TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

On the Physical Security of Falcon

im Rahmen des Studiums
Technische Mathematik

eingereicht von
Markus Schönauer, BSc
Matrikelnummer 01605049

unter der Anleitung von
Associate Prof. Dipl.-Ing. Dr. Georg Fuchsbauer

ausgeführt am
Institut für Logic and Computation
der Fakultät für Informatik
der Technischen Universität Wien

Markus Schönauer

Georg Fuchsbauer

Abstract

FALCON has been selected for standardization by NIST as a post-quantum digital signature scheme. Although there have been several published attacks against FALCON that target vulnerabilities on the implementation side, the physical security of the scheme has not been thoroughly studied yet. We present a broad analysis of the physical security of the FALCON signature scheme that includes attacks from prior publications as well as novel vulnerabilities and takes into account the mathematical foundations of the scheme, such as the Fast Fourier Transform, discrete Gaussian distributions and a tower of fields.

Additionally, we closely investigate one of the new attack vectors, NarrowSampling. We simulate a fault injection attack based on a parallelepiped learning technique applied to a signature distribution with lowered standard deviation. Each individual phase of the attack is analyzed and the influence of the most important attack parameters is measured and evaluated. For FALCON parameter sets with reduced security we are able to fully recover the secret key.

Zusammenfassung

FALCON ist von NIST zur Standardisierung als Post-Quanten digitales Signaturverfahren ausgewählt worden. Zwar sind bereits mehrere Angriffe auf FALCON publiziert worden, die auf Schwachstellen der Implementierung abzielen, jedoch ist die physische Sicherheit des Verfahrens noch nicht allgemein untersucht worden. Wir stellen eine Analyse der physischen Sicherheit des FALCON Signaturverfahrens vor, die sowohl Angriffe aus früheren Publikationen, als auch neue Schwachstellen enthält und in die die mathematischen Grundlagen des Verfahrens (zum Beispiel die Fast Fourier Transformation, diskrete Normalverteilungen und ein Turm von Körpern) miteinfließen.

Weiters betrachten wir einen der neuen Angriffsvektoren, NarrowSampling, genauer. Wir simulieren einen Angriff, der auf einer Parallelepiped-Lernmethode aufbaut, die auf eine Signaturverteilung mit veringerter Standardabweichung angewendet wird. Jede einzelne Phase des Angriffs wird analysiert und der Einfluss der wichtigsten Angriffsparameter wird gemessen und evaluiert. Für FALCON Parametersets mit reduzierter Sicherheit gelingt es uns, damit den vollständigen privaten Schlüssel zu bestimmen.

Acknowledgments

I want to thank NXP, for the possibility to write this thesis with their collaboration. In particular my supervisor Tobias Schneider, who always provided me with his expertise, helpful insights and discussions, and who would put me back on track whenever I got lost in the details. I also thank my colleagues Olivier Bronchain for helping me when the server did not do what I told it to, Thijs Laarhoven for when I got stuck in the lattices, and Mohamed ElGhamrawy and Melissa Azouaoui for saving me from hours of debugging.

I am especially thankful to Georg Fuchsbauer, my thesis supervisor. He provided a lot of valuable feedback in all of our meetings and encouraged me on every step of the process.

Last but not least, thank you to my family and friends. Although a majority of them have no background in mathematics or cryptography, I could always count on them for choosing tasteful color palettes for figures that most of them did not understand and for all the emotional support on my "quest to find the fuzzy shape". And to all my maths-friends, I promise to stop talking about parallelepipeds (for now).

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am _____

_____
Markus Schönauer

Contents

1 Introduction

Security is one of the most important aspects for the digital infrastructure in today's world. Almost every aspect of it relies, in some way or another, on the security of the underlying processes. In the past, cryptographic standards were based on certain mathematical problems, for example integer factorization, discrete logarithms or elliptic curve discrete logarithms. These problems are *hard* in the sense that it would take a significant amount of computing power and/or time to find solutions, using today's technology and methods.

However, the advent of quantum computers threatens to break many cryptosystems that are widely used today, by enabling new algorithms that can solve previously hard problems. A well-known example is Shor's algorithm for integer factorization [Sho97]. Its efficiency heavily relies on the specific properties of quantum computers that simply cannot be replicated on any classical computer. At present, no quantum computer exists that is capable of running Shor's algorithm (or other, similar algorithms) with enough qubits to actually pose a risk. But the continued efforts in this field mean that it might become a problem in the future.

To prepare for this scenario, in 2016 the National Institute of Standards and Technology (NIST) has called for new cryptographic constructions with two main properties: They should withstand attacks even by quantum adversaries, but still be executable on classical, non-quantum machines. This field of research is called Post-Quantum Cryptography (PQC). After a selection process spanning multiple rounds and several years, the winners are: CYSTALS-Kyber [BDK+17] as key encapsulation mechanism (KEM) on the one hand and CRYSTALS-Dilithium [DLL+17], Falcon [FHK+20] and SPHINCS+ [BHK+19] as digital signature schemes (DS) on the other hand.

Although all of these new cryptosystems are proven to be sufficiently secure in theory (under appropriate hardness assumptions), they can still be vulnerable to physical attacks. It has been shown that it is possible to exploit features and properties of the physical implementation of post-quantum cryptographic algorithms in order to break them. In general, there are two attacker models: an *active* and a *passive* one. A passive adversary performs side-channel analysis (SCA), like measurements of power consumption, runtime etc., to extract information about internal values. An active adversary influences the actual computations, like skipping instructions or injecting faults into the processed data, with the goal of altering the output or behavior of an algorithm. This is called fault injection attack (FA). After enough information is obtained through either SCA or FA, a wide range of statistical and other mathematical methods can be used to recover secret information and break the scheme.

Therefore it is of great importance to not only prove the security of new cryptosystems in the black-box model, where an adversary can only observe the inputs and outputs, but also to study them under the viewpoint of physical security, where it is possible to directly interact with the device that runs the cryptographic calculations. From the NIST winners, CYSTALS-Kyber and CRYSTALS-Dilithium have received the most attention in the community in this regard. There are a lot of publications that describe a variety of attacks and suggest suitable counter-measures. In contrast, Falcon is rather less well researched with regards to side-channel analysis and fault attacks and how to protect against them. (See [KA21] or [MHS+19] for examples.) One of the reasons might be that Falcon differs from the other standards in some major aspects. This is mostly due to two features. Firstly, the special algebraic structure that Falcon heavily relies on for efficiency. It allows for a divide-and-conquer principle that is used in most steps of the scheme, but can be tricky to analyze because of its recursive nature. Secondly, Falcon is the only one of the NIST-winners that uses the Fast Fourier Transform (FFT) for many of its calculations. Operations in the FFT domain necessitate the use of floating-point arithmetic, instead of pure integer arithmetic. This creates new potential vulnerabilities, while at the same time significantly limiting the use of standard counter-measures.

In summary, a detailed understanding of Falcon and its physical vulnerabilities is of great interest for future research and of utmost importance for practical deployment.

1.1 Goals and Results

In this thesis we want to contribute to this goal by providing a thorough sensitivity analysis of the full Falcon cryptosystem, including key generation, signature generation and signature verification, together with all relevant subroutines. The analysis is based in part on a literature study of attacks on Falcon from prior publications. Additionally, we suggest multiple new attack vectors that have as of yet been unexploited. For those we used simulations of faults and side-channel information to illustrate how an attack could be mounted and estimate its plausibility. A complete summary of all vulnerabilities from prior publications and newly found ones is presented in this thesis.

Our second contribution consists of an in-depth description of the "NarrowSampling" attack, which exploits one of the novel attack vectors found during the vulnerability analysis. It is a fault attack targeting the standard deviation of Falcon signatures. We take a combined approach of analyzing the theoretical aspects of the attack, as well as running extensive tests and measurements for various attack parameters. We demonstrate the practicability of full key recovery for reduced security levels of Falcon and draw conclusions for Falcon's real-world parameter sets.

1.2 Structure of the Thesis

This thesis is structured as follows: We start Section 2 with an introduction to the most important theoretical concepts that are needed to understand Falcon itself, as well as many of the attacks in later sections. We go over some basic mathematical definitions that many readers may already be familiar with (such as lattices, the Fast Fourier Transform or the Gram-Schmidt orthogonalization), but are nonetheless integral to a complete understanding of all subsequent sections. We also provide an overview of Falcon's algebraic structure and the divide-and-conquer principle that is based on it. In Section 2.2 we explain the processes of key generation, signature generation and signature verification. Subsequently, in Section 2.3 we go through the main subroutines, where a lot of the mathematical details of Falcon come into play.

Then, in Section 3 we present the results of our sensitivity analysis. We categorize almost every variable in Falcon as either public or sensitive and assess the possibilities of side-channel analysis and fault attacks. Based on the analysis, we list all identified attack vectors in Section 3.6. This includes short summaries of known attacks from prior literature, as well as high-level suggestions on how to potentially exploit new vulnerabilities.

From the new attack vectors we choose one specific example, the fault attack NarrowSampling, for an in-depth analysis in Section 4. We identify the point of interest for the fault injection, simulate faults, show that they have a measurable effect on the signatures and analyze what information is leaked from them (see Section 4.1). Next we describe the Hidden Parallelepiped Problem (HPP), which is the central tool enabling the attack, and show how an adversary can learn an approximation of the secret key by solving the HPP (see Section 4.2). Finally, in Section 4.3, the full key is recovered from the approximation. Throughout the attack analysis we use extensive simulations and tests to gain an understanding of how different parameters affect the attack and how they can be tuned to allow a successful attack within reasonable time and resource constraints. Although we mostly work with low-security parameters of Falcon that are defined for testing purposes, we discuss the relevance of our results and how they might generalize for the full parameter sets in Section 4.4. The last Section 4.5 is dedicated to a brief discussion of countermeasures against the NarrowSampling attack.

The overall conclusion of this thesis can be found in the Section 5, where we summarize our findings and give an outlook on possible future research directions.

2 An Introduction to Falcon

The digital signature scheme Falcon is, in essence, a quantum-resistant instantiation of the GPV framework for digital signatures over NTRU lattices that makes heavy use of the Fast Fourier Transform, with a focus on compact signatures and keys. The acronym Falcon stands for **FA**st Fourier **L**attice-based **CO**mpact signatures over **N**TRU. All of these concepts will be defined and explained in the following sections.

2.1 Preliminaries and Theoretical Background

2.1.1 Notation

For the overview of the Falcon signature scheme we will follow the general notation and naming conventions used in the official specification document [FHK+20], which we will refer to as the Falcon specifications for short.

Scalars and polynomials are denoted with regular lower-case letters. For vectors, which are denoted with bold lower-case letters, we exclusively use row-notation. Matrices are bold upper-case letters and consist of row-vectors. Matrix-matrix and vector-matrix multiplication is usually written without an operation symbol, but sometimes we will use $\cdot$ for clarity. The inner product of two vectors $\mathbf{v}$ and $\mathbf{w}$ is denoted as $\langle \mathbf{v}, \mathbf{w} \rangle$, while the coefficient-wise product (also known as Hadamard product) is denoted as $\mathbf{v} \odot \mathbf{w}$.

We write f_i, $\mathbf{z}_i$ and $\mathbf{B}_{ij}$ for coefficients of polynomials and entries of vectors and matrices. All indices start at zero, with only one exception: the two polynomials comprising a Falcon signature $\mathbf{s} = (s_1, s_2)$. This is to keep in line with the original notation.

If A is a statement that is either true or false, then $\mathbb{P}[A]$ is defined as the probability of A being true. The expected value of a random variable X is denoted as $\mathbb{E}[X]$ and $\mathbb{V}[X]$ is its variance. The normal distribution on $\mathbb{R}$ with mean μ and standard deviation σ is denoted as $\mathcal{N}(\mu, \sigma)$.

We will reserve the notation log for the binary logarithm, while the natural logarithm will be denoted as ln.

A few variable names will have a fixed meaning throughout the whole thesis. Those are:

- The integer modulus q. It has a constant value of $q = 12289$ for all security levels of Falcon.
- The cyclotomic polynomial ϕ used to define polynomial rings such as $\mathbb{Z}[x]/\phi$. It is always of the form $x^n + 1$.
- The polynomial degree n. It exclusively refers to the degree of ϕ and subsequently is the number of coefficients for most polynomials that we work with. Its value is always a power of two.
- The security level κ. It defines the level of security of a specific Falcon instantiation, which goes along with a specific set of parameters. It is directly related to n via $n = 2^\kappa$. Security levels $\kappa = 9$ and $\kappa = 10$ are meant for practical use. They correspond to security levels I and V as defined by NIST [NIS16]. Note that the official names of these two instantiations are Falcon-512 and Falcon-1024, but we will usually give the κ value as reference. For testing purposes the authors of Falcon also define complete parameter sets for all lower security levels $\kappa \in \{1, \ldots, 8\}$. We list them in Table 1.

2.1.2 Basic Definitions

Lattice. The most general definition of a lattice Λ is that of a discrete subgroup of a finite-dimensional Euclidean vector space. In the present setting we will only look at lattices with integer coefficients. For

this, let $\mathbf{B} \in \mathbb{Z}^{d\times k}$ (with $d < k$) be a full rank matrix comprised of row vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{d-1} \in \mathbb{Z}^k$. Then the lattice generated by $\mathbf{B}$ is defined as:

$$\Lambda(\mathbf{B}) = \{\mathbf{xB} : \mathbf{x} \in \mathbb{Z}^d\} = \left\{ \sum_{i=0}^{d-1} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \right\},$$

which is the set of all integer linear combinations of the $\mathbf{b}_i$'s. If the basis is clear from context or not important, we will simply write Λ instead of $\Lambda(\mathbf{B})$. The dimension of the lattice is $\dim \Lambda(\mathbf{B}) = d$.

Just like the basis of a vector space, the basis of a lattice is generally not unique. Let $\mathbf{C} \in \mathbb{Z}^{d\times k}$ be another matrix, then the lattices $\Lambda(\mathbf{B})$ and $\Lambda(\mathbf{C})$ are the same, if and only if there exists a transformation $\mathbf{U} \in \mathbb{Z}^{d\times d}$ with:

$$\mathbf{UC} = \mathbf{B} \quad \text{and} \quad \det(\mathbf{U}) = \pm 1.$$

It is important to note that $\mathbf{U}$ must have integer entries. This guarantees that each row of $\mathbf{B}$ is indeed an integer linear combination of rows of $\mathbf{C}$.

For a given lattice Λ generated by $\mathbf{B} \in \mathbb{Z}^{d\times k}$ we can define the orthogonal lattice mod q as:

$$\Lambda_q^\perp = \{\mathbf{x} \in \mathbb{Z}^d : \mathbf{xB} = 0 \mod q\}.$$

This lattice has the additional property that $q\mathbb{Z}^d \subseteq \Lambda_q^\perp \subseteq \mathbb{Z}^d$. Any lattice with this property is called a q-ary lattice.

NTRU Lattice. A special class of lattices are the NTRU lattices. Let f, g, F and G be polynomials in $\mathbb{Z}[x]/\phi$ that satisfy the so-called NTRU equation:

$$fG - Fg = q, \tag{1}$$

with ϕ (for arbitrary n) and q as defined above. If f is invertible as an element of $\mathbb{Z}_q[x]/\phi$, we can define another polynomial $h \in \mathbb{Z}_q[x]/\phi$ via $h = f^{-1}g \mod q$. Then the matrices $\mathbf{A}$ and $\mathbf{A'}$:

$$\mathbf{A} = \begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix}, \qquad \mathbf{A'} = \begin{bmatrix} f & g \\ F & G \end{bmatrix},$$

generate the same lattice $\Lambda(\mathbf{A}) = \Lambda(\mathbf{A'})$, since $\mathbf{A} = \mathbf{UA'}$ with:

$$\mathbf{U} = \begin{bmatrix} f^{-1} & 0 \\ -F & f \end{bmatrix} \quad \text{and} \quad \det(\mathbf{U}) = 1.$$

We call this lattice an NTRU lattice.

NTRU lattices also come with a hardness assumption: Given only h, it is hard to find f and g, or more generally any two polynomials $f', g' \in \mathbb{Z}[x]/\phi$ with small norm that satisfy $(f')^{-1}g' = h \mod q$.

Parallelepiped. A parallelepiped is the analogue of a parallelogram in an arbitrary dimension d. Let $\mathbf{B} \in \mathbb{R}^{d\times d}$ be a matrix with row-vectors $\mathbf{b}_i$. We define the parallelepiped spanned by $\mathbf{B}$ as:

$$\mathcal{P}(\mathbf{B}) = \left\{ \sum_{i=0}^{d-1} x_i \mathbf{b}_i : x_i \in [-1, 1] \right\}.$$

A parallelepiped can be scaled and translated, for example:

$$\frac{1}{2}\mathcal{P}(\mathbf{B}) + \frac{1}{2}\sum_{i=0}^{d-1} \mathbf{b}_i = \left\{ \sum_{i=0}^{d-1} x_i \mathbf{b}_i : x_i \in [0, 1] \right\},$$

is a similar parallelepiped to the one above, but its sides are half the length and it is translated such that one corner lies in the origin.

Gram-Schmidt Orthogonalization. This is a standard method in linear algebra for constructing an orthogonal (or even orthonormal) basis of a vector space, given an arbitrary basis of the same space. It comes up in FALCON itself and will also play an important role later in the analysis of one of the attacks, so we want to give a brief description here. For our purposes it suffices to look at the version of the procedure that produces an orthogonal basis (as opposed to an orthonormal one).

Let $\mathbf{B} \in \mathbb{R}^{d \times d}$ be a full-rank matrix. We can interpret the row-vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{d-1}$ as basis vectors of the vector space $\mathbb{R}^d$. In general, the $\mathbf{b}_i$'s are not orthogonal to each other. If we define:

$$\tilde{\mathbf{b}}_0 = \mathbf{b}_0,$$
$$\tilde{\mathbf{b}}_i = \mathbf{b}_i - \sum_{k=0}^{i-1} \frac{\langle \tilde{\mathbf{b}}_k, \mathbf{b}_i \rangle}{\langle \tilde{\mathbf{b}}_k, \tilde{\mathbf{b}}_k \rangle} \tilde{\mathbf{b}}_k, \text{ for } 1 \le i < d,$$

then the $\tilde{\mathbf{b}}_i$'s are a new basis of $\mathbb{R}^d$ where $\langle \tilde{\mathbf{b}}_i, \tilde{\mathbf{b}}_j \rangle = 0$ for all $i \neq j$. The resulting vectors will be called the Gram-Schmidt vectors of $\mathbf{B}$ and can be summarized as rows of an orthogonal matrix $\tilde{\mathbf{B}}$. We will sometimes also use $\mathsf{GSO}(\mathbf{B})$ to denote the Gram-Schmidt matrix of a matrix $\mathbf{B}$.

Geometrically speaking, the term $\frac{\langle \tilde{\mathbf{b}}_k, \mathbf{b}_i \rangle}{\langle \tilde{\mathbf{b}}_k, \tilde{\mathbf{b}}_k \rangle} \tilde{\mathbf{b}}_k$ describes the part of $\mathbf{b}_i$ that lies in the subspace spanned by $\tilde{\mathbf{b}}_k$. So by subtracting these terms for $k < i$ from $\mathbf{b}_i$, we are left with a vector $\tilde{\mathbf{b}}_i$ that is orthogonal to all previous $\tilde{\mathbf{b}}_k$'s. This can be shown formally with a simple calculation of $\langle \tilde{\mathbf{b}}_i, \tilde{\mathbf{b}}_k \rangle = 0$.

Another useful property of the Gram-Schmidt process is the following. Define H_i to be the subspace spanned by the first $i+1$ row-vectors $\mathbf{b}_0, \ldots, \mathbf{b}_i$. It can be shown that the first $i+1$ Gram-Schmidt vectors $\tilde{\mathbf{b}}_0, \ldots, \tilde{\mathbf{b}}_i$ are an orthogonal basis of the same subspace H_i.

Gram-Schmidt Norm. Based on the Gram-Schmidt orthogonalization we can define a matrix norm for full-rank square matrices. Let $\mathbf{B} \in \mathbb{R}^{d \times d}$ be such a matrix and let $\tilde{\mathbf{b}}_i$ be its Gram-Schmidt vectors. Then the Gram-Schmidt norm of $\mathbf{B}$ is defined as the length of the longest Gram-Schmidt vector:

$$\|\mathbf{B}\|_{\mathrm{GS}} = \max_{0 \le i < d} \|\tilde{\mathbf{b}}_i\|.$$

Number Theoretic Transform. We define the number theoretic transform (NTT) in the context of FALCON. Consider the polynomial $\phi = x^n + 1$ as an element of $\mathbb{Z}_q[x]$ (instead of $\mathbb{Z}[x]$). With the choice of $q = 12289$, $n = 2^\kappa$, and $\kappa \le 10$ it can be shown that ϕ has n distinct roots in $\mathbb{Z}_q$. Let $(\omega_i)_{0 \le i < n}$ be those roots, then the NTT of a polynomial $f \in \mathbb{Z}_q[x]/\phi$ is defined as:

$$\mathsf{NTT}(f) = \big(f(\omega_i)\big)_{0 \le i < n},$$

i.e., the n-tuple of f evaluated at the roots ω_i in $\mathbb{Z}_q$. It can be shown that the NTT is a bijection between $\mathbb{Z}_q[x]/\phi$ and $\mathbb{Z}_q^n$.

One useful feature of the NTT is that it allows us to easily detect whether a polynomial is invertible (as an element of $\mathbb{Z}_q[x]/\phi$) or not. To see this, we use the fact that polynomial multiplication in $\mathbb{Z}_q[x]/\phi$ maps to coefficient-wise multiplication in the NTT domain. Let $f, g \in \mathbb{Z}_q[x]/\phi$ and let $\hat{f}_i, \hat{g}_i$ be their NTT coefficients. Then we have:

$$\mathsf{NTT}(f \cdot g) = \mathsf{NTT}(f) \odot \mathsf{NTT}(g) = (\hat{f}_i \cdot \hat{g}_i)_{0 \le i < n}.$$

Now if we take $g = f^{-1} \bmod q$ we get:

$$f \cdot g = 1 \quad \Rightarrow \quad \mathsf{NTT}(f \cdot g) = \mathsf{NTT}(1) \quad \Rightarrow \quad (\hat{f}_i \cdot \hat{g}_i)_{0 \le i < n} = (1, \ldots, 1) \quad \Rightarrow \quad \hat{g}_i = \frac{1}{\hat{f}_i}.$$

So in order for the inverse f^{-1} to exist, all $\hat{f}_i$ must be invertible mod q. In other words, the NTT of f must not contain any zeros.

Discrete Gaussian Distribution. An analogue of the normal distribution for the discrete set of integers $\mathbb{Z}$ is the discrete Gaussian distribution. For this we define the function $\rho_{\sigma,\mu}(z) = \exp(-|z-\mu|^2/2\sigma^2)$ for all $z \in \mathbb{Z}$. Now the discrete Gaussian distribution is defined via its probability mass function (PMF):

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

Similarly to the normal distribution, μ and σ are the mean and standard deviation of the discrete Gaussian distribution.

We can also generalize this definition to a spherical discrete Gaussian distribution on a d-dimensional lattice Λ:

$$\rho_{\sigma,\boldsymbol{\mu}}(\mathbf{z}) = e^{\frac{-\|\mathbf{z}-\boldsymbol{\mu}\|^2}{2\sigma^2}}, \qquad D_{\Lambda,\sigma,\boldsymbol{\mu}}(\mathbf{z}) = \frac{\rho_{\sigma,\boldsymbol{\mu}}(\mathbf{z})}{\sum_{\mathbf{x}\in\Lambda} \rho_{\sigma,\boldsymbol{\mu}}(\mathbf{x})}.$$

Here, $\mathbf{z} \in \Lambda$ are points on the lattice, $\boldsymbol{\mu} \in \mathbb{R}^d$ is the mean (note that the mean does not need to be a lattice point itself), and σ is again the standard deviation. When $\boldsymbol{\mu} = 0$ (or $\mu = 0$ in the one-dimensional case) we say that the distribution is centered and may omit $\boldsymbol{\mu}$ (or μ) in the notation, e.g., $D_{\mathbb{Z},\sigma,0} = D_{\mathbb{Z},\sigma}$.

A related distribution is the discrete half-Gaussian distribution on the integers, $D_{\mathbb{Z}^+,\sigma}$. It can be thought of as the upper half of the "full" distribution $D_{\mathbb{Z},\sigma,0}$. The definition is similar, but the argument z only ranges in the non-negative integers $\mathbb{Z}^+ = \{0, 1, 2, \dots\}$. Note that σ is not the standard deviation of $D_{\mathbb{Z}^+,\sigma}$, but rather the standard deviation of the corresponding full distribution $D_{\mathbb{Z},\sigma,0}$.

Chi-squared Distribution. Let Z_i, $0 \le i < d$ be d independent, identically distributed random variables with $Z_i \sim \mathcal{N}(0,1)$. Define a random variable Q as:

$$Q = \sum_{i=0}^{d-1} Z_i^2.$$

The resulting distribution is called the chi-squared distribution with d degrees of freedom, $Q \sim \chi^2_d$.

2.1.3 CDT-Sampler

A CDT-sampler is a method to generate random samples by transforming uniform samples into other distributions. This kind of sampler is used as part of Falcon [HPRR19], [PRR19] and some attacks specifically target its functionality [KH18]. It requires a precomputed table containing the values of a cumulative distribution function (CDF), hence the name cumulative distribution table, CDT.

The theory behind CDT-samplers is called inverse transform sampling and is used to transform samples from a uniform distribution (which is usually easy to sample from) into other, more complicated distributions: Let X be a random variable following some distribution $\mathcal{X}$ with cumulative distribution function $F(x) = \mathbb{P}[X \le x]$ and let $U \sim \mathcal{U}[0,1]$ be a uniform random variable on the closed unit interval. It can be shown that[1]:

$$F^{-1}(U) \sim \mathcal{X},$$

meaning that the transformed U is distributed according to the distribution $\mathcal{X}$ of X. We can also apply this to individual samples u of U and transform them into samples x from X via $x = F^{-1}(u)$.

In Falcon, this method is slightly adapted to work with integer values only and to avoid evaluating the inverse CDF explicitly. Suppose we want to generate samples from a discrete distribution on $\{0, \dots, m\}$ given by its CDF $F(x)$. For this we prepare a table of values $1 - F(i)$ with $0 \le i \le m$. To avoid floating-points we scale the values by some (sufficiently large) integer c and round to integers: $\lfloor c \cdot (1 - F(i)) \rceil$.

[1] Not all distributions have invertible CDFs. In these cases a generalized notion of inverse has to be used, which we will not discuss here.

Now we can proceed as follows: Take a sample u from the discrete uniform distribution on $[0, c-1]$. Go through the prepared list and increment a counter whenever $u < \lfloor c \cdot (1 - F(i)) \rfloor$. The eventual value of the counter is then returned.

It can be shown that this process yields samples that are distributed according to the distribution specified by $F(x)$. Without going into detail it should be apparent that the scaling value c and the exact way of rounding $\lfloor c \cdot (1 - F(i)) \rfloor$ effects how well the generated samples follow the target distribution.

2.1.4 The GPV Framework and Trapdoor Samplers

First introduced by Gentry, Peikert and Vaikuntanathan [GPV07], the GPV framework is the blueprint for a class of lattice-based signature schemes. Since FALCON is based on a specific instantiation of this framework, we will give a short description of it.

The secret key $\mathbf{B} \in \mathbb{Z}_q^{m \times m}$ and the public key $\mathbf{A} \in \mathbb{Z}_q^{d \times m}$ (with $d < m$) are matrices whose rows generate two lattices $\Lambda(\mathbf{B})$ and $\Lambda(\mathbf{A})$, respectively. These matrices must meet the additional requirement that the lattices are orthogonal mod q to each other, i.e.:

$$\forall \mathbf{x} \in \Lambda(\mathbf{B}), \forall \mathbf{y} \in \Lambda(\mathbf{A}) : \langle \mathbf{x}, \mathbf{y} \rangle = 0 \mod q, \qquad \text{or equivalently:} \qquad \mathbf{B}\mathbf{A}^\top = 0 \mod q.$$

This guarantees that for any point $\mathbf{v} \in \Lambda(\mathbf{B})$ we have: $\mathbf{v}\mathbf{A}^\top = 0 \mod q$. To see this, note that the columns of $\mathbf{A}^\top$ are just the rows of $\mathbf{A}$, i.e., the vectors that span $\Lambda(\mathbf{A})$. Let $\mathbf{a}_0, \ldots, \mathbf{a}_{d-1}$ be these row vectors, then we have:

$$\mathbf{v}\mathbf{A}^\top = \big(\langle \mathbf{v}, \mathbf{a}_0 \rangle, \ldots, \langle \mathbf{v}, \mathbf{a}_{d-1} \rangle\big) = (0, \ldots, 0) \mod q.$$

All inner products vanish, because $\mathbf{v} \in \Lambda(\mathbf{B})$ and $\mathbf{a}_i \in \Lambda(\mathbf{A})$.

A valid signature for a message m is defined as a short (i.e., shorter than some predetermined bound) vector $\mathbf{s} \in \mathbb{Z}_q^m$ satisfying:

$$\mathbf{s}\mathbf{A}^\top = H(\mathsf{m}), \tag{2}$$

where H is a publicly specified, deterministic hash function.

To generate a signature for a given message m, it is first hashed to a point $\mathbf{c} = H(\mathsf{m}) \in \mathbb{Z}_q^d$. Then a preimage $\mathbf{c}_0 \in \mathbb{Z}_q^m$ under $\mathbf{A}^\top$ is computed, such that $\mathbf{c}_0\mathbf{A}^\top = \mathbf{c}$, which can be done with standard linear algebra. Note that $\mathbf{c}_0$ will in general not be short enough to be used as a signature directly. This is by design, since otherwise anyone who has knowledge of the public key could create signatures for arbitrary messages. Instead we need to make use of a trapdoor, i.e., some additional information to make a computationally hard problem practically solvable. In the present case the trapdoor is the secret matrix $\mathbf{B}$. It allows us to find some $\mathbf{v} \in \Lambda(\mathbf{B})$ that is close to the preimage $\mathbf{c}_0$. (The exact method can vary and will be explained later.) Once a suitable vector is found, we can shorten the preimage and define the signature as the difference $\mathbf{s} = \mathbf{c}_0 - \mathbf{v} \in \mathbb{Z}_q^m$. Since $\mathbf{v}$ is close to $\mathbf{c}_0$, the signature is now a short vector.

Verification is then relatively straightforward. First, the verifier has to check that the received signature is indeed shorter than a set bound. Then the message is hashed to $\mathbf{c}$ and Equation 2 can be verified with the knowledge of the public key alone:

$$\mathbf{s}\mathbf{A}^\top = (\mathbf{c_0} - \mathbf{v})\mathbf{A}^\top = \mathbf{c}_0\mathbf{A}^\top - \mathbf{v}\mathbf{A}^\top = \mathbf{c} - 0 = H(\mathsf{m}).$$

In the second to last equality $\mathbf{v}\mathbf{A}^\top$ vanishes, because $\mathbf{v}$ is a point on $\Lambda(\mathbf{B})$, the orthogonal lattice to $\Lambda(\mathbf{A})$.

One important aspect when instantiating the GPV framework is the choice of the trapdoor sampler. The term refers to the part of the scheme that performs the task of finding $\mathbf{v}$ as described above. A more general definition is the following:

A function that takes a matrix $\mathbf{A}$, a target $\mathbf{c}$ and a trapdoor T as input and calculates either a short vector $\mathbf{s}$ with $\mathbf{s}\mathbf{A}^\top = \mathbf{c}$ or alternatively a vector $\mathbf{v} \in \Lambda_q^\perp$ close to $\mathbf{c}$. Following our expositions, both of these tasks are equivalent. Also note that the trapdoor does not need to be a matrix $\mathbf{B}$ like in the example above. Rather it can be any information that makes the search for a suitable $\mathbf{s}$ or $\mathbf{v}$ computationally feasible.

The way in which the trapdoor sampler is instantiated can have a lot of influence on the overall security of the scheme. It is possible that the signatures can leak information about the secret key if the sampling is done improperly. We will explain this in more detail in later sections.

2.1.5 The Algebraic Structure of Falcon

There are two main algebraic concepts that Falcon is based on. The first one is a correspondence between polynomials on the one hand and vectors and matrices on the other hand. The second one concerns a tower of fields (rings, resp.), together with a chain of vector space isomorphisms (module isomorphisms, resp.).

Polynomials, Vectors and Matrices. In the following we define the operators Mat and Vec that allow us to switch between two viewpoints: polynomials on the one hand and matrices on the other hand. Although they are never explained in detail in the original Falcon specifications [FHK+20], the two viewpoints are prevalent throughout the document, so a formal definition seems useful.

Consider the field $\mathbb{Q}[x]/\phi$ and its subring $\mathbb{Z}[x]/\phi$, with $\phi = x^n + 1$ and $n = 2^\kappa$ a power of two. These are the main algebraic structures where most of the calculations in Falcon take place. Their elements are polynomials of degree $n-1$ with n rational (resp. integer) coefficients.[2] Per the usual definition, the addition and multiplication in $\mathbb{Q}[x]/\phi$ are performed mod ϕ, so we will omit this information from the notation most of the time.

One of the main features of this field is that it allows us to identify polynomials with vectors and matrices and translate addition and multiplication in $\mathbb{Q}[x]/\phi$ to corresponding operations involving matrices and vectors. On the one hand, polynomials can be identified most naturally with their coefficient vectors:

$$a = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Q}[x]/\phi \qquad \rightsquigarrow \qquad \mathrm{Vec}(a) := (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Q}^n.$$

On the other hand, there is no canonical way to map a polynomial to a matrix. For our purposes, the most useful way to do this is as a Toeplitz matrix of the following form:

$$a = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Q}[x]/\phi \qquad \rightsquigarrow \qquad \mathrm{Mat}(a) := \begin{bmatrix} a_0 & a_1 & & \cdots & a_{n-1} \\ -a_{n-1} & a_0 & a_1 & \cdots & a_{n-2} \\ & -a_{n-1} & a_0 & & \vdots \\ \vdots & \vdots & & \ddots & a_1 \\ -a_1 & -a_2 & \cdots & -a_{n-1} & a_0 \end{bmatrix} \in \mathbb{Q}^{n\times n}.$$

If we define two operators $\mathrm{Vec} : \mathbb{Q}[x]/\phi \to \mathbb{Q}^n$ and $\mathrm{Mat} : \mathbb{Q}[x]/\phi \to \mathbb{Q}^{n\times n}$ that transform polynomials into vectors and matrices according to the formulas above, we can make a few observations (which can all be shown by direct calculations). Let a, b, c be polynomials in $\mathbb{Q}[x]/\phi$, satisfying $ab = c$. Then we have:

$$\mathrm{Mat}(a) \cdot \mathrm{Mat}(b) = \mathrm{Mat}(c) \qquad \text{and} \qquad \mathrm{Vec}(a) \cdot \mathrm{Mat}(b) = \mathrm{Vec}(c).$$

This allows us to view most calculations in Falcon either in the context of polynomials or in that of vectors and matrices.

An alternative definition (or rather a useful property) of the Mat operator comes from the following fact:

$$a = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Q}[x]/\phi \qquad \rightsquigarrow \qquad a \cdot x = -a_{n-1} + \sum_{i=0}^{n-2} a_i x^{i+1}.$$

[2] To be precise, the elements of $\mathbb{Q}[x]/\phi$ and $\mathbb{Z}[x]/\phi$ are equivalence classes which we identify with their respective unique representative of that form. Going forward, this distinction is not too important and so we will regard the elements of these fields and rings simply as polynomials.

In other words, when we multiply a polynomial a with the polynomial $x = 0 + 1x + 0x^2 + \ldots$, all coefficients are shifted up by one and the former highest coefficient is negated and becomes the constant coefficient. This is a direct consequence of the equality $x^n \bmod (x^n + 1) = -1$. We immediately see that the i-th row of $\mathrm{Mat}(a)$ is just $\mathrm{Vec}(a \cdot x^i)$, and so:

$$\mathrm{Mat}(a) = \begin{bmatrix} \mathrm{Vec}(a) \\ \mathrm{Vec}(a \cdot x) \\ \ldots \\ \mathrm{Vec}(a \cdot x^{n-1}) \end{bmatrix}.$$

Adjoint Polynomials. In general, the adjoint polynomial $a^\star$ of $a \in \mathbb{Q}[x]/\phi$ is defined via some equality involving the complex roots of the polynomial modulus ϕ and complex conjugation. For Falcon's specific choice of $\phi = x^n + 1$ and n a power of two, the definition simplifies to a formula that only permutes the original coefficients and negates some of them:

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

Crucially, the coefficient vector of $a^\star$ is exactly the first column of $\mathrm{Mat}(a)$ and thanks to its cyclical structure we even have:

$$\mathrm{Mat}(a^\star) = \mathrm{Mat}(a)^\top.$$

Adjoining polynomials is compatible with polynomial operations as one would reasonably expect. In particular, for any $a, b \in \mathbb{Q}[x]/\phi$ we have:

$$(a+b)^\star = a^\star + b^\star, \qquad (ab)^\star = a^\star b^\star, \qquad (a^\star)^\star = a.$$

All three properties can easily be derived from the definition itself and with the help of the Mat operator.

We can extend the definition of adjoint polynomials to matrices whose entries are polynomials of $\mathbb{Q}[x]/\phi$. For example, consider $\mathbf{M} \in (\mathbb{Q}[x]/\phi)^{2\times 2}$ consisting of four polynomials a, b, c, d. The adjoint $\mathbf{M}^\star$ is defined as the transpose of the matrix with individually adjoint entries $a^\star, b^\star, c^\star, d^\star$:

$$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \rightsquigarrow \quad \mathbf{M}^\star = \begin{bmatrix} a^\star & c^\star \\ b^\star & d^\star \end{bmatrix}$$

If we extend the Mat operator in a similar way by defining $\mathrm{Mat}(\mathbf{M})$ as the application of Mat to each entry of $\mathbf{M}$, then those two extended definitions are compatible in the sense that:

$$\begin{aligned} \mathrm{Mat}(\mathbf{M}^\star) &= \begin{bmatrix} \mathrm{Mat}(a^\star) & \mathrm{Mat}(c^\star) \\ \mathrm{Mat}(b^\star) & \mathrm{Mat}(d^\star) \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{Mat}(a)^\top & \mathrm{Mat}(c)^\top \\ \mathrm{Mat}(b)^\top & \mathrm{Mat}(d)^\top \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{Mat}(a) & \mathrm{Mat}(b) \\ \mathrm{Mat}(c) & \mathrm{Mat}(d) \end{bmatrix}^\top = \mathrm{Mat}(\mathbf{M})^\top. \end{aligned}$$

Tower of Fields. Consider again the field of polynomials $\mathbb{Q}[x]/\phi$, with $\phi = x^n + 1$ and n a power of two. The field $\mathbb{Q}[x]/\phi'$, with $\phi' = x^{n/2} + 1$, contains polynomials with half the number of coefficients and has a structure similar to the larger field. We can define an operator that maps elements of the larger field to pairs of elements of the smaller one:

$$\begin{aligned} \mathsf{split}: \qquad & \mathbb{Q}[x]/\phi \to \left(\mathbb{Q}[x]/\phi'\right)^2, \\ & f = \sum_{i=0}^{n-1} f_i x^i \mapsto \left(\sum_{i=0}^{n/2-1} f_{2i} x^i, \sum_{i=0}^{n/2-1} f_{2i+1} x^i \right) = (f_0, f_1). \end{aligned}$$

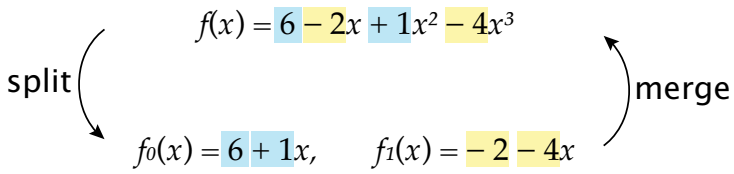

Figure 1: Example for the split and merge operators between the fields $\mathbb{Q}[x]/(x^4+1)$ and $\mathbb{Q}[x]/(x^2+1)$.

This operator splits a polynomial into two polynomials of lower degree[3] that contain the even and odd coefficients, respectively. We can also define a reverse operator that recombines f_0 and f_1 into f by alternating their coefficients:

$$\begin{aligned}\mathsf{merge} : \left(\mathbb{Q}[x]/\phi'\right)^2 &\to \mathbb{Q}[x]/\phi,\\ (f_0, f_1) &\mapsto f_0(x^2) + x f_1(x^2) = f.\end{aligned}$$

The substitution $x \rightsquigarrow x^2$ spaces out the coefficients of f_0 resp. f_1, multiplication of $f_1(x^2)$ with x shifts the (originally odd) coefficients one index over and addition combines them into the original f.

All of the above describes a connection between $\mathbb{Q}[x]/(x^n+1)$ and $\mathbb{Q}[x]/(x^{n/2}+1)$. Since n is a power of two, we can continue this chain down to smaller and smaller fields until we arrive at $\mathbb{Q}[x]/(x^1+1) \cong \mathbb{Q}$, a field containing only constant rational polynomials:

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

For every step in this tower of fields we can define analogues of split and merge. We will use the same notation for all of these functions, regardless of the degree of the field they operate on.

Importantly, if we view these fields as vector spaces, then both split and merge are isomorphisms between the respective spaces they connect. So we get this chain of space isomorphisms:

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

Note that the same holds true for the integer ring $\mathbb{Z}[x]/\phi$ and its subrings, we only have to replace the terms *field* with *ring* and *vector space* with *module*. This algebraic property of $\mathbb{Q}[x]/\phi$ and $\mathbb{Z}[x]/\phi$ gives rise to a divide-and-conquer principle that Falcon uses extensively and that we will explore later on in the description of the scheme (see Sections 2.2 and 2.3).

It is also possible to describe the tower and the isomorphisms from the vector point of view. Let f be a polynomial in $\mathbb{Q}[x]/\phi$ and let $\mathrm{Vec}(f)$ be is its coefficient vector. Now, $\mathrm{Vec}(\mathsf{split}(f))$ contains the same coefficients, just in a different order (first the even coefficients, followed by the odd ones). Going further down the tower, each step corresponds to a new permutation of the same original coefficients, see Figure 2. Similarly, $\mathrm{Mat}(\mathsf{split}(f))$ will give a permuted version of $\mathrm{Mat}(f)$, where the specific permutation is also determined by the position in the tower of fields (resp. rings). The Mat and Vec operators also retain their properties of mapping polynomial arithmetic to matrix and vector operations. For a more in-depth look at the interplay between the tower of fields and the polynomial/matrix duality see [DP15].

2.1.6 Fast Fourier Transform

In Falcon the Fast Fourier Transform (FFT) is used to represent polynomials in $\mathbb{Q}[x]/\phi$ as a vector of complex numbers. The polynomial modulus $\phi = x^n + 1$ has n distinct roots in $\mathbb{C}$. Let ζ_i be these roots

[3] In order to keep in line with the notation from the original Falcon specifications, we use the notation f_0 and f_1 for the split polynomials in this context. It should not be confused with the same notation for the coefficients of f with indices 0 and 1.

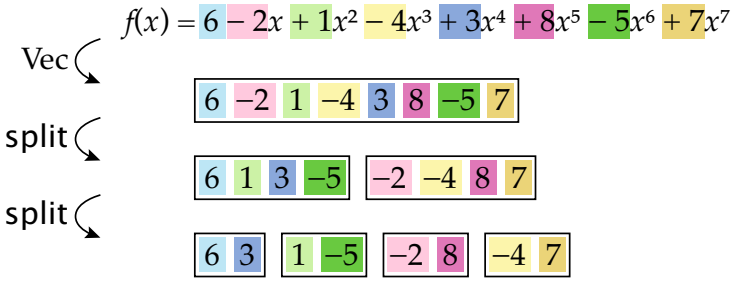

Figure 2: Example of repeated splitting of a polynomial in vector notation. Note that one more additional split in the end would result in single-coefficient polynomials, but not change the order of coefficients any more.

and let $f \in \mathbb{Q}[x]/\phi$ be another polynomial, then the FFT of f is defined as:

$$\mathsf{FFT}(f) = \big(f(\zeta_i)\big)_{0 \le i < n},$$

i.e., the n-tuple of f evaluated at the roots ζ_i in $\mathbb{C}$. This is a very similar situation to the NTT and the theory behind both transforms is closely related. We will, however, not go into detail about this relation.

To emphasize that a polynomial is in FFT representation, we will use $\hat{\cdot}$ as notation. Note that this notation is not consistent in the FALCON specification [FHK+20]. So a polynomial without $\hat{\cdot}$ could either be in coefficient or FFT representation. With the $\hat{\cdot}$ however, it is always in the FFT domain.

The inverse of the FFT transform is denoted as:

$$\mathsf{invFFT}(\hat{f}) = f,$$

which is essentially a polynomial interpolation of the the points $\big(\zeta_i, f(\zeta_i)\big)$.

One of the advantages of using FTT representation is the efficiency increase for multiplication and division. Let $\hat{f} = (\hat{f}_0, \ldots, \hat{f}_{n-1})$ and $\hat{g} = (\hat{g}_0, \ldots, \hat{g}_{n-1})$ be two polynomials in FFT representation. From $(f \cdot g)(x) = f(x) \cdot g(x)$ we get $\mathsf{FFT}(f \cdot g) = \hat{f} \odot \hat{g}$ and so the product $f \cdot g$ can be computed coefficient-wise:

$$f \cdot g = \mathsf{invFFT}\big(\mathsf{FFT}(f \cdot g)\big) = \mathsf{invFFT}(\hat{f} \odot \hat{g}) = \mathsf{invFFT}(\hat{f}_0 \cdot \hat{g}_0, \ldots, \hat{f}_{n-1} \cdot \hat{g}_{n-1}).$$

The same holds for division. Addition and Subtraction are also coefficient-wise, just like in coefficient representation.

We will also use the FFT operator on vectors and matrices containing polynomials. Then it is understood to operate on an entry-wise level.

The algebraic structure described in the previous Section 2.1.5 can be translated into the FFT domain. For this we define two operators, called splitfft and mergefft, that perform the same task of splitting and merging polynomials, but in FFT representation.[4] Importantly, they are compatible with split, merge and FFT:

$$\mathsf{FFT}\big(\mathsf{split}(f)\big) = (\hat{f}_0, \hat{f}_1) = \mathsf{splitfft}\big(\mathsf{FFT}(f)\big),$$
$$\mathsf{invFFT}\big(\mathsf{mergefft}(\hat{f}_0, \hat{f}_1)\big) = f = \mathsf{merge}\big(\mathsf{invFFT}(\hat{f}_0, \hat{f}_1)\big).$$

Here, $\hat{f}_0$ and $\hat{f}_1$ denote the two output polynomials of split in the FFT domain, not the individual FFT coefficients of f.

[4] An algorithmic definition is given in the FALCON specifications [FHK+20].

2.1.7 Babai's Nearest Plane Algorithm

Say we have a full-rank matrix $\mathbf{B} \in \mathbb{Z}^{d \times d}$ that generates the lattice $\Lambda = \Lambda(\mathbf{B})$. We are also given a point $\mathbf{t} \in \mathbb{R}^d$, the so-called target, which does not necessarily lie on the lattice Λ. The goal is to find a point $\mathbf{v} \in \Lambda$ that is close to the target. In other words, $\|\mathbf{v} - \mathbf{t}\|$ should be small. For now, we omit a more formal definition on what *small* means.

One solution to this problem is the Nearest Plane Algorithm, first described by Babai [Bab86]. Consider the $(d-1)$-dimensional hyperplane H_{d-1} spanned by the the first $d-1$ rows of $\mathbf{B}$. From the Gram-Schmidt orthogonalization we know that the row-vectors $\tilde{\mathbf{b}}_0, \ldots, \tilde{\mathbf{b}}_{d-2}$ of $\tilde{\mathbf{B}} = \mathsf{GSO}(\mathbf{B})$ are an orthogonal basis of the same hyperplane H_{d-1}. The space $\mathbb{R}^d$ can now be partitioned into disjoint, affine copies of this hyperplane:

$$\mathbb{R}^d = \bigcup_{\lambda \in \mathbb{R}} H_{d-1} + \lambda \tilde{\mathbf{b}}_{d-1}.$$

The copies are offset from H_{d-1} by some scalar multiple of the last Gram-Schmidt vector $\tilde{\mathbf{b}}_{d-1}$. The idea now is to find the unique hyperplane (defined by the corresponding coefficient λ) that contains the target $\mathbf{t}$. If we write down $\mathbf{t}$ as a linear combination of the $\tilde{\mathbf{b}}_i$'s we get:

$$\mathbf{t} = \sum_{i=0}^{d-1} \frac{\langle \mathbf{t}, \tilde{\mathbf{b}}_i \rangle}{\langle \tilde{\mathbf{b}}_i, \tilde{\mathbf{b}}_i \rangle} \tilde{\mathbf{b}}_i.$$

From this it is easy to see that the terms for $i = 0$ up to $i = d-2$ lie in H_{d-1} (they are a linear combination of $\tilde{\mathbf{b}}_0$ up to $\tilde{\mathbf{b}}_{d-2}$), so the last term for $i = d-1$ must be equal to $\lambda \tilde{\mathbf{b}}_{d-1}$, and we can set:

$$\lambda = \frac{\langle \mathbf{t}, \tilde{\mathbf{b}}_{d-1} \rangle}{\langle \tilde{\mathbf{b}}_{d-1}, \tilde{\mathbf{b}}_{d-1} \rangle}.$$

Unfortunately, λ will usually not be an integer. But we can simply round λ to get the closest hyperplane that does contain lattice points.

Similar to how H_{d-1} contains a copy of the sublattice generated by $\mathbf{b}_0, \ldots, \mathbf{b}_{d-2}$, the chosen hyperplane contains a shifted version of this sublattice. We can repeat the procedure from above inside this hyperplane, always finding the closest integer-shifted subspace to $\mathbf{t}$. Eventually we reach the 0-dimensional case, i.e. a single point $\mathbf{v}$, which will be the output of the algorithm.

The Nearest Plane algorithm has an interesting geometric property. Babai showed [Bab86] that for a lattice $\Lambda(\mathbf{B})$ and a target $\mathbf{t}$, it holds that $\mathbf{v} - \mathbf{t}$ lies in the parallelepiped $\frac{1}{2}\mathcal{P}(\mathsf{GSO}(\mathbf{B}))$. Informally speaking, this is because the coefficients of $\mathbf{t}$ are "rounded in the direction of the $\tilde{\mathbf{b}}_i$'s".

2.2 High-Level Overview of Falcon's Main Algorithms

Throughout the next three Sections 2.2.1, 2.2.2 and 2.2.3, we give an overview of the main components of Falcon. As mentioned before, Falcon is an instantiation of the GPV framework and we will reference how the steps in Keygen (Algorithm 1), Sign (Algorithm 2) and Verify (Algorithm 3) correspond to GPV specifically.

2.2.1 Description of Keygen

In Falcon, the generation of a key pair starts with a call to NTRUGen, which produces the main components of the secret key: four polynomials $f, g, F, G \in \mathbb{Z}[x]/\phi$. The exact process is described in Section 2.3.1. In short, the two polynomials f and g are sampled randomly and f is checked to be invertible mod q (i.e., invertibility as an element of $\mathbb{Z}_q[x]/\phi$). To complete the secret key, two more polynomials $F, G \in \mathbb{Z}[x]/\phi$ are calculated, such that the NTRU equation is satisfied:

$$fG - Fg = q \mod \phi.$$

For this, a variant of the extended Euclidean algorithm is used that leverages the divide-and-conquer principle of the ring $\mathbb{Z}[x]/\phi$.

Since f and g are generated first, we will sometimes refer to them as the primary key components, while F and G will be called the secondary key components. Together they make up the matrix $\mathbf{B}$ that directly corresponds to the matrix of the same name in the GPV framework:

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

In Falcon however, this is not the complete secret key. For efficiency reasons it also contains a Falcon tree T that encodes essentially the same information as $\mathbf{B}$, but in a format that allows for faster signature generation. The computation of the tree T will be explained in Section 2.3.2. It is based on a recursive LDL*-decomposition of the Gram matrix $\mathbf{G} = \hat{\mathbf{B}}\hat{\mathbf{B}}^\star$, where $\hat{\mathbf{B}}$ is the entry-wise FFT of $\mathbf{B}$. After the initial tree is computed, it is normalized using a parameter σ. This is an essential step for the security of the scheme, as σ controls the distribution of the signatures that will be generated from the tree. Now we can combine $\hat{\mathbf{B}}$ and T into the secret key sk.

Lastly, the public key pk consists solely of the polynomial $gf^{-1} \mod q = h \in \mathbb{Z}_q[x]/\phi$. This calculation is the reason why the invertibility of f had to be checked.

Note that in implementations, the memory requirements for the matrix $\hat{\mathbf{B}}$ can be reduced by only storing three polynomials and recomputing the fourth with polynomial arithmetic when needed. In Section 3.2.3 we will see that, in fact, the complete matrix can be recomputed from any single one of the polynomials with additional knowledge of the public key. This could be used as an even more compact secret key format, but it would be computationally inefficient as it would require to run $\mathsf{NTRUSolve}$ (see Section 2.3.1) again to unpack the full key.

It might not be immediately clear how Falcon's public key corresponds to the public key from the GPV framework. But using the polynomial h we can define the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & h^\star \end{bmatrix} \in \left(\mathbb{Z}[x]/\phi\right)^{1\times 2}.$$

The following calculation shows that, indeed, $\mathbf{B}\mathbf{A}^\star = 0 \mod q$, as is required by the GPV framework. For this, we start with the NTRU equation, multiply it with f^{-1} and consider the result mod q:

$$\begin{aligned} fG - Fg &= q, \\ f^{-1}fG - Fgf^{-1} &= q, \\ G - Fh &= 0 \mod q. \end{aligned}$$

Additionally, from the definition of h we directly get $g - fh = 0 \mod q$. Now we write down the matrix product explicitly:

$$\mathbf{B}\mathbf{A}^\star = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix} \cdot \begin{bmatrix} 1 \\ h \end{bmatrix} = \begin{bmatrix} g - fh \\ G - Fh \end{bmatrix} = 0 \mod q.$$

2.2.2 Description of Sign

Before going into detail, we will outline how the steps of Sign compare to the GPV framework. In Line 2 the message m (appended with a random bit-string r) is hashed to a polynomial c, corresponding to the vector $\mathbf{c}$ in GPV. The preimage calculation of $\mathbf{c}$ under $\mathbf{A}$ is trivial with the choice of the Falcon public matrix and we can simply set $\mathbf{c}_0 = (c, 0)$. It is easy to check that $\mathbf{c}_0\mathbf{A}^\star = c$. Next, the preimage $\mathbf{t}$ under $\mathbf{B}$ is calculated in the FFT domain, which has no direct correspondence to GPV. In Line 6 an integer point $\mathbf{z}$ that is close to $\mathbf{t}$ is computed. Here, $\mathsf{ffSampling}$ is the trapdoor sampler of Falcon and the Falcon tree T is the trapdoor. Compared to GPV, $\mathbf{t}\hat{\mathbf{B}}$ and $\mathbf{z}\hat{\mathbf{B}}$ assume the roles of $\mathbf{c}_0$ and $\mathbf{v}$ respectively.

Algorithm 1 $\mathsf{Keygen}(\phi, q)$

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Ensure: A secret key sk, a public key pk

1: $f, g, F, G \leftarrow \mathsf{NTRUGen}(\phi, q)$
2: $\mathbf{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$ ▷ Secret lattice basis
3: $\hat{\mathbf{B}} \leftarrow \mathsf{FFT}(\mathbf{B})$
4: $\mathbf{G} \leftarrow \hat{\mathbf{B}}\hat{\mathbf{B}}^{\star}$
5: $\mathsf{T} \leftarrow \mathsf{ffLDL}^{\star}(\mathbf{G})$ ▷ Initial FALCON tree
6: **for** each leaf leaf of T **do**
7: $\quad \mathsf{leaf.value} \leftarrow \sigma/\sqrt{\mathsf{leaf.value}}$ ▷ Normalize tree
8: $\mathsf{sk} \leftarrow (\hat{\mathbf{B}}, \mathsf{T})$
9: $h \leftarrow gf^{-1} \mod q$ ▷ Non-trivial part of public lattice basis
10: $\mathsf{pk} \leftarrow h$
11: **return** sk, pk

The difference $(\mathbf{t} - \mathbf{z})\hat{\mathbf{B}} = \mathbf{c}_0 - \mathbf{v} = \mathbf{s}$ is the GPV signature (but in the FFT domain) and a candidate for a FALCON signature. All subsequent checks and calculations are specific to FALCON and do not have corresponding steps in GPV.

Before the message is hashed, it is combined with a random salt r, a bit-string of length 320. This is done to add randomness to the hashing process of c, such that repeated hashing of the same message will return different results each time (with negligible probability of collision).

The preimage is defined as $\mathbf{t} = (c, 0)\mathbf{B}^{-1}$. One can check that the inverse of the secret matrix $\mathbf{B}$ is the following:

$$\mathbf{B}^{-1} = \frac{1}{q} \cdot \begin{bmatrix} -F & f \\ -G & g \end{bmatrix} \in \left(\mathbb{Q}[x]/\phi\right)^{2\times 2}.$$

Note that this matrix has rational coefficients in general. We can write down the preimage calculation explicitly to get:

$$\mathbf{t} = (c, 0)\mathbf{B}^{-1} = (c, 0) \cdot \frac{1}{q} \cdot \begin{bmatrix} -F & f \\ -G & g \end{bmatrix} = \frac{1}{q} \cdot (-cF, cf).$$

This is what is calculated in Line 3, albeit in the FFT domain. Since $\mathbf{B}^{-1}$ has rational coefficients, the preimage $\mathbf{t}$ will have rational coefficients as well.

Using $\mathsf{ffSampling}$, a vector $\mathbf{z} \in (\mathbb{Z}[x]/\phi)^2$ is sampled in such a way that the calculation in Line 7 yields signatures $\mathbf{s} \in (\mathbb{Z}[x]/\phi)^2$ that are distributed according to $D_{\Lambda(\mathbf{B})+(c,0),\sigma,0}$. This is a spherical, discrete Gaussian distribution on the shifted lattice $\Lambda(\mathbf{B}) + (c, 0)$ with standard deviation σ and centered around the origin. See Section 2.3.3 for details on the sampling process.

Only if the squared Euclidean length of $\mathbf{s}$ is shorter than the signature bound $\lfloor \beta^2 \rfloor$, the post-processing can begin. First $\mathbf{s}$, which is still in FFT representation, is brought back into coefficient representation. The first component polynomial s_1 can be calculated from public values alone (see verification) and can therefore be omitted from the final output. The other component s_2 is compressed coefficient-wise using a custom encoding. Only when all these steps result in a non-empty output, the bit-string s is concatenated with the salt r and the final signature $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$ is returned.

2.2.3 Description of Verify

Signature verification is a relatively quick and straightforward process. We get a message m and a signature $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$, along with the public key pk and the signature bound $\lfloor \beta^2 \rfloor$ as input. It starts with

Algorithm 2 $\mathsf{Sign}(\mathsf{m}, \mathsf{sk}, \lfloor\beta^2\rfloor)$

Require: A message m, a secret key $\mathsf{sk} = (\hat{\mathbf{B}}, \mathsf{T})$ where $\hat{\mathbf{B}}$ contains polynomials $\hat{f}, \hat{g}, \hat{F}, \hat{G}$, a bound $\lfloor\beta^2\rfloor$
Ensure: A signature sig of m

$\mathsf{r} \leftarrow \{0,1\}^{320}$
$c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m}, q, n)$ ▷ Hash message to point
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathsf{FFT}(c) \odot \hat{F}, \frac{1}{q}\mathsf{FFT}(c) \odot \hat{f}\right)$ ▷ Calculate preimage
do
  do
    $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$ ▷ Trapdoor sampling
    $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$ ▷ GPV signature
  while $\|\mathbf{s}\|^2 > \lfloor\beta^2\rfloor$
  $(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$
  $\mathsf{s} \leftarrow \mathsf{Compress}(s_2, 8 \cdot \mathsf{sbytelen} - 328)$ ▷ Short, compressed Falcon signature
while $\mathsf{s} = \perp$
return $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$

the preparation of the polynomial c as a hash from the salt r and the message m exactly as in Sign. Then the signature component s is decompressed into the polynomial s_2. If the decompression yields an empty result (i.e., the compression is somehow invalid), we can immediately reject the signature. Else, we compute the other signature polynomial s_1 according to $s_1 = c - s_2 h$ and normalize the coefficients mod q to be between $\lceil -q/2 \rceil$ and $\lfloor q/2 \rfloor$. Now we check that s_1 and s_2 together are shorter than the bound $\lfloor\beta^2\rfloor$. If that is the case, we can accept the signature, otherwise we reject it.

The steps described here seem quite different from what the GPV framework would suggest. Remember that the verifier needs to check whether or not $\mathbf{v}$ (corresponding to $\mathbf{zB}$) is a point on the lattice generated by $\mathbf{B}$. The trick is that the computation of s_1 (the "missing" part of the signature) replaces the check on $\mathbf{v}$. To see why, we can make the following calculations:

$$\begin{aligned}
\mathbf{v} \in \Lambda(\mathbf{B}) &\Leftrightarrow \mathbf{v}\mathbf{A}^\star = 0 \mod q \\
&\Leftrightarrow \big((c,0) - (s_1, s_2)\big) \cdot \begin{bmatrix} 1 \\ h \end{bmatrix} = 0 \mod q \\
&\Leftrightarrow c - (s_1 + s_2 h) = 0 \mod q \\
&\Leftrightarrow s_1 = c - s_2 h \mod q.
\end{aligned}$$

So by calculating s_1 from known values c, s_1 and h and comparing it to the original s_1, the verifier would be able to confirm the validity of $\mathbf{v} \in \Lambda(\mathbf{B})$. Note, however, that a direct comparison of the recreated s_1 to the original is not possible, since it was omitted from the signature. What the verifier can do instead is look at the norm. If s_2 was not a valid signature for c, then $c - s_2 h$ would essentially be a random polynomial and the combined (s_1, s_2) would be short only with negligible probability.

2.2.4 List of Parameters

We provide a list of the most important parameters for Falcon in Table 1. This includes the security levels 9 and 10 recommended for practical usage (corresponding to the levels I and V as defined by NIST [NIS16]), as well as lower security levels $\kappa = 1$ to $\kappa = 8$ for testing purposes.

Algorithm 3 Verify$(\mathsf{m}, \mathsf{sig}, \mathsf{pk}, \lfloor\beta^2\rfloor)$

Require: A message m, a signature $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$, a public key $\mathsf{pk} = h \in \mathbb{Z}_q[x]/\phi$, a bound $\lfloor\beta^2\rfloor$
Ensure: Accept or reject

$c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m}, q, n)$
$s_2 \leftarrow \mathsf{Decompress}(\mathsf{s}, 8 \cdot \mathsf{sbytelen} - 328)$
if $s_2 = \perp$ **then**
    reject
$s_1 \leftarrow c - s_2 h \mod q$ ▷ Calculate missing signature component
if $\|(s_1, s_2)\|^2 \leq \lfloor\beta^2\rfloor$ **then**
    accept
else
    reject

κ	$n = 2^\kappa$	$\sigma_{\{f,g\}}$	σ	$\sigma_{\min}$	$\lfloor\beta^2\rfloor$	sbytelen
1	2	64.9	144.8	1.117	101498	44
2	4	45.9	146.8	1.132	208714	47
3	8	32.4	148.8	1.148	428865	52
4	16	22.9	151.8	1.170	892039	63
5	32	16.2	154.7	1.193	1852696	82
6	64	11.5	157.5	1.214	3842630	122
7	128	8.1	160.3	1.236	7959734	200
8	256	5.7	163.0	1.257	16468416	356
9	512	4.1	165.7	1.278	34034726	666
10	1024	2.9	168.4	1.298	70265242	1280

Table 1: Parameter sets of FALCON for security levels from $\kappa = 1$ to $\kappa = 10$. The columns $\sigma_{\{f,g\}}$, σ and $\sigma_{\min}$ have been rounded.

Algorithm 4 NTRUGen(ϕ, q)

Require: A prime modulus q, a polynomial $\phi = x^n + 1 \in \mathbb{Z}[x]$ with $n = 2^\kappa$
Ensure: Polynomials $f, g, F, G \in \mathbb{Z}[x]/\phi$ that verify $fG - Fg = q$

$\sigma_{\{f,g\}} \leftarrow 1.17\sqrt{q/(2n)}$
for $i = 0, \ldots, n-1$ **do** ▷ Randomly generate f, g
    $f_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}}}$
    $g_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}}}$
$f \leftarrow \sum_i f_i x^i$
$g \leftarrow \sum_i f_i x^i$
if NTT(f) contains 0 as a coefficient **then** ▷ Check invertibility of f
    Restart
$\gamma \leftarrow \max\left\{\|(g,-f)\|, \left\|\left(\frac{qf^\star}{ff^\star+gg^\star}, \frac{qg^\star}{ff^\star+gg^\star}\right)\right\|\right\}$ ▷ Calculate $\|\mathbf{B}\|_{\text{GS}}$
if $\gamma > 1.17\sqrt{q}$ **then**
    Restart
$F, G \leftarrow$ NTRUSolve(f, g) ▷ Solve NTRU equation
if $(F, G) = \perp$ **then**
    Restart
return f, g, F, G

2.3 Specific Subroutines in Detail

This section is dedicated to a more in-depth explanation of NTRUGen, NTRUSolve, ffLDL*, SamplerZ and ffSampling. These subroutines contain a lot of the more intricate aspects of Falcon and a thorough understanding of them is necessary for the sensitivity analysis that follows in Section 3.

2.3.1 Description of NTRUGen and NTRUSolve

These algorithms produce four polynomials $f, g, F, G \in \mathbb{Z}[x]/\phi$ that make up a Falcon secret key. The process starts by sampling the coefficients of f and g from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma_{\{f,g\}}}$. The parameter $\sigma_{\{f,g\}}$ sets the standard deviation of this distribution and its value depends on the polynomial degree n:

$$\sigma_{\{f,g\}} = 1.17\sqrt{q/(2n)}.$$

So for a higher degree, the coefficients of f and g are on average smaller in absolute values. This has the effect that the pair (f, g) has a constant expected length, independent of the security level:

$$\mathbb{E}\big[\|(f,g)\|\big] = 1.17\sqrt{q}.$$

For the calculation of the public key, f needs to be invertible as an element of $\mathbb{Z}_q[x]/\phi$. This can be checked by converting f into the NTT representation and checking that none of its coefficients are zero. Otherwise, the process starts over with the sampling of new f and g.[5]

Another check follows to make sure that the average length of signatures that will be generated with the secret key is short enough. The norm of signatures is proportional to the secret key's Gram-Schmidt norm $\|\mathbf{B}\|_{\text{GS}}$. As was shown that this value can be computed quite efficiently from f and g alone, without knowing the other two polynomials in advance [DLP14]. Specifically, the value:

$$\gamma = \max\left\{\|(g,-f)\|, \left\|\left(\frac{qf^\star}{ff^\star+gg^\star}, \frac{qg^\star}{ff^\star+gg^\star}\right)\right\|\right\},$$

[5] It is not clear why g would need to be recomputed too if f fails the invertibility check. We suspect that it is, in fact, not necessary. Note that the C reference implementation performs the various checks in a different order.

is calculated. It is the Gram-Schmidt norm of the secret basis and its value must not be greater than $1.17\sqrt{q}$ in order to guarantee that the secret key will be able to generate sufficiently short signatures.

Now that f and g are generated and checked, the last step is to find suitable solutions F and G with integer coefficients to the NTRU equation $fG - Fg = q$ with NTRUSolve. Note that we cannot simply pick an arbitrary $F \in \mathbb{Z}[x]/\phi$ and then define $G = (q + Fg)f^{-1}$, since f is only guaranteed to be invertible in $Z_q[x]/\phi$. To overcome this problem, the divide-and-conquer principle of the ring $\mathbb{Z}[x]/\phi$ is leveraged: We project the NTRU equation down into smaller rings until we reach $\mathbb{Z}$, find a solution and lift it back up into the original ring. The solution method used in Falcon was first proposed by Pornin and Prest [PP19].

For the descent phase we use the field norm, which is usually defined in terms of field extensions and Galois conjugates. In the context of Falcon its definition simplifies to the following[6]:

$$\begin{aligned} \mathrm{N} : \mathbb{Z}[x]/\phi &\to \mathbb{Z}[x]/\phi' \\ f(x) &\mapsto f(x) \cdot f(-x) \bmod \phi', \end{aligned}$$

where $\phi = x^n + 1$ and $\phi' = x^{n/2} + 1$. To keep the notation simple, we will use N as the name for the field norm between any two rings in the tower, regardless of the degree. Applying the field norm $\log n$ times to f and g will result in two integers $f', g' \in \mathbb{Z}[x]/(x+1) \cong \mathbb{Z}$. We are at the bottom of the tower, where a solution can easily be found. Consider the equation:

$$uf' + vg' = 1,$$

where integer solutions exist if and only if $\gcd(f', g') = 1$. Importantly, values for u and v can efficiently be calculated with the extended Euclidean algorithm. If we define $F' = -qv$ and $G' = qu$, we get solutions to the NTRU equation in $\mathbb{Z}$:

$$f'G' - F'g' = f'qu - (-q)vg' = q(uf' + vg') = q.$$

What follows is the ascent phase, where we iteratively lift this solution back up. We will only illustrate the last step from $\mathbb{Z}[x]/\phi'$ to $\mathbb{Z}[x]/\phi$. All previous steps work equivalently.

Let $N(f) = f'$ and $N(g) = g'$ for some polynomials $f, g \in \mathbb{Z}[x]/\phi$ and say we have already found $F', G' \in \mathbb{Z}[x]/\phi'$ that satisfy $f'G' - F'g' = q$ in this smaller ring. To get $F, G \in \mathbb{Z}[x]/\phi$, such that $fG - Fg = q$ is satisfied in the larger ring, it suffices to choose:

$$F = F'(x^2)g(-x) \quad \text{and} \quad G = G'(x^2)f(-x).$$

A proof why these polynomials, together with f and g, form a valid solution can be found in [PP19], but will be omitted here.

If we were to apply this lifting step as is, we would get polynomials F and G with potentially very large coefficients, which would be undesirable. To remedy this fact, we reduce (F, G) with respect to (f, g) after each step of the ascent phase. The exact details of this can also be found in [PP19].

Finally, note that the described procedure is successful in theory, as long as f and g are coprime. Depending on the implementation, however, further conditions may have to be met.

2.3.2 Description of ffLDL*

This is the subroutine of Keygen that computes a Falcon tree from the secret key $\mathbf{B}$ in the form of $\mathbf{G} = \mathbf{B}\mathbf{B}^\star$. A straightforward calculation shows that $\mathbf{G}$ is self-adjoint, meaning that $\mathbf{G}^\star = \mathbf{G}$. For matrices of this type, the so-called LDL*-decomposition can be defined: $\mathbf{G} = \mathbf{L}\mathbf{D}\mathbf{L}^\star$, with a lower unit

[6] The name *field* norm is justified, because we could define the same function for the field $\mathbb{Q}[x]/\phi$. Since we only ever use it with integer polynomials in the context of NTRUSolve, we restrict its definition accordingly.

Algorithm 5 SamplerZ(μ, σ')

Require: Floating-point values $\mu, \sigma' \in \mathbb{R}$ with $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Ensure: An integer $z \in \mathbb{Z}$ sampled from a distribution very close to $D_{\mathbb{Z},\sigma',\mu}$

$r \leftarrow \mu - \lfloor \mu \rfloor$ ▷ Fractional part of distribution center μ
$ccs \leftarrow \sigma_{\min}/\sigma'$
while True **do**
  $z_0 \leftarrow$ BaseSampler() ▷ $z_0 \sim D_{\mathbb{Z}^+,\sigma_{\max}}$
  $b \leftarrow$ UniformBits(1)
  $z \leftarrow b + (2 \cdot b - 1) z_0$ ▷ $z \sim BG_{\sigma_{\max}}$
  $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$
  if BerExp$(x, ccs) = 1$ **then**
    return $z + \lfloor \mu \rfloor$ ▷ $z \sim D_{\mathbb{Z},\sigma',r}$, $z + \lfloor \mu \rfloor \sim D_{\mathbb{Z},\sigma',\mu}$

triangular matrix $\mathbf{L}$ and a diagonal matrix $\mathbf{D}$. In the case of a 2×2-matrix, the LDL*-decomposition can be explicitly written down as:

$$\mathbf{G} = \begin{bmatrix} G_{00} & G_{01} \\ G_{10} & G_{11} \end{bmatrix} \quad \rightsquigarrow \quad \mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline G_{10}/G_{00} & 1 \end{array}\right], \quad \mathbf{D} = \left[\begin{array}{c|c} G_{00} & 0 \\ \hline 0 & G_{11} - L_{10} L_{10}^\star G_{00} \end{array}\right],$$

where $L_{10} = G_{10}/G_{00}$ is well defined, as long as G_{00} is not zero. This is always the case here, as G_{00} essentially corresponds to the calculation $\langle \mathrm{Vec}(g, -f), \mathrm{Vec}(g, -f) \rangle = \|\mathrm{Vec}(g, -f)\|^2$, which is non-negative for any vector (resp. pair of polynomials) and zero only if the vector is zero.

Note that the format of $\mathbf{L}$ and $\mathbf{D}$ already determines $L_{00} = L_{11} = 1$ and $L_{01} = D_{10} = D_{01} = 0$. So we are left with only three non-trivial entries L_{10}, D_{00}, D_{11} that need to be stored. We could, for example, store L_{10} in the root node of a tree and the two diagonal entries in a left and right child node respectively. The divide-and-conquer principle in FALCON enables us to recursively split the leaves of such a tree. For this, we split the diagonal elements via $\mathsf{split}(D_{00}) = (d_{00}, d_{01})$ and $\mathsf{split}(D_{11}) = (d_{10}, d_{11})$. Then we define two new, self-adjoint matrices:

$$\mathbf{G}_0 = \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline d_{01}^\star & d_{00} \end{array}\right], \quad \mathbf{G}_1 = \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline d_{11}^\star & d_{10} \end{array}\right].$$

Those can again be LDL*-decomposed and the resulting trees replace the leaves where D_{00} and D_{11} were previously stored. Remember that when D_{00} and D_{11} are polynomials in $\mathbb{Q}[x]/(x^n + 1)$, then the d_{ij}'s are elements of the smaller field $\mathbb{Q}[x]/(x^{n/2} + 1)$. So, continuing this procedure eventually results in a tree of height[7] $\log n$, where the leaves contain elements from $\mathbb{Q}[x]/(x^1 + 1)$, i.e. rational numbers. If we count the layers, starting with 0 for the leaves, the nodes in the i-th layer contain polynomials of degree $2^i - 1$ (i.e., polynomials with 2^i coefficients). This structure is the FALCON tree of the original matrix $\mathbf{G}$. So in essence, a FALCON tree is a way to store a recursive LDL*-decomposition of a matrix.

In implementations the calculations presented above are performed completely in FFT representation (using splitfft instead of split), hence the name ffLDL*. This improves the efficiency, because it greatly simplifies the multiplications and divisions of polynomials.

2.3.3 Description of ffSampling and SamplerZ

The sampling of discrete Gaussian distributions is perhaps the most involved part of the whole scheme. The method that is used in FALCON is based on many previous works, such as [DP15], [HPRR19]

[7] In graph theory, the height of a tree is usually defined as the maximum number of edges from the root node to one of the leaves. So a height of $\log n$ means that there are $1 + \log n$ layers of nodes.

and [PRR19]. On a very high level, the subroutine SamplerZ is a function that generates individual integer samples for a given distribution $D_{\mathbb{Z},\sigma',\mu}$. Then ffSampling takes those samples and combines them into a vector $\mathbf{z} \in (\mathbb{Z}[x]/(x^n+1))^2$ that is close to the input point $\mathbf{t} \in (\mathbb{Q}[x]/(x^n+1))^2$. It does so in a secure way, meaning that no information about the secret key is leaked through the signatures. Just as with NTRUSolve and ffLDL*, the divide-and-conquer principle is essential for ffSampling.

SamplerZ. We want to produce an integer sample from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma',\mu}$. A useful property of such distributions is that the shape of their probability mass function (PMF) stays the same for integer shifts of the mean. Specifically, if $p_{\sigma,\mu}(x)$ and $p_{\sigma,\mu+k}(x)$ are the PMFs of two distributions $D_{\mathbb{Z},\sigma,\mu}$ and $D_{\mathbb{Z},\sigma,\mu+k}$ with $k \in \mathbb{Z}$, then we have $p_{\sigma,\mu}(x) = p_{\sigma,\mu+k}(x+k)$. So if we are able to sample from distributions with means $\mu \in [0,1)$, then we can already sample from distributions with arbitrary means $\mu \in \mathbb{R}$ by applying integer shifts. This allows us to remove the integer part of the desired distribution center μ in Line 1 of Algorithm 5, take its fractional part r as the mean and to later add back the integer part $\lfloor \mu \rfloor$ in Line 9.

The function BaseSampler is a CDT-sampler that generates an integer $z_0 \in [0,1,\ldots,18]$ according to the half-Gaussian distribution $D_{\mathbb{Z}^+,\sigma_{\max}}$ (Line 4). The CDT-table is provided as part of the Falcon specifications. In Lines 5 and 6 we take a single random bit $b \in \{0,1\}$ and apply it to z_0, almost like a sign-bit. If $b=0$, then $z=-z_0$ and if $b=1$, then $z=1+z_0$. In effect, this "unfolds" the half-Gaussian distribution. Note that the resulting distribution is, however, not exactly $D_{\mathbb{Z},\sigma_{\max}}$. Instead, the upper half of the PMF coming from $z=1+z_0$ is shifted by one compared to the real Gaussian distribution. Let us define the distribution of z as $BG_{\sigma_{\max}}$, with:

$$BG_{\sigma_{\max}}(z) = \begin{cases} \frac{1}{2}D_{\mathbb{Z}^+,\sigma_{\max}}(-z), & z \le 0, \\ \frac{1}{2}D_{\mathbb{Z}^+,\sigma_{\max}}(z-1), & z > 0. \end{cases}$$

Now follows a rejection step, where samples are rejected in such a way that the final distribution almost exactly matches the desired distribution specified by the inputs. We know that if we accept a sample z with probability:

$$\frac{D_{\mathbb{Z},\sigma',r}(z)}{BG_{\sigma_{\max}}(z)},$$

then the resulting samples follow $D_{\mathbb{Z},\sigma',r}$. A short calculation reveals that this acceptance probability is equal to:

$$\exp\left(\frac{z_0^2}{2\sigma_{\max}^2} - \frac{(z-r)^2}{2\sigma'^2}\right) = \exp(-x),$$

with x the value calculated in Line 7. Lastly, the function BerExp returns 1 with probability $\exp(-x)$ and 0 otherwise.

The variable *ccs*, calculated in Line 2 and used as second input to BerExp, is introduced for security and serves to make the rejection isochronous with respect to μ, σ' and z. This means that the runtime of BerExp does not depend on the value of these variables and so no information can be obtained from timing measurements. See [HPRR19] for details and the proof of isochrony.

ffSampling. The origin of the ffSampling subroutine is Babai's Nearest Plane algorithm from Section 2.1.7. There are essentially two steps of evolution between the two samplers: an adaptation to Falcon's algebraic structure that comes with an efficiency gain and a randomization that makes the sampled points independent of the secret lattice basis. Interestingly, these two steps are more or less independent. Applying a randomization to the Nearest Plane algorithm leads to Klein's trapdoor sampler [Kle00]. On the other hand, Ducas and Prest introduced Fast Fourier Nearest Plane [DP15] as a version of Babai's Nearest Plane that takes into account a divide-and-conquer principle similar to that of Falcon. Finally, Falcon's ffSampling can be viewed either as randomized Fast Fourier Nearest Plane or as Klein's sampler for lattices with a ring structure. In the following explanation we will focus on the randomization, as it directly relates to the security aspect and will play an important role

in the later sensitivity analysis in Section 3 and also in the NarrowSampling attack presented in Section 4.

Usually, Babai's Nearest Plane algorithm runs deterministically, i.e., for a fixed lattice and target it will output the same lattice point. The idea now is to replace the rounding that happens when determining the closest hyperplane with random rounding according to a discrete Gaussian distribution.

More precisely, in the notation of Section 2.1.7, if λ is the parameter determining the exact hyperplane (in the first iteration of Nearest Plane), then we do not take $\lfloor \lambda \rceil$ as our choice for the coefficient. Instead we sample an integer from a discrete Gaussian distribution with center λ and standard deviation proportional to $1/\|\tilde{\mathbf{b}}_{d-1}\|$. Subsequent iterations operate similarly in lower-dimensional hyperplanes.

Visually speaking, since the width of the coefficient-wise distributions are inversely proportional to the lengths of the Gram-Schmidt vectors, the parallelepiped $\mathcal{P}(\mathsf{GSO}(\mathbf{B}))$ from Babai's Nearest Plane algorithm is blurred into a spherical Gaussian distribution and therefore made unrecognizable. This guarantees that no sensitive information about the secret matrix $\mathbf{B}$ is leaked through the signatures.

3 Sensitivity Analysis for FALCON Variables

In this section, we present our analysis of the physical security of FALCON. We first define the categories and labels that will be used in the rest of Section 3 and outline the scope of the analysis. Following that we proceed with the analysis proper, going through Keygen (Section 3.2, including NTRUGen), Sign (Section 3.3, with its subroutines in a dedicated Section 3.4) and Verify (Section 3.5). We identify possible vulnerabilities, including known attack vectors from prior works as well as vulnerabilities that have remained unexploited to the best of our knowledge. All results are summarized in Tables 2, 3, 4, 5 and 6. Lastly, in Section 3.6 we compile a list of all attack vectors that we identified.

Note that in the scope of this thesis we did not perform physical experiments on actual hardware but rather relied on simulations of attacks in code.

3.1 Explanation of Tables, Labels and Acronyms

3.1.1 General Organization

In the following tables we summarize every considered variable that occurs in the FALCON signature scheme, from Keygen through Sign to Verify, roughly in the order they appear in. We also look at most of the subroutines. Our notation is the same as used in the original FALCON specifications. This has the effect that sometimes different variables have the same name. We try to avoid confusion by grouping variables according to subroutines, separating them into different tables and adding short descriptions to each variable.

On the other hand, some variables occur in more than one part of the scheme. For example, the secret and public keys are generated in Keygen and later used in Sign and Verify respectively. In those cases we list all occurrences of the variable and assess the sensitivity depending on the context. In particular, attacks that target such a variable are always listed in the table corresponding to where in the scheme the attack is happening.

Apart from the variable name and description, each table entry consists of three more columns: SCA, FA and Public, which are explained hereafter. Most attacks that can be found in the literature either have very generic names or no name at all. For easier reference within this thesis we will give each attack a short and somewhat descriptive name. Additionally we cite the paper(s) they were presented or improved in.

Sometimes a variable's sensitivity is directly linked to that of another variable. In these cases we will make references of the form "→ Variable" in the tables. Similarly, when describing a vulnerability in the text, we reference attacks that exploit that particular vulnerability via "(→ Attack)". A more detailed description of these attacks can then be found in Section 3.6. To differentiate between attacks from prior literature and novel ones we use the markers "Attack^P" and "Attack^N" respectively. When an existing attack method or parts thereof are adapted to a different context we mark it as "Attack^A". For readability we only use these markers in the references to the sensitivity tables, in the tables themselves and in the subsection titles of the Section 3.6.

3.1.2 SCA, Side-Channel Attacks

The attacker model for a side-channel attack is a passive one. Through physical measurements the adversary gains information about the internal state of the device that runs the cryptosystem that is to be attacked. Examples of measurements include precise power consumption on the clock-cycle level, timing attacks that measure the runtime of certain parts of the algorithm, or even electromagnetic radiation that is emitted from the device. From this data, the adversary can then try to infer the value of sensitive variables with a wide range of techniques, such as statistical analysis or approaches using machine learning.

To simulate a side-channel attack in code, we simply print the value we are interested in either directly to the console or store it in a file for later processing.

The column SCA in the following tables denotes the possibility of side-channel analysis to retrieve the value of the variable. A dash means that no side-channel attack is known, or that such an attack does not apply to the value in question (which is usually the case for public parameters). The designation *(potential)* either means that no method to obtain the value is known but partial or full knowledge of it could be used to decrease the security of the scheme or break it entirely, or when a connection to sensitive values cannot definitively be ruled out.

3.1.3 FA, Fault Injection Attacks

The attacker model for a fault injection attack (or fault attack for short) is an active one. The adversary can directly influence the execution of the cryptographic algorithm or corrupt the data stored on the device. This is possible via laser pulses, electromagnetic fields or high voltage, among others. From the public output (that is the result of a faulted execution) the adversary can again learn information about sensitive internal values.

For our simulations we can alter the code (add or remove lines, change the value of parameters, etc.) to behave in the way that we need for any given attack scenario.

In the following tables, the column FA provides a similar assessment for fault attacks as SCA does for side-channel attacks. Here, the key word *(potential)* is used to note a measurable influence on public values (e.g., the public key, signatures), but it is not clear how this can be used to gain knowledge of secret values.

In this column we also differentiate between data faults DF (causing the value of a variable to be changed) and control flow faults CFF (causing the algorithm to behave differently, for example skipping instructions or aborting loops). Control flow faults target operations (loops, function calls, etc.) instead of variables. However, they always have an effect on the immediate result of the targeted operation. The result is again a variable and the attack is listed under this variable.

3.1.4 Public Classification

Lastly, in the column Public we explain whether and why a variable can be considered public. This is most often the case for parameters that are part of the Falcon specifications, as well as parts of the public key, the signature or the message. Note that for public variables we can automatically assign a dash to SCA, since there is no need to use side-channel analysis to find a value that is publicly known. Vice versa we can mark a variable as non-public (i.e., sensitive) if a side-channel attack is known. To be on the safe side, we also assign a dash to variables that have a *(potential)* designation in the SCA column. Importantly, fault attacks are not only possible for secret values but for public ones as well. Both can have an influence on the execution of the scheme and reveal secret information through the eventual public output.

3.1.5 Scope of the Analysis

In this thesis we only consider the current version v1.2 of the standard Falcon signature scheme [FHK+20]. No other modes (e.g., key-recovery mode) or variants (e.g., deterministic Falcon) will be considered. We do, however, look at both Falcon-512 and Falcon-1024, as the analysis is identical up to the value of some parameters.

As mentioned, we only examine vulnerabilities to side-channel and fault attacks with the goal of partial or full key recovery. Fault attacks that cause a denial of service (DOS) are generally not part of the analysis, because of the abundance of possibilities for them.

When the vulnerability of values depends on implementation specific details we will rely on the reference implementations in C and Python. They are provided by the authors of Falcon as open source code[8].

[8] `https://falcon-sign.info/`

There are also some aspects that we explicitly exclude from our analysis. They are the following:

- The sampling of coefficients of f and g in NTRUGen. The process itself is not specified, but we consider some attack scenarios.
- Also in NTRUGen, the number theoretic transform NTT(f) and the subsequent check for zeros.
- Similarly, the particular implementation of the FFT.
- All polynomial arithmetic including adjoint polynomials, calculation of norms, splitting, merging, etc.
- The use of a Residue Number System (RNS) in NTRUSolve.
- The subroutine Reduce in NTRUSolve.
- The intermediate variables in ffLDL*, but we consider its output T in detail.
- Side-channel attacks on the subroutine HashToPoint. It operates on public values in a deterministic fashion. We do, however, consider faults on the hash output c.
- Similarly, Compress. Side-channel analysis does not apply and fault attacks can only cause a denial of service.
- Similarly, Decompress and sbytelen in Verify.
- For efficiency reasons there are many values in the C reference implementation that are hard-coded, like $\ln 2$ etc. We only consider them in the case where they are explicitly declared as variables in the specifications, e.g., the polynomial coefficients C in ApproxExp.
- Any auxiliary variables like loop counters, etc. Faulting them would be equivalent to control flow faults, which we generally do consider.
- Some variables, like *ccs* in SamplerZ, that are handed down as inputs to subroutines are only considered in the top-most instance.
- The final output of Verify, i.e., the information of whether a signature is accepted or rejected. It is not declared as a variable in the specifications, side-channel attacks do not apply and a fault attack (causing it to accept or alternatively reject every signature) would be trivial.

3.2 Sensitivity Analysis of Keygen

The results of this section are summarized in Table 2.

3.2.1 Basic Parameters

The polynomial modulus ϕ, its degree n and the integer modulus q are public parameters of Falcon. A fault attack is generally not possible, since the whole scheme depends on their specific values, even when they are not referenced explicitly. For example, a function that transforms a polynomial in $\mathbb{Z}[x]/\phi$ into its FFT representation might never explicitly use ϕ as a value, but is still highly dependent on it and its properties. The same analysis for ϕ, n and q also applies in the context of Sign, Verify and their respective subroutines, which is why we will not list these variables later on.

Variable	Description	SCA	FA	Public
ϕ	polynomial modulus	-	-	parameter
q	integer modulus	-	-	parameter
n	degree of ϕ	-	-	parameter
$\sigma_{\{f,g\}}$	standard deviation for secret key coefficients	-	-	parameter
f, g	secret key components	CDT-SPA^A	HalfKey^N(CFF)	-
F, G	secret key components	*(potential)*	*(potential)*	-
γ	secret key norm	*(potential)*	-	-
$\mathbf{B}$	secret matrix	*(potential)*	$\rightarrow f, g, F, G$	-
$\mathbf{G}$	FALCON tree matrix	*(potential)*	$\rightarrow \mathsf{T}$	-
T	FALCON tree	*(potential)*	SmallLeaves^N(DF)	-
σ	standard deviation for signature coefficients	-	NarrowSampling^N(DF)	parameter
h	public key	-	-	public key

Table 2: Sensitivity Table for FALCON Keygen, including NTRUGen.

3.2.2 Sampling of Primary Secret Key Components

The standard deviation $\sigma_{\{f,g\}}$ specifies the distribution for the coefficients of f and g. It is a public parameter, so side-channel attacks do not apply. Depending on the implementation, the sampling of coefficients of f and g might follow a range of different methods. For example, the Python reference implementation uses SamplerZ with $\sigma_{\{f,g\}}$ as an input. On the other hand, the C implementation never explicitly uses $\sigma_{\{f,g\}}$, but instead relies on a CDT-sampler with a precomputed table of values. Since $\sigma_{\{f,g\}}$ is a fixed parameter with a specific value for each security level, most practical implementations will probably use the second option, making fault attacks quite unpractical.

Even if a given implementation allowed fault attacks, the effects would not be far-reaching. In the FALCON specifications, the authors consider the possibility of so-called overstreched NTRU attacks, where the coefficients of f and g are very small. They conclude that the choice of q as the rather low value of 12289 makes these kinds of attacks irrelevant. Faulting $\sigma_{\{f,g\}}$ in the other direction, increasing its value, is also not very promising. The immediate effect would be larger polynomial coefficients for f and g, making a direct attack via lattice reduction easier. However, a bound of $1.17\sqrt{q}$ is enforced on the Gram-Schmidt norm of $\mathbf{B}$ in NTRUGen, greatly restricting the average size of coefficients. On that note, there are also restrictions on the encoding of key coefficients (at least in the C implementation), limiting each individual coefficient to a certain bit-size.

Increasing $\sigma_{\{f,g\}}$ (and assuming that the key is not rejected in the process) would also have an effect on the length of signatures. A larger key will produce longer signatures on average, which are theoretically easier to forge. But there is another norm check in Sign that limits the length of signatures for exactly this reason, preventing this attack vector too.

3.2.3 Side-Channel Analysis of Secret Key Components

The primary secret key components f and g together with the secondary secret key components F and G are probably the most vulnerable values of Keygen. Furthermore, all four polynomials are closely related. The following equations show how to retrieve the full secret key (i.e., all four polynomials) if only one of

its components is known.

$$
\begin{aligned}
f &\rightsquigarrow g = fh \mod q &&\rightsquigarrow (F, G) = \mathsf{NTRUSolve}(f, g),\\
g &\rightsquigarrow f = gh^{-1} \mod q &&\rightsquigarrow (F, G) = \mathsf{NTRUSolve}(f, g),\\
F &\rightsquigarrow G = Fh \mod q &&\rightsquigarrow (g, f) = \mathsf{NTRUSolve}(G, F),\\
G &\rightsquigarrow F = Gh^{-1} \mod q &&\rightsquigarrow (g, f) = \mathsf{NTRUSolve}(G, F).
\end{aligned}
$$

Note that the first equation in each line only holds mod q. The coefficients of all the polynomials follow a discrete Gaussian distribution centered around 0, with a standard deviation much smaller than q. This makes it easy to guess the correct representative, which is the one closest to 0. When f and g are known, an attacker can simply use NTRUSolve to get F and G, exactly how they are calculated during Falcon Keygen. Similarly (and perhaps surprisingly) NTRUSolve can also find f and g when F and G are known.[9] Note, however, that the order of inputs and outputs is reversed (e.g., (g, f) instead of (f, g)). Also note that in lines 2 and 4 (known g and known G respectively) we have to invert h, which is not always possible. But given a random polynomial (h looks uniformly random according to Falcon specifications and [SS13]) the probability of it being invertible are relatively high.[10]

Now the question remains how to gain knowledge of one of the key components initially. Prior work has shown that constant-time CDT-samplers are vulnerable to side-channel attacks [KH18]. It is possible to predict the exact output with high accuracy using a single power trace. When applied to the sampling of f and g, this makes it almost trivial to recover both polynomials in full ($\rightarrow$ CDT-SPA^A). On the contrary, there are no published attacks revealing F and G through side-channel analysis (during Keygen). But the explanations above show that they are still vulnerable targets, if such attacks are ever found.

Note that an adversary can theoretically only use single-trace attacks, because Keygen is only performed once per key. Looking into the specific implementations, however, there could be more possibilities for an attack. For example, the C reference code only stores the polynomials f, g and F as the secret key. The fourth component G is recomputed every time during Sign. These additional operations might provide further opportunities for side-channel analysis.

3.2.4 Fault Attacks on Secret Key Components

Depending on when in Keygen a fault is injected to one or more key components, it can have varying effects, due to how the fault propagates. There are essentially two scenarios: Firstly, faulting f or g during or right after they are generated and secondly, faulting f, g, F or G during or right after NTRUSolve.

Faults before NTRUSolve. If an attacker is able to completely fault f or g to known values instead of randomly sampling their coefficients, then they can, of course, work out the complete secret key. But it is unlikely that such a strong attack is possible in practice. A simpler attacker model could, for example, abort the sampling of f or g and cause the latter coefficients to be zero (depending on the implementation). The reduced value space could be enough to calculate the other, unknown coefficients using the relation with the public key, $fh = g$. However, experiments show that the probability that a faulted f or g can still be made into a full valid key are lower, the more coefficients are faulted to zero, see Figure 3 ($\rightarrow$ HalfKey^N).

[9] The roles of the pairs (f, g) and (F, G) are symmetric in the context of the NTRU equation, so it is not surprising that F and G can be calculated for given f and g. What is surprising, however, is that NTRUSolve can be used for this. It is based on the extended Euclidean algorithm, which can generally not be inverted like that. Also, we do not claim that this method will always work, but we have never observed a case for which the inverted NTRUSolve fails.

[10] This is easy to see if we consider the NTT representation of polynomials in $\mathbb{Z}_q[x]/\phi$. There are q^n polynomials in total, $(q-1)^n$ of which have only non-zero coefficients. Assuming that coefficients are distributed uniformly (as would be the case for h), we expect a probability of $\frac{(q-1)^n}{q^n}$ for h to be invertible. For security levels $\kappa = 9$ and $\kappa = 10$ this evaluates to around 96% and 92% respectively. Experimental evidence closely matches these numbers.

Faults after NTRUSolve. After all four secret key components are established, it is significantly harder to fault them, because the adversary has to make sure that the NTRU equation is still satisfied. A "broken" key is easy to detect (by simply checking the NTRU equation) and also signature generation would be significantly restricted, if not completely impossible. In theory we can show that the solution to the NTRU equation is not unique. If we have a valid secret key with $fG - Fg = q$, then for any $k \in \mathbb{C}[x]/\phi$ the polynomials $F' = F - kf$ and $G' = G - kg$ are also solutions:

$$fG' - F'g = f(G - kg) - (F - kf)g = fG - fkg - Fg + kfg = fG - Fg = q.$$

Similarly, when assuming fixed F and G, alternative solutions for f and g can be constructed. So in practice, whenever an adversary would want to fault any of the key components, they would need to fault the others in a very precise way to retain the solution property. Furthermore, some implementations (like the reference implementation in C) also enforce limitations on the size of coefficients, further limiting the space of possible key faults. More research needs to be done to assess whether these restrictions are too tight to be exploited into a full attack.

3.2.5 The Secret Matrix and Side-Channel Attacks on the Falcon Tree

The matrix $\mathbf{B}$ directly contains all four secret polynomials f, g, F and G and is therefore highly sensitive. The Gram matrix $\mathbf{G}$ is derived from $\mathbf{G} = \mathbf{B}\mathbf{B}^\star$. In turn, the Falcon tree T is a recursive LDL*-decomposition of $\mathbf{G}$ and it is possible to work out $\mathbf{G}$ from the tree T.

Fouque et al. [FKT+20] present a method for full key recovery, given the exact values stored in the leaves of T. They also point out that the diagonal part of the LDL*-decomposition of $\mathbf{G}$ directly contains those values. However, the suggested way in which the values in the leaves of T are recovered is based on a timing attack that is no longer applicable to the current version of Falcon. Furthermore, the timing information only reveals approximations of the leaf values. This problem could be dealt with at least in the original attack scenario [FKT+20] targeting DLP [DLP14], which is another signature scheme based on NTRU lattices. The adaptation to Falcon was deemed more complex and was left as an open problem.

Fault attacks on $\mathbf{G}$ are believed to be equivalent to faulting the Falcon tree T. Since the matrix is exclusively used to generate T, any fault would have an effect that could in theory be produced by directly faulting T instead. Still, the possibility remains that certain (for an attacker desirable) effects could be achieved more easily by faulting $\mathbf{G}$.

The value γ, computed in Line 9 of Algorithm 4, corresponds to the Gram-Schmidt norm of the matrix $\mathbf{B}$. Leveraging the underlying NTRU structure of $\mathbf{B}$, γ can be calculated as the maximum of $\|\tilde{\mathbf{b}}_0\|$ and $\|\tilde{\mathbf{b}}_n\|$, the lengths of the 0-th and n-th row vectors of $\mathsf{GSO}(\mathbf{B})$. Both of these norms can be calculated from the key components f and g, without knowing the full matrix, as is done in Falcon Keygen. Critically, these norms also occur as two of the leaf values in the Falcon tree T. Although we think it to be very unlikely that parts of the secret key can be recovered from the knowledge of γ alone, we cannot definitively classify it as non-sensitive.

Fault attacks on γ can either increase or decrease its value. An increase would lead to more rejections (and subsequent recomputation) of otherwise valid keys, but would have no effect on the security of the scheme. A decrease of γ (or alternatively skipping the norm check that is associated with it) would mean that a secret key with a Gram-Schmidt norm $\|\mathbf{B}\|_{\mathrm{GS}} > 1.17\sqrt{q}$ could pass the check. Signatures produced from this key would be longer on average, since their length is proportional to $\|\mathbf{B}\|_{\mathrm{GS}}$. This would slow down the generation of signatures, due to more rejections in Line 8 of Algorithm 2. Other than that the signing process would not be effected and still function as intended, so no sensitive information would be leaked. Furthermore, the key norm γ is on average close to the intended bound, so the overall effect would be rather limited.

3.2.6 Fault Attacks on the Falcon Tree

The tree T is used during Sign to create a spherical Gaussian distribution. Howe et al. [HPRR19] point out that mistakes in the implementation of the tree generation or in its use when signing can lead to a defective distribution. It is not described in detail how to achieve such defects and whether they can leak sensitive information. But this could be a possible attack vector for fault attacks.

Experiments show that changes to the values stored in the leaves of the Falcon tree do not hinder the creation of valid signatures, as long as the set values do not significantly deviate from 1. In particular, experiments show that larger values up to around 1.7 can still be used to successfully sign messages most of the time. More interestingly, smaller values that approach zero can be used with almost no restriction. Only the signing process takes longer the lower the value is. If an attacker is able to set the value of all (or at least a significant number of) leaves to, say, 0.1 the effect would be a significantly decreased standard deviation of ffSampling. This could create the possibility of parallelepiped learning attacks à la [GMRR22]. But the practicability of such an attack is rather low, as the attacker would need to inject a high number of faults to almost every leaf ($\rightarrow$ SmallLeaves^N).

An easier way to achieve the same result is to fault σ, which is used to normalize the leafs of the Falcon tree. It subsequently controls the standard deviation of the signature coefficients via ffSampling. The advantage of faulting σ with the goal of influencing the standard deviation of ffSampling is that only this one value needs to be changed, in comparison to each individual leaf value in the Falcon tree ($\rightarrow$ NarrowSampling^N).

It should be noted that the value of σ exactly corresponds to the standard deviation of signature coefficients (before rejection checks), as long as it is equal to or grater than the value prescribed in the parameter sets (see Table 1). Due to technical reasons, faulting σ to lower values will generally result in a lower standard deviation, but the correspondence is not one-to-one in this case.

3.2.7 Public Key

Finally, the public key h is, as the name suggests, a public value. Its generation is the last step of Keygen and so fault attacks cannot reveal any sensitive information at this point. For attacks on h in the context of Verify, see Section 3.5.

3.3 Sensitivity Analysis of Sign

The results of this section are summarized in Table 3.

3.3.1 Hashing the Challenge and Computing the Preimage

The salt r is a uniformly random bit-string of length 320 bits. It is part of the signature and therefore public. A fault attack is possible, where r is faulted to a fixed value, removing some of the randomized behavior of Falcon ($\rightarrow$ BadDeterminism^A).

The message m is public under our assumptions.

Because the challenge polynomial c is a deterministic hash value of $(\mathsf{r}\|\mathsf{m})$, it can be calculated from public information. Faulting c to an arbitrary but fixed value would result in different signatures for the same message hash. In theory, this could lead to a fault attack ($\rightarrow$ ConstantHash^A). Such an attack is easily detectable though: As c is recalculated during signature verification, the signature created with a faulted c' is not valid.

The polynomial f is the specific target in an attack, where side-channel analysis is applied to the FFT multiplication $\mathsf{FFT}(c) \odot \hat{f}$ during the computation of the preimage t [KA21]. The same attack could target $\mathsf{FFT}(c) \odot \hat{F}$, retrieve F and from there the full secret key ($\rightarrow$ MultFFT^P). An important part is that one of the factors in the targeted multiplication must be known to the adversary (c in this case). Since the other two secret polynomials g and G are never multiplied with any public values, they are safe from this specific attack.

Variable	Description	SCA	FA	Public
r	salt	-	BadDeterminism^{A}(DF)	signature component
m	message	-	-	message
c	challenge	-	ConstantHash^{A}(DF)	deterministic hash value
f, F	secret key components	MultFFT^{P}	RejectionLeakage^{N}(DF)	-
$\mathbf{t}$	preimage vector	HiddenSlice^{A}, HiddenHalfspace^{A}	GaussShift^{N}(DF)	-
T	FALCON tree	*(potential)*	SmallLeaves^{N}(DF), FaultyTree (DF, CFF)	-
$\mathbf{z}$	Gaussian sampled vector	CDTZero^{P}, HalfGaussSign^{P}	BEARZ^{P}(DF/CFF), GaussShift^{N}(DF)	-
g, G	secret key components	*(potential)*	RejectionLeakage^{N}(DF)	-
$\mathbf{s}$	signature	-	-	signature
β	signature norm bound	-	-	parameter
$\mathsf{sbytelen}$	maximum compressed signature length	-	-	parameter
s	compressed signature	-	-	deterministic compression

Table 3: Sensitivity Table for FALCON Sign, including its subroutine Compress.

For the preimage $\mathbf{t}$ no known side-channel attacks exist, but it is possible to recover f and F from its components due to:

$$-q \cdot c^{-1} t_0 = F, \qquad q \cdot c^{-1} t_1 = f,$$

and from either one of these the full secret key can be calculated. This relies on the invertibility of c, but as mentioned earlier most random polynomials are invertible.

3.3.2 Trapdoor Sampling

To assess the possibility of fault attacks on $\mathbf{t}$ (as well as f and F, which are involved in the preimage calculation) one has to consider the specific implementation of Falcon. On a high level the trapdoor sampler ffSampling takes as input, among other values, a vector $\mathbf{t}$ and outputs a vector $\mathbf{z}$ that is close to $\mathbf{t}$. The parameters of the function are chosen in such a way that the difference between input and output is small enough to guarantee that $\mathbf{s} = (\mathbf{t} - \mathbf{z})\mathbf{B}$ is a sufficiently short vector.

However, there is a crucial difference between the pseudocode and the reference implementation, both in Python as well as in C. In code, the signature is calculated according to $\mathbf{s} = (c, 0) - \mathbf{z}\mathbf{B}$, which is mathematically equivalent to $\mathbf{s} = (\mathbf{t} - \mathbf{z})\mathbf{B}$. A faulted $\mathbf{t}'$ would not correspond to c anymore. If the difference to the original preimage $\mathbf{t}$ is significant, the resulting signature will very likely be longer than the norm check allows and the attacker does not get any information because the signature would be recomputed.

The possibility remains that faulting only a part of $\mathbf{t}$ might be enough to keep the signature short and gain secret information from it. Experiments show that certain coefficients (of the FFT representation) of $\mathbf{t}$ are more sensitive to faults in the sense that they more likely cause a denial of service than other coefficients. Another possibility would be to fault all coefficients at once, but only by a very small amount. Both of these approaches could be pursued in different directions and still need to be researched further.

One promising attack vector of the above kind are faults to $\mathbf{t}$ that add a known fault vector $\boldsymbol{\delta}$, i.e. $\mathbf{t}' = \mathbf{t} + \boldsymbol{\delta}$. This causes the distribution of signatures to be shifted away from the origin by an error term $\boldsymbol{\delta}\hat{\mathbf{B}}$. Under the assumption that an adversary can finely tune $\boldsymbol{\delta}$ to suitable values, this reveals the full secret key. Applying the same logic to $\mathbf{z}$ we get a similar attack ($\rightarrow$ GaussShift^N).

Another fault attack on $\mathbf{z}$ has already been successfully performed [MHS+19]. If the call to ffSampling is skipped at a specific point of the recursion or alternatively certain coefficients of $\mathbf{z}$ are set to 0, then the adversary can recover F using lattice reduction ($\rightarrow$ BEARZ^P). Furthermore, side-channel attacks that recover the full secret key from partial knowledge of $\mathbf{z}$ have been performed ($\rightarrow$ CDTZero^P, $\rightarrow$ HalfGaussSign^P). These attacks essentially rely on the filtering of signatures that lie in certain regions of space determined by parts of $\mathbf{z}$. Due to $\mathbf{z}\mathbf{B} = (c, 0) - \mathbf{s}$, the filtered signatures then reveal information about the secret matrix $\mathbf{B}$.

Similarly, although not tested in practice yet, knowledge of certain parts of $\mathbf{t}$ could also lead to full key recovery via the relation $\mathbf{t}\mathbf{B} = (c, 0)$ ($\rightarrow$ HiddenSlice^A, $\rightarrow$ HiddenHalfspace^A).

Faulting the secret key components g and G in the multiplication $\mathbf{z}\mathbf{B}$ can result in a signature that does not pass the norm check. If by chance, $\mathbf{s}$ is short enough, another problem arises, due to the post-processing steps. When we write down the matrix multiplication explicitly we have:

$$(c, 0) - \mathbf{z}\mathbf{B} = (c - z_0 g - z_1 G, z_0 f + z_1 F) = (s_1, s_2),$$

meaning that g and G can only influence s_1. But this signature component is discarded and omitted, and only s_2 is part of the final output. So, faulting g and G either results in a DOS (the norm check never passes) or in an unaltered output as if no fault was injected in the first place.

There is, however, one crucial step where altering a coefficient of the secret key polynomials could reveal information. Experiments show that the probability of a successful norm check varies with the size of the faulted coefficient. Say, for example, that an adversary can fault one coefficient of either f, g, F or G to zero. If the original value was low to begin with, the fault does not make a big difference

and the resulting signature will most likely pass the norm check. If, on the other hand, the value of the coefficient was larger, the change to zero has a bigger impact and the probability of a short signature is lower. This means that at least some information is leaked that could theoretically be used for an attack ($\rightarrow$ RejectionLeakage^N). Whether this is enough to break the scheme is unknown at the moment.

3.3.3 Signature Post-Processing

We refer to $\mathbf{s}$ as the signature, although only a compressed version of its second component s_2 will be part of the output. The other component s_1 can be calculated from s_2 and additional public values, which is done during verification. Any fault attack on $\mathbf{s}$ would only invalidate the signature and not reveal secret information.

The signature norm bound β is a public parameter that restricts the Euclidean length of $\mathbf{s}$ in order to make it harder to forge signatures. Since $\mathbf{s}$ is the output of a mathematically secure trapdoor sampler, it does not leak sensitive information, no matter its length. A fault that increases the value of β or alternatively skips the norm check could create signatures that are valid, except for their length. But since the norm check is also performed in the verification step, this does not pose a problem. Also, most signatures already lie below this bound, so the probability of a longer than expected signature is quite low. In the FALCON specifications [FHK+20], the following formula gives an upper bound on the rejection probability:

$$\mathbb{P}\left[\|(s_1, s_2)\|^2 > \lfloor \beta^2 \rfloor\right] \leq \tau_{\text{SIG}}^{2n} \cdot e^{n(1-\tau_{\text{SIG}}^2)}.$$

The expression on the right evaluates to around $4.5 \cdot 10^{-5}$ and $2.4 \cdot 10^{-9}$ for FALCON-512 and FALCON-1024 respectively.[11]

The public parameter sbytelen controls the bit-size of the compressed signature. Since the verifier and the signer need to agree to the same value, a fault attack is not sensible.

The compressed signature $\mathbf{s}$ is the output of the deterministic (and invertible) function Compress. Faulting it will only invalidate the signature.

3.4 Sensitivity Analysis of Gaussian Sampling

The results of this section are summarized in Tables 4 and 5.

3.4.1 SamplerZ

The function ffSampling takes the preimage $\mathbf{t} = (t_0, t_1)$ in FFT representation as input and recursively splits it until elements of $\mathbb{Q}/(x^1 + 1) \cong \mathbb{Q}$ are reached, i.e. single-coefficient constant polynomials. Note that for polynomials of this kind, the FFT representation is equivalent to the coefficient representation. So the values of μ in the first two calls to SamplerZ are the unaltered last (i.e. highest-index) coefficients of t_1. After that however, the values of μ are slightly shifted due to the computation $t_0' = t_0 + (t_1 - z_1) \odot \ell$ between the first and second recursive call to ffSampling. Without additional knowledge it seems unlikely to recover $\mathbf{t}$ from all the values of μ alone. But they do describe a vector that is at least close to $\mathbf{t}$ and therefore also close to $\mathbf{z}$. Maybe this opens up the possibility for a viable attack.

Remember that the PDF p of a discrete Gaussian distribution with mean $\mu \in \mathbb{R}$ and p' with shifted mean $\mu + k$ with $k \in \mathbb{Z}$ satisfy $p'(x) = p(x - k)$. SamplerZ samples from $D_{\sigma',r}$ with r being the fractional part of the target mean μ and then simply adds $\lfloor \mu \rfloor$ to the result. Faulting r to zero is equivalent to faulting μ to $\lfloor \mu \rfloor$, meaning that the sampled values are slightly lower than expected. A negligible effect on the resulting signatures is that their norm is very slightly increased. More importantly, if we take the mean $\bar{s}_i$ of the i-th coefficient of $\mathbf{s}$ and collect them into a vector of two polynomials we can observe a systematic deviation from the expected $(0, 0)$. At the moment it is not clear what information can be learned from this mean signature vector.

Similarly to how the values of μ are probably not enough to learn any valuable information, we deem it even more unlikely that an attacker can gain knowledge of secret values from r alone.

[11] The parameter $\tau_{\text{SIG}} = 1.1$ is fixed for all security levels.

Variable	Description	SCA	FA	Public
μ	input mean for SamplerZ	*(potential)*	$\rightarrow \mathbf{t}$	-
r	non-integer part of μ	*(potential)*	*(potential)*	-
σ'	input standard deviation for SamplerZ	$\rightarrow$ T (leaf values)	SmallLeaves^N(DF)	-
$\sigma_{\min}$	lower bound for σ'	-	$\rightarrow$ *ccs*	parameter
ccs	auxiliary variable to make SamplerZ time independent of σ'	$\rightarrow \sigma'$	StdFault	-
u	random seed for BaseSampler	$\rightarrow z_0$	$\rightarrow z_0$	-
z_0	BaseSampler output	CDTZero^P, CDT-SPA^A	CDTZero^A(DF)	-
b	sign of z_0	HalfGaussSign^P	HalfGaussSign^A(DF)	-
z	z_0 with applied sign b	$\rightarrow z_0, b$	$\rightarrow z_0, b$	-
$\sigma_{\max}$	upper bound for σ'	-	$\rightarrow x$	parameter
x	exponent difference between sampled Gaussian and target Gaussian	$\rightarrow z_0, r$	StdFault	-

Table 4: Sensitivity Table for SamplerZ with its subroutine BaseSampler.

The standard deviations σ' are just the values stored in the leaves of the FALCON tree. Faulting these values in order to decrease them is equivalent to the SmallLeaves attack in Keygen, with the exception that the faults would need to be injected to every signature generation instead of once at the calculation of the FALCON tree T. So, a high number of faults is necessary to decrease the standard deviation of ffSampling every time, making this attack rather impractical. Whether a fault to a single call of SamplerZ (per signature) is sufficient to mount an attack needs to be researched.

Since $\sigma_{\min}$ is a public parameter, side-channel attacks are no concern. The auxiliary variable *ccs* is calculated as $\sigma_{\min}/\sigma'$, so it contains the same information as σ', i.e. the leaf values. Increasing *ccs* via fault injection has the same effect as decreasing σ'. It is not known at the moment whether the value space of *ccs* is large enough to influence the standard deviation of SamplerZ in a meaningful way.

3.4.2 BaseSampler

The BaseSampler is a CDT-sampler that takes a uniformly random seed u and generates a half-Gaussian sample z_0 from it using a cumulative distribution table. Any side-channel or fault attack targeting u would only reveal information or influence the value of z_0. As mentioned before, CDT-samplers are vulnerable to single trace power analysis that can reveal the exact output with very high accuracy. Concrete examples are the attacks by Guerreau et al. [GMRR22] and Zhang et al. [ZLYW23] ($\rightarrow$ CDTZero^P, $\rightarrow$ CDT-SPA^A). In its original form it uses the knowledge of which calls to BaseSampler result in $z_0 = 0$ to filter signatures accordingly. A variant of the attack injects faults to deliberately produce $z_0 = 0$.

Another attack, also presented by Zhang et al. [ZLYW23], targets the uniformly random sign bit b and recovers its value to, again, sort signatures accordingly ($\rightarrow$ HalfGaussSign^P). The variable z is calculated as either $z = -z_0$ or $z = z_0 + 1$, depending on whether $b = 0$ or $b = 1$. Note that $z_0 \in \{0, \ldots, 18\}$, therefore knowing z directly gives us z_0 and b: If $z \leq 0$ we have $z_0 = |z|, b = 0$ and if $z > 0$ we get $z_0 = z - 1, b = 1$.

The upper bound for σ' is $\sigma_{\max}$, which is defined as the standard deviation of the BaseSampler. Note that the BaseSampler uses a hard-coded CDT, so its standard deviation is fixed. Faulting $\sigma_{\max}$ can therefore only effect x.

Variable	Description	SCA	FA	Public
s	integer part of logarithmic decomposition	$\rightarrow x$ (SamplerZ)	$\rightarrow x$ (SamplerZ)	-
r	non-integer part of logarithmic decomposition	$\rightarrow x$ (SamplerZ)	$\rightarrow x$ (SamplerZ)	-
z	integer approximation of $2^{64} \cdot ccs \cdot e^{-x}$	$\rightarrow x$ (SamplerZ), $\rightarrow ccs$ (SamplerZ)	StdFault	-
w	auxiliary variable in BerExp	-	StdFault	resampled until $w = 1$
C	polynomial coefficients of approximation for e^x	-	$\rightarrow y$	list of fixed coefficients
z	auxiliary variable in ApproxExp	$\rightarrow r$ (BerExp), $\rightarrow ccs$ (SamplerZ)	$\rightarrow y$	-
y	integer approximation of $2^{63} \cdot ccs \cdot e^{-r}$	$\rightarrow r$ (BerExp), $\rightarrow ccs$ (SamplerZ)	StdFault	-

Table 5: Sensitivity Table for BerExp (upper four items) and ApproxExp (lower three items).

3.4.3 SamplerZ, cont.

The variable x is used in rejection sampling to get from the unfolded BaseSampler distribution to the target discrete Gaussian distribution. It is calculated as:

$$x = \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2},$$

and describes the ratio of the initial to the target distribution. A change to the value of x will propagate to the standard deviation of SamplerZ. However, more research needs to be done to study the exact influence and assess the possibility for a fault attack.

Regarding side-channel analysis, tests show that there is at least a slight correlation between the value of x and the BaseSampler output z_0, where $z_0 = 0$ generally leads to lower values for x. So by gathering enough traces of x, it might be possible for an adversary to distinguish zero from non-zero z_0, which would lead to a similar situation as in the CDTZero attack ($\rightarrow$ CDTZero^P).

Further tests show non-uniform patterns when plotting values of x against corresponding values of $r = \mu - \lfloor \mu \rfloor$. Whether this relation can be exploited into a full attack is left as an open question.

3.4.4 BerExp and ApproxExp

These two functions work together to generate a Bernoulli sample, i.e., a single bit, returning 1 with probability $ccs \cdot e^{-x}$. ApproxExp simply calculates the value $2^{63} \cdot ccs \cdot e^{-x}$ as an integer approximation, whereas BerExp does some pre-processing and then performs the actual sampling.

The input to BerExp is decomposed into $s \cdot \ln 2 + r$. Any fault to s or r that does not result in a DOS could alternatively be performed by altering x from SamplerZ. Similarly, knowledge of the two values only reveals information about x.

The variable z (in BerExp) contains an integer approximation of $2^{64} \cdot ccs \cdot e^{-x}$, so it could reveal information about x and ccs.

The variable w is the eventual output of BerExp. A value of 1 prompts SamplerZ to return an output. If, on the other hand $w = 0$, a new Gaussian sample is generated. A side-channel attack would not make sense, as BerExp is called until its output is 1. This also means that when w is faulted to 1 on every

Variable	Description	SCA	FA	Public
r	salt	-	-	signature component
m	message	-	-	message
c	challenge	-	ZeroChallenge^N(DF/CFF)	deterministic hash value
s	compressed signature	-	-	signature component
sbytelen	maximum compressed signature length	-	-	parameter
s_2	decompressed signature	-	-	deterministic decompression
h	public key	-	KeyReplacement^N(DF)	public key
s_1	secondary signature component	-	ShortenedSignature^N(DF)	calculated from public values
β	signature norm bound	-	LargerBound^N(DF), SkipCheck^N(CFF)	parameter

Table 6: Sensitivity Table for Falcon Verify.

call, no rejection sampling is taking place in SamplerZ and its output corresponds to a bimodal Gaussian distribution with standard deviation $\sigma_{\max}$. This alters the distribution of signatures, which could reveal secret information.

The funtion ApproxExp works entirely with integer values. But since its inputs r and ccs are floating-point values, they have to be converted into suitable integer representations first. The auxiliary variable z in ApproxExp (not to be confused with the value from BerExp by the same name) serves the purpose of storing those values: first $\lfloor 2^{63} \cdot r \rfloor$ and later $\lfloor 2^{63} \cdot ccs \rfloor$.

Similarly, the output y of ApproxExp only depends on r (from BerExp) and ccs and can therefore only reveal information about these two values.

3.5 Sensitivity Analysis of Verify

The results of this section are summarized in Table 6.

When verifying a signature we only deal with public values and parameters, or values that can be derived from them. This means that side-channel attacks do not need to be considered as there are no sensitive values to be revealed.

Fault attacks on r, m and c would cause the re-calculated s_1 to have a large norm with very high probability and therefore the signature would be rejected. Faulting s or sbytelen would most likely invalidate the encoding and also lead to rejection. A faulty s_2 or h will lead to a larger than expected s_1 and again a rejected signature.

More interesting are fault attacks that lead to accepting signatures that would otherwise be rejected. One possibility is to attack s_1, β or the norm check itself. If the norm check is skipped or β is faulted to a very high value, then any arbitrary message-signature pair will be accepted (→ LargerBound^N, → SkipCheck^N). Similarly, if s_1 is faulted to zero (or at least a very short polynomial), then any signature with sufficiently short s_2 will be accepted (→ ShortenedSignature^N).

Another way to force a short signature is to chose s such that the decompressed s_2 is exactly zero and then to fault the hashing of c to also be zero. This leads to $s_1 = c - s_2 h = 0$ and so the norm check passes

and the signature is accepted (→ ZeroChallenge^N).

Finally, the attacker could replace the verifier's public key h with their own h' and then sign any message with the corresponding secret key (→ KeyReplacement^N), but this assumes a quite strong attacker model.

3.6 List of Attacks

In this section we briefly describe each of the attacks from the tables above and, where possible, refer to the publications where they were first presented or improved upon.

3.6.1 CDT-SPA^A

Kim and Hong [KH18] have shown that constant-time CDT-samplers are vulnerable to SPA. When using such a sampler, a counter is incremented according to whether a uniformly random value is below (or above, depending on implementation details) a comparison value from a precomputed table. (See Section 2.1.3 for more details.) Comparisons between the uniform sample and the values from the table are usually implemented as subtraction. Different subtraction results can have varying Hamming weights, depending on whether they are positive or negative. This has a measurable effect on the power consumption, making it possible to count the number of increments and to reveal the final output of the sampler.

Applying this to Keygen, where the coefficients of f and g are sampled in this fashion, the full polynomials could be revealed. Another application is the attack CDTZero where it is already enough to differentiate zero from non-zero outputs of the BaseSampler, see Section 3.6.7.

3.6.2 HalfKey^N

This attack requires the adversary to inject two control flow faults into the sampling of coefficients for f and g. For the example presented here, the goal is to abort the loop after exactly half of the coefficients were sampled. Assuming that all coefficients are initialized with zero, we get:

$$f = f_0 + f_1 x + \cdots + f_{n/2-1} x^{n/2-1} + 0 + \cdots + 0,$$
$$g = g_0 + g_1 x + \cdots + g_{n/2-1} x^{n/2-1} + 0 + \cdots + 0.$$

Let h be the corresponding public key, defined as usual via $f^{-1}g = h \bmod q$. This relation can be rewritten using the vector and matrix operators Vec and Mat discussed in Section 2.1.5. For clarity we introduce the notation:

$$\mathrm{Mat}(h) = H = \begin{bmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{bmatrix}, \qquad \mathrm{Vec}(f) = (f', 0), \qquad \mathrm{Vec}(g) = (g', 0),$$

where the H_{ij}'s are blocks of size $n/2 \times n/2$, f' and g' are the coefficient vectors of the non-zero parts of f and g and 0 is the zero-vector of appropriate dimension. The relation $fh = g \bmod q$ (that follows from the definition of h) can now be rewritten as a vector-matrix product:

$$(f', 0) \cdot \begin{bmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{bmatrix} = (g', 0) \mod q.$$

Block-wise multiplication gives us two linear systems of equations:

$$f' \cdot H_{00} = g' \mod q, \qquad f' \cdot H_{01} = 0 \mod q.$$

The second equation only contains f' as unknown and can be solved using standard methods from linear algebra. Since it is a homogeneous system and f' is non-zero we expect a linear subspace of solutions. The highly unlikely case that all of the sampled coefficients of f' are also zero cannot happen, because then f would not be invertible, which is a requirement.

Although we now have a number of possible solutions, we do know that the coefficients we are looking for come from a Gaussian distribution centered around zero and they should therefore be small compared

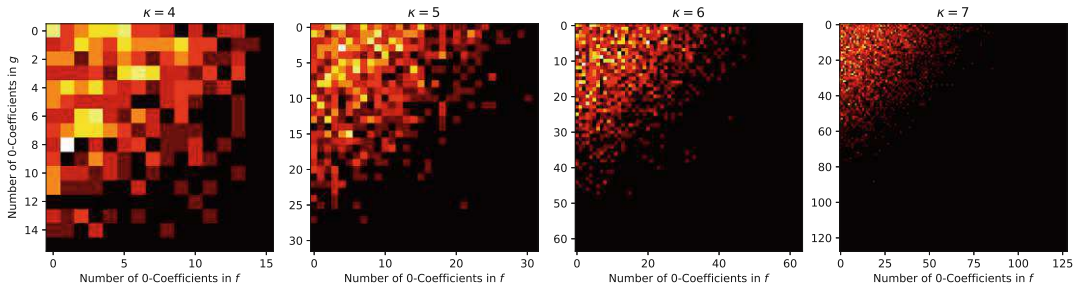

Figure 3: Probability of generating a full, valid secret key from polynomials f and g, where leading coefficients have been faulted to zero. A darker color indicates a lower probability. We expect that for higher security level $\kappa > 7$, the region with non-zero probability (in the top-left of the figures) will become even more restricted.

to $q = 12289$. Any reasonable solution for f' can then be substituted into the other system of equations. Solving for g' we may also get multiple solutions. Finding the correct one is again only a matter of choosing the solution with the smallest coefficients.

What this simple example does not take into account is the effect that zeroing coefficients has on the Gram-Schmidt norm of the secret key. While f and g naturally get smaller, tests show that F and G, on the other hand, have larger coefficients. In effect, this increases the overall Gram-Schmidt norm of the key. So a higher number of zeros in f and g generally decreases the probability that the norm stays below the bound that is required by the FALCON specifications (see Figure 3). In such a case, Keygen would simply start over and generate completely new polynomials, rendering the injected faults useless.

This negatively effects the attack, since injecting fewer zeros (to try to keep the rejection probability low) will lead to a system of equations that is harder to solve due to the higher number of unknown variables. Nonetheless it might still be possible to deduce the secret key from the reduced value space. One might compare this situation to a similar one proposed for LWE-problems[12], where it is possible to incorporate so-called hints, e.g., known coefficients, into the solution process. A detailed analysis of the possibilities for this attack and of the connection to LWE with hints will be left as an open question.

3.6.3 SmallLeaves^N, NarrowSampling^N

According to the FALCON specifications [FHK+20], if the standard deviation of the trapdoor sampler ffSampling is too low, it cannot be guaranteed that the sampler does not leak the secret basis. To see this, consider an alternative signature scheme where ffSampling were replaced by a simpler calculation, inspired by the round-off algorithm due to Babai [Bab86]:

$$\mathbf{z} = \mathsf{ffSampling}(\mathbf{t}, \mathsf{T}) \qquad \rightsquigarrow \qquad \mathbf{z} = \lfloor \mathbf{t} \rceil = \lfloor (c, 0)\mathbf{B}^{-1} \rceil.$$

The signatures derived from this have the form:

$$\mathbf{s} = (\mathbf{t} - \mathbf{z})\mathbf{B} = \big(\mathbf{t} - \lfloor \mathbf{t} \rceil\big)\mathbf{B},$$

where the expression in the parenthesis evaluates to a vector with coefficients in $[-1/2, 1/2]$ and so the signature lies in the parallelepiped $\frac{1}{2}\mathcal{P}(\mathbf{B})$, meaning that the distribution of sampled points is not independent of the secret $\mathbf{B}$.

[12] LWE (Learning With Errors) is a hard lattice problem that some PQC schemes (e.g., CRYSTALS-Dilithium) base their security on. We will not discuss the details here, the interested reader is referred to the extensive literature on the topic. For LWE with hints in particular, see [DSDGR20]

Notably, the signature schemes GGH and NTRUSign use versions of this round-off method and they were subsequently shown to be insecure by Nguyen and Regev [NR06], where the full key was recovered for several parameter sets.

To prevent this, FALCON uses the trapdoor sampler ffSampling that (among other improvements) applies a spherical Gaussian distribution to the sampled signatures. This approach effectively hides the parallelepiped as long as the standard deviation is high enough. If an injected fault now causes the standard deviation to significantly decrease (either by setting the leaves in the FALCON tree to a low value, or by directly faulting σ), the Gaussian distribution loses its hiding property and the scheme becomes vulnerable again.

3.6.4 BadDeterminism^A, ConstantHash^A

These are adaptations of an attack that was proposed for deterministic FALCON by Bauer et al. [BS23]. Deterministic FALCON removes the two sources of randomness in FALCON: the random salt r generated before each signing and the non-deterministic parts of ffSampling. In the original attack by Bauer et al., randomness is reintroduced into ffSampling via faults and a fault attack is mounted.

In our context, the goal is to start with standard (non-deterministic) FALCON, remove the randomness from the salt r, but leave ffSampling as is. This leads to different signatures $\mathbf{s}$ and $\mathbf{s}'$ for the same hashed message. With $(c, 0) - \mathbf{s} = \mathbf{zB}$ and similarly $(c, 0) - \mathbf{s}' = \mathbf{z}'\mathbf{B}$, after some calculations we get:

$$\mathbf{s}' - \mathbf{s} = (\mathbf{z} - \mathbf{z}')\mathbf{B}.$$

Now $\mathbf{z} - \mathbf{z}'$ is an integer vector and therefore $\mathbf{s} - \mathbf{s}'$ is a point in the lattice generated by the secret basis $\mathbf{B}$. With enough of these points it is possible to recover the secret basis using lattice reduction. Bauer et al. also show a way to greatly reduce the complexity of the post-processing, but unfortunately this method is not applicable to standard FALCON.

Note that faulting c directly instead of r has to be done via data fault and can probably not be achieved with a control flow fault. Even though it is theoretically possible to abort the hashing after the first iteration, the attack described above only works if the challenges are exactly equal. The probability for this to happen by chance (after successful loop abortion) are quite low with $1/q \approx 0.008\%$.

Another advantage of targeting r instead of c is the following: Since r is part of the signature output (s, r), the resulting challenge can be recalculated during signature verification without a problem. On the other hand, a faulted c cannot be reliably reproduced from public values and would very likely lead to a rejected signature, making it quite easy to detect an attack.

3.6.5 MultFFT^P

Karabulut and Aysu [KA21] present this attack on the multiplication in the FFT domain. The specific target is $\mathsf{FFT}(c) \odot \hat{f}$, where c is a known polynomial (the message hash) and $\hat{f}$ is part of the secret key. The multiplication in the FFT domain is performed on individual floating-point coefficients of $\mathsf{FFT}(c)$ and $\hat{f}$ and consists of multiplication of the mantissas, addition of the exponents and handling of the signs. All three of these components are targeted separately. The attack on the exponent and on the sign are relatively straightforward, while the mantissa needs a more refined approach.

When the two mantissas are multiplied, they are first split into 27 high- and 25 low-order bits, $A + B$ and $C + D$, corresponding to a single FFT coefficient of c and f respectively. The implemented multiplication follows the distributive law: $(A + B)(C + D) = AC + AD + BD + BD$. The immediate targets are now the multiplications BD and AC. During this phase (the extend phase), false positives can occur when guessing the value of C and D from measurements. In the next phase (the prune phase), the additions of the four parts are taken into account and false positives can be corrected.

Guerreau et al. further improved this attack [GMRR22]. First they take into account that the recovered values represent floating-point coefficients of $\hat{f}$. So, when applying the inverse transform,

$f = \mathsf{FFT}^{-1}(\hat{f})$, a slight error in the recovered values will result in non-integer polynomial coefficients of f. As long as the error is not too high, simply rounding will still give the correct integer coefficients. This reduces the necessary precision to less than 10 bits per FFT coefficient.

Two more improvements concern a redundancy in multiplication of complex numbers and noise mitigation by grouping measurements for similar values. In summary, the necessary number of traces could be lowered with this improved attack approach.

3.6.6 RejectionLeakage^{N}

In a very general sense the idea behind Falcon (as well as other signature algorithms) is to find a short vector that satisfies certain conditions. The signer relies on information stored in the secret key to enable the efficient search or generation of such short vectors. Faulting the secret key components f, g, F, and G before or during signing therefore usually results in the inability to generate signatures. Sometimes a small fault (for example setting a single coefficient to zero) only lowers the probability for finding a short vector but does not make it entirely impossible.

In the case of Falcon, faults to a single coefficient of f, g, F or G result in longer runtimes, because more signatures have to be created until a sufficiently short one is found. The idea now is to measure this timing information (or alternatively to count the number of rejections) and work out the faulted coefficient from it. Initial tests show that when setting a coefficient to zero, there is indeed a correlation between the absolute value of the original coefficient and the probability that a generated signature is shorter than the prescribed threshold. Whether this timing information is enough to recover secret key coefficients needs to be studied further.

3.6.7 CDTZero^{P,A}

This attack was originally discussed by Guerreau et al. [GMRR22] and further developed by Zhang et al. [ZLYW23]. It is a side-channel attack targeting the BaseSampler output z_0. Using single-trace side-channel analysis of the CDT-sampler an attacker can gain knowledge of z_0 with almost perfect accuracy. When $z_0 = 0$, we know that SamplerZ will output either 0 or 1 for the corresponding coefficient of $\mathbf{z}$. Usually the value space would be much broader, with $z_0 \in [-18, 19]$. This then results in a limited range for the signature $\mathbf{s}$ in the direction of the dimension where this output was produced.

The attacker can collect a number of signatures and group them according to which coefficient was produced from a 0-output of the BaseSampler. Geometrically speaking, the signatures in those groups are restricted to slices of the usual spherical Gaussian distribution. Each slice is roughly orthogonal to one of the row-vectors in $\tilde{\mathbf{B}}$, the Gram-Schmidt orthogonalized secret basis. It is now possible to approximate those vectors by solving a variant of the so-called Hidden Parallelepiped Problem. (We will define, solve and use the original form of this problem ourselves in Section 4.2.1.)

If the recovered vector is an approximation of $\tilde{\mathbf{b}}_0$, it directly corresponds to the secret polynomials $(g, -f)$. Approximations of other rows, however, can also be used in further post-processing steps.

As an alternative, this attack could be realized as a fault attack, where an adversary would fault the values z_0 to zero and apply the same post-processing. Furthermore, Zhang et al. pointed out that a combined attack with HalfGaussSign is possible, as described in the next section.

3.6.8 HalfGaussSign^{P,A}

This attack was first described by Zhang et al. [ZLYW23]. The idea is similar to that of CDTZero: sorting signatures according to certain intermediate values in SamplerZ. The point of interest here is the sign bit b, that gets applied to the output z_0 of the BaseSampler via:

$$z = b + (2b - 1) \cdot z_0 = \begin{cases} -z_0, & b = 0, \\ 1 + z_0, & b = 1. \end{cases}$$

Side-channel analysis can reveal whether b is zero or one. With this information, signatures can be sorted into two disjoint half-spaces $\mathcal{H}^+ = \{\mathbf{s} : \langle \mathbf{s}, \mathbf{b}_0 \rangle \geq 0\}$ and $\mathcal{H}^- = \{\mathbf{s} : \langle \mathbf{s}, \mathbf{b}_0 \rangle < 0\}$, where $\mathbf{b}_0$ is the first row of the FALCON secret matrix $\mathbf{B}$. Multiplying all signatures from one of these half-spaces with -1 moves them to the other half-space. So, without loss of generality, all signatures now lie in $\mathcal{H}^+$.

The recovery of $\mathbf{b}_0$ is done in two steps. First, a rough approximation $\mathbf{v}$ of the direction of $\mathbf{b}_0$ is found via statistical methods. Now we filter signatures from the half-space $\mathcal{H}^+$ by intersecting it with the slice $\{\mathbf{s} : |\langle \mathbf{s}, \mathbf{v} \rangle| \leq a\}$ for some threshold $a > 0$. A similar method to the HPP can be applied to learn $\mathbf{b}_0$ from this collection of signatures.

Zhang et al. [ZLYW23] also present a combined approach of HalfGaussSign with the CDTZero attack. Essentially, the side-channel information from the sign bit b and from the BaseSampler output z_0 can be used to filter signatures from an even more restricted area of space. This increases the success probability of the attack and lowers the number of necessary traces.

3.6.9 HiddenSlice^A, HiddenHalfspace^A

These attacks apply the post-processing of CDTZero [GMRR22], [ZLYW23] and HalfGaussSign [ZLYW23] to partial knowledge of the preimage vector $\mathbf{t}$ instead of $\mathbf{z}$. Whether the necessary information can be obtained through side-channel analysis in practice is not known at the moment. Also, the application of these methods to this new use case solely relies on the observation that the equations:

$$\mathbf{zB} = (c, 0) - \mathbf{s} \quad \text{and} \quad \mathbf{tB} = (c, 0),$$

contain public and secret values in the same configurations. The success of the attack might also depend on the distributions of the variables involved and is uncertain for now.

3.6.10 GaussShift^N

For this attack we assume that an adversary can control a fault vector $\boldsymbol{\delta}$ that gets added to $\mathbf{z}$ or $\mathbf{t}$ respectively. In the first case, consider a faulted $\mathbf{z}' = \mathbf{z} + \boldsymbol{\delta}$. A signature is derived in the usual way, $\mathbf{s}' = (\mathbf{t} - \mathbf{z}')\mathbf{B}$. After some rearranging we see that $\mathbf{s}'$ is comprised of the unfaulted signature $\mathbf{s}$ and an error term:

$$\mathbf{s}' = (\mathbf{t} - \mathbf{z}')\mathbf{B} = (\mathbf{t} - \mathbf{z} - \boldsymbol{\delta})\mathbf{B} = \mathbf{s} - \boldsymbol{\delta}\mathbf{B}.$$

The usual approach would be to mount a differential fault attack, where the unfaulted signature $\mathbf{s}$ could be used to calculate the error term as $\boldsymbol{\delta}\mathbf{B} = \mathbf{s} - \mathbf{s}'$. However, the randomized setup of FALCON prevents us from gaining knowledge of $\mathbf{s}$. We can work around this problem by considering the distributions of the values involved.

From the specifications we know that FALCON signatures are distributed according to a discrete Gaussian distribution $D_{\Lambda(\mathbf{B})+(c,0),\sigma,0}$ centered around the origin. If an attacker can repeatedly inject the same fault vector, then the constant error term $\boldsymbol{\delta}\mathbf{B}$ shifts the center of the distribution to $D_{\Lambda(\mathbf{B})+(c,0),\sigma,-\boldsymbol{\delta}\mathbf{B}}$.[13] Now, from enough faulted signatures the center of the distribution can be calculated with relatively high accuracy.

The question remains how to choose a suitable $\boldsymbol{\delta} = (\delta_0, \delta_1)$ in order to extract useful information. For this we consider the vector-matrix product explicitly and in FFT representation:

$$\hat{\boldsymbol{\delta}}\hat{\mathbf{B}} = (\hat{\delta}_0 \odot \hat{g} + \hat{\delta}_1 \odot \hat{G}, -\hat{\delta}_0 \odot \hat{f} - \hat{\delta}_1 \odot \hat{F}).$$

Now an attacker can choose, for example, $\hat{\delta_0} = (1, \ldots, 1)$ and $\hat{\delta_1} = (0, \ldots, 0)$ to directly reveal $\boldsymbol{\delta}\mathbf{B} = (g, -f)$ (see Figure 4). If we assume a less powerful attacker model that can only inject a single fault at a time, the secret key can still be recovered, albeit one FFT coefficient after the other. A choice of $\hat{\delta_0} = (1, 0, \ldots, 0)$

[13] This is a distribution with the same support as before, but shifted center. It is not to be confused with $D_{\Lambda(\mathbf{B})+(c,0)-\boldsymbol{\delta}\mathbf{B},\sigma}$, where the support is shifted, but not the center, which would still be 0.

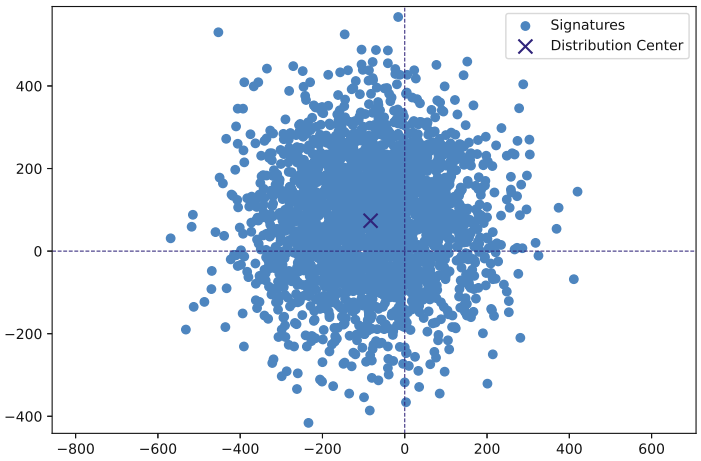

Figure 4: Example of the GaussShift attack with $\kappa = 1$, projected down from four to two dimensions. We generated 2 500 signatures with the fault vector $\boldsymbol{\delta} = ((1,1),(0,0))$ added to $\mathbf{z}$. The resulting distribution of signatures has an empirical center of $(-83.0376, 73.6172, 15.7484, -11.6612)$, which closely matches the coefficients of the corresponding key components $g = 83 - 72x$ and $f = 22 - 11x$, in accordance with the theoretical calculations.

and $\hat{\delta_1} = (0, \ldots, 0)$ would reveal the first FFT coefficient of $\hat{g}$, for example.

A variant of this attack injects the same fault, but on $\mathbf{t}$ instead of $\mathbf{z}$. When calling ffSampling with $\mathbf{t}' = \mathbf{t} + \boldsymbol{\delta}$ as input, the fault propagates to $\mathbf{z}'$ and finally $\mathbf{s}' = (c, 0) - \mathbf{z}'\mathbf{B}$. Here it is important to once again note the actual signature calculation does not include $\mathbf{t}'$, but the (in this context unfaulted) challenge c. We have:

$$(c, 0)\mathbf{B}^{-1} = \mathbf{t} = \mathbf{t}' - \boldsymbol{\delta} \quad \Rightarrow \quad (c, 0) = \mathbf{t}'\mathbf{B} - \boldsymbol{\delta}\mathbf{B}.$$

And from this we get:

$$\mathbf{s}' = (c, 0) - \mathbf{z}'\mathbf{B} = \mathbf{t}'\mathbf{B} - \boldsymbol{\delta}\mathbf{B} - \mathbf{z}'\mathbf{B} = (\mathbf{t}' - \mathbf{z}')\mathbf{B} - \boldsymbol{\delta}\mathbf{B}.$$

Recall that $\mathbf{t}'$ and $\mathbf{z}'$ are an input-output pair of ffSampling, so the term $(\mathbf{t}' - \mathbf{z}')\mathbf{B}$ follows the distribution $D_{\Lambda(\mathbf{B})+(c,0),\sigma,0}$. We conclude that the signatures are distributed according to $\mathbf{s}' \sim D_{\Lambda(\mathbf{B})+(c,0),\sigma,-\boldsymbol{\delta}\mathbf{B}}$, resulting in exactly the same situation as when faulting $\mathbf{z}$.

3.6.11 BEARZ^P

McCarthy et al. [MHS+19] originally described this attack. The main idea for this attack is to force the complete first and parts of the second polynomial in $\mathbf{z} = (z_0, z_1)$ to be zero. If this is the case, the resulting signature component s_2 generates a sub-lattice of the lattice generated by F. From a few hundred faulty signatures, F can be recovered using lattice reduction, and from this an adversary can calculate the full secret key.

In order to achieve the goal of zero-coefficients in $\mathbf{z}$, different methods are proposed. First, the attacker can abort the sampling of $\mathbf{z}$ in the top-level recursion after z_1 is sampled. This leaves z_0 unsampled, i.e., all its coefficients are zero (assuming that it is initialized with zero-coefficients). Alternatively, it is possible to overwrite the required coefficients with zero manually via a data fault attack. Another option is to abort mid-recursion. However, then it is required to work out which coefficients are merged in the recursion, so that the right ones in $\mathbf{z}$ are set to zero in the end.

Depending on the fault strategy, the number of forced zero-coefficients varies. The higher this number the easier and faster the post-processing becomes. Additionally, the number of traces needed for a

successful attack goes down when more coefficients are set to zero. For a typical attack less than a few hundred traces are needed to fully recover the secret key.

It has to be noted that this attack most likely requires an additional fault to skip the norm check $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ in Sign. Since the attacker changes $\mathbf{z}$, its distance to the preimage $\mathbf{t}$ is generally increased, making the resulting signature longer. Tests have shown that even a single faulted coefficient of $\mathbf{z}$ can cause the signature to be above the required norm and therefore trigger recomputation. This makes the attack less practical, as there are now at least two faults that need to be injected per signature.

3.6.12 StdFault

This is not an actual attack, but rather a reminder that the standard deviation of the Gaussian distribution used in ffSampling is generally quite vulnerable. Since a wide range of variables is necessary to produce the exact distribution, faults to any of them may influence the sampler output. Whether such an influence can be leveraged in an attack is sometimes difficult to assess and strongly depends on the circumstances. It is known that lowering the standard deviation below a certain point opens the possibility of attacks like SmallLeaves and NarrowSampling. However, it will be left as an open question if and how an attack on the variables with the StdFault designation can be mounted.

3.6.13 Verification Attacks

The attacks that target Verify are all rather simple in concept, but we still include them for completeness.

ZeroChallenge^N, ShortenedSignature^N. The goal is to make s_1 as short as possible in order to pass the norm check.

An adversary could choose s such that the decompressed s_2 is zero. To fault c to zero, either a data fault directly to c or a control flow fault during the call to HashToPoint could be injected. When s_1 is calculated with these faulted values it will result in zero as well, so the norm check passes and the signature is accepted.

An alternative approach is to directly fault s_1 to zero or any other sufficiently short polynomial.

LargerBound^N, SkipCheck^N. Instead of forcing a short signature, one could raise the bound β or skip the norm check entirely.

KeyReplacement^N. An attacker generates their own key pair (sk', h') and signs an arbitrary message with sk'. If they are then able to fault the verifier's public key h to h', the signature will be accepted.

4 Attack Analysis of NarrowSampling

From the new vulnerabilities that we found in the previous Section 3, we choose the NarrowSampling fault attack for a detailed analysis. Our choice is motivated by the belief that NarrowSampling is not only one of the more promising attacks but also because the theory behind the different attack phases is rather interesting and well worth presenting.

We structure the analysis into several subsections. First we locate the point of interest for the fault injection, simulate a fault and verify experimentally that some information will be leaked by the faulted signatures. Next, we explain the experimental results and find what information specifically we get from the fault. Then we get to the post-processing, which is split into two distinct phases: approximate key recovery, followed by a rounding step and finally full key recovery.

Whenever we perform experiments or simulations, we rely on the FALCON reference implementations that are provided on the official website[14]. For quick tests and proofs-of-concept we use the Python implementation[15], as it is generally easier to work with and get quick initial results. On the other hand, the C implementation[16] is much faster, which is an advantage when generating a large number of signatures, and we believe that it is closer to what real-world applications of FALCON will be like, which is why we use it for more involved tests.

4.1 Uncovering a Vulnerability

As stated before in Section 2.3.3, in contrast to other signature schemes such as GGH and NTRUSign, regular FALCON signatures do not reveal secret information, thanks to a discrete Gaussian distribution used in the trapdoor sampler ffSampling and its subroutine SamplerZ. The value range for the standard deviation is finely tuned to be neither too low nor too high. If the standard deviation were to fall below a certain bound ($\sigma_{\min}$ in the specifications, see Table 1), the distribution loses its hiding property and signatures could potentially reveal secret information. And that is precisely the goal of the NarrowSampling attack.

4.1.1 Point of Interest

Regular FALCON signatures are distributed according to $D_{\Lambda(\mathbf{B})+(c,0),\sigma}$, where $\mathbf{B}$ is the secret basis, c is the hashed message and σ is the standard deviation. The parameter σ is never directly used in Sign, but the FALCON tree T is normalized during Keygen by applying:

$$\mathsf{leaf.value} \leftarrow \sigma/\sqrt{\mathsf{leaf.value}},$$

on each leaf of the initial tree, see Lines 6 and 7 in Algorithm 1. This is a significant advantage for an adversary, as they only need to inject a fault once during Keygen, altering the value of σ. As a consequence, all leaf values of the tree will be effected and signatures sampled with the faulted tree will exhibit the faulted distribution.

4.1.2 Simulating Fault Injection

In the Python reference code, fault injection is straightforward. The parameter σ can directly be set to any arbitrary value and a key pair (including the tree T) can be generated.

As an initial test we want to compare the distributions of signatures for different values of σ visually. We set the security level to $\kappa = 1$ in order to be able to reasonably plot the results. Then we generate a regular key $(\hat{\mathbf{B}}, \mathsf{T})$ with the original $\sigma \approx 144.8$. From this we then generate faulted keys that have the same matrix $\hat{\mathbf{B}}$, but where the trees are normalized with faulted $\sigma' \in \{0.7\sigma, 0.4\sigma, 0.1\sigma\}$. Each of these

[14] https://falcon-sign.info/
[15] https://github.com/tprest/falcon.py
[16] https://falcon-sign.info/Falcon-impl-20211101.zip

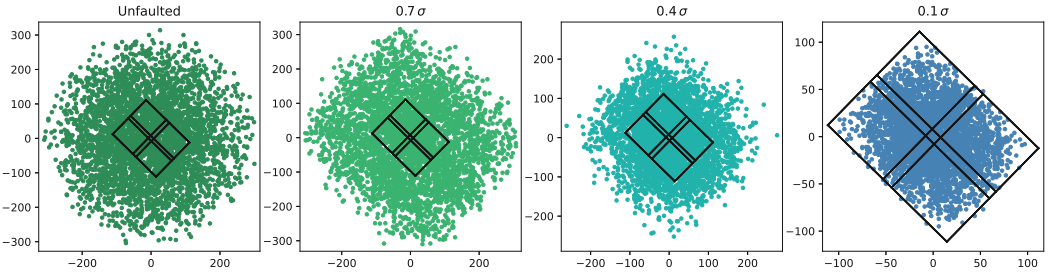

Figure 5: Signatures generated from a regular FALCON key (left), together with faulted signatures with $\sigma' = 0.7\,\sigma$ (middle left), $\sigma' = 0.4\,\sigma$ (middle right) and $\sigma' = 0.1\,\sigma$ (right). The corresponding key components are $f = -46 + 53x$, $g = -52 - 45x$, $F = 33 - 63x$, and $G = -53 - 100x$. Superimposed are the edges of the parallelepiped (projected down from four to two dimensions) that the signatures approximate, see Section 4.1.3. The leftmost figure also illustrates that the signature distribution is independent of the secret key coefficients in the unfaulted case.

keys is then used to sign a number of arbitrary messages. We make sure to skip the signature compression (Algorithm 2, Line 10), since we want to plot the polynomial form of signatures.

In Figure 5 the unfaulted signatures depict the expected spherical Gaussian distribution. The distribution of faulted signatures start to approximate a parallelepiped (projected down to two dimensions for plotting). For the lowest tested value of $0.1\,\sigma$, the parallelepiped is already clearly visible.

Repeating the experiment with the C implementation we observe the same general results. However, it is not as straightforward to reproduce the fault.

The C code provides a few options on how to handle floating point values and operations. Mainly, there a two modes to chose from: FPNative and FPEmu. FPNative uses the native C data type `double` for floating point arithmetic. On the other hand, FPEmu encodes floating point values into the C data type `uint64_t` and performs calculations based on the integer types `uint64_t` and `uint32_t`. Independently of the mode, FALCON reads the value of σ from a hard-coded table.[17] So in theory, it should not make a difference whether emulated or actual floating point arithmetic is used, as we can inject a fault by changing the value in either table. In practice, however, it turns out that the FPEmu mode is quite susceptible to changes of parameters. This is because many of the custom subroutines written to handle the encoded floating point values suffer from overflows and similar issues when called with inputs outside of the intended value ranges.

An example of this is the function `fpr_trunc` used to round floating point values to integers. It is called as a subroutine of BerExp, which in turn controls the rejection loop in SamplerZ. If σ is faulted to a value below $\sigma_{\min}$, then $ccs = \frac{\sigma_{\min}}{\sigma} > 1$. A specific line then calls `fpr_trunc(fpr_mul(ccs, fpr_ptwo63))`, which roughly corresponds to the calculation $\lfloor ccs \cdot 2^{63} \rfloor$. The problem is that `fpr_trunc` only takes inputs from the range $[-2^{63} + 1, 2^{63} - 1]$. Since ccs is faulted to be greater than 1, the argument lies outside the intended range. In the end, this leads to a situation where the rejection sampling step of SamplerZ virtually always rejects and never produces an output.

This means that the FPEmu mode of FALCON already provides some protection against NarrowSampling, in the sense that it denies service if an adversary attempts this attack. No sensitive information is leaked, since no signatures are produced. Initial tests showed that the FPNative mode can still produce signatures, even for very low values of σ (with scaling factors as low as 0.1). This led us to using FPNative to generate faulted signatures for all further simulations of NarrowSampling.

[17] Depending on the implementation, the table might contain $1/\sigma$ to reduce the number of divisions, as is the case for the C reference code. An attacker would need to take this into account when injecting a fault.

4.1.3 Consequences of Fault Injection

After successfully simulating a fault and showing that it has an effect on the distribution of signatures, we still need to show that the altered distribution actually contains sensitive information which can be exploited in an attack. The key to this lies in the shape that the faulted signatures approximate, which is a parallelepiped spanned by a matrix closely related to the Falcon secret key $\mathbf{B}$.

Consider Falcon's trapdoor sampler ffSampling, which was first introduced by Ducas and Prest [DP15] in a non-randomized variant. They show that when given a basis $\mathbf{B}$ and a target $\mathbf{t}$, the sampler returns a vector $\mathbf{z}$ close to $\mathbf{t}$, such that:

$$\mathsf{V}\big((\mathbf{z}-\mathbf{t})\mathbf{B}\big) \in \tfrac{1}{2}\cdot\mathcal{P}\big(\mathsf{GSO}(\mathsf{M}(\mathbf{B}))\big). \tag{3}$$

Here, the operators V and M are permutations on coefficient vectors and matrices, which we will define in the following.

Let $p \in \mathbb{Z}[x]/\phi$ be an integer polynomial. From Section 2.1.5 we already know that we can identify p with its coefficient vector. In the same section we also introduced the split operator, that splits p into polynomials p_0 and p_1 of half the degree, going down one step in the tower of rings. Repeated splitting results in a permuted coefficient vector, as seen in the example in Figure 2. We define $\mathsf{V}(p)$ as the "fully permuted" coefficient vector i.e., the end result of splitting when we reach the bottom of the tower. When applying V to a vector of polynomials, as in Equation 3, we just apply it to each entry separately (which, importantly, is not the same as applying it to the combined coefficient vector).

The operator M can be derived from V:

$$\mathsf{M}(p) = \begin{bmatrix} \mathsf{V}(a) \\ \mathsf{V}(xa) \\ \vdots \\ \mathsf{V}(x^{n-1}a) \end{bmatrix},$$

or in other words, we apply V to each row of $\mathrm{Mat}(a)$. Just like with V, when M is used on a matrix with polynomials as entries, it is applied to each polynomial separately.

In the context of Falcon and the NarrowSampling attack, the conclusion is the following: By lowering the value of σ through fault injection, the discrete Gaussian distribution that is used in ffSampling loses its hiding property and the sampler starts to resemble its non-randomized version. Thus the signatures approximate a parallelepiped as in Equation 3 and the matrix that spans it is derived from the secret key matrix $\mathbf{B}$. Crucially, the first row of $\mathsf{M}(\mathbf{B})$ directly contains $(\mathsf{V}(g), \mathsf{V}(-f))$:

$$\mathsf{M}(\mathbf{B}) = \begin{bmatrix} \mathsf{M}(g) & \mathsf{M}(-f) \\ \mathsf{M}(G) & \mathsf{M}(-F) \end{bmatrix} = \begin{bmatrix} \mathsf{V}(g) & \mathsf{V}(-f) \\ \vdots & \vdots \\ \mathsf{V}(x^{n-1}g) & \mathsf{V}(-x^{n-1}f) \\ \mathsf{V}(G) & \mathsf{V}(-F) \\ \vdots & \vdots \\ \mathsf{V}(x^{n-1}G) & \mathsf{V}(-x^{n-1}F) \end{bmatrix}.$$

Furthermore, the Gram-Schmidt orthogonalization preserves the first row vector, so $(\mathsf{V}(g), \mathsf{V}(-f))$ is still the first row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. Therefore it seems plausible that an adversary can learn sensitive information from analyzing the distribution of faulted signatures.

4.2 Approximate Key Recovery via the Hidden Parallelepiped Problem

After we established that faulted signatures leak sensitive information, we need to find a way to process them. The situation is not unlike some other attacks from prior works, described in Section 3.6, such as CDTZero and HalfGaussSign. In both those attacks, the adversary collects signatures and wants to extract some geometric information from them. CDTZero in particular uses methods that were originally presented by Nguyen and Regev [NR06], which can be directly applied to the NarrowSampling attack.

4.2.1 The Hidden Parallelepiped Problem

Nguyen and Regev [NR06] introduced the so-called Hidden Parallelepiped Problem (HPP) and present a solution method. The problem can be summarized as follows: Let $\mathbf{V} \in \mathrm{GL}_d(\mathbb{R})$ be a full-rank, $d \times d$-matrix, with row vectors $\mathbf{v}_1, \ldots, \mathbf{v}_d$ and let $\mathcal{P}(\mathbf{V})$ be the parallelepiped spanned by $\mathbf{V}$. Now, given samples $\mathbf{z}_1, \ldots, \mathbf{z}_N$ drawn uniformly from $\mathcal{P}(\mathbf{V})$, find an approximation of the rows $\pm\mathbf{v}_i$.

A solution to this problem can be obtained from the following steps. For a detailed explanation of the process, including proofs and justifications of all statements, we refer to [NR06].

1. Compute an approximation of $\mathbf{G} = \mathbf{V}^\top\mathbf{V}$.
   It can be shown that $\mathbb{E}[\mathbf{z}^\top\mathbf{z}] = \frac{1}{3}\mathbf{V}^\top\mathbf{V}$. This means that we can simply take the average of $\mathbf{z}_i^\top\mathbf{z}_i$ over all samples and multiply by 3 to get an approximation of $\mathbf{G}$.

2. Compute the Cholesky factorization of $\mathbf{G}^{-1} = \mathbf{L}\mathbf{L}^\top$, where $\mathbf{L}$ is a lower triangular matrix.
   Both the inverse of $\mathbf{G}$ and its Cholesky factorization can be found using standard linear algebra.

3. Multiply the samples with the Cholesky factor, yielding $\mathbf{z}_i\mathbf{L}$.
   This transforms the parallelepiped into a d-dimensional hypercube with sidelength 2 centered around the origin. The matrix $\mathbf{C} = \mathbf{V}\mathbf{L}$ is then an orthogonal matrix in $\mathrm{O}_d(\mathbb{R})$ and it spans this hypercube $\mathcal{P}(\mathbf{C})$.

4. Compute approximations of rows $\mathbf{c}_i$ of $\pm\mathbf{C}$ via gradient descent.
   For this we define the k-th moment for an arbitrary, regular matrix $\mathbf{M} \in \mathrm{GL}_d(\mathbb{R})$ at a point $\mathbf{w} \in \mathbb{R}^d$ as:
   $$\mathrm{mom}_{\mathbf{M},k}(\mathbf{w}) = \mathbb{E}\left[\langle \mathbf{u}, \mathbf{w}\rangle^k\right] \in \mathbb{R},$$
   where $\mathbf{u}$ is a random vector uniformly distributed over $\mathcal{P}(\mathbf{M})$. The fourth moment of an orthogonal matrix (like $\mathbf{C}$) has a special property: Restricted to the unit sphere the global minimum of $\mathrm{mom}_{\mathbf{C},4}$ has a value of 0.2, it is obtained at $\pm\mathbf{c}_i$ and there are no other minima. Applying a standard gradient descent method to this function will therefore give us an approximation of one row $\pm\mathbf{c}_i$. Repeated descents with random starting points can result in the recovery of different rows.

5. Multiply the recovered rows with the inverse Cholesky factor: $\pm\mathbf{v}_i = \mathbf{c}_i\mathbf{L}^{-1}$.
   This transforms the hypercube back into the original parallelepiped and the rows $\mathbf{c}_i$ into $\pm\mathbf{v}_i$.

Instead of implementing an HPP solver from scratch, we rely in part on code provided by Guerreau et al. [GMRR22] from their git repository[18]. The attack from that paper (→ CDTZero) actually uses a variant of the HPP, but its implementation contains functions and methods that can be used in other, more general contexts. We take advantage of this by adapting their code to our needs.

4.2.2 The Gradient Descent in Detail

Before we investigate the output of the HPP solver, we take a closer look at the gradient descent phase that looks for minima of the target function $\mathrm{mom}_{\mathbf{C},4}$. It performs the following steps:

1. Choose a random starting point $\mathbf{w}$ on the unit sphere.

2. Evaluate $\mathrm{mom}_{\mathbf{C},4}(\mathbf{w})$.

3. Calculate the gradient $\nabla\mathrm{mom}_{\mathbf{C},4}$ at the point $\mathbf{w}$.

4. From $\mathbf{w}$ take a step in the opposite direction of the gradient. The size of the step is determined by the length of the gradient times some scaling factor. Normalize the result to get a point that lies on the unit sphere again. Update $\mathbf{w}$ with this new point.

[18] `https://github.com/mguerrea/FalconPowerAnalysis`

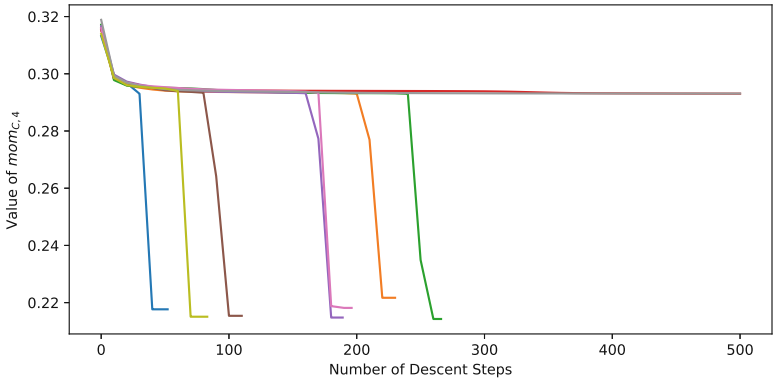

Figure 6: The progress of nine gradient descents with different random starting points in the HPP solver. The x-axis shows the number of descent steps, the y-axis shows the value of the target function $\mathrm{mom}_{\mathbf{C},4}$, which we want to minimize. The parameters for this specific test were $\kappa = 7$, $\sigma' = 0.15\,\sigma$ and we used 100 000 signatures.

5. Evaluate $\mathrm{mom}_{\mathbf{C},4}(\mathbf{w})$ with the updated point. If the target function has a lower value than before, continue with step 3. If, on the other hand, the target function has a higher value, stop the descent and return the previous point.

In order to abort descents that make little to no progress, we also return the current point if a specified maximum number of descent steps have passed. Also note that the target function and its gradient can only be approximated from the given samples (i.e., the faulted signatures) and not evaluated directly, since we have to assume that the matrix $\mathbf{C}$ defining the function $\mathrm{mom}_{\mathbf{C},4}$ is unknown to the adversary.

In Figure 6 we depict several example descents for security level $\kappa = 7$ and 100.000 signatures. We see that, after some initial progress in the first few steps, the descents slow down. Only after a while, some descents seem to find a minimum which they then quickly converge towards. After 500 steps, all remaining descents are aborted.

Further tests revealed a significant influence of the number of signatures on the average number of steps until a minimum is found. The relationship seems to be exponential, meaning that from κ to $\kappa + 1$ around four times as many signatures are needed to keep the average number of descent steps constant. This is an important factor when considering the total runtime of the attack (see also Section 4.4.4). Note that these numbers are very rough estimates and only based on some rudimentary tests, so we cannot say with certainty how descents behave for security levels upwards of $\kappa = 7$.

Looking at the value of the target function at the returned points we can already be quite certain that the actual minima are approximated reasonably well. From the theory we know that the minima of $\mathrm{mom}_{\mathbf{C},4}$ are exactly 0.2. Depending on the parameters of our tests we get quite close to this theoretical value (compare Figure 6). How well the HPP solver actually approximates rows of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ will be one of the central question in the next section.

4.3 Full Key Recovery from Approximations

The quality of the recovered, approximated rows is mostly controlled by one parameter, the sigma scaling factor. It influences the values in the leaves of the FALCON tree, which in turn have an effect on the distribution of faulted signatures. Since the tree generation and subsequent sampling are rather complex

and recursive processes, we split the analysis into two parts. First, how does the sigma scaling factor influence the distribution of the error on a coefficient-wise level? For this we resort to empirical tests and measurements. Second, how can we calculate the total error, given the error distribution on the individual coefficients? Here we are able to derive exact formulas.

4.3.1 The Error of Approximation

As a first step we want to verify whether the faulted signatures actually approximate a parallelepiped spanned by the rows of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. To this end we conduct multiple experiments of the following structure: We choose a security level κ, a scaling factor for σ and generate a faulty key pair. Using the secret key we then sign a number of arbitrary messages. Next, we input the collected signatures into the HPP solver and get some vector $\mathbf{r} = (r_0, r_1) \in (\mathbb{Z}[x]/\phi)^2$ as return, which we then permute via $\mathsf{V}(\mathbf{r}) = (\mathsf{V}(r_0), \mathsf{V}(r_1))$. If our previous assumptions are correct, this should correspond to one row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. Since we know the secret matrix $\mathbf{B}$ (which would not be the case in an actual attack scenario), we can easily check this by comparing $\mathsf{V}(\mathbf{r})$ to every row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. Keep in mind that the HPP solver can only find rows up to the sign, so when making the comparison we also always check $\mathsf{V}(-\mathbf{r})$.

When using the Euclidean distance as a measure of closeness, we find that for every $\mathbf{r}$ we get from the HPP solver, there is a row in $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ that is significantly closer to it than any other row. This motivates the definition of the term *error of approximation*:

$$E_A = \min_{0 \le i < 2n} \|\mathsf{V}(\pm\mathbf{r}) - \mathsf{GSO}(\mathsf{M}(\mathbf{B}))_i\|,$$

where $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))_i$ is the i-th row of the permuted, orthogonalized secret matrix $\mathbf{B}$. The difference between the recovered and the actual row, for which this minimum occurs will be denoted $\mathbf{e}$ and called the error vector.

The error of approximation contains all the accumulated error up to this point in the attack, so the lower the value, the shorter the error vector and the better the approximation. Most of the error comes from the fact that σ cannot be faulted to zero but must remain a positive value. This has the effect that the signatures do not disclose the parallelepiped exactly and the approximation of the target function $\mathrm{mom}_{\mathbf{C},4}$ in the gradient descent phase is not precise. The numerical nature of the gradient descent method itself also adds error, although only a negligible amount. Tests have shown that when two descents from different starting points converge on the same minimum, the distance between the results is usually lower than 10^{-4}.

4.3.2 Measuring the Coeffient-Wise Error of Approximation

Although we are able to find approximations of every row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ via HPP, going forward we are only interested in the first row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$, since it contains the coefficients of the secret key components f and g. It will be useful to know the coefficient-wise error of the approximation obtained from the HPP. We already know that our solution method recovers one random row from $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ per descent but we have essentially no control which one is found. This forces us to adopt a brute-force approach, where we run descents until the recovered row is closer to the first row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ than any other. When that happens, we calculate the coefficient-wise differences and append them to a list of samples. This way we get $2n$ samples per key, with $n = 2^\kappa$ corresponding to the security level. We repeat this with multiple keys until we have sufficiently many samples to make assumptions about the error distribution.

From these measurements we initially examined the distribution of errors visually and formulated the null-hypothesis stating that, for a fixed parameter set, the errors follow a centered normal distribution with the sample standard deviation as parameter. We performed the Kolmogorov-Smirnov test with significance level $\alpha = 0.05$ for each parameter set and concluded that we could indeed model the

κ	100k signatures				400k signatures				1.6M	6.4M
	$0.10\,\sigma$	$0.12\,\sigma$	$0.15\,\sigma$	$0.20\,\sigma$	$0.10\,\sigma$	$0.12\,\sigma$	$0.15\,\sigma$	$0.20\,\sigma$	$0.10\,\sigma$	$0.10\,\sigma$
2	0.481	0.629	1.235	3.521	0.280					
3	0.456	0.581	0.988	2.332	0.282					
4	0.437	0.491	0.766	1.845	0.241	0.311	0.590	1.659	0.172	
5	0.426	0.479	0.615	1.406	0.230	0.274	0.438	1.176	0.149	
6	0.423	0.461	0.568	1.119	0.222	0.273	0.333	0.944	0.118	0.074
7					0.217	0.236	0.307	0.762	0.110	0.073
8					0.229	0.233			0.115	0.064

Table 7: Measured standard deviation of single-coefficient error, rounded values.

coefficient-wise error as normal distributions.[19] The corresponding standard deviations calculated from our measurements are presented in Table 7.

4.3.3 Calculating the Error of Approximation

Now that we know the distribution of single error coefficients, we can derive the distribution of the quality of approximation, i.e., the length $\|\mathbf{e}\|$ of the error vector. To make things easier, we will find the distribution of the squared length $\|\mathbf{e}\|^2$. We define a random error vector of dimension d as:

$$\mathbf{e} = (e_0, e_1, \ldots, e_{d-1}), \quad \text{with} \quad e_i \sim \mathcal{N}(0, \sigma_e), \quad \text{iid}, \tag{4}$$

where σ_e will be the empirically measured, coefficient-wise standard deviation from Table 7. Furthermore, let $Z_i \sim \mathcal{N}(0,1)$ be iid. random variables of the standard normal distribution. Then for the squared Euclidean length of $\mathbf{e}$ we have:

$$E_A^2 = \|\mathbf{e}\|^2 = \sum_{i=0}^{d-1} e_i^2 = \sum_{i=0}^{d-1} (\sigma_e Z_i)^2 = \sigma_e^2 \sum_{i=0}^{d-1} Z_i^2 \sim \sigma_e^2 \chi_d^2,$$

where χ_d^2 is the chi-squared distribution with d degrees of freedom. The resulting distribution describes (the square of) the error of approximation.

Knowing this distribution, we can now make statements of the form "p percent of error vectors have a maximum length of x". For this we can simply use the cumulative distribution function of $\sigma_e^2 \chi_d^2$:

$$\mathbb{P}\big[\|\mathbf{e}\| \le x\big] = F_{\sigma_e^2 \chi_d^2}(x) = F_{\chi_d^2}\Big(\tfrac{x}{\sigma_e^2}\Big).$$

In the last equality we scaled the argument by a factor of $\frac{1}{\sigma_e^2}$. This is not strictly necessary for our calculations, but can be useful when working with software packages that only provide the standard chi-squared distribution.

Finally, combining the measurements of the coefficient-wise error with the theoretical considerations we get the values in Table 8.

4.3.4 Full Key Recovery through Rounding

The results form our measurements and calculations in the previous section show clearly that the key approximation increases in quality, when the sigma scaling factor goes down and the number of signatures

[19] The only instance where the p-value was below 0.05 was for $\kappa = 2$, $\sigma' = 0.10\,\sigma$ and 100.000 signatures. However, this single result is of little concern as we are mostly interested in higher security levels.

κ	100k signatures				400k signatures				1.6M	6.4M
	$0.10\,\sigma$	$0.12\,\sigma$	$0.15\,\sigma$	$0.20\,\sigma$	$0.10\,\sigma$	$0.12\,\sigma$	$0.15\,\sigma$	$0.20\,\sigma$	$0.10\,\sigma$	$0.10\,\sigma$
2	1.757	2.298	4.513	12.870	1.023					
3	2.210	2.819	4.795	11.316	1.369					
4	2.853	3.202	4.996	12.041	1.575	2.032	3.853	10.829	1.120	
5	3.783	4.254	5.457	12.485	2.043	2.356	3.979	10.447	1.327	
6	5.167	5.624	6.925	13.657	2.709	3.331	4.065	11.518	1.444	0.908
7					3.666	3.990	5.187	12.870	1.866	1.241
8					5.382	5.474			2.695	1.498

Table 8: 90%-percentiles for error of approximation. For a given security level κ and σ scaling factor, 90% of error vectors are shorter than the provided values.

goes up. This can be explained geometrically, as both aspects result in a better defined parallelepiped that the HPP solver can then pick up on. Interestingly, a *higher* security level also decreases the error. The reason for this is that the coefficients of the key components f and g follow a discrete Gaussian distribution:

$$f_i, g_i \sim D_{\mathbb{Z},\sigma_{\{f,g\}},0}, \quad \text{with} \quad \sigma_{\{f,g\}} = 1.17\sqrt{q/(2n)},$$

where the standard deviation is dependent on n. (See also Table 1 and Line 1 in Algorithm 4.) Therefore the coefficients tend to cover a narrower value space for higher security levels and so the absolute error gets smaller too. This effect gives the adversary an advantage when trying to mount the attack on the practical parameter sets $\kappa = 9$ and $\kappa = 10$.

Since the coefficients of key polynomials are integers, the simplest method of eliminating the error is rounding. We already know the distribution of the coefficient-wise error (Table 7) and so it is easy to derive the probability for a successful full key recovery through rounding. For this, we use the notation and definition of a random error vector from Equation 4. The probability for successful rounding is now simply the following:

$$\mathbb{P}[\forall 0 \le i < 2n : -0.5 < e_i < 0.5] = \big(\mathbb{P}[-0.5 < e_0 < 0.5]\big)^{2n} = \big(F_{e_0}(0.5) - F_{e_0}(-0.5)\big)^{2n}.$$

Here, F_{e_0} is the cumulative distribution function of the distribution $e_0 \sim \mathcal{N}(0, \sigma_e)$.

The calculations reveal that rounding can be a very successful strategy, given a sufficiently high number of signatures (see Figure 7). For example, for security level $\kappa = 8$ and 1.6 million signatures, rounding will give the correct secret key with a probability of around 99.3%. With 6.4 million signatures it even increases to practically 100%[20].

Indeed, during our measurements for the coefficient-wise error (in Section 4.3.2) we observed many cases, where rounding would be sufficient to recover the correct secret key. This verifies our assumptions and methods and shows the practicability of the NarrowSampling attack.

4.3.5 Recovering Equivalent Keys

Up to this point we analyzed the different aspects of the attack in a theoretical setting, where we always had full access to every value, specifically the secret key polynomials f and g. This allowed us to directly compare recovered rows from the HPP solver to the key. As a last step we propose a method of filtering the row containing the coefficients of f and g, without knowledge of any secret values. The basic idea is

[20] The exact calculated value is 99.999999999784%

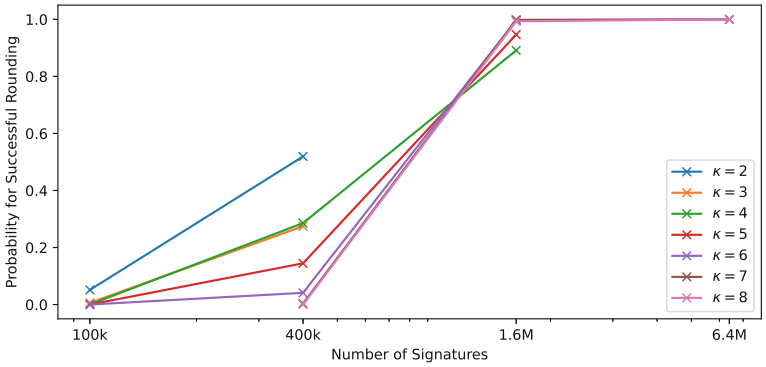

Figure 7: The probability that rounding results in a successful full key recovery, for different security levels and numbers of faulted signatures, with fixed $\sigma' = 0.10\,\sigma$.

to leave the question "Which row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ does the recovered row correspond to?" unanswered and instead ask "Does the recovered row function as a valid secret key?".

Say we have $\mathbf{r} = (r_0, r_1)$ that is an approximation of an unknown row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. We also assume that we used attack parameters that give us a sufficiently low error of approximation with high confidence, i.e., if $\mathbf{r}$ were to correspond to $(g, -f)$, rounding would be successful. After rounding and permuting we get $\mathsf{V}(\lfloor r_0 \rceil, \lfloor r_1 \rceil) = (g', -f')$, which we consider a secret key candidate. We now perform the checks in $\mathsf{NTRUGen}$ (see Algorithm 4) to make sure that the candidate is a valid Falcon key. This means we test the invertibility of f' as an element of $\mathbb{Z}_q[x]/\phi$ and check whether the Gram-Schmidt norm defined in Line 9 of Algorithm 4 is at most $1.17\sqrt{q}$. If the key candidate passes both checks, we know that a valid Falcon key (including the other components F and G as well as the tree T) can be produced from it. Finally, we compare f' and g' to the public key h. If they verify $f'h = g'$, then we can be sure to have found not just any secret key, but one that fits the known public key.

It is important to note that even after these tests, an attacker might not have recovered the original secret key, but an equivalent one. It can be shown that for each key pair $(\mathsf{sk}, \mathsf{pk})$, Falcon permits at least $2n-1$ additional secret keys. They cannot be distinguished from the original sk with the public key alone and therefore can be used interchangeably to generate signatures that pk will be able to verify. We briefly describe this phenomenon in the following.

Let $\mathbf{B}$ be a secret Falcon basis containing the polynomials $f, g, F, G \in \mathbb{Z}[x]/\phi$. Now define the matrix:

$$\mathbf{B}' = \begin{bmatrix} gx^k & -fx^k \\ Gx^{-k} & -Fx^{-k} \end{bmatrix},$$

where $0 \le k < 2n$. The polynomial entries of $\mathbf{B}$ satisfy the NTRU equation:

$$(fx^k)(Gx^{-k}) - (Fx^{-k})(gx^k) = \underbrace{x^k x^{-k}}_{=1} \cdot (fG - Fg) = q,$$

and due to:

$$(fx^k)^{-1}(gx^k) = f^{-1}x^{-k}gx^k = f^{-1}g = h,$$

they have the same relation to the public key as the original f and g. A final calculation shows that $\mathbf{B}'$ is orthogonal to $\mathbf{A}$ derived from the given public key:

$$\mathbf{B}'\mathbf{A} = \begin{bmatrix} gx^k & -fx^k \\ Gx^{-k} & -Fx^{-k} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ h \end{bmatrix} = \begin{bmatrix} x^k \cdot (g - f^{-1}gf) \\ x^{-k} \cdot (G - f^{-1}gF) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \mod q.$$

These three properties are sufficient to show that a signature generated from $\mathbf{B}'$ is a valid Falcon signature and verifiable with h. For $k = 0$ we simply get the original key $\mathbf{B}$, but for $0 < k < 2n$ we get $2n - 1$ different, equivalent Falcon keys.

In the context of the NarrowSampling attack this means that an adversary might recover a key that is not the original, but can still be used in the same way, with the same public key. In this case we still consider the attack successful, since valid signatures can be generated for arbitrary messages, and so the scheme is fully broken.

4.4 Practical Considerations for the Attack

Here we discuss the practical aspects of NarrowSampling, including possible attack scenarios, a precise fault description, time and memory requirements and scaling our simulations up to the full parameter sets of Falcon.

4.4.1 Fault Scenario

The point of interest for the fault injection in NarrowSampling was specifically chosen to minimize the necessary effort. A single fault to σ before the Falcon tree is generated will alter every value in its leaves and later the distribution of signatures. So once the key (and in particular the tree) is established, no further faults are required and every signature can be used as part of the key recovery. If Keygen and Sign closely resemble Algorithms 1 and 2, and in particular if the full Falcon tree is stored as part of the secret key, then NarrowSampling can be executed as described. However, the tree requires a rather large amount of memory (around 90 kB for $\kappa = 10$ according to the Falcon specification document), which is why the C reference implementation offers a mode of operation (that the specification document specifically recommends) to build the tree dynamically for each individual signature. Crucially though, this does not mean that the fault would need to be injected for every message that is signed. Since the value of σ is a fixed parameter and not derived from other values during runtime, it still needs to be read from a hard-coded table. So it suffices to change its value once as a so-called persistent fault (i.e., a fault that stays present for multiple executions or even indefinitely).

In a real-world attack scenario this could be executed in the following way: Say a device can run Falcon Sign for digital signatures. But rather than generating a secret key itself, it instead uses a key that was precomputed during the fabrication and then loaded and stored onto the device. If an attacker is able to inject a fault into the original execution of Keygen or alternatively fault the key that is stored on the device, then all signatures will exhibit a faulted distribution and be vulnerable to the NarrowSampling attack. Since the device cannot generate new key pairs on its own, the fault cannot be repaired and persists throughout the operational lifetime of the device (assuming that there are no other changes, e.g., updates to the device's software).

4.4.2 Scaling Factor for σ and Exact Fault Description

Another question is the exact value that σ is faulted to. For our tests we were able to freely chose almost any value and simulate the fault. In practice we would have to take into account the actual representation in register. The decimal values of $1/\sigma$ for levels $\kappa = 9$ and $\kappa = 10$ are:

$$\kappa = 9: \quad 1/\sigma = 0.006\,033\,669\,668\,157\,724\,103\,166\,806\,251\,095\,302\,2,$$
$$\kappa = 10: \quad 1/\sigma = 0.005\,938\,645\,309\,533\,115\,995\,025\,012\,433\,647\,748\,2.$$

When using the FPNative option (as discussed in Section 4.1.2) these numbers are represented by a bit-string consisting of the sign-bit, followed by an 11-bit exponent and a 52-bit mantissa:

$$\kappa = 9: \quad 0\ \ 01111110111\ \ 1000101101101100001011011110011001001100011111001010,$$
$$\kappa = 10: \quad 0\ \ 01111110111\ \ 1000010100110001111011110110001100010001101011100011.$$

If an adversary chooses $\sigma' = 0.10\,\sigma$, as we did for many of our tests, they would need to fault the mantissa (and possibly the exponent) to a completely different value. A more practical approach would be to only fault certain bits of the exponent to cause a shift by a power of two. For example, scaling by a factor of 8, i.e., $\sigma' = 0.125\,\sigma$, could be achieved by flipping the three underlined bits:

$$\kappa = 9: \quad 0 \;\; 01111111\underline{0}1\underline{0} \;\; 1000101101101100001011011110011001001100011111001010,$$
$$\kappa = 10: \quad 0 \;\; 01111111\underline{0}1\underline{0} \;\; 1000010100110001111011110110001100010001101011100011.$$

A scaling factor of 8 might also be the only practical option. The next higher one is $0.25\,\sigma$, which would result in an increased error of approximation, making the attack less efficient or even impossible. On the other hand, the next lower factor of $0.0625\,\sigma$ significantly reduces the speed at which messages can be signed, due to a higher rejection rate at the various checks in Sign. As long as there are no time restrictions, this could still be an option and we expect a reduced number of necessary signatures for full key recovery.

4.4.3 Memory Requirements

The collection of faulted signatures and the subsequent post-processing can be quite memory intensive. As an example, say an adversary generates 6.4 million signatures for security level $\kappa = 9$. For the key recovery (in particular the HPP solver) the signatures must be decompressed via $\mathsf{Decompress}(\mathsf{s}) = s_2$ and the other component s_1 must be calculated. So a full signature consists of two polynomials, each with $n = 2^\kappa$ integer coefficients. If we assume the usual 32-bit representation for integer values, this amounts to a total of:

$$6.4 \cdot 10^6 \cdot 2 \cdot 2^9 \cdot 32 = 209,715,200,000\,\text{bit} = 26.2144\,\text{GB}.$$

Instead of keeping all signatures in memory simultaneously, one could also read and/or decompress each signature only when it is needed. However, the HPP solver that we use accesses every signature for each step of the gradient descent to estimate the moment $\text{mom}_{\mathbf{C},4}$ as well as its gradient. The streamed approach would therefore have a significant computational overhead.

4.4.4 Runtime of the Attack

The runtime of the attack (after enough faulty signatures are collected) is mostly determined by two aspects: First the recomputation of the polynomials s_1 and s_2 from the compressed signature s. This part can benefit greatly from parallelization, since signatures are naturally independent of each other. In our tests the preparation of signatures never took longer than 30 minutes on a standard laptop using the methods provided in the Python reference implementation. On a server[21] with 16 parallel processes it took around 210 seconds to prepare 1.6 million signatures of level $\kappa = 8$. For lower κ and fewer signatures the runtime will of course be even lower.

The most time-consuming step in the post-processing of NarrowSampling is the gradient descent phase of the HPP solver. A typical, individual descent usually takes around 5 minutes on our hardware, for $\kappa = 8$ with 1.6 million signatures and $\sigma' = 0.1\,\sigma$, but without parallelization. Of course, the runtime can vary in either direction, depending on the security level, the number of signatures and the sigma scaling factor. When implementing this process we can utilize parallelization again. The descents are independent of each other and only need to access the signatures without altering them. We opted for a strategy where we ran multiple descents simultaneously, collecting all their outputs. Then (for the measurements of the coefficient-wise error of approximation in Section 4.3.2) we tested if any of them corresponded to the first row of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$. If the first row was not among the recovered rows, we would start another batch of descents. The same parallelization can be used in an actual attack scenario, where the adversary runs batches of descents until a valid secret key is found.

One aspect that we did not consider in detail is the fact that the number of rows of $\mathsf{GSO}(\mathsf{M}(\mathbf{B}))$ increases exponentially with the security level, doubling from κ to $\kappa + 1$. So the probability that the

[21] AMD Ryzen Threadripper PRO 5975WX 32-Cores

row which contains an approximation of the secret key is recovered by the HPP solver is lower for each individual descent. The situation is similar to the so-called Coupon Collector Problem, a well-studied problem in the field of probability theory. The interested reader is referred to the general mathematical literature about this problem, but we will leave the analysis of the probabilistic aspects of row recovery as an open question.

4.4.5 Scaling the Attack to Practical Security Levels

During our simulations we only ever considered security levels up to $\kappa = 8$. The recommended security levels $\kappa = 9$ and $\kappa = 10$ (corresponding to the security levels I and V as defined by NIST [NIS16]) were left out mostly due to the higher time and memory requirements. Throughout the analysis of NarrowSampling however, we collected enough data and gained practical experience to give at least a rough suggestion for suitable attack parameters.

From our measurements and calculations we estimate that not more than 10 million signatures with $\sigma' = 0.10\,\sigma$ should be sufficient to recover the secret key for both $\kappa = 9$ and $\kappa = 10$. This estimation is mostly based on the numbers in Table 7 and Figure 7. We see that the standard deviation of the error of approximation is already very low for levels $\kappa \in \{6, 7, 8\}$ and as discussed in Section 4.3.2 we expect it to decrease even further for higher levels. More signatures could additionally speed up the gradient descent if runtime is a limiting factor.

4.5 Discussion of Countermeasures

We briefly discuss countermeasures against the NarrowSampling attack.

A standard approach for fault attacks that target fixed parameters of an algorithm is to use check sums over the values that need protection. In this case they would be applied to the entries of the table containing the values of σ to make sure that no change occurred before the normalization of the FALCON tree.

Another widely used protection against fault attacks is to run the verification right after a signature is generated but before it is released. In certain cases this can prevent the release of faulty signatures and in turn avoid the leakage of sensitive information. Importantly though, NarrowSampling operates in a way that even faulted signatures are still valid. The presence of a fault cannot be observed on any individual signature, since it influences only the overall distribution.

Other, novel methods to protect FALCON against NarrowSampling are left for future work.

5 Conclusion

In this thesis we presented a comprehensive vulnerability analysis of the digital signature scheme Falcon. This included a summary of known attacks as well as an assessment of the sensitivity of most variables and values in Keygen, Sign, Verify and their subroutines. Our literature study showed that many attacks from prior publications focus on the trapdoor sampler at the heart of Falcon Sign. This aligns with previous observations that the sampling of signatures in Falcon is the most sensitive part of the scheme. Furthermore we pointed out multiple, as of yet unexploited vulnerabilities and outlined possible attacks based on them. This is highlighting once again that physical security is essential for the deployment of Falcon specifically and for the adoption of post-quantum cryptographic algorithms in general.

For one vulnerability in particular (NarrowSampling) we demonstrated that a fault injected into Keygen leads to a leakage of secret information through the distribution of signatures. We analyzed the leaked information, derived a connection to the Falcon secret key and demonstrated that full key recovery is practically possible for parameter sets with reduced security. After additional measurements, simulations and tests, we are confident that our results will translate to the full parameter sets without problem and that the NarrowSampling attack can reveal the secret key in those instances as well, given sufficient computing resources.

5.1 Open Questions and Future Work

Further Sensitivity Analysis. Although this thesis provides the first systematic analysis of the physical security of Falcon that we know of, not every variable was conclusively categorized with regards to its sensitivity (compare the list of exclusions from our analysis in Section 3.1 as well as all variables with a *potential* designation). More research needs to be done to further the understanding of possible side-channel and fault attacks, as well as other categories of attacks that we did not consider for this thesis. We strongly believe that there are still more vulnerabilities to be exploited. With a cryptographic scheme as complex as Falcon, it is all but guaranteed that further attacks will be discovered and known attacks will be improved in the future.

The NarrowSampling Attack in Practice. When considering the practicality of our attack in Section 4.4, we could only make reasonable assumptions based on our simulations and extrapolate from the gathered data. The obvious next step is to perform the attack in practice, on actual hardware and for the full security levels $\kappa = 9$ and $\kappa = 10$ and verify our predictions.

Improving the NarrowSampling Attack. Closely related to the aspect of practicality, there are various ways to making NarrowSampling more efficient. One example is the implementation of post-processing. At the moment we use Python to decompress the signatures, where an implementation in C would surely be faster. Then for the HPP solver we use C code via Cython. Here too, the execution time could benefit from a native C version of the solver.

From a theoretical point of view the solution method to the HPP could also be improved. The choice of the target function in the gradient descent phase, $\text{mom}_{\mathbf{C},4}$, was originally guided at least in part by certain uniformity assumptions (see Section 4.2.1 and [NR06]). The signatures that we collect from a faulted key do not conform to these assumptions. So there might be other target functions that are more stable in the presence of noise and therefore more suited to our situation. These could potentially yield better approximations and allow for a relaxation of the attack parameters such as the sigma scaling factor or the number of signatures. Nguyen and Regev [NR06] mention Independent Component Analysis (ICA), which is a class of statistical methods, as a promising direction for future research.

Another approach to mitigating the noise would be to pre-process the signatures before feeding them into the HPP solver. We suspect there are various methods to "sharpen" the parallelepiped, be that through statistical, geometrical or other means.

Countermeasures. One topic that we only briefly touched are countermeasures for NarrowSampling. In general, the goal of investigating attacks is to find ways to thwart them. Since the focus of this thesis lied more on pointing out vulnerabilities, a natural next step should be to protect FALCON against NarrowSampling. This should include an analysis of possible countermeasures with regards to their effectiveness and efficiency.

On a more general note, protection of other vulnerabilities that we found as part of our sensitivity analysis in Section 3 will also be of high importance. Although standard countermeasures for many kinds of attacks exist, it is often necessary to adapt them to specific use cases in order to make them practically viable. This can have different reasons, such as limited computational resources on embedded devices, among others.

Choice of Falcon Parameter σ. The parameter sets for FALCON, and in particular the values that govern signature generation, have been chosen on a theoretical basis that provably guarantees a certain level of security. Our NarrowSampling attack can be viewed not only as a deliberate change of the σ parameter value, in order to deteriorate the security, but also as a method to verify the choice in the first place. In this thesis we only worked with values in the range between $0.1\,\sigma$ up to $0.2\,\sigma$, but higher and lower values are possible. As discussed, the closer the faulted σ is to its original value, the less likely it is for NarrowSampling to recover the secret key. In that sense our attack can provide practical verification for the choice of σ and act as a measure of security when defining possible alternative parameter sets for different security requirements.

References

[Bab86] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Paper 2017/634, 2017. `https://eprint.iacr.org/2017/634`.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. Cryptology ePrint Archive, Paper 2019/1086, 2019. `https://eprint.iacr.org/2019/1086`.

[BS23] Sven Bauer and Fabrizio De Santis. A differential fault attack against deterministic Falcon signatures. *IACR Cryptol. ePrint Arch.*, page 422, 2023.

[DLL+17] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. `https://eprint.iacr.org/2017/633`.

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. Cryptology ePrint Archive, Paper 2014/794, 2014. `https://eprint.iacr.org/2014/794`.

[DP15] Léo Ducas and Thomas Prest. Fast Fourier orthogonalization. Cryptology ePrint Archive, Paper 2015/1014, 2015. `https://eprint.iacr.org/2015/1014`.

[DSDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. Cryptology ePrint Archive, Paper 2020/292, 2020. `https://eprint.iacr.org/2020/292`.

[FHK+20] 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 - specifications v1.2 - 01/10/2020. `https://falcon-sign.info/falcon.pdf`, 2020.

[FKT+20] Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Key recovery from Gram-Schmidt norm leakage in hash-and-sign signatures over NTRU lattices. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 34–63. Springer, 2020.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on Falcon. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):141–164, 2022.

[GPV07] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. Cryptology ePrint Archive, Paper 2007/432, 2007. https://eprint.iacr.org/2007/432.

[HPRR19] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous Gaussian sampling: From inception to implementation. Cryptology ePrint Archive, Paper 2019/1411, 2019.

[KA21] Emre Karabulut and Aydin Aysu. FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 691–696. IEEE, 2021.

[KH18] Suhri Kim and Seokhie Hong. Single trace analysis on constant time CDT sampler and its countermeasure. *Applied Sciences*, 8(10):1809, 2018.

[Kle00] Philip N. Klein. Finding the closest lattice vector when it's unusually close. In *ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[MHS+19] 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 Mohammad S. Obaidat and Pierangela Samarati, editors, *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, pages 61–71. SciTePress, 2019.

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`.

[NR06] Phong Q Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 271–288. Springer, 2006.

[PP19] Thomas Pornin and Thomas Prest. More efficient algorithms for the NTRU key generation using the field norm. Cryptology ePrint Archive, Paper 2019/015, 2019. `https://eprint.iacr.org/2019/015`.

[PRR19] Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Simple, fast and constant-time Gaussian sampling over the integers for Falcon. 2019.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[SS13] Damien Stehlé and Ron Steinfeld. Making NTRUEncrypt and NTRUSign as secure as standard worst-case problems over ideal lattices. Cryptology ePrint Archive, Paper 2013/004, 2013. `https://eprint.iacr.org/2013/004`.

[ZLYW23] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on Falcon. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 565–595. Springer, 2023.

List of Tables

List of Algorithms

List of Figures