RUHR UNIVERSITÄT BOCHUM

RUHR-UNIVERSITÄT BOCHUM

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

Optimizing the post-quantum signature scheme HAWK for Cortex-M4

Christian Böhm

Masters's Thesis – December 11, 2024.
Max Planck Institute for Security and Privacy (MPI-SP)

1st Supervisor: Prof. Dr. Peter Schwabe
2nd Supervisor: Prof. Dr. -Ing. Tim Güneysu

Abstract

With the rise of quantum computers, widely deployed public-key cryptographic schemes are at significant risk of compromise. To address this, post-quantum cryptography has emerged as a vital area of research, aiming to develop cryptographic algorithms resistant to quantum attacks. HAWK, a lattice-based post-quantum signature scheme, is one of the 14 candidates advancing to the second round of the National Institute of Standards and Technology (NIST) Additional Digital Signatures for Post-Quantum Cryptography (PQC) Standardization Process, announced in October 2024. Its inclusion highlights its relevance in the evolving cryptographic landscape.

This thesis focuses on optimizing the sign routine of HAWK-256. While Keccak-related functions are already optimized in software and do not present further optimization opportunities for the scope of this thesis, the remaining components heavily rely on polynomial arithmetic, which is generally expensive and thus a prime target for optimization. Notably, this work achieves a 5.10x speedup for the Number Theoretic Transform (NTT), one of the most performance-critical functions in the entire scheme. Furthermore, the sign process achieves a speedup of 2.78x when excluding functions related to SHAKE.

Our implementation is written in the Jasmin programming language, specifically designed for high-assurance and high-speed cryptography. This choice not only facilitates efficient implementation but also allows us to prove that our optimizations maintain constant-time execution, a critical property for cryptographic security. By leveraging architecture-specific techniques tailored for the ARM Cortex-M4 processor, this work contributes to the efficient implementation of post-quantum signature schemes on resource-constrained embedded systems, advancing the feasibility of secure post-quantum cryptography in practical applications.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or a similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every part of the text that constitutes quotes in original wording or in its essence has been explicitly referred to by me by using official marking and proper quotation. This is also valid for used drafts, pictures, and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Datum	Unterschrift

Contents

Acronyms **5**

List of Figures **9**

List of Tables **11**

List of Algorithms **13**

1 Introduction **15**
1.1 Motivation 15
1.2 Related Work 17
1.3 Contributions of This Thesis 19
1.4 Outline of This Thesis 20

2 Preliminaries **21**
2.1 Digital Signatures 21
2.1.1 Digital Signature Scheme 22
2.1.2 Hash-then-Sign Paradigm 23
2.1.3 Formal Security Definitions for Signature Schemes 23
2.2 Quantum Supremacy 25
2.2.1 Qubits & Superposition 26
2.2.2 Entanglement 26
2.2.3 Shor's Algorithm 27
2.3 Lattice-based Cryptography 30
2.3.1 Basis Representation of Lattices 30
2.3.2 Cryptography with Lattices 32
2.3.3 Lattice Isomorphism Problem 34
2.4 HAWK 35
2.4.1 Advantages of HAWK 35
2.4.2 Background of HAWK 36
2.4.3 Introduction to HAWK 36
2.4.4 Key Pairs 37
2.4.5 Algorithmic Overview of HAWK 38
2.5 Polynomial Arithmetic 41
2.5.1 Naive Multiplication 41
2.5.2 Multiplication Leveraging the NTT 42

2.5.3 Fast NTT . . . 43
2.6 Modular Multiplication and Reduction Techniques . . . 46
2.6.1 Montgomery Multiplication . . . 46
2.6.2 Lazy Reduction . . . 48
2.7 Jasmin . . . 49
2.7.1 Fundamentals of Jasmin Syntax and Usage . . . 50
2.8 Arm Cortex-M4 . . . 52
2.8.1 Architecture Overview . . . 52
2.8.2 Development Board Overview . . . 56

3 HAWK on Cortex-M4 57
3.1 The pqm4 Library . . . 57
3.2 Benchmarking . . . 57
3.2.1 Methodology . . . 58
3.2.2 Benchmarking Results . . . 59
3.3 Profiling . . . 60
3.3.1 Methodology . . . 60
3.3.2 Profiling Results . . . 61

4 Optimizations 65
4.1 Fast Constant-Time NTTs . . . 65
4.1.1 Polynomial Representation on Cortex-M4 . . . 66
4.1.2 32-bit Arithmetic . . . 66
4.1.3 Pre-computation of Twiddle Factors . . . 67
4.1.4 Layer Merging . . . 68
4.1.5 Use of Incomplete NTTs . . . 70
4.1.6 Performance-Driven Implementation Choices . . . 73
4.2 Other Routines . . . 76
4.2.1 Memory Operations . . . 76
4.2.2 Unrolling Loops . . . 76
4.2.3 Merging `poly_snorm` and `poly_symbreak` . . . 77
4.2.4 Optimizing Leaf Functions in `basis_m2_mul` . . . 77

5 Results 79
5.1 Integrating Jasmin with HAWK-256 on Cortex-M4 . . . 79
5.2 Testing . . . 80
5.3 Benchmarking & Profiling . . . 80
5.3.1 Benchmarking . . . 80
5.3.2 Profiling . . . 81
5.4 Cycle Count Evaluation . . . 84
5.4.1 Cycle Analysis of the NTT Implementation . . . 84
5.4.2 Cycle Analysis of the INTT Implementation . . . 87
5.5 Comparison . . . 88

5.6 Security Evaluation 90
5.6.1 Constant-Time Type System 90
5.7 Implementation Details and Limitations 92

6 Conclusion **93**
6.1 Future Work 93

Bibliography **95**

A Appendix **105**

Acronyms

ABI Application Binary Interface

CPU Central Processing Units

CRT Chinese Remainder Theorem

DFT Discrete Fourier Transform

DGS Discrete Gaussian Sampling

DHKE Diffie-Hellman Key Exchange

DLP Discrete Logarithm Problem

DSP Digital Signal Processing

ECC Elliptic-Curve Cryptography

EUF-CMA Existential Unforgeability under Chosen Message Attacks

FFT Fast Fourier Transform

FPU Floating Point Unit

IoT Internet of Things

ISA Instruction Set Architecture

INTT Inverse Number Theoretic Transform

KEM Key Encapsulation Mechanisms

LIP Lattice Isomorphism Problem

LWE Learning With Errors

NIST National Institute of Standards and Technology

NTT Number Theoretic Transform

PKI Public Key Infrastructures

PQC Post-Quantum Cryptography

QFT Quantum Fourier Transform

RAM Random-Access Memory

RISC Reduced Instruction Set Computer

RNG Random Number Generator

SIMD Single Instruction Multiple Data

SSL Secure Sockets Layer

SUF-CMA Strong Unforgeability under Chosen Message Attacks

SVP Shortest Vector Problem

TLS Transport Layer Security

USB Universal Serial Bus

List of Figures

2.1 Digital signatures in software distribution 21
2.2 EUF-CMA security of a signature scheme $\mathcal{S} = (\mathit{Gen}, \mathit{Sign}, \mathit{Vrfy})$, following [22, Fig. 1. (left)] 24
2.3 SUF-CMA security of a signature scheme $\mathcal{S} = (\mathit{Gen}, \mathit{Sign}, \mathit{Vrfy})$, following [22, Fig. 1. (right)] 24
2.4 Probability distribution of first register 29
2.5 Lattice examples 31
2.6 Good and bad bases for lattice representation 32
2.7 Encrypting a message 32
2.8 Decrypting a message 33
2.9 New perspective 34
2.10 Signature creation 37
2.11 Butterfly constructions 43

3.1 Profiling results for Function 7 divided into three categories. 63

4.1 Layers of NTT for Hawk-256 68
4.2 First two layers of NTT for Hawk-256 68
4.3 Merging first two layers of NTT 69
4.4 Merging layers of XOR computations 77

5.1 Profiling results for optimized Function 7 divided into three categories 83

List of Tables

3.1 Performance metrics for key generation, signing, and verification routines of HAWK-256. 59
3.2 Profiling results for major routines in HAWK-256's sign routine over 10 000 iterations, average coefficient of variation of 0,012 %. 61
3.3 Profiling results for Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations. . . 62

5.1 Performance metrics for signing routine of HAWK-256 80
5.2 Profiling results for major routines in HAWK-256's sign routine over 10 000 iterations 81
5.3 Profiling results for optimized Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations 82
5.4 Comparison of performance metrics for signing routine of HAWK-256 88
5.5 Comparison of the profiling results for major routines in HAWK-256's sign routine over 10 000 iterations 89
5.6 Comparison of profiling results for Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations 90

List of Algorithms

1 High level HawkKeyGen, following [20, Algorithm Sketch 1] 38
2 High level HawkSign, following [20, Algorithm Sketch 2] 39
3 High level HawkVerify, following [20, Algorithm Sketch 3] 40
4 CT FFT implementing NTT, following [53, Algorithm 3] 45
5 GS FFT implementing INTT, following [53, Algorithm 4] 45
6 Montgomery reduction `montyR`, following [63, Algorithm REDC] . . . 47
7 Signed Montgomery reduction, following [73, Algorithm 3] 48

8 montgomeryR, following [25, Artifacts]^1 71
9 4 x 4 schoolbook, following [25, Algorithm 17] 72

1 Introduction

1.1 Motivation

In the ever-evolving landscape of cybersecurity, the advent of quantum computing poses significant threats to traditional cryptographic systems. Public Key Infrastructures (PKIs), which form the backbone of secure communications and data protection, are particularly vulnerable. Quantum algorithms, such as Shor's algorithm, proposed by Peter Shor in 1994, have the potential to break widely used cryptographic schemes like RSA and ones based on Elliptic-Curve Cryptography (ECC) [74]. Shor's algorithm leverages the ability of quantum computers to perform certain types of calculations exponentially faster than classical computers, particularly factoring large integers and computing discrete logarithms. The urgency to develop and optimize quantum-resistant alternatives has never been more pressing, given that sufficiently powerful quantum computers could render current cryptographic standards obsolete. This is further emphasized by a report published by NIST in 2016, which highlights that quantum technology could compromise the widely used RSA algorithm by 2030 [23].

Recognizing this imminent threat, NIST initiated its PQC standardization process in 2016. Initially, 23 signature schemes and 59 Key Encapsulation Mechanisms (KEMs) were submitted.
In July 2022, NIST announced the first group of winners to be standardized from its six-year PQC competition after the third round. The selected algorithms include one KEM with CRYSTALS-Kyber [19] and three signature schemes represented by CRYSTALS-Dilithium [33], FALCON [70], and SPHINCS+ [51].
Four additional KEMs continued into the fourth round. However, as there were no remaining digital signature candidates NIST issued a call for additional digital signature proposals in September 2022. NIST was primarily interested in signature schemes not based on structured lattices as two of the three signature schemes, CRYSTALS-Dilithium and FALCON are based on lattice-based cryptography. Nevertheless, NIST was open to lattice-based submissions that could significantly outperform CRYSTALS-Dilithium and FALCON in practical scenarios while maintaining robust security characteristics. One notable example is HAWK [34], a signature scheme inspired by the Lattice Isomorphism Problem (LIP) which demonstrates a fourfold increase in signing speed on x86 architectures and a fifteenfold increase in speed when floating-point operations are unavailable compared to FALCON [34].

In the first round of the PQC standardization process, performance did not play a significant role, as the focus was primarily on the security and theoretical robustness of the candidate algorithms. This approach shifted in the second round, where performance became more important [7]. Initially, implementations were geared towards Intel processors with AVX2 instructions, which are not representative of the majority of cryptographic devices in use today. Most cryptographic applications operate on much smaller devices with limited RAM, no or limited vector instructions, and constrained flash memory. This raises essential questions about whether these schemes can fit within the limited resources of such devices and how efficiently they can perform. The ARM Cortex-M4, a widely used processor in embedded systems, emerged as a key target for these performance evaluations. Its relevance stems from its widespread use in Internet of Things (IoT) and other low-power applications, making it crucial to ensure that post-quantum cryptographic algorithms can be efficiently implemented on these smaller, resource-constrained platforms. As a result, NIST recommends the Cortex-M4 for PQC evaluation.

Regardless of this focus, there has been little to no work done on efficiently implementing the HAWK signature scheme for the Cortex-M4. This gap in research motivates this thesis, which aims to optimize and evaluate the performance of HAWK on the Cortex-M4, addressing both the practical feasibility and the efficiency of deploying a post-quantum signature scheme on constrained devices.

1.2 Related Work

This section reviews prior work relevant to efficient implementations of PQC on resource-constrained devices, with a particular focus on lattice-based schemes and optimization techniques involving the NTT. While no prior research directly addresses Hawk on the ARM Cortex-M4, existing works on lattice-based PQC provide valuable insights that guide the optimization strategies developed in this thesis.

Introduction to Lattice-Based Cryptography Research on Constrained Devices

Lattice-based schemes dominate as candidates in the NIST PQC competition, owing to their balance between security and efficiency. For instance, [45] explores lattice-based signature schemes such as GLP, BLISS, and Crystals-Dilithium, presenting optimized implementations for ARM Cortex-M4 microcontrollers. By discussing the unique properties and implementation challenges of these schemes, the work underscores the feasibility of lattice-based cryptography for practical applications in a post-quantum era.
Similarly, [56] evaluates the practicality of deploying lattice-based schemes on constrained devices, benchmarking state-of-the-art implementations in terms of area, power, and performance metrics critical for IoT applications.
These research efforts highlight the growing importance of performance and practicality in lattice-based cryptography and provide valuable context and inspiration for the strategies explored in this thesis.

NTT Optimizations for PQC on Cortex-M4

The NTT is a critical subroutine in many lattice-based schemes, and optimizing it has been a key focus of research for constrained devices like the ARM Cortex-M4. For instance, [21] achieves an 18 % improvement in Crystals-Kyber's performance on Cortex-M4 by doubling the speed of the NTT. Similarly, [2] improves both Kyber and Dilithium by implementing faster NTT and Inverse Number Theoretic Transform (INTT) routines, achieving speedups of approximately 5 % compared to the prior state-of-the-art.
In [42], the authors accelerate Dilithium by optimizing the NTT, achieving overall speedups of 7 %, 15 %, and 9 % for key generation, signing, and verification, respectively.
In addition to Kyber and Dilithium, [25] demonstrates the advantages of NTT-based polynomial multiplication over Toom–Cook for Saber on Cortex-M4, achieving a 61 % faster matrix-vector multiplication and a 22 % reduction in encapsulation cycles. Building on this, [1] extends these optimizations to masked implementations, showing 16 %–41 % speedups while maintaining similar stack usage.
The KEM NewHope has also been optimized for embedded processors like the

Cortex-M4. The authors of [9] demonstrate the efficiency of their NTT implementation, which is twice as fast as prior work, contributing to their original NewHope proposal [8].
Another strategy is presented in [24], which focuses on optimizing the PQC scheme Falcon. The authors utilize signed Plantard arithmetic instead of unsigned Montgomery arithmetic for modular multiplication, leveraging the NTT along with Cortex-M4's Digital Signal Processing (DSP) instructions. This approach achieves a 75 % speedup in the verify routine compared to existing implementations. Similarly, [50] improves Plantard arithmetic within the context of the NTT for lattice-based cryptography on the Cortex-M4, resulting in faster implementations of both Kyber and NTTRU.
While the technique of using Plantard instead of Montgomery presents a promising avenue for potential performance improvements for Hawk, it is currently not feasible to implement in this work due to limitations of the Jasmin programming language, which currently does not support all needed DSP instructions.
Overall, these works demonstrate the central role of NTT optimizations in enhancing performance on constrained devices and serve as groundwork for the strategies employed in this thesis. Drawing on applicable techniques from prior work, including incomplete NTTs and layer merging, this thesis incorporates these optimizations into the NTT to enhance the efficiency of our Hawk implementation.

Framework for Benchmarking and Testing

Another significant contribution is the pqm4 project [54]. It stands out by providing a comprehensive framework for testing and benchmarking NIST PQC candidates on the Cortex-M4. This tool enables a standardized and systematic evaluation of various post-quantum schemes under consistent conditions, facilitating direct comparisons and highlighting the practical performance aspects across different implementations. The pqm4 project thus serves as a crucial resource for the post-quantum cryptography community and this thesis, offering insights that extend beyond individual optimizations to encompass the broader landscape of PQC performance on constrained devices.

1.3 Contributions of This Thesis

The transition to PQC is a critical step in securing modern communication systems against the threat of quantum computers. This thesis contributes to this transition by focusing on the optimization of the Hawk-256 signature scheme, a lattice-based cryptographic algorithm that has advanced to the second round of the NIST Additional Digital Signatures for PQC Standardization Process in October 2024. The contributions of this work are outlined below.

Performance Analysis of the Reference Implementation

To identify optimization opportunities, this thesis conducts an in-depth performance analysis of the reference implementation of Hawk-256. The analysis profiles the computational costs of the sign routine, highlighting the distribution of cycles across various functions. This includes the identification of computational bottlenecks.

Optimized Sign Routine

Building on the performance analysis, this thesis develops an optimized and high-assurance implementation of the sign routine for Hawk-256. The optimizations focus on improving the efficiency of polynomial arithmetic, with a particular emphasis on the NTT and INTT, which are central to the computational workload. To ensure both high performance and high assurance, the implementation is written in the Jasmin language, which features a constant-time type system. This security type system helps to enforce the absence of timing variations caused by secret-dependent branches or memory accesses, effectively addressing critical security concerns and mitigating the risk of side-channel attacks. In parallel, the implementation leverages architecture-specific features of the ARM Cortex-M4 processor to optimize performance and efficiency.

Comparison of Optimized Sign Routine to the Reference

A comparison between the optimized sign routine and the reference implementation demonstrates significant performance improvements. The results highlight the effectiveness of the optimization strategies applied in this work, particularly in addressing computational bottlenecks and enhancing the feasibility of deploying post-quantum cryptography on embedded systems.

1.4 Outline of This Thesis

Chapter 2 lays the groundwork for the thesis by introducing key concepts and tools. It starts with an overview of digital signatures and their importance in secure communication. The chapter then explains the impact of quantum computing on cryptography and the role of lattice-based cryptography as a post-quantum solution. A detailed introduction to HAWK, the signature scheme central to this thesis, is provided, along with a discussion of the computational techniques required for its implementation, such as polynomial arithmetic and modular reduction. Finally, the chapter highlights the Jasmin programming language and the ARM Cortex-M4 architecture, both of which are pivotal to the optimization strategies presented in this work.

Chapter 3 focuses on the reference implementation of HAWK on the ARM Cortex-M4 processor. It begins by introducing the pqm4 library, which serves as the foundation for running and testing HAWK on the Cortex-M4, including benchmarking. The chapter then outlines the benchmarking methodology and presents the results of the implementation, providing valuable insights into the performance of the HAWK reference implementation. Following this, a detailed profiling section is included, where the methodology for performance profiling is explained and the results are discussed, with an emphasis on identifying areas for optimization. In summary, this chapter offers a comprehensive performance analysis of the HAWK sign implementation, combining benchmarking and profiling to pinpoint key areas for optimization.

Chapter 4 focuses on the various optimization techniques applied to improve the performance of HAWK on the ARM Cortex-M4 processor. The chapter begins with an overview of the optimizations made to the critical NTT routine, followed by a discussion of optimizations to other key areas, including memory operations, loop unrolling, and specific polynomial routines. These enhancements collectively aim to reduce the computational cost of the HAWK sign routine and improve overall execution efficiency, providing significant performance improvements for the signature scheme.

Chapter 5 presents the results of the optimizations and implementation efforts for HAWK on the ARM Cortex-M4. The chapter begins with an exploration of how our Jasmin implementation is integrated with HAWK-256. It then discusses the testing methodology and the results of the benchmarking and profiling processes, highlighting the performance improvements achieved through the optimizations. A detailed comparison of the optimized implementation with the reference version follows, showing the impact of the changes. Finally, the chapter covers the security evaluation of the optimized implementation, including the use of a constant-time type system to ensure the implementation meets constant-time execution requirements.

2 Preliminaries

2.1 Digital Signatures

Digital signatures are cryptographic mechanisms used to ensure the authenticity and integrity of digital messages or documents. They play a critical role in various security protocols, including Secure Sockets Layer (SSL)/Transport Layer Security (TLS), and are fundamental to the functioning of PKIs [55].

Digital signatures provide the following security properties [36]:

- **Message Integrity:** The recipient of the message must be certain that the message has not been tampered with since its transmission.
- **Proof Of Origin:** The recipient of the message must be assured of its source.
- **Non-Repudiation:** One of the parties in a transaction should not be able to disavow their participation at a later time.

To understand how digital signatures achieve these security properties, it is essential to understand their operational mechanic. For instance, a simple use case involving software distribution is shown (cf. Figure 2.1).

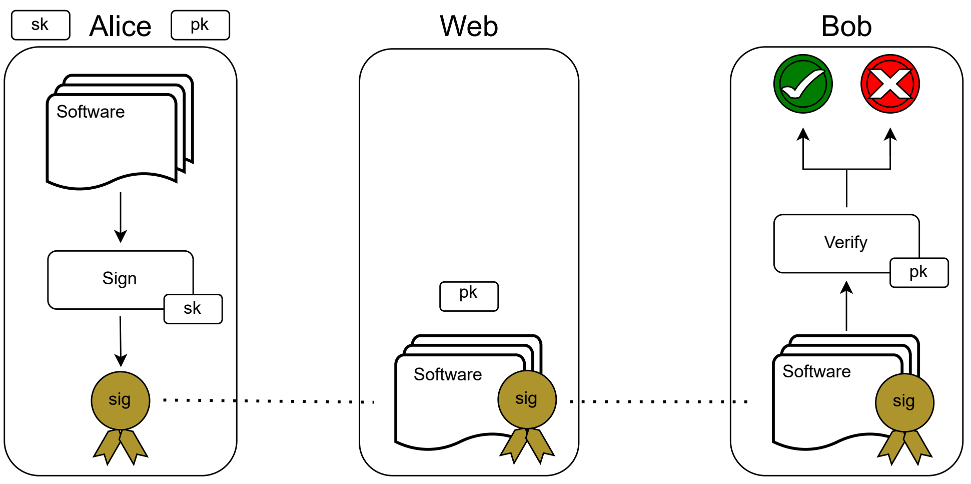

Figure 2.1: Digital signatures in software distribution

Alice, holding a secret key *sk* and a public key *pk*, intends to distribute her software over the internet securely. She signs her software package with her private key, generating a digital signature *sig*. Alice then uploads both the signed software package and her public key to the web. When Bob downloads the signed software package, he uses Alice's public key to verify the signature, ensuring that the software is indeed from Alice and has not been tampered with.

2.1.1 Digital Signature Scheme

Definition 1 (Digital Signature Scheme, following [55, Definition 12.1])

A digital signature scheme comprises three probabilistic polynomial-time algorithms: *(Gen, Sign, Vrfy)* such that:

1. *Gen*, the key-generation algorithm accepts a security parameter 1^n as input and generates a key pair *(pk, sk)*. The pair consists of the public key and the private key, respectively. We assume that *pk* and *sk* each have lengths of at least n, and that n can be deduced from *pk* or *sk*.
2. *Sign*, the signing algorithm accepts a private key *sk* and a message m from some message space as inputs. It generates a signature σ that we denote as $\sigma \leftarrow Sign_{sk}(m)$.
3. *Vrfy*, the deterministic verification algorithm accepts a public key *pk*, a message m, and a signature σ as inputs. It generates a bit b, which is 1 if the signature is valid and 0 if it is invalid.

Additionally, it is required that $Vrfy_{pk}(m, Sign_{sk}(m)) = 1$ holds for every message m except with negligible probability over *(pk, sk)*.

2.1.2 Hash-then-Sign Paradigm

The hash-then-sign method is a cryptographic approach where a message is first hashed using a cryptographic hash function, and then the resulting hash is signed with the signer's private key. This technique improves efficiency by reducing the data size for the signing process.

Definition 2 (Hash-then-Sign Paradigm, following [55, Construction 12.3][1])

Let Π = *(Gen, Sign, Vrfy)* denote a signature scheme for messages of length $l(n)$, and let H be a hash function with output length $l(n)$. Compose the signature scheme Π' = *(Gen', Sign', Vrfy')* as shown:

1. *Gen'*, the key-generation algorithm accepts a security parameter 1^n as input and generates *(pk, sk)* by running $Gen(1^n)$. The key pair consists of the public key with pk and the private key with sk.
2. *Sign'*, the signing algorithm accepts a private key sk and a message $m \in \{0,1\}^*$ as inputs. It generates a signature σ that we denote as $\sigma \leftarrow Sign_{sk}(H(m))$.
3. *Vrfy'*, the verification algorithm accepts a public key pk, a message $m \in \{0,1\}^*$, and a signature σ as inputs. It generates a bit b, which is 1 if the signature is valid and 0 if it is invalid. The signature is only valid when $Vrfy_{pk}(H(m), \sigma) \overset{?}{=} 1$.

2.1.3 Formal Security Definitions for Signature Schemes

In this section, we introduce the formal security definitions that are commonly used to assess the security of digital signature schemes. Two key security notions, Existential Unforgeability under Chosen Message Attacks (EUF-CMA) and Strong Unforgeability under Chosen Message Attacks (SUF-CMA), serve as the foundation for evaluating the robustness of a signature scheme against potential attacks.

EUF-CMA requires that an adversary cannot construct a signature for a message that the key owner did not sign previously. This ensures that even with access to valid signatures for other messages, the attacker cannot forge a signature for a new message [22].

A stronger property that can be met by signature schemes is SUF-CMA. This property not only prevents the adversary from constructing a signature for an unsigned message but also ensures that the adversary cannot generate an alternative signature for a given message that has already been signed [22].

[1]The original definition in [55, Construction 12.3] employs a keyed hash function. For simplicity and consistency with the context of this work, the definition has been adapted to use a regular hash function instead.

```
$\mathsf{G}^{\mathsf{euf\text{-}cma}}_{\mathcal{S},\mathcal{A}}(\mathsf{pp})$:
1   $\mathcal{L}_{Sign} \leftarrow \emptyset$
2   $(pk, sk) \xleftarrow{\$} Gen(\mathsf{pp})$
3   $(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}}}(pk)$
4   return $[\![ Vrfy(pk, \sigma^*, m^*) \wedge m^* \notin \mathcal{L}_{Sign} ]\!]$

$\mathcal{O}_{\mathsf{Sign}}(m)$:
5   $\sigma \xleftarrow{\$} Sign(sk, m)$
6   $\mathcal{L}_{Sign} \leftarrow \mathcal{L}_{Sign} \cup \{m\}$
7   return $\sigma$
```

Figure 2.2: EUF-CMA security of a signature scheme $\mathcal{S} = (Gen, Sign, Vrfy)$, following [22, Fig. 1. (left)]

```
$\mathsf{G}^{\mathsf{suf\text{-}cma}}_{\mathcal{S},\mathcal{A}}(\mathsf{pp})$:
1   $\mathcal{L}_{Sign} \leftarrow \emptyset$
2   $(pk, sk) \xleftarrow{\$} Gen(\mathsf{pp})$
3   $(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}}}(pk)$
4   return $[\![ Vrfy(pk, \sigma^*, m^*) \wedge (m^*, \sigma^*) \notin \mathcal{L}_{Sign} ]\!]$

$\mathcal{O}_{\mathsf{Sign}}(m)$:
5   $\sigma \xleftarrow{\$} Sign(sk, m)$
6   $\mathcal{L}_{Sign} \leftarrow \mathcal{L}_{Sign} \cup \{(m, \sigma)\}$
7   return $\sigma$
```

Figure 2.3: SUF-CMA security of a signature scheme $\mathcal{S} = (Gen, Sign, Vrfy)$, following [22, Fig. 1. (right)]

Definition 3 (EUF-CMA & SUF-CMA security, following [22, Definition 2])

Let $\mathcal{S} = (Gen, Sign, Vrfy)$ be a signature scheme. Consider the security games

$$\mathsf{G}^{\mathsf{euf\text{-}cma}}_{\mathcal{S},\mathcal{A}} \quad \text{and} \quad \mathsf{G}^{\mathsf{suf\text{-}cma}}_{\mathcal{S},\mathcal{A}}$$

as defined in Figure 2.2 and Figure 2.3.

We say that a signature scheme $\mathcal{S}$ is (t, ϵ, Q_s)-EUF-CMA-secure or existentially unforgeable under chosen message attacks, if for any adversary $\mathcal{A}$ running in time at most t, making at most Q_s queries to the signing oracle, the probability

$$\Pr\left[\mathsf{G}^{\mathsf{euf\text{-}cma}}_{\mathcal{S},\mathcal{A}}(\mathsf{pp}) = 1\right] \leq \epsilon.$$

Analogously, we say that $\mathcal{S}$ is (t, ϵ, Q_s)-SUF-CMA-secure or strongly unforgeable under chosen message attacks.

2.2 Quantum Supremacy

Quantum supremacy refers to the point at which a quantum computer can solve a problem that classical computers cannot solve in a feasible amount of time [69]. This concept is significant because it demonstrates the potential of quantum computers to outperform classical computers in specific tasks. However, quantum superiority does not necessarily imply that quantum computers are universally faster than classical ones. Rather, they can handle certain problems far more efficiently. This can be attributed to quantum features such as superposition and entanglement.
The concept of quantum supremacy was popularized by a seminal experiment conducted by Google in 2019, where their quantum processor, Sycamore, completed a computation in 200 seconds that would take the world's most powerful supercomputer at that time approximately 10 000 years to solve [17].
In the realm of cryptography, two algorithms of particular interest regarding quantum supremacy are Grover's algorithm [43] and Shor's algorithm [74].
Grover's algorithm significantly impacts cryptography by providing a quadratic speedup for search tasks, including brute-force attacks. This improvement means that symmetric-key encryption is effectively less secure, as a 128-bit key, for example, offers only the security of a 64-bit key against such attacks. Consequently, cryptographic systems may need to use longer key lengths to maintain security in the face of quantum threats posed by Grover's algorithm.
Shor's algorithm has a profound impact on cryptography by providing a dramatic speedup in the factoring of large numbers or calculating discrete logarithms, tasks that underpin the security of widely used public-key cryptosystems like RSA or Diffie-Hellman Key Exchange (DHKE). While classical algorithms require exponential time to factorize large integers or calculate discrete logarithms, Shor's algorithm achieves this in polynomial time using quantum computing. This substantial speedup means that cryptography based on the Factorization Problem or the Discrete Logarithm Problem (DLP) could be compromised by quantum attacks, thus, posing an even bigger threat than Grover's algorithm.
Although these algorithms are highly impactful, they cannot yet be applied in practice because current quantum computers are not yet advanced enough.

2.2.1 Qubits & Superposition

Quantum computing represents a paradigm shift from classical computing by harnessing the principles of quantum mechanics to perform computations in fundamentally new ways. Unlike classical computers, which use bits to represent information as either 0 or 1, quantum computers utilize so-called qubits or quantum bits. A qubit can also exist in the states 0 or 1, represented in Dirac notation as $|0\rangle$ or $|1\rangle$, respectively. The crucial distinction from a classical bit lies in the qubit's ability to exist in a superposition of both states $|0\rangle$ and $|1\rangle$ that can be expressed as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \text{ with } |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

where the qubit $|\psi\rangle$ is the superposition of the states $|0\rangle$ and $|1\rangle$ with amplitudes α and β, where α and β are represented by complex numbers. Thus, $|\psi\rangle$ is an element of a two-dimensional complex vector space, which can also be referred to as a Hilbert space. Physically, the qubit $|\psi\rangle$ represents both states at the same time [58]. While there are no direct classical analogies to quantum superposition, it can be loosely compared to a coin spinning in mid-air, being partially heads and partially tails at the same time. Only when the coin lands is it determined whether it is heads or tails, similar to how a quantum state collapses to a definite outcome upon measurement. In quantum mechanics, bringing quantum information to the classical level requires measuring the qubit. This measurement process inevitably disturbs the qubit's state, causing a non-deterministic collapse of $|\psi\rangle$ to either $|0\rangle$ or $|1\rangle$. The outcome is probabilistic, with the state collapsing to $|0\rangle$ with probability $|\alpha|^2$ or to $|1\rangle$ with probability $|\beta|^2$. The exact values of α and β remain indeterminate until the measurement is made [58].
In summary, superposition allows quantum computers with n qubits to encode and process information across all 2^n possible states at once, vastly surpassing the capabilities of classical computers that can only process information on a single state at once. When measured, both systems yield n bits of classical information.

2.2.2 Entanglement

When qubits are entangled, their states become interdependent, meaning the state of one qubit instantaneously influences the state of another, even if they are far apart. Thus, entanglement indicates a loss of independence among qubits, as their individual states are no longer isolated but are linked together.
For example, consider two entangled qubit registers, $|t\rangle$ and $|d\rangle$. Suppose $|t\rangle$ encodes a date, such as 25.05.1977, and $|d\rangle$ encodes the corresponding day of the week, such as Wednesday. If we measure $|t\rangle$ and observe Friday, due to the entanglement, we now know that measuring $|s\rangle$ will yield a date that falls on a Friday. In this way, by measuring part of an entangled system, we can constrain the remaining solution space.

2.2.3 Shor's Algorithm

Shor's algorithm, published by Peter Shor in 1994, is an algorithm that can solve the factorization problem as well as the DLP by utilizing a quantum computer [74]. To gain a deeper insight into Shor's algorithm, we will explore its application to integer factorization. The algorithm can factorize an L-bit odd composite integer $N = p \cdot q$, where p and q are prime numbers, using $\mathcal{O}(L^3)$ operations and $\mathcal{O}(L^3)$ quantum gates [76]. The following section presents a simple example of Shor's algorithm in action, inspired by [47].

Consider an odd composite integer $N = 35$. Our goal is to factorize it into its prime factors, which are 5 and 7 in this case. Although this is a simple example, we will use Shor's algorithm to achieve this result, demonstrating the process that becomes crucial for much larger N, where factorization is far from trivial.

1. Choose a random number $r < N$:

$$r = 8$$

2. Determine the order o of r mod N:

$$r^1 = 8,\ r^2 = 29 \bmod 35,\ r^3 = 22 \bmod 35,\ \underline{r^4 = 1 \bmod 35}$$

$$\longrightarrow \text{ order } o = 4$$

3. Verify that o is even and $r^{\frac{o}{2}} + 1 \neq 0 \bmod N$:

$$o = 4 \text{ is even } \checkmark$$

$$r^{\frac{o}{2}} + 1 = 8^{\frac{4}{2}} + 1 = 8^2 + 1 = 30 \neq 0 \bmod 35 \checkmark$$

4. Factor N as $p = gcd(r^{\frac{o}{2}} - 1, N)$ and $q = gcd(r^{\frac{o}{2}} + 1, N)$:

$$p = gcd(28, 35) = 7 \qquad q = gcd(30, 35) = 5$$

Examining the components of Shor's algorithm, we find that steps 1, 3, and 4 can be solved efficiently using classical computers. However, step 2—finding the order of an element in a multiplicative group modulo N—remains a challenging problem. Thus, the problem of factorization can be reduced to the problem of order-finding [74, 76]. Moreover, the problem of order-finding can be further reduced to the problem of finding the period of any number in the sequence r^i mod N [47].

To demonstrate how this problem can be efficiently addressed by quantum computers, we will examine a simplified depiction of Quantum Period Finding:

1. Construct two qubit registers of suitable size.
2. Concatenate both registers to obtain the quantum state s:
$$|s\rangle = |0\rangle|0\rangle$$
3. Put the first register into a superposition of integers i:
$$|s\rangle = \sum_i c \cdot |i\rangle|0\rangle$$
4. Entangle the second register to the associated $r^i \bmod N$:
$$|s\rangle = \sum_i c \cdot |i\rangle|r^i \bmod N\rangle$$
$$= c \cdot |0\rangle|1\rangle + c \cdot |1\rangle|8\rangle + c \cdot |2\rangle|29\rangle + c \cdot |3\rangle|22\rangle + c \cdot |4\rangle|1\rangle + ...$$
All amplitudes are equal, thus, all states are equally probable.
5. Measure the second register and denote the result as x:

For example, we measure $x = 1$.
$$|s\rangle = c' \cdot |0\rangle|1\rangle + 0 \cdot |1\rangle|8\rangle + 0 \cdot |2\rangle|29\rangle + 0 \cdot |3\rangle|22\rangle + c' \cdot |4\rangle|1\rangle + ...$$
Due to entanglement, any state where the second register does not equal x has a probability of zero.
$$|s\rangle = c' \cdot |0\rangle|1\rangle + c' \cdot |4\rangle|1\rangle + +c' \cdot |8\rangle|1\rangle + +c' \cdot |12\rangle|1\rangle + ...$$
No matter the value we measure in step 5, the remaining values in the first registers will consistently have a distance of 4.

Generally speaking, because the period is a global property, the spikes are always equally spaced, with only the initial offset being unknown (cf. Figure 2.4).

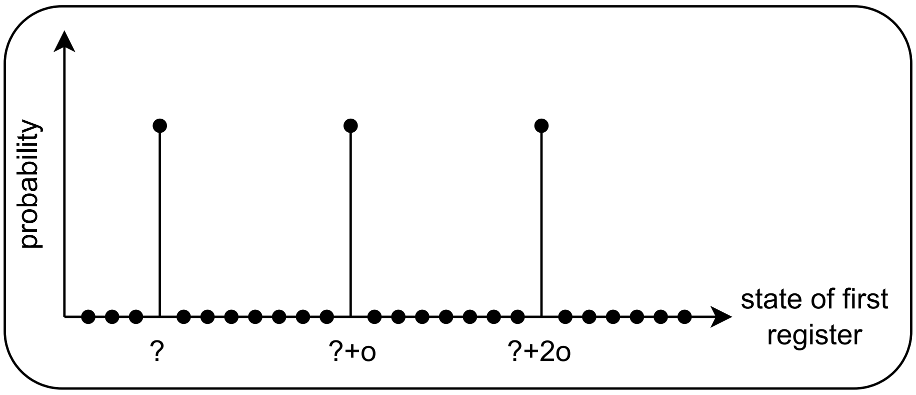

Figure 2.4: Probability distribution of first register

In simplified terms, to recover the order o, we measure a few times. Regardless of the distribution shift, the peaks recur with a period of o, thus, having a frequency of $\mathrm{f} = \frac{1}{o}$. To analyze frequencies, with the specific aim of retrieving the order o, we employ the Fourier transform. However, because the classical Fourier transform and its fast variant, the Fast Fourier Transform (FFT), exhibit exponential complexity, we turn to the Quantum Fourier Transform (QFT), which operates with polynomial complexity [58, 47].
A more detailed explanation of the QFT can be found in [58].

In summary, Shor's algorithm can break asymmetric cryptography schemes such as RSA by reducing the underlying problem—specifically, the factorization problem in the case of RSA—to the problem of order finding. This problem can be efficiently solved on quantum computers due to the Quantum Fourier Transform.

However, in the case of RSA with a 1 024-bit key, an attacker would require 2 050 qubits and approximately $5.81 \cdot 10^{11}$ quantum gates, which is currently infeasible with existing technology [71].

2.3 Lattice-based Cryptography

In recent years, lattices have gained popularity in the field of PQC. This is due to the fact, that lattice-based cryptography is a promising alternative for public-key cryptography, particularly if quantum computers become practical or if significant advancements are made in solving the Factorization Problem or the DLP. Despite their simplicity, these structures conceal an intricate geometrical complexity that has been extensively explored over the past few decades [46]. This exploration began in 1982 with the discovery of the renowned LLL algorithm [59] and gained further momentum by Ajtai's cryptographic application of lattices in 1996 [5] which was later used to construct the first lattice-based cryptographic scheme in 1997 [6].
Lattice-based cryptography leverages the complexity of certain computational problems associated with lattices, such as the Shortest Vector Problem (SVP), the Learning With Errors (LWE) problem, or the Lattice Isomorphism Problem (LIP) to create secure cryptographic systems.

A lattice is a grid of points in multidimensional space with a periodic structure.
A lattice with n dimensions and n basis vectors is called a full-rank lattice.

Definition 4 (Lattice, following [28, Definition 1])

$\mathcal{L} \subseteq \mathbb{R}^n$ is a *lattice* if $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$. $\mathcal{L}$ is a rank k lattice, or k-dimensional, if $\dim(\mathcal{L}) = k$. $\mathcal{L}$ is called *full-rank* if $\dim(\mathcal{L}) = n$.

In this context, $\mathcal{L}$ is discrete if the inherited topology on $\mathcal{L}$ is discrete, i.e., every subset of $\mathcal{L}$ is open.
For additive subgroups, the following properties have to align:

$$\forall x, y \in \mathcal{L}, x + y \in \mathcal{L}.$$

$$\forall x \in \mathcal{L}, -x \in \mathcal{L}.$$

2.3.1 Basis Representation of Lattices

To describe a lattice, we can make use of a set of linearly independent generators, also known as basis. Formally, if $b_1, b_2, \ldots, b_n$ are linearly independent vectors in $\mathbb{R}^n$, the lattice $\mathcal{L}$ generated by these vectors is:

$$\mathcal{L}(b_1, b_2, ..., b_n) = \left\{ \sum_{i=1}^{n} z_i \mathbf{b}_i \mid z_i \in \mathbb{Z} \right\}.$$

This is referred to as the basis representation [28, Chapter 3].

The points in the lattice are formed by linear combinations of a set of basis vectors with integer coefficients. To further visualize this concept we can take an integer lattice in two dimensions: $\mathbb{Z}^2 = \{(x, y) \mid x, y \in \mathbb{Z}\}$. In Figure 2.5, the left diagram illustrates a basis of $\mathbb{Z}^2$ because the vectors $(1, 0)$ and $(1, 1)$ span the entire grid through their linear combinations, whereas the right diagram does not form a basis of $\mathbb{Z}^2$ because the vectors $(2, 0)$ and $(1, 1)$ fail to generate every point in the grid.

More precisely, for the left diagram, we can say that $\mathbb{Z}^2 = \mathcal{L}(\mathbf{b}_1, \mathbf{b}_2)$ where $\mathbf{b}_1 = (1, 1)$ and $\mathbf{b}_2 = (1, 0)$. Moreover, $\mathbb{Z}^2$ has more than one basis, for example, the basis ((1,0),(0,1)) still generates the same lattice. As a matter of fact, every lattice $\mathcal{L}$ of rank n > 1 has infinitely many distinct bases [28, Chapter 3].

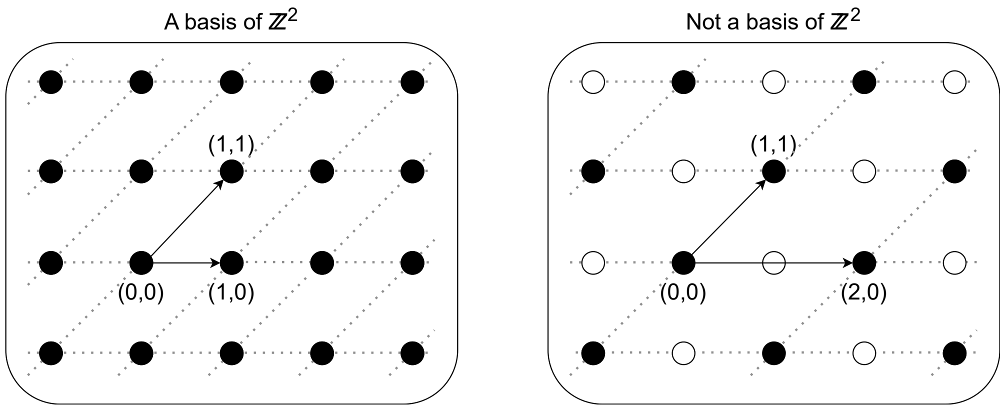

Figure 2.5: Lattice examples

Besides normal lattices that are defined over $\mathbb{R}^n$, there also exists *module lattices.* A module lattice is typically considered over a ring $\mathcal{R}$. If $\mathcal{R}$ is a ring and $b_1, b_2, \ldots, b_n$ are elements of $\mathcal{R}^n$, then the module lattice $\mathcal{M}$ generated by these elements is:

$$\mathcal{M}(b_1, b_2, ..., b_n) = \left\{ \sum_{i=1}^{k} r_i \mathbf{b}_i \mid r_i \in \mathcal{R} \right\}.$$

2.3.2 Cryptography with Lattices

To get a basic understanding of how lattices are used in cryptography we provide a small example of a legacy approach for encryption using lattices in $\mathbb{Z}^2$. Figure 2.6 depicts a good basis on the left side because its vectors are short and relatively orthogonal. On the right side, a bad basis is shown that consists of long vectors that are not orthogonal. The good basis serves as the secret key while the bad basis serves as the public key. Two bases B and B' generate the same lattice if there exists a unimodular matrix $U \in \mathrm{GL}_n(\mathbb{Z})$ such that $B \cdot U = B'$. Transitioning from the bad basis to the good basis is as difficult as solving the SVP.

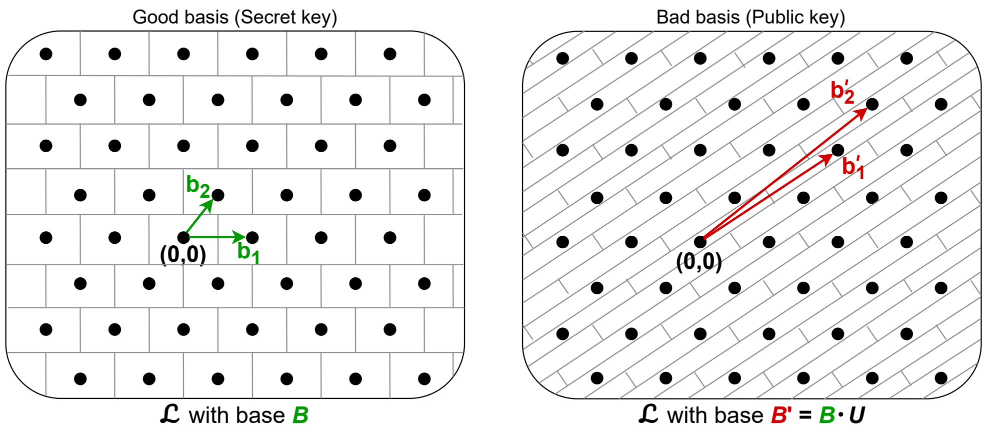

Figure 2.6: Good and bad bases for lattice representation

For encryption, as shown in Figure 2.7, we encode a message m as a lattice point by using the public key which is our bad basis. Then, to encrypt we add a small error to our message to get the ciphertext c.

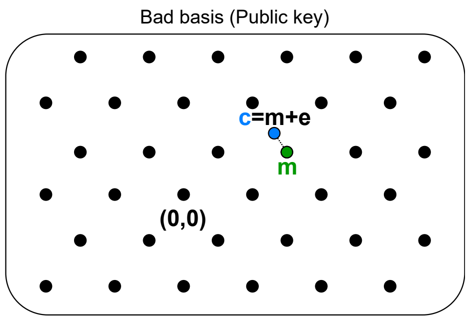

Figure 2.7: Encrypting a message

For decryption, we utilize Babai's nearest plane algorithm. As illustrated in Figure 2.8, this algorithm divides the space into smaller segments. When an element falls within a segment, it is rounded to the lattice point at the center of the corresponding box. To understand why decryption with the public key fails, refer to the bad basis in Figure 2.8. Babai's nearest plane algorithm rounds the ciphertext c not to the correct message m, but to a different lattice point because the red lattice point is at the center of the box where the ciphertext falls. However, with access to the secret key, i.e. the good basis, the ciphertext c is correctly rounded to the message m, enabling its recovery.
Note, that recovering the message with the bad basis seems easy in $\mathbb{Z}^2$, however, this becomes a hard problem in high dimensions.

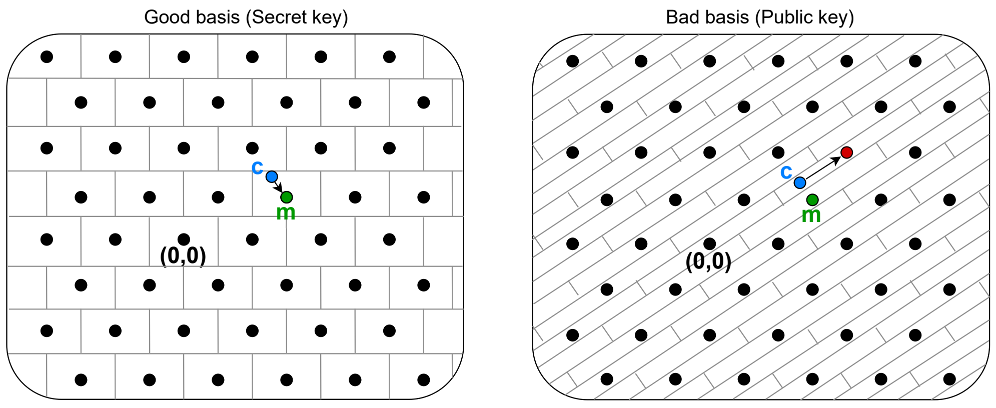

Figure 2.8: Decrypting a message

One problem that arises from this construction is the need to sample nearby lattice vectors for any generated lattice without leaking information about the secret key [31]. To address this, Discrete Gaussian Sampling (DGS) is employed [40]. However, as implementing DGS is still quite inefficient [34] one solution involves using a simple lattice, such as $\mathbb{Z}^n$ with favorable properties to avoid the aforementioned issue. However, in this framework, this approach is nonsensical because fixing a lattice would mean the good basis is no longer secret. To pursue the idea of having a fixed lattice with favorable properties, a new perspective is required, as presented in the Lattice Isomorphism Framework [31].
This new perspective uses a rotation of the lattice as the secret rather than a good basis. More precisely, instead of providing a public key, which is a bad basis for the known lattice, we offer a public key for the rotated version of the lattice, with the rotation being the secret.

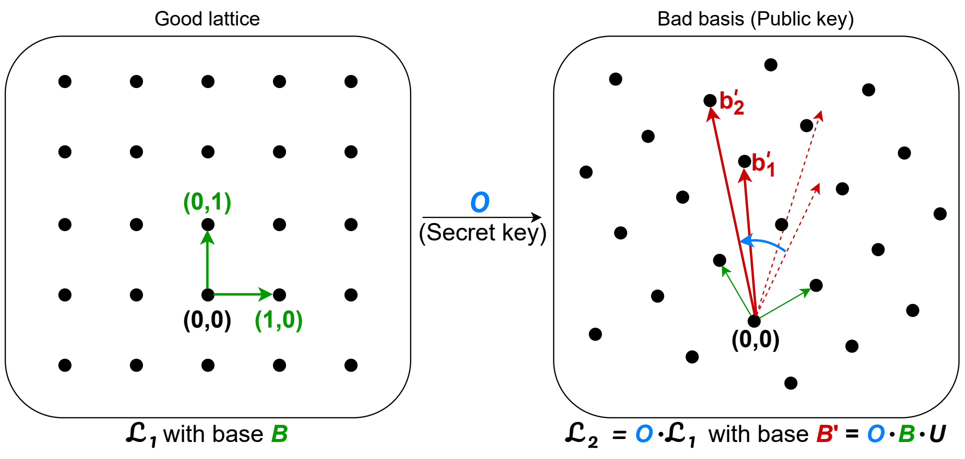
Good lattice
Bad basis (Public key)
(0,1)
(0,0)
(1,0)
O
(Secret key)
b'2
b'1
(0,0)
$\mathcal{L}_1$ with base B
$\mathcal{L}_2 = O \cdot \mathcal{L}_1$ with base $B' = O \cdot B \cdot U$

Figure 2.9: New perspective

Figure 2.9 demonstrates this new perspective. On the left side, a good basis for a lattice with favorable properties is shown. This good basis is no longer a secret. On the right side, a bad basis is shown for the rotated lattice $O \cdot \mathcal{L}_1$ with the rotation being secret. If there exists such an orthonormal transformation $O \in O_n(\mathbb{R})$ such that $\mathcal{L}_2 = O \cdot \mathcal{L}_1$, then the lattices $\mathcal{L}_1$ and $\mathcal{L}_2$ are considered isomorphic [31]. Recovering this transformation is known as the LIP.

2.3.3 Lattice Isomorphism Problem

The Lattice Isomorphism Problem (LIP) is to decide if there exists an orthogonal linear transformation mapping one lattice $\mathcal{L}_1$ to another lattice $\mathcal{L}_2$ [46].
Researchers previously studied the problem and proposed algorithms capable of solving it in low dimensions for specific lattices of interest [67, 66]. Later, the asymptotic complexity of the LIP was considered [75], alongside its relevance to cryptographic applications involving lattices [78].

The LIP is known to reside in $\mathcal{NP}$, however, it is unlikely to be $\mathcal{NP}$-hard [46]. Despite that, it is believed that LIP is a hard problem - something, that cannot be said for some other lattice problems [46].

Definition 5 (LIP, according to [31, Definition 2.1])

Given two isomorphic lattices $\mathcal{L}_1, \mathcal{L}_2 \subseteq \mathbb{R}^n$, find an orthonormal transformation $O \in O_n(\mathbb{R})$ such that $\mathcal{L}_2 = O \cdot \mathcal{L}_1$.

2.4 HAWK

HAWK is a post-quantum signature scheme that was submitted to NISTs *Call for additional Digital Signature Schemes* in June 2023. It is intended to be secure against classical and quantum computers [34].
HAWK operates using lattice-based cryptography, specifically leveraging the LIP as its foundation. The scheme builds upon module lattices, borrowing algorithms and concepts from NTRUSIGN [48] and FALCON [70], which enhances its simplicity and competitiveness. This approach allows HAWK to avoid floating-point arithmetic and rejection sampling [34].

2.4.1 Advantages of HAWK

Speed

The processes for generating and verifying signatures are fast across all devices, including low-end ones. In numbers, this means that HAWK-512 signature generation and verification take $<$0.1ms on an average desktop PC [34].
Compared to the PQC signature scheme FALCON-512, HAWK-512 on AVX2 Central Processing Units (CPUs) performs roughly twice as fast for key generation and verification, and four times faster for signing [34].

Compactness

The key and signature sizes are relatively small. A HAWK-512 public key is 1024 bytes, a private key is 184 bytes, and a signature is 555 bytes.
Compared to FALCON, signatures from HAWK are roughly 15 % more compact [34].

Suitability

HAWK does not employ floating-point arithmetic, which contributes to its suitability for various hardware platforms. Furthermore, HAWK-512 operates with just 14 kiB of Random-Access Memory (RAM) [34].

2.4.2 Background of HAWK

Currently, FALCON stands out as the most efficient lattice-based signature scheme and one of the top performers among post-quantum options [34]. Like its predecessor NTRUSIGN, FALCON employs a hash-then-sign design but addresses signature transcript leakage [64] through DGS [40].
There has been significant progress in optimizing DGS [32, 37, 30], however, securely and efficiently implementing DGS remains challenging, especially on resource-constrained devices and in the presence of side-channel attack vulnerabilities [34].
To avoid the costly DGS described above, HAWK is basing its security on the problem of finding isometries between lattices, also known as the LIP. This allows sampling from the $\mathbb{Z}^n$ lattice which is significantly simpler than DGS [34, 31].

2.4.3 Introduction to HAWK

For efficiency reasons HAWK is modified to number rings, specifically utilizing the number ring $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ with $n \geq 2$, which represents the ring of integers for a power of two cyclotomic field and can be naturally viewed as an orthogonal lattice [34]. As illustrated in Figure 2.9, the new perspective benefits from lattices with favorable properties, such as simple module lattices. An appealing option would be to examine the most structured case, specifically modules of rank one (ideal lattices) over number fields. However, this constrained version of the LIP is known to be solvable in classical polynomial time [39, 60].
Thus, HAWK utilizes the most basic rank 2 module lattice, $\mathcal{R}^2 \cong \mathbb{Z}^{2n}$ [31], which has already been the focus of cryptanalytic examination [78].

To generate a basis for $\mathcal{R}^2$ for key generation, HAWK follows a distribution-based approach similar to that used in NTRUSIGN's key generation, but with the modulus q set to 1.
The secret key is a basis $B \in SL_2(\mathcal{R})$ that generates $\mathcal{R}^2$ and represents a basis transformation of the trivial basis $I_2(\mathbb{K})$ of $\mathcal{R}^2$.
The public key is represented by the Hermitian form $Q = B^* \cdot B$.
To generate a signature for a message m we first hash m and a salt r to a point $h = (h_0, h_1)^T \in \{0,1\}^{2n}$. Then, we get our target $\frac{1}{2}B \cdot h$ by applying the transformation B to $\frac{1}{2}h$. Next, we sample a short vector x from the coset $\mathcal{R}^2 + \frac{1}{2}B \cdot h$ using discrete Gaussian samples over $\mathbb{Z}$ and $\mathbb{Z} + \frac{1}{2}$. We then compute the signature s as $\frac{1}{2}h \pm B^{-1}x$ by applying the inverse transformation B^{-1} to x, resulting in $s \in \mathcal{R}^2$. A visual representation of this process for n = 1 is given in Figure 2.10, following the illustration in [34, Figure 1].
To verify the signature we check if the distance $\|\frac{1}{2}h - s\|_Q$ is within bounds by utilizing the public key Q.

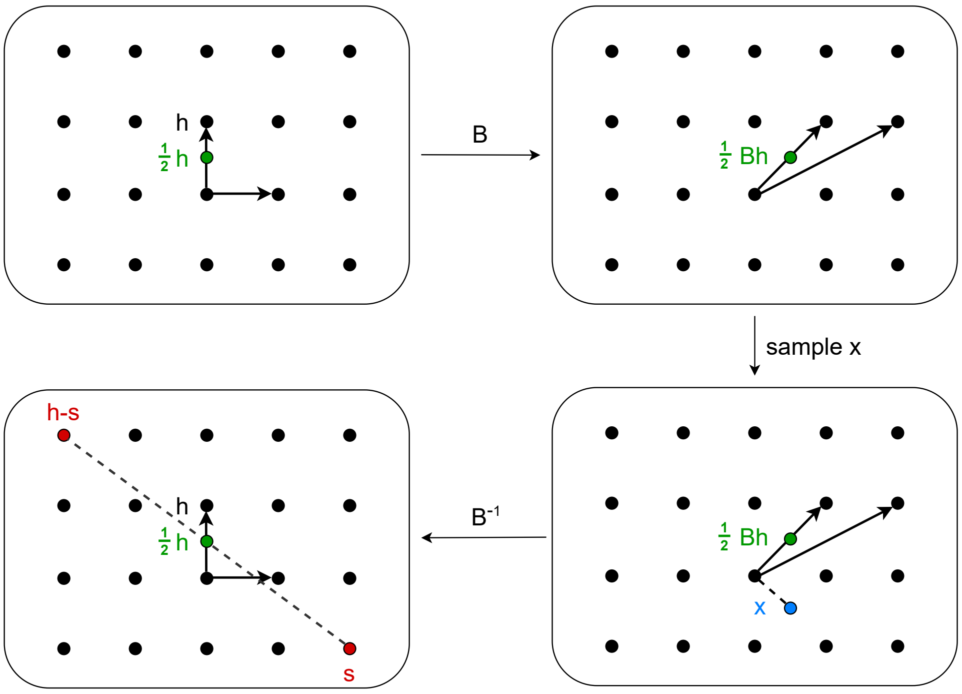

Figure 2.10: Signature creation

2.4.4 Key Pairs

HAWK is defined for degrees n that are powers of two (256, 512, or 1 024). Most computations are done on polynomials with integer coefficients modulo $X^n + 1$. A HAWK private key is a random basis for the lattice $\mathbb{Z}^{2n}$, consisting of four polynomials f, g, F, and G. Here, f and g have small coefficients and satisfy the NTRU equation:

$$fG - gF = 1 \mod (X^n + 1).$$

The lattice secret basis B and its inverse B^{-1} are

$$B = \begin{pmatrix} f & F \\ g & G \end{pmatrix}, \quad B^{-1} = \begin{pmatrix} G & -F \\ -g & f \end{pmatrix}.$$

The public key is $Q = B^{*} \cdot B$ with

$$Q = \begin{pmatrix} q_{00} & q_{01} \\ q_{10} & q_{11} \end{pmatrix} = \begin{pmatrix} f^{*}f + g^{*}g & f^{*}F + g^{*}G \\ F^{*}f + G^{*}g & F^{*}F + G^{*}G \end{pmatrix}.$$

The matrix Q is self-adjoint, meaning: $q_{00} = q_{00}^{\star}$, $q_{11} = q_{11}^{\star}$ and $q_{10} = q_{01}^{\star}$. Additionally, a consequence of the NTRU equation is that $q_{00}q_{11} - q_{01}q_{10} = 1$ holds. Thus, we can always reconstruct Q using only q_{00} and q_{01}.

2.4.5 Algorithmic Overview of HAWK

Here, we present a general description of the HAWK algorithms, following [20, Chapter 2.1]. For high-level overviews on key generation, signature generation, and signature verification, see Algorithm 1, Algorithm 2, and Algorithm 3. Note, that n is a power of two, and H is realized as SHAKE256.

Algorithm 1 High level HawkKeyGen, following [20, Algorithm Sketch 1]

```
Ensure: $\mathbf{B} \in \mathrm{GL}_2(\mathcal{R}_n)$ and $\mathbf{Q} = \mathbf{B}^* \cdot \mathbf{B}$
 1: Sample coefficients of $f, g \in \mathcal{R}_n$ i.i.d. from $\mathrm{Bin}(\eta)$
 2: if $f$-g-conditions$(f, g)$ is false then
 3:     restart
 4: end if
 5: $r \leftarrow \mathrm{NTRUSolve}(f, g)$
 6: if $r$ is $\perp$ then
 7:     restart
 8: end if
 9: $(F, G) \leftarrow r$
10: $\mathbf{B} \leftarrow \begin{pmatrix} f & F \\ g & G \end{pmatrix}, \quad \mathbf{Q} \leftarrow \mathbf{B}^* \cdot \mathbf{B} = \begin{pmatrix} q_{00} & q_{01} \\ q_{10} & q_{11} \end{pmatrix}$
11: if KGen-encoding$(\mathbf{Q}, \mathbf{B})$ is false then
12:     restart
13: end if
14: $h_{\mathrm{pub}} \leftarrow H(\mathbf{Q})$
15: return $(\mathrm{pk}, \mathrm{sk}) \leftarrow (\mathbf{Q}, (\mathbf{B}, h_{\mathrm{pub}}))$
```

In HAWK, key generation involves sampling a secret unimodular matrix $B \in GL_2(\mathcal{R}_n)$ from a distribution based on a centered binomial distribution and the NTRUSolve algorithm. Initially, two polynomials f and g in $\mathcal{R}_n$ are sampled with coefficients from $\mathrm{Bin}(\eta)$, where η is linked to the security parameter n. These polynomials must meet certain conditions (f-g-conditions) to enable efficient procedures within NTRUSolve and align with practical cryptanalysis results. NTRUSolve then produces F and $G \in \mathcal{R}_n$ such that $fG - gF = 1_{\mathcal{R}_n}$, with F and G having relatively small entries. If NTRUSolve fails to find such F and G, the process restarts. The public key Q is the Hermitian matrix $B^\star \cdot B$. The KGen-encoding conditions ensure proper key encoding, and h_{pub}, a hash of the public key, is included in the secret key. The recovery of the secret key from the public key involves solving the LIP. Essentially, knowing B helps find short vectors in various cosets of $\mathbb{Z}^{2n}$, while Q enables the calculation of element lengths.

Algorithm 2 High level HawkSign, following [20, Algorithm Sketch 2]

Require: A message m and secret key $\mathrm{sk} = (\mathbf{B}, h_{\mathrm{pub}})$
Ensure: A signature sig formed of a uniform salt $\mathrm{salt} \in \{0,1\}^{\mathrm{saltlen_{bits}}}$ and $s_1 \in R_n$

```
1: $M \leftarrow H(m \parallel h_{\mathrm{pub}})$
2: $\mathrm{salt} \leftarrow \mathrm{Rnd}(\mathrm{saltlen_{bits}})$
3: $h \leftarrow H(M \parallel \mathrm{salt})$
4: $t \leftarrow \mathbf{B} \cdot h \mod 2$
5: $x \leftarrow D_{2\mathbb{Z}^{2n}+t, 2\sigma_{\mathrm{sign}}}$
6: **if** $\|x\|^2 > 4 \cdot \sigma_{\mathrm{verify}}^2 \cdot 2n$ **then**
7:     restart
8: **end if**
9: $w \leftarrow \mathbf{B}^{-1}x$
10: **if** sym-break($w$) is false **then**
11:     $w = -w$
12: **end if**
13: $s \leftarrow \frac{1}{2}(h - w)$
14: $s_1 \leftarrow \mathrm{Compress}(s)$
15: **if** sig-encoding(salt, $s_1$) is false **then**
16:     restart
17: **end if**
18: **return** sig $\leftarrow (\mathrm{salt}, s_1)$
```

In Hawk, signing involves using the message to select a target coset $2\mathcal{R}_n^2 + h$ from $2\mathcal{R}_n^2$, where a random salt ensures this choice is non-deterministic. The signature comprises the salt and a compressed vector s. This vector s is related to a vector w via h in the target coset and is short under $\|\cdot\|_Q$. First, two hashes are computed: the first over the message m and h_{pub}, then a uniform salt is sampled and hashed with the first hash's output. The function $\mathrm{Rnd}(r)$ produces a uniform bitstring of length r, which is based on n. This double-hashing approach facilitates adaptation to the BUFF transform [27]. The result of the second hash, a binary vector of length $2n$, is interpreted as an element in $\mathcal{R}_n^2$. Next, a vector x is sampled from the discrete Gaussian distribution over $2\mathbb{Z}^{2n} + t$, where $t = \mathrm{vec}(Bh)$ and σ_{sign} is the width parameter. We only need t modulo 2. The sampled x is then treated as an element of $\mathcal{R}_n^2$. The vector $w = B^{-1}x \in 2\mathcal{R}_n^2 + h$ falls within the target coset and has $\|w\|_Q = \|x\|$. If $\|x\|$ is too large, the signature verification will fail, requiring a restart. The condition sym-break prevents weak forgery attacks. Given that $h - w \in 2\mathcal{R}_n^2$ and $s \in \mathcal{R}_n^2$, w can be computed from s and h publicly. The signature consists of the salt and a compressed version of s, with no secret information involved in the compression. The sig-encoding step checks whether the signature fits within the provided buffer. If it does, the signature is encoded.

Algorithm 3 High level HawkVerify, following [20, Algorithm Sketch 3]

```
Require: A message $m$, a public key $\mathrm{pk} = \mathbf{Q}$, and a signature $\mathrm{sig} = (\mathrm{salt}, s_1)$
Ensure: A bit determining whether sig is a valid signature on $m$
 1: $h_{\mathrm{pub}} \leftarrow H(\mathbf{Q})$
 2: $M \leftarrow H(m \parallel h_{\mathrm{pub}})$
 3: $h \leftarrow H(M \parallel \mathrm{salt})$
 4: $s \leftarrow \mathrm{Decompress}(s_1, h, \mathbf{Q})$
 5: $w \leftarrow h - 2s$
 6: if $\mathrm{len}_{\mathrm{bits}}(\mathrm{salt}) = \mathrm{saltlen}_{\mathrm{bits}}$ & $s \in \mathcal{R}_n^2$ & sym-break$(w)$ & $\|w\|_{\mathbf{Q}}^2 \leq 4 \cdot \sigma_{\mathrm{verify}}^2 \cdot 2n$
    then
 7:     return 1
 8: else
 9:     return 0
10: end if
```

In Hawk, verification involves recalculating values from Algorithm 2 using the signature sig, message m, and public key pk. This process recomputes h, then derives s via Decompress, and subsequently computes w. Several conditions are checked to ensure correctness. Since Compress and Decompress are public functions, they do not impact security. However, Compress and Decompress must be accurate, as discrepancies such as $\mathrm{Decompress}(\mathrm{Compress}(s), h, Q) \neq s$ can affect correctness. Hawk carefully selects parameters to ensure such errors occur with negligible probability [20].

2.5 Polynomial Arithmetic

Polynomial arithmetic, particularly multiplication, is computationally demanding in cryptographic schemes. While the addition of two polynomials in $\mathcal{R} = \mathbb{Z}[X]/(X^n+1)$ is efficient, taking $\mathcal{O}(n)$ time, naive multiplication requires $\mathcal{O}(n^2)$ integer multiplications, which can be prohibitive as polynomial degrees grow. To ensure that all computations are performed in $\mathcal{O}(n \log n)$ time, as specified by the Hawk scheme's goals [34], techniques like the NTT can be employed. The NTT significantly reduces the complexity of polynomial multiplication by transforming polynomials into a domain where multiplication is performed coefficient-wise. This transformation enhances the efficiency and scalability of cryptographic operations.

2.5.1 Naive Multiplication

To review the base case for polynomial multiplication, we start with naive multiplication, also known as schoolbook convolutions. This algorithm performs $\mathcal{O}(n^2)$ coefficient multiplications to multiply two polynomials $P(x)$, $Q(x) \in \mathbb{Z}[X]$ of degree $n-1$.

Definition 6 (Polynomial Multiplication, following [72, Definition 2.1 (1)])

Let $P(x) = \sum_{i=0}^{n-1} p_i x^i$ and $Q(x) = \sum_{j=0}^{n-1} q_j x^j$. The product $R(x)$ is defined as:

$$R(x) = P(x) \cdot Q(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} p_i q_j \cdot x^{i+j} = \sum_{k=0}^{2(n-1)} r_k x^k.$$

Definition 7 (Linear Convolution, following [72, Definition 2.1 (2)])

Polynomial multiplication is equivalent to a discrete linear convolution of the coefficient vectors p and q:

$$r[k] = (p * q)[k] = \sum_{i=0}^{k} p[i]q[k-i].$$

Definition 8 (Cyclic Convolution, following [72, Definition 2.2])

A cyclic or positive-wrapped convolution $R(x)$ is denoted as:

$$R(x) = \sum_{k=0}^{n-1} r_k x^k$$

where $r_k = \sum_{i=0}^{k} g_i h_{k-i} + \sum_{i=k+1}^{n-1} g_i h_{k+n-i} \bmod q$.

2.5.2 Multiplication Leveraging the NTT

Building on the basic naive multiplication, we now turn to more advanced methods. The NTT is a finite field variant of the Discrete Fourier Transform (DFT) [68] that is frequently employed in implementations of lattice-based cryptography [41, 61, 29]. It works by transforming polynomials into a domain where multiplication is simplified to a coefficient-wise operation, thereby reducing computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ in the case of the Fast-NTT.
To clarify, the direct computation of the NTT has a quadratic complexity of $\mathcal{O}(n^2)$. However, calculating the NTT similar to the FFT, known as Fast-NTT, achieves a more efficient quasi-linear complexity of $\mathcal{O}(n \log n)$ [72].
Besides this chapter, mentions of NTT will always refer to the more efficient Fast-NTT. Although the NTT of a polynomial lacks physical interpretation, unlike the DFT, it retains one of the DFT's most crucial properties: the convolution theorem. This property is essential for polynomial multiplication.

To understand the definition of the NTT we first need to understand the concept of primitive n-th roots of unity in $\mathbb{Z}_q$. An element $e \in \mathbb{Z}_q$ is called a n-th root of unity (where $n \in \mathbb{N}$) if $e^n \equiv 1 \pmod{q}$. It is considered a primitive n-th root of unity if there is no integer $k < n$ such that $e^k \equiv 1 \pmod{q}$. A primitive n-th root of unity exists if and only if n divides $q-1$, or more generally, if n divides the order of the multiplicative group $\mathbb{Z}_q^*$. We denote a primitive n-th root of unity as ω_n.

Definition 9 (Number Theoretic Transform, following [72, Definition 3.2])

Let ω_n be a primitive n-th root of unity, $a = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$, and q be a prime. The NTT of a polynomial coefficient vector a is denoted as $\hat{a} = \mathrm{NTT}(a)$, where:

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega_n^{ij} a_i \mod q,\ j \in [0, n-1].$$

Definition 10 (Inverse NTT, following [72, Definition 3.3])

The INTT of an NTT vector $\hat{a}$ is denoted as $a = \mathrm{INTT}(\hat{a})$, where:

$$\hat{a}_i = n^{-1} \sum_{j=0}^{n-1} \omega_n^{-ij} \hat{a}_j \mod q,\ j \in [0, n-1].$$

Note that the INTT has a formula very similar to that of the NTT. The only differences are that ω is replaced by its inverse in $\mathbb{Z}_q$ and a scaling factor of n^{-1} is applied. It is always true that $a = \mathrm{INTT}(\mathrm{NTT}(a))$.

Proposition 1 (following [72, Proposition 3.1])

Since the NTT is a polynomial-ring analog of the DFT, the convolution theorem from the DFT can be applied to compute the cyclic convolution [4], thereby facilitating polynomial multiplication.
Let a and b be the vectors of polynomial coefficients for the multiplicands. The positive-wrapped convolution of a and b, denoted by c, can be computed as:

$$c = INTT(NTT(a) \circ NTT(b))$$

with $\circ$ being the element-wise vector multiplication in $\mathbb{Z}_q$.

2.5.3 Fast NTT

In the preceding section, the discussed NTT and INTT transformations exhibit a complexity of $\mathcal{O}(n^2)$, which is consistent with the traditional naive approach. However, since the NTT is essentially the DFT within a different ring, the optimization techniques applied to the DFT are also applicable to the NTT. A prominent DFT optimization method is the FFT, introduced independently by Cooley-Tukey (CT) [26] and Gentleman-Sande (GS) [38]. Both methods employ a divide-and-conquer approach, utilizing butterflies to reduce complexity to $\mathcal{O}(n \log n)$ by splitting the computation of an NTT of size $2n$ into two NTTs of size n, a process facilitated by the Chinese Remainder Theorem (CRT). It is standard practice to use the Cooley-Tukey FFT algorithm for computing the NTT, while the Gentleman-Sande FFT algorithm is typically employed for the INTT [53], however, both are capable of calculating the NTT and INTT.
These FFT algorithms are distinguished by their butterfly operations. Figure 2.11 illustrates the butterfly structures used in the Cooley-Tukey and Gentleman-Sande FFT algorithms, respectively.

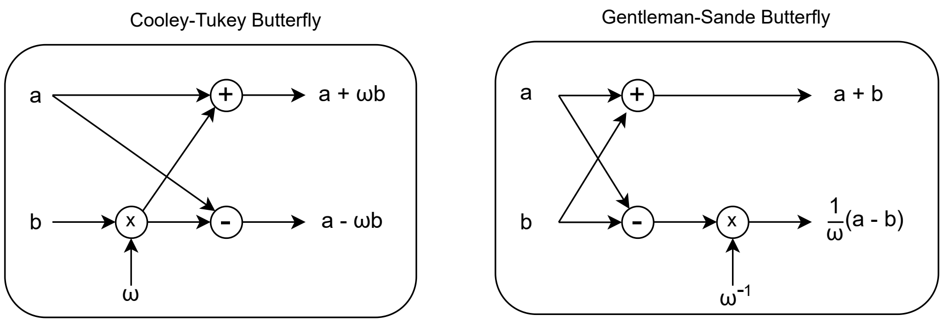

Figure 2.11: Butterfly constructions

As described in [53, Chapter 2.2.5] the CT algorithm straightforwardly decomposes $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ into $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2k-1} + c)$ (cf. Algorithm 4), where c is a constant known as the twiddle factor. In contrast, the Gentleman-Sande butterfly operation calculates the CRT for elements in $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$ to obtain an element in $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ (cf. Algorithm 5), using the twiddle factor c^{-1}.

In the following, Algorithm 4 and Algorithm 5 demonstrate the use of these butterfly constructions to acquire the NTT and INTT in-place. FFT algorithms have a characteristic where their outputs appear in a sequence known as bit-reversed order. Bit-reversing an array of length 2^k involves interpreting the index of each element as a binary string of length k and then reversing this string. For an index i, where i_j denotes the j-th bit, this process can be represented as:

$$i = \sum_{j=0}^{k-1} i_j \cdot 2^j \quad \text{and} \quad \mathrm{brv}_k(i) = \sum_{j=0}^{k-1} i_{k-j} \cdot 2^j.$$

The reversed index is used as the new index for each element. This operation is invertible, as applying the transformation again restores the original order.

We give a small example for the bit-reversing of an array of length four: $[a_0, a_1, a_2, a_3]$

1. Express the indices in binary form: $[a_{00}, a_{01}, a_{10}, a_{11}]$
2. Reverse the binary digits: $[a_{00}, a_{10}, a_{01}, a_{11}]$
3. Array in bit-reversed order: $[a_0, a_2, a_1, a_3]$

As a side note, due to the rather high computational cost of bit-reversing in software [53], it is advisable to maintain values in bit-reversed order within the NTT domain. This practice does not affect the element-wise polynomial operations in that domain, and the INTT will naturally restore the normal order. Moreover, as shown in Algorithm 4 and Algorithm 5, the exponents of the twiddle factors are arranged in bit-reversed order (brv), which makes on-the-fly computation impractical. Consequently, the twiddle values are typically pre-computed and stored in memory.

Algorithm 4 CT FFT implementing NTT, following [53, Algorithm 3]

```
Require: Polynomial $a \in \mathbb{Z}_q/(x^{2^k} \pm 1)$; time domain, normal order
Require: Root of unity: $\omega_{2^k}$ for $\mathbb{Z}_q/(x^{2^k} - 1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k} + 1)$
Ensure: $\hat{a}$; 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:         $\psi \leftarrow \begin{cases} \omega_{2^k}^{\mathrm{brv}_{k-1}(i)} & \text{for } \mathbb{Z}_q/(x^{2^k} - 1) \\ \omega_{2^{k+1}}^{\mathrm{brv}_k(2^{k-1-l}+i)} & \text{for } \mathbb{Z}_q/(x^{2^k} + 1) \end{cases}$
 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
```

Algorithm 5 GS FFT implementing INTT, following [53, Algorithm 4]

```
Require: $\hat{a}$; NTT domain, bit-reversed order
Require: Root of unity: $\omega_{2^k}$ for $\mathbb{Z}_q/(x^{2^k} - 1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k} + 1)$
Ensure: 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:         $\psi \leftarrow \begin{cases} \omega_{2k}^{-\mathrm{brv}_{k-1}(i)} & \text{for } \mathbb{Z}_q/(x^{2^k} - 1) \\ \omega_{2k}^{-\mathrm{brv}_k(2^{k-1-l}+i)} & \text{for } \mathbb{Z}_q/(x^{2^k} + 1) \end{cases}$
 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
```

2.6 Modular Multiplication and Reduction Techniques

While the NTT provides a highly efficient method for polynomial multiplication at a higher level, achieving high-speed performance in the underlying integer operations is equally critical. As seen in Algorithm 4 and Algorithm 5, the key integer operations involved are modular multiplication, addition, and subtraction within $\mathbb{Z}_q$. These operations serve as fundamental building blocks, and because they are called repeatedly during the NTT and INTT, their efficiency directly impacts the overall performance, making their optimization essential.

2.6.1 Montgomery Multiplication

Montgomery multiplication is a powerful technique used to perform modular multiplications efficiently, particularly in the context of cryptographic applications. Peter Montgomery introduced it in 1985 [63] and it has since become a fundamental method in cryptographic algorithms.

Montgomery multiplication is designed to optimize the modular reduction process that is often required in cryptographic computations, such as RSA and ECC. In the realm of PQC it is employed, for example, in the reference implementations of CRYSTALS-Kyber [19] and CRYSTALS-Dilithium [33]. The key advantage of Montgomery multiplication is that it avoids the need for expensive division operations by replacing them with simpler arithmetic operations [63].

To utilize Montgomery multiplication, numbers are transformed into Montgomery form, also known as q-residues, In this context, a q-residue is defined as a residue class modulo q.

Definition 11 (q-residue, following [63, Description])

Let a radix R be an integer greater than q and coprime to q, i.e., $\gcd(R, q) = 1$. The q-residue of $x \in \mathbb{Z}_q$ is defined as

$$x' = x \cdot R \bmod q.$$

Suppose we wish to multiply two integers x and y by utilizing Montgomery multiplication. First, they are converted into their respective Montgomery representations, which are the q-residues x' and y' of x and y:

$$x' = x \cdot R \bmod q$$

$$y' = y \cdot R \bmod q.$$

Their product, $x' \cdot y'$, equates to $x \cdot y \cdot R^2$. To reduce this product modulo q, Algorithm 6 is applied, resulting in $\texttt{montyR}(x \cdot y \cdot R^2) = x \cdot y \cdot R \bmod q$, which corresponds to the product of x and y reduced modulo q in the Montgomery domain.

Algorithm 6 Montgomery reduction `montyR`, following [63, Algorithm REDC]

```
Require: Modulus $q$
Require: Radix $R = 2^n > q$
Require: $-q^{-1} \bmod R$
Require: $T$ with $0 \leq T < q \cdot R$
Ensure: $\mathrm{montyR}(T) = T \cdot R^{-1},\ 0 \leq t < 2 \cdot q$
1: $m \leftarrow T \cdot (-q^{-1}) \bmod R$
2: $t \leftarrow \frac{(T+m \cdot q)}{R}$          ▷ Efficiently obtainable by shifting
3: if $t \geq q$ then
4:     return $t - q$
5: else
6:     return $t$
7: end if
```

To revert the result back to the standard domain, Algorithm 6 is used again, resulting in $\texttt{montyR}(x \cdot y \cdot R) = x \cdot y \bmod q$. However, this final step is usually delayed until multiple modular operations within the Montgomery domain are completed, as switching between domains is costly.

Following this, the signed Montgomery reduction (cf. Algorithm 7) can be employed to handle cases where the input and output are signed values. By adopting signed arithmetic, it becomes sufficient to compute only the upper half of $m \cdot q$, which is typically less expensive than calculating the entire product [53]. Additionally, it is necessary to use q^{-1} instead of $-q^{-1}$, and thus, a subtraction is required instead of an addition in the final step.

One application of the Montgomery multiplication in the context of lattice-based cryptography is in the NTT and INTT (cf. Algorithm 4, Algorithm 5), where one of the two factors is a pre-computed constant, also known as twiddle factor. When applying Montgomery multiplication, it is advantageous to convert these constants into the Montgomery domain by storing $T \cdot R \bmod q$ instead of T [8]. This approach ensures that the result of a Montgomery multiplication is immediately in the normal domain, eliminating the need for further transformations.

Algorithm 7 Signed Montgomery reduction, following [73, Algorithm 3]

```
Require: Modulus $q$
Require: Radix $R = 2^n > q$
Require: $q^{-1}$ mod $\pm R$
Require: $T$ with $0 \leq T < q \cdot R$
Ensure: $t \equiv T \cdot R^{-1},\ -q \leq t \leq q$
1: $m \leftarrow T \cdot (q^{-1})$ mod $\pm R$
2: $t \leftarrow \frac{(m \cdot q)}{R}$                ▷ Efficiently obtainable by shifting
3: $t \leftarrow \lfloor \frac{T}{R} \rfloor - t$
4: return $t$
```

Montgomery multiplication does not generally stand out as superior to other methods such as Barrett or Plantard multiplication. While looking at 32-bit arithmetic these three techniques require the same number of arithmetic operations, however, their memory demands differ. Specifically, 32-bit Barrett multiplication relies on two 32-bit constants, and 32-bit Plantard multiplication requires a 64-bit constant, whereas 32-bit Montgomery multiplication only utilizes a single 32-bit constant. This makes Montgomery multiplication more memory-efficient and, therefore, the preferred choice for our work involving 32-bit arithmetic. Yet, it remains uncertain whether 32-bit Montgomery outperforms 16-bit Plantard [49, Algorithms 14, 15] in our scenario. We expect similar performance, however, we have not tested this in detail and thus leave this as future work. Another potential candidate, 16-bit Barrett, requires one additional instruction (three in total) compared to 16-bit Plantard (two instructions), which disqualifies it, as this additional instruction prevents it from being competitive in terms of performance [52]. These considerations justify our choice of 32-bit Montgomery.

2.6.2 Lazy Reduction

Lazy reduction is a technique used in modular arithmetic to defer the reduction of values modulo a given number, typically until it becomes necessary. Rather than reducing intermediate results immediately after each operation, values are allowed to grow beyond the modulus for a while, provided they remain within a manageable range. This approach can improve performance by reducing the number of costly reduction operations during computations. In the context of PQC, especially in algorithms like the NTT, Lazy reduction allows multiple operations to be performed on unreduced numbers, with a final reduction step applied only when required.

2.7 Jasmin

Jasmin [12, 14] is a programming framework specifically developed for creating cryptographic software that balances high performance with strong security guarantees. It centers around the Jasmin programming language, designed to improve portability and streamline the verification process, and its formally verified compiler. The Jasmin language, an assembly-like language with a structured control flow, gives developers explicit access to assembly instructions for the target architecture while supporting zero-cost abstractions that enhance readability without sacrificing efficiency. A key example of a zero-cost abstraction in Jasmin is the use of variables [65]. In a Jasmin program, a variable is represented as an n-bit word with either the *reg* or *stack* storage class, which determines where it is stored. Usual control flow constructs in Jasmin include if conditions, while loops, function calls, and unrolled loops (using for). In general, each line of Jasmin code corresponds to a single line of assembly code. Additionally, the compiler does not perform code scheduling or register spilling. As a result, the developer must manually handle register spilling when register pressure requires it. The compiler, which has been formally verified using the Coq proof assistant [12], produces highly efficient assembly code that rivals the performance of fast, hand-crafted implementations. Notable implementations include scalar multiplication on the Curve25519 elliptic curve [12], the ChaCha20 stream cipher [14], and the SHA-3 hash function [13]. Thus, Jasmin has been successfully used in the development of efficient, formally verified cryptographic implementations, also including those in the Libjade library for post-quantum cryptographic algorithms [35].

The framework provides rigorous semantics for Jasmin programs, written in Coq, and compiles them through a series of verified passes, including source-to-source transformations. Examples include the *lowering pass* that replaces high-level code with assembly instructions (ARMv7-M in our case), such as `a = b^c` with `EOR a,b,c`. Another example is the *inlining pass* that eliminates inline functions by replacing their calls with the corresponding function bodies at the call sites. Moreover, the *unrolling pass* eliminates for loops by unrolling them. All passes besides a few exceptions [65] are certified in Coq, ensuring the correctness of the compiler and preserving the functional correctness and security properties from source code to generated assembly.

Additionally, Jasmin integrates with the EasyCrypt proof assistant [18], facilitating proofs of functional correctness and reductionist security. The framework also includes type systems to ensure constant-time security, which is essential for protection against timing attacks and speculative vulnerabilities such as Spectre v1 [57].

2.7.1 Fundamentals of Jasmin Syntax and Usage

In this section, we will present an overview of the fundamental building blocks utilized in our optimized implementation. This is not an exhaustive list; it is meant to give the reader a general intuition about working with Jasmin.

- **Storage Types**
  In Jasmin, there exists three storage types [12]:
  - stack: store on the stack
  - register: store in a register
  - inline: resolve at compile time

  For instance, a `reg u32` represents a 32-bit value stored in a register:
  ```
  reg u32 a = 42;
  ```

- **Arrays**
  We only utilize arrays that are stored on the stack. However, Jasmin also provides options to allocate arrays in registers, or global memory (code segment).

  First, we specify the amount of space needed on the stack for our array, then we declare a pointer using a `reg ptr`. Finally, we assign the address of our array to the register named `pointer_a` in this example:
  ```
  stack u32[8] a;
  reg ptr u32[8] pointer_a;
  pointer_a = a;
  ```

  When calculating the address of an array element, the index is implicitly scaled by the declared size of the elements in the array. Thus, we can for example write to the array as follows:
  ```
  pointer_a[2] = 1138;
  ```
  This means we access the 32 bits in memory located at an offset of 2×4 bytes (since each element is 4 bytes) from the start of the array `a`. As a result, the third 32-bit element of array `a` is assigned the value 1138.

  However, Jasmin allows disabling implicit scaling using the following syntax, which accesses memory directly at the specified offset in bytes:
  ```
  pointer_a.[8] = 1138;
  ```
  This way, we write value 1138 at an offset of 8 Bytes, representing again writing to the third 32-bit element of the array.

There is also the option of type punning. Arrays in memory are essentially a continuous sequence of bytes with a specific total size. Throughout the lifetime of an array, this sequence of bytes can be interpreted in different ways. For example, on access, we can specify that we want to treat the array as a sequence of 16-bit values:
`reg u32 value = (32u)pointer_a[u16 0];`
The type specified after the left bracket `u16` indicates the type of the array elements for this particular access. This is also the type of the value being read or written during that access. Since we want to store a 16-bit value in a 32-bit register, we also need to cast the value to 32 bits using `(32u)`.

Arrays in Jasmin do not support pointer arithmetic, meaning we cannot, for instance, increment the value of a `reg ptr`.

- **Other Memory Access**
  In Jasmin, memory can also be accessed in a different way. For instance, if an export function receives a pointer from C as a `reg u32` argument, such as `reg u32 a`, this value can be treated as the address of an array. Memory can then be accessed like this:
  `reg u32 value = [a + 4];`
  This instruction accesses the 32-bit value located 4 bytes offset from the beginning of the array pointed to by `a`. In other words, it retrieves the second 32-bit value stored in the memory addressed by `a`.

  This approach to memory access allows for pointer arithmetic, as the address is stored in a `reg u32`.

- **Control Flow Constructs**
  In Jasmin, the following control structures are supported:
  - `if`
  - `while`
  - `for`
  with `for` loops being unrolled at compile time. Additionally, function calls are available as another way of controlling the flow of a program.

2.8 Arm Cortex-M4

The Arm Cortex-M4 is a widely used 32-bit microcontroller, particularly in embedded systems, due to its efficiency, performance, and rich feature set. It serves as the target platform for implementation throughout this thesis. The main reason for this choice stems from NIST's recommendation for this architecture as the primary microcontroller optimization target.

2.8.1 Architecture Overview

The Arm Cortex-M4 is based on the ARMv7E-M architecture which extends the ARMv7-M architecture [15] by adding support for DSP instructions, and single-cycle multiplication while retaining the core features of
ARMv7-M. It is built on a Reduced Instruction Set Computer (RISC) architecture, which emphasizes simplicity and efficiency by using a reduced set of simple instructions that can be executed within a single clock cycle, resulting in faster and more power-efficient processing. Features of the ARMv7-M architecture include:

Pipeline

Typically, for each execution of an instruction, a microcontroller must load the instruction (Fetch), interpret it, and load the required data (Decode), and then execute the instruction and store the results (Execute). Processing instructions one after another can lead to inefficiencies, as the processor remains idle during each stage of instruction handling. A pipeline in a processor addresses this by creating a series of stages through which instructions pass sequentially, allowing multiple instructions to be processed simultaneously at different stages and significantly improving overall execution efficiency. The Cortex-M4 using the ARMv7-M Instruction Set Architecture (ISA) features a three-stage pipeline: Fetch, Decode, and Execute. For example, while one instruction is being decoded, another can be fetched, and a third can be executed, reducing idle time and increasing the overall instruction throughput compared to processing each instruction sequentially. In this setup, each instruction's result is immediately available after execution, allowing the next instruction to be used without delay. This design simplifies code optimization by eliminating additional latency caused by instruction dependencies. The concurrent operation of each stage improves performance while maintaining the power efficiency required in embedded systems.

Registers

The Cortex-M4 hosts 16 32-bit registers `r0-r15`. 13 of these are general-purpose registers (`r0-r12`) that the developer can use while the remaining three are the stack pointer (`r13`), the link register (`r14`), and the program counter (`r15`) [16]. Note that the link register holds the return address for subroutine calls and can be pushed to the stack, allowing it to be used for computations, too. Arm's Application Binary Interface (ABI) defines registers `r0-r3` for passing arguments to subroutines, with any additional arguments stored on the stack. The return value is stored in `r0`. As a result, `r0-r3` are caller-saved registers, while `r4-r11` are callee-saved, meaning the callee must save and restore their values if they are modified. Register `r12` does not require preservation as it acts as a scratch register.

Barrel Shifter

A key feature of the Arm architecture is its barrel shifter, also known as the flexible second operand, which allows shifting or rotating one of the operands in most data-processing instructions without additional cost. This means that an operand can be shifted or rotated by an arbitrary number of bits with no added overhead, and in some cases, even help set the carry flag.
For instance, if we want to compute `r1 = r1 - (r3 / 8)` we can do so in one instruction by utilizing the barrel shifter: `SUB r1, r1, r3, LSR #3`. The `LSR #3` operation performs a logical shift right by 3 bits, effectively dividing the value in `r3` by 8. The `SUB` instruction then subtracts this result from `r1` and stores the outcome in `r1`.

Load Instructions

Load instructions are used to transfer data from memory into a register. The Cortex-M4 supports several load operations, including:

`LDR` (Load Register): Loads a word from memory into a register. For example, `LDR r0, [r1]` loads a 32-bit value from the address contained in `r1` into `r0`.

`LDRH` (Load Register Halfword): Loads a half-word (16 bits) from memory into a register, zero-extending the value to 32 bits. For example, `LDRH r0, [r1]` loads a 16-bit value from the address in `r1` into `r0`.

`LDRB` (Load Register Byte): Loads a byte from memory into a register, zero-extending the value to 32 bits. For instance, `LDRB r0, [r1]` loads an 8-bit value from the address in `r1` into `r0`, filling the upper bits with zeros.

Store Instructions

Store instructions are used to write data from a register to memory. The Cortex-M4 provides several store operations, including:

`STR` (Store Register): Stores a word from a register into memory. For instance, `STR r0, [r1]` writes the 32-bit value in `r0` to the address specified in `r1`.

`STRH` (Store Register Halfword): Stores a half-word (16 bits) from a register into memory. The lower 16 bits of the register are written to memory, with the upper bits being discarded. For instance, `STRH r0, [r1]` stores the lower 16 bits of `r0` into the memory address specified by `r1`.

`STRB` (Store Register Byte): Stores a byte from a register into memory. The lower 8 bits of the register are written to memory, with the remaining bits being ignored. For example, `STRB r0, [r1]` stores the lower 8 bits of `r0` into the memory location specified by `r1`.

Addressing Modes

The Cortex-M4 supports several addressing modes for load and store instructions, such as:

Immediate offset: An offset is added to a base register to determine the memory address. For example, `LDR r0, [r1, #4]` loads a value from the address `r1 + 4` into `r0`.

Register offset: The address is calculated by adding a value from another register to a base register. For instance, `LDR r0, [r1, r2]` loads a value from the address `r1 + r2` into `r0`.

Register offset with shift: The address is calculated by adding a value from another register that is shifted by a specified number to a base register. For instance, `LDR r0, [r1, r2, LSL#2]` loads a value from the address `r1 + 4` · `r2` into `r0`.

Pre-indexed: This mode modifies the base register before the memory access. For example, `LDR r0, [r1, #4]!` loads from `r1 + 4` and then updates `r1` to `r1 + 4`.

Post-indexed: This mode modifies the base register after the memory access. For example, `LDR r0, [r1], #4` loads from `r1` and then updates `r1` to `r1 + 4`.

Immediate Values

Immediate values are constants embedded directly within instructions, providing a way to encode fixed values without requiring additional memory accesses. However, there are only a few bits for an immediate value in data processing instructions. In most instructions, we are limited to an 8-bit immediate. To get larger values we can use the barrel shifter.
`MOV` instructions that load a register with an immediate value are limited to 16-bit immediates, for example, `MOV r0, #327`. If we want to load a register with a bigger value, the assembler uses the barrel shifter, if the number can be represented by shifting a smaller number. For example, we specify `MOV r0, #0x10000` and the assembler compiles this to `MOV r0, #0x40, LSL #10`. Loading arbitrary 32-bit constants is achieved by writing them to memory after the code. The assembler does this automatically if we use the syntax `LDR Rn, =#constant`. There is one exception for numbers that surpass the maximum allowed size for immediates, which form specific patterns, including `0x00XY00XY`, `0xXY00XY00`, and `0xXYXYXYXY`.

The ARMv7E-M architecture extends the ARMv7-M architecture by adding:

DSP Instructions

The Arm Cortex-M4 processor includes DSP instructions designed to accelerate signal processing tasks such as audio and communications. These instructions leverage the processor's Single Instruction Multiple Data (SIMD) capabilities to enhance performance. Thus, the processor performs computations on multiple data units in parallel, such as simultaneously processing two halfwords packed into a single 32-bit register.
Take, for example, `SMUAD r0, r1, r2`: The `SMUAD` instruction achieves dual 16-bit signed multiply with addition of products, and optional exchange of operand halves. More specifically, `SMUAD` multiplies the bottom halfword of `r1` with the bottom halfword of `r2`, and the top halfword of `r1` with the top halfword of `r2`. It then adds the products and stores the sum to `r0`.

Single-Cycle Multiplication

On the Cortex-M4 processor, multiplication instructions are optimized to execute within a single cycle, the same duration as simple data-processing instructions such as additions. This efficiency is particularly advantageous for applications involving polynomial multiplications, as it allows for the multiplication of 32-bit integers to be completed in just one cycle. Some instructions that leverage this capability include: `MUL`, `MLA`, `MLS`, `UMULL`, `SMULL`, `UMLAL`, `SMLAL`, `SMUAD`, `SMLAD`, `SMULxy`, `SMULWy`

Floating-point registers

The Arm Cortex-M4 optionally includes a Floating Point Unit (FPU) that hosts 32 32-bit floating-point registers. An interesting option of utilizing the FPU is to use it to spill intermediate results to. This can be beneficial because moving data to and from the FPU is faster than moving data to and from memory.

2.8.2 Development Board Overview

For this thesis, the STM32F4 Discovery board is used as the development platform. This board is equipped with an STM32F407VG microcontroller [77], which is based on the ARM Cortex-M4 architecture. The STM32F4 Discovery board offers a range of features that make it suitable for embedded system development and experimentation.
Key features of this board include a 32-bit ARM Cortex-M4 processor running at a maximum of 168 MHz, 1 MB of Flash memory to store code and data, and 192 KB of SRAM. It also includes a variety of peripherals such as LEDs, push-buttons, and a built-in ST-LINK/V2 debugger/programmer, which simplifies the development and debugging process. The board supports a wide range of applications, from simple control systems to complex digital signal processing tasks.
The board features a 1 024-byte instruction cache and a 128-byte data cache, with optional instruction prefetching. While it operates at up to 168 MHz, the slower flash memory can cause up to seven-cycle delays on cache misses. To mitigate these delays and better reflect practical deployment conditions, we reduce the core's clock speed to 24 MHz. This frequency ensures that flash access does not introduce stalls, offering a more accurate assessment of performance in real-world scenarios.

3 HAWK on Cortex-M4

In this chapter, we first introduce the pqm4 library used to run HAWK on Cortex-M4. Next, we provide an overview of the initial benchmarking and profiling conducted on the reference implementation of HAWK's signing routine, including a detailed analysis of its individual components. The results from the pqm4 reference implementation are presented to highlight areas most in need of optimization, laying the groundwork for the targeted improvements discussed in the next chapter.

3.1 The pqm4 Library

The pqm4 library [54] is a benchmarking and testing framework for post-quantum cryptographic schemes, developed as part of the PQCRYPTO[1] project funded by the European Commission's H2020 program[2]. Tailored to the ARM Cortex-M4 microcontroller family, pqm4 provides implementations of post-quantum KEMs and digital signature schemes. For each implementation, the build system compiles six binaries that can be used to test and benchmark the schemes. After we flash and run a binary on the board we can retrieve the results over a serial Universal Serial Bus (USB) connection using pqm4's Python script `host_unidirectional.py` that reads and outputs the data to the console for analysis.

3.2 Benchmarking

Benchmarking provides an overall measure of a scheme's performance by assessing its execution under typical conditions. In this context, benchmarking involves measuring the cycle counts for HAWK-256's key generation, signing, and verification routines. This provides an overall assessment of the scheme's performance and sets a baseline for more in-depth profiling of specific subfunctions within the signing routine in the next section.

[1] https://pqcrypto.eu.org/
[2] https://cinea.ec.europa.eu/programmes/horizon-europe/h2020-programme_en

3.2.1 Methodology

The pqm4 library provides comprehensive support for benchmarking cryptographic schemes on the Cortex-M4 platform. Among these tests is `speed.c`, a benchmarking script that evaluates cycle counts for key generation, signing, and signature verification. The signing process involves a random 59-byte message[3], which serves as the default size in pqm4, providing a baseline for assessing each scheme's performance. For all benchmarking and profiling purposes, we use modified versions of `speed.c`; this involves flashing the compiled `speed` binary of Hawk-256 onto the board and executing it to gather performance data.

To obtain these cycle measurements, pqm4 relies on its HAL timing functions. Specifically, we use `hal_get_time()` to capture cycle counts, allowing us to determine the cycle consumption of each function by calculating the difference between start and end times. These cycle counts are then sent through the serial interface via the `printcycles()` function where we can collect them by running `host_unidirectional.py`.

[3]Despite contacting the maintainers for an explanation, no compelling reason for choosing a 59-byte message length was offered.

```
t0 = hal_get_time();              // Start cycle count
foo(A, B);                        // Function being profiled
t1 = hal_get_time();              // End cycle count
printcycles("cycles: ", t1-t0); // Print cycle count
```

Listing 3.1: Cycle counting example

To extend these benchmarking capabilities, we adapt the `speed.c` file to compute additional metrics, including the average cycle count and standard deviation over a configurable number of repetitions, offering a more detailed statistical analysis of HAWK-256's performance. Again, we use the `printcycles()` function to send the results over to our host machine.

3.2.2 Benchmarking Results

To begin, we measure the total cycle counts for key generation, signing random 59-byte messages, and verification by performing 10 000 runs of each routine. We report the following data:

Routine	Average	Standard Deviation	Coefficient of Variation (%)
Keygen	14 826 389.78	4 140 745.85	27.93
Sign	1 001 410.10	286 861.33	28.65
Verify	580 671.97	115.59	0.02

Table 3.1: Performance metrics for key generation, signing, and verification routines of HAWK-256.

Since this thesis focuses on optimizing the sign routine, the following sections will address it exclusively.

Sign

It is important to note that the signing process occasionally requires multiple attempts to meet specific algorithmic conditions (cf. Algorithm 2), which impacts the cycle count. In a broader test of 100 signing operations with random test vectors, 87 % of runs were completed in a single attempt, 10 % required two attempts, and 3 % required three attempts. In these cases, the cycle counts increase proportionally, with two-attempt runs averaging approximately 1 528 127 cycles and three-attempt runs around 2 161 722 cycles. However, this increase in cycle count occurs because parts of the signing process are simply repeated; no individual subroutine takes additional cycles. For benchmarking, we thus focus on single-attempt runs, as re-attempts merely repeat the same subroutines.
The mean cycle count across these remaining single-attempt runs is 892 469 cycles, with a standard deviation of 5 cycles and a coefficient of variation of 0.00053 %, indicating extremely low variability.

3.3 Profiling

Profiling, in contrast, breaks down the sign routine into its component functions to analyze each operation in detail. Post-quantum cryptographic schemes like HAWK often rely on resource-intensive operations such as polynomial arithmetic and modular reduction, which can be challenging to execute efficiently on constrained hardware. Profiling is therefore a critical step in the optimization process, as it allows us to pinpoint which functions and operations within the HAWK sign routine consume the most cycles and identify the potential causes of inefficiency.

3.3.1 Methodology

When profiling each major function of HAWK's sign routine, we cannot solely rely on `speed.c`. Since our focus is not on the total cycle count of the entire signing routine this time, we remove the benchmarking aspect from this test. Instead, we use this test as our platform to sign a random message with a newly generated keypair during each iteration. Because we are interested in the profiling of each major component within the sign routine we need to make adjustments to the code of the signature creation itself by inserting the above-mentioned cycle count technique (cf. Listing 3.1) for every major function within the sign routine. Once these adjustments are made, we build the modified codebase and flash the resulting binary onto the microcontroller. This binary includes calls to `hal_get_time()` and `printcycles()`, which capture and output cycle counts for each profiled function. Once the binary is flashed, we collect the cycle count data using an adapted version of `host_unidirectional.py`. Instead of outputting the data to the console it writes the cycles for each profiled function to a file on the host machine. After completing all iterations and populating this file we utilize another newly created Python script to parse the data and calculate the average cycle count, standard deviation, and coefficient of variation for each profiled function.

3.3.2 Profiling Results

The cumulative cycle count of 889 867 cycles is the sum of the cycles of the main functions within HAWK-256's sign routine, as listed below. This total is slightly lower than the baseline of 892 469 cycles because it only includes the primary functions, excluding some minor auxiliary operations.

Function	Description	Average Cycles	Contribution	References from Algorithm 2
1	Re-expand k_{priv}	48 588	5.46 %	n.a.
2	$M \leftarrow H(m\|\|h_{pub})$	10 615	1.19 %	Step 1
3	$salt \leftarrow Rnd(saltlen)$	11 517	1.29 %	Step 2
4	$h \leftarrow H(m\|\|salt)$	11 026	1.24 %	Step 3
5	$t \leftarrow B \cdot h \mod 2$	19 815	2.23 %	Step 4
6	$x \leftarrow D_{t/2}(\sigma_{sign})$	589 435	66.24 %	Step 5
7	$w \leftarrow B^{-1}x$	176 908	19.88 %	Step 9
8	snorm(w)	2 351	0.26 %	Step 9
9	sym-break(w)	3 375	0.38 %	Step 10, 11
10	$s_1 \leftarrow Compress(s)$	6 197	0.70 %	Step 13, 14
11	sig-encoding$(salt, s_1)$	10 040	1.13 %	Step 15

Table 3.2: Profiling results for major routines in HAWK-256's sign routine over 10 000 iterations, average coefficient of variation of 0,012 %.

Function 8 is technically part of Function 7, as it represents the final operation within it. However, for ease of comparison with the optimized version in the following chapter, we have separated the two functions.

1. Largest contributor

As shown in Table 3.2, the largest contributor to the total cycles for the sign routine is Function 6, which generates samples from a discrete Gaussian distribution, consuming roughly two-thirds of the overall cycle count. The high cycle count is primarily because this function calls `shake_extract` 128 times, which, in turn, invokes `keccak_inc_squeeze` in each call. Although the reference implementation of HAWK already utilizes an optimized version of Keccak, it remains expensive in software. Because this step is already optimized in software, we will not go into further detail here. We anticipate that once hardware support for Keccak becomes more widespread, the performance impact of this step will be substantially reduced. This is supported by examples such as the ML-DSA scheme, which demonstrates speedups of 4 to 17 times when using hardware-accelerated versions of Keccak on OpenTitan [3].

2. Largest contributor

The second largest contributor to the cycle count is Function 7, which inverts matrix B and multiplies it by vector x, accounting for around 20 % of the cycles in the sign routine. In the following section, we will examine the steps of Function 7 and assess their impact on performance. Since Function 7 has the highest cycle count which is not associated with Keccak, it represents the most critical target for optimization, as there is still room for improvement.

Table 5.3 presents how Function 7 is comprised, detailing each function as it appears in the C reference implementation of HAWK's sign routine in pqm4. It also includes the average cycle count for each function and its contribution to the total cycle count of Function 7.

Step	Function	Average Cycles	Contribution
1	mq18433_poly_set_small_inplace_low(w1)	2 479	1.39 %
2	mq18433_poly_set_small_inplace_high(w2)	2 865	1.61 %
3	mq18433_NTT(w1)	30 583	17.20 %
4	mq18433_NTT(w2)	30 583	17.20 %
5	for (size_t u = 0; u < n; u ++) {w1[u] = q18433_montymul(w1[u], w2[u]);}	4 147	2.33 %
6	mq18433_poly_set_small(w2, x1)	2 353	1.32 %
7	mq18433_poly_set_small_inplace_high(w3)	2 865	1.61 %
8	mq18433_NTT(w2)	30 583	17.20 %
9	mq18433_NTT(w3)	30 583	17.20 %
10	for (size_t u = 0; u < n; u ++) {w3[u] = mq18433_tomonty(mq18433_sub(mq18433_montymul(w2[u], w3[u]), w1[u]));}	7 732	4.34 %
11	mq18433_iNTT(w3)	33 084	18.60 %

Table 3.3: Profiling results for Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations.

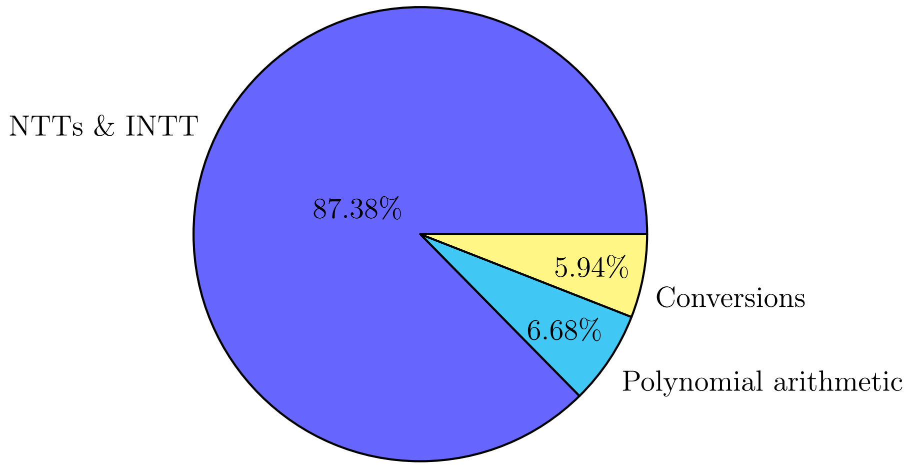

Figure 3.1: Profiling results for Function 7 divided into three categories.

Out of 177 857 total cycles, 155 416 cycles (87.38 %) are spent on the NTT and INTT, 11 879 cycles (6.68 %) on polynomial arithmetic, and 10 562 cycles (5.94 %) on converting polynomials to and from an 8-bit signed representation. These results highlight the NTT and INTT as key optimization targets.

Description of steps from Function 7

- **Step 1**
  Convert a small polynomial with signed 8-bit coefficients to an unsigned 16-bit representation where each coefficient lies within the modulus q. (Works on the first half of the input array.)
- **Step 2, 7**
  Convert a small polynomial with signed 8-bit coefficients to an unsigned 16-bit representation where each coefficient lies within the modulus q. (Works on the second half of the input array.)
- **Step 3, 4, 8, 9**
  Transform polynomial into NTT-domain.
- **Step 5**
  Apply component-wise multiplication to both polynomials in NTT-domain.
- **Step 6**
  Convert a small polynomial with signed 8-bit coefficients to an unsigned 16-bit representation where each coefficient lies within the modulus q. (Does not work in place.)
- **Step 10**
  Apply component-wise multiplication to both polynomials in NTT-domain. Then, subtract each resulting coefficient from each coefficient in w1. Finally, apply Montgomery reduction to each coefficient.

- **Step 11**
  Transform polynomial from NTT-domain back into the normal domain (inverse transformation).

Computational cost of the NTT and INTT
With the NTT and INTT consuming most cycles in Function 7, we now explain how these cycles arise in the C reference implementation. Due to factors like compiler optimizations, a detailed cycle count analysis is challenging, so we provide only a high-level overview of the performance. For Hawk-256 the NTT and INTT are calculated over 256 coefficients each, which results in a total of 1 024 butterfly operations each. Every butterfly operation contains one Montgomery multiplication, one addition with reduction, one subtraction with reduction, two reads from memory, and two writes to memory (+ one halving operation in the case of the INTT). The efficiency of these operations is crucial for the overall performance of the NTT and INTT, making their optimization essential for reducing the execution time of Function 7. Additionally, reassessing memory access patterns for potential improvements is worthwhile. Specific strategies for optimizing the NTT and INTT are discussed in the next chapter.

3. Largest contributor

The third largest contributor to the cycle count is Function 1, which re-expands the private key, accounting for roughly 5 % of the cycles in the sign routine. The majority of the cycles stem from `shake_extract` which is called four times in this function. For the same reasons outlined for Function 6, we will not go into further detail here.

Other contributors
The same applies to Functions 2, 3, and 4, where most of the cycles are consumed by a single call to `shake_extract` in each function.
Function 5 calculates the product of B and h reducing modulo 2 using a Karatsuba-like approach.
The remaining Functions 8, 9, and 10 primarily involve bit manipulations and memory operations.

4 Optimizations

In this thesis, we primarily focus on optimizing for speed, which typically comes at the expense of increased code size and higher stack usage. These trade-offs arise from techniques such as loop unrolling, which reduces the overhead of control flow at the cost of more instructions, and the use of additional local variables or temporary buffers to minimize recomputation and enhance performance. By prioritizing execution speed, we accept the impact on memory resources, as these changes are, in certain cases, essential to achieve significant gains in processing efficiency.

However, increased code size and stack usage are particularly critical on resource-constrained devices like the Cortex-M4, where available memory is limited. As such, a balance must be struck between speed optimization and memory usage. In some cases, such as with the NTT and INTT, limitations in Jasmin's current capabilities made it impossible to achieve acceptable performance without loop unrolling. Therefore, for these operations, unrolling was necessary to meet the performance goals as we will outline in Section 4.1.6.

4.1 Fast Constant-Time NTTs

As shown in the profiling results (cf. Figure 3.1), the NTT and INTT are the top priorities for optimization. Thus, this chapter primarily revolves around optimization techniques for these transformations. All listed optimizations for the NTT also apply to the INTT.

4.1.1 Polynomial Representation on Cortex-M4

Polynomials in HAWK have coefficients in a finite field $\mathbb{Z}_q$, where $q = 18\,433$ is the 15-bit prime modulus. Depending on the version of HAWK, the number of coefficients varies, with HAWK-256 using 256 coefficients, HAWK-512 using 512 coefficients, and HAWK-1024 using 1 024 coefficients. A coefficient can be represented as either a signed or unsigned integer.
The choice of representation affects both the arithmetic operations and modular reductions performed during the NTT as well as the implementation's overall efficiency. Using signed representation stems from the need to prevent the result from wrapping around zero when performing operations like $x - y$ with $y > x$. In the unsigned case, it would be necessary to repeatedly add multiples of the modulus q to bring the result back into the positive range. This issue is avoided when the result is interpreted as a signed integer, thus saving operations.

In contrast to the C reference implementation of HAWK, which uses an unsigned 16-bit integer representation for coefficients within the NTT, we adopt a signed representation. This approach eliminates concerns about underflows during subtractions in $\mathbb{Z}_q$ and saves us a multiple of q additions.

4.1.2 32-bit Arithmetic

While we could store the coefficients as signed 16-bit integers, there is a lot to gain when switching to signed 32-bit integers. As demonstrated in Algorithm 4, each step of the NTT involves a multiplication, addition, and subtraction operation. By adopting 32-bit arithmetic and using arrays of 32-bit signed integers for storing coefficients, we can optimize the way we apply these additions and subtractions. In the C reference implementation, each addition and subtraction function is designed to automatically perform a modulo q reduction after the operation. When we utilize 32-bit arithmetic, we do not need reductions after every addition or subtraction, thus, we can simply use plus (+) and minus (-) and let the coefficients grow. This is also known as Lazy Reduction (cf. Chapter 2.6.2).
When using 32-bit arithmetic, we can let the values grow during addition and subtraction without performing a reduction after each operation because the arithmetic operations are designed to work within the larger 32-bit space. This approach is valid due to the *natural injection*, which maps elements from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^k}$ without altering their values. The operations in $\mathbb{Z}_{2^k}$ are preserved through a *monomorphism*, ensuring that results in $\mathbb{Z}_{2^k}$ correctly correspond to those in $\mathbb{Z}_q$.
For signed arithmetic, this preservation is straightforward because modular operations naturally align with signed integer representations. This means that addition, subtraction, and multiplication in 32-bit signed integers directly reflect modular arithmetic properties.
In contrast, unsigned arithmetic requires additional steps to ensure that results,

particularly for subtraction, correctly map back to the modular space. Without these corrections, unsigned arithmetic might not preserve the desired modular results, which is why signed arithmetic is often preferred for this reason. By leveraging the larger 32-bit space and allowing values to grow, we simplify the implementation while still maintaining correctness through modular properties for signed integers.

4.1.3 Pre-computation of Twiddle Factors

An optimization technique for NTT computation involves pre-computing the twiddle factors, which are the powers of the root of unity, and storing them in flash memory. To improve the efficiency of modular reduction after multiplying by the twiddle factors, we adopt the method first presented in [8], where the twiddle factors are stored in Montgomery representation.
The reference implementation also uses this technique by scaling all values by 2^{32} mod $q = 4\,564$. Subsequently, we can reuse their table of twiddle factors for the NTTs. For the INTT, to avoid the additional halving operation used in the reference implementation, we account for this by scaling our twiddle factors by $4\,564$, whereas the reference implementation uses a scaling factor of $2\,282$.

4.1.4 Layer Merging

Layers of the NTT

The amount of layers of the NTT differs between the different HAWK candidates. HAWK-256 comprises 8 layers, HAWK-512 comprises 9 layers and HAWK-1024 comprises 10 layers to compute the NTT.
Before we go into detail about how to further optimize NTTs, we present a visual representation of the layers of HAWK-256's NTT (cf. Figure 4.1). At the top, the array `Coef` with 256 coefficients is shown which is to be transformed by using the NTT. For better visibility, we only show the array up to index 32. Each layer shows which pairs of coefficients pass through the *Cooley-Tukey Butterfly* (cf. Figure 2.11), referred to simply as a butterfly in this chapter.

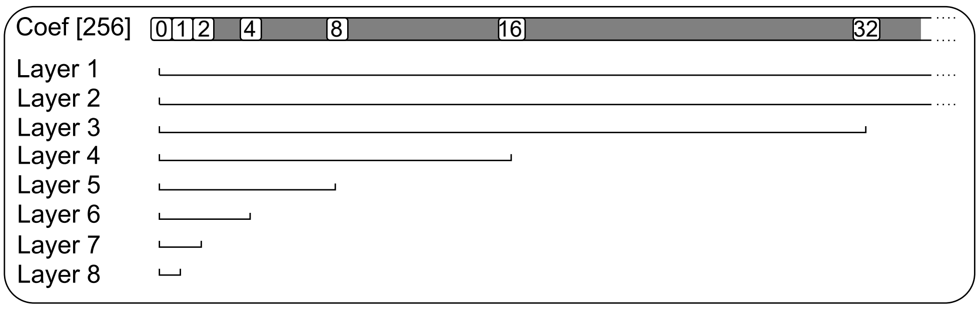

Figure 4.1: Layers of NTT for HAWK-256

A more detailed visualization for the first two layers is given in Figure 4.2.

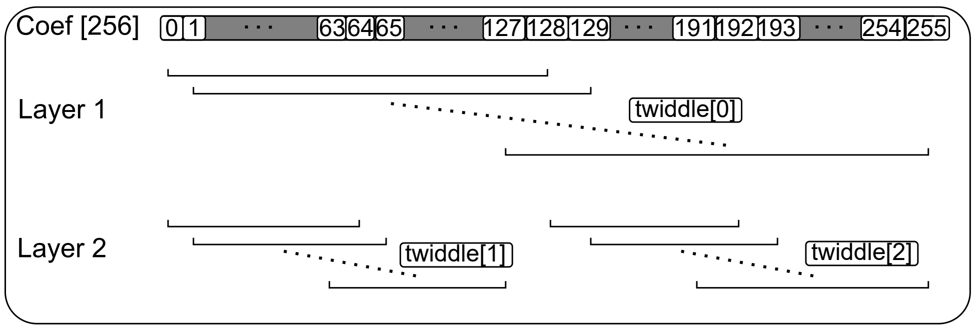

Figure 4.2: First two layers of NTT for HAWK-256

In **Layer 1** of the NTT, first, the pair consisting of `Coef[0]` and `Coef[128]` passes through the butterfly, then `Coef[1]` and `Coef[129]` until `Coef[127]` and `Coef[255]` are processed to finish the first layer. For Layer 1, the twiddle factor ω used in the butterfly operations is `twiddle[0]`.

In **Layer 2** of the NTT, first, the pair consisting of `Coef[0]` and `Coef[64]` passes through the butterfly, then `Coef[1]` and `Coef[65]` until `Coef[191]` and `Coef[255]` are processed to finish the second layer. For Layer 2, the twiddle factors used in the butterfly operations are `twiddle[1]` for all the pairs (`Coef[0]`, `Coef[64]`) to (`Coef[63]`, `Coef[127]`) and `twiddle[2]` for all the pairs (`Coef[128]`, `Coef[192]`) to (`Coef[191]`, `Coef[255]`).

This pattern continues for the remaining six layers as presented in Figure 4.1.
For example, in Layer 3, we utilize:

- twiddle factor `twiddle[3]` for all the pairs from (`Coef[0]`, `Coef[32]`) to (`Coef[31]`, `Coef[63]`).
- twiddle factor `twiddle[4]` for all the pairs from (`Coef[64]`, `Coef[96]`) to (`Coef[95]`, `Coef[127]`).
- twiddle factor `twiddle[5]` for all the pairs from (`Coef[128]`, `Coef[160]`) to (`Coef[159]`, `Coef[191]`).
- twiddle factor `twiddle[6]` for all the pairs from (`Coef[192]`, `Coef[224]`) to (`Coef[223]`, `Coef[255]`).

Merging Layers of the NTT

Layer merging is a technique used to optimize the computation of the NTT that has been incorporated into numerous optimization approaches [9, 44, 25]. It works by merging several layers before performing calculations, thus, saving memory operations. Theoretically, we could merge as many layers as we desire, but increasing the number of merged layers demands more space to store intermediate results. For this reason, on Cortex-M4, one typically has the choice to merge two or three layers at a time [25]. Our strategy focuses on merging two layers at a time. To illustrate this concept, we present a visual representation in Figure 4.3 that shows how the first two layers are merged.

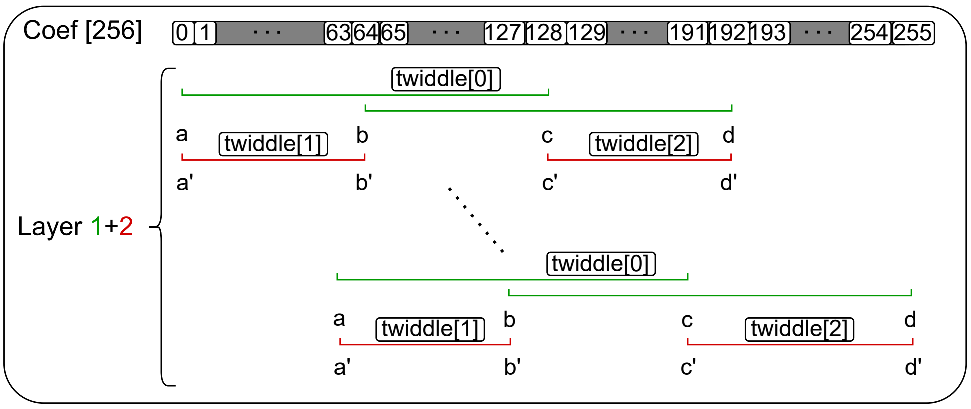

Figure 4.3: Merging first two layers of NTT

In the merged layer 1+2 of the NTT, first, we load the four coefficients `Coef[0]`, `Coef[64]`, `Coef[128]`, and `Coef[192]`. With these values loaded, we apply butterflies of layer 1 to the pair of (`Coef[0]`, `Coef[128]`) and (`Coef[64]`, `Coef[192]`) to obtain (`a`, `c`), and (`b`, `d`), respectively. Then, we take these results from layer 1 and use them to form pairs as inputs for the butterflies of layer 2, which are (`a`, `b`) and (`c`, `d`). After computing the butterflies of the second layer, we obtain `a'`, `b'`, `c'` and `d'`. These are the results after the first two layers of NTT for 4 of the 256 coefficients. We store the results back into the array, such that `Coef[0]` = `a'`, `Coef[64]` = `b'`, `Coef[128]` = `c'`, and `Coef[192]` = `d'`. This procedure is repeated another 63 times by loading the next pairs of coefficients, i.e. `Coef[1]`, `Coef[65]`, `Coef[129]` and `Coef[193]` and repeating the previously outlined steps.
The same approach can be used to merge other layers, such as layers 3 and 4, layers 5 and 6, and layers 7 and 8.

This example shows that with just four load and four store operations, we can compute two NTT layers for four coefficients. If we did not merge layers, the same number of operations would be required for four coefficients of only one layer. Therefore, merging two layers cuts the load and store operations by 50 %.

Merging three layers at a time is another viable approach and can, in some cases, outperform the two-layer merging strategy we employ [25]. However, to achieve this efficiently, we must utilize the floating-point registers (cf. Chapter 2.8.1) on our board to store frequently accessed variables, thereby minimizing memory load and store operations. Thus, data transfer to and from floating-point registers is performed using the `vmov` instruction, which requires only one cycle in each direction. This method, initially introduced in [11] to efficiently merge three layers, has demonstrated greater efficiency compared to merging two layers at a time.

We do not implement the three-layer merging strategy because the Jasmin programming language currently does not support the `vmov` instruction. Furthermore, since this thesis focuses on optimizing Hawk for the Cortex-M4, utilizing this approach would be unsuitable, as the standard Cortex-M4 lacks a FPU and, consequently, does not have floating-point registers. Therefore, this technique is only applicable to the Cortex-M4F.

4.1.5 Use of Incomplete NTTs

Although incomplete NTTs were first introduced into lattice-based cryptography due to moduli that do not support a complete NTT [62], they were later intentionally used even when full NTTs were feasible, as demonstrated in [10]. This choice was motivated by performance benefits associated with using incomplete NTTs. When working with incomplete NTTs, it is necessary to decide at what layer to stop performing butterflies and switch to multiplying polynomials using the schoolbook method. For example, in the case of Hawk-256 with a total of 8 layers, options include using 7 layers followed by 2x2 schoolbook multiplication, 6 layers followed by 4x4 schoolbook multiplication, or 5 layers followed by 8x8 schoolbook multiplication. In this thesis, we decide on 6 layers followed by 4x4 schoolbook multiplication for Hawk-256 because it strikes the best balance between computational efficiency, register usage, and minimizing cycle count, ensuring optimal performance on constrained platforms like the ARM Cortex-M4 [25, 1]. Thus, we compute an incomplete NTT for the layers 1 to 6 (skipping 7 & 8). For layers 1 to 6, we apply layer merging to 1 & 2, 3 & 4, and 5 & 6 (cf. Chapter 4.1.4).

Algorithm 9 outlines the procedure for performing 4x4 schoolbook multiplications. In this process, we take two input polynomials, `a` and `b`, each consisting of four coefficients. Also, we choose the corresponding twiddle factor ζ'. The resulting reduced product is `c`, represented by four coefficients as well.
Here, we utilize Cortex-M4's powerful 1-cycle long multiplications `smull` and `smlal`, that achieve a signed long multiply, and a signed long multiply with accumulate, respectively. Thus, the 64-bit result of these operations is stored in the two 32-bit registers `x` and `y`. To reduce the result we make use of the Montgomery Reduction (cf. Algorithm 8), labeled here with `montgomeryR(x,y)` that takes two 32-bit input values `x` and `y` instead of one 64-bit value, because on this architecture we cannot operate on 64-bit registers.

Algorithm 8 montgomeryR, following [25, Artifacts][1]

```
Require: x, y, q = 18433, q^-1 = 3955247103
Ensure: x||y (64-bit) ↦ (x||y)·$R^{-1}$ mod q
1: mul temp, y, q^-1
2: smlal x, y, temp, q
3: return y
```

[1] https://artifacts.iacr.org/tches/2021/a7/

Algorithm 9 4 x 4 schoolbook, following [25, Algorithm 17]

```
Require: {a_0, a_1, a_2, a_3}
Require: {b_0, b_1, b_2, b_3}
Require: $\zeta' = \left(2^{32}\zeta \mod q'\right)$
Ensure: {c_0, c_1, c_2, c_3} such that
        $c_0 = (a_0b_0 + \zeta(a_1b_3 + a_2b_2 + a_3b_1))/2^{32} \mod q'$
        $c_1 = (a_0b_1 + a_1b_0 + \zeta(a_2b_3 + a_3b_2))/2^{32} \mod q'$
        $c_2 = (a_0b_2 + a_1b_1 + a_2b_0 + \zeta a_3b_3)/2^{32} \mod q'$
        $c_3 = (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)/2^{32} \mod q'$
 1: smull x, y, $a_1$, $b_3$
 2: smlal x, y, $a_2$, $b_2$
 3: smlal x, y, $a_3$, $b_1$
 4: y = montgomeryR(x, y)
 5: smull x, y, y, $\zeta'$
 6: smlal x, y, $a_0$, $b_0$
 7: y = montgomeryR(x, y)
 8: $c_0$ = y
 9: smull x, y, $a_2$, $b_3$
10: smlal x, y, $a_3$, $b_2$
11: y = montgomeryR(x, y)
12: smull x, y, y, $\zeta'$
13: smlal x, y, $a_0$, $b_1$
14: smlal x, y, $a_1$, $b_0$
15: y = montgomeryR(x, y)
16: $c_1$ = y
17: smull x, y, $a_3$, $b_3$
18: y = montgomeryR(x, y)
19: smull x, y, y, $\zeta'$
20: smlal x, y, $a_0$, $b_2$
21: smlal x, y, $a_1$, $b_1$
22: smlal x, y, $a_2$, $b_0$
23: y = montgomeryR(x, y)
24: $c_2$ = y
25: smull x, y, $a_0$, $b_3$
26: smlal x, y, $a_1$, $b_2$
27: smlal x, y, $a_2$, $b_1$
28: smlal x, y, $a_3$, $b_0$
29: y = montgomeryR(x, y)
30: $c_3$ = y
```

4.1.6 Performance-Driven Implementation Choices

In this section, we outline key design choices made during the implementation of the NTT and INTT. We opted to use `for` loops for our implementation instead of `while` loops. This decision was driven by performance considerations, as using `while` loops would have led to a 30 % increase in cycle counts. Interestingly, this performance difference was not due to the typical penalties associated with `while` loops, such as the overhead of branching in each iteration. Instead, it was primarily related to memory access patterns and the constraints of pointer arithmetic in Jasmin. To understand the problem, we present our code for three implementations of the NTT for merging the first two layers.

The first implementation of the NTT utilizes a `while` loop and is getting the pointer of array `a` passed as a `reg u32` in Jasmin. As a result, this function can only be used as an export function in C, not in our optimized sign routine. In the optimized sign routine, we must work with Jasmin arrays, where pointers are stored in `reg ptr` rather than `reg u32`.

```
 1 reg u32 m = a + 256;
 2 while (a < m)
 3 {
 4     x1 = (32u)[a];
 5     x2 = (32u)[a + 256];
 6     x3 = (32u)[a + 512];
 7     x4 = (32u)[a + 768];
 8     x1, x2, x3, x4 = butterfly_over_two_layers(x1, x2, x3, x4, ...);
 9     (u32)[a] = x1;
10     (u32)[a + 256] = x2;
11     (u32)[a + 512] = x3;
12     (u32)[a + 768] = x4;
13     a += 4;
14 }
```

Listing 4.1: Code snippet for merging the first two layers, using a while loop and pointer arithmetic

From Listing 4.1, it is evident that pointer arithmetic is used: In each iteration of the while loop, the address `a` is incremented by 4, causing `a` to point to the next 32-bit value. During each memory access (lines 4 to 7), offsets are taken into account without causing any extra instructions. Therefore, the only overhead introduced by using a `while` loop is the comparison and branching in each iteration, as well as the increment of `a` in each iteration.

Next, we will examine what happens when we adapt the approach from Listing 4.1 to work with a Jasmin array, where the pointer is stored as a `reg ptr`. Since pointer arithmetic is not possible in this case, we developed the approach shown in Listing 4.2.

```
 1  reg u32 ctr = 0;
 2  while (ctr < 256)
 3  {
 4      temp1 = ctr+256;
 5      temp2 = ctr+512;
 6      temp3 = ctr+768;
 7      x1 = (32u)a.[ctr];
 8      x2 = (32u)a.[temp1];
 9      x3 = (32u)a.[temp2];
10      x4 = (32u)a.[temp3];
11      x1, x2, x3, x4 = butterfly_over_two_layers(x1, x2, x3, x4, ...);
12      a.[ctr] = (32u) x1;
13      temp1 = ctr+256;
14      a.[temp1] = (32u) x2;
15      temp2 = ctr+512;
16      a.[temp2] = (32u) x3;
17      temp3 = ctr+768;
18      a.[temp3] = (32u) x4;
19      ctr += 4;
20  }
```

Listing 4.2: Code snippet for merging the first two layers, using a while loop and no pointer arithmetic

Due to architectural constraints, we cannot utilize operations of the sort: `x2 = (32u)a.[ctr + 256];` where we would calculate an address by taking the base address and adding a register offset and immediate offset in one operation. As a result, this line fails to compile due to an error indicating that the address computation is too complex. To work around this limitation, we introduce additional instructions to calculate separate offsets (lines 4 to 6) to comply with the addressing modes of our architecture (cf. Section 2.8.1). However, due to register pressure when using this NTT function, we must recompute the temporary values (lines 13, 15, 17). While spilling the values to the stack before line 11 and retrieving them afterward is an option, it does not improve performance.
In the absence of pointer arithmetic, we end up adding six extra instructions per iteration, further impacting performance. We sought solutions in a Zulip channel hosted by Formosa Crypto; several were suggested, but despite minor improvements, none matched the efficiency of using pointer arithmetic.

When this version of the NTT was integrated into the optimized sign implementation, it was not possible to keep all three temporary variables live simultaneously due to register allocation constraints. This prevented consecutive loading, further degrading performance.

As a result, we had to develop a different approach. Therefore, in our optimized approach we switch to `for` loops as shown in Listing 4.3.

```
1  inline int i;
2  inline int index1 = 0;
3  inline int index2 = 256;
4  inline int index3 = 512;
5  inline int index4 = 768;
6  for i = 0 to 64
7  {
8      x1 = (32u)a.[index1];
9      x2 = (32u)a.[index2];
10     x3 = (32u)a.[index3];
11     x4 = (32u)a.[index4];
12     x1, x2, x3, x4 = butterfly_over_two_layers(x1, x2, x3, x4, ...);
13     a.[index1] = (32u) x1;
14     a.[index2] = (32u) x2;
15     a.[index3] = (32u) x3;
16     a.[index4] = (32u) x4;
17     index1 += 4;
18     index2 += 4;
19     index3 += 4;
20     index4 += 4;
21 }
```

Listing 4.3: Code snippet for merging the first two layers, using a for loop

In this `for` loop, we utilize `inline ints`, which are inlined at compile time. As a result, when computing two layers of butterflies, as shown in Listing 4.3, the only cycles used are for loading, computing the butterflies, and storing the results. Lines 17-20 incur no cycles because they are inlined within the unrolled loop. Compared to the `while` loop implementation with pointer arithmetic in Listing 4.1, we also avoid the overhead associated with the `while` loop. As such, this unrolled version utilizing a `for` loop is roughly 13 % faster than the implementation of Listing 4.1. Thus, we use this strategy for implementing our optimized NTT. The same applies to our optimized implementation of the INTT.

However, it is important to note that this unrolling results in an increased code size, which remains a significant concern on resource-constrained devices such as the Cortex-M4.

4.2 Other Routines

After discussing optimization techniques for the NTT, INTT, and polynomial multiplication, we now focus on the remaining functions. Based on the profiling results, we found that, aside from the NTT, INTT, polynomial arithmetic, and SHAKE-related components, most of the remaining operations involve bit-level manipulations and memory operations. As such, we will outline the general strategies used to optimize these routines as effectively as possible.

4.2.1 Memory Operations

On the Cortex-M4, the pipeline architecture is optimized for consecutive memory operations, allowing multiple memory reads to be processed efficiently by the pipeline stages. To take advantage of this, we aim to group memory operations wherever possible, provided that register pressure permits. Specifically, according to ARM's documentation on load/store timings for the M4 processor[2], for an `LDR` instruction, the address calculation occurs during the first cycle of execution, with the data being retrieved in the following cycle. As a result, each `LDR` instruction requires two cycles. When consecutive load operations are performed, the processor can recognize that the next operation is also a memory access and begins calculating its address while the pipeline is still waiting for the data from the previous load. Consequently, for a series of N consecutive load operations, the total cycle count is $N + 1$. On the other hand, an `STR` instruction always takes one cycle. This is because the address generation occurs during the first cycle, and the data is stored simultaneously with the execution of the next instruction.

4.2.2 Unrolling Loops

In most of our implementations, we utilize Jasmin for loops, which the compiler unrolls. This approach eliminates the loop overhead caused by comparisons and branching in each iteration. Combined with inline integers, we avoid using additional registers for a loop counter. Moreover, due to the unrolled nature of the loop, we can directly access memory at specific locations without needing to compute the index in each iteration, further saving operations and registers in some cases. By conserving registers where possible, we also minimize the need for spilling, which occurs when register contents must be moved to memory due to insufficient register space, for example, if all 14 general-purpose registers are occupied.

[2] https://developer.arm.com/documentation/100166/0001/ric1417175925887

4.2.3 Merging `poly_snorm` and `poly_symbreak`

Function `poly_snorm` is the last operation within Function 7: $w \leftarrow B^{-1}x$ (cf. Table 5.3). It is directly followed by `poly_symbreak`. Both functions iterate through the same array during each iteration. To optimize these functions, the approach is to merge them, cutting memory operations—specifically reads—in half, by fetching the value once and reusing it in both functions.

4.2.4 Optimizing Leaf Functions in `basis_m2_mul`

Function `basis_m2_mul` is the dominant factor to cycles within Function 5: $t \leftarrow B \cdot h \mod 2$ (cf. Table 5.3). It employs a Karatsuba-like multiplication, where a function XORs 64-bit values at the leaf level. It achieves this by running a loop with 8 iterations, where in each iteration, it loads one byte from each array, XORs the values, and stores the result back in memory. A simple optimization for this function is to leverage the full width of the processor, i.e., 32 bits. By loading 4 bytes at once from each array, XORing them, and storing the result in a single operation, we reduce both memory and XOR operations by a factor of 4. To optimize this function further, we eliminate the loop to group memory operations together. This way, we perform four consecutive read operations to fetch 2 words (64 bits) from each array, calculate two XORs, and then store the results with two consecutive store operations. The same procedure is repeated for functions XORing 128-bit values and 256-bit values.

Additionally, deeper in the function `basis_m2_mul`, we can reorder the code so that there are instances where we perform an XOR on 128-bit values followed by two XORs on 64-bit values consecutively using the same input. To optimize this, we merge the three separate functions into a single one, allowing us to load the values only once and reuse them through both levels.

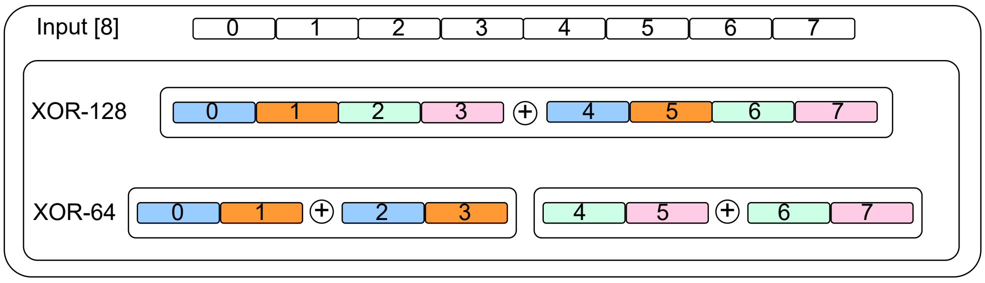

Figure 4.4: Merging layers of XOR computations

This approach is illustrated in Figure 4.4. We have an array of 8 words (32 bytes) that serves as the input for both the XOR-128 and XOR-64 functions. To optimize, we begin by loading all 8 words from the input and using them to compute both XOR functions. Specifically, for XOR-128, we XOR word 0 with word 4, word 1 with word 5, word 2 with word 6, and word 3 with word 7. Similarly, for XOR-64, we XOR word 0 with word 2, and word 1 with word 3, etc. The results are then stored in the appropriate output arrays for both functions.

The XOR-128 function requires 12 memory instructions (8 loads and 4 stores), and two XOR-64 functions require another 12 instructions (8 loads and 4 stores), adding up to 24 memory operations. In this merged version, the total of memory operations is reduced by one-third to 16 memory operations, consisting of 8 loads and 8 stores. The amount of XOR operations stays the same with a total of 8.

5 Results

In this chapter, we demonstrate how to integrate our Jasmin code within the pqm4 framework. Next, we present the methods used to test our implementation of Hawk for correctness. This is followed by the benchmarking and profiling results of our optimized implementation of Hawk-256's sign routine, comparing them to the baseline implementation. Lastly, we present a security evaluation of our approach.

5.1 Integrating Jasmin with HAWK-256 on Cortex-M4

After compiling our Jasmin code (`.jazz` file), we obtain an assembly file (`.s` file). To integrate the Jasmin code with the Hawk-256 implementation on the Cortex-M4, we begin by duplicating the existing Hawk-256 implementation in pqm4 and renaming the copy to include the `m4` suffix. This naming convention is flexible and can be adjusted as needed. Finally, the resulting assembly file is placed in the new Hawk-256/m4 directory.

To test or run benchmarks on the optimized sign routine, we need to modify the `crypto_sign` function in the `api.c` file within this folder to call our Jasmin function `sign_finish_inner_jazz(sm+mlen, sm, sk, mlen)` instead of the C-based version `hawk_sign_finish`. Once this change is made, we can rebuild the code to include our new Hawk-256/m4 scheme. This ensures that when we execute the tests provided by pqm4, they will evaluate our Jasmin implementation.

For profiling, the `crypto_sign` function in the `api.c` file must call the C-based version `hawk_sign_finish`, as we will invoke our export functions and perform profiling from there. The Jasmin function we want to profile must be an export function. If ABI restrictions prevent us from directly turning an inline function export, we need to wrap it in an export function. As before, we then compile the Jasmin code into an assembly file, place it in the appropriate location, and add the export function as an extern declaration in the `modq.h` file. This allows us to call the function from within the C code of the signature routine and to perform profiling.

The methodology for benchmarking and profiling remains the same as in Section 3.2.1 & Section 3.3.1.

5.2 Testing

To test the correctness of our implementation of HAWK-256's sign routine, we leverage two tests from pqm4:
The first, called `test.c`, repeatedly generates key pairs, then signs random messages of length N, and checks if the signatures are correctly verified under valid and invalid conditions. It also employs canary values around allocated memory to detect buffer overflows, ensuring that implementations do not inadvertently alter memory boundaries. To employ this test, we flash the resulting `test` binary for HAWK-256 onto the board, run it, and retrieve the results. We repeat this test 10 000 times for message lengths of 32 bytes (the default).
The second, called `testvectors.c`, uses a deterministic random number generator to generate test vectors for the implementation[1]. To employ this test we utilize pqm4's Python script `testvectors.py` that executes all available implementations for HAWK-256 on the board and the reference implementation on the host. It then checks if all schemes produce the same output, confirming that each generates the same signature. For each run of this test, random messages of varying lengths (ranging from 0 to a maximum of 2 048 bytes) are used to create signatures, with the length doubling at each step. This serves to further validate that our implementation matches the reference.

We consider this testing setup effective for identifying significant implementation flaws, though it is not a substitute for formal verification of the implementation.

5.3 Benchmarking & Profiling

5.3.1 Benchmarking

To initiate the evaluation of our optimized implementation, we measure the total cycle counts required to sign random 59-byte messages over 10 000 times. The resulting data is presented as follows:

Routine	Average	Standard Deviation	Coefficient of Variation (%)
Sign	887 028.10	289 261.96	32.61

Table 5.1: Performance metrics for signing routine of HAWK-256

As in Section 3.3.2, we will now concentrate exclusively on single-attempt runs. The mean cycle count across these remaining single-attempt runs is 776 387 cycles,

[1] To ensure consistency with our implementation, the buffers of the reference implementation, which would otherwise be filled with random bytes, must be prefilled with the same constant values as those used in our optimized implementation prior to executing this test (cf. Section 5.7).

with a standard deviation of 12 cycles and a coefficient of variation of 0.00155 %, demonstrating the same low variability as the reference.

5.3.2 Profiling

The cumulative cycle count of 774 786 cycles is the sum of the cycles of the main functions within HAWK-256's sign routine, as listed below. This total is slightly lower than the baseline of 776 387 cycles because it only includes the primary functions, excluding some minor auxiliary operations.

Function	Description	Average Cycles	Contribution	References from Algorithm 2
1	Re-expand k_{priv}	47 025	6.07%	n.a.
2	$M \leftarrow H(m\|\|h_{pub})$	14 253	1.84%	Step 1
3	$salt \leftarrow Rnd(saltlen)$	11 480	1.48%	Step 2
4	$h \leftarrow H(m\|\|salt)$	11 612	1.50%	Step 3
5	$t \leftarrow B \cdot h \mod 2$	14 058	1.81%	Step 4
6	$x \leftarrow D_{t/2}(\sigma_{sign})$	610 926	78.85%	Step 5
7	$w \leftarrow B^{-1}x$	51 045	6.59%	Step 9
8	snorm-sym-break(w)	4 167	0.54%	n.a.
9	$s_1 \leftarrow Compress(s)$	4 424	0.57%	Step 13, 14
10	sig-encoding$(salt, s_1)$	5 796	0.75%	Step 15

Table 5.2: Profiling results for major routines in HAWK-256's sign routine over 10 000 iterations

This breakdown shown in Table 5.2 contains the same functions as the reference implementation with the exception of the `snorm-sym-break` function. This function is the result of the optimization discussed in Section 4.2.3, where the `snorm` and `sym-break` functions are merged.

Additionally, we exclude all functions (grayed out) that primarily incur their cycles due to SHAKE, as they are not part of the optimization process, for the reasons outlined in Section 3.3.2. However, since our goal is to implement the entire sign routine in Jasmin, and the optimized Keccak version from the reference is not suitable because external functions cannot be used in Jasmin, we opted for an optimized Jasmin implementation of Keccak[2]. In fact, the optimized Keccak version in Jasmin is slightly slower than the one used in the reference. Regardless, we have strived to optimize these functions as much as possible to minimize their computational overhead.

[2] https://gitlab.com/formosa-dilithium/dilithium-arm/-/blob/8dd851a1436b96c6af718e4e44ed4a5aee2db4b3/src/libjade/dilithium5/keccakf1600_pqm4.jinc

1. Largest contributor

As shown in Table 5.2, the largest contributor to the total cycles for the sign routine (excluding SHAKE-related functions) is Function 7, which inverts matrix B and multiplies it by vector x, accounting for around 7 % of the cycles in the sign routine. In the following, we will present how our optimized Function 7 is comprised[3] (cf. Table 5.3). As mentioned in Section 4.1.2, our optimized approach for polynomial arithmetic in Function 7 utilizes 32-bit arithmetic. As a result, the functions in the table below are not compatible with the 16-bit input arguments used in the reference, since our implementation uses 32-bit sizes. Consequently, we omit the input arguments and return parameters, focusing solely on the performance of the functions.

Step 5 employs a different function compared to the reference. In the reference, the polynomials are multiplied component-wise. However, in our optimized approach, we use an implementation that multiplies polynomials of size four. This is due to our use of an incomplete NTT, which prevents component-wise multiplication (cf. Section 4.1.5).
The same applies to Step 10, where we additionally use a single function that performs both the subtraction and the scaling.

Step	Function	Average Cycles	Contribution
1	polysetsmallinplacelow_jazz	2 114	4.11%
2	polysetsmallinplacehigh_jazz	2 113	4.10%
3	NTT_jazz	5 993	11.64%
4	NTT_jazz	5 993	11.64%
5	4x4schoolbook_jazz	4 849	9.42%
6	polysetsmall_jazz	1 685	3.27%
7	polysetsmallinplacehigh_jazz	2 113	4.10%
8	NTT_jazz	5 993	11.64%
9	NTT_jazz	5 993	11.64%

10	w3 = 4x4schoolbook_jazz
	w3 = subtractscale_jazz
		6 817	13.24%

11	iNTT_jazz	7 826	15.20%

Table 5.3: Profiling results for optimized Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations

[3]The total of all individual steps in Function 7 amounts to 51,490 cycles, approximately 400 cycles more than the measured value for the entire function. This discrepancy arises from measurement overhead and potential optimization effects, such as caching and instruction pipelining, when the functions are executed sequentially.

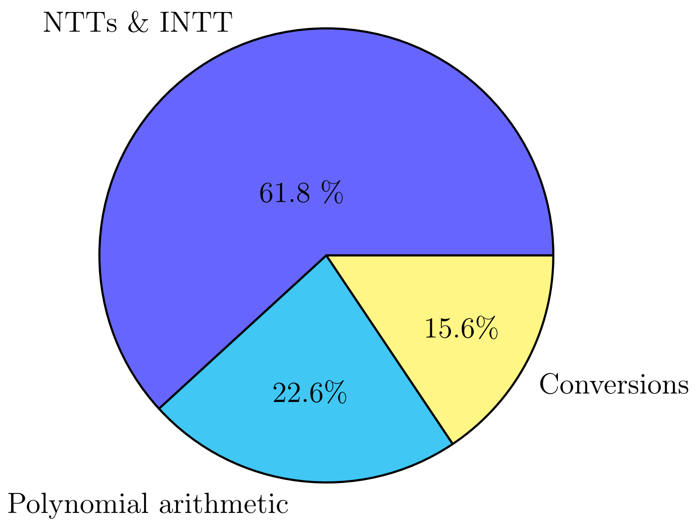

Figure 5.1: Profiling results for optimized Function 7 divided into three categories

Out of 51 490 total cycles, 31 798 cycles (61.8 %) are spent on the NTT and INTT, 11 666 cycles (22.6 %) on polynomial arithmetic, and 8 025 cycles (15.6 %) on converting polynomials to and from an 8-bit signed representation.

5.4 Cycle Count Evaluation

This section provides a breakdown of the cycle counts for our NTT and INTT implementation, covering both arithmetic and memory operation cycles. We base our cycle count evaluation on the *ARM Cortex-M4 Instruction Set Summary*[4].
We exclude prologue and epilogue instructions from our analysis because they are not executed in the final function due to inlining. Prologue instructions set up the stack frame and save the state of registers at the start of a function. However, when a function is inlined, its code is directly embedded into the calling function, making a separate stack frame unnecessary. Consequently, these instructions do not contribute to the cycle count in the optimized implementation. However, when profiling in pqm4, the prologue and epilogue are included in the cycle count because the function must be converted into an exportable function to be used in pqm4. This is one of the reasons why the cycle counts we measure are slightly higher than our calculations.

5.4.1 Cycle Analysis of the NTT Implementation

By inlining the butterfly functions and using unrolled loops, we eliminate extra cycles from branching.

Arithmetic Operations
First, we exclude memory operations and focus on the cycle count for the butterfly operations. A butterfly operation refers to the computation of butterflies across two layers of the NTT. The butterfly operations are grouped as follows:

- Merged Layers 1 & 2: 64 butterfly operations
- Merged Layers 3 & 4: 64 butterfly operations
- Merged Layers 5 & 6: 64 butterfly operations

Each butterfly operation involves 4 `ADD`, 4 `SUB`, and 4 calls to Montgomery multiplication functions. The Montgomery multiplication is implemented using 1 `SMULL`, 1 `MUL`, and 1 `SMLAL`.

[4]https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC

Each instruction is executed in a single cycle, leading to a total of 20 cycles per butterfly operation. With 192 butterfly operations in total, the arithmetic operations require:

$$192 \times 20 = 3\,840 \text{ cycles.}$$

Thus, the total arithmetic cycle count is 3 840 cycles.

Memory Operations
Next, we examine the cycles required for memory operations, specifically focusing on load and store operations, where consecutively loading N values requires $N + 1$ cycles and storing N values requires N cycles.

Loading the Twiddle Factors
The twiddle factors are loaded in the following sequence:

- Merged Layers 1 & 2: 1 group of 3 consecutive loads, requiring 4 cycles
- Merged Layers 3 & 4: 4 groups of 3 consecutive loads, requiring 16 cycles
- Merged Layers 5 & 6: 16 groups of 3 consecutive loads, requiring 64 cycles

The total cycles required for loading the twiddle factors are:

$$4 + 16 + 64 = 84 \text{ cycles.}$$

Loading the Coefficients
The coefficient loading operations occur as follows:

- Merged Layers 1 & 2: 1 set of 64 groups of 4 consecutive loads, requiring 320 cycles
- Merged Layers 3 & 4: 4 sets of 16 groups of 4 consecutive loads, requiring 320 cycles
- Merged Layers 5 & 6: 16 sets of 4 groups of 4 consecutive loads, requiring 320 cycles

The total cycles for loading the coefficients are:

$$3 \times 320 = 960 \text{ cycles.}$$

Storing the Coefficients
The store operations are required to store the computed coefficients:

- Merged Layers 1 & 2: 1 set of 64 groups of 4 stores, requiring 256 cycles
- Merged Layers 3 & 4: 4 sets of 16 groups of 4 stores, requiring 256 cycles
- Merged Layers 5 & 6: 16 sets of 4 groups of 4 stores, requiring 256 cycles

The total cycles for storing the coefficients are:

$$3 \times 256 = 768 \text{ cycles.}$$

Total Cycle Count
By summing the cycles for the arithmetic operations and the memory operations, we obtain the total cycle count for the NTT:

$$\begin{aligned} & 3\,840\,\text{(butterfly operations)} \\ + & \quad 84\,\text{(load twiddle factors)} \\ + & \quad 960\,\text{(load coefficients)} \\ + & \quad 768\,\text{(store coefficients)} \\ = & \; 5\,652\,\text{cycles.} \end{aligned}$$

Given that the total measured cycles for the NTT is 5 993, the remaining cycles are:

$$5\,993 - 5\,652 = 341 \text{ cycles.}$$

This difference can be attributed to the fact that the numbers provided above are based on an ideal scenario, which is unlikely to occur in real-world conditions. Factors such as pipeline stalls and occasional cache misses result in additional cycles beyond the ideal case.

5.4.2 Cycle Analysis of the INTT Implementation

From a cycle count perspective, the INTT operates equivalently to the NTT until the final two merged layers. This is because the inverse butterfly operations utilize essentially the same instructions as the forward butterfly operations, albeit in a different sequence. To reduce memory operations during the final scaling step, this scaling is integrated directly into the last two merged layers (Layers 1 and 2). Consequently, the following explanation focuses solely on the origin of the additional cycles.

To incorporate the scaling into the Merged Layers 1 & 2 which is handled last in the INTT, we need the following:

- 4 Montgomery multiplication functions (its components require 3 cycles, plus 1 cycle because one parameter must be passed as a constant due to register pressure, necessitating it to be loaded into a register each time the function is called.)
- 4 `MLS` instructions, that consume 2 cycles each.
- 4 `SMMUL` instructions, that consume 1 cycle each.

This adds 28 cycles per iteration in Merged Layers 1 & 2, totaling 1 792 additional cycles over 64 iterations. Thus, the INTT theoretically requires 7 444 cycles ($5\,652 + 1\,792$), slightly less than the measured 7 826 cycles, due to the same factors mentioned in Section 5.4.1.

5.5 Comparison

In the following section, we will compare the performance of our optimized sign routine with the reference. This involves analyzing and comparing the benchmarking and profiling results of both routines. Additionally, we evaluate the performance of our NTT and INTT against another optimized implementation. We start by giving a comparison of the benchmarking results:

Benchmarking

Routine	Average	Standard Deviation	Coefficient of Variation (%)
Sign (ref.)	1 001 410.10	286 861.33	28.65
Sign (opt.)	887 028.10	289 261.96	32.61

Table 5.4: Comparison of performance metrics for signing routine of Hawk-256

The benchmarking comparison in Table 5.4 shows that our optimized implementation is roughly 115 000 cycles (or 11.42 %) faster, averaged over 10 000 runs. Examining the remaining single-attempt runs, the reference takes 892 469 cycles, while our implementation uses 776 387 cycles. This results in approximately 116 600 fewer cycles, making our implementation 13.00 % faster. To pinpoint the source of this improvement, we will now examine and compare the profiling results:

Profiling

For the same reasons outlined for Table 5.2, we have grayed out all SHAKE-related functions. To briefly explain, most of our functions that utilize SHAKE in our optimized version are marginally slower than the reference due to the slightly slower performance of the underlying optimized Keccak in Jasmin.

The largest speedup, a factor of 3.47, is achieved in Function 7, where we save roughly 126 000 cycles. For the remaining functions, which do not include SHAKE operations, we observe moderate speedups ranging from 1.37 x to 1.73x. This is primarily because these functions involve bit manipulation and memory operations, which have limited potential for optimization. The methods employed for these optimizations are described in Section 4.2.
In total, the functions not related to SHAKE account for approximately 221 000 cycles in the reference implementation. In our optimized version, this portion is reduced to about 79 500 cycles. This results in a reduction of 141 500 cycles, which is roughly 64 % fewer cycles than the reference, corresponding to a speedup of 2.78x.

Function	Description	Average Cycles (ref.)	Average Cycles (opt.)	Speedup
1	Re-expand k_{priv}	48 588	47 025	
2	$M \leftarrow H(m\|\|h_{pub})$	10 615	14 253	
3	$salt \leftarrow Rnd(saltlen)$	11 517	11 480	
4	$h \leftarrow H(m\|\|salt)$	11 026	11 612	
5	$t \leftarrow B \cdot h \mod 2$	19 815	14 058	1.41
6	$x \leftarrow D_{t/2}(\sigma_{sign})$	589 435	610 926	
7	$w \leftarrow B^{-1}x$	176 908	51 045	3.47
8	snorm(w)	2 351	4 167	1.37
9	sym-break(w)	3 375		
10	$s_1 \leftarrow Compress(s)$	6 197	4 424	1.40
11	sig-encoding$(salt, s_1)$	10 040	5 796	1.73
Total		892 218	774 786	1.15

Table 5.5: Comparison of the profiling results for major routines in HAWK-256's sign routine over 10 000 iterations

To gain a deeper understanding of the significant performance improvement in Function 7, we compare the profiling results of our optimized version with the reference in Table 5.6.

First, we observe that all of our optimized functions are faster than the reference, except for Step 5. This is not surprising, as the multiplication in Step 5 operates on polynomials of size 4, which is more computationally expensive than performing component-wise multiplication.

The largest speedups are seen in Steps 3, 4, 8, and 9, which involve computing the NTT. In these steps, our optimized version is over five times faster than the reference. The second largest speedup occurs in the INTT, which is over four times faster than the reference.

To understand how our implementation performs relative to the literature, we compare our results with those reported in [1], where their incomplete NTT for 256 coefficients requires 5 853 cycles and the INTT requires 7 137 cycles. Our implementation is just 140 cycles more for the NTT and 689 cycles more for the INTT. The implementation described in [1] achieves higher performance by employing the strategy of merging three layers at a time, as detailed in Section 4.1.5.

Stack Usage

We quantify the stack usage of our optimized implementation and the reference using the pqm4 stack test. The reference implementation requires 3 320 bytes of stack memory, while our optimized implementation uses 8 064 bytes.

Step	Function	Average Cycles (ref.)	Average Cycles (opt.)	Speedup
1	Polysetsmallinplacelow	2 479	2 114	1.17
2	Polysetsmallinplacehigh	2 865	2 113	1.36
3	NTT	30 583	5 993	5.10
4	NTT	30 583	5 993	5.10
5	Multiplication	4 147	4 849	0.86
6	Polysetsmall	2 353	1 685	1.40
7	Polysetsmallinplacehigh	2 865	2 113	1.36
8	NTT	30 583	5 993	5.10
9	NTT	30 583	5 993	5.10
10	Multiplication, Subtraction & Scaling	7 732	6 817	1.13
11	INTT	33 084	7 826	4.23

Table 5.6: Comparison of profiling results for Function 7: $w \leftarrow B^{-1}x$ over 10 000 iterations

5.6 Security Evaluation

In developing our implementation of the sign routine, we closely followed the C reference implementation of HAWK's sign routine from pqm4. This reference implementation is isochronous, meaning that any function involving secret data executes in constant time, with its running time remaining independent of the secret data [20].

5.6.1 Constant-Time Type System

To prove that our code is constant-time, we utilize Jasmin's constant-time type system. Its constant-time checker ensures that the implementation adheres to the principles of constant-time execution, thereby mitigating timing-based side-channel attacks. Specifically, it verifies that control flow branches, memory accesses, and variable-time instructions are independent of secret data, ensuring that no sensitive information is revealed.

The process begins by annotating the inputs of our export function, `sign_finish_inner_jazz`. The function takes four parameters: a buffer where the signature is written, a buffer containing the message, a buffer containing the private key, and an unsigned 32-bit value representing the message length.
We annotate the private key as `#secret` and all other parameters as `#public`, as the message, message length, and signature are considered public information in a signature scheme. These annotations help the constant-time type system ensure that there are no branches dependent on secret data, for example.

In certain cases, `#declassify` annotations are necessary to signal to the type system that disclosing specific data is acceptable. In our implementation, we employ a few such annotations. For example, despite the message being marked as `#public`, it is still necessary to add `#declassify` annotations when loading the message because only the pointers, not the referenced data, are annotated as `#public`.
Furthermore, we include `#declassify` annotations in three other locations:

The first annotation indicates that it is permissible to leak the result of the comparison, which checks whether the restart of the Gaussian generation for x is necessary by checking if the squared norm of x is too high.

The second annotation indicates that it is permissible to leak the result of the comparison, which checks whether one of the coefficients of $s1$ is larger than a given limit.

The third group of annotations declassifies data during the reading process from the array $s1$ within the function responsible for encoding the signature. This is necessary because the number of iterations in the loop varies based on the value read from $s1$.

These annotations are essential because the branching in all three cases depends on secret data. However, in these instances, disclosing the data is justified. This was confirmed through discussions with the authors of the HAWK scheme and the developers of the reference implementation, with the reasoning provided below also validated by them as aligned with the HAWK specification sheet [20].

In the first case, declassification is acceptable because x is no longer secret after rejection, relying on the assumption that the cryptographically secure Random Number Generator (RNG) ensures that revealing part of its output does not compromise the rest. In the second and third cases, declassification is permissible because $s1$ (the value to be encoded into the signature) is inherently public. Although the rejection rate—how often the signer repeats the process—could theoretically reveal information about the private key, the HAWK specification explicitly states that this is not a concern.

Running the type system on the function with all annotations with the command `jasmin-ct arithmetic.jazz --slice sign_finish_inner_jazz --arch arm-m4` produces no errors, confirming that the function does not leak secret data and is verified to be constant-time.

5.7 Implementation Details and Limitations

Our implementation includes two buffers that, unlike the reference implementation, are currently filled with constants rather than random bytes. Specifically, the buffer `pseed` in function `sig_gauss_jazz` and the buffer `psalt` in function `sign_finish_inner_jazz`. To achieve full equivalence with the reference implementation, this functionality will need to be added. Furthermore, our implementation only supports the case where the private key is encoded, as this is the default behavior in the reference implementation in pqm4.

6 Conclusion

In this thesis, we have successfully implemented the sign routine of the post-quantum signature scheme Hawk on a Cortex-M4 platform, focusing heavily on optimizing the core polynomial arithmetic operations. The main optimization efforts were directed toward the NTT and its inverse (INTT), which are critical components for efficient performance in lattice-based cryptography. Our optimizations resulted in significant speedups, achieving a factor of 5.1 for the NTT and 4.23 for the INTT over the reference implementation, demonstrating that our methods approach the efficiency of state-of-the-art solutions found in the literature.

Overall, these enhancements translated to an 11 % improvement in total performance compared to the reference, and when excluding SHAKE-related functions, the optimization led to a 178 % increase in speed. This work not only highlights the potential of the Cortex-M4 for handling post-quantum cryptographic tasks but also underscores the importance of efficient polynomial arithmetic in such schemes. These results provide a solid foundation for future explorations in optimizing post-quantum signature schemes on resource-constrained embedded devices, offering valuable insights for both academic research and practical implementations in secure communication systems.

6.1 Future Work

A logical next step would be to optimize the verification and key generation algorithms in Jasmin. However, in Hawk, key generation, signing, and verification are three distinct components with minimal shared code[1]. As a result, much of the code developed in this thesis cannot be directly reused for these optimizations.
Another avenue for future work could involve further optimizing the NTT and INTT on the Cortex-M4F by implementing layer merging across three layers as described in [11], which would require support for the `vmov` instruction. If Jasmin eventually supports this instruction, it could open up new opportunities for optimization.
Additionally, if Jasmin integrates a safety checker for ARM in the future, the optimized code that has been verified using Jasmin's constant-time type system could also be evaluated with the safety checker, once that feature is available.

[1]This was confirmed through personal communication with the authors of the reference implementation

Bibliography

[1] A. Abdulrahman, J.-P. Chen, Y.-J. Chen, V. Hwang, M. J. Kannwischer, and B.-Y. Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2022. doi: 10.46586/tches.v2022.i1.127-151. https://eprint.iacr.org/2021/995.pdf. (Cited on pages 17, 71, and 89)

[2] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In G. Ateniese and D. Venturi, editors, *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, Heidelberg, June 2022. doi: 10.1007/978-3-031-09234-3_42. https://eprint.iacr.org/2022/112.pdf. (Cited on page 17)

[3] A. Abdulrahman, F. Oberhansl, H. N. H. Pham, J. Philipoom, P. Schwabe, T. Stelzer, and A. Zankl. Towards ML-KEM ML-DSA on OpenTitan. Cryptology ePrint Archive, Report 2024/1192, 2024. https://eprint.iacr.org/2024/1192.pdf. (Cited on page 61)

[4] R. C. Agarwal and C. S. Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975. doi: 10.1109/PROC.1975.9791. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1451721. (Cited on page 43)

[5] M. Ajtai. Generating hard instances of lattice problems. In *28th Annual ACM Symposium on Theory of Computing*, pages 99–108. ACM Press, May 1996. doi: 10.1145/237814.237838. https://eccc.weizmann.ac.il/eccc-reports/1996/TR96-007/Paper.pdf. (Cited on page 30)

[6] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *29th Annual ACM Symposium on Theory of Computing*, pages 284–293. ACM Press, May 1997. doi: 10.1145/258533.258604. https://eccc.weizmann.ac.il/report/1996/065/. (Cited on page 30)

[7] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. Status report on the second round of the nist post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf. (Cited on page 16)

[8] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In T. Holz and S. Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343. USENIX Association, Aug. 2016. https://eprint.iacr.org/2015/1092.pdf. (Cited on pages 18, 47, and 67)

[9] E. Alkim, P. Jakubeit, and P. Schwabe. A new hope on ARM Cortex-M. Cryptology ePrint Archive, Report 2016/758, 2016. https://eprint.iacr.org/2016/758.pdf. (Cited on pages 18 and 69)

[10] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, 2020. ISSN 2569-2925. doi: 10.13154/tches.v2020.i3.336-357. https://tches.iacr.org/index.php/TCHES/article/view/8593/8160. (Cited on page 70)

[11] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde, and B.-Y. Yang. Polynomial multiplication in NTRU prime: Comparison of optimization strategies on Cortex-M4. Cryptology ePrint Archive, Report 2020/1216, 2020. https://eprint.iacr.org/2020/1216.pdf. (Cited on pages 70 and 93)

[12] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1807–1823. ACM Press, Oct. / Nov. 2017. doi: 10.1145/3133956.3134078. https://acmccs.github.io/papers/p1807-almeidaA.pdf. (Cited on pages 49 and 50)

[13] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1607–1622. ACM Press, Nov. 2019. doi: 10.1145/3319535.3363211. https://eprint.iacr.org/2019/1155.pdf. (Cited on page 49)

[14] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy*, pages 965–982. IEEE Computer Society Press, May 2020. doi: 10.1109/SP40000.2020.00028. https://arxiv.org/pdf/1904.04606. (Cited on page 49)

[15] ARM Ltd. Armv7-m architecture reference manual, 2021. https://developer.arm.com/documentation/ddi0403/latest/. (Cited on page 52)

[16] ARM Ltd. Procedure call standard for the arm architecture, 2023. https://github.com/ARM-software/abi-aa/releases/download/2023Q3/aapcs32.pdf. (Cited on page 53)

[17] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. R. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. N. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandra, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019. doi: 10.1038/s41586-019-1666-5. https://arxiv.org/pdf/1910.11333. (Cited on page 25)

[18] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, Heidelberg, Aug. 2011. doi: 10.1007/978-3-642-22792-9_5. https://www.iacr.org/archive/crypto2011/68410071/68410071.pdf. (Cited on page 49)

[19] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE Symposium on Security and Privacy*, page 353–367. IEEE Computer Society Press, May 2018. doi: 10.1109/EuroSP.2018.00032. https://eprint.iacr.org/2017/634.pdf. (Cited on pages 15 and 46)

[20] J. W. Bos, O. Bronchain, L. Ducas, S. Fehr, Y. Huang, T. Pornin, E. W. Postlethwaite, T. Prest, L. N. Pulles, and W. van Woerden. HAWK. Technical report, National Institute of Standards and Technology, 2023. https://hawk-sign.info/hawk-spec.pdf. (Cited on pages 13, 38, 39, 40, 90, and 91)

[21] L. Botros, M. J. Kannwischer, and P. Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In J. Buchmann, A. Nitaj, and T. eddine Rachidi, editors, *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, Heidelberg, July 2019. doi: 10.1007/978-3-030-23696-0_11. https://eprint.iacr.org/2019/489.pdf. (Cited on page 17)

[22] J. Brendel, C. Cremers, D. Jackson, and M. Zhao. The provable security of Ed25519: Theory and practice. In *2021 IEEE Symposium on Security and Privacy*, pages 1659–1676. IEEE Computer Society Press, May 2021. doi: 10.1109/SP40001.2021.00042. https://eprint.iacr.org/2020/823.pdf. (Cited on pages 9, 23, and 24)

[23] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. Technical report, National Institute of Standards and Technology, 2016. https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf. (Cited on page 15)

[24] J. Choi, S. Yoon, and S. C. Seo. Optimized falcon verify on cortex-m4 for post-quantum secure uav communications. *ICT Express*, 2024. doi: https://doi.org/10.1016/j.icte.2024.11.002. https://www.sciencedirect.com/science/article/pii/S2405959524001401. (Cited on page 18)

[25] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang. NTT multiplication for NTT-unfriendly rings. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. ISSN 2569-2925. doi: 10.46586/tches.v2021.i2.159-188. https://eprint.iacr.org/2020/1397.pdf. (Cited on pages 13, 17, 69, 70, 71, and 72)

[26] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965. doi: 10.2307/2003354. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf. (Cited on page 43)

[27] C. Cremers, S. Düzlü, R. Fiedler, M. Fischlin, and C. Janson. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. In *2021 IEEE Symposium on Security and Privacy*, pages 1696–1714. IEEE Computer Society Press, May 2021. doi: 10.1109/SP40001.2021.00093. https://eprint.iacr.org/2020/1525.pdf. (Cited on page 39)

[28] D. Dadush and L. Ducas. Intro to lattice algs & crypto. Mastermath, 2018. https://homepages.cwi.nl/~dadush/teaching/lattices-2018/notes/lecture-1.pdf. (Cited on pages 30 and 31)

[29] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-LWE encryption. In *2015 Design, Automation Test in Europe Conference Exhibition (2015)*, page 339–344. EDA Consortium, 2015. doi: 10.7873/DATE.2015.0378. https://eprint.iacr.org/2014/725.pdf. (Cited on page 42)

[30] L. Ducas and P. Q. Nguyen. Faster Gaussian lattice sampling using lazy floating-point arithmetic. In X. Wang and K. Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 415–432. Springer, Heidelberg, Dec. 2012. doi: 10.1007/978-3-642-34961-4_

26. https://www.iacr.org/archive/asiacrypt2012/76580410/76580410.pdf. (Cited on page 36)

[31] L. Ducas and W. P. J. van Woerden. On the lattice isomorphism problem, quadratic forms, remarkable lattices, and cryptography. In O. Dunkelman and S. Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part III*, volume 13277 of *Lecture Notes in Computer Science*, pages 643–673. Springer, Heidelberg, May / June 2022. doi: 10.1007/978-3-031-07082-2_23. https://eprint.iacr.org/2021/1332.pdf. (Cited on pages 33, 34, and 36)

[32] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal Gaussians. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, Heidelberg, Aug. 2013. doi: 10.1007/978-3-642-40041-4_3. https://eprint.iacr.org/2013/383.pdf. (Cited on page 36)

[33] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1): 238–268, 2018. ISSN 2569-2925. doi: 10.13154/tches.v2018.i1.238-268. https://eprint.iacr.org/2017/633.pdf. (Cited on pages 15 and 46)

[34] L. Ducas, E. W. Postlethwaite, L. N. Pulles, and W. P. J. van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. In S. Agrawal and D. Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 65–94. Springer, Heidelberg, Dec. 2022. doi: 10.1007/978-3-031-22972-5_3. https://eprint.iacr.org/2022/1155.pdf. (Cited on pages 15, 33, 35, 36, and 41)

[35] Formosa Crypto Team. Libjade, 2023. https://github.com/formosa-crypto/libjade. (Cited on page 49)

[36] M. Ganley. Digital signatures and their uses. *Computers & Security*, 13:385–391, 1994. doi: 10.1016/0167-4048(94)90031-0. https://www.sciencedirect.com/science/article/pii/0167404894900310. (Cited on page 21)

[37] N. Genise and D. Micciancio. Faster Gaussian sampling for trapdoor lattices with arbitrary modulus. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 174–203. Springer, Heidelberg, Apr. / May 2018. doi: 10.1007/978-3-319-78381-9_7. https://eprint.iacr.org/2017/308.pdf. (Cited on page 36)

[38] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, page 563–578. Association for Computing Machinery, 1966. doi: 10.1145/1464291.

1464352. https://dl.acm.org/doi/pdf/10.1145/1464291.1464352. (Cited on page 43)

[39] C. Gentry and M. Szydlo. Cryptanalysis of the revised NTRU signature scheme. In L. R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 299–320. Springer, Heidelberg, Apr. / May 2002. doi: 10.1007/3-540-46035-7_20. http://www.szydlo.com/ntru-revised-full02.pdf. (Cited on page 36)

[40] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In R. E. Ladner and C. Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 197–206. ACM Press, May 2008. doi: 10.1145/1374376.1374407. https://eprint.iacr.org/2007/432.pdf. (Cited on pages 33 and 36)

[41] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, Heidelberg, Sept. 2012. doi: 10.1007/978-3-642-33027-8_30. https://www.iacr.org/archive/ches2012/74280511/74280511.pdf. (Cited on page 42)

[42] D. O. C. Greconici, M. J. Kannwischer, and A. Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2021. ISSN 2569-2925. doi: 10.46586/tches.v2021.i1.1-24. https://tches.iacr.org/index.php/TCHES/article/view/8725/8325. (Cited on page 17)

[43] L. K. Grover. A fast quantum mechanical algorithm for database search. In *28th Annual ACM Symposium on Theory of Computing*, pages 212–219. ACM Press, May 1996. doi: 10.1145/237814.237866. https://arxiv.org/pdf/quant-ph/9605043. (Cited on page 25)

[44] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe. Software speed records for lattice-based signatures. In P. Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013*, pages 67–82. Springer, Heidelberg, June 2013. doi: 10.1007/978-3-642-38616-9_5. https://cryptojedi.org/papers/lattisigns-20130328.pdf. (Cited on page 69)

[45] T. Güneysu, M. Krausz, T. Oder, and J. Speith. Evaluation of lattice-based signature schemes in embedded systems. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 385–388, 2018. doi: 10.1109/ICECS.2018.8617969. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8617969. (Cited on page 17)

[46] I. Haviv and O. Regev. On the lattice isomorphism problem. In C. Chekuri, editor, *25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 391–404. ACM-SIAM, Jan. 2014. doi: 10.1137/1.9781611973402.29. https://arxiv.org/pdf/1311.0366. (Cited on pages 30 and 34)

[47] M. Hoffmann. Lecture 10 – quantum computers vs. security. Software Implementation of Cryptographic Schemes, 2022. (Cited on pages 27 and 29)

[48] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte. NTRUSIGN: Digital signatures using the NTRU lattice. In M. Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 122–140. Springer, Heidelberg, Apr. 2003. doi: 10.1007/3-540-36563-X_9. https://www.math.brown.edu/jpipher/NTRUSign_RSA.pdf. (Cited on page 35)

[49] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen. Improved plantard arithmetic for lattice-based cryptography. Cryptology ePrint Archive, Report 2022/956, 2022. https://eprint.iacr.org/2022/956. (Cited on page 48)

[50] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022. doi: 10.46586/tches.v2022.i4.614-636. https://scholars.cityu.edu.hk/files/117273207/115475213.pdf. (Cited on page 18)

[51] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2020. https://sphincs.org/data/sphincs+-round3-specification.pdf. (Cited on page 15)

[52] V. Hwang. Private communication. Unpublished thesis. (Cited on page 48)

[53] M. J. Kannwischer. Polynomial multiplication for post-quantum cryptography. PhD thesis, 2022. https://kannwischer.eu/thesis/phd-thesis-2024-02-13.pdf. (Cited on pages 13, 43, 44, 45, and 47)

[54] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.pdf. (Cited on pages 18 and 57)

[55] J. Katz and Y. Lindell. In *Introduction to Modern Cryptography, Second Edition*, pages 439–443. Chapman & Hall/CRC, 2014. https://eclass.uniwa.gr/modules/document/file.php/CSCYB105/Reading%20Material/%5BJonathan_

Katz%2C_Yehuda_Lindell%5D_Introduction_to_Mo%282nd%29.pdf. (Cited on pages 21, 22, and 23)

[56] A. Khalid, S. McCarthy, W. Liu, and M. O'Neill. Lattice-based cryptography for IoT in A quantum world: Are we ready? Cryptology ePrint Archive, Report 2019/681, 2019. https://eprint.iacr.org/2019/681.pdf. (Cited on page 17)

[57] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019. doi: 10.1109/SP.2019.00002. https://arxiv.org/pdf/1801.01203. (Cited on page 49)

[58] C. Lavor, L. R. U. Manssur, and R. Portugal. Shor's algorithm for factoring large integers. arXiv.org e-Print archive, quant-ph/0303175, 2003. http://www.arxiv.org/pdf/quant-ph/0303175. (Cited on pages 26 and 29)

[59] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982. doi: 10.1007/BF01457454. https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1129_LLL.pdf. (Cited on page 30)

[60] H. W. Lenstra Jr. and A. Silverberg. Lattices with symmetry. *Journal of Cryptology*, 30(3):760–804, July 2017. doi: 10.1007/s00145-016-9235-7. https://eprint.iacr.org/2014/1026.pdf. (Cited on page 36)

[61] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-LWE encryption on 8-bit AVR processors. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 663–682. Springer, Heidelberg, Sept. 2015. doi: 10.1007/978-3-662-48324-4_33. https://eprint.iacr.org/2015/410.pdf. (Cited on page 42)

[62] V. Lyubashevsky and G. Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019. ISSN 2569-2925. doi: 10.13154/tches.v2019.i3.180-201. https://eprint.iacr.org/2019/040.pdf. (Cited on page 70)

[63] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985. doi: 10.1090/S0025-5718-1985-0777282-X. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf. (Cited on pages 13, 46, and 47)

[64] P. Q. Nguyen and O. Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 271–288. Springer, Heidelberg, May / June 2006. doi: 10.1007/11761679_17.

https://iacr.org/archive/eurocrypt2006/40040273/40040273.pdf. (Cited on page 36)

[65] S. A. Olmos, G. Barthe, R. Gonzalez, B. Grégoire, V. Laporte, J.-C. Léchenet, T. Oliveira, and P. Schwabe. High-assurance zeroization. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):375–397, 2024. doi: 10.46586/tches.v2024.i1.375-397. https://eprint.iacr.org/2023/1713.pdf. (Cited on page 49)

[66] W. Plesken and M. Pohst. Constructing integral lattices with prescribed minimum. i. *Mathematics of Computation*, 45:209–221, 1985. doi: 10.2307/2008059. https://www.jstor.org/stable/2008059?seq=3. (Cited on page 34)

[67] W. Plesken and B. Souvignier. Computing isometries of lattices. *Journal of Symbolic Computation*, 24:327–334, 1997. doi: 10.1006/jsco.1996.0130. https://www.sciencedirect.com/science/article/pii/S0747717196901303. (Cited on page 34)

[68] J. M. Pollard. The fast fourier transform in a finite field. *Mathematics of Computation*, 25:365–374, 1971. doi: 10.1090/S0025-5718-1971-0301966-0. https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/S0025-5718-1971-0301966-0.pdf. (Cited on page 42)

[69] J. Preskill. Quantum computing and the entanglement frontier. arXiv.org e-Print archive, arXiv:1203.5813, 2012. https://arxiv.org/pdf/1203.5813. (Cited on page 25)

[70] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. https://falcon-sign.info/falcon.pdf. (Cited on pages 15 and 35)

[71] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. arXiv.org e-Print archive, arXiv:1706.06752, 2017. https://arxiv.org/pdf/1706.06752. (Cited on page 29)

[72] A. Satriawan, R. Mareta, and H. Lee. A complete beginner guide to the number theoretic transform (ntt). Cryptology ePrint Archive, Report 2024/585, 2024. https://eprint.iacr.org/2024/585.pdf. (Cited on pages 41, 42, and 43)

[73] G. Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.pdf. (Cited on pages 13 and 48)

[74] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, Nov. 1994. doi: 10.1109/SFCS.

1994.365700. https://users.cs.duke.edu/~reif/courses/randlectures/Quantum.papers/shor.factoring.pdf. (Cited on pages 15, 25, and 27)

[75] M. D. Sikirić, A. Schürmann, and F. Vallentin. Complexity and algorithms for computing voronoi cells of lattices. *Mathematics of Computation*, 78:713—731, 2009. doi: 10.1090/S0025-5718-09-02224-8. https://www.ams.org/journals/mcom/2009-78-267/S0025-5718-09-02224-8/S0025-5718-09-02224-8.pdf. (Cited on page 34)

[76] U. Skosana and M. Tame. Demonstration of shor's factoring algorithm for n = 21 on ibm quantum processors. *Scientific Reports*, 11, 2021. doi: 10.1038/s41598-021-95973-w. https://arxiv.org/pdf/2103.13855. (Cited on page 27)

[77] STMicroelectronics. Stm32f4-discovery user manual, 2020. https://www.st.com/resource/en/user_manual/um1472-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf. (Cited on page 56)

[78] M. Szydlo. Hypercubic lattice reduction and analysis of GGH and NTRU signatures. In E. Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 433–448. Springer, Heidelberg, May 2003. doi: 10.1007/3-540-39200-9_27. https://www.iacr.org/archive/eurocrypt2003/26560433/26560433.pdf. (Cited on pages 34 and 36)

A Appendix

This chapter provides a guide for setting up the software to run our optimized version of the HAWK-256 signing routine on the Cortex-M4. Thus, we provide all the files needed in a GitHub[1] repository. This repository contains the following:

- A directory `ext` that contains required files (related to Keccak) from other repositories[2]
- A `hawk256/m4` directory that can be integrated into pqm4 to run our optimized code directly, without the need for compilation or modifications.
- A Makefile for generating assembly code
- Our optimized Jasmin implementation of the HAWK-256 signing routine in `arithmetic.jazz`
- Some tables required by our implementation in `arrays.jinc`

In the following, we assume that Jasmin and pqm4 are already installed.

For installation instructions regarding Jasmin, refer to:
`https://github.com/jasmin-lang/jasmin/wiki/Installation-instructions`.
At the time of writing this thesis, the newest Jasmin version with commit hash `fbe992f33df12a3605aeffdadaeaf6675a3dcccd` works with our code.

For installation instructions regarding pqm4, refer to:
`https://github.com/mupq/pqm4`.

Once Jasmin and pqm4 are installed, we can proceed with setting up our code.
Subsequently, Listing A.1 outlines the commands required to acquire the source code and generate the assembly code for the optimized sign routine.

[1] `https://github.com/Ch3s26/hawk-cortex-m4-jasmin`
[2] `https://gitlab.com/formosa-dilithium/dilithium-arm/` (on the branch: benchmark) and `https://github.com/formosa-crypto/libjade/` (at commit: 94608b8)

```
# Obtain the source code
git clone https://github.com/Ch3s26/hawk-cortex-m4-jasmin
# Change to new directory
cd hawk-cortex-m4-jasmin
# Adjust the path to "jasminc" in the Makefile of this repository to
    match your installation of Jasmin.
# Build the code (this may take a couple of minutes)
make
```

Listing A.1: Instructions for obtaining and building the code

Once all steps from Listing A.1 have been completed, an assembly file called `arithmetic.s` will be generated. The steps needed to get our optimized version of the Hawk-256 signing routine running in pqm4 are described in Section 5.1.

To avoid the need for manual compilation or modifications to run the optimized version in pqm4, the `hawk256/m4` directory from our repository can be used as it contains our compiled file as well as needed modifications. This directory can be copied into the `crypto_sign` folder within pqm4 for direct use.