Universidade Estadual de Campinas
Instituto de Computação

Caio Teixeira

A Software Implementation of FALCON on the ARM Architecture

Uma Implementação em Software do Algoritmo FALCON na Plataforma ARM

CAMPINAS
2023

Caio Teixeira

A Software Implementation of FALCON on the ARM Architecture

Uma Implementação em Software do Algoritmo FALCON na Plataforma ARM

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Julio César López Hernández
Co-supervisor/Coorientador: Prof. Dr. Ricardo Dahab

Este exemplar corresponde à versão final da Dissertação defendida por Caio Teixeira e orientada pelo Prof. Dr. Julio César López Hernández.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

T235s

Teixeira, Caio, 1994-
A software implementation of FALCON on the ARM architecture / Caio Teixeira. – Campinas, SP : [s.n.], 2023.

Orientador: Julio César López Hernández.
Coorientador: Ricardo Dahab.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia pós-quântica. 2. Assinaturas digitais. 3. Microprocessadores ARM. I. López Hernández, Julio César, 1961-. II. Dahab, Ricardo, 1957-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Uma implementação em software do algoritmo FALCON na plataforma ARM
Palavras-chave em inglês:
Post-quantum cryptography
Digital signatures
ARM processors
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Julio César López Hernández [Orientador]
Edson Borin
Eduardo Moraes de Morais
Data de defesa: 12-05-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-9902-0866
- Currículo Lattes do autor: http://lattes.cnpq.br/1305917621544020

Universidade Estadual de Campinas
Instituto de Computação

Caio Teixeira

A Software Implementation of FALCON on the ARM Architecture

Uma Implementação em Software do Algoritmo FALCON na Plataforma ARM

Banca Examinadora:

- Prof. Dr. Julio César López Hernández
  IC/UNICAMP
- Prof. Dr. Edson Borin
  IC/UNICAMP
- Dr. Eduardo Moraes de Morais
  Lurk Lab

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 12 de maio de 2023

Dedicatória

Em memória de Antônio Caio Teixeira das Neves.

Agradecimentos

O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq), processo #155312/2019-7.

Da mesma forma, o presente trabalho é também fruto de uma parceria entre o Instituto de Computação (através da FUNCAMP – Fundação de Desenvolvimento da Unicamp) e a Samsung Eletrônica da Amazônia Ltda., realizada no contexto do projeto *Post-Quantum Cryptography*.

Agradeço à minha mãe, Gisela, minha tia, Taïsa, e meus avós, Deise e Antônio Caio, por todo carinho e incentivo à minha educação e às minhas aspirações, sempre apoiando minhas decisões e me dando todas ferramentas para que pudesse alcançar aquilo que almejo.

Agradeço a todos colegas, amigos e camaradas que fizeram parte do meu dia a dia durante o desenvolvimento desta pesquisa, especialmente em tempos de pandemia e no ano que se seguiu. Apesar de isolado, nunca estive sozinho, e hoje me vejo cercado de pessoas que me inspiram e me dão energia para lutar.

Agradeço em particular aos dois irmãos que a vida me deu, Tomás Silva e Luann Dias, por sempre estarem ali nos melhores e nos piores momentos. Também agradeço aos meus dois gatos, Theo e Hari, que adotei ainda filhotes no próprio Instituto de Computação, onde haviam sido abandonados, e que são os melhores companheiros que alguém poderia ter.

Por fim, agradeço ao corpo de funcionários do IC, especialmente a secretaria e as porteiras do instituto, por todos esclarecimentos, serviços e companhia. Agradeço aos professores Ricardo Dahab e Julio López pela orientação e por todo aprendizado que proveram nestes longos anos de trabalho, e a todos colegas de trabalho do projeto *Post-Quantum Cryptography*, ressaltando em particular Jheyne Ortiz e Décio Gazzoni Filho, com quem tive o prazer de trabalhar e aprender.

Resumo

O estudo e a criação de esquemas de assinatura digital são uma subárea da Criptografia que tem um papel importante nas comunicações digitais: eles possibilitam tanto a autenticação quanto a verificação da integridade dos dados enviados. No entanto, os esquemas criptográficos mais difundidos hoje em dia estão sob ataque, devido a um algoritmo quântico capaz de resolver seus problemas matemáticos subjacentes em tempo polinomial [43] e, com isso, minar sua segurança. Isso inclui todos os padrões clássicos de assinaturas digitais: RSA, DSA e ECDSA. Portanto, esquemas baseados em problemas matemáticos para os quais não se conhece, nem se espera encontrar, algoritmos quânticos capazes de resolvê-los em tempo polinomial tem sido propostos, criando uma subárea chamada "criptografia pós-quântica".

Um destes esquemas é o esquema de assinaturas digitais pós-quântico baseado em reticulados FALCON [22], recentemente padronizado pelo NIST [4]. FALCON é projetado para minimizar o custo de comunicação do sistema; isto é, os tamanhos das chaves públicas e das assinaturas, que são enviadas entre as partes. Mesmo assim, esquemas pós-quânticos são, em geral, expressivamente mais lentos que suas contrapartes clássicas, e FALCON não é uma exceção. Sem uma implementação cuidadosa, o custo de trocar um esquema clássico por um pós-quântico pode ser inviável para algumas aplicações práticas.

Tendo isto em vista, nesta dissertação, apresentamos uma implementação otimizada para o FALCON, tendo como alvo a arquitetura ARMv8-A. Esta arquitetura é muito difundida entre dispositivos *mobile* e *IoT*, e sua fatia de mercado entre computadores pessoais e servidores vem crescendo. Apesar de sua relevância, poucas implementações do FALCON voltadas para ARMv8-A estão presentes na literatura, enquanto arquiteturas como Intel ou ARMv8-M, o perfil voltado para microcontroladores da ARMv8, tem recebido muito mais atenção.

Começamos nosso trabalho descrevendo os algoritmos e técnicas usado pelo FALCON em detalhe, destacando os desafios de implementação e oportunidades de otimização que as seguem. Então, usamos as várias ferramentas presentes na plataforma, como instruções *Single Input, Multiple Data* (SIMD) e instruções criptográficas especializadas, e descrevemos diversas técnicas que aumentam a velocidade do esquema na arquitetura ARMv8-A.

Por fim, medimos o desempenho da nossa implementação em três plataformas diferentes, focando em seus melhores núcleos disponíveis: Cortex-A57 para placas de desenvolvimento NVIDIA® Jetson Nano™; Cortex-X2 para dispositivos *mobile* Samsung Galaxy S22; e Apple M1, o *System-on-Chip* presente nos modelos M1 da linha de laptops MacBook® da Apple. Também comparamos nossos resultados com a implementação de referência disponibilizada pelos autores do FALCON, alcançando velocidades até 79% maiores para geração de assinaturas, e até 61% maiores para verificação das mesmas.

Abstract

The study and design of digital signature schemes is a subfield of cryptography that plays an important role in digital communications: they provide means for both authentication and integrity verification of exchanged data. However, the most widespread cryptographic schemes nowadays are under attack, due to quantum algorithms capable of solving their underlying mathematical problems in polynomial time [43], therefore undermining their security. This includes all classical digital signature standards: RSA, DSA and ECDSA. Thus, schemes based on mathematical problems for which no quantum algorithm is known or expected to be able to solve efficiently have been proposed, creating a subfield called "post-quantum cryptography".

One such post-quantum scheme is the lattice-based digital signature scheme FALCON [22], recently standardized by NIST [4]. FALCON is designed to minimize the communication cost of the system; that is, the size of both public keys and signatures, which are exchanged between parties. Even so, post-quantum schemes are, in general, noticeably slower to compute than their classical counterparts, and FALCON is no exception. Without careful implementation, the cost of switching from a classical scheme to a post-quantum one may not be viable for some practical applications.

Therefore, in this thesis, we present an optimized implementation of FALCON targeting the ARMv8-A architecture. This architecture is widespread among mobile and *IoT* devices, and is rising in market share among personal computers and servers. Despite its relevance, few implementations of FALCON targeting ARMv8-A are present in the literature, while architectures such as Intel and ARMv8-M, the microcontroller-centric profile of ARMv8, have received much more attention.

We begin our work describing the algorithms and techniques used by FALCON in detail, highlighting the implementation challenges and optimization opportunities that arise. Then, we leverage many tools present in the platform, such as *Single Input, Multiple Data* (SIMD) instructions and specialized cryptographic instructions, and describe several techniques to speed up the scheme in the ARMv8-A architecture.

Finally, we benchmark our results across three different platforms, focusing on their best available core: Cortex-A57 for NVIDIA® Jetson Nano™ development boards; Cortex-X2 for Samsung Galaxy S22 mobile devices; and Apple M1, the System-on-Chip featured on M1 models of Apple's MacBook® line of laptops. We also compare our results with the published reference implementation by the authors of FALCON, achieving up to 79% higher speeds for signature generation, and up to 61% higher for their verification.

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Cryptography is one of the cornerstones of modern digital communications. Most internet applications we enjoy today would not be possible if not for a reliable way of communicating through long distances without the risk of someone stealing or changing the data: digital banking, public services, online shopping, streaming services, and even chat applications or social media. Furthermore, the recent trend of "internet of things" (IoT) devices also relies on the collected data being securely stored and communicated without intrusion, as is the case, for example, of smart medical devices.

Modern cryptography is split into two major fields: symmetric-key (or private-key) cryptography, where the communicating parties use the same previously agreed upon key to encrypt and decrypt their messages; and assymetric-key (or public-key) cryptography, where each user has a key pair of private and public keys, the former kept secret from everyone else, and the latter being distributed to their peers.

One of public-key cryptography's most important contributions was the capability of long distance communication without the need of using a secured channel beforehand to agree on a symmetric key. Even more, this public-key communication may be used to agree on a symmetric key for another, faster communication system, through which future messages will be encrypted. Another equally important development of public-key cryptography was a mean to establish trust over insecure channels, through what is called *digital signatures*.

Similarly to handmade signatures, digital signatures allow us to verify the identity of another party (authenticity), or the validity of a certain document (integrity), without the need for direct communication. Signing a document is done by using a private key to generate a solution of a computationally hard problem, which is linked to the document through the scheme in use. Then, verifying the signature is done by using a public key, and thus can be done by any party with access to it. The public and private keys are linked in such a manner that, when someone uses the public key to verify a signature, they are assured that only the holder of the private key could have generated that solution. Any other party trying to falsify the signature, or recover the private key from the public one, would need to solve a computationally hard problem that could not be solved in any reasonable amount of time without secret information.

The most common problems used to base security nowadays are the discrete logarithm problem and the factorization of large integers. Indeed, the digital signature algorithms

standardized by the North American agency NIST (*National Institute of Standards and Technology*) are: DSA, which is based on the discrete logarithm problem; RSA, which is based on the factorization problem; and ECDSA, which is based on the discrete logarithm over elliptic curves.

However, all of these schemes are currently under attack. In his seminal work in 1997, Peter Shor [43] described a *quantum algorithm* – that is, one that makes use of essential properties of a quantum computer – which is able to solve such problems in polynomial time. This means that, from the moment a powerful enough quantum computer is built, every following encrypted communication would be broken; moreover, every *past* communication would also be retroactively compromised. To deal with this threat, a new area of cryptography has emerged: *post-quantum cryptography*.

1.1 Post-Quantum Cryptography

Post-quantum cryptography, abbreviated as "PQC", is a subfield of cryptography that focuses on cryptosystems based on mathematical problems for which no quantum algorithm is likely to solve in polynomial time. The most prominent classes of such problems are: those based on algebraic structures known as lattices; problems based on error-correcting codes; problems based on solving systems of multivariate polynomial equations; and problems regarding the security of hash functions. Of these, the most prominent is lattice-based cryptography, as its derived schemes have shown the most resistance against different attack scenarios and withstood the test of time with their assumptions, while achieving comparable speed and complexity to classical algorithms. This claim will be further backed later on, when we discuss standardization of PQC schemes.

The quest for new, reliable and fast algorithms is one with many facets. Algorithms must be provably secure, which means having extensive care in elaborating mathematical proofs of their underlying structures' security and the (in)efficacy of known attacks against them. They must also be fast, that is, have low computational cost, so that they can be run repetitively without blocking communications on applications such as servers, and also be able to be run on low-end devices and microcontrollers to ensure the security of IoT data. They should also have small communication cost, to avoid clogging networks due to increased sizes of transferred encrypted data. Finally, implementations of such algorithms must take every advantage of the devices' capabilities to improve performance while making sure they are not vulnerable to side-channel attacks.

These characteristics mean that research on new algorithms is a long process, and we also need to consider that it will take time for real-world operations to adapt to the post-quantum reality. Some systems, such as OpenSSH, have already started securing themselves against the quantum threat [21], but most are still completely dependent on classical algorithms. Moreover, these changes must happen before a quantum computer of sufficient capabilities is developed. Shor's algorithm is able to factor an n-bit integer using $2n + 3$ *qubits* [8], which means for current applications of, e.g., RSA-2048, it would take 4099 *qubits* to break them. As of the time of this document's writing, the largest known quantum computer is IBM's Osprey, featuring 433 qubits, with plans to achieve

over 4000 qubits until 2025 [29].

In order to further develop the area and select PQC algorithms for applications to adopt as soon as possible, as to avoid the quantum threat, NIST has opened a process for standardizing post-quantum cryptographic algorithms, which we shall discuss in the next section.

1.2 NIST PQC Standardization Process

The NIST PQC standardization process [39] started in 2016 and is currently in its fourth round, having already selected one key-establishment mechanism and three digital signature schemes as new PQC standards. There were 82 initial candidates in the first round [2], of which 28 were lattice-based, 24 code-based, 13 multivariate-based, 4 hash-based, and 13 were based on other classes of problems. On the second round [3], out of 26 candidates approved to continue in the selection process, 12 were lattice-based, 6 code-based, 4 multivariate-based, 2 hash-based, and 2 were based on other classes of problems. Then, on the third and most recently concluded round [4], we had, among finalists and alternative candidates, a total of 15 candidates: 7 were lattice-based, 3 code-based, 2 multivariate-based, 2 hash-based, and 1 was based on isogenies of elliptic curves.

These statistics show the prominence of lattice-based cryptography. As a matter of fact, from the third to the fourth round, 4 algorithms were standardized: CRYSTALS-Kyber [13], a lattice-based key-establishment mechanism; FALCON [22] and CRYSTALS-Dilithium [16], both lattice-based digital signature schemes; and finally, SPHINCS+ [10], a hash-based digital signature scheme that was the only one chosen amongst the alternative candidates, and the only one not lattice-based. In the fourth round, only 4 KEM algorithms are still under evaluation, and none of them are lattice-based, as to avoid problems should lattices be attacked in the future, which was also one of the arguments for standardizing SPHINCS+.

CRYSTALS-Dilithium and FALCON are both lattice-based digital signature schemes, but they serve different purposes: CRYSTALS-Dilithium is more easily understood and implemented, and has very fast key generation times; FALCON, on the other hand, is designed to minimize communication cost, at the expense of increased complexity and key-generation times. While NIST recommends general applications to use CRYSTALS-Dilithium, FALCON will serve an important role to applications where communication cost is critical, such as certificate servers and network-intensive applications, as shown in Tables 1.1 and 1.2.

Scheme Variant	**Parameter size (in bytes)**	
	Public Key	**Signature**
FALCON-512	897	666
Dilithium1024x1024	1312	2420
Ratio	1.46×	3.63×

Table 1.1: Parameter size comparison between FALCON and CRYSTALS-Dilithium for 128-bit security

Scheme Variant	Parameter size (in bytes)	
	Public Key	**Signature**
FALCON-1024	1793	1280
Dilithium2048x1792	2592	4595
Ratio	1.45×	3.59×

Table 1.2: Parameter size comparison between FALCON and CRYSTALS-Dilithium for 256-bit security

1.3 Our Contributions

In this thesis, we present an optimized implementation of FALCON tailored for the ARMv8 architecture [38], which is prominent in mobile computing and has recently been used in more powerful applications, such as personal computing and cloud servers. We focused on FALCON's strengths and optimized both signature generation and verification using a number of different methods, and were able to achieve up to 79% faster signature generation and 61% faster verification over the reference implementation submitted to NIST.

We also discuss the scheme itself, highlighting its main algorithms and their intricacies, as well as the challenges of implementing them securely and optimized. Finally, we provide experimental data for our implementation's execution time improvements in three different ARM processors: Cortex-A57, which is featured in the NVIDIA® Jetson Nano™ development board; Cortex-X2, which is featured on Samsung Galaxy S22 smartphones; and the Apple M1 system-on-chip, featured on M1 versions of Apple's MacBook® line of laptops.

Related Works. While the literature regarding hardware implementations of FALCON is rich, there are few works on software implementations. Post-quantum libraries such as Open Quantum Safe's `liboqs` [44] and PQCRYPTO's `pqm4` [30] provide only the reference implementation submitted to NIST, pre-configured to run in specific environments (such as the Cortex-M4). The reference implementation [22] itself provides an optimized version of FALCON for AVX2, but no optimizations were made available for non-microcontroller ARM devices, for which the authors provide an implementation that emulates floating-point operations using integer registers and arithmetic.

On the other hand, literature on FALCON has presented many versions of the algorithm, aiming to optimize it on an algorithmic level with little impact on the parameters required for the security assumptions to be valid, and make it more robust against side-channel attacks. One such version is Mitaka [19], which changes the integer sampling method to avoid using floating-point operations, making it more parallelizable and suitable to masking techniques. ModFalcon [14] is another version and relies on Module-NTRU lattices, allowing it to provide intermediate levels of security, whereas FALCON only provides 128-bit and 256-bit security. Another recent version, due to Espitau *et al.* [20], reduces the modulus q, changes the spherical Gaussian distribution to an ellipsoidal one and applies different codification techniques, allowing it to reduce signature sizes by up to

40%. The last alternative version is Peregrine [42], which changes the Gaussian sampling to sampling from a centered binomial distribution and avoids FFT operations by replacing them with NTT with a residue number system, replacing floating-point operations altogether and allowing for masking techniques to be applied.

As for that speed up portions of the algorithm, Sun *et al.* [46] reduce the memory consumption and generation time of the FALCON Tree by exploring a symmetric structure, and implement this technique on both Intel Core i7 and ARM Cortex-M4, generating the tree up to 48% faster and saving up to 45% memory. In another work, Sun *et al.* [45] propose a different integer sampling method that uses an exponential Bernoulli sampling algorithm, speeding up signature generation by up to 14%.

1.4 Organization of this Thesis

In Section 2, we present the notation used throughout this document, as well as the mathematical background necessary for understanding the algorithms. In Section 3, we describe FALCON and its algorithms, highlighting the key elements that make FALCON a robust scheme. In Section 4, we describe the ARMv8 architecture, discuss the challenges of implementing the scheme in the C language, as well as describe our optimization techniques. In Section 5, we present our results and discuss the impact of our optimizations. Finally, in Section 6, we make our final remarks and discuss future lines of research that could further improve our implementation.

Chapter 2

Mathematical Foundation

In this section, we establish the notation used throughout this document, and enunciate the necessary mathematical definitions so that we may understand FALCON and techniques it applies.

Notations. We denote vectors as lowercase bold letters (e.g. $\mathbf{b} \in K^n$, where K is a field) and matrices as uppercase bold letters (e.g. $\mathbf{B} \in \text{Mat}_{m \times n}(K)$, where K is a field). Unless otherwise stated, we use row convention for vectors. Polynomials $f \in \mathbb{Z}[x]$ may be written in terms of its coefficients as $f(x) = \sum_{i=0}^{n-1} f_i x^i, f_i \in \mathbb{Z}$. We say that the coefficient vector of f, denoted by $\mathcal{C}(f)$, is the vector whose entries are the coefficients f_i of f, i.e., the vector $\mathcal{C}(f) = [f_0 \ f_1 \ \cdots \ f_{n-1}]$, $\mathcal{C}(f) \in \mathbb{Z}^n$. $\|\mathbf{b}\|$ denotes the Euclidean norm of a vector $\mathbf{b}$. We also use the same notation for polynomials: $\|f\|$ denotes the Euclidean norm of the coefficient vector of f. For a two-element vector of polynomials, e.g. $[f \ g] \in \mathbb{Z}[x]^2$, we take the norm $\|[f \ g]\|$ as the norm of the concatenation of the coefficient vectors $\mathcal{C}(f)$ and $\mathcal{C}(g)$. $(a\|b)$ denotes concatenation of either bitstrings or bytestrings. Finally, $a \hookleftarrow D$ denotes sampling from a random variable that follows a probability distribution D.

2.1 Lattices

First, we define our main algebraic structure, namely lattices, as well as computationally hard problems based on them and other necessary information for understanding the FALCON scheme.

Definition 1 (Lattices)**.** Given a set $\mathcal{B}$ of n linearly independent vectors

$$\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}, \mathbf{b}_i \in \mathbb{R}^m,$$

a *lattice* Λ is defined as the set of all integer linear combinations of vectors in $\mathcal{B}$. The integer n is called the *dimension* of Λ, the integer m is called the *rank* of Λ, and when $m = n$, the lattice is said to be *full-rank*. $\mathcal{B}$ is the *basis* of the lattice, and can also be described as an $m \times n$ matrix $\mathbf{B}$ containing the n vectors as columns. A lattice Λ can be described, in terms of the matrix definition of its basis $\mathbf{B}$, as the set

$$\Lambda = \{\mathbf{Bz} \ : \ \mathbf{z} \in \mathbb{Z}^n\}.$$

In this document, we assume that all lattices are full-rank, and use the matrix definition when referring to a lattice basis.

Hard Problems. There are two major computationally hard problems regarding lattices: the *Shortest Vector Problem* (SVP) and the *Closest Vector Problem* (CVP). These problems form the basis of lattice-based security, and we define their search variants as follows.

Definition 2 (Shortest Vector Problem (SVP))**.** Given a lattice Λ, find the nonzero vector $\mathbf{v} \in \Lambda$ for which $\|\mathbf{v}\| = \min_{\mathbf{x}\in\Lambda} \|\mathbf{x}\|$.

Definition 3 (Closest Vector Problem (CVP))**.** Given a lattice Λ and a point $\mathbf{c} \in \mathbb{R}^n$, find the lattice point closest to $\mathbf{c}$; that is, find a lattice point $\mathbf{v} \in \Lambda$ that minimizes $\|\mathbf{t}-\mathbf{v}\|$.

These are the exact versions of such problems, but often knowing approximate information about the lattice is enough to prove security. A decisional, approximated version of SVP, called GapSVP_γ, parameterized by an approximation factor γ, asks us to determine whether a lattice has a vector shorter than a length d, or whether it has no vectors shorter than γd. This problem is defined next.

Definition 4 (GapSVP_γ)**.** Given a lattice Λ, a length d and an approximation factor γ polynomial in n, such that $\gamma(n) \geq 1$, for $\lambda = \min_{\mathbf{x}\in\Lambda} \|\mathbf{x}\|$, output `YES` if $\lambda \leq d$ or `NO` if $\lambda > \gamma d$.

Another relevant approximate problem is the Shortest Independent Vectors Problem (SIVP_γ), that asks to find n linearly independent vectors in Λ such that their length is at most an approximation factor λ of the length of the n-th shortest vector in Λ.

Definition 5 (Shortest Independent Vectors Problem (SIVP_γ))**.** Given a lattice Λ and an approximation factor γ polynomial in n, such that $\gamma(n) \geq 1$, and letting λ be the length of the n-th shortest vector in Λ, output n linearly independent lattice vectors of length at most $\gamma(n) \cdot \lambda$.

These problems are all worst-case problems: they are only hard to solve for a subset of instances. Modern lattice-based cryptography, instead, makes use of problems whose average case can be reduced to such worst cases, a technique introduced by Ajtai in 1996 [1]. One of such problems is the *Short Integer Solution* (SIS) problem, which is an average-case problem that is reducible to the worst-case decisional GapSVP_γ. The SIS problem is parameterized by $n, m, q \in \mathbb{Z}^+$ and $\beta \in \mathbb{R}^+$, and is defined as follows.

Definition 6 (Short Integer Solution ($\text{SIS}_{n,m,q,\beta}$))**.** Given m uniformly random vectors $\mathbf{a}_i \in \mathbb{Z}_q^n$, forming the columns of a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, find a nonzero integer vector $\mathbf{z} \in \mathbb{Z}^m$ of norm $\|\mathbf{z}\| \leq \beta$ such that

$$\mathbf{A} \cdot \mathbf{z} = \sum_i \mathbf{a}_i \cdot z_i = \mathbf{0} \in \mathbb{Z}_q^n.$$

Reductions of SIS to GapSVP_γ follow a template presented by Peikert [40], which we enunciate next.

Theorem 1 ([40], Theorem 4.1.2)**.** For any $m = \text{poly}(n)$ any $\beta > 0$ and any sufficiently large $q \geq \beta \cdot \text{poly}(n)$, solving $\text{SIS}_{n,q,\beta,m}$ with non-negligible probability is at least as hard as solving the decisional approximate shortest vector problem GapSVP_γ and the approximate shortest independent vectors problem SIVP_γ (among others) on *arbitrary* n-dimensional lattices (i.e., in the worst case) with overwhelming probability, for some $\gamma = \beta \cdot \text{poly}(n)$.

This reduction is crucial for FALCON, as both the framework it is based on and itself rely on the SIS problem to prove their security in both classic and quantum cases.

NTRU Lattices. We now define the *class* of lattices FALCON's security proofs rely on. First, we define a way of mapping polynomials into matrices, and then describe how a set of polynomials can be used to define two different lattice bases for the same lattice.

Definition 7 (Anti-circulant Matrix)**.** Let $f \in \mathbb{Z}[x]/(x^n + 1)$, with coefficients f_i such that $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$. The *anti-circulant matrix* of f, denoted as $\mathcal{A}(f)$, is defined as

$$\mathcal{A}(f) = \begin{bmatrix} f_0 & f_1 & f_2 & \cdots & f_{n-1} \\ -f_{n-1} & f_0 & f_1 & \cdots & f_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -f_1 & -f_2 & -f_3 & \cdots & f_0 \end{bmatrix} = \begin{bmatrix} \mathcal{C}(f) \\ \mathcal{C}(x \cdot f) \\ \vdots \\ \mathcal{C}(x^{n-1} \cdot f) \end{bmatrix},$$

where multiplication is done modulo $x^n + 1$ and $\mathcal{C}(f)$ denotes the coefficient vector of f.

Definition 8 (NTRU Lattices)**.** Let $n \in \mathbb{Z}$ be a power of two, $q \in \mathbb{Z}^+$, and $f, g \in \mathbb{Z}[x]/(x^n + 1)$. Let $h = g \cdot f^{-1} \mod q$. The *NTRU lattice* $\Lambda_{h,q}$ associated with h and q is

$$\Lambda_{h,q} = \left\{(u, v) \in (\mathbb{Z}[x]/(x^n + 1))^2 \mid u + v \cdot h = 0 \bmod q\right\}.$$

$\Lambda_{h,q}$ is full-rank in $\mathbb{Z}^{2n}$ and is generated by the matrix

$$\mathbf{A}_{h,q} = \begin{bmatrix} -\mathcal{A}(h) & \mathbf{I}_n \\ q \cdot \mathbf{I}_n & \mathbf{0}_{n\times n} \end{bmatrix},$$

where $\mathbf{I}_n$ denotes the n-dimensional identity matrix, and $\mathbf{0}_{n\times n}$ denotes the $n \times n$ matrix of zeroes. Furthermore, if there are $F, G \in \mathbb{Z}[x]/(x^n + 1)$, such that

$$f \cdot G - g \cdot F = q \bmod (x^n + 1), \tag{2.1}$$

called the *NTRU equation*, is satisfied, then the same lattice accepts another basis, namely

$$\mathbf{B}_{f,g} = \begin{bmatrix} \mathcal{A}(g) & -\mathcal{A}(f) \\ \mathcal{A}(G) & -\mathcal{A}(F) \end{bmatrix}.$$

This class of lattices has the advantage of being easily and compactly represented by the polynomials f, g, F, G and h, making it particularly interesting when our aim is to minimize communication cost.

Finally, we describe some techniques that are both used as a part of FALCON and also important to understand the cryptanalysis of lattice-based systems.

Basis Reduction. There are infinitely many bases for any lattice; as a matter of fact, given a basis $\mathbf{B}$ and a unitary matrix $\mathbf{U}$ (i.e. $\det(\mathbf{U}) = \pm 1$), $\mathbf{B}' = \mathbf{UB}$ generates the same lattice. Informally, the *quality* of a lattice basis is measured by how *orthogonal* and *short* the basis vectors are. For example, both the basis $\mathbf{B}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and the basis $\mathbf{B}_2 = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}$ generate the same lattice ($\mathbb{Z}^2$); however, as $\mathbf{B}_1$ has shorter and more orthogonal vectors, it is considered a "better" basis. The quality of a basis is relevant when solving hard problems such as SVP; in general, algorithms that estimate solutions yield more accurate results the better the basis used as input is. Indeed, if the chosen basis consists of the n shortest linearly independent vectors of a lattice, then solving SVP is trivial, as the solution would be part of the basis itself. Therefore, algorithms that improve the quality of a given basis are of particular interest in lattice-based cryptography, and the parameters chosen for FALCON take them into account for security estimates. Next, we define one of the most fundamental of such algorithms, which serves as the foundation for some of the most used lattice reduction algorithms (such as LLL [34]), as well as an important value that helps us determine the quality of a basis.

Definition 9 (Gram-Schmidt Orthogonalization)**.** The *Gram-Schmidt Orthogonalization* process (GSO) is a matrix decomposition process that takes as input a matrix $\mathbf{B}$ and outputs two matrices, $\mathbf{L}$ and $\tilde{\mathbf{B}}$, such that $\mathbf{B} = \mathbf{L} \cdot \tilde{\mathbf{B}}$, where $\mathbf{L}$ is unit lower triangular and $\tilde{\mathbf{B}}$ spans the same space as $\mathbf{B}$. Then, if $\mathbf{B}$ is a lattice basis, $\tilde{\mathbf{B}}$ defines the same lattice. We calculate these matrices as follows.

Let $\tilde{\mathbf{b}}_1 = \mathbf{b}_1$. For every other integer k with $2 \leq i \leq n$, the vector $\tilde{\mathbf{b}}_i$ is iteratively calculated as:

$$\tilde{\mathbf{b}}_i = \mathbf{b}_i - \sum_{k=1}^{i-1} \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_k \rangle}{\|\tilde{\mathbf{b}}_k\|^2} \tilde{\mathbf{b}}_k.$$

Therefore, $\tilde{\mathbf{B}}$ has its rows defined by the vectors $\tilde{\mathbf{b}}_i$, and the lower triangular matrix $\mathbf{L}$ is defined as

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_1 \rangle}{\|\tilde{\mathbf{b}}_1\|^2} & \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_2 \rangle}{\|\tilde{\mathbf{b}}_2\|^2} & \dots & \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_{i-1} \rangle}{\|\tilde{\mathbf{b}}_{i-1}\|^2} & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_1 \rangle}{\|\tilde{\mathbf{b}}_1\|^2} & \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_2 \rangle}{\|\tilde{\mathbf{b}}_2\|^2} & \dots & \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_{i-1} \rangle}{\|\tilde{\mathbf{b}}_{i-1}\|^2} & \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_i \rangle}{\|\tilde{\mathbf{b}}_i\|^2} & \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_{i+1} \rangle}{\|\tilde{\mathbf{b}}_{i+1}\|^2} & \dots & \frac{\langle \mathbf{b}_n, \tilde{\mathbf{b}}_{n-1} \rangle}{\|\tilde{\mathbf{b}}_{n-1}\|^2} & 1 \end{bmatrix}.$$

We call $\tilde{\mathbf{B}}$ the *Gram-Schmidt matrix* of $\mathbf{B}$, and if $\mathbf{B}$ is full-rank, then its Gram-Schmidt matrix is unique.

Definition 10 (Gram-Schmidt Norm)**.** The *Gram-Schmidt Norm* of a full-rank matrix, denoted by $\|\mathbf{B}\|_{GS}$, is defined by the norm of the longest vector of its Gram-Schmidt

matrix. That is to say,

$$\|\mathbf{B}\|_{GS} = \max_{1 \le i \le n} \|\tilde{\mathbf{b}_i}\|.$$

Even though the Gram-Schmidt orthogonalization process is the most well-known basis reduction technique, other matrix decomposition techniques may also be useful. In the following, we define an important algebraic tool that lets us find the same lower-triangular matrix $\mathbf{L}$ through another decomposition method, which is used by FALCON instead.

Definition 11 (Hermitian Adjoint)**.** Let $\phi \in \mathbb{R}[x]$ be a monic polynomial with distinct roots over $\mathbb{C}$ and $a = \sum_{i=0}^{n-1} a_i x^i$ be an arbitrary element of the number field $\mathcal{Q} = \mathbb{Q}[x]/(\phi)$. The *(hermitian) adjoint* of a, denoted by a^*, is the unique element of $\mathcal{Q}$ such that, for any root ζ of ϕ, $a^*(\zeta) = \overline{a(\zeta)}$, where $\bar{x}$ denotes the complex conjugation of x over $\mathbb{C}$. For the particular case where $\phi(x) = x^n + 1$, a^* can be expressed as

$$a^* = a_0 - \sum_{i=1}^{n-1} a_i x^{n-i}.$$

For a matrix $\mathbf{B} \in \mathcal{Q}^{n \times m}$ (respectively, a vector $\mathbf{b} \in \mathcal{Q}^n$), its adjoint $\mathbf{B}^*$ (resp., $\mathbf{b}^*$) is the component-wise adjoint of the transpose of $\mathbf{B}$ (resp, $\mathbf{b}$). For example, for an arbitrary $\mathbf{B} \in \mathcal{Q}^{2\times 2}$,

$$\mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \Leftrightarrow \mathbf{B}^* = \begin{bmatrix} a^* & c^* \\ b^* & d^* \end{bmatrix}$$

If $a^* = a$ (resp., $\mathbf{b}^* = \mathbf{b}$, and $\mathbf{B}^* = \mathbf{B}$), then a (resp. $\mathbf{b}$ and $\mathbf{B}$) is called *self-adjoint*.

Proposition 1. Any positive-definite, self-adjoint matrix $\mathbf{G}$ can be uniquely decomposed as $\mathbf{G} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{L}^*$, through a method called *LDL* decomposition*. Furthermore, if $\mathbf{G}$ can be written as $\mathbf{G} = \mathbf{B} \cdot \mathbf{B}^*$, then

$$\mathbf{L} \cdot \tilde{\mathbf{B}} \text{ is the GSO of } \mathbf{B} \iff \mathbf{L} \cdot (\tilde{\mathbf{B}} \cdot \tilde{\mathbf{B}}^*) \cdot \mathbf{L}^* \text{ is the LDL}^* \text{ decomp. of } (\mathbf{B} \cdot \mathbf{B}^*).$$

As we shall discuss in Section 3, the LDL* decomposition helps us accelerate FALCON in lieu of traditional Gram-Schmidt orthogonalization.

2.2 Discrete Gaussian Sampling

Next, we define sampling from a *discrete* Gaussian distribution. Gaussian distributions are usually defined over real numbers; however, we are interested in sampling only integers according to it, which will then be used to introduce randomness and thwart attacks that might attempt to recover the secret key through linear combinations of signatures.

Definition 12 (Discrete Gaussian distribution)**.** For $c, \sigma \in \mathbb{R}$, with $\sigma > 0$, the Gaussian function $\rho_{\sigma,c}$, with standard deviation σ and centered at c, is defined as $\rho_{\sigma,c}(x) = \exp(-|x-$

$c|^2/2\sigma^2)$, and the discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over the integers is defined as

$$D_{\mathbb{Z},\sigma,c}(x) = \frac{\rho_{\sigma,c}(x)}{\sum_{z\in\mathbb{Z}}\rho_{\sigma,c}(z)}.$$

We also define the Gaussian function for vectors in $\mathbb{R}^n$. For $\sigma \in \mathbb{R}$ and $\mathbf{c} \in \mathbb{R}^n$, the Gaussian function is defined as $\rho_{\sigma,\mathbf{c}}(\mathbf{x}) = \exp(-\pi\|\mathbf{x}-\mathbf{c}\|^2/\sigma^2)$. Finally, we define the discrete Gaussian distribution over a lattice Λ as

$$D_{\Lambda,\sigma,\mathbf{c}}(\mathbf{x}) = \frac{\rho_{\sigma,\mathbf{c}}(\mathbf{x})}{\sum_{\mathbf{z}\in\Lambda}\rho_{\sigma,\mathbf{c}}(\mathbf{z})}.$$

The center c (resp. $\mathbf{c}$) may be omitted when equal to 0 (resp. $\mathbf{0}$).

2.3 FFT and NTT

Our next set of tools to be introduced is a pair of *transforms* that take our polynomials in $\mathbb{R}[x]$ or $\mathbb{Z}[x]$ into other domains, where certain operations, such as polynomial multiplications, can be done faster. While these transforms do introduce some latency by requiring additional mapping operations between domains, we may keep our polynomials as long as possible in such representations to make full use of their advantages. These transforms are the *Fast Fourier Transform* (FFT) and the *Number Theoretic Transform* (NTT), defined as follows.

Definition 13 (Fast Fourier Transform)**.** Let $\phi \in \mathbb{Q}[x]$ be a monic polynomial of degree n with distinct roots over $\mathbb{C}$, such that $\phi(x) = \prod_{k=0}^{n}(x-\zeta_k)$, $\zeta_k \in \mathbb{C}$. Let Ω_ϕ denote the set of complex roots of ϕ. The *Discrete Fourier Transform* (DFT) of a polynomial $f \in \mathbb{Q}[x]/(\phi)$, with respect to ϕ, is denoted by the set of complex numbers $\hat{f}$,

$$\hat{f} = (f(\zeta))_{\zeta\in\Omega_\phi}.$$

This transformation with respect to ϕ is a ring isomorphism $\mathbb{Q}[x]/(\phi) \cong \mathbb{Q}[\Omega_\phi]$, and therefore has an inverse. For the particular case where $\phi(x) = x^n + 1$, Ω_ϕ is the set of n complex numbers

$$\Omega_\phi = \left\{ \exp\left(\frac{i\pi(2k+1)}{n}\right) \;\middle|\; 0 \le k < n \right\}.$$

Finally, the *Fast Fourier Transform*[15] (FFT) is an algorithm that computes the DFT (or its inverse) of a polynomial in $O(n\log n)$ operations. Since this algorithm is the most common application of the Discrete Fourier Transform in cryptography, we will simply refer to both the algorithm and the transform itself as FFT, and $\hat{f}$ as FFT(f). We also define FFT for a matrix $\mathbf{B} \in (\mathbb{Q}[x]/(\phi))^{\ell\times m}$ as applying the FFT operator to every element B_{ij}.

Definition 14 (Number Theoretic Transform)**.** The *Number Theoretic Transform* (NTT) is analog to the FFT, but maps a polynomial in $\mathbb{Z}_p[x]$ to a set of evaluations in $\mathbb{Z}_p$ instead, for a prime p such that $p = 1 \bmod 2n$. In this particular case, Ω_ϕ represents the set of n integer roots ω_k of ϕ over $\mathbb{Z}_p$, and in a way similar to the FFT, a polynomial $f \in \mathbb{Z}_p[x]/(\phi)$ can be represented by the set

$$\mathrm{NTT}_p(f) = (f(\omega))_{\omega \in \Omega_\phi}.$$

This operation also has an inverse, and both can be performed in $O(n \log n)$ operations.

Finally, we note that the domain of our polynomials in FALCON is mostly $\mathbb{Q}[x]/(x^n + 1)$, where $n = 2^k$ for some $k \in \mathbb{Z}^+$. An interesting structure that arises from this choice of field, using a power-of-two exponent, is

$$\mathbb{Q} \subseteq \mathbb{Q}[x]/(x^2 + 1) \subseteq \cdots \subseteq \mathbb{Q}[x]/(x^{n/2} + 1) \subseteq \mathbb{Q}[x]/(x^n + 1).$$

This structure, called a *tower of fields*, enables us to use the following chain of isomorphisms:

$$\mathbb{Q}^n \cong (\mathbb{Q}[x]/(x^2 + 1))^{n/2} \cong \cdots \cong (\mathbb{Q}[x]/(x^{n/2} + 1))^2 \cong \mathbb{Q}[x]/(x^n + 1). \tag{2.2}$$

Going back and forth from different levels of this tower of fields structure lets us solve problems such as sampling random polynomials according to a certain distribution, or computing an extended greatest common divisor (GCD) between polynomials, in a more familiar domain such as $\mathbb{Q}$, and then lift the result back up through the isomorphism chain to find a result in $\mathbb{Q}[x]/(x^n + 1)$.

To make use of this structure, we define two operators, `split` and `merge`, to allow us to traverse this chain, as well as a particular case of the *field norm*, that gives us an alternative way of identifying polynomials with elements of subfields.

Definition 15 (Split and merge operators)**.** Let $n \in \mathbb{Z}$ be a power of two, $\phi(x) = x^n + 1$, $\phi'(x) = x^{n/2} + 1$ and $f \in \mathbb{Q}[x]/(\phi)$, written as $f(x) = \sum_{i=0}^{n-1} a_i x^i$. We can uniquely decompose f as $f(x) = f_0(x^2) + x f_1(x^2)$, with $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$; f_0 and f_1 can then be written, in terms of the coefficients of f, i.e.,

$$f_0 = \sum_{0 \le i < n/2} a_{2i} x^i \text{ and } f_1 = \sum_{0 \le i < n/2} a_{2i+1} x^i.$$

We define the `split` operator as $\texttt{split}(f) = (f_0, f_1)$, and its inverse, named the `merge` operator, as

$$\texttt{merge}(f_0, f_1) = f_0(x^2) + x f_1(x^2) \in \mathbb{Q}[x]/(\phi).$$

Definition 16 (Field Norm)**.** The field norm $\mathcal{N}_{\mathbb{L}/\mathbb{K}}$ is a map of elements of a field $\mathbb{L}$ onto a subfield $\mathbb{K}$. We define it for a particular case of interest. Let $n \in \mathbb{Z}$ be a power of two, $\mathbb{L} = \mathbb{Q}[x]/(x^n + 1)$ and $\mathbb{K} = \mathbb{Q}[x]/(x^{n/2} + 1)$. The field norm, for this case, is defined as

$$\mathcal{N}_{\mathbb{L}/\mathbb{K}}(f) = f_0^2 - x f_1^2,$$

where $(f_0, f_1) = \texttt{split}(f) \in \mathbb{K}^2$. When $\mathbb{L}$ and $\mathbb{K}$ are such as this particular case requires, we denote $\mathcal{N}_{\mathbb{L}/\mathbb{K}}(f)$ simply as $\mathcal{N}(f)$. Finally, another equivalent formulation for $\mathcal{N}_{\mathbb{L}/\mathbb{K}}(f)$ is

$$\mathcal{N}_{\mathbb{L}/\mathbb{K}}(f)(x^2) = f(x) \cdot f(-x) \bmod \phi',$$

which is more convenient when f is represented in either NTT or FFT.

Chapter 3

The FALCON Scheme

3.1 Overview

FALCON [22] is a lattice-based digital signature scheme whose main objective is to minimize communication cost, that is, the sum of both public key length and signature length. It was designed by Foque *et al.*, and submitted to the NIST PQC standardization process in 2017, with its third iteration (v1.2) being chosen as a PQC digital signature standard, along with CRYSTALS-Dilithium.

Although other schemes might have faster execution times for some algorithms, the authors predict that communication cost will have a heavier impact during the transition from classical cryptography to post-quantum cryptography, and therefore chose to focus on minimizing the length of public data.

FALCON is based on a framework called the GPV Framework, named after its authors Gentry, Peikert and Vaikuntanathan [23], that establish some core elements to achieve a hash-and-sign signature scheme whose security is based on the SIS problem.

In general, the framework may be described as follows:

- The public key is a full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ that generates a q-ary (that is, with coefficients in $\mathbb{Z}_q$) lattice Λ, and the private key is a matrix $\mathbf{B} \in \mathbb{Z}_q^{m \times m}$ that generates the lattice *orthogonal* to Λ, denoted by $\Lambda_q^\perp$. That is, for any $\mathbf{x} \in \Lambda$ and $\mathbf{y} \in \Lambda_q^\perp$, $\langle \mathbf{x}, \mathbf{y} \rangle = 0 \bmod q$. Another way of defining this is saying that the rows of $\mathbf{A}$ and $\mathbf{B}$ are pairwise orthogonal, that is, $\mathbf{B} \cdot \mathbf{A}^t = \mathbf{0}$.

- A signature for a message m is a short vector $\mathbf{s} \in \mathbb{Z}_q^m$ such that $\mathbf{s}\mathbf{A}^t = H(m)$, where $H(m)$ denotes a hash function applied to m. One may verify that a signature is valid simply by checking that $\mathbf{s}$ is short enough and that $\mathbf{s}\mathbf{A}^t = H(m)$, using the public key $\mathbf{A}$.

- Computing the signature is done by computing a preimage $\mathbf{c}_0 \in \mathbb{Z}_q^m$ that verifies $\mathbf{c}_0\mathbf{A}^t = H(m)$, which may be done through standard linear algebra, as there are no length requirements on $\mathbf{c}_0$. Then, using the public key $\mathbf{B}$, we compute a vector $\mathbf{v} \in \Lambda_q^\perp$ close to $\mathbf{c}_0$, and then derive the signature $\mathbf{s}$ as the difference $\mathbf{s} = \mathbf{c}_0 - \mathbf{v}$. Since $\mathbf{v} \in \Lambda_q^\perp$, multiplying the previous equation by $\mathbf{A}^t$, we have that $\mathbf{s}\mathbf{A}^t = \mathbf{c}_0\mathbf{A}^t - \mathbf{v}\mathbf{A}^t = \mathbf{c}_0\mathbf{A}^t - \mathbf{0} = H(m)$. The key element in the GPV framework is that $\mathbf{v}$ is not computed

deterministically, but rather uses a randomized variant of Babai's Nearest Plane algorithm [6], due to Klein [33], to sample $\mathbf{v}$ over a spherical Gaussian distribution over the lattice $\Lambda_q^\perp$.

This framework is proven to be secure in the random oracle model under the SIS assumption [23], and also in the quantum oracle model [12]. One limitation of this framework is that two different signatures for the same hash $H(m)$ cannot be published at the same time, as that would break the system's security proof [23]. Therefore, FALCON chooses to randomize the hash by prepending a "salt" $r \in \{0,1\}^k$, for some large enough integer k, sampled during the signature generation, and instead computes $H(r||m)$.

FALCON instantiates the GPV framework using NTRU lattices, as defined on Definition 8. To do so, using the same notation for the elements of NTRU lattices, the scheme's public basis is $\mathbf{A}^{1\times 2} = [1 \ h^*]$, which is equivalent to knowing just h, and the secret basis $\mathbf{B}^{2\times 2}$ is

$$\mathbf{B} = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}.$$

They are orthogonal, as required, since $\mathbf{B} \cdot \mathbf{A}^* = 0 \bmod q$. Furthermore, the signature of a message m and a "salt" r is a pair of polynomials (s_1, s_2) such that $s_1 + s_2 h = H(r||m)$. We may check this is valid by taking $\mathbf{s} = [s_1 \ s_2]$, as we'd have $\mathbf{s}\mathbf{A}^* = H(r||m)$, as required by the framework. We separate s into s_1 and s_2 because we note that s_1 may be recomputed by knowing s_2, m, r and h, as $s_1 = H(r||m) - s_2 h$ and, therefore, our signature may be simply the pair (r, s_2), saving communication cost.

Furthermore, FALCON accelerates the process of sampling $\mathbf{v} \in \Lambda_q^\perp$ by using a sampler due to Ducas and Prest [18], called "*fast Fourier nearest plane*", that uses the tower-of-fields structure and the `split` and `merge` operators from Definition 15, along with the Fast Fourier Transform, to perform sampling efficiently.

3.2 Parameters and Keys

Parameters. FALCON is fundamentally parameterized by the choice of the exponent n, with either $n = 512$, defining the "Falcon512" parameter set, which achieves NIST's Security Level I (or 128-bit security), or $n = 1024$, defining the "Falcon1024" parameter set, which achieves NIST's Security Level V (or 256-bit security). While FALCON may be instantiated with a number of different parameters, we only consider these two sets, as defined by the version standardized by NIST.

These two parameter sets achieve the values shown in Table 3.1. We also show, in Table 3.2, a few derived parameters from each parameter set that help us understand the algorithms we will enunciate next.

	Falcon512	**Falcon1024**
Security Level	I	V
Private Key Bytelength	1281	2305
Public Key Bytelength	897	1793
Signature Bytelength	666	1280

Table 3.1: Summary of FALCON's parameter sets' properties

	Falcon512	**Falcon1024**
Modulus (q)	12289	
Signature Standard Deviation (σ_{sig})	165.736617183	168.388571447
Minimum Sampler Std. Dev. (σ_{min})	1.277833697	1.298280334
Maximum Sampler Std. Dev. (σ_{max})	1.8205	

Table 3.2: Relevant parameters from each of FALCON's parameter sets

Keys. A *secret key* in FALCON is composed of two elements: a matrix $\mathbf{B} \in (\mathbb{Q}[x]/(x^n + 1))^{2\times 2}$, and a *FALCON tree* T. $\mathbf{B}$ corresponds to the NTRU polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^n + 1))$ disposed in the following form:

$$\mathbf{B} = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}.$$

However, since this matrix is composed of four polynomials, we may instead take the set (f, g, F, G) as an equivalent key element. Finally, we note that G can be recalculated from (f, g, F), as

$$G = (q + g \cdot F) \cdot f^{-1} \bmod (x^n + 1),$$

and thus may not be explicitly stored in order to save storage space and/or bandwidth.

The other element, the FALCON tree T, is a binary tree defined through the following induction:

- a FALCON tree T of height 0 is composed of a single node whose value T_{val} is $\sigma \in \mathbb{R}^+$;
- a FALCON tree T of height k has a root node, whose value T_{val} is a polynomial $\ell \in \mathbb{Q}[x]/(x^{2^k} + 1)$, and its left and right children T_{left} and T_{right} are FALCON trees of height $k - 1$.

This tree is derived from $\mathbf{B}$ so that it corresponds to the matrix $\mathbf{L}$ of its LDL* decomposition, structured in such a way that the tower of fields $\mathbb{Q} \subseteq Q[x]/(x^2 + 1) \subseteq \cdots \subseteq \mathbb{Q}[x]/(x^n + 1)$ may be used to speed up the signature's sampling process. This tree corresponds to the compact representation of $\mathbf{L}$ in the "fast Fourier LDL" algorithm of [18], albeit instantiated in the field $\mathbb{Q}[x]/(x^n + 1)$ instead of the convolution ring $\mathbb{Q}[x]/(x^n - 1)$. This tree may be preprocessed during key generation and included in the secret key, or recomputed dynamically from $\mathbf{B}$ during the signing process. Our implementation chooses the second approach, minimizing communication cost once again.

The public key corresponding to this secret key is the polynomial $h = g \cdot f^{-1} \bmod (\phi, q)$, as per the definition of NTRU lattices.

In short, FALCON's pair of secret (sk) and public (pk) keys, i.e., the tuple (sk, pk), is defined as $sk = (f, g, F)$ and $pk = h$. These keys are encoded as bytestrings, starting with a header byte, and followed by a concatenation of the polynomials' coefficients. For the public key, the header byte is `0000nnnn`, where `nnnn` corresponds to the bit (base-2) encoding of $\log(n)$, with either $n = 512$ or $n = 1024$ (depending on the chosen parameter set), and each coefficient of h is encoded as a 14-bit sequence (as $q = 12289$ for both parameter sets). As for the private key, the header byte is instead `0101nnnn`; the coefficients of f and g are encoded as 6-bit sequences for $n = 512$ and 5-bit sequences for $n = 1024$; and the coefficients of F are encoded using 8 bytes each.

3.3 Discrete Gaussian Sampling in FALCON

FALCON uses discrete Gaussian sampling techniques in both key generation and signature processes. While achieving a similar goal, the implementation of such techniques is different for each process. We now highlight the similarities and differences in Gaussian sampling in FALCON.

Both key generation and signing use discrete Gaussian sampling to sample integer polynomials in $\mathbb{Z}[x]/(x^n + 1)$. This is done by sampling each coefficient individually. However, while key generation requires only that we sample these polynomials from an n-dimensional Gaussian distribution over $\mathbb{Z}^n$ with standard deviation σ_{fg} and centered at 0 – that is, $D^n_{\mathbb{Z},\sigma_{fg},0}$ –, signing requires us to sample over the lattice Λ defined by the key pair, with standard deviation σ_{sig} centered on a point $\mathbf{c}$ – that is, $D_{\Lambda,\sigma_{sig},\mathbf{c}}$, following Definition 12.

This difference implies that, while key generation uses a single standard deviation σ_{fg} for every coefficient, signature generation uses the idea introduced in [23] and requires a different standard deviation σ_i for each i-th coefficient. Each σ_i is derived from σ_{sig} and the norm of the vector $\tilde{\mathbf{b}}_i$ of the Gram-Schmidt matrix of the basis $\mathbf{B}$ of Λ.

Despite their differences, both algorithms use the same type of probability look-up table, called a *Reverse Cumulative Distribution Table* (RCDT). Given the probability $p(x_i)$ of an integer x_i being sampled, the RCDT, indexed as $\texttt{RCDT}[i]$, is calculated as

$$\texttt{RCDT}[i] = \begin{cases} \sum_{j>i} p(x_j) & \text{if } i \text{ is positive, or} \\ \sum_{j<i} p(x_j) & \text{if } i \text{ is negative.} \end{cases} \tag{3.1}$$

Since we are dealing with a Gaussian distribution, we have that $\texttt{RCDT}[-i] = \texttt{RCDT}[i]$. One then may be tempted to discard every table entry for $i < 0$, in order to save storage space and algorithm complexity, and instead just sample a sign bit b along with the sampled integer z, and change its sign accordingly. However, applying this change skews the output probability distribution in an undesirable way, as 0 may be sampled as either $+0$ or -0, doubling the probability of it being sampled. This problem is dealt with in different ways for each sampling routine.

We now describe the integer sampling algorithm for key generation, that samples from

$D_{\mathbb{Z},\sigma_{fg},0}$. Instead of using the RCDT described above, it uses the following probability table (denoted, in this document, as KGPT, for ***K**ey**G**en **P**robability **T**able*):

$$\texttt{KGPT}[i] = \begin{cases} p(0) & \text{if } i = 0, \\ p(x >= i+1 \mid x \neq 0) & \text{otherwise.} \end{cases} \tag{3.2}$$

By using this table instead of the RCDT, we may uniformly sample k bits to determine whether we have sampled 0 and, if not, sample another k bits, along with a sign bit b to determine what integer we have sampled. This technique is described in Algorithm 3.3.1.

Algorithm 3.3.1 KeygenSampler(σ)

Require: A standard deviation σ
Ensure: An integer $z \hookleftarrow D_{\mathbb{Z},\sigma,0}$

KGPT[] $\leftarrow$ Precomputed KGPT for σ with k bits of precision and length len(KGPT)
$u \hookleftarrow \{0,1\}^k$, uniformly, as an integer
if $u <$ KGPT[0] **then**
    $z \leftarrow 0$
else
    $u \hookleftarrow \{0,1\}^k$, uniformly, as an integer
    $b \hookleftarrow \{0,1\}^1$, uniformly, as an integer
    $z \leftarrow 1$
    for i from 1 to len(KGPT) **do**
        if $u <$ KGPT[i] **then**
            $z \leftarrow z + 1$
    $z \leftarrow z \cdot (-1)^b$
return z

As long as comparisons are done in constant-time, and the algorithm reads through the whole table at every execution, this algorithm is constant-time. Building the KGPT can be done following the Tailcut Lemma and Rényi divergence analysis provided in [41], and for FALCON's key generation, k is taken as $k = 63$.

The sampler used during signature generation, however, does integer sampling in three steps: first, sample $z \in \mathbb{Z}$ from a "base integer sampler" that relies only on $\sigma_{\max}$, from the parameter set; then, sample a sign bit b and rescale the sampled value to a corresponding one for a discrete Gaussian with standard deviation σ_i; and finally, use rejection sampling [35] to adjust the output probability to the one expected for $D_{\mathbb{Z},\sigma_i,c}$.

The base sampler, described in Algorithm 3.3.2, uses the RCDT for the half-Gaussian defined by $D_{\mathbb{Z}^+,\sigma_{\max},0}$. Since we do not have negative numbers in this distribution, sampling is straightforward from the lookup table and easily implemented in constant time.

Rejection sampling is also done in constant time, using the approximation for $\exp(-x)$ described in [48], as well as using $ccs = \sigma_{\min}/\sigma' \in [0,1]$ as inputs, so that the running time is independent of σ'. This process is described in Algorithm 3.3.3.

Finally, we describe the integer sampler used for signature generation in Algorithm 3.3.4.

The parameters are chosen so that the Rényi divergence between the distribution of outputs of BaseSampler and $D_{\mathbb{Z}^+,\sigma_{\max},0}$ are acceptable from the arguments of [41], and

Algorithm 3.3.2 BaseSampler()

Require: -

Ensure: An integer $z \hookleftarrow D_{\mathbb{Z},\sigma_{\max},0}$

$\texttt{RCDT}[] \leftarrow$ Precomputed RCDT for $\sigma_{\max}$ with k bits of precision and length $len(\texttt{RCDT})$
$u \hookleftarrow \{0,1\}^k$, uniformly, as an integer
$z \leftarrow 0$
for i from 0 to $len(\texttt{RCDT})$ **do**
    if $u < \texttt{RCDT}[i]$ **then**
        $z \leftarrow z + 1$
return z

Algorithm 3.3.3 SampleRejection()

Require: Floating-point values $x, css \geq 0$

Ensure: A single bit b, equal to 1 with probability $\approx ccs \cdot \exp(-x)$

$s \leftarrow \lfloor x/\ln(2) \rfloor$
$r \leftarrow x - s \cdot \ln(2)$ ▷ $x = 2^s \cdot r$, with $r \in [0, \ln(2))$ and $s \in \mathbb{Z}^+$
$s \leftarrow \min(s, 63)$ ▷ Saturate s to 63 to avoid invalid operations
$z \leftarrow (2^{64-s} \cdot ccs \cdot \exp(-r)) \gg s$ ▷ Approximated using [48]
$i \leftarrow 64$
repeat
    $i \leftarrow i - 8$
    $j \hookleftarrow \{0,1\}^8$, uniformly, as an integer
    $w \leftarrow j - ((z \gg i)$ `& 0xFF`$)$
until $(w \neq 0)$ or $(i \leq 0)$
if $w < 0$ **then**
    $b \leftarrow 1$
else
    $b \leftarrow 0$
return b

Algorithm 3.3.4 SignSampler(μ, σ_i)

Require: Floating-point values $\mu, \sigma_i \in \mathbb{R}$ such that $\sigma_i \in [\sigma_{\min}, \sigma_{\max}]$

Ensure: An integer $z \in \mathbb{Z}$ sampled approximately close enough to $D_{\mathbb{Z},\mu,\sigma_i}$

$r \leftarrow \mu - \lfloor \mu \rfloor$ ▷ $r \in [0, 1)$
$ccs \leftarrow \sigma_{\min}/\sigma_i$ ▷ This makes rejection sampling time-independent of σ_i
repeat
    $z_0 \leftarrow$ `BaseSampler`()
    $b \leftarrow \{0,1\}^1$, uniformly
    $z \leftarrow b + (2 \cdot b - 1)z_0$
    $x \leftarrow \frac{(z-r)^2}{2\sigma_i^2} - \frac{z_0^2}{2\sigma_{\max}^2}$
    if `SampleRejection`$(x, ccs) = 1$ **then**
        return $z + \lfloor \mu \rfloor$
until forever

the rejection sampling statistically approximates the distribution to $D_{\mathbb{Z},\sigma_i,\mu}$. For this case, the precision k for the RCDT is taken as $k = 72$. This sampler is then used for the "Fast Fourier sampler" to sample in the lattice Λ according to $D_{\Lambda,\sigma_{sig},\mathbf{c}}$, as required. This procedure will be described later, in Algorithm 3.5.1.

3.4 Key Generation

Generating a new key pair is done in two separate steps: solving the NTRU equation (Equation 2.1) and calculating the FALCON tree. As previously stated, our implementation does not compute the FALCON tree at key generation but, rather, during signing; however, we follow the original `Keygen` algorithm for our description, as the FALCON tree computation step may be separated into its own module inside the implementation without losing correctness. We present the outline of the algorithm in Algorithm 3.4.1, and discuss the subroutines that accomplish the aforementioned, `NTRUGen` and `ffLDL`*, respectively, steps later on.

Algorithm 3.4.1 Keygen(n, q)

Require: A power-of-two $n \in \mathbb{Z}$, a modulus $q \in \mathbb{Z}$.
Ensure: A key pair (sk, pk).

$\phi(x) \leftarrow x^n + 1$
$f, g, F, G \leftarrow \texttt{NTRUGen}(\phi, q)$ ▷ $f, g, F, G \in \mathbb{Z}[x]/(\phi)$
$\mathbf{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$ ▷ $\mathbf{B} \in (\mathbb{Z}[x]/(\phi))^{2\times 2}$
$\mathbf{G} \leftarrow \mathrm{FFT}(\mathbf{B}) \cdot \mathrm{FFT}(\mathbf{B})^*$ ▷ $\mathbf{G} \in \mathrm{FFT}(\mathbb{Q}[x]/(\phi))^{2\times 2}$
$T \leftarrow \texttt{ffLDL}^*(\mathbf{G})$ ▷ T is a FALCON tree with $\|\tilde{\mathbf{b}}_i\|^2$ as its leaves
for each leaf ℓ_i of T **do**
    $\ell_i \leftarrow \sigma_{sig}/\sqrt{\ell_i}$ ▷ Each ℓ_i is now the trapdoor sampler's std. deviation σ_i
$h \leftarrow g \cdot f^{-1} \mod q$ ▷ $h \in \mathbb{Z}_q[x]/(\phi)$
$sk \leftarrow (\mathbf{B}, T)$ ▷ Encoded as discussed in Section 3.2
$pk \leftarrow h$
return (sk, pk)

3.4.1 Solving the NTRU equation

Generating the secret key polynomials requires sampling $f, g \in \mathbb{Z}[x]/(\phi)$, with $\phi(x) = x^n + 1$, from an n-dimensional discrete Gaussian distribution, making sure that f is invertible modulo q and that they are short enough to guarantee security, and then computing $F, G \in \mathbb{Z}[x]/(\phi)$ that solve the NTRU equation.

Sampling is done by using Algorithm 3.3.1 to sample n coefficients for f and g. After sampling, we may easily check if f is invertible modulo q by looking at the coefficients of $\mathrm{NTT}_q(f)$: if $\mathrm{NTT}_q(f)$ contains a zero coefficient, we will not be able to find any polynomial f' for which $\mathrm{NTT}_q(f) \cdot \mathrm{NTT}_q(f')$ contains only ones, which is the case when $f \cdot f' = 1 \bmod (\phi, q)$. Furthermore, this is the only case where such a polynomial cannot

be found, as q is prime and thus $\mathbb{Z}_q^+$ is a field. Therefore, checking if $\text{NTT}_q(f)$ has a zero coefficient is sufficient to check for invertibility modulo q.

The last check with regard to f, g is that $\|\mathbf{B}\|_{GS}$ is short enough. This can be done by checking that

$$\max\left\{\|[g,-f]\|, \left\|\left[\frac{qf^*}{ff^*+gg^*}, \frac{qg^*}{ff^*+gg^*}\right]\right\|\right\} > 1.17\sqrt{q},$$

as per Lemmas 2 and 3 of [17]. Then, we proceed with finding F and G through the `NTRUSolve` subroutine, described in detail later on, and return $(f, g, F, G$. Algorithm 3.4.2 formalizes this process.

Algorithm 3.4.2 NTRUGen(ϕ, q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree n, a modulus $q \in \mathbb{Z}^+$.
Ensure: Polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ satisfying the NTRU equation.

$\sigma_{fg} \leftarrow 1.17\sqrt{q/2n}$ ▷ Chosen so that $\mathbb{E}[\|(f,g)\|] = 1.17\sqrt{q}$
for i from 0 to $n-1$ **do**
  $f_i \hookleftarrow D_{\mathbb{Z},\sigma_{fg},0}$ ▷ Using Algorithm 3.3.1
  $g_i \hookleftarrow D_{\mathbb{Z},\sigma_{fg},0}$
$(f,g) \leftarrow \left(\sum_{i=0}^{n-1} f_i x^i, \sum_{i=0}^{n-1} g_i x^i\right)$
if $\text{NTT}_q(f)$ contains a zero coefficient **then** ▷ Check for invertibility modulo q
  restart
if either $\|[g,-f]\| > 1.17\sqrt{q}$ or $\left\|\left[\frac{qf^*}{ff^*+gg^*}, \frac{qg^*}{ff^*+gg^*}\right]\right\| > 1.17\sqrt{q}$ **then**
  restart ▷ Check that $\|\mathbf{B}\|_{GS}$ is short
$(F,G) \leftarrow \texttt{NTRUSolve}_{n,q}(f,g)$ ▷ Calculate F, G that solve $fG - gF = q \bmod \phi$
if $(F,G) = \perp$ **then**
  restart
return (f, g, F, G)

Solving the NTRU equation requires us to find $F, G \in \mathbb{Z}[x]/(\phi)$ such that $fG - gF = q \bmod \phi$. We can look at this problem as solving the extended GCD for $f, g \bmod \phi$: if we find polynomials $u, v \in \mathbb{Z}[x]/(\phi)$ such that $uf - vg = 1 \bmod \phi$, then we may trivially assign $(F, G) = (-vq, uq)$. Solving the GCD itself is less trivial, given that we are working with polynomials modulo ϕ; however, the process may be made more efficient by exploring the tower of fields structure (Equation 2.2) and the field norm, as defined in Definition 16, to map f and g from $\mathbb{Z}[x]/(\phi)$ down to $\mathbb{Z}$, where the extended GCD can be easily solved. The solution is then lifted back up recursively, using the properties of the field norm, until we recover $F, G \in \mathbb{Z}[x]/(\phi)$.

Note, however, that the field norm operations do not perform reduction modulo q, and therefore lead to small polynomials with large coefficients (up to thousands of bits per coefficient in the deepest recursion level), which must be dealt with using arbitrary-size integer arithmetic in the implementation. Furthermore, at every recursion level, we control the coefficient size when mapping back to higher rings by performing a *reduction* step. This reduction can be seen as a linear operation in the NTRU lattice basis $\mathbf{B}_{f,g}$,

as defined in Definition 8; by the properties of matrices and lattice bases, by finding a polynomial $k \in \mathbb{Z}[x]/(\phi)$, we may perform $F = F - kf$ and $G = G - kf$ and still generate the same lattice. A suitable k may be found by calculating

$$k = \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil .$$

However, due to coefficient sizes being way larger than the average IEEE-754 floating point can represent, this process may be done iteratively by taking only the most representative bits of (f, g, F, G) and performing reductions until $k = 0$. This is described in Algorithm 3.4.3.

Algorithm 3.4.3 NTRUSolve$_{n,q}(f, g)$

Require: $f, g \in \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two.
Ensure: Polynomials $F, G \in \mathbb{Z}[x]/(x^n + 1)$ satisfying the NTRU equation.

if $n = 1$ **then**
  $(u, v, d) \leftarrow \texttt{xgcd}(f, g)$ ▷ $\texttt{xgcd}(f, g)$ finds $u, v, d \in \mathbb{Z}$ that solve $uf + vg = d$
  if $d \neq 1$ **then**
    return $\perp$
  else
    $(F, G) \leftarrow (-vq, uq)$
    return (F, G)
else
  $f' \leftarrow \mathcal{N}(f)$ ▷ $\mathcal{N}(f)$ is the field norm, as per Definition 16,
  $g' \leftarrow \mathcal{N}(g)$ ▷ and thus $f', g' \in \mathbb{Z}[x]/(x^{n/2} + 1)$
  $(F', G') \leftarrow \texttt{NTRUSolve}_{n/2,q}(f', g')$ ▷ $F', G' \in \mathbb{Z}[x]/(x^{n/2} + 1)$
  $F \leftarrow F'(x^2)g(-x)$
  $G \leftarrow G'(x^2)f(-x)$ ▷ $F, G \in \mathbb{Z}[x]/(x^n + 1)$
  repeat ▷ Reduction loop
    $k \leftarrow \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil$ ▷ $\frac{Ff^* + Gg^*}{ff^* + gg^*} \in \mathbb{Q}[x]/(x^n + 1)$, $k \in \mathbb{Z}[x]/(x^n + 1)$
    $F \leftarrow F - kf$
    $G \leftarrow G - kg$
  until $k = 0$
return (F, G)

3.4.2 Generating the FALCON tree

The second part of the secret key is a FALCON tree, generated from the lattice basis $\mathbf{B}$, that is used to perform sampling during the signature process. To create the FALCON tree from a basis $\mathbf{B}$, we decompose the matrix using the LDL* matrix decomposition method. As stated in Proposition 1, given a matrix $\mathbf{G}$, this method finds a triple of matrices $\mathbf{L}, \mathbf{D}, \mathbf{L}^*$ such that $\mathbf{G} = \mathbf{LDL}^*$, with $\mathbf{L}$ being a lower-triangular matrix with ones on its principal diagonal, and $\mathbf{D}$ a diagonal matrix. By performing this procedure in the FFT domain, and given that $\mathbf{G} \in \text{FFT}(\mathbb{Q}[x]/(\phi))^{2\times 2}$ (which matches our lattice basis'

dimensions), the resulting $\mathbf{L}, \mathbf{D}, \mathbf{L}^*$ matrices will be of the form:

$$\begin{bmatrix} G_{00} & G_{01} \\ G_{10} & G_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{10} & 1 \end{bmatrix} \cdot \begin{bmatrix} D_{00} & 0 \\ 0 & D_{11} \end{bmatrix} \cdot \begin{bmatrix} 1 & L_{10}^* \\ 0 & 1 \end{bmatrix}$$

The procedure to generate the FALCON tree is then as follows: perform LDL* reduction on $\mathbf{G}$, and store the polynomial L_{10} at the root of the current subtree. Then, split D_{00} and D_{11} into four new polynomials lying in the ring below using the `split` operator, and create new matrices $\mathbf{G}_0$ and $\mathbf{G}_1$ using such polynomials (as described in the algorithm below). Finally, recursively use this method with $\mathbf{G}_0$ and $\mathbf{G}_1$ to compute the left and right children of the root node, respectively. In the case $n = 2$, the values of the left and right leaves are assigned as D_{00} and D_{11}, respectively. This is presented in Algorithm 3.4.4.

Algorithm 3.4.4 ffLDL$^*(\mathbf{G})$

Require: A full-rank, self-adjoint matrix $\mathbf{G} = (G_{ij}) \in \text{FFT}(\mathbb{Q}[x]/(x^n + 1))^{2\times 2}$
Ensure: A FALCON tree T

$D_{00} \leftarrow G_{00}$
$L_{10} \leftarrow G_{10}/G_{00}$
$D_{11} \leftarrow G_{11} - L_{10} \cdot L_{10}^* \cdot G_{00}$
$T_{val} \leftarrow L_{10}$
if $n = 2$ **then**
    $T_{left} \leftarrow D_{00}$
    $T_{right} \leftarrow D_{11}$
else
    $d_{00}, d_{01} \leftarrow \texttt{splitfft}(D_{00})$ ▷ $d_{ij} \in \text{FFT}(\mathbb{Q}[x]/(x^{n/2} + 1))$
    $d_{10}, d_{11} \leftarrow \texttt{splitfft}(D_{11})$
    $(\mathbf{G}_0, \mathbf{G}_1) \leftarrow \left(\begin{bmatrix} d_{00} & d_{01} \\ d_{01}^* & d_{00} \end{bmatrix}, \begin{bmatrix} d_{10} & d_{11} \\ d_{11}^* & d_{10} \end{bmatrix} \right)$
    $T_{left} \leftarrow \texttt{ffLDL}^*(\mathbf{G}_0)$
    $T_{right} \leftarrow \texttt{ffLDL}^*(\mathbf{G}_1)$
return T

Both `split` and `merge` operations are calculated inside the FFT domain, following Algorithms 3.4.5 and 3.4.6.

Algorithm 3.4.5 splitfft(FFT(f))

Require: FFT$(f) = f(\zeta)$ for $f \in \mathbb{Q}[x]/(\phi)$ and $\zeta \in \Omega_\phi$
Ensure: FFT$(f_0) = f_0(\zeta')$ and FFT$(f_1) = f_1(\zeta')$, where $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$ and $\zeta' \in \Omega_{\phi'}$

for ζ in Ω_ϕ **do**
    $\zeta' \leftarrow \zeta^2$
    $f_0(\zeta') \leftarrow \frac{1}{2}\left(f(\zeta) + f(-\zeta)\right)$
    $f_1(\zeta') \leftarrow \frac{1}{2\zeta}\left(f(\zeta) - f(-\zeta)\right)$
return $(\text{FFT}(f_0), \text{FFT}(f_1))$

Algorithm 3.4.6 mergefft(FFT(f))

Require: FFT(f_0) $= f_0(\zeta')$ and FFT(f_1) $= f_1(\zeta')$ for $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$ and $\zeta' \in \Omega_{\phi'}$
Ensure: FFT(f) $= f(\zeta)$, where $f \in \mathbb{Q}[x]/(\phi)$ and $\zeta \in \Omega_\phi$

for ζ in Ω_ϕ **do**
    $\zeta' \leftarrow \zeta^2$
    $f(\zeta) \leftarrow f_0(\zeta') + \zeta f_1(\zeta')$
return FFT(f)

3.5 Signature Generation

Signing a message in FALCON is done in three different steps: generating a random salt, hashing it along with the message to a point $c \in \mathbb{Z}_q[x]/(\phi)$; sampling a lattice point $\mathbf{zB} \in (\mathbb{Z}[x]/(\phi))^2$ near the point $\mathbf{tB} \in (\mathbb{Z}[x]/(\phi))^2$, where $\mathbf{t} = (c, 0)$, calculating the short vector $\mathbf{s} = \mathbf{tB} - \mathbf{zB}$; and compressing this vector as a bytestring following a specified format, as outlined by Algorithm 3.5.1. We detail the `HashToPoint`, `ffSampling` and `Compress` subroutines later on.

Algorithm 3.5.1 Sign($\mathtt{m}$, sk, β)

Require: A message $\mathtt{m}$, a secret key $sk = (\mathbf{B}, T)$ and a norm bound β
Ensure: A signature sig of $\mathtt{m}$

$r \hookleftarrow \{0,1\}^{320}$, uniformly
$c \leftarrow \mathtt{HashToPoint}(r||\mathtt{m}, q, n)$
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\,\mathrm{FFT}(c) \cdot \mathrm{FFT}(F)\,,\ \frac{1}{q}\,\mathrm{FFT}(c) \cdot \mathrm{FFT}(f)\right)$ ▷ $\mathbf{t} = \mathrm{FFT}((c, 0) \cdot \mathbf{B})$
repeat
    repeat
        $\mathbf{z} \leftarrow \mathtt{ffSampling}_n(\mathbf{t}, T)$
        $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z}) \cdot \mathrm{FFT}(\mathbf{B})$ ▷ $\mathbf{s} = \mathrm{FFT}((s_1, s_2)) \in \mathrm{FFT}(\mathbb{Z}[x]/(\phi))^2$
    until $\|\mathbf{s}\|^2 \leq \beta$
    $(s_1, s_2) \leftarrow \mathrm{FFT}^{-1}(\mathbf{s})$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
    $s \leftarrow \mathtt{Compress}(s_2, 8 \cdot sbytelen - 328)$
until $s \neq \perp$
return $sig = (r, s)$

We hash a message to a polynomial by using SHAKE-256 to absorb the message, and then output as many pseudo-random bits as necessary to derive n coefficients modulo q, as described in Algorithm 3.5.2. We extract 16 bits (2 bytes) at a time from SHAKE-256, and thus, to preserve the output distribution after modular reduction, only considering valid elements that are below $\lfloor 2^{16}/q \rfloor = 5$ times q.

Due to the fact that we may sample invalid values, making this process constant-time is less trivial. The technique we employed in our implementation, as well as FALCON's authors did in their constant-time version of the algorithm, is *oversampling*: we continue sampling beyond n coefficients, keeping a temporary buffer for the new samples, so that the probability of not having n valid coefficients is below 2^{-256}. Then, we check, in constant-time, whether c has invalid coefficients, and swap them in constant time with the temporary buffer coefficients.

Algorithm 3.5.2 HashToPoint($\mathtt{m}, q, n$)

Require: A message string $\mathtt{m}$, a modulus $q \leq 2^{16}$ and a degree $n \in \mathbb{Z}^+$.
Ensure: A polynomial $c \in \mathbb{Z}_q[x]$ correlated to $\mathtt{m}$

$state \leftarrow \mathtt{SHAKE256\text{-}Init}()$
$\mathtt{SHAKE256\text{-}Absorb}(state, \mathtt{m})$
$\mathtt{SHAKE256\text{-}Finalize}(state)$
$i \leftarrow 0$
while $i < n$ **do**
    $t \leftarrow \mathtt{SHAKE256\text{-}Squeeze}(state, 2)$
    if $t < 5q$ **then**
        $c_i \leftarrow t \bmod q$
        $i \leftarrow i + 1$
$c \leftarrow \sum_{i=0}^{n-1} c_i x^i$
return c

Sampling a lattice point near the point $\mathbf{tB}$ is done according to Algorithm 3.5.3, using the Fast Fourier Nearest Plane algorithm described in [18], again instantiated in $\mathbb{Z}[x]/(x^n + 1)$. This algorithm is a variation of the randomized Babai's Nearest Plane algorithm [6], and makes use of both FFT and `splitfft`/`mergefft` to sample an integer polynomial vector $\mathbf{z} \in (\mathbb{Z}[x]/(x^n+1))^2$ so that the lattice point $\mathbf{zB}$ is close enough to $\mathbf{tB}$. Sampling each coefficient according to σ_i is done using Algorithm 3.3.4.

Algorithm 3.5.3 ffSampling$_n(\mathbf{t}, T)$

Require: $\mathbf{t} = (t_0, t_1) \in \mathrm{FFT}(\mathbb{Z}[x]/(x^n + 1))^2$, a FALCON tree T
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathrm{FFT}(\mathbb{Z}[x]/(x^n + 1))^2$

if n = 1 **then**
    $\sigma' \leftarrow T_{val}$ ▷ T_{val} stores each σ_i
    $z_0 \hookleftarrow D_{\mathbb{Z},\sigma',t_0}$ ▷ Using Algorithm 3.3.4
    $z_1 \hookleftarrow D_{\mathbb{Z},\sigma',t_1}$ ▷ No need to apply FFT^{-1} since $\mathrm{FFT}(a) = \mathrm{FFT}^{-1}(a)$ when $n = 1$
else
    $\mathbf{t}_1 \leftarrow \mathtt{splitfft}(t_1)$ ▷ $\mathbf{t}_0, \mathbf{t}_1 \in \mathrm{FFT}((\mathbb{Z}[x]/(x^{n/2} + 1))^2$
    $\mathbf{z}_1 \leftarrow \mathtt{ffSampling}_{n/2}(\mathbf{t}_1, T_{right})$ ▷ $\mathbf{z}_0, \mathbf{z}_1 \in \mathrm{FFT}((\mathbb{Z}[x]/(x^{n/2} + 1))^2$
    $z_1 \leftarrow \mathtt{mergefft}(\mathbf{z}_1)$
    $t'_0 \leftarrow t_0 + (t_1 - z_1) \cdot T_{val}$
    $\mathbf{t}_0 \leftarrow \mathtt{splitfft}(t'_0)$
    $\mathbf{z}_0 \leftarrow \mathtt{ffSampling}_{n/2}(\mathbf{t}_0, T_{left})$
    $z_0 \leftarrow \mathtt{mergefft}(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$

We note that s_1 may be computed from the public key $pk = h$, the hashed polynomial c and s_2 by calculating $s_1 = c - s_2 h \bmod q$. Since this process only uses s_2 and public information, it suffices to send s_2 as the signature polynomial, along with the "salt" r.

Finally, we compress the signature to the format specified in FALCON's documentation through Algorithm 3.5.4. This format encompasses a header byte structured as `0cc1nnnn`, where `nnnn` corresponds to $\log_2 n$ in bit format, and `cc` is either 01 or 10, de-

termining if the signature vector bytestring is compressed or not. In our implementation, we opted to always compress the signature, and thus, the header byte is always `0011nnnn`. The 40-byte salt string r is then concatenated to the header as generated, without any processing.

As for the polynomial $s_2 \in \mathbb{Z}[x]/(\phi)$, instead of using a fixed size bitstring for each coefficient s_i, we follow FALCON's compression algorithm, that encodes them as follows:

1. the first bit corresponds to s_i's sign bit;
2. the next 7 bits correspond to the 7 least significant bits of s_i, from most to least significant;
3. lastly, we encode the remaining, most significant bits of s_i using unary encoding. That is, if the remaining bits equal a certain $k = \lfloor |s_i|/2^7 \rfloor$, then its encoding is equal to k zeroes, followed by a 1. We denote this string as 0^k1.

For example, the coefficient $459_{10} = 000111001011_2$ would be encoded as the bitstring "010010110001". Finally, the complete bytestring is appended with zeroes until the maximum length *slen* is achieved.

Algorithm 3.5.4 Compress(s, *slen*)

Require: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]/(x^n + 1)$, a string bitlength *slen*
Ensure: A compressed representation `str` of s of bitlength *slen*, or $\perp$

`str` $\leftarrow \{\}$
for i from 0 to $n - 1$ **do**
    `str` $\leftarrow$ (`str`$||b$), where $b = 0$ if $s_i \geq 0$, or $b = 1$ if $s_i < 0$
    `str` $\leftarrow$ (`str`$||b_6 b_5 \dots b_0$), where $b_j = (|s_i| \gg j)$ `& 0x1`
    $k \leftarrow |s_i| \gg 7$
    `str` $\leftarrow$ (`str`$||0^k1$)
if |`str`| $>$ *slen* **then**
    `str` $\leftarrow \perp$
else
    `str` $\leftarrow$ (`str`$||0^{slen-|\texttt{str}|}$)
return `str`

3.6 Signature Verification

Signature verification is relatively simpler than the other two algorithms. Given a signature in the form (r, s), we only need to decompress s back into s_2, use r and the message m to calculate the corresponding point c, and use them along with the public key $pk = h$ to recompute s_2. Finally, we make sure that $\|(s_1, s_2)\|^2$ is within the expected bound β, and, if so, we accept the signature. Decompression is described in Algorithm 3.6.1, and verification in Algorithm 3.6.2.

Security is guaranteed by the fact that s_1 can be recomputed from s_2 and the public key h, and that $\mathbf{s} = (s_1, s_2)$ is short enough. This implies that, for $\mathbf{t} = (c, 0)$, if we

translate our lattice Λ by $\mathbf{t}$ so that $\mathbf{t}$ corresponds to the origin – say, $\Lambda' = (\Lambda - \mathbf{t})$ –, then $\mathbf{s}$ is a "hard enough" solution to SVP for Λ', meaning it must have been computed using the corresponding secret key.

Algorithm 3.6.1 Decompress($\texttt{str}, slen$)

Require: A bitstring `str` of bitlength $slen$
Ensure: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]/(x^n + 1)$, or $\perp$

if $|\texttt{str}| \neq slen$ **then**
    return $\perp$
for i from 0 to $n - 1$ **do**
    $\ell \leftarrow \sum_{j=0}^{6} 2^{6-j} \cdot \texttt{str}[1 + j]$ ▷ Decode the lowest bits of $|s_i|$
    $k \leftarrow 0$
    while $\texttt{str}[8 + k] = 0$ **do** ▷ Decode the highest bits of $|s_i|$, in unary representation
        $k \leftarrow k + 1$
    $s_i \leftarrow (-1)^{\texttt{str}[0]} \cdot (\ell + 2^7 k)$ ▷ Compute $|s_i|$ and apply the sign bit
    if $s_i = 0$ and $\texttt{str}[0] = 1$ **then** ▷ Avoid "-0" encoding
        return $\perp$
    $\texttt{str} \leftarrow \texttt{str}[9 + k ... slen - 1]$ ▷ Update `str`, consuming the bits we have already decoded
if $\texttt{str} \neq 0^{slen - |\texttt{str}|}$ **then** ▷ Check for trailing 0 bits
    return $\perp$
return $s = \sum_{i=0}^{n-1} s_i x^i$

Algorithm 3.6.2 Verify($\texttt{m}, sig, pk, \beta$)

Require: A message m, a signature $sig = (r, s)$, a secret key $pk = h \in \mathbb{Z}_q[x]/(\phi)$ and a norm bound β
Ensure: `Accept` or `Reject`

$c \leftarrow \texttt{HashToPoint}(r||m, q, n)$
$s_2 \leftarrow \texttt{Decompress}(s, 8 \cdot sbytelen - 328)$
if $s_2 = \perp$ **then**
    return `Reject`
$s_1 \leftarrow c - s_2 h \mod q$
if $\|[s_1,\ s_2]\|^2 \leq \beta$ **then**
    return `Accept`
else
    return `Reject`

Chapter 4

Implementation and Optimization

4.1 The ARMv8 Platform

The ARM architecture is a reduced instruction set computer (RISC) architecture that targets a wide range of applications. Its current version is ARMv9, officially launched on March 30, 2021 [5]. However, due to its recent release, it is not as widespread as its predecessor, ARMv8, which is the focus of this research. While ARMv9 introduces new technologies for signal processing and machine learning, such as SVE (*Scalable Vector Extension*), most of its general-purpose computing capabilities can be found in ARMv8, with complete backwards compatibility. Therefore, optimizations that target ARMv8 also improve performance on ARMv9, impacting devices now and in the near future.

The ARMv8 architecture has three different profiles: the Application profile (or A-Profile), targeting most computation needs; the Microcontroller profile (M-Profile), that specifically targets embedded systems with constrained power; and the Real-time profile (R-Profile), that targets time-sensitive and safety-critical environments. We chose to work with A-Profile architectures (ARMv8-A), as they are used in a variety of applications: personal computers (e.g., Apple M1), servers (e.g., Ampere Altra), mobile devices (e.g., most Android devices) and higher-end IoT devices (e.g., Raspberry Pi).

ARMv8-A has two different execution modes: AArch64, which uses 64-bit wide registers and the A64 instruction set, introduced on its release, and AArch32, a 32-bit register mode that uses the A32 instruction set, providing compatibility with ARMv7 and earlier systems, as ARMv8 was the first to introduce 64-bit capabilities. In this document, we assume and work only with the AArch64 execution mode. Furthermore, ARMv8-A does not specify a paradigm for instruction execution, and processors may implement in-order execution (such as Cortex-A72) or out-of-order execution (such as Apple M1). This may imply a need for different optimizations depending on the target, as in-order processors might benefit greatly from instruction ordering, while out-of-order processors may achieve a similar performance by themselves, depending on the size of the look-ahead window.

ARMv8 and AArch64 capabilities. The AArch64 execution mode uses the A64 instruction set, which employs a load-store architecture. Its register bank includes 31 general-purpose, scalar, 64-bit wide registers, along with three special registers: a zero register, a stack pointer register and the program counter register (which is not directly

accessible). Alongside the scalar register bank, AArch64 also includes a *NEON* register bank, used by a specialized *Advanced SIMD* (Single-Instruction, Multiple Data) ALU that executes vectorized and floating-point operations. This register bank has 32 128-bit wide register that may be interpreted as 16 bytes, 8 half-words (16-bit values), 4 words (32-bit values), 2 doublewords (64-bit values) or a single 128-bit quadword. Loading values into the NEON register bank introduces some latency, as values must either be reloaded from memory or migrated from the scalar registers; however, the speed gain of processing vectorized data, when compared to processing the same amount of data with scalar operations, far outweighs its cost.

ARMv8 extensions. ARMv8 has multiple versions, ranging from ARMv8.1 to v8.6. These versions introduce requirements for certain instruction set extensions; however, manufacturers may include any number of extensions paired with earlier versions of the architecture with no restriction. The most relevant extensions for this work are the AES and SHA-3 cryptographic extensions, which include specialized instructions for these algorithms that significantly speed up their execution.

Interesting instructions. We now highlight some instructions of interest for our optimizations:

- **Multiple structure load/store:** The `LD1`, `LD2`, `LD3` and `LD4` instructions can be used to load values from a contiguous memory block into one to four different NEON registers, in an interleaved pattern. For example, suppose our values in memory are 32-bit integers, ranging from 1 to 16. By using the `LD1` instruction with four registers r_1, r_2, r_3, r_4, our values would be loaded as $r_1 = (1, 5, 9, 13), r_2 = (2, 6, 10, 14), r_3 = (3, 7, 11, 15), r_4 = (4, 8, 12, 16)$. This type of loading is more efficient than performing four different load operations, as long as the data supports this structure. Storing from one to four registers into contiguous memory may similarly be done with the `ST1` to `ST4` instructions.

- **Bitwise Select:** The bitwise select instruction `BSL` takes two source registers and one destination register as input. It reads each bit from the destination register and sets it as either the corresponding bit of the first source register, if the destination register bit was 1, or the second source register, if it was 0. This is done in constant-time by a series of logical micro-operations. This can be paired with comparison operations that write an all zero value when the comparison is not met and all ones when it is. This is particularly useful for Algorithms 3.3.1 and 3.3.2, as it speeds up table comparisons during integer sampling.

- **Zip and Unzip:** Zip and unzip instructions (`ZIP1`, `ZIP2`, `UZP1` and `UZP2`) are useful instruments to reorder NEON vectors. Zip instructions interleave elements from either the lower or upper half of two different vector registers into the destination register, while unzip does the reverse operation. This is useful for reordering data, but can also be useful when we only want the high or low bits of vector elements: by reinterpreting, say, two 32×4 bit vector as 16×8 bit vectors and taking only

the elements corresponding to the higher half of the original values, we are effectively computing a 16-bit right shift operation along with a load operation into the destination with a single instruction.

4.2 Implementation Challenges

As with any complex algorithm, going from theoretical descriptions to an implementation may pose significant problems. Nuances on how to implement certain operations, how to represent some data or how to create data structures and functions that conform to the expected behavior become apparent only when we turn pseudocode into actual code. In the following, we describe some of these challenges when implementing FALCON in C language, or in assembly code for ARMv8.

Floating-point Arithmetic. One of such challenges is managing floating-point precision and rounding issues throughout the algorithm. FALCON only requires floating-point numbers for specific operations, such as FFT and discrete Gaussian sampling. However, due to the tower of fields structures exploited by Algorithms 3.4.5 and 3.4.6, rounding errors and precision losses get propagated fast and impact the outcome of functions that use them as subroutines. One such problem might arise from a compiler optimization flag named `FP_CONTRACT`, which takes multiply instructions followed by additions and swaps them with fused multiply-add instructions, if available. While this optimization does improve running time, architectures such as ARMv8 may not round the intermediate value (which does happen when using `MUL` and `ADD` separately), leading to different results. This error is also intensified in recursive routines, such as `NTRUSolve` (Algorithm 3.4.3) and `ffSampling` (Algorithm 3.5.3). This leads to different algorithm outputs, which are incompatible with the provided test vectors and with the reference code. Disabling this optimization in both GCC and Clang helps ensure interoperability and that probability distributions follow the security proofs.

Arbitrary-sized Integer Arithmetic. As previously discussed, during the execution of `NTRUSolve` (Algorithm 3.4.3), polynomial coefficient size can go up to thousands of bits, which means we must have some way of representing them in memory and operating on them. A naive representation would be to split such coefficients in 31-bit or 63-bit "limbs" (leaving the most significant bit as zero, to keep track of carries during computations), and implement arithmetic similar to what would be a "base 2^{31}" arithmetic.

However, FALCON's authors implement a different technique, using the naive representation only when absolutely necessary. The first thing we note is that using NTT for representing polynomials leads not only to faster times in multiplications, but also simplifies other operations, such as the field norm. From Definition 16, we may express

$$f'(\omega^2) = \mathcal{N}(f)(\omega^2) = f(\omega)f(-\omega),$$

which means we can easily calculate f' from f in NTT representation. Similarly, com-

puting Lines 13 and 14 of Algorithm 3.4.3 may be done as

$$F(\omega) = F'(\omega^2)g(-\omega)$$

in NTT representation. The problem would then be finding a prime p large enough to fit even the largest coefficients, and operating inside $\mathbb{Z}_p$, in this case, could be potentially slower than the alternative. However, we can split our "large" polynomials into a *residue number system* (RNS). Namely, for a set of primes $(p_1, p_2, ..., p_j)$, we can split our polynomial $f(x) \in \mathbb{Z}[x]/(\phi)$ as

$$f(x) = \begin{cases} f_1(x) \bmod p_1 \\ f_2(x) \bmod p_2 \\ \vdots \\ f_j(x) \bmod p_j \end{cases}, \tag{4.1}$$

which fits our polynomials as long as the bitsize of $\Pi_j p_j$ is larger than our largest coefficient. Furthermore, if we have that $\forall p_j,\ p_j = 1 \bmod 2n$, we may apply NTT_{p_j} to every $f_j(x) \in \mathbb{Z}_{p_j}[x]/(\phi)$, and make our calculations inside the NTT domains with smaller primes, making full use of the benefits of the transform. Therefore, for most operations, every polynomial f is expressed as the set of transformed polynomials

$$(\text{NTT}_{p_1}(f_1), \text{NTT}_{p_2}(f_2), \ldots,\ \text{NTT}_{p_j}(f_j)).$$

We also note that, when f is small enough to fit only some primes inside the RNS, every subsequent $f_j(x)$ will be zero and does not need to be explicitly calculated or stored, making this method adaptable to different sizes.

The only limitation of this technique is that some operations, such as finding the Extended GCD and computing the polynomial reduction steps of Lines 15 to 19 of Algorithm 3.4.3, cannot be done inside this domain. Therefore, we still need to implement (although in a limited fashion) conversion to and from the naive representation, as well as multiplication by a "small" polynomial k (Lines 17 and 18) and Extended GCD using arbitrary-size integer arithmetic.

FFT and NTT. Implementing FFT and NTT may be done in many different ways: FFT may be structured as *decimation in time* (DIT) or *decimation in frequency* (DIF); both algorithms may be out-of-place or in-place; root values (ζ for FFT, ω for NTT, as per Definitions 13 and 14, respectively) may be calculated during execution or pre-computed and stored as a look-up table; and, if look-up tables are used, they may be ordered sequentially by their i index, or in *bit-reversal* ordering. Each choice offers certain trade-offs (execution time or memory usage, for example), and it is up to the programmer to decide how to implement them.

Our FALCON implementation uses decimation in frequency for FFT, calculate values in-place, and store pre-computed values in look-up tables following a bit-reversal ordering. These choices allow for faster operations with little increase in memory cost. We also note

that, since we have $\phi(x) = x^n + 1$ and n is a power-of-two, if ζ is a root of ϕ, then so is $\bar{\zeta}$, and $f(\bar{\zeta}) = \overline{f(\zeta)}$. A naive implementation would store n complex numbers, one for each root ζ, which leads to $2n$ floating-point numbers. However, due to this property, we may only store half the values, leading to n floating-point numbers, which may replace the original n-th degree polynomial f in memory in-place, as long as it is converted to a floating-point type before the function is called. Furthermore, we split our vector FFT(f) such that real values are stored in the first $n/2$ coefficients, and imaginary values are stored in the latter half of the vector. For example, noting $v[i]$ as the i-th vector position, $\texttt{Re}(x)$ (resp. $\texttt{Im}(x)$) as the real (resp. imaginary) part of a complex number and $\texttt{rev}(i)$ as applying bit-reversal to an integer i, $\texttt{Re}(f(\zeta_i))$ would be stored in position $v[\texttt{rev}(i)/2]$, but $\texttt{Im}(f(\zeta_i))$ would be in position $v[\texttt{rev}(i)/2 + n/2]$. This method for implementing the FFT is presented in Algorithm 4.2.1.

Algorithm 4.2.1 FFT($f, \texttt{ZETAS}$)

Require: $f = \sum_{i=0}^{n} f_i x^i \in \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two, and `ZETAS` a table containing $n/2$ complex numbers, corresponding to the complex roots of $\phi = x^n + 1$ in bit-reversal order.

Ensure: FFT(f), calculated in-place.

1: $t \leftarrow n/2$
2: $m \leftarrow 2$
3: **for** i from 1 to $\log(n)$ **do**
4: $\quad s \leftarrow 0$
5: $\quad$ **for** k from 0 to $m/2$ **do**
6: $\quad\quad$ **for** j from s to $s + (t/2)$ **do**
7: $\quad\quad\quad x \leftarrow (f_j, f_{j+n/2})$ ▷ $x \in \mathbb{C}$
8: $\quad\quad\quad y \leftarrow (f_{j+t/2}, f_{j+t/2+n/2})$ ▷ $y \in \mathbb{C}$
9: $\quad\quad\quad y \leftarrow y \cdot \texttt{ZETAS}[m + k]$ ▷ $\texttt{ZETAS}[i] \in \mathbb{C}$
10: $\quad\quad\quad (f_j, f_{j+n/2}) \leftarrow (\texttt{Re}(x + y), \texttt{Im}(x + y))$
11: $\quad\quad\quad (f_{j+t/2}, f_{j+t/2+n/2}) \leftarrow (\texttt{Re}(x - y), \texttt{Im}(x - y))$
12: $\quad\quad s \leftarrow s + t$
13: $\quad t \leftarrow t/2$
14: $\quad m \leftarrow m \cdot 2$
15: **return** f.

Finally, we note that, due to the bit-reversal ordering, the `ZETAS` table may be used for both $n = 512$ and $n = 1024$. Applying FFT^{-1} follows the same structure, calculating the inverse of the forward function. Our NTT_q implementation follows the same basic structure, and is described in Algorithm 4.2.2. Multiplication modulo q is done using Montgomery modular multiplication [37]. $\text{NTT}_{\text{q}}^{-1}$ applies the inverse operation following the same structure.

4.3 Side-Channel Attacks

Another challenge to overcome when implementing a cryptographic algorithm is securing the implementation against side-channel attacks. We now review the attacks that target

Algorithm 4.2.2 $\mathrm{NTT}_q(f, \texttt{OMEGAS})$

Require: $f = \sum_{i=0}^{n} f_i x^i \in \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two, and `OMEGAS` a table containing n integers, corresponding to the roots of $\phi = x^n + 1$ in $\mathbb{Z}_q$, in bit-reversal order.

Ensure: $\mathrm{NTT}_q(f)$, calculated in-place.

1: $k \leftarrow 1$
2: $t \leftarrow n/2$
3: **while** $t \geq 1$ **do**
4: $\quad s \leftarrow 0$
5: $\quad$ **while** $s < n$ **do**
6: $\quad\quad$ **for** j from s to $s + t$ **do**
7: $\quad\quad\quad m \leftarrow f_{j+t} \cdot \texttt{OMEGAS}[k] \bmod q$ ▷ Using Montgomery multiplication
8: $\quad\quad\quad f_{j+t} \leftarrow f_j - m \bmod q$
9: $\quad\quad\quad f_j \leftarrow f_j + m \bmod q$
10: $\quad\quad k \leftarrow k + 1$
11: $\quad\quad s \leftarrow s + 2t$
12: $\quad t \leftarrow t/2$
13: **return** f.

FALCON and analyze their applicability to our implementation.

McCarthy *et al.* [36] proposed a fault attack and a timing attack against the second round implementation of FALCON. The timing attack is no longer possible, as the third round version of the algorithm has already implemented countermeasures against it. The timing attack, however, can be done by interrupting a loop on the signature process – more specifically, a loop that is part of the trapdoor sampling –, so that it forces the sampled vector to have an arbitrary number of zeroes. These zeroes, then, create signatures that depend only on a subset of the subjacent lattice basis; as such, by sampling enough faulty signatures, an attacker may then run an SVP solver with small enough parameters fast enough for the attack to be practical, recovering the key through smaller steps. A simple countermeasure is checking the sampled vector for repeated zeroes, and our implementation chooses this approach, as it has minimal impact on performance.

Karabulut and Aysu [31] propose a side-channel attack against FALCON through differential electromagnetic analysis against floating-point operations. However, this analysis relies on a particular mode of the reference implementation; namely, the one geared towards architectures that do not have FPUs, in which floating-point operations are simulated using 64-bit integer variables. The attack focuses on floating-point FFT multiplications, since the bitsize limitation on the variables calls for partial results, which can be analyzed by a side-channel attacker and later expanded to the whole secret key. This attack does not pose a challenge to our implementation, however, as our target architecture does not possess such limitations, as it does indeed have an FPU available and does not require partial results.

Guerreau *et al.* [26] present two new side-channel attacks on FALCON. The first attack is an improved version of the one presented by Karabulut and Aysu [31], with lower complexity and requiring less samples. By proving that values can be estimated through

rounding and later corrected through lattice operations, the authors reduce the amount of bits that need to be extracted from the partial results, and also make use of later operations on the code to infer more data from the same samples. However, as stated before, this attack is on a specific technique used by the reference implementation to emulate floating point operations on devices that lack an FPU, which is not the case of our target architecture.

Finally, Howe and Westerbaan [28] show timing leakages on floating-point operations in Cortex-M7, as operations between numbers with the same exponent take less time than when the exponents are largely different in this architecture. The authors also show that, on Cortex-A53, the type casting between `double` and `int64_t`, which implies a round-to-zero operation, leaks the sign bit due to variable timing. This problem arises from LLVM's non-constant time implementation of round-to-zero in a chip (A53) that does not have a specific instruction for this operation. The authors do not mount an attack on these leakages, but point out the importance of checking constant-timeness of floating-point operations before real-world deployment.

4.4 Optimization

We focused on optimizing signature generation and verification over key generation, for two reasons: first, they are the most frequently executed algorithms, as key generation tends to be sporadic (the same keypair may be used for generating a large number of signatures, which are in turn verified multiple times); second, since they share many subroutines, optimizations for, say, `HashToPoint` (Algorithm 3.5.2), impact both routines simultaneously, and offer a greater benefit for the scheme as a whole.

To guide our choice of which routines focused on, we profiled our initial C implementation to find bottlenecks and costly routines for each algorithm. Tables 4.1 and 4.2 show the most expensive functions of each algorithm, including those shared by both `Sign` and `Verify`. The list is not extensive, both for simplicity and because smaller functions have been inlined by the compiler during optimization. These results have been obtained running the GNU Profiler tool [24] on an NVIDIA® Jetson Nano™ development board, which features a Cortex-A57 processor. We note that other platforms present similar results, and thus we do not list them for brevity.

Our optimizations focused mainly on fundamental algorithms shared across routines, such as FFT/NTT operations, as they are executed many times across the whole scheme, and the `HashToPoint` subroutine, which takes a noticeable amount of execution time in both algorithms, as the profiling results show. Integer sampling inside signature generation was another primary target, as it takes the largest portion of signature generation time.

We now discuss the different techniques used in each optimized routine, highlighting the reasoning behind each optimization.

4.4.1 Loop vectorization

The first and most simple optimization is using SIMD instructions to vectorize iterative functions. For simpler loops, the compiler is able to achieve this by itself; however, when

Table 4.1: Profiling results ranking the most expensive functions of Falcon512.

Function	**Sign**	**Verify**
`HashToPoint`	9.67%	43.67%
$\text{NTT}_q(f)$	6.47%	20.83%
$\text{NTT}_q^{-1}(f)$	0.72%	16.67%
`Decompress`	–	10.45%
`Verify` (inlined subroutines)	–	8.35%
`SampleInteger`	36.07%	–
`ffSampling` (with FFT subroutines)	13.97%	–
`ffLDL` (with FFT subroutines)	9.25%	–
Division modulo q	8.27%	–
`Sign` (inlined subroutines)	2.52%	–
Total	86.94%	99.97%

Table 4.2: Profiling results ranking the most expensive functions of Falcon1024.

Function	**Sign**	**Verify**
`HashToPoint`	7.53%	39.65%
$\text{NTT}_q(f)$	7.61%	30.64%
$\text{NTT}^{-1}{}_q(f)$	2.32%	8.17%
`Decompress`	–	4.10%
`Verify` (inlined subroutines)	–	17.42%
`SampleInteger`	36.71%	–
`ffSampling` (with FFT subroutines)	12.98%	–
`ffLDL` (with FFT subroutines)	9.13%	–
Division modulo q	6.45%	–
`Sign` (inlined subroutines)	6.53%	–
Total	89.26%	99.98%

the values are, for example, complex numbers (as the roots and coefficients of FFT are), this is not as straightforward. We detail each general case for this type of optimization below.

Transforms with Look-up Tables. As shown in Algorithms 4.2.1 and 4.2.2, applying transforms with look-up tables requires several iterative operations, which may be optimized with SIMD instructions. For NTT_q, during the loop in Algorithm 4.2.2, Line 6, whenever $t \geq 8$, we may load 8 16-bit elements into a 128-bit NEON register, replicate `OMEGAS`$[k]$ 8 times into another NEON register, and apply the operations simultaneously, saving them as contiguous 8-element blocks. Special cases must be done for $t = 4$, $t = 2$ and $t = 1$; however, other strategies, such as loading multiple copies of different `OMEGAS`$[k]$ elements may be employed, such that the whole function benefits from SIMD instructions. However, for NTT_q^{-1}, while similar optimizations may be performed, the order in which different `OMEGAS`$[k]$ are applied is different, and SIMD instruction loads require contiguous memory blocks. Therefore, by creating a new, reordered table, we may optimize NTT_q^{-1} similarly, at the cost of storing a new table in memory.

For FFT, our strategy is slightly different. We work with 64-bit floating-point numbers and, therefore, may only load two at the same time in a NEON register. Furthermore,

we may choose, instead, to load both the real and imaginary parts of a complex number in one register, if they are contiguous in memory. As stated before, this is not the case for the polynomial f; however, we may organize our `ZETAS` table so that the ζ coefficients are. Our optimization is then as thus: we load both $\texttt{Re}(\zeta)$ and $\texttt{Im}(\zeta)$ into a single NEON register when loading $\texttt{ZETAS}[m+k]$ in Line 9 of Algorithm 4.2.1; however, we instead load two contiguous elements of f, say f_i and f_{i+1}, inside the `for` loop of Line 6, and apply the complex number arithmetic operations for both simultaneously. Finally, for special cases when $\log(n)$ is small enough not to allow such operations, we make use of more NEON registers and load values with interleaving.

Coefficient-wise operations. Another interesting targets for loop vectorization are coefficient-wise operations, such as polynomial addition/subtraction and operations in the FFT and NTT domains. While the compiler may find opportunities to vectorize integer, coefficient-wise operations, others, such as Montgomery multiplication, are less trivial. We have applied a number of SIMD instructions such as widening vector multiplication, multiply-add and zipping/unzipping (in place of shifts, as described earlier) to process 8 elements at a time, while keeping the algorithm constant-time. Furthermore, we make use of multi-register loads such as `LD4` to further minimize time spent on memory access.

Integer Sampling. Finally, we optimize `Sign`'s integer sampling, in particular Algorithm 3.3.2, by using SIMD instructions to load values from the RCDT and compare them to the sampled values. The first thing we note is that, since the RCDT requires 72-bit precision, we split it into a 64-bit table and an 8-bit table, with the smaller table representing the most significant bits. Uniform bit sampling is also done in two steps: one for 64 bits, and another for 8 bits. Both tables have a total of 19 entries.

With this, we were able to parallelize comparison in the following manner: We load the first 16 64-bit entries in 8 different NEON registers, and duplicate our 64-bit random sample into another one. We then compare them in pairs, overwriting the registers containing table values. We then use zipping and unzipping instructions to gather these results as 8-bit entries, since they are now either sequences of 1s or 0s, and their length does not matter. This leaves us with a 16×8-bit register containing all 64-bit comparison results. We load and compare the 8-bit segments using a single NEON register for each, and then use constant time techniques to write the total comparison values. For the last 3 entries, they are treated as 8-bit literals due to their short length, and are computed in constant time using scalar instructions.

4.4.2 Pseudorandom Number Generation Using AES

FALCON's specification requires cryptographically secure pseudorandom number generation every time random bits are uniformly sampled. However, which algorithm is used to derive random bits is left to the programmer. The reference implementation uses a PRNG based on the ChaCha20 algorithm, keeping a buffer of 512 bytes and regenerating random bits in 512-byte blocks whenever the buffer is emptied. This choice appears to fit Intel architectures well, as the authors' comments state that AVX2 can process up to

eight ChaCha20 instances in parallel. However, this is not the case for ARMv8. While this PRNG's performance is not unacceptable in ARMv8, we may use specialized instructions from the cryptographic extensions to speed up random bit generation. Therefore, we evaluated the performance of using SHAKE256 and AES as sources of randomness.

SHAKE256 is part of FALCON's specification for message hashing, and is also used to initialize PRNGs with a certain amount of random information, as its sponge function may output variable-length bit sequences. However, as a PRNG, SHAKE256 is slower than ChaCha20, even when using specialized instructions, due to its complexity. Even so, we still achieved higher speeds for sampling by using ARM-specific instructions and applying the SHAKE256 improvements that will be discussed in the next section.

AES, on the other hand, is able to achieve significantly higher speeds than its other options. We chose to implement AES CTR-DRBG [7], a standardized version of deterministic random bit generation using AES, using specialized ARMv8 instructions. We were able to compute up to 20 times more bits in the same time frame as ChaCha20, and also keep a buffer of 768 bytes without losing performance due to cache misses and other memory issues. This led to significant improvements in signature and key generation, the latter of which will be further elaborated in Section 4.4.4.

4.4.3 Improving SHAKE256

While this optimization is not of FALCON itself, SHAKE256 plays a big role in the `HashToPoint` function, which is relevant for both `Sign` and `Verify` algorithms, as shown in Tables 4.1 and 4.2. Another factor is that, as previously stated, ARMv8's SHA3 extension provides specialized instructions that speed up the algorithm through a straightforward reimplementation. However, not all ARMv8 processors have these instructions and, so, optimization strategies must change for such platforms.

We start by analyzing our gains using specialized instructions. While reimplementing the algorithm using all specialized instructions is the simplest way of achieving better performance, it is not always the best. An interesting discovery during development was that even though powerful processors may include this extension, they may not be as useful as expected. One such case is the Cortex-X2: in this platform, even though it has 4 SIMD execution pipelines (in which the extension instructions execute), only one of them implements the SHA3 extension when it is available. While such instructions, taken as single units, take less time than computing them with the A64 base instruction set, only using them to compute a SHA3 round means we bottleneck our processing capacity into a single pipeline, making the overall performance of using only SHA3 extensions on Cortex-X2 worse than a scalar implementation.

For platforms without this extension, we explored a number of different techniques. First, we employed techniques such as in-place computation [11] and lane-complementing [9] to reduce the number of instructions required for computing each round of the algorithm. These optimizations proved beneficial on almost every platform; however, in cases such as the Apple M1, instructions that include implicit rotations take longer than regular instructions. This means that lane-complementing techniques, which make use of such instructions, perform worse than its regular counterparts. However, Apple M1 implements

the SHA3 extension, and therefore does not require these techniques.

Next, we noted that most platforms we experimented on that did not have this extension – namely, Cortex-A53, Cortex-A57 and Cortex-A72 – were in-order platforms. This means that instruction scheduling plays a relevant part in optimization, queuing instructions of different latencies and data dependencies to make the most of the processors' pipelines. Scheduling instructions by hand proved inefficient, as the balancing between instruction types, latencies, data dependencies, decodification windows and many other factors were hard to predict. Therefore, we implemented a combinatorial optimization algorithm using Simulated Annealing [32] to reorder and swap instructions, then patch the executable in memory and benchmark the algorithm running on hardware, until reorderings close to a theoretical optimal were found.

Therefore, for our final implementations, we use only SHA3 instructions for Apple M1; we use an interleaved SIMD implementation using both regular and SHA3 instructions, balanced as to fill every pipeline properly, for Cortex-X2; and for Cortex-A57, we have a tailored scalar implementation with heavy reordering and instruction swapping to overcome the processor's limitations.

4.4.4 Integer Sampling for Key Generation

Finally, our last optimization was regarding the integer sampling done by Algorithm 3.3.1, and therefore, exclusive to key generation. The two main differences, in terms of implementation, from the signature sampling are that we use 63 bits of precision (and, thus, a single 64-bit entry table), and that our table follows the distribution indicated by the `KGPT` in Section 3.3. This means that comparisons are slightly different and the sample for the value 0 must be treated differently.

The first optimization of note is that we are able to compute a batch of $2n$ 64-bit samples right from the start, making better use of caches and using our AES-based PRNG to its fullest, minimizing sampling time drastically. Since we must treat 0 as a different distribution, we need 2 samples for each coefficient and, therefore, calculate $2n$ samples.

Next, we are able to load the whole `KGPT` into SIMD registers. For $n = 512$, the table has 27 entries, but since the first one must be treated separately (since it represents the probability of sampling zero), we use a total of 13 NEON registers to represent it. For $n = 1024$, we have 37 entries, and similarly, we use a total of 18 NEON registers, leaving the first entry as scalar. Differently from signature generation, we do not overwrite these values; rather, we keep them in the register bank until we sample all n integers. This means that, especially for $n = 1024$, we have to be careful about register allocation, to avoid unnecessary spills. Our implementation uses a combination of comparison, regular addition and pairwise addition for a total of 24 NEON registers when $n = 1024$, leaving ample space to avoid spilling.

Unfortunately, due to development time and project structure limitations, we were not able to include this optimization in our final version of `Keygen`, meaning our results will not reflect this change. However, we include separate results comparing only the reference implementation's sampling technique (using ChaCha20) with ours in Section 5, to quantify the improvement in performance.

Chapter 5

Experimental Results

In this section, we describe and discuss our experiments and implementation results, detailing the methodology used and how our techniques have improved the scheme as a whole in different platforms.

Methodology. Performance was measured using the Google Benchmark [25] framework. We used three devices to collect experimental results: a Samsung Galaxy S22 (Cortex-X2), a NVIDIA® Jetson Nano™ development board (Cortex-A57) and an Apple MacBook Air® featuring the Apple M1 system-on-chip. Details on each platform are as follows.

Jetson Nano contains four Cortex-A57 cores, and supports the AES cryptographic instructions, but does not feature the SHA-3 extension.

Galaxy S22 has 8 cores, being four Cortex-A510, three Cortex-A710 and the performance Cortex-X2 core, which was the one used for our experiments. It implements both the AES and SHA-3 cryptographic extensions. Although this platform's architecture is ARMv9, rather than ARMv8, it has complete compatibility with ARMv8 implementations and no optimizations restricted to the ARMv9 platform were performed.

Finally, MacBook Air M1 features eight cores: four Icestorm units (high-efficiency microarchitecture) and four Firestorm units (high-performance microarchitecture), and we use the latter for our experiments. They implement the ARMv8.4-A instruction set, and feature both AES and SHA-3 extensions.

Other processor specifications, such as clock frequencies and cache sizes, are presented in Table 5.1

Processor	**Clock Frequency**		**Cache Sizes**			**Extensions**	
	Min.	**Max.**	**L1**	**L2**	**L3**	**AES**	**SHA-3**
Cortex-A57	102 MHz	1479 MHz	32K	48K	3M	✓	-
Cortex-X2	2840 MHz	3920 MHz	128K	1M	8M	✓	✓
M1 Firestorm	600 MHz	3204 MHz	192+128K	12M	-	✓	✓

Table 5.1: Summary of specifications of processors used for experiments.

The binaries were compiled using the Clang compiler for all platforms, with the -O3 optimization flag enabled. For the benchmarks, the binaries were linked as a static library.

AES instructions are available in all platforms, while only the Apple M1 and Galaxy S22 devices make use of SHA-3 cryptographic instructions.

Results. In Tables 5.2 and 5.3, we summarize the results of our measurements for each parameter set. We compare the reference implementation [22] with our optimized implementation, and calculate the improvement percentage as

$$\text{Gain } (\%) = \frac{\text{Ref. Time} - \text{Our Time}}{\text{Our Time}} \times 100$$

More detailed results, with concrete numbers, along with data on the most relevant sub-algorithms can be found in Appendix A.

Falcon512			
Algorithm	**Cortex-A57**	**Cortex-X2**	**Apple M1**
Keygen	8%	3%	8%
Sign	44%	56%	79%
Verify	44%	33%	61%

Table 5.2: Summary of gains of our results comparing the C reference code with our optimized ARMv8 implementation for Falcon512

Falcon1024			
Algorithm	**Cortex-A57**	**Cortex-X2**	**Apple M1**
Keygen	7%	3%	6%
Sign	36%	56%	78%
Verify	49%	32%	58%

Table 5.3: Summary of gains of our results comparing the C reference code with our optimized ARMv8 implementation for Falcon1024

We also compare our optimized implementation using AES instructions for integer sampling during key generation, which, as previously stated, we were not able to include in the final key-generation code, and therefore serves as a separate result. Improvement percentage is calculated in the same manner as before. Detailed results, including results for our SHAKE256-based implementation of integer sampling, can also be found in Appendix A.

Parameter Set	**Cortex-A57**	**Cortex-X2**	**Apple M1**
Falcon512	1179%	2058%	1746%
Falcon1024	655%	1283%	1145%

Table 5.4: Summary of gains of our results comparing the C reference code and our optimized ARMv8 implementation for integer sampling in key generation

We note that these algorithms take around 20% of key generation time, and therefore these optimizations are not only relevant but mean a gain of up to 14% to the algorithm as a whole.

Discussion. From our experimental results, we were able to see significant performance improvements across all platforms on our primary targets, `Sign` and `Verify`. The M1 SoC was able to make the most of our optimizations due to its complete cryptographic instruction set, accelerating both hashing and sampling, making this platform the best performing for signature generation. Cortex-X2 lags behind the M1 chip on `Sign` due to its pipeline issue with SHA-3 instructions, losing considerably in performance in the `HashToPoint` algorithm. Cortex-A57 lacks SHA-3 instructions altogether, and is not able to make the most of our optimized FFT instructions either, coming last in terms of signature generation.

Verification, however, brought interesting results: Cortex-X2 was not able to perform well with our optimizations for `Decompress`, and despite having a reasonable improvement in NTT function timings, this improvement was less pronounced than it was on the other platforms, and therefore had less accentuated gains. Finally, between the M1 SoC and Cortex-A57, we once again notice the impact of SHA-3 instructions being available on the former, increasing performance significantly more. We note that, since `Verify` is a fast routine overall, small performance changes on its inner functions and sub-algorithms have a great impact on the overall result.

Key generation makes use of our improvements for FFT and NTT functions, as well as encoding and decoding; however, the bulk of the algorithm, namely solving the NTRU equation, was not optimized, and therefore relied on the compiler's capabilities of vectorization and using platform-specific instructions where appropriate.

Finally, our results regarding key generation's integer sampling exemplify the power of specialized instructions. We note that the reference results are similar for both parameter sets since the authors use only a single RCDT, calculated using $n = 1024$, and sample twice from it for $n = 512$, employing a result that states that the sum of two independent values of standard deviation σ has a standard deviation $\sigma\sqrt{2}$, which is precisely the standard deviation needed in the Falcon512 parameter set. Therefore, while the sampler needs to sample half the values than it does for $n = 1024$, it samples twice for each value, resulting in the same number of operations for both parameter sets.

Chapter 6

Conclusion and Future Work

We have presented an optimized implementation of FALCON for the ARMv8-A architecture that achieves up to 79% faster speeds for signature generation, and up to 61% faster for verification. We have achieved this using techniques that leverage the platform's capabilities, such as loop vectorization using SIMD instructions, faster pseudorandom number generation and hash computation using cryptographic instructions, among others.

We do note, however, that key generation still has a long way to go. We were not able to implement new ideas for `Keygen` due to development time constraints, as well as not being able to introduce our new AES-based sampling method into the final code. Even so, we have prototyped and confirmed the viability of a few ideas, which we discuss next, and leave as future work.

Vectorization. While FFT operations and similar sub-algorithms have been optimized, there is still much that can be done in terms of vectorizing operations during key generation. When "small" polynomials are operated on (such as the NTT+RNS representation, or when dealing with small coefficients), we might be able to improve performance through SIMD instructions, in the same vein that was done for the two other main algorithms.

Arbitrary Precision Floating-Point Arithmetic. Another interesting idea would be to use arbitrary precision floating-point arithmetic to compute the polynomial reduction on Algorithm 3.4.3 (Lines 15 to 19) in a single step. We have prototyped this idea using SageMath [47] and FLINT [27], and found out that when we are on the recursive calls ranging from $\log n = 4$ to $\log n = 1$, the cost of using arbitrary precision becomes noticeably smaller than the cost of computing k with double precision multiple times. This may improve one of the most costly steps of key generation; however, it requires either careful implementation or the usage of external libraries such as FLINT.

Tailoring RNS primes for ARMv8. A further optimization we propose is tailoring the primes p_j used by the Residue Number System to make full usage of the ARMv8 platform. The reference implementation uses a list of 521 32-bit primes to express the polynomials, such that computations may be done even on lower-end chips and microcontrollers, such as the Cortex-M4. However, we have prototyped a different prime list using Sage, one that uses 150 64-bit primes and 225 32-bit primes, and guaranteed its

correctness. The advantage of using this list is that we would be able to make full use of both scalar and SIMD APUs: we can use SIMD to calculate 32x4-bit operations, while scalar operations calculate values using 64-bit arithmetic. By parallelizing operations not only through vectorization, but also making full use of the architecture's capabilities by using both APUs simultaneously, we believe NTT+RNS operations can be significantly sped up.

Bibliography

[1] M. Ajtai. Generating hard instances of lattice problems (extended abstract). *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 99–108, 1996.

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the first round of the NIST post-quantum cryptography standardization process. Technical report, NIST, 2019.

[3] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical report, NIST, 2020.

[4] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, NIST, 2022.

[5] Arm. Arm's solution to the future needs of ai, security and specialized computing is v9. Arm Newsroom, 2021. `https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture`.

[6] L Babai. On lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.

[7] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2015.

[8] Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits. *arXiv*, 2002.

[9] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems, 2022.

[10] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS⁺ signature framework.

In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, 2012.

[12] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ASIACRYPT'11, page 41–69, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 353–367. IEEE, 2018. `http://cryptojedi.org/papers/#kyber`.

[14] Chitchanok Chuengsatiansup, Thomas Prest, Damien Stehlé, Alexandre Wallet, and Keita Xagawa. Modfalcon: Compact signatures based on module-ntru lattices. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2020*, pages 853–866, 10 2020.

[15] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[16] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. *Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018. `http://cryptojedi.org/papers/#dilithium`.

[17] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over ntru lattices. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 22–41, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[18] Léo Ducas and Thomas Prest. Fast fourier orthogonalization. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, page 191–198, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Thomas Espitau, Pierre Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13277 LNCS:222–253, 2022.

[20] Thomas Espitau, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Shorter hash-and-sign lattice-based signatures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13508 LNCS:245–275, 2022.

[21] OpenBSD Foundation. Openssh, dec 2021.

[22] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. `https://falcon-sign.info/falcon-round3.zip`.

[23] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 197–206, New York, NY, USA, 2008. Association for Computing Machinery.

[24] GNU. The gnu profiler, jan 2023.

[25] Google. Google benchmark, jan 2023.

[26] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 141–164, 6 2022.

[27] W. B. Hart. Fast Library for Number Theory: An Introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. `https://flintlib.org`.

[28] James Howe and Bas Westerbaan. Benchmarking and analysing the nist pqc finalist lattice-based signature schemes on the arm cortex m7. *Cryptology ePrint Archive*, 2022.

[29] IBM. Ibm unveils new roadmap to practical quantum computing era; plans to deliver 4,000+ qubit system. IBM Newsroom, 2022. `https://newsroom.ibm.com/2022-05-10-IBM-Unveils-New-Roadmap-to-Practical-Quantum-Computing-Era-Plans-to-Deliver-4,000-Qubit-System`.

[30] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

[31] Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. *Proceedings - Design Automation Conference*, 2021-December:691–696, 12 2021.

[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[33] Philip Klein. Finding the closest lattice vector when it's unusually close. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 937–941, USA, 2000. Society for Industrial and Applied Mathematics.

[34] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 12 1982.

[35] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[36] Sarah McCarthy., James Howe., Neil Smyth., Séamus Brannigan., and Máire O'Neill. Bearz attack falcon: Implementation attacks with countermeasures on the falcon signature scheme. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - SECRYPT,*, pages 61–71. INSTICC, SciTePress, 2019.

[37] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[38] Arm Newsroom. Arm discloses technical details of the next version of the arm architecture. `https://web.archive.org/web/20111030002208/http://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-architecture.php`, 2011. Accessed: 2023-02-23.

[39] National Institute of Standards and Technology - NIST. Post-Quantum Cryptography, 2016. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.

[40] Chris Peikert. A decade of lattice cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10:283–424, 2016.

[41] Thomas Prest. Sharper bounds in lattice-based cryptography using the rényi divergence. *Advances in Cryptology – ASIACRYPT 2017*, pages 347–374, 2017.

[42] Eun-Young Seo, Young-Sik Kim, Joon-Woo Lee, and Jong-Seon No. Peregrine: Toward fastest falcon based on gpv framework. *Cryptology ePrint Archive*, 2022.

[43] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 10 1997.

[44] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 14–37, Cham, 2017. Springer International Publishing.

[45] Shuo Sun, Yongbin Zhou, Yunfeng Ji, Rui Zhang, and Yang Tao. Generic, efficient and isochronous gaussian sampling over the integers. *Cybersecurity*, 5:10, 12 2022.

[46] Shuo Sun, Yongbin Zhou, Rui Zhang, Yang Tao, Zehua Qiao, and Jingdian Ming. Fast fourier orthogonalization over ntru lattices. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13407 LNCS:109–127, 2022.

[47] The Sage Developers. *SageMath, the Sage Mathematics Software System*. `https://www.sagemath.org`.

[48] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. Facct: Fast, compact, and constant-time discrete gaussian sampler over integers. *IEEE Transactions on Computers*, 69(1):126–137, 2020.

Appendix A

Experimental Data Tables

Falcon512									
Algorithm	**Cortex-A57**			**Cortex-X2**			**Apple M1**		
	Ref.	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**
Keygen	24872	22934	**8**%	5917	5733	**3**%	5669	5266	**8**%
Sign	1137	790	**44**%	223	143	**56**%	206	115	**79**%
Verify	147	102	**44**%	32.2	24.2	**33**%	27.3	17	**61**%

Table A.1: Benchmark results comparing the C reference code and our optimized ARMv8 implementation for the main algorithms of Falcon512. Timings are in μs.

Falcon1024									
Algorithm	**Cortex-A57**			**Cortex-X2**			**Apple M1**		
	Ref.	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**
Keygen	72035	67628	**7**%	18608	18056	**3**%	17283	16380	**6**%
Sign	2308	1695	**36**%	452	289	**56**%	414	233	**78**%
Verify	297	199	**49**%	66.3	50.2	**32**%	55	34.8	**58**%

Table A.2: Benchmark results comparing the C reference code and our optimized ARMv8 implementation for the main algorithms of Falcon1024. Timings are in μs.

Falcon512									
Sub-Algorithm	**Cortex-A57**			**Cortex-X2**			**Apple M1**		
	Ref.	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**
FFT	9.884	7.424	**33**%	1.296	0.899	**44**%	0.924	0.635	**46**%
FFT^{-1}	10.91	8.491	**29**%	1.275	0.930	**37**%	0.959	0.669	**43**%
ffSampling	622.9	395.5	**58**%	131.5	72.55	**81**%	132.2	72.63	**82**%
NTT	16.48	6.461	**155**%	3.044	1.284	**137**%	1.905	0.532	**258**%
NTT^{-1}	15.89	6.531	**143**%	2.903	1.196	**143**%	1.847	0.567	**226**%
Montg. Mul. (NTT)	2.795	0.609	**359**%	0.640	0.107	**498**%	0.438	0.057	**669**%
HashToPoint	74.98	66.94	**12**%	19.55	18.11	**8**%	18.31	13.78	**33**%
Compress	6.211	6.169	**1**%	2.023	1.371	**48**%	1.278	0.973	**31**%
Decompress	7.858	8.301	**−5**%	1.461	1.518	**−4**%	1.409	1.100	**28**%
SignSampler	0.505	0.277	**83**%	0.103	0.052	**96**%	0.108	0.053	**103**%
Encode (Priv. Key)	4.113	2.215	**86**%	1.017	0.525	**94**%	0.876	0.312	**181**%
Decode (Priv. Key)	3.322	3.477	**−4**%	0.873	0.971	**−10**%	0.496	0.518	**−4**%
Encode (Pub. Key)	8.419	1.161	**625**%	1.413	0.276	**413**%	1.300	0.192	**578**%
Decode (Pub. Key)	4.089	1.236	**231**%	0.789	0.294	**168**%	0.489	0.209	**134**%

Table A.3: Sub-algorithm benchmark results comparing the C reference code and our optimized ARMv8 implementation for Falcon512. Timings are in μs.

Falcon1024									
Sub-Algorithm	**Cortex-A57**			**Cortex-X2**			**Apple M1**		
	Ref.	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**	**Ref.**	**Ours**	**Gain**
FFT	21.21	16.93	**25**%	2.777	1.990	**40**%	1.994	1.412	**41**%
FFT^{-1}	23.44	19.06	**23**%	2.747	2.052	**34**%	2.052	1.496	**37**%
ffSampling	1248.6	802.0	**56**%	259.3	145.2	**79**%	261.9	142.4	**84**%
NTT	35.39	13.88	**155**%	6.528	2.799	**133**%	4.014	1.164	**245**%
NTT^{-1}	33.74	13.91	**143**%	6.116	2.598	**135**%	3.864	1.235	**213**%
Montg. Mul. (NTT)	5.566	1.188	**368**%	1.272	0.213	**497**%	0.871	0.113	**669**%
HashToPoint	147.9	126.9	**17**%	38.32	36.90	**4**%	37.01	27.63	**34**%
Compress	11.89	11.48	**4**%	3.627	2.413	**50**%	2.284	1.679	**36**%
Decompress	20.87	13.33	**57**%	3.464	3.026	**14**%	3.224	2.197	**47**%
SignSampler	0.495	0.275	**80**%	0.101	0.051	**97**%	0.106	0.051	**106**%
Encode (Priv. Key)	9.128	4.226	**116**%	1.918	1.034	**85**%	1.274	0.617	**106**%
Decode (Priv. Key)	8.022	6.942	**16**%	1.680	1.897	**−11**%	0.975	1.013	**−4**%
Encode (Pub. Key)	16.83	2.275	**640**%	2.818	0.551	**412**%	1.711	0.360	**375**%
Decode (Pub. Key)	8.159	2.452	**233**%	1.581	0.586	**170**%	0.971	0.414	**135**%

Table A.4: Sub-algorithm benchmark results comparing the C reference code and our optimized ARMv8 implementation for Falcon1024. Timings are in μs.

Algorithm	Cortex-A57		Cortex-X2		Apple M1	
	n = 512	**n = 1024**	**n = 512**	**n = 1024**	**n = 512**	**n = 1024**
Reference	348	348	96.7	97.7	53.0	53.4
Ours + SHAKE	127	242	37.6	72.3	18.2	35.3
Ours + AES	27.2	46.1	4.48	7.06	2.87	4.29
Gain (Ref. to AES)	**1179**%	**655**%	**2058**%	**1283**%	**1746**%	**1145**%

Table A.5: Benchmark results comparing the C reference code and our two optimized ARMv8 implementations for integer sampling in key generation. Timings are in μs.