implementation viewpoint, the first major challenge will be to extend existing cryptographic libraries for support of the new post-quantum primitive in a vast number of programming languages.

NISTPQC signature on-ramp. Recently, NIST announced[2] that due to recent attacks on GeMSS [TPD21] and Rainbow [Beu21], they are concerned about the security of the current MQ proposals. Consequently, the only signature schemes left are SPHINCS$^+$ and the lattice-based signatures Falcon and Dilithium. As NIST would like to see more diversity, they will publish a call for more signature proposals soon. It is expected that such a call will be published at the end of round three with a deadline in 2022. Since none of the recent attacks fundamentally breaks MQ-based cryptography, it is expected that more MQ proposals will be submitted. Additionally, some newer isogeny-based proposals may be promising, e.g., SQISign [FKL$^+$20]. It will be essential to understand the performance characteristics of these new proposals on a wide variety of platforms.

[2] `https://csrc.nist.gov/CSRC/media/Presentations/status-update-on-the-3rd-round/images-media/session-1-moody-nist-round-3-update.pdf`

Acronyms

The following lists common acronyms in alphabetical order that are commonly used throughout this thesis. Scheme or algorithm names (like AES or CT) are not included.

ABI Application binary interface

AES Advanced encryption standard

API Application programming interface

CACR Chinese Association of Cryptologic Research

CCA Chosen-ciphertext attack

CPA Chosen-plaintext attack

CPU Central processing unit

CRT Chinese remainder theorem

DFT Discrete Fourier transform

DIF Decimation in frequency

DIT Decimation in time

DLP Discrete-logarithm problem

DPKE Deterministic public-key encryption

DSP Digital signal processing

ECC Elliptic-curve cryptography

FFT Fast Fourier transform

FPU Floating point unit

ISA Instruction set architecture

KEM Key-encapsulation mechanism

LWE Learning with errors

LWR Learning with rounding

MAC Message authentication code

MLWE Module-LWE

MVE M-Profile vector extension

NIST National Institute for Standards and Technology

NTT Number-theoretic transform

PKC Public-key cryptography

PKE Public-key encryption

PQC Post-quantum cryptography

RAM Random-access memory

RLWE Ring-LWE

RNG Random-number generator

ROM Read-only memory

SHA Secure hash algorithm

SIMD Single instruction, multiple data

SRAM Static random-access memory

XOF Extensible-output function

Bibliography

[AAB+17] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. `https://newhopecrypto.org/`.

[AAK21] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–13, 2021. `https://eprint.iacr.org/2021/115.pdf`.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, 2020. `https://eprint.iacr.org/2020/012`.

[ABD+] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. ARM Cortex-M4 optimized implementation of Kyber. `https://github.com/pq-crystals/kyber/tree/cm4/cm4`.

[ABD+17] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. `https://pq-crystals.org/kyber`.

[ABE+21] Diego F. Aranha, Sebastian Berndt, Thomas Eisenbarth, Okan Seker, Akira Takahashi, Luca Wilke, and Greg Za-

verucha. Side-channel protections for Picnic signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):239–282, 2021. https://eprint.iacr.org/2021/735.

[ACC+21a] 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2021. https://eprint.iacr.org/2021/995.

[ACC+21b] 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://eprint.iacr.org/2020/1216.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*, pages 327–343. USENIX Association, 2016. https://eprint.iacr.org/2015/1092.

[AEL+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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):219–242, 2020. https://eprint.iacr.org/2020/049.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. A new hope on ARM Cortex-M. In *Security, Privacy, and Advanced Cryptography Engineering – SPACE 2016*, LNCS, pages 332–349. Springer, 2016. https://eprint.iacr.org/2016/758.

[AOTZ20] Diego F. Aranha, Claudio Orlandi, Akira Takahashi, and Greg Zaverucha. Security of hedged Fiat–Shamir signatures under fault attacks. In *Advances in Cryptology – EUROCRYPT 2020*, LNCS, pages 644–674. Springer, 2020. https://eprint.iacr.org/2019/956.

[ARM12] Arm Limited. *Arm Cortex-M Programming Guide to Memory Barrier Instructions*, 2012. https://developer.arm.com/documentation/dai0321/a.

[ARM20] Arm Limited. *Procedure Call Standard for the Arm Architecture - ABI*, 2020. https://developer.arm.com/documentation/ihi0042/j/.

[AYS15] Aydin Aysu, Bilgiday Yuce, and Patrick Schaumont. The Future of Real-Time Security: Latency-Optimized Lattice-Based Digital Signatures. *ACM Transactions on Embedded Computing Systems*, 14(3), 2015. https://dl.acm.org/doi/abs/10.1145/2724714.

[BAA+17] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Kramer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://www.informatik.tu-darmstadt.de/tesla.

[Ban17] Rachid El Bansarkhani. KINDI: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. http://kindi-kem.de.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology – CRYPTO 1986*, LNCS, pages 311–323. Springer, 1986.

[BBF+19] Hayo Baan, Sauvik Bhattacharya, Scott R. Fluhrer, Óscar García-Morchón, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In *Post-Quantum Cryptography – PQCrypto 2019*, LNCS, pages 83–102. Springer, 2019. https://eprint.iacr.org/2018/725.

[BCLv17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://ntruprime.cr.yp.to/.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE*

Symposium on Security and Privacy - S&P 2015, pages 553–570. IEEE, 2015. https://eprint.iacr.org/2014/599.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Kyber: A cca-secure module-lattice-based KEM. In *IEEE European Symposium on Security and Privacy – EuroS&P 2018*, pages 353–367. IEEE, 2018. https://eprint.iacr.org/2017/634.

[BDPA11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. The Keccak reference. Submission to the NIST SHA-3 competition (round 3), 2011. https://keccak.team/files/Keccak-reference-3.0.pdf.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians, 2001. http://cr.yp.to/papers.html#m3.

[Beu21] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In *Advances in Cryptology – EUROCRYPT 2021*, LNCS, pages 348–373. Springer, 2021. https://eprint.iacr.org/2020/1343.

[BFM+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! Faster Frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Report 2018/1116, 2018. https://eprint.iacr.org/2018/1116.

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In *Theory of Cryptography – TCC 2016*, LNCS, pages 209–224. Springer, 2016. https://eprint.iacr.org/2015/769.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):173–214, 2021. https://eprint.iacr.org/2021/483.

[BHK+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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2021. https://eprint.iacr.org/2021/986.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology – Africacrypt 2019*, LNCS, pages 209–228. Springer, 2019. https://eprint.iacr.org/2019/489.

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to.

[BMK+21] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, 2021. https://eprint.iacr.org/2021/998.

[BMKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, 2020. https://eprint.iacr.org/2020/268.

[BMTK+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 key encapsulation mechanism. In *Design Automation Conference – DAC 2020*. IEEE, 2020. https://eprint.iacr.org/2020/321.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Advances in Cryptology – EUROCRYPT 2012*, LNCS, pages 719–737. Springer, 2012. https://eprint.iacr.org/2011/401.

[BS07] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Mathematics of Computation*, 76(257):443–454, 2007. http://cr.yp.to/papers.html#meecrt.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, 2019. https://eprint.iacr.org/2019/1140.

[CAC19] Chinese Association of Cryptologic Research CACR. National cryptographic algorithms design contest, 2019. http://sfjs.cacrnet.org.cn/site/content/309.html.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):125–148, 2021. https://eprint.iacr.org/2021/492.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):97–124, 2021. https://eprint.iacr.org/2021/493.

[CHK+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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://eprint.iacr.org/2020/1397.

[CKM+20] Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with dummy-operations against fault injection attacks. In *Workshop on Fault Detection and Tolerance in Cryptography*, pages 57–65, 2020. https://eprint.iacr.org/2020/1005.

[CKY21] Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):650–675, 2021. https://eprint.iacr.org/2021/532.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In *Advances in Cryptology – ASIACRYPT 2018*, LNCS, pages 395–427. Springer, 2018. https://eprint.iacr.org/2018/383.

[Coo66] Stephen Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

[CPL+17] Jung Hee Cheon, Sangjoon Park, Joohee Lee, Duhyeong Kim, Yongsoo Song, Seungwan Hong, Dongwoo Kim, Jinsu

Kim, Seong-Min Hong, Aaram Yun, Jeongsu Kim, Haeryong Park, Eunyoung Choi, Kimoon kim, Jun-Sub Kim, and Jieun Lee. Lizard: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965. https://www.jstor.org/stable/2003354.

[DCK+19] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://www.pqcrainbow.org/.

[Den03] Alexander W. Dent. A designer's guide to KEMs. In *Cryptography and Coding – IMACC 2003*, LNCS, pages 133–151. Springer, 2003. https://eprint.iacr.org/2002/174.

[dG15] Wouter de Groot. A performance study of X25519 on Cortex-M3 and M4. Master's thesis, Technische Universiteit Eindhoven, 2015. https://pure.tue.nl/ws/portalfiles/portal/47038543.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DHP+] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018. https://eprint.iacr.org/2017/633.

[DKRV17] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Algorithm specification and supporting documentation. Submission to

the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[EGM90] Shimon Even, Oded Goldreich, and Silvio Micali. On-Line/Off-Line Digital Signatures. In *Advances in Cryptology – CRYPTO 1989*, LNCS, pages 263–275. Springer, 1990. https://link.springer.com/chapter/10.1007/0-387-34805-0_24.

[FBR+21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):414–460, 2021. https://eprint.iacr.org/2021/479.

[FKL+20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In *Advances in Cryptology – ASIACRYPT 2020*, LNCS, pages 64–93. Springer, 2020. https://eprint.iacr.org/2020/1240.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO 1999*, LNCS, pages 537–554. Springer, 1999. http://dx.doi.org/10.1007/3-540-48405-1_34.

[Fog20] Agner Fog. Instruction tables, 2020. http://www.agner.org/optimize/instruction_tables.pdf.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, 2020. https://eprint.iacr.org/2020/446.

[GBP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):21–43, Aug. 2018. https://eprint.iacr.org/2018/355.pdf.

[GHK+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 – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. https://eprint.iacr.org/2021/662.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki–Okamoto transformation and its application on FrodoKEM. In *Advances in Cryptology – CRYPTO 2020*, LNCS, pages 359–386. Springer, 2020. https://eprint.iacr.org/2020/743.

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of lattice-based signature schemes in embedded systems. In *ICECS 2018*, pages 385–388, 2018. https://www.seceng.ruhr-uni-bochum.de/media/seceng/veroeffentlichungen/2018/10/17/paper.pdf.

[GKPM18] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. In *Kangacrypt*, 2018. https://eprint.iacr.org/2018/674.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. https://eprint.iacr.org/2020/1278.

[GMZB+17] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, and Jose-Luis Torre-Arce. Round2: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://www.onboardsecurity.com/nist-post-quantum-crypto-submission.

[Goo51] Irving J. Good. Random motion on a finite abelian group. *Proceedings of the Cambridge Philosophical Society*, 47:756–762, 1951. MR 13,363e.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *Post-Quantum Cryptography –*

PQCrypto 2013, LNCS, pages 67–82. Springer, 2013. http://cryptojedi.org/papers/#lattisigns.

[GOPT09] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *Information, Security and Cryptology – ICISC 2009*, LNCS, pages 176–192. Springer, 2009. https://eprint.iacr.org/2009/538.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qTESLA, 2019. https://eprint.iacr.org/2019/606.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC 1996*, pages 212–219, 1996. https://arxiv.org/abs/quant-ph/9605043.

[GS66] W. Morven Gentleman and G. Sande. Fast Fourier transforms: for fun and profit. In *AFIPS Fall Joint Computing Conference 1966*, pages 563–578. AFIPS, 1966. https://www.cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf.

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018. https://rfc-editor.org/rfc/rfc8391.txt.

[HGSW05] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3. In *Topics in Cryptology – CT-RSA 2005*, LNCS, pages 118–135. Springer, 2005. https://eprint.iacr.org/2005/045.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki–Okamoto transformation. In *Theory of Cryptography – TCC 2016*, LNCS, pages 341–371. Springer, 2017. https://eprint.iacr.org/2017/604.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):88–113, Aug. 2021. https://eprint.iacr.org/2021/956.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory – ANTS 1998*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998. http://dx.doi.org/10.1007/BFb0054868.

[HRSS17a] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, LNCS, pages 232–252. Springer, 2017. https://eprint.iacr.org/2017/667.

[HRSS17b] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. NTRU-KEM-HRSS: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://ntru-hrss.org.

[JAC+17] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://sike.org/.

[KBMSRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, 2018. https://eprint.iacr.org/2018/682.

[KGB+18] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2018*, pages 168–188. Springer, 2018. https://eprint.iacr.org/2018/673.

[KO63] Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. Scanned version on http://cr.yp.to/bib/1963/karatsuba.html.

[KPP21] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):243–268, 2021. https://eprint.iacr.org/2020/371.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security – ACNS 2019*, LNCS, pages 281–301. Springer, 2019. https://eprint.iacr.org/2018/1018.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST-PQC on ARM Cortex-M4. In *Second NIST PQC Standardization Conference*, 2019. https://eprint.iacr.org/2019/844.

[LDK+17] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://pq-crystals.org/dilithium.

[LLJ+17] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, JingnanHe, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kunpeng Wang. LAC: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical Ring-LWE based public-key encryption with byte-level modulus. 2018. https://eprint.iacr.org/2018/1009.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: truly fast NTRU using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019. https://eprint.iacr.org/2019/040.

[Lyu09] Vadim Lyubashevsky. Fiat–Shamir with aborts: Applications to lattice and factoring-based signatures. In *Advances in Cryptology – ASIACRYPT 2009*, LNCS, pages 598–616. Springer, 2009. https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019. https://rfc-editor.org/rfc/rfc8554.txt.

[MGTF19] Vincent Migliore, Benoit Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium: Efficient implementation and side-channel evaluation, 2019. https://eprint.iacr.org/2019/394.

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology – CRYPTO 1985*, LNCS, pages 417–426. Springer, 1985.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf.

[MP21] Michele Mosca and Marco Piani. Quantum threat timeline report 2020, January 2021. https://globalriskinstitute.org/publications/quantum-threat-timeline-report-2020/.

[Nat13] National Institute of Standards and Technology. FIPS186-4: Digital Signature Standard (DSS), 2013. https://doi.org/10.6028/NIST.FIPS.186-4.

[Nat18] National Institute of Standards and Technology. NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3.

[Nat19a] National Institute of Standards and Technology. NIST SP 800-56B Rev. 2: Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography, 2019. https://doi.org/10.6028/NIST.SP.800-56Br2.

[Nat19b] National Institute of Standards and Technology. NISTIR 8240: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process, 2019. https://doi.org/10.6028/NIST.IR.8240.

[Nat20] National Institute of Standards and Technology. NISTIR 8309: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process, 2020. https://doi.org/10.6028/NIST.IR.8309.

[NIS15a] FIPS PUB 180-4: Secure hash standard, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[NIS15b] FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[NIS16] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer, 1982.

[PFH+17] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://falcon-sign.info/.

[Por19] Thomas Pornin. New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893, 2019. https://eprint.iacr.org/2019/893.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *Progress in Cryptology – LATINCRYPT 2019*, LNCS, pages 130–149. Springer, 2019. https://eprint.iacr.org/2019/795.

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):37–60, 2021. https://eprint.iacr.org/2021/064.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, LNCS, pages 513–533. Springer, 2017. https://eprint.iacr.org/2017/594.

[PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in TLS. In *Post-Quantum Cryptography – PQCrypto 2020*, LNCS, pages 72–91. Springer, 2020. https://eprint.iacr.org/2019/1447.

[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 – CARDIS 2019*, LNCS, pages 57–73. Springer, 2019. https://eprint.iacr.org/2019/420.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. https://people.csail.mit.edu/rivest/Rsapaper.pdf.

[Saa17] Markku-Juhani O. Saarinen. HILA5: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://mjos.fi/hila5.

[SAJA20] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (SIKE) round 2 on ARM Cortex-M4. *IEEE Transactions on Computers*, pages 1–14, 2020. https://eprint.iacr.org/2020/410.

[SBGM+18] Markku-Juhani O. Saarinen, Sauvik Bhattacharya, Oscar Garcia-Morchon, Ronald Rietman, Ludo Tolhuizen, and Zhenfei Zhang. Shorter messages and faster post-quantum encryption with Round5 on Cortex M. In *Smart Card Research and Advanced Applications – CARDIS 2018*, LNCS, pages 95–110. Springer, 2018. https://eprint.iacr.org/2018/723.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Annual Symposium on Foundations of Computer Science – FOCS 1994*, pages 124–134. IEEE, 1994. https://ieeexplore.ieee.org/abstract/document/365700.

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in TLS 1.3: A perfor-

mance study. In *Annual Network and Distributed System Security Symposium – NDSS 2020*. The Internet Society, 2020. https://eprint.iacr.org/2020/071.

[Sol99] Jerome A. Solinas. Generalized Mersenne numbers. Technical report, 1999. http://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971.

[SS17] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptology – SAC 2016*, LNCS, pages 180–194. Springer, 2017. https://eprint.iacr.org/2016/714.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *ACM SIGSAC Conference on Computer and Communications Security – CCS 2020*, pages 1461–1480. ACM, 2020. https://eprint.iacr.org/2020/534.

[SSW21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. More efficient post-quantum KEMTLS with pre-distributed public keys. In *European Symposium on Research in Computer Security – ESORICS 2021 – Part I*, LNCS, pages 3–22. Springer, 2021. https://eprint.iacr.org/2021/779.

[STM21] STMicroelectronics N.V. *Reference manual - STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs*, 2021. https://www.st.com/resource/en/reference_manual/rm0090-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

[TFMP21] Elise Tasso, Luca De Feo, Nadia El Mrabet, and Simon Pontié. Resistance of isogeny-based cryptographic implementations to a fault attack, 2021. https://eprint.iacr.org/2021/850.

[Too63] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. www.de.ufpe.br/~toom/my-articles/engmat/MULT-E.PDF.

[TPD21] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Efficient key recovery for all HFE signature variants. In *Advances in Cryptology – CRYPTO 2021*, LNCS, pages 70–93. Springer, 2021. https://eprint.iacr.org/2020/1424.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[WTJ+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the HW/SW co-design of qTESLA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):269–306, 2020. https://eprint.iacr.org/2020/054.

[ZCD+17] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, Vladmir Kolesnikov, and Daniel Kales. Picnic: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://microsoft.github.io/Picnic/.

[ZCH+19] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hülsing, Joost Rijneveld, Peter Schwabe, and Oussama Danba. NTRU: Algorithm specification and supporting documentation, 2019. https://ntru.org/.

[ZCHW17] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. NTRUEncrypt: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[ZYF+20] Jiang Zhang, Yu Yu, Shuqin Fan, Zhenfeng Zhang, and Kang Yang. Tweaking the asymmetry of asymmetric-key cryptography on lattices: KEMs and signatures of smaller sizes, 2020. https://eprint.iacr.org/2019/510.

Appendix A

Research Data Management

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.[1]

The research datasets produced during this PhD research packaged into a single archive are available at `https://doi.org/10.5281/zenodo.5555735`. The following research datasets, exclusively consisting of source code, have been produced during this PhD research and are also available individually on Github:

- Chapter 2: Polynomial Multiplication for Computer Scientists.
  `https://github.com/mkannwischer/polymul`
- Chapter 3: Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4.
  `https://github.com/mupq/nttm4`
- Chapter 4: Compact Dilithium Implementations on Cortex-M3 and Cortex-M4.
  `https://github.com/dilithium-cortexm/dilithium-cortexm`
- Chapter 5: Toom–Cook and Karatsuba Multiplication for $\mathbb{Z}_{2^m}[x]$.
  `https://github.com/mupq/polymul-z2mx-m4`
- Chapter 6: NTT Multiplication for NTT-unfriendly Rings.
  `https://github.com/ntt-polymul/ntt-polymul`

[1] ru.nl/icis/research-data-management/, last accessed October 08th, 2021.

Summary

With the advance of quantum computers, there is an urgent need to find replacements for public-key cryptography threatened by Shor's quantum algorithm. This thesis presents work towards understanding post-quantum replacements for key-encapsulation mechanisms and digital signatures from an implementation perspective. The focus of this thesis lies on polynomial multiplication which is at the core of most post-quantum cryptography based on hard lattice problems. Chapters 1 and 2 present the background and common concepts among the remainder of the thesis. The main body is then divided into two parts: Part I (Chapter 3 and Chapter 4) covers cryptographic schemes specifically designed to benefit from a particular polynomial multiplication technique: Number-theoretic transforms. Part II (Chapter 5 and Chapter 6) covers the other lattice-based schemes that were designed without tailoring parameter choices to a specific multiplication algorithm.

Chapter 3. This chapter studies microcontroller (Arm Cortex-M4) implementations of the post-quantum key-encapsulation mechanism Kyber. It speeds up the polynomial multiplication within Kyber by improving upon previous implementations of the number-theoretic transform. This is achieved by making use of the single-instruction-multiple-data arithmetic available on the Arm Cortex-M4. The resulting implementation of the NTT is more than twice as fast as the previous speed record. Furthermore, it studies how to minimize the memory usage of Kyber implementations. Kyber is much more favorable for memory-efficient implementations than other candidates.

Chapter 4. The next chapter presents Arm Cortex-M3 and Arm Cortex-M4 implementations of the post-quantum digital signature scheme Dilithium. The focus again lies on the number-theoretic transform and reducing the memory consumption of the implementation. A substantial challenge on the Arm Cortex-M3 is the data-dependent timing of certain multiplication instructions often used within the polynomial arithmetic of Dilithium. As this data dependence leaks information about sensitive data within the computation, one has to find an alternative way of implementing polynomial multiplication while avoiding

variable-time instructions. We propose one such way that exclusive uses constant-time multiplications. The core contributions are new speed records for Dilithium on Arm Cortex-M3 and Arm Cortex-M4 as well as a study of different ways of implementing Dilithium with varying memory constraints.

Chapter 5. In this chapter, Arm Cortex-M4 implementations of five post-quantum key-encapsulation mechanisms are studied: RLizard, NTRU-HRSS, NTRU-KEM-743, Saber, Kindi. The chapter focuses on multiplication using algorithms by Toom–Cook and Karatsuba. The core contribution is a code generator capable of generating polynomial-multiplication code for a wide variety of parameters. By plugging the multiplication routines into various post-quantum schemes we achieve performance superior to the previous state of the art. For most of the studied schemes, the only previous implementation that executes on the Arm Cortex-M4 is the reference implementation; for some of those schemes, our optimized software is more than a factor of 20 faster. One of the schemes, namely Saber, has been optimized on the Cortex-M4 in previous work; the multiplication routine for Saber we present here outperforms the multiplication from that work by almost 2×. Out of the five schemes optimized in this chapter, the best performance for encapsulation and decapsulation is achieved by NTRU-HRSS. Specifically, encapsulation takes just over 400 000 cycles, which is more than twice as fast as for any other NIST candidate that has previously been optimized on the Arm Cortex-M4.

Chapter 6. The last chapter of the main body of this thesis further improves upon Chapter 5 by studying the use of the number-theoretic transform in NTRU, LAC, and Saber. It shows that implementations using the number-theoretic transform are superior to Toom–Cook and Karatsuba on the Arm Cortex-M4 and Intel Skylake. Interestingly, these two platforms mandate different approaches: On the Cortex-M4, we use 32-bit arithmetic, while on Intel we use two 16-bit NTT-based polynomial multiplications and combine the products using the Chinese Remainder Theorem (CRT). For Saber, the performance gain is particularly pronounced. On Cortex-M4, the Saber NTT-based matrix-vector multiplication is 2.5× faster than previous implementations using Toom–Cook multiplication. For NTRU, the speedup is less impressive but still performs better than Toom–Cook for all parameter sets on Cortex-M4.

Samenvatting

Door de opmars van de quantumcomputers is er een dringende behoefte om vervanging te vinden voor publieke-sleutelcryptografie die wordt bedreigd door het algoritme van Shor. Dit proefschrift levert een bijdrage aan het begrip van post-quantum-alternatieven voor sleutel-inkapselmechanismen en digitale handtekeningen, vanuit een implementatie-perspectief. De focus van dit proefschrift ligt op polynoomvermenigvuldigingen, wat de kern is van de meeste op roosters gebaseerde post-quantumcryptografie. Hoofdstuk 1 en 2 gaan in op de achtergrond, en op algemene concepten voor de rest van het proefschrift. De kern is vervolgens opgedeeld in twee delen: deel I (hoofdstuk 3 en hoofdstuk 4) beslaat cryptografische systemen die zijn ontworpen om te profiteren van een specifieke techniek voor polynoomvermenigvuldiging: de getaltheoretische transformatie (NTT). Deel II (hoofdstuk 5 en hoofdstuk 6) gaat in op de andere op roosters gebaseerde systemen, waarbij in het ontwerp geen rekening is gehouden met een specifiek vermenigvuldigingsalgoritme.

Hoofdstuk 3. Dit hoofdstuk bekijkt implementaties van het post-quantum sleutel-inkapselingsmechanisme Kyber voor microcontrollers (specifiek: Arm Cortex-M4). We versnellen de polynoomvermenigvuldiging in Kyber door een eerdere implementatie van de NTT te verbeteren. Dit wordt bereikt door gebruik te maken van de enkelvoudige-instructie-meervoudige-data-berekeningen (SIMD) die beschikbaar zijn op de Arm Cortex-M4. De daaruit volgende implementatie van de NTT is meer dan twee keer zo snel als het voormalige snelheidsrecord. Ook bestudeert dit hoofdstuk hoe het geheugengebruik van implementaties van Kyber kan worden geminimaliseerd. Wat betreft geheugen-efficiënte implementaties is Kyber veel kansrijker dan andere kandidaten.

Hoofdstuk 4. Het volgende hoofdstuk presenteert implementaties van het post-quantum-handtekeningensysteem Dilithium voor de Arm Cortex-M3 en de M4. De focus ligt weer op de getaltheoretische transformatie en op het reduceren van het geheugengebruik van de implementatie. Op de Arm Cortex-M3 vormt de van data afhankelijke tijdsduur van bepaalde vermenigvuldigingsoperaties een wezenlijke uitdaging. Deze komen veelvuldig voor in de berekeningen van Dilithium. Omdat deze

afhankelijkheid van data informatie lekt over geheime waarden die worden gebruikt in de berekening, moet er een andere manier worden gevonden om de polynoomvermenigvuldigingen te implementeren zonder gebruik te maken van instructies met een variabele tijdsduur. We dragen een methode aan die uitsluitend vermenigvuldigingen met een constante tijdsduur gebruikt. De voornaamste resultaten zijn nieuwe snelheidsrecords voor Dilithium op Arm Cortex-M3 en Arm Cortex-M4, en een uiteenzetting van de verschillende manieren om Dilithium te implementeren met variërende beperkingen op het geheugengebruik.

Hoofdstuk 5. In dit hoofdstuk behandelen we implementaties van een vijftal post-quantum sleutel-inkapselingsmechanismen: RLizard, NTRU-HRSS, NTRU-KEM-743, Saber en Kindi. Dit hoofdstuk focust op vermenigvuldigingen door middel van algoritmen van Toom–Cook en Karatsuba. De voornaamste bijdrage is een codegenerator waarmee code kan worden gegenereerd voor vermenigvuldigingen voor een brede selectie aan parameters. Door deze vermenigvuldigingsroutines in diverse post-quantumsystemen te gebruiken behalen we betere prestaties dan de voormalige state-of-the-art. Voor de meeste van deze systemen is de referentie-implementatie de enige die uitvoerbaar is op de Arm Cortex-M4; voor een aantal van deze systemen is onze implementatie meer dan een factor twintig sneller. Een van de systemen, Saber, is al eerder geoptimaliseerd op de Cortex-M4; de vermenigvuldigingsroutine die we hier presenteren overtreft de vermenigvuldiging uit het eerdere werk met bijna een factor twee. Van de vijf systemen die we hier optimaliseren worden de beste prestaties voor inkapseling en ontkapseling behaald door NTRU-HRSS. Inkapseling kost net meer dan 400 000 kloktikken, en is daarmee meer dan twee keer zo snel als elk van de andere NIST-kandidaten die zijn geoptimaliseerd op de Arm Cortex-M4.

Hoofdstuk 6. Het laatste hoofdstuk van de kern van dit proefschrift verbetert de resultaten van hoofdstuk 5 door het gebruik van de getaltheoretische transformatie in NTRU, LAC en Saber te onderzoeken. We tonen aan dat implementaties die gebruik maken van de getaltheoretische transformatie beter zijn dan Toom–Cook en Karatsuba op de Arm Cortex-M4 en Intel Skylake. Interessant genoeg vergen deze beide platformen een verschillende aanpak: op de Cortex-M4 maken we gebruik van 32-bit-berekeningen, terwijl we op de Intel gebruik maken van twee 16-bit NTT-gebaseerde polynoomvermenigvuldigingen en de producten combineren middels de Chinese reststelling. Vooral bij Saber is de snelheidswinst opvallend. Op de Cortex-M4 is de NTT-gebaseerde matrix-vector-vermenigvuldiging 2.5× sneller dan implementaties die gebruik maken van Toom–Cook-vermenigvuldiging. Bij NTRU is de winst minder indrukwekkend, maar ook hier presteert de NTT-gebaseerde Cortex-M4-implementatie beter voor alle varianten.

About the Author

Matthias was born in Tübingen, Germany, on April 20, 1993. After obtaining his Abitur from Gewerbliche Schule Tübingen in 2012, he started studying applied computer science at DHBW Stuttgart, Germany, in cooperation with Hewlett-Packard. He completed his B.Sc. in computer science in 2015. Matthias then enrolled in the IT security master's program of Technical University Darmstadt while continuing to work for Hewlett-Packard, and later the spin-offs Hewlett Packard Enterprise and DXC Technology. Matthias obtained his master's degree in IT security in 2017. His M.Sc. thesis on "*Physical Attack Vulnerability of Hash-Based Signature Schemes*" was supervised by Johannes Buchmann, Juliane Krämer, and Denis Butin. In 2017, Matthias started his Ph.D. studies on post-quantum cryptography at the University of Surrey, Guildford, United Kingdom, under the supervision of Liqun Chen. In 2018, he moved to Radboud University, Nijmegen, The Netherlands, to continue his studies in the group of Peter Schwabe. Part of his position was funded by the European Commission through the ERC Starting Grant of Peter Schwabe (EPOQUE). In 2020, Matthias joined his supervisor in moving to the Max Planck Institute for Security and Privacy in Bochum, Germany. For most of 2020 and 2021, Matthias was a visiting scholar at Academia Sinica, Taipei, Taiwan, hosted by Bo-Yin Yang. This thesis contains the result of his work from 2017 to 2021.

Academic publications

The following is a list of academic publications that Matthias coauthored (in reverse-chronological order).

11. 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2021. `https://eprint.iacr.org/2021/986`

10. 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2021. https://eprint.iacr.org/2021/995

9. 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 – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. https://eprint.iacr.org/2021/662

8. Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):650–675, 2021. https://eprint.iacr.org/2021/532

7. 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://eprint.iacr.org/2020/1397

6. Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with dummy-operations against fault injection attacks. In *Workshop on Fault Detection and Tolerance in Cryptography*, pages 57–65, 2020. https://eprint.iacr.org/2020/1005

5. Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. https://eprint.iacr.org/2020/1278

4. Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):243–268, 2021. https://eprint.iacr.org/2020/371

3. Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology – Africacrypt 2019*, LNCS, pages 209–228. Springer, 2019. https://eprint.iacr.org/2019/489

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

LNCS, pages 281–301. Springer, 2019. https://eprint.iacr.org/2018/1018

1. Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2018*, pages 168–188. Springer, 2018. https://eprint.iacr.org/2018/673

Technical publications

The following is a list of technical publications that Matthias coauthored (in reverse-chronological order).

3. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NISTPQC on ARM Cortex-M4. In *Second NIST PQC Standardization Conference*, 2019. https://eprint.iacr.org/2019/844

2. Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. In *Kangacrypt*, 2018. https://eprint.iacr.org/2018/674

1. Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://www.pqcrainbow.org/