Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU

Specification v1.2 — 01/10/2020

Pierre-Alain Fouque Jeffrey Hoffstein Paul Kirchner
Vadim Lyubashevsky Thomas Pornin Thomas Prest Thomas Ricosset
Gregor Seiler William Whyte Zhenfei Zhang

falcon@ens.fr

Contents

Chapter 1

Introduction

FALCON is a lattice-based signature scheme. It stands for the following acronym:

Fast Fourier lattice-based compact signatures over NTRU

The high-level design of FALCON is simple: we instantiate the theoretical framework described by Gentry, Peikert and Vaikuntanathan [GPV08] for constructing hash-and-sign lattice-based signature schemes. This framework requires two ingredients:

- A class of cryptographic lattices. We chose the class of NTRU lattices.
- A trapdoor sampler. We rely on a new technique which we call fast Fourier sampling.

In a nutshell, the FALCON signature scheme may therefore be described as follows:

FALCON = GPV framework + NTRU lattices + Fast Fourier sampling

This document is the supporting documentation of FALCON. It is organized as follows. Chapter 2 explains the overall design of FALCON and its rationale. Chapter 3 is a complete specification of FALCON. Chapter 4 discusses implementation issues and possible optimizations, and described measured performance.

1.1 Genealogy of Falcon

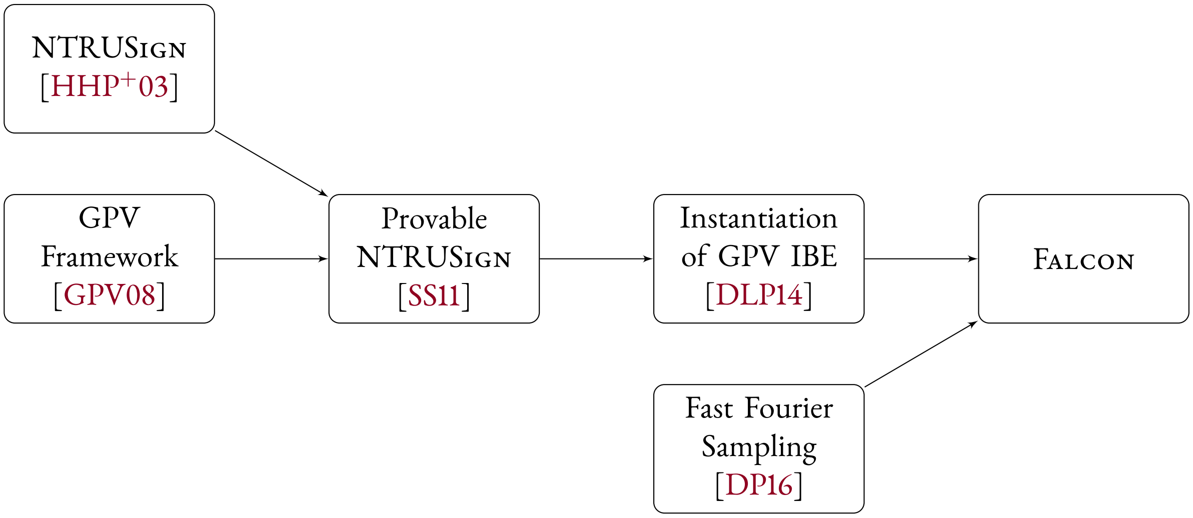

Figure 1.1: The genealogic tree of Falcon

Falcon is the product of many years of work, not only by the authors but also by others. This section explains how these works gradually led to Falcon as we know it.

The first work is the signature scheme NTRUSign [HHP+03] by Hoffstein *et al.*, which was the first, along with GGH [GGH97], to propose lattice-based signatures. The use of NTRU lattices by NTRUSign allows it to be very compact. However, both had a flaw in the deterministic signing procedure which led to devastating key-recovery attacks [NR06, DN12].

At STOC 2008, Gentry, Peikert and Vaikuntanathan [GPV08] proposed a method which not only corrected the flawed signing procedure but, even better, did it in a provably secure way. The result was a generic framework (the GPV framework) for building secure hash-and-sign lattice-based signature schemes.

The next step towards Falcon was the work of Stehlé and Steinfeld [SS11], who combined the GPV framework with NTRU lattices. The result could be called a provably secure NTRUSign.

In a more practical work, Ducas *et al.* [DLP14] proposed a practical instantiation and implementation of the IBE part of the GPV framework over NTRU lattices. This IBE can be converted in a straightforward manner into a signature scheme. However, doing this would have resulted in a signing time in $O(n^2)$.

To address the issue of a slow signing time, Ducas and Prest [DP16] proposed a new algorithm running in time $O(n \log n)$. However, how to practically instantiate this algorithm remained a open question.

Falcon builds on these works to propose a practical lattice-based hash-and-sign scheme. The fig. 1.1 shows the genealogic tree of Falcon, the first of the many trees that this document contains.

1.2 Subsequent Related Work

This section presents a non-exhaustive list of work related to Falcon, and subsequent to the Round 1 version (1.0) of the specification.

Isochronous Gaussian sampling. Realising efficient isochronous Gaussian sampling over the integers has long been identified as an important problem. Recent works by Zhao *et al.* [ZSS20], Karmakar *et al.* [KRVV19] and Howe *et al.* [HPRR20], have proposed new techniques. The sampler in the Round 3 version of Falcon relies on [ZSS20, HPRR20]. Recent work by Fouque *et al.* [FKT$^+$20] shows that isochrony is indeed an important requirement for the embedded security of Falcon.

Raptor: Ring signatures using Falcon. Lu, Au and Zhang [LAZ18] have proposed Raptor, a ring signature scheme which uses Falcon as a building block. The authors provided a security proof in the random oracle model, as well as an efficient implementation.

Implementation on ARM Cortex. Works by Oder *et al.* [OSHG19] and Pornin [Por19] have implemented Falcon on ARM Cortex-M microprocessors. See also pqm4 [KRSS19].

Key generation. Pornin and Prest [PP19] have formally studied the part of the key generation where polynomials F, G are computed from f, g. This paper can be used as a complement for readers willing to understand more thoroughly this part of the key generation.

Deployment in TLS 1.3. Sikeridis *et al.* [SKD20] studied the performance of various NIST candidate signature schemes in TLS 1.3. Falcon and Dilithium were the most favorably rated schemes.

1.3 NIST Requirements

In this section, we provide a mapping of the requirements by NIST to the appropriate sections of this document. This document adresses the requirements in [NIS16, Section 2.B].

- The complete specification as per [NIS16, Section 2.B.1] can be found in Chapter 3. A design rationale can be found in Chapter 2.
- A performance analysis, as per [NIS16, Section 2.B.2], is provided in Chapter 4.
- The security analysis of the scheme as per [NIS16, Section 2.B.4], and the analysis of known cryptographic attacks against the scheme as per [NIS16, Section 2.B.5], are contained in Section 2.5.
- Advantages and limitations as per [NIS16, Section 2.B.6] are listed in Section 2.7.
- Two sets of parameters as per NIST [NIS16, Section 4.A.5] can be found in Section 3.13.

Other requirements in [NIS16] are not addressed in this document, but in other parts of the submission package.

- A reference implementation as per [NIS16, Section 2.C.1] and Known Answer Test values as per [NIS16, Section 2.B.2] are present in this submission package.

1.4 Changelog

This is the version 1.2 of Falcon's specification. The differences with the version 1.0 [PFH+17] are:

- We removed the level II-III set of parameters, which entailed $n = 768$ and $\phi = x^n - x^{n/2} + 1$; interested readers and implementers can read the version 1.0 of the specification, in which this set of parameters remains for historical purposes.
- We added a section about the related work (Section 1.2);
- We now describe a key-recovery mode which makes Falcon even more compact (Section 3.12);
- We did a few other minor additions which essentially consist of clarifying and detailing a few points.

The differences with the version 1.1 [PFH+19] are:

- We propose a formal specification of the Gaussian sampler over the integers, see Section 3.9.3. This specification consists of four algorithms (algorithms 12 to 15). In addition, Table 3.2 and `Supporting_Documentation/additional/test-vector-sampler-falcon{512,1024}.txt` provide test vectors to validate the implementation of SamplerZ.
- We tweak Compress (algorithm 17) and Decompress (algorithm 18) in order to enforce a unique encoding of signatures. We are thankful to Quan Nguyen for pointing out to us the (benign) malleability of the original encodings.
- We provide updated parameters, see Table 3.3. The parameter sets are more detailed and, in the case of Falcon-512, now provide a few more bits of security. In addition, we now detail our parameter selection process in Section 2.6 and `Supporting_Documentation/additional/parameters.py`. We discuss the concrete security of our parameter sets in Section 2.5.1.
- We make incremental changes to some algorithms. Most reflect optimizations that the reference code was already doing (e.g. loop unrolling). Others are introduced by SamplerZ and our modified Compress/Decompress. Finally, we correct some typos (marked with † below).
  - NTRUGen: lines 3, 7, 9, 13 and 14;
  - NTRUSolve: lines 4, 11 and 12†;
  - LDL*;
  - ffLDL*: line 10;
  - Sign: lines 3, 4, 8 and 11;
  - ffSampling: lines 3 and 4;
  - Compress: lines 4†, 5, 6, 7, 8, 9 and 10;
  - Decompress: lines 1, 2, 4†, 6†, 9, 10, 12 and 13;
  - Verify: lines 3, 4 and 6.

Chapter 2

The Design Rationale of Falcon

2.1 A Quest for Compactness

The design rationale of Falcon stems from a simple observation: when switching from RSA- or discrete logarithm-based signatures to post-quantum signatures, communication complexity will likely be a larger problem than speed. Indeed, many post-quantum schemes have a simple algebraic description which makes them fast, but all require either larger keys than pre-quantum schemes, larger signatures, or both.

We expect such performance issues will hinder transition from pre-quantum to post-quantum schemes. Hence our leading design principle was to minimize the following quantity:

$$|\mathsf{pk}| + |\mathsf{sig}| = (\text{bitsize of the public key}) + (\text{bitsize of a signature}).$$

This led us to consider lattice-based signatures, which manage to keep both $|\mathsf{pk}|$ and $|\mathsf{sig}|$ rather small, especially for structured lattices. When it comes to lattice-based signatures, there are essentially two paradigms: Fiat-Shamir or hash-and-sign.

Both paradigms achieve comparable levels of compactness, but hash-and-sign have interesting properties: the GPV framework [GPV08], which describes how to obtain hash-and-sign lattice-based signature schemes, is secure in the classical and quantum oracle models [GPV08, BDF$^+$11]. In addition, it enjoys message-recovery capabilities [dLP16]. So we chose this framework. Details are given in Section 2.2.

Next, we chose a class of cryptographic lattices to instantiate this framework. A close to optimal choice with respect to our main design principle – compactness – is NTRU lattices: they allow to obtain a compact instantiation [DLP14] of the GPV framework. In addition, their structure speeds up many operations by two orders of magnitude. Details are given in Section 2.3.

The last step was the trapdoor sampler. We devised a new trapdoor sampler which is asymptotically as fast as the fastest generic trapdoor sampler [Pei10] and provides the same level of security as the most secure sampler [Kle00]. Details are given in Section 2.4.

2.2 The Gentry-Peikert-Vaikuntanathan Framework

In 2008, Gentry, Peikert and Vaikuntanathan [GPV08] established a framework for obtaining secure lattice-based signatures. At a very high level, this framework may be described as follows:

- The public key contains a full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ (with $m > n$) generating a q-ary lattice Λ.
- The private key contains a matrix $\mathbf{B} \in \mathbb{Z}_q^{m\times m}$ generating $\Lambda_q^\perp$, where $\Lambda_q^\perp$ denotes the lattice orthogonal to Λ modulo q: for any $\mathbf{x} \in \Lambda$ and $\mathbf{y} \in \Lambda_q^\perp$, we have $\langle \mathbf{x}, \mathbf{y} \rangle = 0 \bmod q$. Equivalently, the rows of $\mathbf{A}$ and $\mathbf{B}$ are pairwise orthogonal: $\mathbf{B} \times \mathbf{A}^{\mathsf{t}} = \mathbf{0}$.
- Given a message m, a signature of m is a short value $\mathbf{s} \in \mathbb{Z}_q^m$ such that $\mathbf{s}\mathbf{A}^{\mathsf{t}} = H(\mathsf{m})$, where $H : \{0,1\}^* \to \mathbb{Z}_q^n$ is a hash function. Given $\mathbf{A}$, verifying that $\mathbf{s}$ is a valid signature is straightforward: it only requires to check that $\mathbf{s}$ is indeed short and verifies $\mathbf{s}\mathbf{A}^{\mathsf{t}} = H(\mathsf{m})$.
- Computing a valid signature is more delicate. First, a preimage $\mathbf{c}_0 \in \mathbb{Z}_q^m$ is computed, which verifies $\mathbf{c}_0\mathbf{A}^{\mathsf{t}} = H(\mathsf{m})$. As $\mathbf{c}_0$ is not required to be short and $m \geq n$, this is simply done via standard linear algebra. $\mathbf{B}$ is then used in order to compute a vector $\mathbf{v} \in \Lambda_q^\perp$ close to $\mathbf{c}_0$. The difference $\mathbf{s} = \mathbf{c}_0 - \mathbf{v}$ is a valid signature: indeed, $\mathbf{s}\mathbf{A}^{\mathsf{t}} = \mathbf{c}_0\mathbf{A}^{\mathsf{t}} - \mathbf{v}\mathbf{A}^{\mathsf{t}} = \mathbf{c} - \mathbf{0} = H(\mathsf{m})$, and if $\mathbf{c}_0$ and $\mathbf{v}$ are close enough, then $\mathbf{s}$ is short.

This high-level description of a signature scheme is not exclusive to the GPV framework: it was first instantiated in the GGH [GGH97] and NTRUSign [HHP+03] signature schemes. However, these schemes suffered total break attacks, whereas the GPV framework is proven secure in the (quantum) random oracle model assuming the hardness of SIS for some parameters. This is because GGH/NTRUSign and the GPV framework have radically different ways of computing $\mathbf{v}$ in the signing procedure.

Computing v in GGH and NTRUSign. In GGH and NTRUSign, $\mathbf{v}$ is computed using an algorithm called the round-off algorithm and first formalized by Babai [Bab85, Bab86]. In this deterministic algorithm, $\mathbf{c}_0$ is first expressed as a real linear combination of the rows of $\mathbf{B}$, the vector of these real coordinates is then rounded coefficient-wise and multiplied again by $\mathbf{B}$: in a nutshell, $\mathbf{v} \leftarrow \lfloor \mathbf{c}_0\mathbf{B}^{-1} \rceil \mathbf{B}$, where $\lfloor \cdot \rceil$ denotes coefficient-wise rounding. At the end of the procedure, $\mathbf{s} = \mathbf{v} - \mathbf{c}_0$ is guaranteed to lie in the parallelepiped $[-1,1]^m \times \mathbf{B}$, which allows to tightly bound the norm $\|\mathbf{s}\|$. The problem with this approach is that each signature $\mathbf{s}$ lies in $[-1,1]^m \times \mathbf{B}$, and therefore leaks information about the basis $\mathbf{B}$. This fact was exploited by several key-recovery attacks [NR06, DN12].

Computing v in the GPV framework. A major contribution of [GPV08], which is also the key difference between the GPV framework and GGH/NTRUSign, is the way $\mathbf{v}$ is computed. Instead of the round-off algorithm, the GPV framework relies on a randomized variant by [Kle00] of the nearest plane algorithm, also formalized by Babai. Just as for the round-off algorithm, using the nearest plane algorithm would have leaked the secret basis $\mathbf{B}$ and resulted in a total break of the scheme. However, Klein's algorithm prevents this: it is randomized in a way such that for a given m, $\mathbf{s}$ is sampled according to a spherical Gaussian distribution over the shifted lattice $\mathbf{c}_0 + \Lambda_q^\perp$. This method is proven to leak no information about the basis $\mathbf{B}$. Klein's algorithm was in fact the first of a family of algorithms called *trapdoor samplers*. More details about trapdoor samplers are given in Section 2.4.

2.2.1 Features and instantiation of the GPV framework

Security in the classical and quantum oracle models. In the original paper [GPV08], the GPV framework has been proven to be secure in the random oracle model under the SIS assumption. In our case, we use NTRU lattices so we need to adapt the proof for a "NTRU-SIS" assumption, but this adaptation is straightforward. In addition, the GPV framework has also been proven to be secure in the quantum oracle model [BDF+11].

Identity-based encryption. Falcon can be turned into an identity-based encryption scheme, as described in [DLP14]. However, this requires de-randomizing the signature procedure (see Section 2.2.2).

2.2.2 Statefulness, de-randomization or hash randomization

In the GPV framework, two different signatures s, s' of a same hash $H(\mathsf{m})$ can never be made public simultaneously, because doing so breaks the security proof [GPV08, Section 6.1].

Statefulness. A first solution proposed in [GPV08, Section 6.1] is to make the scheme stateful by maintaining a list of the signed messages and of their signatures. However, maintaining such a state poses a number of operational issues, so we do not consider it as a credible solution.

De-randomization. A second possibility proposed by [GPV08] is to de-randomize the signing procedure. However, pseudorandomness would need to be generated in a consistent way over all the implementations (it is not uncommon to have a same signing key used in different devices). While this solution can be applied in a few specific usecases, we do not consider it for Falcon.

Hash randomization. A third solution is to prepend a salt $\mathsf{r} \in \{0,1\}^k$ to the message m before hashing it. Provided that k is large enough, this prevents collisions from occurring. From an operational perspective, this solution is the easiest to apply, and it is still covered by the security proof of the GPV framework (see [GPV08, Section 6.2]). For a given security level λ and up to q_s signature queries, taking $k = \lambda + \log_2(q_s)$ is enough to guarantee that the probability of collision is less than $q_s \cdot 2^{-\lambda}$.

Out of the three solutions, Falcon opts for hash randomization: a salt $\mathsf{r} \in \{0,1\}^{320}$ is randomly generated and prepended to the message before hashing it. The bitsize 320 is equal to $\lambda + \log_2(q_s)$ for $\lambda = 256$ the highest security level required by NIST, and $q_s = 2^{64}$ the maximal number of signature which may be queried from a single signer. This size is actually overkill for security levels $\lambda < 256$, but fixing a single size across all the security levels makes things easier from an API perspective: for example, one can hash a message without knowing the security level of the private signing key.

2.3 NTRU Lattices

The first choice when instantiating the GPV framework is the class of lattices to use. The design rationale obviously plays a large part in this. Indeed, if emphasis is placed on security without compromise, then the logical choice is to use standard lattices without any additional structure, as was done e.g. in the key-exchange scheme Frodo [BCD+16].

Our main design principle is compactness. For this reason, Falcon relies on the class of NTRU lattices, introduced by Hoffstein, Pipher and Silverman [HPS98]; they come with an additional ring structure which not only does allow to reduce the public keys' size by a factor $O(n)$, but also speeds up many computations by a factor at least $O(n/\log n)$. Even in the broader class of lattices over rings, NTRU lattices are among the most compact: the public key can be reduced to a single polynomial $h \in \mathbb{Z}_q[x]$ of degree at most $n-1$. In doing this we follow the idea of Stehlé and Steinfeld [SS11], who showed that the GPV framework can be used with NTRU lattices in a provably secure way.

Compactness, however, would be useless without security. From this perspective, NTRU lattices also have reasons to inspire confidence as they have resisted extensive cryptanalysis for about two decades, and we parameterize them in a way which we believe makes them even more resistant.

2.3.1 Introduction to NTRU lattices

Let $\phi = x^n + 1$ for $n = 2^\kappa$ a power of two, and $q \in \mathbb{N}^\star$. A set of NTRU secrets consists of four polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ which verify the NTRU equation:

$$fG - gF = q \mod \phi \tag{2.1}$$

Provided that f is invertible modulo q, we can define the polynomial $h \leftarrow g \cdot f^{-1} \bmod q$.

Typically, h will be a public key, whereas f, g, F, G will be secret keys. Indeed, one can check that the matrices $\left[\begin{array}{c|c} 1 & h \\ \hline 0 & q \end{array}\right]$ and $\left[\begin{array}{c|c} f & g \\ \hline F & G \end{array}\right]$ generate the same lattice, but the first matrix contains two large polynomials (h and q), whereas the second matrix contains only small polynomials, which allows to solve problems as illustrated in Section 2.2. If f, g are generated with enough entropy, then h will look pseudo-random [SS11]. However in practice, even when f, g are quite small, it remains hard to find small polynomials f', g' such that $h = g' \cdot (f')^{-1} \bmod q$. The hardness of this problem constitutes the NTRU assumption.

2.3.2 Instantiation with the GPV framework

We now instantiate the GPV framework described in Section 2.2 over NTRU lattices:

- The public basis is $\mathbf{A} = \left[\begin{array}{c|c} 1 & h^\star \end{array}\right]$, but this is equivalent to knowing h.
- The secret basis is

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right] \tag{2.2}$$

One can check that the matrices $\mathbf{A}$ and $\mathbf{B}$ are indeed orthogonal: $\mathbf{B} \times \mathbf{A}^\star = 0 \bmod q$.

- The signature of a message m consists of a salt r plus a pair of polynomials (s_1, s_2) such that $s_1 + s_2 h = H(\mathsf{r}\|\mathsf{m})$. We note that since s_1 is completely determined by m, r and s_2, there is no need to send it: the signature can simply be (r, s_2).

2.3.3 Choosing optimal parameters

Our trapdoor sampler samples signatures of norm essentially proportional to $\|\mathbf{B}\|_{\mathrm{GS}}$, where $\|\mathbf{B}\|_{\mathrm{GS}}$ denotes the Gram-Schmidt norm of $\mathbf{B}$.

Previous works ([DLP14] and [Pre15, Sections 6.4.1 and 6.5.1]) have provided heuristic and experimental evidence that in practice, $\|\mathbf{B}\|_{\mathrm{GS}}$ is minimized for $\|(f, g)\| \approx 1.17\sqrt{q}$. Therefore, we generate f, g as discrete Gaussians in $\mathbb{Z}[x]/(\phi)$ centered in 0, so that the expected value of $\|(f, g)\|$ is about $1.17\sqrt{q}$. Once this is done, very efficient ways to compute $\|\mathbf{B}\|_{\mathrm{GS}}$ are known, and if this value is more than $1.17\sqrt{q}$, new polynomials f, g's are regenerated and the procedure starts over.

Quasi-optimality. The bound $\|\mathbf{B}\|_{\mathrm{GS}} \leq 1.17\sqrt{q}$ that we reach in practice is within a factor 1.17 of the theoretic lower bound for $\|\mathbf{B}\|_{\mathrm{GS}}$. Indeed, for any $\mathbf{B}$ of the form given in (2.2) with f, g, F, G verifying (2.1), we have $\det(\mathbf{B}) = fG - gF = q$. So $\sqrt{q}$ is a theoretic lower bound of $\|\mathbf{B}\|_{\mathrm{GS}}$.

2.4 Fast Fourier Sampling

The second choice when instantiating the GPV framework is the trapdoor sampler. A trapdoor sampler takes as input a matrix $\mathbf{A}$, a trapdoor T, a target $\mathbf{c}$ and outputs a short vector $\mathbf{s}$ such that $\mathbf{s}^t\mathbf{A} = \mathbf{c} \bmod q$. With the notations of Section 2.2, this is equivalent to finding $\mathbf{v} \in \Lambda_q^\perp$ close to $\mathbf{c}_0$, so we may indifferently refer by the term "trapdoor samplers" to algorithms which perform one task or the other.

We now list the existing trapdoor samplers, their advantages and limitations. Obviously, being efficient is important for a trapdoor sampler. However, an equally important metric is the "quality" of the sampler: the shorter the vector $\mathbf{s}$ is (or equivalently, the closer $\mathbf{v}$ is to $\mathbf{c}_0$), the more secure this sampler will be.

1. Klein's algorithm [Kle00] takes as a trapdoor the matrix $\mathbf{B}$. It outputs vectors $\mathbf{s}$ of norm proportional to $\|\mathbf{B}\|_{\mathrm{GS}}$, which is short and therefore good for security. On the downside, its time and space complexity are in $O(m^2)$.

2. Just like Klein's algorithm is a randomized version of the nearest plane algorithm, Peikert proposed a randomized version of the round-off algorithm [Pei10]. A nice thing about it is that when $\mathbf{B}$ has a structure over rings – as in our case – then it can be made to run in time and space $O(m \log m)$. However, it outputs vectors of norm proportional to the spectral norm $\|\mathbf{B}\|_2$ of $\mathbf{B}$. This is larger than what we get with Klein's algorithm, and therefore it is worse security-wise.

3. Micciancio and Peikert [MP12] proposed a novel approach in which $\mathbf{A}$ and its trapdoor are constructed in a way which allows simple and efficient trapdoor sampling. Unfortunately, it is not

straightforwardly compatible with NTRU lattices and yet has to reach the same level of compactness as with NTRU lattices [CGM19].

4. Ducas and Prest [DP16] proposed "*fast Fourier nearest plane*", a variant of Babai's nearest plane algorithm for lattices over rings. It proceeds in a recursive way which is very similar to the fast Fourier transform, hence the name. This algorithm can be randomized: it results in a trapdoor sampler which combines the quality of Klein's algorithm, the efficiency of Peikert's and can be used over NTRU lattices.

Of the four approaches we just described, it seems clear to us that a randomized variant of the fast Fourier nearest plane [DP16] is the most adequate choice given our design rationale and our previous design choices (NTRU lattices). For this reason, it is the trapdoor sampler used in Falcon.

Sampler	**Fast**	**Short output s**	**NTRU-friendly**
Klein [Kle00]	No	Yes	Yes
Peikert [Pei10]	Yes	No	Yes
Micciancio-Peikert [MP12]	Yes	Yes	No
Ducas-Prest [DP16]	Yes	Yes	Yes

Table 2.1: Comparison of the different trapdoor samplers

Choosing the standard deviation. When using a trapdoor sampler, an important parameter to set is the standard deviation σ. If it is too low, then it is no longer guaranteed that the sampler not leak the secret basis (and indeed, for all known samplers, a value $\sigma = 0$ opens the door to learning attacks à la [NR06, DN12]). But if it is too high, the sampler does not return optimally short vectors and the scheme is not as secure as it could be. So there is a compromise to be found. Our fast Fourier sampler shares many similarities with Klein's sampler, including the optimal value for σ. Following [Pre17, Section 4.4], we take $\sigma = \eta_\epsilon(\mathbb{Z}^{2n}) \cdot \|\mathbf{B}\|_{\mathrm{GS}}$.

2.5 Security

2.5.1 Known Attacks

Key Recovery. The most efficient attacks come from lattice reduction. We start by considering the lattice generated by the columns of $\left[\begin{array}{c|c} q & h \\ \hline 0 & 1 \end{array}\right]$. After using lattice reduction on this basis, we enumerate all lattice points in a ball of radius $\sqrt{2n} \cdot \sigma_{\{f,g\}}$, centered on the origin. With significant probability, we are therefore able to find $\left[\begin{array}{c|c} g & f \end{array}\right]$.

Let λ be the $(2n - B)$th Gram-Schmidt norm, which is approximately the norm of the shortest vector of the lattice generated by the last B vectors projected orthogonally to the first $2n - B - 1$ vectors. A sieve algorithm performed on this projected lattice will recover all vectors of norm smaller than $\sqrt{4/3}\lambda$

(see [Duc18] for instance). If the projection of the key is among them, that is when

$$\sqrt{B}\sigma_{\{f,g\}} \leq \sqrt{4/3}\lambda,$$

we can recover a secret key vector from its projection by using Babai's Nearest Plane algorithm on all sieved vectors with high probability. This is because all remaining Gram-Schmidt norms are larger than λ, which is much larger than $\sigma_{\{f,g\}}$.

For the best known lattice reduction algorithm, DBKZ [MW16, Corollary 2], we get

$$\lambda = \left(\frac{B}{2\pi e}\right)^{1-n/B}\sqrt{q},$$

and

$$(B/2\pi e)^{1-n/B}\sqrt{q} = \sqrt{3/4B}\sigma_{\{f,g\}} \tag{2.3}$$

Note that we conservatively assumed that we could perform a sieve algorithm in dimension B for the same cost as the SVP oracle inside the DBKZ algorithm, which is a slight overestimate [Duc18]. It is then easy to deduce B. Note that the given value for the Gram-Schmidt norm is correct only when the basis is first randomized, and it is necessary to do so (asymptotically).

Forging a Signature. Forging a signature can be performed by finding a lattice point at distance bounded by β from a random point, in the same lattice as above. This task can also be solved by lattice reduction. One possibility is to use Kannan's embedding, that is add $(H(r||m), 0, K)$ to the lattice basis, extended by a row of zeroes, which gives the following matrix:

$$\left[\begin{array}{c|c|c} q & h & H(r\|m) \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & K \end{array}\right].$$

As sieve algorithms generate many short vectors, we can certainly find among them a vector of the form $(c, *, K)$ and then $H(r||m) - c$ is a lattice point.

Taking $K \approx \sqrt{q}$, the DBKZ algorithm [MW16, Corollary 2] gives as a success condition for the forgery:

$$\left(\frac{B}{2\pi e}\right)^{n/B}\sqrt{q} \leq \beta. \tag{2.4}$$

Interestingly, since the factor $\sqrt{q}$ is also present in β, the modulus q has virtually no effect on the best forgery attack. This is the best attack against our instantiations. We convert the blocksize B into concrete bit-security following the methodology of New Hope [ADPS16], sometimes called "core-SVP methodology". This gives the bit-security as per [BDGL16, Laa16]:

$$\text{Classical: } \lfloor 0.292 \cdot B \rfloor \tag{2.5}$$

$$\text{Quantum:} \lfloor 0.262 \cdot B \rfloor \tag{2.6}$$

This gives the following table.

	Key recovery				Forgery			
n	B	B'	Classical	Quantum	B	B'	Classical	Quantum
512	458	418	133	121	411	374	120	108
1024	936	869	273	248	952	884	277	252

Concrete cost of the best attacks. For Falcon-512, we estimate the complexity of the best attack as equivalent to a BKZ with block size $B = 411$. The latest method [ADH+19] suggests that the cost in dimension n is close to solving $\frac{n^3}{4B^2}$ shortest vector problem instances in dimension B. The optimization of Ducas [Duc18] decreases the dimension of the lattice sieved by $\left\lfloor \frac{B\ln(4/3)}{\ln(B/(2\pi e))} \right\rceil = 37$ to $B' = 374$.

Taking only the first asymptotical term in the complexity of a sieve [BDGL16] leads to a number of $\frac{n^3}{4B^2} \cdot (\sqrt{1.5})^{B'} \approx 2^{120}$ classical operations (where $\sqrt{1.5} \approx 2^{0.292}$). This is believed to be a conservative estimate, as we neglect the lower order subexponential terms in the Nearest Neighbor Search. Each operation includes a random access of at least one bit to a memory which has to contain 2^{77} vectors.

A recent record [ADH+19] used $2^{19}(\sqrt{1.5})^{112}$ cycles for a sieve in dimension 112, and an average cycle certainly used more than 16 gates. We therefore regard an estimate of the minimum number of gates of $2^{120+19+4} = 2^{143}$ as conservative.

For Falcon-1024, key recovery is slightly more efficient. The first part of the attack uses lattice reduction, and cost more than 2^{10} calls to a SVP instance in dimension $B = 936$, which corresponds to a sieve in dimension $B' = 869$. This indicates a total of at least classical 2^{264} operations; and a number of gates larger than 2^{287}.

For the quantum cost, we take [JNRV20, Table 10] as a baseline. For key search on AES-{128,256}, it indicates a cost of $\{2^{82}, 2^{143}\}$ gates. This is far below the estimated quantum cost for breaking Falcon.

Hybrid attack. The hybrid attack [How07] combines a meet-in-the-middle algorithm and the key recovery algorithm. It was used with great effect against NTRU, due to its choice of *sparse* polynomials. This is however not the case here, so that its impact is much more modest, and counterbalanced by the lack of sieve-enumeration.

Dense, high rank sublattice. Recent works [ABD16, CJL16, KF17] have shown that when f, g are extremely small compared to q, it is easy to attack cryptographic schemes based on NTRU lattices. To the contrary, in Falcon we take f, g to be not too small while q is hardly large: a side-effect is that this makes our scheme impervious to the so-called "overstretched NTRU" attacks. In particular, even if f, g were taken to be binary, we would have to select $q > n^{2.83}$ for this property to be useful for cryptanalysis. Our large margin should allow even significant improvements of this algorithm to be irrelevant to our case.

Algebraic attacks. While there is a rich algebraic structure in Falcon, there is no known way to improve all the algorithms previously mentioned with respect to their general lattice equivalent by more than a factor n^2. However, there exist efficient algorithms for finding not-so-small elements in *ideals* of $\mathbb{Z}[x]/(\phi)$ [CDW17].

2.5.2 Precision of the Floating-Point Arithmetic

Trapdoor samplers usually require the use of floating-point arithmetic, and our fast Fourier sampler is no exception. This naturally raises the question of the precision required to claim meaningful security bounds. A naive analysis would require a precision of $O(\lambda)$ bits (notwithstanding logarithmic factors), but this would result in a substantially slower signature generation procedure.

In order to analyze the required precision, we use a Rényi divergence argument. As in [MW17], we denote by $a \lesssim b$ the fact that $a \leq b + o(b)$, which allows discarding negligible factors in a rigorous way. Our fast Fourier sampler is a recursive algorithm which relies on $2n$ discrete samplers $D_{\mathbb{Z},c_j,\sigma_j}$. We suppose that the values c_j (resp. σ_j) are known with an *absolute* error (resp. *relative* error) at most δ_c (resp. δ_σ) and denote by $\mathcal{D}$ (resp. $\bar{\mathcal{D}}$) the output distribution of our sampler with infinite (resp. finite) precision. We can then re-use the precision analysis of Klein's sampler in [Pre17, Section 4.5]. For any output of our sampler with non-negligible probability, in the worst case:

$$\left|\log\left(\frac{\bar{\mathcal{D}}(\mathbf{z})}{\mathcal{D}(\mathbf{z})}\right)\right| \lesssim 2n\left[\frac{\sqrt{154}}{1.312}\delta_c + (2\pi+1)\delta_\sigma\right] \leq 20n(\delta_c+\delta_\sigma) \tag{2.7}$$

In the average case, the value $2n$ in (2.7) can be replaced with $\sqrt{2n}$. Following the security arguments of [Pre17, Section 3.3], this allows to claim that in average, no security loss is expected if $(\delta_c + \delta_\sigma) \leq 2^{-46}$.

To check if this is the case for Falcon, we have run Falcon in two different precisions, a high precision of 200 bits and a standard precision of 53 bits, and compared the values of the c_j, σ_j's. The result of these experiments is that we always have $(\delta_c + \delta_\sigma) \leq 2^{-40}$: while this is higher than 2^{-46}, the difference is of only 6 bits. Therefore, we consider that 53 bits of precision are sufficient for NIST's parameters (security level $\lambda \leq 256$, number of queries $q_s \leq 2^{64}$), and that the possibility of our signature procedure leaking information about the secret basis is a purely theoretic threat.

2.6 Summary of Parameters

In this section, we summarize the interplay between parameters. The resulting parameter selection process is automatized in `Supporting_Documentation/additional/parameters.py`, which also gives the core-SVP hardness of key recovery and forgery.

Number of queries Q_s, targeted security level λ and ring degree n. We start with three initial parameters: the maximal number of signing queries Q_s, the targeted security level λ and the degree n of the ring $\mathbb{Z}[x]/(x^n+1)$. As per [NIS16], $Q_s = 2^{64}$. Also as per [NIS16], it suffices to take $\lambda = 128$ for NIST Level I and $\lambda = 256$ for NIST Level V. Finally, we take:

$$n = 512 \qquad \text{for NIST Level I,} \tag{2.8}$$

$$n = 1024 \qquad \text{for NIST Level V.} \tag{2.9}$$

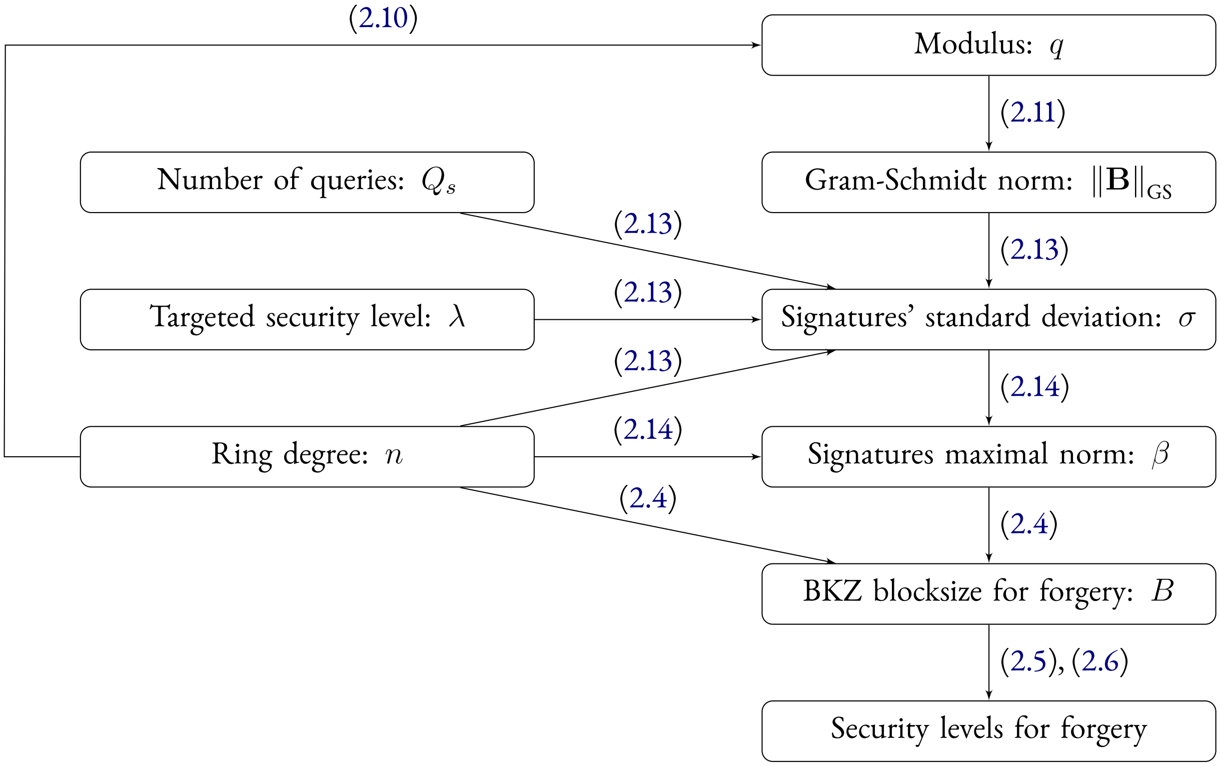
(2.10)
Modulus: q
(2.11)
Number of queries: Q_s
Gram-Schmidt norm: $\|\mathbf{B}\|_{\mathrm{GS}}$
(2.13)
(2.13)
(2.13)
Targeted security level: λ
Signatures' standard deviation: σ
(2.13)
(2.14)
(2.14)
Ring degree: n
Signatures maximal norm: β
(2.4)
(2.4)
BKZ blocksize for forgery: B
(2.5), (2.6)
Security levels for forgery

Figure 2.1: Parameters of Falcon and security estimates. Initial parameters are on the left side of the figure. Parameters on the right side of the figure (which include concrete security estimates) are derived systematically from initial parameters.

Integer modulus q**.** The modulus q needs to be a prime of the form $k \cdot 2n + 1$ in order to maximize the efficiency of the NTT. The smallest prime of this form is

$$q = 12 \cdot 1024 + 1 = 12289. \tag{2.10}$$

For this value, q has essentially no influence on security: it is large enough to resist hybrid attacks and trivial attacks on SIS, and small enough to resist overstetched NTRU attacks.

Gram-Schmidt norm $\|\mathbf{B}\|_{\mathrm{GS}}$**.** We wish to minimize $\|\mathbf{B}\|_{\mathrm{GS}}$. It has been shown in [DLP14, Section 3] that in practice we can ensure (upon resampling a finite number of times) that:

$$\|\mathbf{B}\|_{\mathrm{GS}} \leq 1.17\sqrt{q}. \tag{2.11}$$

In order to do that, each coefficient of f and g is sampled from the discrete Gaussian $D_{\mathbb{Z},\sigma_{\{f,g\}}}$ with:

$$\sigma_{\{f,g\}} = 1.17\sqrt{q/2n}. \tag{2.12}$$

Standard deviation σ of the signatures. Signatures are sampled from a discrete Gaussian distribution using the fast Fourier sampling algorithm (with $\mathbf{B}$ as a basis and a standard deviation σ). It suffices to take $\epsilon \leq 1/\sqrt{Q_s \cdot \lambda}$ and:

$$\begin{aligned} \sigma &= \frac{1}{\pi} \cdot \sqrt{\frac{\log(4n(1+1/\epsilon))}{2}} \cdot 1.17 \cdot \sqrt{q} \\ &\geq \eta_\epsilon(\mathbb{Z}^{2n}) \cdot \|\mathbf{B}\|_{\mathrm{GS}} \end{aligned} \tag{2.13}$$

Following [Pre17, Lemma 6], this ensures that $R_{2\lambda}(\mathcal{D} \cdot \mathbf{B} \| D_{\Lambda_q^\perp, \sigma, \mathbf{c}}) \lesssim 1 + O(1)/Q_s$, where $\mathcal{D}$ is the output of the sampler, $D_{\Lambda_q^\perp, \sigma, \mathbf{c}}$ is an ideal Gaussian and $R_{2\lambda}$ is the Rényi divergence between them. Following [Pre17, Section 3.3], $O(1)$ bits of security are lost by using our sampler instead of $D_{\Lambda_q^\perp, \sigma, \mathbf{c}}$.

Maximal norm β of the signatures. During the signing and verification procedures, signatures (s_1, s_2) must verify $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ in order to be accepted, with:

$$\beta = \tau_{\text{SIG}} \cdot \sigma\sqrt{2n}, \qquad \tau_{\text{SIG}} = 1.1 \tag{2.14}$$

We call τ_{SIG} the tailcut rate of signatures, because the expected value of $\|(s_1, s_2)\|$ is $\sigma\sqrt{2n}$; any signature larger than this expected value by a factor more than τ_{SIG} is rejected. By applying [Lyu12, Lemma 4.4, Item 3], the probability that a sampled signature is larger than β (hence that the signing procedure has to restart) is upper bounded as follows:

$$\mathbb{P}[\|(s_1, s_2)\|^2 > \lfloor \beta^2 \rfloor] \leq \tau_{\text{SIG}}^{2n} \cdot e^{n\left(1 - \tau_{\text{SIG}}^2\right)}. \tag{2.15}$$

2.7 Advantages and Limitations of Falcon

2.7.1 Advantages

Compactness. The main advantage of Falcon is its compactness. This doesn't really come as a surprise as Falcon was designed with compactness as the main criterion. Stateless hash-based signatures often have small public keys, but large signatures. Conversely, some multivariate schemes achieve very small signatures but require large public keys. Lattice-based schemes [LDK+19] can offer the best of both worlds, but no NIST candidate gets $|\mathsf{pk}| + |\mathsf{sig}|$ to be as small as Falcon does.

Fast signature generation and verification. The signature generation and verification procedures are very fast. This is especially true for the verification algorithm, but even the signature algorithm can perform more than 1000 signatures per second on a moderately-powered computer.

Security in the ROM and QROM. The GPV framework comes with a security proof in the random oracle (ROM), and a security proof in the quantum random oracle model (QROM) was later provided in [BDF+11]. See also [CD20]. In contrast, the Fiat-Shamir heuristic has only recently been proven secure in the QROM, and under certain conditions [LZ19, DFMS19].

Modular design. The design of FALCON is modular. Indeed, we instantiate the GPV framework with NTRU lattices, but it would be easy to replace NTRU lattices with another class of lattices if necessary. Similarly, we use fast Fourier sampling as our trapdoor sampler, but it is not necessary either. Actually, an extreme simplicity/speed trade-off would be to replace our fast Fourier sampler with Klein's sampler: signature generation would be two orders of magnitudes slower, but it would be simpler to implement and its black-box security would be the same.

Signatures with message recovery. In [dLP16], it has been shown that a preliminary version of FALCON can be instantiated in message-recovery mode: the message m can be recovered from the signature sig. It makes the signature twice longer, but allows to entirely recover a message which size is slightly less than half the size of the original signature. In situations where we can apply it, it makes FALCON even more competitive from a compactness viewpoint.

Key recovery mode. FALCON can also be instantiated in key-recovery mode. In this mode, The signature becomes twice longer but the key is reduced to a single hash value. In addition to incurring a very short key, this reduces the total size $|\mathsf{pk}| + |\mathsf{sig}|$ by about 15%. More details are given in Section 3.12.

Identity-based encryption. As shown in [DLP14], FALCON can be converted into an identity-based encryption scheme in a straightforward manner.

Easy signature verification. The signature procedure is very simple: essentially, one just needs to compute $[H(\mathsf{r}\|\mathsf{m}) - s_2 h] \bmod q$, which boils down to a few NTT operations and a hash computation.

2.7.2 Limitations

Delicate implementation. We believe that both the key generation procedure and the fast Fourier sampling are non-trivial to understand and delicate to implement, and constitute the main shortcoming of FALCON. On the bright side, the fast Fourier sampling uses subroutines of the fast Fourier transform as well as trees, two objects most implementers are familiar with.

Floating-point arithmetic. Our signing procedure uses floating-point arithmetic with 53 bits of precision. While this poses no problem for a software implementation, it may prove to be a major limitation when implementation on constrained devices – in particular those without a floating-point unit – will be considered.

We previously listed "unclear side-channel resistance" as a limitation of FALCON, due to discrete Gaussian sampling over the integers. This is much less the case now: constant-time implementations for this step and for the whole scheme are provided in [HPRR20] and [Por19], respectively. A challenging next step is to implement FALCON in a masked fashion.

Chapter 3

Specification of Falcon

3.1 Overview

Main elements in Falcon are polynomials of degree n with integer coefficients. The degree n is normally a power of two (typically 512 or 1024). Computations are done modulo a monic polynomial of degree n denoted ϕ (which is always of the form $\phi = x^n + 1$).

Mathematically, within the algorithm, some polynomials are interpreted as vectors, and some others as matrices: a polynomial f modulo ϕ then stands for a square $n \times n$ matrix, whose rows are $x^i f \mod \phi$ for all i from 0 to $n - 1$. It can be shown that addition and multiplication of such matrices map to addition and multiplication of polynomials modulo ϕ. We can therefore express most of Falcon in terms of operations on polynomials, even when we really are handling matrices that define a *lattice*.

The public key is a basis for a lattice of dimension $2n$:

$$\left[\begin{array}{c|c} -h & I_n \\ \hline qI_n & O_n \end{array}\right] \tag{3.1}$$

where I_n is the identity matrix of dimension n, O_n contains only zeros, and h is a polynomial modulo ϕ that stands for an $n \times n$ sub-matrix, as explained above. Coefficients of h are integers that range from 0 to $q - 1$, where q is a specific small prime (in the recommended parameters, $q = 12289$).

The corresponding private key is another basis for the very same lattice, expressed as:

$$\left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right] \tag{3.2}$$

where f, g, F and G are short integral polynomials modulo ϕ, that fulfil the two following relations:

$$\begin{array}{rcll} h & = & g/f & \mod \phi \mod q \\ fG - gF & = & q & \mod \phi \end{array} \tag{3.3}$$

Such a lattice is known as a *complete NTRU lattice*, and the second relation, in particular, is called the *NTRU equation*. Take care that while the relation $h = g/f$ is expressed modulo q, the lattice itself, and the polynomials, use nominally unbounded integers.

Key pair generation involves choosing random f and g polynomials using an appropriate distribution that yields short, but not too short, vectors; then, the NTRU equation is solved to find matching F and G. Keys are described in Section 3.4, and their generation is covered in Section 3.8.

Signature generation consists in first hashing the message to sign, along with a random nonce, into a polynomial c modulo ϕ, whose coefficients are uniformly mapped to integers in the 0 to $q - 1$ range; this process is described in Section 3.7. Then, the signer uses his knowledge of the secret lattice basis (f, g, F, G) to produce a pair of short polynomials (s_1, s_2) such that $s_1 = c - s_2 h \bmod \phi \bmod q$. The signature properly said is s_2.

Finding small vectors s_1 and s_2 is, in all generality, an expensive process. Falcon leverages the special structure of ϕ to implement it as a divide-and-conquer algorithm similar to the Fast Fourier Transform, which greatly speeds up operations. Moreover, some "noise" is added to the sampled vectors, with carefully tuned Gaussian distributions, to prevent signatures from leaking too much information about the private key. The signature generation process is described in Section 3.9.

Signature verification consists in recomputing s_1 from the hashed message c and the signature s_2, and then verifying that (s_1, s_2) is an appropriately short vector. Signature verification can be done entirely with integer computations modulo *q*; it is described in Section 3.10.

Encoding formats for keys and signatures are described in Section 3.11. In particular, since the signature is a short polynomial s_2, its elements are on average close to 0, which allows for a custom compressed format that reduces signature size.

Recommended parameters for several security levels are defined in Section 3.13.

3.2 Technical Overview

In this section, we provide an overview of the used techniques. As Falcon is arguably math-heavy, a clear comprehension of the mathematical principles in action goes a long way towards understanding and implementing it.

Falcon works with elements in number fields of the form $\mathbb{Q}[x]/(\phi)$, with $\phi = x^n + 1$ for $n = 2^\kappa$ a power-of-two. We note that ϕ is a cyclotomic polynomial, therefore it can be written as $\phi(x) = \prod_{k \in \mathbb{Z}_m^\times}(x - \zeta^k)$, with $m = 2n$ and ζ an arbitrary primitive m-th root of 1 (e.g. $\zeta = \exp(\frac{2i\pi}{m})$).

The interesting part about these number fields $\mathbb{Q}[x]/(\phi)$ is that they come with a tower-of-fields structure. Indeed, we have the following tower of fields:

$$\mathbb{Q} \subseteq \mathbb{Q}[x]/(x^2+1) \subseteq \cdots \subseteq \mathbb{Q}[x]/(x^{n/2}+1) \subseteq \mathbb{Q}[x]/(x^n+1) \tag{3.4}$$

We will rely on this tower-of-fields structure. Even more importantly for our purposes, by splitting polynomials between their odd and even coefficients we have the following chain of space isomorphisms:

$$\mathbb{Q}^n \cong (\mathbb{Q}[x]/(x^2+1))^{n/2} \cong \ldots \cong (\mathbb{Q}[x]/(x^{n/2}+1))^2 \cong \mathbb{Q}[x]/(x^n+1) \tag{3.5}$$

(3.4) and (3.5) remain valid when replacing $\mathbb{Q}$ by $\mathbb{Z}$, in which case they describe a tower of rings and a chain of module isomorphisms.

We will see in Section 3.6 that for appropriately defined multiplications, these are actually chains of *ring* isomorphisms. (3.5) will be used to make our signature generation fast and “good”: in lattice-based cryptography, the smaller the norm of signatures are, the better. So by “good” we mean that our signature generation will output signatures with a small norm.

On one hand, classical algebraic operations in the field $\mathbb{Q}[x]/(x^n+1)$ are fast, and using them will make our signature generation fast. On the other hand, we will use the isomorphisms exposed in (3.5) as a leverage to output signatures with small norm. Using these endomorphisms to their full potential entails manipulating individual coefficients of polynomials (or of their Fourier transform) and working with binary trees.

3.3 Notations

Cryptographic parameters. For a cryptographic signature scheme, λ denotes its security level and Q_s the maximal number of signing queries. Following [NIS16], we assume $Q_s = 2^{64}$.

Matrices, vectors and scalars. Matrices will usually be in bold uppercase (e.g. $\mathbf{B}$), vectors in bold lowercase (e.g. $\mathbf{v}$) and scalars – which include polynomials – in italic (e.g. s). We use the row convention for vectors. The transpose of a matrix $\mathbf{B}$ may be noted $\mathbf{B}^t$. It is to be noted that for a polynomial f, we do *not* use f' to denote its derivative in this document.

Quotient rings. For $q \in \mathbb{N}^\star$, we denote by $\mathbb{Z}_q$ the quotient ring $\mathbb{Z}/q\mathbb{Z}$. In Falcon, our integer modulus $q = 12289$ is prime so $\mathbb{Z}_q$ is also a finite field. We denote by $\mathbb{Z}_q^\times$ the group of invertible elements of $\mathbb{Z}_q$, and by φ Euler's totient function: $\varphi(q) = |\mathbb{Z}_q^\times| = q - 1 = 3 \cdot 2^{12}$ since q is prime.

Number fields. Falcon uses a polynomial modulus $\phi = x^n + 1$ (for $n = 2^\kappa$). It is a monic polynomial of $\mathbb{Z}[x]$, irreducible in $\mathbb{Q}[x]$ and with distinct roots over $\mathbb{C}$.

Let $a = \sum_{i=0}^{n-1} a_i x^i$ and $b = \sum_{i=0}^{n-1} b_i x^i$ be arbitrary elements of the number field $\mathcal{Q} = \mathbb{Q}[x]/(\phi)$. We note $a^\star$ and call (Hermitian) adjoint of a the unique element of $\mathcal{Q}$ such that for any root ζ of ϕ, $a^\star(\zeta) = \overline{a(\zeta)}$, where $\bar{\cdot}$ is the usual complex conjugation over $\mathbb{C}$. For $\phi = x^n + 1$, the Hermitian adjoint $a^\star$ can be expressed simply:

$$a^\star = a_0 - \sum_{i=1}^{n-1} a_i x^{n-i} \tag{3.6}$$

We extend this definition to vectors and matrices: the adjoint $\mathbf{B}^\star$of a matrix $\mathbf{B} \in \mathcal{Q}^{n\times m}$ (resp. a vector $\mathbf{v}$) is the component-wise adjoint of the transpose of $\mathbf{B}$ (resp. $\mathbf{v}$):

$$\mathbf{B} = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \quad \Leftrightarrow \quad \mathbf{B}^\star = \left[\begin{array}{c|c} a^\star & c^\star \\ \hline b^\star & d^\star \end{array}\right] \tag{3.7}$$

Inner product. The inner product $\langle \cdot, \cdot \rangle$ over $\mathcal{Q}$ and its associated norm $\| \cdot \|$ are

$$\langle a, b \rangle = \frac{1}{\deg(\phi)} \sum_{\phi(\zeta)=0} a(\zeta) \cdot \overline{b(\zeta)} \qquad (3.8) \qquad\qquad \|a\| = \sqrt{\langle a, a \rangle} \qquad (3.9)$$

We extend these definitions to vectors: for $\mathbf{u} = (u_i)_i$ and $\mathbf{v} = (v_i)_i$ in $\mathcal{Q}^m$, $\langle \mathbf{u}, \mathbf{v} \rangle = \sum_i \langle u_i, v_i \rangle$. For our choice of ϕ, the inner product coincides with the usual coefficient-wise inner product:

$$\langle a, b \rangle = \sum_{0 \leq i < n} a_i b_i; \tag{3.10}$$

From an algorithmic point of view, computing the inner product or the norm is most easily done by using (3.8) if polynomials are in FFT representation, and by using (3.10) if they are in coefficient representation.

Ring Lattices. For the rings $\mathcal{Q} = \mathbb{Q}[x]/(\phi)$ and $\mathcal{Z} = \mathbb{Z}[x]/(\phi)$, positive integers $m \geq n$ and a full-rank matrix $\mathbf{B} \in \mathcal{Q}^{n \times m}$, we denote by $\Lambda(\mathbf{B})$ and call lattice generated by $\mathbf{B}$ the set $\mathcal{Z}^n \cdot \mathbf{B} = \{\mathbf{z}\mathbf{B} | \mathbf{z} \in \mathcal{Z}^n\}$. By extension, a set Λ is a lattice if there exists a matrix $\mathbf{B}$ such that $\Lambda = \Lambda(\mathbf{B})$. We may say that $\Lambda \subseteq \mathcal{Z}^m$ is a q-ary lattice if $q\mathcal{Z}^m \subseteq \Lambda$.

Discrete Gaussians. For $\sigma, \mu \in \mathbb{R}$ with $\sigma > 0$, we define the Gaussian function $\rho_{\sigma,\mu}$ as $\rho_{\sigma,\mu}(x) = \exp(-|x - \mu|^2/2\sigma^2)$, and the discrete Gaussian distribution $D_{\mathbb{Z},\sigma,\mu}$ over the integers as

$$D_{\mathbb{Z},\sigma,\mu}(x) = \frac{\rho_{\sigma,\mu}(x)}{\sum_{z \in \mathbb{Z}} \rho_{\sigma,\mu}(z)}. \tag{3.11}$$

The parameter μ may be omitted when it is equal to zero.

The Gram-Schmidt orthogonalization. Any matrix $\mathbf{B} \in \mathcal{Q}^{n \times m}$ can be decomposed as follows:

$$\mathbf{B} = \mathbf{L} \times \tilde{\mathbf{B}}, \tag{3.12}$$

where $\mathbf{L}$ is lower triangular with 1's on the diagonal, and the rows $\tilde{\mathbf{b}}_i$'s of $\tilde{\mathbf{B}}$ verify $\langle \mathbf{b}_i, \mathbf{b}_j \rangle = 0$ for $i \neq j$. When $\mathbf{B}$ is full-rank , this decomposition is unique, and it is called the Gram-Schmidt orthogonalization (or GSO). We will also call Gram-Schmidt norm of $\mathbf{B}$ the following value:

$$\|\mathbf{B}\|_{\text{GS}} = \max_{\tilde{\mathbf{b}}_i \in \tilde{\mathbf{B}}} \|\tilde{\mathbf{b}}_i\|. \tag{3.13}$$

The LDL* decomposition. The LDL* decomposition writes any full-rank Gram matrix as a product $\mathbf{LDL}^\star$, where $\mathbf{L} \in \mathcal{Q}^{n \times n}$ is lower triangular with 1's on the diagonal, and $\mathbf{D} \in \mathcal{Q}^{n \times n}$ is diagonal.

The LDL* decomposition and the GSO are closely related as for a basis $\mathbf{B}$, there exists a unique GSO $\mathbf{B} = \mathbf{L} \cdot \tilde{\mathbf{B}}$ and for a full-rank Gram matrix $\mathbf{G}$, there exists a unique LDL* decomposition $\mathbf{G} = \mathbf{LDL}^\star$.

If $\mathbf{G} = \mathbf{B}\mathbf{B}^\star$, then $\mathbf{G} = \mathbf{L} \cdot (\tilde{\mathbf{B}}\tilde{\mathbf{B}}^\star) \cdot \mathbf{L}^\star$ is a valid LDL* decomposition of $\mathbf{G}$. As both decompositions are unique, the matrices $\mathbf{L}$ in both cases are actually the same. In a nutshell:

$$\left[\mathbf{L} \cdot \tilde{\mathbf{B}} \text{ is the GSO of } \mathbf{B}\right] \Leftrightarrow \left[\mathbf{L} \cdot (\tilde{\mathbf{B}}\tilde{\mathbf{B}}^\star) \cdot \mathbf{L}^\star \text{ is the LDL}^\star \text{ decomposition of } (\mathbf{B}\mathbf{B}^\star)\right]. \tag{3.14}$$

The reason why we present both equivalent decompositions is because the GSO is a more familiar concept in lattice-based cryptography, whereas the use of LDL* decomposition is faster and therefore makes more sense from an algorithmic point of view.

3.4 Keys

3.4.1 Public Parameters

Public keys use some public parameters that are shared by many key pairs:

1. The cyclotomic polynomial $\phi = x^n + 1$, where $n = 2^\kappa$ is a power of 2. We note that ϕ is monic and irreducible.
2. A modulus $q \in \mathbb{N}^\star$. In Falcon, $q = 12289$. We note that $(\phi \bmod q)$ splits over $\mathbb{Z}_q[x]$.
3. A real bound $\lfloor \beta^2 \rfloor > 0$.
4. Standard deviations σ and $\sigma_{\min} < \sigma_{\max}$.
5. A signature bytelength sbytelen.

For clarity, public parameters may be omitted (e.g. in algorithms' headers) when clear from context.

3.4.2 Private Key

The core of a Falcon private key sk consists of four polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ with short integer coefficients, verifying the NTRU equation:

$$fG - gF = q \bmod \phi. \tag{3.15}$$

The polynomial f shall furthermore be invertible in $\mathbb{Z}_q[x]/(\phi)$.

Given f and g such that there exists a solution (F, G) to the NTRU equation, F and G may be recomputed dynamically, but that process is computationally expensive; therefore, it is normally expected that at least F will be stored along f and g (given f, g and F, G can be efficiently recomputed).

Two additional elements are computed from the private key, and may be recomputed dynamically, or stored along f, g and F:

- The FFT representations of f, g, F and G, ordered in the form of a matrix:

$$\hat{\mathbf{B}} = \left[\begin{array}{c|c} \mathsf{FFT}(g) & -\mathsf{FFT}(f) \\ \hline \mathsf{FFT}(G) & -\mathsf{FFT}(F) \end{array}\right], \tag{3.16}$$

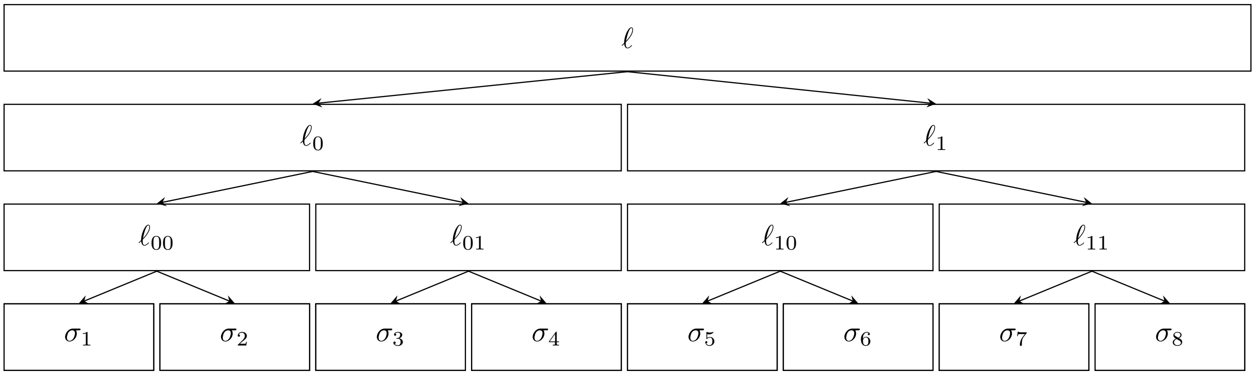

Figure 3.1: A Falcon tree of height 3

$\mathsf{FFT}(a)$ being the fast Fourier transform of a in the underlying ring (here, $\mathbb{R}[x]/(\phi)$).

- A Falcon tree T, described at the end of this section.

FFT representations are described in Section 3.5. The FFT representation of a polynomial formally consists of n complex numbers (a complex number is normally encoded as two 64-bit floating-point values); however, the FFT representation of a *real* polynomial f is redundant, because for each complex root ζ of ϕ, its conjugate $\overline{\zeta}$ is also a root of ϕ, and $f(\overline{\zeta}) = \overline{f(\zeta)}$. Therefore, the FFT representation of a polynomial may be stored as $n/2$ complex numbers, and $\hat{\mathbf{B}}$, when stored, requires $2n$ complex numbers.

Falcon trees. Falcon trees are binary trees defined inductively as follows:

- A Falcon tree T of height 0 consists of a single node whose value is a real $\sigma > 0$.
- A Falcon tree T of height κ verifies these properties:
  - The value of its root, noted $\mathsf{T.value}$, is a polynomial $\ell \in \mathbb{Q}[x]/(x^n + 1)$ with $n = 2^\kappa$.
  - Its left and right children, noted $\mathsf{T.leftchild}$ and $\mathsf{T.rightchild}$, are Falcon trees of height $\kappa - 1$.

The values of internal nodes – which are real polynomials – are stored in FFT representation (i.e. as complex numbers, see Section 3.5 for a formal definition). Hence all the nodes of a Falcon tree contain polynomials in FFT representation, except the leaves which contain real values > 0.

A Falcon tree of height 3 is represented in fig. 3.1. As illustrated by the figure, a Falcon tree can be easily represented by an array of $2^\kappa(1 + \kappa)$ complex numbers (or exactly half as many, if the redundancy of FFT representation is leveraged, as explained above), and access to the left and right children can be performed efficiently using simple pointer arithmetic.

The contents of a Falcon tree T are computed from the private key elements f, g, F and G using the algorithm described in Section 3.8.3 (see also algorithm 4).

3.4.3 Public key

The FALCON public key pk corresponding to the private key sk $= (f, g, F, G)$ is a polynomial $h \in \mathbb{Z}_q[x]/(\phi)$ such that:

$$h = gf^{-1} \bmod (\phi, q). \tag{3.17}$$

3.5 FFT and NTT

The FFT. Let $f \in \mathbb{Q}[x]/(\phi)$. We note Ω_ϕ the set of complex roots of ϕ. We suppose that ϕ is monic with distrinct roots over $\mathbb{C}$, so that $\phi(x) = \prod_{\zeta \in \Omega_\phi} (x - \zeta)$. We denote by $\mathsf{FFT}_\phi(f)$ the fast Fourier transform of f with respect to ϕ:

$$\mathsf{FFT}_\phi(f) = (f(\zeta))_{\zeta \in \Omega_\phi} \tag{3.18}$$

When ϕ is clear from context, we simply note $\mathsf{FFT}(f)$. We may also use the notation $\hat{f}$ to indicate that $\hat{f}$ is the FFT of f. FFT_ϕ is a ring isomorphism, and we note invFFT_ϕ its inverse. The multiplication in the FFT domain is denoted by $\odot$. We extend the FFT and its inverse to matrices and vectors by component-wise application.

Additions, subtractions, multiplications and divisions of polynomials modulo ϕ can be computed in FFT representations by simply performing them on each coordinate. In particular, this makes multiplications and divisions very efficient.

For $\phi = x^n + 1$, the set of complex roots ζ of ϕ is:

$$\Omega_\phi = \left\{ \exp\left(\frac{i(2k+1)\pi}{n}\right) \middle| \, 0 \le k < n \right\} \tag{3.19}$$

A note on implementing the FFT. There exist several ways of implementing the FFT, which may yield slightly different results. For example, some implementations of the FFT scale our definition by a constant factor (e.g. $1/\deg(\phi)$). Another differentiation point is the order of (the roots of) the FFT. Common orders are the increasing order (i.e. the roots are sorted by their order on the unit circle, starting at 1 and moving clockwise) or (variants of) the bit-reversal order. In the case of FALCON:

- The FFT is not scaled by a constant factor.
- There is no constraint on the order of the FFT, the choice is left to the implementer. However, the chosen order shall be consistent for all the algorithms using the FFT.

Representation of polynomials in algorithms. The algorithms which specify FALCON heavily rely on the fast Fourier transform, and some of them explicitly require that the inputs and/or outputs are given in FFT representation. When the directive "Format:" is present at the beginning of an algorithm, it specifies in which format (coefficient or FFT representation) the input/output polynomials shall be represented. When the directive "Format:" is absent, no assumption on the format of the input/output polynomials is made.

The NTT. The NTT (Number Theoretic Transform) is the analog of the FFT in the field $\mathbb{Z}_p$, where p is a prime such that $p = 1 \bmod 2n$. Under these conditions, ϕ has exactly n roots (ω_i) over $\mathbb{Z}_p$, and any polynomial $f \in \mathbb{Z}_p[x]/(\phi)$ can be represented by the values $f(\omega_i)$. Conversion to and from NTT representation can be done efficiently in $O(n \log n)$ operations in $\mathbb{Z}_p$. When in NTT representation, additions, subtractions, multiplications and divisions of polynomials (modulo ϕ and p) can be performed coordinate-wise in $\mathbb{Z}_p$.

In Falcon, the NTT allows for faster implementations of public key operations (using $\mathbb{Z}_q$) and key pair generation (with various medium-sized primes p). Private key operations, though, rely on the fast Fourier sampling, which uses the FFT, not the NTT.

3.6 Splitting and Merging

In this section, we make explicit the chains of isomorphisms described in Section 3.2, by presenting splitting (resp. merging) operators which allow to travel these chains from right to left (resp. left to right).

Let ϕ, ϕ' be cyclotomic polynomials such that $\phi(x) = \phi'(x^2)$ (for example, $\phi(x) = x^n + 1$ and $\phi'(x) = x^{n/2}+1$). We define operators which are at the heart of our signing algorithm. Our algorithms require the ability to split an element of $\mathbb{Q}[x]/(\phi)$ into two smaller elements of $\mathbb{Q}[x]/(\phi')$. Conversely, we require the ability to merge two elements of $\mathbb{Q}[x]/(\phi')$ into an element of $\mathbb{Q}[x]/(\phi)$.

The splitfft operator. Let n be the degree of ϕ, and $f = \sum_{i=0}^{n-1} a_i x^i$ be an arbitrary element of $\mathbb{Q}[x]/(\phi)$, f can be decomposed uniquely as $f(x) = f_0(x^2) + x f_1(x^2)$, with $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$. In coefficient representation, such a decomposition is straightforward to write:

$$f_0 = \sum_{0 \le i < n/2} a_{2i} x^i \quad \text{and} \quad f_1 = \sum_{0 \le i < n/2} a_{2i+1} x^i \tag{3.20}$$

In (3.20), we simply split f with respect to its even or odd coefficients. With this notation, we note:

$$\mathsf{split}(f) = (f_0, f_1). \tag{3.21}$$

In Falcon, polynomials are repeatedly split, multiplied together, split again and so forth. To avoid switching back and forth between the coefficient and FFT representation, we always perform the split operation in the FFT representation. It is defined in splitfft (algorithm 1).

splitfft is split realized in the FFT representation: for any f, $\mathsf{FFT}(\mathsf{split}(f)) = \mathsf{splitfft}(\mathsf{FFT}(f))$. Readers familiar with the Fourier transform will recognize that splitfft is a subroutine of the inverse fast Fourier transform, more precisely the part which from $\mathsf{FFT}(f)$ computes two FFT's twice smaller.

The mergefft operator. With the previous notations, we define the operator merge as follows:

$$\mathsf{merge}(f_0, f_1) = f_0(x^2) + x f_1(x^2) \in \mathbb{Q}[x]/(\phi). \tag{3.22}$$

Algorithm 1 splitfft($\mathsf{FFT}(f)$)

```
Require: FFT(f) = $(f(\zeta))_\zeta$ for some $f \in \mathbb{Q}[x]/(\phi)$
Ensure: FFT($f_0$) = $(f_0(\zeta'))_{\zeta'}$ and FFT($f_1$) = $(f_1(\zeta'))_{\zeta'}$ for some $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$
Format: All polynomials are in FFT representation.
for $\zeta$ such that $\phi(\zeta) = 0$ and $\mathrm{Im}(\zeta) > 0$ do        ▷ See eq. (3.19) with $0 \leq k < n/2$
    $\zeta' \leftarrow \zeta^2$
    $f_0(\zeta') \leftarrow \frac{1}{2}[f(\zeta) + f(-\zeta)]$
    $f_1(\zeta') \leftarrow \frac{1}{2\zeta}[f(\zeta) - f(-\zeta)]$
return (FFT($f_0$), FFT($f_1$))
```

Algorithm 2 mergefft(f_0, f_1)

```
Require: FFT($f_0$) = $(f_0(\zeta'))_{\zeta'}$ and FFT($f_1$) = $(f_1(\zeta'))_{\zeta'}$ for some $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$
Ensure: FFT(f) = $(f(\zeta))_\zeta$ for some $f \in \mathbb{Q}[x]/(\phi)$
Format: All polynomials are in FFT representation.
for $\zeta$ such that $\phi(\zeta) = 0$ do        ▷ See eq. (3.19)
    $\zeta' \leftarrow \zeta^2$
    $f(\zeta) \leftarrow f_0(\zeta') + \zeta f_1(\zeta')$
return FFT(f)
```

Similarly to split, it is often relevant from an efficiently standpoint to perform merge in the FFT representation. This is done in mergefft (algorithm 2).

It is immediate that split and merge are inverses of each other, and equivalently splitfft and mergefft are inverses of each other. Just as for splitfft, readers familiar with the Fourier transform can observe that mergefft is a step of the fast Fourier transform: it is the reconstruction step which from two small FFT's computes a larger FFT.

Relationship with the FFT. There is no requirement on the order in which the values $f(\zeta)$ (resp. $f_0(\zeta')$, resp. $f_1(\zeta')$) are to be stored, and the choice of this order is left to the implementer. It is however recommended to use a unique order convention for the FFT, invFFT, splitfft and mergefft operators. Since the FFT and invFFT need to implemented anyway, this unique convention can be achieved e.g. by implementing splitfft as part of invFFT, and mergefft as part of the FFT.

The intricate relationships between the split and merge operators, their counterparts in the FFT representation and the (inverse) fast Fourier transform are illustrated in the commutative diagram of fig. 3.2.

3.6.1 Algebraic interpretation

The purpose of the splitting and merging operators that we defined is not only to represent an element of $\mathbb{Q}[x]/(\phi)$ using two elements of $\mathbb{Q}[x]/(\phi')$, but to do so in a manner compatible with ring operations.

$$\begin{tikzcd}
f \in \mathbb{Q}[x]/(\phi) \arrow[r, shift left, "\mathsf{split}\ (3.21)"] \arrow[d, shift right, "\mathsf{FFT}"'] & f_0, f_1 \in \mathbb{Q}[x]/(\phi') \arrow[l, shift left, "\mathsf{merge}\ (3.22)"] \arrow[d, shift right, "\mathsf{FFT}"'] \\
\hat{f} \in \mathsf{FFT}(\mathbb{Q}[x]/(\phi)) \arrow[u, shift right, "\mathsf{invFFT}"'] \arrow[r, shift left, "\mathsf{splitfft}"] & \hat{f}_0, \hat{f}_1 \in \mathsf{FFT}(\mathbb{Q}[x]/(\phi')) \arrow[l, shift left, "\mathsf{mergefft}"] \arrow[u, shift right, "\mathsf{invFFT}"']
\end{tikzcd}$$

Figure 3.2: Relationship between FFT, invFFT, split, merge, splitfft and mergefft

As an illustration, we consider the operation:

$$a = bc \tag{3.23}$$

where $a, b, c \in \mathbb{Q}[x]/(\phi)$. For $f \in \mathbb{Q}[x]/(\phi)$, we consider the associated endomorphism $\psi_f : z \in \mathbb{Q}[x]/(\phi) \mapsto fz$. (3.23) can be rewritten as $a = \psi_c(b)$. By the split isomorphism, a and b (resp. ψ_c) can also be considered as elements (resp. an endomorphism) of $(\mathbb{Q}[x]/(\phi'))^2$. We can rewrite (3.23) as:

$$\left[\begin{array}{c|c} a_0 & a_1 \end{array}\right] = \left[\begin{array}{c|c} b_0 & b_1 \end{array}\right] \left[\begin{array}{c|c} c_0 & c_1 \\ \hline xc_1 & c_0 \end{array}\right] \tag{3.24}$$

More formally, we have used the fact that splitting operators are isomorphisms between $\mathbb{Q}[x]/(\phi)$ and $(\mathbb{Q}[x]/(\phi'))^k$, which express elements of $\mathbb{Q}[x]/(\phi)$ in the $(\mathbb{Q}[x]/(\phi'))$-basis $\{1, x\}$ (hence "breaking" a, b in vectors over a smaller field). Similarly, writing the transformation matrix of the endomorphism ψ_c in the basis $\{1, x\}$ yields the 2×2 matrix of (3.24).

Relationship with the field norm. The field norm (or relative norm) $\mathrm{N}_{\mathbb{L}/\mathbb{K}}$ maps elements of a larger field $\mathbb{L}$ onto a subfield $\mathbb{K}$. It is an important notion in field theory, but in this document, we only need to define it for a simple, particular case. Let $n = 2^\kappa$ a power of two, $\mathbb{L} = \mathbb{Q}[x]/(x^n + 1)$ and $\mathbb{K} = \mathbb{Q}[x]/(x^{n/2} + 1)$. We define the field norm $\mathrm{N}_{\mathbb{L}/\mathbb{K}}$ as follows:

$$\begin{array}{rcccc} \mathrm{N}_{\mathbb{L}/\mathbb{K}} & : & \mathbb{L} & \to & \mathbb{K} \\ & & f & \mapsto & f_0^2 - xf_1^2 \end{array} \tag{3.25}$$

where $(f_0, f_1) = \mathsf{split}(f) \in \mathbb{K}^2$, see (3.20) and (3.21) for explicit formulae. When $\mathbb{L}$ and $\mathbb{K}$ are clear from context, we simply note $\mathrm{N}(f) = \mathrm{N}_{\mathbb{L}/\mathbb{K}}(f)$. An equivalent formulation for $\mathrm{N}_{\mathbb{L}/\mathbb{K}}$ is:

$$\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f) = f(x) \cdot f(-x) \tag{3.26}$$

Both (3.25) and (3.26) are valid formulae for $\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f)$, but (3.25) is more suited to the coefficient representation, and (3.26) is more suited to the NTT representation.

3.7 Hashing

As for any hash-and-sign signature scheme, the first step to sign a message or verify a signature consists of hashing the message. In our case, the message needs to be hashed into a polynomial in $\mathbb{Z}_q[x]/(\phi)$. An approved extendable-output hash function (XOF), as specified in FIPS 202 [NIS15], shall be used during this procedure.

This XOF shall have a security level at least equal to the security level targeted by our signature scheme. In addition, we should be able to start hashing a message without knowing the security level at which it will be signed. For these reasons, we use a unique XOF for all security levels: SHAKE-256.

- SHAKE-256 -Init () denotes the initialization of a SHAKE-256 hashing context;
- SHAKE-256 -Inject (ctx, str) denotes the injection of the data str in the hashing context ctx;
- SHAKE-256 -Extract (ctx, b) denotes extraction from a hashing context ctx of b bits of pseudo-randomness.

HashToPoint (algorithm 3) defines the hashing process used in Falcon. It is defined for any $q \leq 2^{16}$. In Falcon, big-endian convention is used to interpret a chunk of b bits, extracted from a SHAKE-256 instance, into an integer in the 0 to $2^b - 1$ range (the first of the b bits has numerical weight 2^{b-1}, the last has weight 1).

```
Algorithm 3 HashToPoint(str, $q$, $n$)
Require: A string str, a modulus $q \leq 2^{16}$, a degree $n \in \mathbb{N}^\star$
Ensure: An polynomial $c = \sum_{i=0}^{n-1} c_i x^i$ in $\mathbb{Z}_q[x]$
 1: $k \leftarrow \lfloor 2^{16}/q \rfloor$
 2: ctx ← SHAKE-256-Init()
 3: SHAKE-256-Inject(ctx, str)
 4: $i \leftarrow 0$
 5: while $i < n$ do
 6:     $t \leftarrow$ SHAKE-256-Extract(ctx, 16)
 7:     if $t < kq$ then
 8:         $c_i \leftarrow t \bmod q$
 9:         $i \leftarrow i + 1$
10: return $c$
```

Possible variants.

- If $q > 2^{16}$, then larger chunks can be extracted from SHAKE-256 at each step.
- HashToPoint may be difficult to efficiently implement in a constant-time way; constant-timeness may be a desirable feature if the signed data is also secret.

  A variant which is easier to implement with constant-time code extracts 64 bits instead of 16 at step 6, and omits the conditional check of step 7. While the omission of the check means that some outputs are slightly more probable than others, a Rényi argument [BLL+15, Pre17] allows to claim that this variant is secure for the parameters set by NIST [NIS16].

Of course, any variant deviating from the procedure expressed in algorithm 3 implies that the same message will hash to a different value, which breaks interoperability.

3.8 Key Pair Generation

3.8.1 Overview

The key pair generation can be decomposed in two clearly separate parts.

- *Solving the NTRU equation.* The first step of the key pair generation consists of computing polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ which verify (3.15) – the NTRU equation. Generating f and g is easy; the hard part is to efficiently compute polynomials F, G such that (3.15) is verified.

  To do this, we propose a novel method that exploits the tower-of-rings structure highlighted in (3.4). We use the field norm N to map the NTRU equation onto a smaller ring $\mathbb{Z}[x]/(\phi')$ of the tower of rings, all the way down to $\mathbb{Z}$. We then solve the equation in $\mathbb{Z}$ – using an extended gcd – and use properties of the norm to lift the solutions (F, G) back to the original ring $\mathbb{Z}[x]/(\phi)$.

  Implementers should be mindful that this step does *not* perform modular reduction modulo q, which leads us to handle polynomials with large coefficients (a few thousands of bits per coefficient in the lowest levels of the recursion). See Section 3.8.2 for a formal specification of this step, and [PP19] for an in-depth analysis.

- *Computing a Falcon tree.* Once suitable polynomials f, g, F, G are generated, the second part of the key generation consists of preprocessing them into an adequate format: by adequate we mean that this format should be reasonably compact and allow fast signature generation on-the-go.

  Falcon trees are precisely this adequate format. To compute a Falcon tree, we compute the LDL* decomposition $\mathbf{G} = \mathbf{LDL}^\star$ of the matrix $\mathbf{G} = \mathbf{BB}^\star$, where

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right], \tag{3.27}$$

  which is equivalent to computing the Gram-Schmidt orthogonalization $\mathbf{B} = \mathbf{L} \times \tilde{\mathbf{B}}$. If we were using Klein's well-known sampler (or a variant thereof) as a trapdoor sampler, knowing $\mathbf{L}$ would be sufficient but a bit unsatisfactory as we would not exploit the tower-of-rings structure of $\mathbb{Q}[x]/(\phi)$.

  So instead of stopping there, we store $\mathbf{L}$ (or rather L_{10}, its bottom-left and only non-trivial term) in the root of a tree, use the splitting operators defined in Section 3.6 to "break" the diagonal elements D_{ii} of $\mathbf{D}$ into matrices $\mathbf{G}_i$ over smaller rings $\mathbb{Q}[x]/(\phi')$, at which point we create subtrees for each matrix $\mathbf{G}_i$ and recursively start over the process of LDL* decomposition and splitting.

  The recursion continues until the matrix $\mathbf{G}$ has its coefficients in $\mathbb{Q}$, which correspond to the bottom of the recursion tree. How this is done is specified in Section 3.8.3.

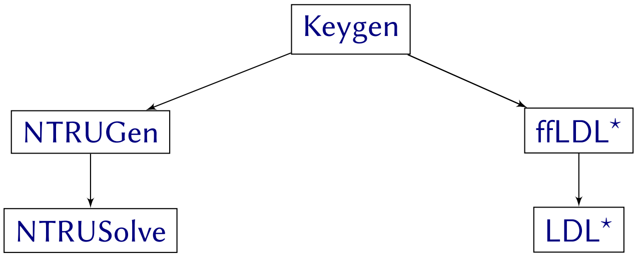

Figure 3.3: Flowchart of the key generation

> The main technicality of this part is that it exploits the tower-of-rings structure of $\mathbb{Q}[x]/(\phi)$ by breaking its elements onto smaller rings. In addition, intermediate results are stored in a tree, which requires precise bookkeeping as elements of different tree levels do not live in the same field. Finally, for performance reasons, the step is realized completely in the FFT domain.

Once these two steps are done, the rest of the key pair generation is straightforward. A final step normalizes the leaves of the LDL tree to turn it into a Falcon tree. The result is wrapped in a private key sk and the corresponding public key pk is $h = gf^{-1} \bmod q$.

A formal description is given in algorithms 4 to 9, the main algorithm being the procedure Keygen (algorithm 4). The general architecture of the key pair generation is also illustrated in fig. 3.3.

```
Algorithm 4 Keygen($\phi, q$)
Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus $q$
Ensure: A secret key sk, a public key pk
$f, g, F, G \leftarrow$ NTRUGen($\phi, q$)                              ▷ Solving the NTRU equation
$\mathbf{B} \leftarrow \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$
$\hat{\mathbf{B}} \leftarrow$ FFT($\mathbf{B}$)        ▷ Compute the FFT for each of the 4 components $\{g, -f, G, -F\}$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^\star$
T $\leftarrow$ ffLDL*($\mathbf{G}$)                                      ▷ Computing the LDL* tree
for each leaf leaf of T do                                          ▷ Normalization step
    leaf.value $\leftarrow \sigma/\sqrt{\text{leaf.value}}$
sk $\leftarrow (\hat{\mathbf{B}}, \text{T})$
$h \leftarrow gf^{-1} \bmod q$
pk $\leftarrow h$
return sk, pk
```

3.8.2 Generating the polynomials f, g, F, G.

The first step of the key pair generation generates suitable polynomials f, g, F, G verifying (3.15). This is specified in NTRUGen (algorithm 5). We provide a general explanation of NTRUGen:

1. First, the polynomials f, g are generated randomly. A few conditions over f, g are checked to ensure they are suitable for our purposes (line 7 to line 11). It particular:
    (a) Line 7 ensures a public key h can be computed from f, g. This is true if and only if f is invertible $\bmod q$, which is true if and only if $\mathsf{NTT}(f)$ contains no coefficient set to 0.

    (b) The polynomials f, g, F, G must allow to generate short signatures. This is true if:

    $$\gamma = \max\left\{\|(g, -f)\|, \left\|\left(\frac{qf^\star}{ff^\star + gg^\star}, \frac{qg^\star}{ff^\star + gg^\star}\right)\right\|\right\} \leq 1.17\sqrt{q}. \tag{3.28}$$

    We recall that the norm $\|\cdot\|$ is easily computed by using (3.9) with either (3.8) or (3.10), depending on the representation (FFT or coefficient).

2. Second, short polynomials F, G are computed such that f, g, F, G verify (3.15). This is done by the procedure NTRUSolve (algorithm 6).

```
Algorithm 5 NTRUGen($\phi, q$)
Require: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree $n$, a modulus $q$
Ensure: Polynomials $f, g, F, G$
 1: $\sigma_{\{f,g\}} \leftarrow 1.17\sqrt{q/2n}$          ▷ $\sigma_{\{f,g\}}$ is chosen so that $\mathbb{E}[\|(f,g)\|] = 1.17\sqrt{q}$
 2: for $i$ from 0 to $n-1$ do
 3:     $f_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$          ▷ See also (3.29)
 4:     $g_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$
 5: $f \leftarrow \sum_i f_i x^i$          ▷ $f \in \mathbb{Z}[x]/(\phi)$
 6: $g \leftarrow \sum_i g_i x^i$          ▷ $g \in \mathbb{Z}[x]/(\phi)$
 7: if $\mathsf{NTT}(f)$ contains 0 as a coefficient then          ▷ Check that $f$ is invertible $\bmod q$
 8:     restart
 9: $\gamma \leftarrow \max\left\{\|(g,-f)\|, \left\|\left(\frac{qf^\star}{ff^\star+gg^\star}, \frac{qg^\star}{ff^\star+gg^\star}\right)\right\|\right\}$          ▷ Using (3.9) with (3.8) or (3.10)
10: if $\gamma > 1.17\sqrt{q}$ then          ▷ Check that $\gamma = \|\mathbf{B}\|_{\mathrm{GS}}$ is short
11:     restart
12: $F, G \leftarrow$ NTRUSolve$_{n,q}(f, g)$          ▷ Computing $F, G$ such that $fG - gF = q \bmod \phi$
13: if $(F, G) = \perp$ then
14:     restart
15: return $f, g, F, G$
```

One way to sample $z \leftarrow D_{\sigma_{\{f,g\}}}$ (lines 5 and 6) is to perform:

$$z = \sum_{i=1}^{4096/n} z_i, \quad \text{where} \begin{cases} z_i \leftarrow \mathsf{SamplerZ}(0, \sigma^*), \\ \sigma^* = 1.17 \cdot \sqrt{\frac{q}{8192}} \approx 1.43300980528773 \end{cases} \tag{3.29}$$

This exploits the fact the sum of k Gaussians of standard deviation σ^* is a Gaussian of standard deviation $\sigma^*\sqrt{k}$. Here σ^* is chosen so that $\sigma^* \leq \sigma_{\max}$, see Section 3.9.3. Note that the reference code currently implements a similar idea, but with a $\sigma^* > \sigma_{\max}$ for which we sample using a precomputed table.

Solving the NTRU equation (3.15)

We now explain how to solve (3.15). As mentioned in Section 3.8.1, we repeatedly use the field norm N to map f, g to a smaller ring $\mathbb{Z}[x]/(x^{n/2}+1)$, until we reach the ring $\mathbb{Z}$. Solving (3.15) then amounts to computing an extended GCD over $\mathbb{Z}$, which is simple. We then use the multiplicative properties of the field norm to repeatedly lift the solutions up to $\mathbb{Z}[x]/(x^n+1)$, at which point we have solved (3.15).

Algorithm 6 $\mathsf{NTRUSolve}_{n,q}(f, g)$

```
Require: $f, g \in \mathbb{Z}[x]/(x^n+1)$ with $n$ a power of two
Ensure: Polynomials $F, G$ such that (3.15) is verified
if $n = 1$ then
    Compute $u, v \in \mathbb{Z}$ such that $uf - vg = \gcd(f, g)$        ▷ Using the extended GCD
    if $\gcd(f, g) \neq 1$ then
        abort and return $\perp$
    $(F, G) \leftarrow (vq, uq)$
    return $(F, G)$
else
    $f' \leftarrow \mathrm{N}(f)$        ▷ $f', g', F', G' \in \mathbb{Z}[x]/(x^{n/2}+1)$
    $g' \leftarrow \mathrm{N}(g)$        ▷ N as defined in either (3.25) or (3.26)
    $(F', G') \leftarrow \mathsf{NTRUSolve}_{n/2,q}(f', g')$        ▷ Recursive call
    $F \leftarrow F'(x^2)g(-x)$        ▷ $F, G \in \mathbb{Z}[x]/(x^n+1)$
    $G \leftarrow G'(x^2)f(-x)$
    $\mathsf{Reduce}(f, g, F, G)$        ▷ $(F, G)$ is reduced with respect to $(f, g)$
return $(F, G)$
```

NTRUSolve uses Reduce (algorithm 7) as a subroutine to reduce the size of the solutions F, G. The principle of Reduce is a simple generalization of textbook vectors' reduction. Given vectors $\mathbf{u}, \mathbf{v} \in \mathbb{Z}^k$, reducing $\mathbf{u}$ with respect to $\mathbf{v}$ is done by simply performing $\mathbf{u} \leftarrow \mathbf{u} - \left\lfloor \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{v}, \mathbf{v} \rangle} \right\rceil \mathbf{v}$. Reduce does the same by replacing $\mathbb{Z}^k$ by $(\mathbb{Z}[x]/(\phi))^2$, $\mathbf{u}$ by (F, G) and $\mathbf{v}$ by (f, g). A detailed explanation of the mathematical and algorithmic principles underlying NTRUSolve can be found in [PP19].

Algorithm 7 $\mathsf{Reduce}(f, g, F, G)$

```
Require: Polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$
Ensure: $(F, G)$ is reduced with respect to $(f, g)$
do
    $k \leftarrow \left\lfloor \frac{Ff^\star + Gg^\star}{ff^\star + gg^\star} \right\rceil$        ▷ $\frac{Ff^\star + Gg^\star}{ff^\star + gg^\star} \in \mathbb{Q}[x]/(\phi)$ and $k \in \mathbb{Z}[x]/(\phi)$
    $F \leftarrow F - kf$
    $G \leftarrow G - kg$
while $k \neq 0$        ▷ Multiple iterations may be needed, e.g. if $k$ is computed in small precision.
```

3.8.3 Computing a Falcon Tree

The second step of the key generation consists of preprocessing the polynomials f, g, F, G into an adequate secret key format. The secret key is of the form $\mathsf{sk} = (\hat{\mathbf{B}}, \mathsf{T})$, where:

- $\hat{\mathbf{B}} = \left[\begin{array}{c|c} \mathsf{FFT}(g) & -\mathsf{FFT}(f) \\ \hline \mathsf{FFT}(G) & -\mathsf{FFT}(F) \end{array}\right]$
- T is a Falcon tree computed in two steps:
  1. First, a tree T is computed from $\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^\star$, called an *LDL tree*. This is specified in ffLDL* (algorithm 9). At this point, T is a Falcon tree but it is not normalized.
  2. Second, T is normalized with respect to a standard deviation σ. It is described in steps 6-7 of Keygen (algorithm 4).

  For efficiency reasons, polynomials manipulated in LDL* (algorithm 8) and ffLDL* (algorithm 9) always remain in FFT representation.

At a high level, the method for computing the LDL tree at step 1 (before normalization) is simple:

1. We compute the LDL decomposition of $\mathbf{G}$: we write $\mathbf{G} = \mathbf{L} \times \mathbf{D} \times \mathbf{L}^\star$, with $\mathbf{L}$ a lower triangular matrix with 1's on the diagonal and $\mathbf{D}$ a diagonal matrix. See LDL* (algorithm 8).

   We store $\mathbf{L}$ in T.value, which is the value of the root of T. Since $\mathbf{L}$ is of the form $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right]$, we only need to store $L_{10} \in \mathbb{Q}[x]/(\phi)$.

2. We then use the splitting operator to "break" each diagonal element of $\mathbf{D}$ into a matrix of smaller elements. More precisely, for a diagonal element $d \in \mathbb{Q}[x]/(x^n + 1)$, we consider the associated endomorphism $\psi_d : z \in \mathbb{Q}[x]/(x^n + 1) \mapsto dz$ and write its transformation matrix over the smaller ring $\mathbb{Q}[x]/(x^{n/2} + 1)$. Following the argument of Section 3.6.1, the transformation matrix of ψ_d can be written as

$$\left[\begin{array}{c|c} d_0 & d_1 \\ \hline x d_1 & d_0 \end{array}\right] \left(= \left[\begin{array}{c|c} d_0 & d_1 \\ \hline d_1^\star & d_0 \end{array}\right] \right)^{1}. \tag{3.30}$$

   For each diagonal element broken into a self-adjoint matrix $\mathbf{G}_i$ over a smaller ring, we recursively compute its LDL tree as in step 1 and store the result in the left or right child of T (which we denote T.leftchild and T.rightchild respectively).

   We continue the recursion until we end up with coefficients in the ring $\mathbb{Q}$.

An implementation of this "LDL tree" strategy is given in ffLDL* (algorithm 9). Note that in Falcon, the input of ffLDL* is always a matrix of dimension 2×2, which greatly simplifies the implementation of its subroutine LDL* (algorithm 8).

[1] The equality in parentheses is true if and only if d is self-adjoint, i.e. $d^\star = d$. This is the case in ffLDL* (algorithm 9).

Algorithm 8 $\mathsf{LDL}^\star(\mathbf{G})$

```
Require: A full-rank self-adjoint matrix $\mathbf{G} = (G_{ij}) \in \mathsf{FFT}(\mathbb{Q}[x]/(\phi))^{2\times 2}$
Ensure: The LDL$^\star$ decomposition $\mathbf{G} = \mathbf{LDL}^\star$ over $\mathsf{FFT}(\mathbb{Q}[x]/(\phi))$
Format: All polynomials are in FFT representation.
$D_{00} \leftarrow G_{00}$
$L_{10} \leftarrow G_{10}/G_{00}$
$D_{11} \leftarrow G_{11} - L_{10} \odot L_{10}^\star \odot G_{00}$
$\mathbf{L} \leftarrow \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} \leftarrow \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
return $(\mathbf{L}, \mathbf{D})$
```

Algorithm 9 $\mathsf{ffLDL}^\star(\mathbf{G})$

```
Require: A full-rank Gram matrix $\mathbf{G} \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^n+1)\right)^{2\times 2}$
Ensure: A binary tree T
Format: All polynomials are in FFT representation.
$(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^\star(\mathbf{G})$        ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
T.value $\leftarrow L_{10}$
if $(n = 2)$ then
    T.leftchild $\leftarrow D_{00}$
    T.rightchild $\leftarrow D_{11}$
    return T
else
    $d_{00}, d_{01} \leftarrow \mathsf{splitfft}(D_{00})$        ▷ $d_{ij} \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)$
    $d_{10}, d_{11} \leftarrow \mathsf{splitfft}(D_{11})$
    $\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline d_{01}^\star & d_{00} \end{array}\right], \mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline d_{11}^\star & d_{10} \end{array}\right]$        ▷ Since $D_{00}, D_{11}$ are self-adjoint, (3.30) applies
    T.leftchild $\leftarrow \mathsf{ffLDL}^\star(\mathbf{G}_0)$        ▷ Recursive calls
    T.rightchild $\leftarrow \mathsf{ffLDL}^\star(\mathbf{G}_1)$
    return T
```

3.9 Signature Generation

3.9.1 Overview

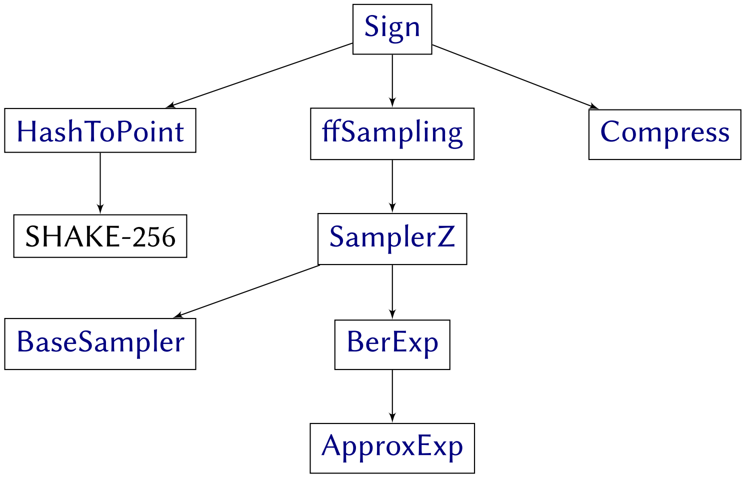

Figure 3.4: Flowchart of the signature

At a high level, the principle of the signature generation algorithm is simple: it first computes a hash value $c \in \mathbb{Z}_q[x]/(\phi)$ from the message m and a salt r, and it then uses its knowledge of the secret key f, g, F, G to compute two short values s_1, s_2 such that $s_1 + s_2 h = c \bmod q$.

A naive way to find such short values (s_1, s_2) would be to compute $\mathbf{t} \leftarrow (c, 0) \cdot \mathbf{B}^{-1}$, round it coefficient-wise to a vector $\mathbf{z} = \lfloor \mathbf{t} \rceil$ and output $(s_1, s_2) \leftarrow (\mathbf{t} - \mathbf{z})\mathbf{B}$; it fulfils all the requirements to be a legitimate signature, but this method is known to be insecure and to leak the private key.

The proper way to generate (s_1, s_2) without leaking the private key is to use a trapdoor sampler (see Section 2.4 for a brief reminder on trapdoor samplers). In Falcon, we use a trapdoor sampler called fast Fourier sampling. The computation of the falcon tree T by ffLDL* (algorithm 9) during the key pair generation is the initialization step of this trapdoor sampler.

The heart of our signature generation, ffSampling (algorithm 11) applies a randomized rounding (according to a discrete Gaussian distribution) on the coefficients of $\mathbf{t}$. But it does so in an adaptive manner, using the information stored in the Falcon tree T.

At a high level, our fast Fourier sampling algorithm can be seen as a recursive variant of Klein's well known trapdoor sampler (also known as the GPV sampler). Klein's sampler uses a matrix $\mathbf{L}$ (and the norm of Gram-Schmidt vectors) as a trapdoor, whereas ours uses a tree of such matrices (or rather, a tree of their non-trivial elements). Given $\mathbf{t} = (t_0, t_1) \in \mathbb{Q}[x]/(\phi))^2$, our algorithm first splits t_1 using the splitting operator, recursively applies itself to it (using the right child T.rightchild of T), and uses the merging operator to lift the solution to the ring $\mathbb{Z}[x]/(\phi)$; it then applies itself again recursively with t_0. Note that the recursions cannot be done in parallel: the second recursion takes into account the result of the first recursion, and this is done using information contained in T.value.

The most delicate part of our signature algorithm is the fast Fourier sampling described in ffSampling, because it makes use of the Falcon tree and of discrete Gaussians over $\mathbb{Z}$. The rest of the algorithm, including the compression of the signature, is rather straightforward to implement.

Formally, given a private key sk and a message m, the signer uses sk to sign m as follows:

1. A random salt r is generated uniformly in $\{0,1\}^{320}$. The concatenated string $(\mathsf{r}\|\mathsf{m})$ is then hashed to a point $c \in \mathbb{Z}_q[x]/(\phi)$ as specified by HashToPoint (algorithm 3).
2. A (not necessarily short) preimage $\mathbf{t}$ of c is computed, and is then given as input to the fast Fourier sampling algorithm, which outputs two short polynomials $s_1, s_2 \in \mathbb{Z}[x]/(\phi)$ (in FFT representation) such that $s_1 + s_2 h = c \bmod q$, as specified by ffSampling (algorithm 11).
3. s_2 is encoded (compressed) to a bitstring s as specified in Compress (algorithm 17).
4. The signature consists of the pair (r, s).

```
Algorithm 10 Sign (m, sk, $\lfloor\beta^2\rfloor$)
Require: A message m, a secret key sk, a bound $\lfloor\beta^2\rfloor$
Ensure: A signature sig of m
r ← $\{0,1\}^{320}$ uniformly
$c$ ← HashToPoint(r‖m, $q$, $n$)
$\mathbf{t}$ ← $\left(-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f)\right)$    ▷ $\mathbf{t} = (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
do
    do
        $\mathbf{z}$ ← ffSampling$_n$($\mathbf{t}$, T)
        s = $(\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$    ▷ At this point, s follows a Gaussian distribution: $\mathsf{s} \sim D_{(c,0)+\Lambda(\mathbf{B}),\sigma,0}$
    while $\|\mathsf{s}\|^2 > \lfloor\beta^2\rfloor$    ▷ Since s is in FFT representation, one may use (3.8) to compute $\|\mathsf{s}\|^2$
    $(s_1, s_2)$ ← invFFT(s)    ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
    s ← Compress($s_2$, $8 \cdot$ sbytelen $- 328$)    ▷ Remove 1 byte for the header, and 40 bytes for r
while (s = $\perp$)
return sig = (r, s)
```

3.9.2 Fast Fourier Sampling

This section describes our fast Fourier sampler: ffSampling (algorithm 11). We note that we perform all the operations in FFT representation for efficiency reasons, but the whole algorithm could also be executed in coefficient representation instead, at a price of a $O(\log n)$ penalty in speed.

Algorithm 11 ffSampling $_n(\mathbf{t}, \mathsf{T})$

```
Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^n+1)\right)^2$, a Falcon tree T
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^n+1)\right)^2$
Format: All polynomials are in FFT representation.
if $n = 1$ then
    $\sigma' \leftarrow \mathsf{T.value}$        ▷ It is always the case that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
    $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$        ▷ Since $n = 1$, $t_i = \mathsf{invFFT}(t_i) \in \mathbb{Q}$ and $z_i = \mathsf{invFFT}(z_i) \in \mathbb{Z}$
    $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
    return $\mathbf{z} = (z_0, z_1)$
$(\ell, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
$\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$        ▷ $\mathbf{t}_0, \mathbf{t}_1 \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)^2$
$\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$        ▷ First recursive call to ffSampling $_{n/2}$
$z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$        ▷ $\mathbf{z}_0, \mathbf{z}_1 \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^{n/2}+1)\right)^2$
$t_0' \leftarrow t_0 + (t_1 - z_1) \odot \ell$
$\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t_0')$
$\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$        ▷ Second recursive call to ffSampling $_{n/2}$
$z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$
```

3.9.3 Sampler over the Integers

Let $1 \leq \sigma_{\min} < \sigma_{\max}$. This section shows how to sample securely Gaussian samples $z \sim D_{\mathbb{Z},\sigma',\mu}$ for any $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$ and $\mu \in \mathcal{R}$. This is done by SamplerZ (algorithm 15), which calls BaseSampler (algorithm 12) and BerExp (algorithm 14) as subroutines. We use the notations (>>) and (&) to denote the bitwise right-shift and AND, respectively. We also introduce the notations $[\![\cdot]\!]$ and UniformBits:

$$\text{For any logical proposition } P, \quad [\![P]\!] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{if } P \text{ is false} \end{cases} \tag{3.31}$$

Note that $[\![\cdot]\!]$ needs to be realized in constant time for our algorithms to be resistant against timing attacks.

$$\forall k \in \mathbb{Z}^+, \quad \mathsf{UniformBits}(k) \text{ samples } z \text{ uniformly in } \{0, 1, ..., 2^k - 1\}. \tag{3.32}$$

BaseSampler. Let pdt be as in Table 3.1. Our first procedure is BaseSampler (algorithm 12). It samples an integer $z_0 \in \mathbb{Z}^+$ according to the distribution χ of support $\{0, \ldots, 18\}$ uniquely defined as:

$$\forall i \in \{0, \ldots, 18\}, \qquad \chi(i) = 2^{-72} \cdot \mathtt{pdt}[i] \tag{3.33}$$

The distribution χ is extremely close to the "half-Gaussian" $D_{\mathbb{Z}^+,\sigma_{\max}}$ in the sense that $R_{513}(\chi \| D_{\mathbb{Z}^+,\sigma_{\max}}) \leq 1 + 2^{-78}$, where R_* is the Rényi divergence. For completeness, Table 3.1 provides the values of:

- the (scaled) probability distribution table $\texttt{pdt}[i]$;
- the (scaled) cumulative distribution table $\texttt{cdt}[i] = \sum_{j \leq i} \texttt{pdt}[j]$;
- the (scaled) reverse cumulative distribution table $\texttt{RCDT}[i] = \sum_{j > i} \texttt{pdt}[j] = 2^{72} - \texttt{cdt}[i]$.

Table 3.1: Values of the {probability/cumulative/reverse cumulative} distribution table for the distribution χ, scaled by a factor 2^{72}.

i	pdt[i]	cdt[i]	RCDT[i]
0	1 697 680 241 746 640 300 030	1 697 680 241 746 640 300 030	3 024 686 241 123 004 913 666
1	1 459 943 456 642 912 959 616	3 157 623 698 389 553 259 646	1 564 742 784 480 091 954 050
2	928 488 355 018 011 056 515	4 086 112 053 407 564 316 161	636 254 429 462 080 897 535
3	436 693 944 817 054 414 619	4 522 805 998 224 618 730 780	199 560 484 645 026 482 916
4	151 893 140 790 369 201 013	4 674 699 139 014 987 931 793	47 667 343 854 657 281 903
5	39 071 441 848 292 237 840	4 713 770 580 863 280 169 633	8 595 902 006 365 044 063
6	7 432 604 049 020 375 675	4 721 203 184 912 300 545 308	1 163 297 957 344 668 388
7	1 045 641 569 992 574 730	4 722 248 826 482 293 120 038	117 656 387 352 093 658
8	108 788 995 549 429 682	4 722 357 615 477 842 549 720	8 867 391 802 663 976
9	8 370 422 445 201 343	4 722 365 985 900 287 751 063	496 969 357 462 633
10	476 288 472 308 334	4 722 366 462 188 760 059 397	20 680 885 154 299
11	20 042 553 305 308	4 722 366 482 231 313 364 705	638 331 848 991
12	623 729 532 807	4 722 366 482 855 042 897 512	14 602 316 184
13	14 354 889 437	4 722 366 482 869 397 786 949	247 426 747
14	244 322 621	4 722 366 482 869 642 109 570	3 104 126
15	3 075 302	4 722 366 482 869 645 184 872	28 824
16	28 626	4 722 366 482 869 645 213 498	198
17	197	4 722 366 482 869 645 213 695	1
18	1	4 722 366 482 869 645 213 696	0

```
Algorithm 12 BaseSampler()
Require: -
Ensure: An integer $z_0 \in \{0, \ldots, 18\}$ such that $z \sim \chi$    ▷ $\chi$ is uniquely defined by (3.33)
1: $u \leftarrow$ UniformBits$(72)$    ▷ See (3.32)
2: $z_0 \leftarrow 0$
3: for $i = 0, \ldots, 17$ do
4:     $z_0 \leftarrow z_0 + [\![u < \texttt{RCDT}[i]]\!]$    ▷ Note that one should use RCDT, not pdt or cdt
5: return $z_0$
```

BerExp and ApproxExp. BerExp (algorithm 14) and its subroutine ApproxExp (algorithm 13) serve to perform rejection sampling. Let C be the following list of 64-bit numbers (in hexadecimal form):

$C =$ [0x00000004741183A3, 0x00000036548CFC06, 0x0000024FDCBF140A, 0x0000171D939DE045, 0x0000D00CF58F6F84, 0x000680681CF796E3, 0x002D82D8305B0FEA, 0x011111110E066FD0, 0x0555555555070F00, 0x155555555581FF00, 0x400000000002B400, 0x7FFFFFFFFFFF4800, 0x8000000000000000].

Let $f \in \mathbb{R}[x]$ be the polynomial defined as:

$$f(x) = 2^{-63} \cdot \sum_{i=0}^{12} C[i] \cdot x^{12-i}.$$

$f(-x)$ serves as a very good approximation of $\exp(-x)$ over $[0, \ln(2)]$, see [ZSS20]. This is leveraged by ApproxExp (algorithm 13) to compute integral approximations of $2^{63} \cdot ccs \cdot \exp(-x)$ for x in a certain range. Note that the intermediate variables y, z in ApproxExp are always in the range $\{0, ..., 2^{63} - 1\}$, with one exception: if $x = 0$, then at the end of the for loop (lines 4 and 5) we have $y = 2^{63}$. This makes it easy to represent x, y using, for example, the C type `uint64_t`.

Algorithm 13 ApproxExp(x, ccs)

```
Require: Floating-point values $x \in [0, \ln(2)]$ and $ccs \in [0, 1]$
Ensure: An integral approximation of $2^{63} \cdot ccs \cdot \exp(-x)$
1: $C$ = [0x00000004741183A3,0x00000036548CFC06,0x0000024FDCBF140A,0x0000171D939DE045,
   0x0000D00CF58F6F84, 0x000680681CF796E3, 0x002D82D8305B0FEA, 0x011111110E066FD0,
   0x0555555555070F00, 0x155555555581FF00, 0x400000000002B400, 0x7FFFFFFFFFFF4800,
   0x8000000000000000]
2: $y \leftarrow C[0]$                    ▷ $y$ and $z$ remain in $\{0, ..., 2^{63} - 1\}$ the whole algorithm.
3: $z \leftarrow \lfloor 2^{63} \cdot x \rfloor$
4: for $1 = 1, \ldots, 12$ do
5:     $y \leftarrow C[u] - (z \cdot y) >> 63$     ▷ $(z \cdot y)$ fits in 126 bits, but we only need the top 63 bits
6: $z \leftarrow \lfloor 2^{63} \cdot ccs \rfloor$
7: $y \leftarrow (z \cdot y) >> 63$
8: return $y$
```

Given inputs $x, ccs \geq 0$, BerExp (algorithm 14) returns a single bit 1 with probability $\approx ccs \cdot \exp(-x)$.

SamplerZ. Finally, SamplerZ (algorithm 15) use the previous algorithms as subroutines and, given inputs μ, σ' in a certain range, outputs an integer $z \sim D_{\mathbb{Z},\sigma',\mu}$ in an isochronous manner.

Known Answer Tests (KAT). To help the proper implementation of SamplerZ (algorithm 15) and its subroutines, Table 3.2 provides test vectors. Let $\sigma_{\min} = 1.277\,833\,697$ (the value of $\sigma_{\min}$ for Falcon-512). Each line of Table 3.2 provides a tuple $(\mu, \sigma', \texttt{randombytes}, z)$ such that when replacing internal

```
Algorithm 14 BerExp($x, ccs$)
Require: Floating point values $x, ccs \geq 0$
Ensure: A single bit, equal to 1 with probability $\approx ccs \cdot \exp(-x)$
  $s \leftarrow \lfloor x/\ln(2) \rfloor$    ▷ Compute the unique decomposition $x = 2^s \cdot r$, with $(r, s) \in [0, \ln 2) \times \mathbb{Z}^+$
  $r \leftarrow x - s \cdot \ln(2)$
  $s \leftarrow \min(s, 63)$
  $z \leftarrow (2 \cdot \mathsf{ApproxExp}(r, ccs) - 1) >> s$    ▷ $z \approx 2^{64-s} \cdot ccs \cdot \exp(-r) = 2^{64} \cdot ccs \cdot \exp(-x)$
  $i \leftarrow 64$
  do
      $i \leftarrow i - 8$
      $w \leftarrow \mathsf{UniformBits}(8) - ((z >> i)\ \&\ \texttt{0xFF})$
  while $((w = 0)$ and $(i > 0))$    ▷ This loop does not need to be done in constant-time
  return $[\![w < 0]\!]$    ▷ Return 1 with probability $2^{-64} \cdot z \approx ccs \cdot \exp(-x)$
```

```
Algorithm 15 SamplerZ($\mu, \sigma'$)
Require: Floating-point values $\mu, \sigma' \in \mathcal{R}$ such that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Ensure: An integer $z \in \mathbb{Z}$ sampled from a distribution very close to $D_{\mathbb{Z},\mu,\sigma'}$
 1: $r \leftarrow \mu - \lfloor \mu \rfloor$    ▷ $r$ must be in $[0, 1)$
 2: $ccs \leftarrow \sigma_{\min}/\sigma'$    ▷ $ccs$ helps to make the algorithm running time independent of $\sigma'$
 3: while (1) do
 4:     $z_0 \leftarrow \mathsf{BaseSampler}()$
 5:     $b \leftarrow \mathsf{UniformBits}(8)\ \&\ \texttt{0x1}$
 6:     $z \leftarrow b + (2 \cdot b - 1)z_0$
 7:     $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$
 8:     if $(\mathsf{BerExp}(x, ccs) = 1)$ then
 9:         return $z + \lfloor \mu \rfloor$
```

calls to UniformBits() with reading bytes from `randombytes` (acting as a random bytestring):

$$\mathsf{SamplerZ}(\mu, \sigma') \rightarrow z \tag{3.34}$$

For readability, Table 3.2 splits `randombytes` according to each iteration of SamplerZ. As an example, line 1 of Table 3.2 indicates that for $\mu = -91.90471153063714$, $\sigma' = 1.7037990414754918$, `randombytes` = `0fc5442ff043d66e91d1eacac64ea5450a22941edc6c` and $z = -92$, the equation (3.34) is verified when running SamplerZ with randomness `randombytes`. In addition, SamplerZ iterates twice before terminating. More precisely, `randombytes` is used as follows:

$$\underbrace{\texttt{0fc5442ff043d66e91}|\texttt{d1}|\texttt{ea}}_{\text{Iteration 1}}\,|\,\underbrace{\texttt{cac64ea5450a22941e}|\texttt{dc}|\texttt{6c}}_{\text{Iteration 2}}$$

At each iteration, the first 9 random bytes are used by BaseSampler, the next one by line 5 and the last one(s) by BerExp. Note that at each call, BerExp has a probability $\frac{1}{2^8}$ of using more than 1 random byte;

this is rare, but happens. This is illustrated by line 9 of Table 3.2, which contain an example for which one iteration of BerExp uses 2 random bytes.

For further testing, this submission package contains more extensive and detailed test vectors. See:

```
Supporting_Documentation/additional/test-vector-sampler-falcon{512,1024}.txt
```

Table 3.2: Test vectors for SamplerZ ($\sigma_{\min} = 1.277\,833\,697$)

	Center μ	Standard deviation σ'	randombytes	Output z
1	-91.90471153063714	1.7037990414754918	0fc5442ff043d66e91d1ea cac64ea5450a22941edc6c	-92
2	-8.322564895434937	1.7037990414754918	f4da0f8d8444d1a77265c2 ef6f98bbbb4bee7db8d9b3	-8
3	-19.096516109216804	1.7035823083824078	db47f6d7fb9b19f25c36d6 b9334d477a8bc0be68145d	-20
4	-11.335543982423326	1.7035823083824078	ae41b4f5209665c74d00dc c1a8168a7bb516b3190cb4 2c1ded26cd52aed770eca7 dd334e0547bcc3c163ce0b	-12
5	7.9386734193997555	1.6984647769450156	31054166c1012780c603ae 9b833cec73f2f41ca5807c c89c92158834632f9b1555	8
6	-28.990850086867255	1.6984647769450156	737e9d68a50a06dbbc6477	-30
7	-9.071257914091655	1.6980782114808988	a98ddd14bf0bf22061d632	-10
8	-43.88754568839566	1.6980782114808988	3cbf6818a68f7ab9991514	-41
9	-58.17435547946095	1.7010983419195522	6f8633f5bfa5d26848668e 3d5ddd46958e97630410587c	-61
10	-43.58664906684732	1.7010983419195522	272bc6c25f5c5ee53f83c4 3a361fbc7cc91dc783e20a	-46
11	-34.70565203313315	1.7009387219711465	45443c59574c2c3b07e2e1 d9071e6d133dbe32754b0a	-34
12	-44.36009577368896	1.7009387219711465	6ac116ed60c258e2cbaeab 728c4823e6da36e18d08da 5d0cc104e21cc7fd1f5ca8 d9dbb675266c928448059e	-44
13	-21.783037079346236	1.6958406126012802	68163bc1e2cbf3e18e7426	-23
14	-39.68827784633828	1.6958406126012802	d6a1b51d76222a705a0259	-40
15	-18.488607061056847	1.6955259305261838	f0523bfaa8a394bf4ea5c1 0f842366fde286d6a30803	-22
16	-48.39610939101591	1.6955259305261838	87bd87e63374cee62127fc 6931104aab64f136a0485b	-50

3.10 Signature Verification

3.10.1 Overview

The signature verification procedure is much simpler than the key pair generation and the signature generation. Given a public key $\mathsf{pk} = h$, a message m, a signature sig = (r,s) and an acceptance bound $\lfloor \beta^2 \rfloor$, the verifier uses pk to verify that sig is a valid signature for the message m as specified hereinafter:

1. The value r (called "the salt") and the message m are concatenated to a string $(\mathsf{r} \| \mathsf{m})$ which is hashed to a polynomial $c \in \mathbb{Z}_q[x]/(\phi)$ as specified by HashToPoint (algorithm 3).

2. s is decoded (decompressed) to a polynomial $s_2 \in \mathbb{Z}[x]/(\phi)$, see Decompress (algorithm 18).

3. The value $s_1 = c - s_2 h \bmod q$ is computed.

4. If $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$, then the signature is accepted as valid. Otherwise, it is rejected.

3.10.2 Specification

The specification of the signature verification is given in Verify (algorithm 16).

Algorithm 16 Verify (m, sig, pk, $\lfloor \beta^2 \rfloor$)

```
Require: A message m, a signature sig = (r, s), a public key pk = $h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor \beta^2 \rfloor$
Ensure: Accept or reject
$c \leftarrow$ HashToPoint($\mathsf{r}\|\mathsf{m}, q, n$)
$s_2 \leftarrow$ Decompress(s, $8 \cdot$ sbytelen $- 328$)
if ($s_2 = \perp$) then
    reject                                        ▷ Reject invalid encodings
$s_1 \leftarrow c - s_2 h \bmod q$                ▷ $s_1$ should be normalized between $\left\lceil -\frac{q}{2} \right\rceil$ and $\left\lfloor \frac{q}{2} \right\rfloor$
if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ then
    accept
else
    reject                                        ▷ Reject signatures that are too long
```

Computation of s_1 can be performed entirely in $\mathbb{Z}_q[x]/(\phi)$; the resulting values should then be normalized to the $\lceil -q/2 \rceil$ to $\lfloor q/2 \rfloor$ range. In order to avoid computing a square root, the squared norm can be computed, using only integer operations, and then compared to $\lfloor \beta^2 \rfloor$.

3.11 Encoding Formats

3.11.1 Bits and Bytes

A *byte* is a sequence of eight bits (formally, an *octet*). Within a byte, bits are ordered from left to right. A byte has a numerical value, which is obtained by adding the weighted bits; the leftmost bit, also called "top bit" or "most significant", has weight 128; the next bit has weight 64, and so on, until the rightmost bit, which has weight 1.

Some of the encoding formats defined below use sequences of bits. When a sequence of bits is represented as bytes, the following rules apply:

- The first byte will contain the first eight bits of the sequence; the second byte will contain the next eight bits, and so on.

- Within each byte, bits are ordered left-to-right in the same order as they appear in the source bit sequence.

- If the bit sequence length is not a multiple of 8, up to 7 extra padding bits are added at the end of the sequence. The extra padding bits MUST have value zero.

This handling of bits matches widely deployed standard, e.g. bit ordering in the SHA-2 and SHA-3 functions, and `BIT STRING` values in ASN.1.

3.11.2 Compressing Gaussians

In Falcon, the signature of a message essentially consists of a polynomial $s \in \mathbb{Z}_q[x]/(\phi)$ which coefficients are distributed around 0 according to a discrete Gaussian distribution of standard deviation $\sigma \approx 1.55\sqrt{q} \ll q$. A naive encoding of s would require about $\lceil \log_2 q \rceil \cdot \deg(\phi)$ bits, which is far from optimal for communication complexity.

In this section we specify algorithms for compressing and decompressing efficiently polynomials such as s. The description of this compression procedure is simple:

1. For each coefficient s_i, a compressed string str_i is defined as follows:

   (a) The first bit of str_i is the sign of s_i;

   (b) The 7 next bits of str_i are the 7 least significant bits of $|s_i|$, in order of significance, i.e. most to least significant;

   (c) The last bits of str_i are an encoding of the most significant bits of $|s_i|$ using unary coding. If $\lfloor |s_i|/2^7 \rfloor = k$, then its encoding is $\underbrace{0\dots0}_{k \text{ times}}1$, which we also note 0^k1;

2. The compression of s is the concatenated string $\mathsf{str} \leftarrow (\mathsf{str}_0 \| \mathsf{str}_1 \| \dots \| \mathsf{str}_{n-1})$.

3. str is padded with zeroes to a fixed width slen.

This encoding is based on two observations. First, since $s_i \bmod 2^7$ is close to uniform, it is pointless to compress the 7 least significant bits of s_i. Second, if a Huffman table is computed for the most significant bits of $|s_i|$, it results in the unary code we just described. So our unary code is actually a Huffman code for the distribution of the most significant bits of $|s_i|$. A formal description is given in Compress (algorithm 17).

```
Algorithm 17 Compress($s$, slen)
Require: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]$ of degree $< n$, a string bitlength slen
Ensure: A compressed representation str of $s$ of slen bits, or $\perp$
str ← {}                                                  ▷ str is the empty string
for $i$ from 0 to $n-1$ do          ▷ At each step, str ← (str||str$_i$), where str$_i$ encodes $s_i$
    str ← (str||$b$), where $b = 1$ if $s_i < 0$, $b = 0$ otherwise          ▷ Encode the sign of $s_i$
    str ← (str||$b_6 b_5 \dots b_0$), where $b_j = (|s_i| >> j)$ & 0x1    ▷ Encode in binary the low bits of $|s_i|$
    $k \leftarrow |s_i| >> 7$
    str ← (str||$0^k 1$)                              ▷ Encode in unary the high bits of $|s_i|$
if ($|$str$| >$ slen) then
    str ← $\perp$                                          ▷ Abort if str is too long
else
    str ← (str||$0^{\text{slen}-|\text{str}|}$)                          ▷ Pad str to slen bits
return str
```

The corresponding decompression algorithm is given in Decompress (algorithm 18). For any polynomial $s \in \mathbb{Z}[x]$ such that Compress(s, slen) $\neq \perp$, it holds that Decompress(Compress(s, slen), slen) $= s$. We now enforce unique encodings: a polynomial s should have at most one valid encoding str. This is done via three additional checks in Decompress:

1. only accept bitstrings of length slen $= 8 \cdot$ sbytelen $- 328$ (see lines 1 and 2);
2. only accept 000000001 – and not 100000001 – as a valid encoding of the coefficient 0 (see lines 9 and 10);
3. force the last bits of str to be 0 (see lines 12 and 13).

3.11.3 Signatures

A Falcon signature consists of two strings r and s. They may conceptually be encoded separately, because the salt r must be known *before* beginning to hash the message itself, while the s value can be obtained or verified only after the whole message has been processed. In a format that supports streamed processing of long messages, the salt r would normally be encoded before the message, while the s value would appear after the message bytes. However, we here define an encoding that includes both r and s.

The first byte is a header with the following format (bits indicated from most to least significant):

```
0 c c 1 n n n n
```

Algorithm 18 Decompress(str, slen)

```
Ensure: A bitstring $\mathsf{str} = (\mathsf{str}[i])_{i=0,\dots,\mathsf{slen}-1}$, a bitlength $\mathsf{slen}$
Require: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]$, or $\perp$
if $|\mathsf{str}| \neq \mathsf{slen}$ then                                  ▷ Enforce fixed bitlength
    return $\perp$
for $i$ from 0 to $(n-1)$ do
    $s'_i \leftarrow \sum_{j=0}^{6} 2^{6-j} \cdot \mathsf{str}[1+j]$               ▷ We recover the lowest bits of $|s_i|$.
    $k \leftarrow 0$                                                      ▷ We recover the highest bits of $|s_i|$.
    while $\mathsf{str}[8+k] = 0$ do
        $k \leftarrow k+1$
    $s_i \leftarrow (-1)^{\mathsf{str}[0]} \cdot (s'_i + 2^7 k)$                        ▷ We recompute $s_i$.
    if $(s_i = 0)$ and $(\mathsf{str}[0] = 1)$ then                     ▷ Enforce unique encoding if $s_i = 0$
        return $\perp$
    $\mathsf{str} \leftarrow \mathsf{str}[9+k \dots \ell - 1]$                   ▷ We remove the bits of str that encore $s_i$.
if $(\mathsf{str} \neq 0^{|\mathsf{str}|})$ then                                 ▷ Enforce trailing bits at 0
    return $\perp$
return $s = \sum_{i=0}^{n-1} s_i x^i$
```

with these conventions:

- The leftmost bit is 0, and the fourth bit from the left is 1 (in previous versions of Falcon, these bits may had have different values).
- Bits cc are 01 or 10 to specify the encoding method for s. Encoding 01 uses the compression algorithm described in Section 3.11.2; encoding 10 is alternate uncompressed encoding in which each coefficient of s is encoded over a fixed number of bits.
- Bits nnnn encode a value ℓ such that the Falcon degree is $n = 2^\ell$. ℓ must be in the allowed range (1 to 10).

Following the header byte are the nonce string r (40 bytes), then the encoding of s itself.

Signatures are then normally padded with zeros up to the prescribed length (sbytelen). Verifiers may also support unpadded signatures, which do not have a fixed size, but are (on average) slightly shorter than padded signatures. *Partial* padding is not valid: if the signature has padding bytes, then all padding bytes must be zero, and the total padded length must be equal to sbytelen.

When using the alternate uncompressed format (cc is 10 in the header byte), all elements of s are encoded over exactly 12 bits each (signed big-endian encoding, using two's complement for negative integers; the valid range is -2047 to $+2047$, the value -2048 being forbidden)[2]. This uncompressed format yields larger signatures and is meant to support the uncommon situations in which signature values and signed messages are secret: uncompressed signatures can be decoded and encoded with constant-time implementations that do not leak information through timing-based side channels.

[2] In some reduced versions of Falcon, with degree 16 or less, fewer bits may be used. These reduced versions do not offer any security and are used only for research and tests.

3.11.4 Public Keys

A Falcon public key is a polynomial h whose coefficients are considered modulo q. An encoded public key starts with a header byte:

```
0 0 0 0 n n n n
```

with these conventions:

- The four leftmost bits are 0 (in some previous versions of Falcon, the leftmost bit could have been non-zero).
- Bits nnnn encode a value ℓ such that the degree is $n = 2^{\ell}$. ℓ must be in the allowed range (1 to 10).

After the header byte comes the encoding of h: each value (in the 0 to $q - 1$ range) is encoded as a 14-bit sequence (since $q = 12289$, 14 bits per value are used). The encoded values are concatenated into a bit sequence of $14n$ bits, which is then represented as $\lceil 14n/8 \rceil$ bytes.

3.11.5 Private Keys

Private keys use the following header byte:

```
0 1 0 1 n n n n
```

with these conventions:

- The four leftmost bits are `0101`.
- Bits nnnn encode the value ℓ such that the degree is $n = 2^{\ell}$. ℓ must be in the allowed range (1 to 10).

Following the header byte are the encodings of f, g, and F, in that order. Each coordinate is encoded over a fixed number of bits, which depends on the degree:

- Coefficients of f and g use:
  - 8 bits each for degrees 2 to 32;
  - 7 bits each for degrees 64 and 128;
  - 6 bits each for degrees 256 and 512;
  - 5 bits each for degree 1024.
- Coefficients of F use 8 bits each, regardless of the degree.

Of course, small degrees do not offer real security, and are meant only for test and research purposes. In practical situations, the degree should be 512 or 1024.

Each coefficient uses signed encoding, with two's complement for negative values. Moreover, the minimal value is forbidden; e.g. when using degree 512, the valid range for a coefficient of f or g is -31 to $+31$; -32 is not allowed.

The polynomial G is not encoded. It is recomputed when the key is loaded, thanks to the NTRU equation:

$$G = (q + gF)/f \mod \phi \tag{3.35}$$

Since the coefficients of f, g, F and G are small, this computation can be done modulo q as well, using the same techniques as signature verification (e.g. the NTT).

3.11.6 NIST API

The API to be implemented by candidates to the NIST call for post-quantum algorithms mandates a different convention, in which the signed message and the signature are packed into a single aggregate format. In this API, the following encoding is used:

- First two bytes are the "signature length" (big-endian encoding).
- Then follows the nonce r (40 bytes).
- The message data itself appears immediately after the nonce.
- The signature comes last. This signature uses a nonce-less format:
  - Header byte is: `0010nnnn`
  - Encoded s immediately follows, using compressed encoding.

There is no signature padding; the signature has a variable length. The length specified in the first two bytes of the package is the length, in bytes, of the signature, including its header byte, but not including the nonce length.

3.12 A Note on the Key-Recovery Mode

We mentioned in Section 2.7 that Falcon can be implemented in key-recovery mode. While we do not propose this mode as part of the specification, we outline here how this can be done:

- The public key becomes $\mathsf{pk} = H(h)$ for some collision-resistant hash function H;
- The signature becomes $(\mathsf{s}_1, \mathsf{s}_2, \mathsf{r})$, with $\mathsf{s}_i = \mathsf{Compress}(s_i)$;
- The verifier accepts the signatures if and only if:
  - (s_1, s_2) is short;
  - $\mathsf{pk} = H\left(s_2^{-1}\left(\mathsf{HashToPoint}(\mathsf{r}\|\mathsf{m}, q, n) - s_1\right)\right)$

We note that $h = s_2^{-1}\left(\mathsf{HashToPoint}(\mathsf{r}\|\mathsf{m}, q, n) - s_1\right)$, so the verifier can recover h during the verification process, hence the name *key-recovery mode*. We also note that unlike the other modes, this one requires s_2 to be invertible $\mathrm{mod}(\phi, q)$. Finally, the output of H should be 2λ bits long to ensure collision-resistance, but if we assume that the adversary can query at most q_s public keys (similarly to the bound imposed on the number of signatures), perhaps it can be shortened to $\lambda + \log_2 q_s$.

The main impact of this mode is that the public key becomes extremely compact: $|\mathsf{pk}| = 2\lambda$. The signature becomes about twice larger, but the total size $|\mathsf{pk}| + |\mathsf{sig}|$ becomes about 15% shorter. Indeed, we trade h with s_1; the bitsize of s_1 can be reduced by about 35% using Compress, whereas h cannot be compressed efficiently (it is assumed to be computationally indistinguishable from random).

3.13 Recommended Parameters

We specify two sets of parameters that address security levels I and V as defined by NIST [NIS16, Section 4.A.5]. These can be found in Table 3.3. Core-SVP hardness is given for the best known classical (C) and quantum (Q) algorithms.

	Falcon-512	Falcon-1024
Target NIST Level	I	V
Ring degree n	512	1024
Modulus q	12289	
Standard deviation σ	165.736 617 183	168.388 571 447
$\sigma_{\min}$	1.277 833 697	1.298 280 334
$\sigma_{\max}$	1.8205	
Max. signature square norm $\lfloor \beta^2 \rfloor$	34 034 726	70 265 242
Public key bytelength	897	1 793
Signature bytelength sbytelen	666	1 280
Key-recovery: BKZ blocksize B (2.3)	458	936
Core-SVP hardness (C)	133	273
Core-SVP hardness (Q)	121	248
Forgery: BKZ blocksize B (2.4)	411	952
Core-SVP hardness (C)	120	277
Core-SVP hardness (Q)	108	252

Table 3.3: Falcon parameter sets.

Chapter 4

Implementation and Performances

We list here a number of noteworthy points related to implementation.

4.1 Floating-Point

Signature generation, and also part of key pair generation, involve the use of complex numbers. These can be approximated with standard IEEE 754 floating-point numbers ("binary64" format, commonly known as "double precision"). Each such number is encoded over 64 bits, that split into the following elements:

- a sign $s = \pm 1$ (1 bit);
- an exponent e in the -1022 to $+1023$ range (11 bits);
- a mantissa m such that $1 \leq m < 2$ (52 bits).

In general, the represented value is $sm2^e$. The mantissa is encoded as $2^{52}(m - 1)$; it has 53 bits of precision, but its top bit, of value 1 by definition, is omitted in the encoding.

The exponent e uses 11 bits, but its range covers only 2046 values, not 2048. The two extra possible values for that field encode special cases:

- The value zero. IEEE 754 has two zeros, that differ by the sign bit.
- Subnormals: they use the minimum value for the exponent (-1022) but the implicit top bit of the mantissa is 0 instead of 1.
- Infinites (positive and negative).
- Erroneous values, known as NaN (Not a Number).

Apart from zero, Falcon does not exercise these special cases; exponents remain relatively close to zero; no infinite or NaN is obtained.

The C language specification does not guarantee that its `double` type maps to IEEE 754 "binary64" type, only that it provides an exponent range and precision that match at least that IEEE type. Support of subnormals, infinites and NaNs is left as implementation-defined. In practice, most C compilers will provide what the underlying hardware directly implements, and *may* include full IEEE support for the special cases at the price of some non-negligible overhead, e.g. extra tests and supplementary code for subnormals, infinites and NaNs. Common x86 CPU, in 64-bit mode, use SSE2 registers and operations for floating-point, and the hardware already provides complete IEEE 754 support. Other processor types have only a partial support; e.g. many PowerPC cores meant for embedded systems do not handle subnormals (such values are then rounded to zeros). Falcon works properly with such limited floating-point types.

Some processors do not have a FPU at all. These will need to use some emulation using integer operations. As explained above, special cases need not be implemented.

4.2 FFT and NTT

4.2.1 FFT

The Fast Fourier Transform for a polynomial f computes $f(\zeta)$ for all roots ζ of ϕ (over $\mathbb{C}$). It is normally expressed recursively. If $\phi = x^n + 1$, and $f = f_0(x^2) + xf_1(x^2)$, then the following holds for any root ζ of ϕ:

$$\begin{array}{rcl} f(\zeta) & = & f_0(\zeta^2) + \zeta f_1(\zeta^2) \\ f(-\zeta) & = & f_0(\zeta^2) - \zeta f_1(\zeta^2) \end{array} \tag{4.1}$$

ζ^2 is a root of $x^{n/2} + 1$: thus, the FFT of f is easily computed, with $n/2$ multiplications and n additions or subtractions, from the FFT of f_0 and f_1, both being polynomials of degree less than $n/2$, and taken modulo $\phi' = x^{n/2} + 1$. This leads to a recursive algorithm of cost $O(n \log n)$ operations.

The FFT can be implemented iteratively, with minimal data movement and no extra buffer: in the equations above, the computed $f(\zeta)$ and $f(-\zeta)$ will replace $f_0(\zeta^2)$ and $f_1(\zeta^2)$. This leads to an implementation known as "bit reversal", due to the resulting ordering of the $f(\zeta)$: if $\zeta_j = e^{i(\pi/2n)(2j+1)}$, then $f(\zeta_j)$ ends up in slot rev(j), where rev is the bit-reversal function over $\log_2 n$ bits (it encodes its input in binary with left-to-right order, then reinterprets it back as an integer in right-to-left order).

In the iterative, bit-reversed FFT, the first step is computing the FFT of $n/2$ sub-polynomials of degree 1, corresponding to source index pairs $(0, n/2)$, $(1, n/2 + 1)$, and so on.

Some noteworthy points for FFT implementation in Falcon are the following:

- The FFT uses a table of pre-computed roots $\zeta_j = e^{i(\pi/2n)(2j+1)}$. The inverse FFT nominally requires, similarly, a table of inverses of these roots. However, $\zeta_j^{-1} = \overline{\zeta_j}$; thus, inverses can be efficiently recomputed by negating the imaginary part.
- ϕ has n distinct roots in $\mathbb{C}$, leading to n values $f(\zeta_j)$, each being a complex number, with a real and an imaginary part. Storage space requirements are then $2n$ floating-point numbers. However, if f is real, then, for every root ζ of ϕ, $\overline{\zeta}$ is also a root of ϕ, and $\overline{f(\zeta)} = f(\overline{\zeta})$. Thus, the FFT representation is redundant, and half of the values can be omitted, reducing storage space requirements

to $n/2$ complex numbers, hence n floating-point values.

- The Hermitian adjoint of f is obtained in FFT representation by simply computing the conjugate of each $f(\zeta)$, i.e. negating the imaginary part. This means that when a polynomial is equal to its Hermitian adjoint (e.g. $ff^\star + gg^\star$), then its FFT representation contains only real values. If then multiplying or dividing by such a polynomial, the unnecessary multiplications by 0 can be optimized away.
- The C language (since 1999) offers direct support for complex numbers. However, it may be convenient to keep the real and imaginary parts separate, for values in FFT representation. If the real and imaginary parts are kept at indexes k and $k + n/2$, respectively, then some performance benefits are obtained:
  - The first step of FFT becomes free. That step involves gathering pairs of coefficients at indexes $(k, k + n/2)$, and assembling them with a root of $x^2 + 1$, which is i. The source coefficients are still real numbers, thus $(f_0, f_{n/2})$ yields $f_0 + if_{n/2}$, whose real and imaginary parts must be stored at indexes 0 and $n/2$ respectively, where they already are. The whole loop disappears.
  - When a polynomial is equal to its Hermitian adjoint, all its values in FFT representation are real. The imaginary parts are all null, and they represent the second half of the array. Storage requirements are then halved, without requiring any special reordering or move of values.

4.2.2 NTT

The *Number Theoretic Transform* is the analog of the FFT, in the finite field $\mathbb{Z}_p$ of integers modulo a prime p. $\phi = x^n + 1$ will have roots in $\mathbb{Z}_p$ if and only if $p = 1 \bmod 2n$. The NTT, for an input polynomial f whose coefficients are integers modulo p, computes $f(\omega) \bmod p$ for all roots ω of ϕ in $\mathbb{Z}_p$.

Signature verification is naturally implemented modulo q; that modulus is chosen precisely to be NTT-friendly:

$$q = 12289 = 1 + 12 \cdot 2048.$$

Computations modulo q can be implemented with pure 32-bit integer arithmetics, avoiding divisions and branches, both being relatively expensive. For instance, modular addition of x and y may use this function:

```
static inline uint32_t
mq_add(uint32_t x, uint32_t y, uint32_t q)
{
    uint32_t d;

    d = x + y - q;
    return d + (q & -(d >> 31));
}
```

This code snippet uses the fact that C guarantees operations on uint32_t to be performed modulo 2^{32}; since operands fits on 15 bits, the top bit of the intermediate value d will be 1 if and only if the subtraction of q yields a negative value.

For multiplications, Montgomery multiplication is effective:

```
static inline uint32_t
mq_montymul(uint32_t x, uint32_t y, uint32_t q, uint32_t q0i)
{
    uint32_t z, w;

    z = x * y;
    w = ((z * q0i) & 0xFFFF) * q;
    z = ((z + w) >> 16) - q;
    return z + (q & -(z >> 31));
}
```

The parameter q0i contains $1/q \bmod 2^{16}$, a value which can be hardcoded since q is also known at compile-time. Montgomery multiplication, given x and y, computes $xy/(2^{16}) \bmod q$. The intermediate value z can be shown to be less than $2q$, which is why a single conditional subtraction is sufficient.

Modular divisions are not needed for signature verification, but they are handy for computing the public key h from f anf g, as part of key pair generation. Inversion of x modulo q can be computed in a number of ways; exponentation is straightforward: $1/x = x^{q-2} \bmod q$. For 12289, minimal addition chains on the exponent yield the result in 18 Montgomery multiplications (assuming input and output are in Montgomery representation).

Key pair generation may also use the NTT, modulo a number of small primes p_i, and the branchless implementation techniques described above. The choice of the size of such small moduli p_i depends on the abilities of the current architecture. The Falcon reference implementation, that aims at portability, uses moduli p_i which are slightly below 2^{31}, a choice which has some nice properties:

- Modular reductions after additions or subtractions can be computed with pure 32-bit unsigned arithmetics.
- Values may fit in the *signed* int32_t type.
- When doing Montgomery multiplications, intermediate values are less than 2^{63} and thus can be managed with the standard type uint64_t.

On a 64-bit machine with $64 \times 64 \rightarrow 128$ multiplications, 63-bit moduli would be a nice choice.

4.3 LDL Tree

From the private key properly said (the f, g, F and G short polynomials), signature generation involves two main steps: building the LDL tree, and then using it to sample a short vector. The LDL tree depends only on the private key, not the data to be signed, and is reusable for an arbitrary number of signatures; thus, it can be considered part of the private key. However, that tree is rather bulky (about 90 kB for $n = 1024$), and will use floating-point values, making its serialization complex to define in all generality. Therefore, the Falcon reference code rebuilds the LDL tree dynamically when the private key is loaded; its API still allows a built tree to be applied to many signature generation instances.

It would be possible to regenerate the LDL tree on the go, for a computational overhead similar to that of sampling the short vector itself; this would save space, since at no point would the full tree need to be present in RAM, only a path from the tree root to the current leaf. For degree n, a saved path would amount to about $2n$ floating-point values, i.e. roughly 16 kB. On the other hand, computational cost per signature would double.

Both LDL tree construction and sampling involve operations on polynomials, including multiplications (and divisions). It is highly recommended to use FFT representation, since multiplication and division of two degree-n polynomials in FFT representation requires only n elementary operations. The LDL tree is thus best kept in FFT.

4.4 Key Pair Generation

4.4.1 Gaussian Sampling

The f and g polynomials must be generated with an appropriate distribution. It is sufficient to generate each coefficient independently, with a Gaussian distribution centered on 0; values are easily tabulated.

4.4.2 Filtering

As per the Falcon specification, once f and g have been generated, some tests must be applied to determine their appropriateness:

- $(g, -f)$ and its orthogonalized version must be short enough.

- f must be invertible modulo ϕ and q; this is necessary in order to be able to compute the public key $h = g/f \bmod \phi \bmod q$. In practice, the NTT is used on f: all the resulting coefficients of f in NTT representation must be distinct from zero. Computing h is then straightforward.

- The Falcon reference implementation furthermore requires that $\mathrm{Res}(f, \phi)$ and $\mathrm{Res}(g, \phi)$ be both odd. If they are both even, the NTRU equation does not have a solution, but our implementation cannot tolerate that one is even and the other is odd. Computing the resultant modulo 2 is inexpensive; here, this is equal to the sum of the coefficients modulo 2.

If any of these tests fails, new (f, g) must be generated.

4.4.3 Solving The NTRU Equation

Solving the NTRU equation is formally a recursive process. At each depth:

1. Input polynomials f and g are received as input; they are modulo $\phi = x^n + 1$ for a given n.
2. New values $f' = \mathrm{N}(f)$ and $g' = \mathrm{N}(g)$ are computed; they live modulo $\phi' = x^{n/2} + 1$, i.e. half the degree of ϕ. However, their coefficients are typically twice longer than those of f and g.
3. The solver is invoked recursively over f' and g', and yields a solution (F', G') such that
$$f'G' - g'F' = q.$$
4. Unreduced values (F, G) are generated, as:
$$\begin{array}{rcll} F &=& F'(x^2)g'(x^2)/g(x) & \mod \phi \\ G &=& G'(x^2)f'(x^2)/f(x) & \mod \phi \end{array} \tag{4.2}$$
F and G are modulo ϕ (of degree n), and their coefficients have a size which is about three times that of the coefficients of inputs f and g.
5. Babai's nearest plane algorithm is applied, to bring coefficients of F and G down to that of the coefficients of f and g.

RNS and NTT

The operations implied in the recursion are much easier when operating on the NTT representation of polynomials. Indeed, if working modulo p, and ω is a root of $x^n + 1$ modulo p, then:

$$\begin{array}{rcl} f'(\omega^2) &=& N(f)(\omega^2) = f(\omega)f(-\omega) \\ F(\omega) &=& F'(\omega^2)g(-\omega) \end{array} \tag{4.3}$$

Therefore, the NTT representations of f' and g' can be easily computed from the NTT representations of f and g; and, similarly, the NTT representation of F and G (unreduced) are as easily obtained from the NTT representations of F' and G'.

This naturally leads to the use of a Residue Number System (RNS), in which a value x is encoded as a sequence of values $x_j = x \bmod p_j$ for a number of distinct small primes p_j. In the Falcon reference implementation, the p_j are chosen such that $p_j < 2^{31}$ (to make computations easy with pure integer arithmetics) and $p_j = 1 \bmod 2048$ (to allow the NTT to be applied).

Conversion from the RNS encoding to a plain integer in base 2^{31} is a straightforward application of the Chinese Remainder Theorem; if done prime by prime, then the only required big-integer primitives will be additions, subtractions, and multiplication by a one-word value. In general, coefficient values are signed, while the CRT yields values ranging from 0 to $\prod p_j - 1$; normalisation is applied by assuming that the final value is substantially smaller, in absolute value, than the product of the used primes p_j.

Coefficient Sizes

Key pair generation has the unique feature that it is allowed occasional failures: it may reject some cases which are nominally valid, but do not match some assumptions. This does not induce any weakness or substantial performance degradation, as long as such rejections are rare enough not to substantially reduce the space of generated private keys.

In that sense, it is convenient to use *a priori* estimates of coefficient sizes, to perform the relevant memory allocations and decide how many small primes p_j are required for the RNS representation of any integer at any point of the algorithm. The following maximum sizes of coefficients, in bits, have been measured over thousands of random key pairs, at various depths of the recursion:

depth	**max f, g**	**std. dev.**	**max F, G**	**std. dev.**
10	6307.52	24.48	6319.66	24.51
9	3138.35	12.25	9403.29	27.55
8	1576.87	7.49	4703.30	14.77
7	794.17	4.98	2361.84	9.31
6	400.67	3.10	1188.68	6.04
5	202.22	1.87	599.81	3.87
4	101.62	1.02	303.49	2.38
3	50.37	0.53	153.65	1.39
2	24.07	0.25	78.20	0.73
1	10.99	0.08	39.82	0.41
0	4.00	0.00	19.61	0.49

These sizes are expressed in bits; for each depth, each category of value, and each key pair, the maximum size of the absolute value is gathered. The array above lists the observed averages and standard deviations for these values.

A Falcon key pair generator may thus simply assume that values fit correspondingly dimensioned buffers, e.g. by using the measured average added to, say, six times the standard deviation. This would ensure that values almost always fit. A final test at the end of the process, to verify that the computed F and G match the NTRU equation, is sufficient to detect failures.

Note that for depth 10, the maximum size of F and G is the one resulting from the extended GCD, thus similar to that of f and g.

Binary GCD

At the deepest recursion level, inputs f and g are plain integers (the modulus is $\phi = x + 1$); a solution can be computed directly with the Extended Euclidean Algorithm, or a variant thereof. The Falcon reference implementation uses the binary GCD. This algorithm can be expressed in the following way:

- Values a, b, u_0, u_1, v_0 and v_1 are initialized and maintained with the following invariants:

$$\begin{array}{rcl} a & = & fu_0 - gv_0 \\ b & = & fu_1 - gv_1 \end{array} \tag{4.4}$$

  Initial values are:

$$\begin{array}{rcl} a & = & f \\ u_0 & = & 1 \\ v_0 & = & 0 \\ b & = & g \\ u_1 & = & g \\ v_1 & = & f-1 \end{array} \tag{4.5}$$

- At each step, a or b is reduced: if a and/or b is even, then it is divided by 2; otherwise, if both values are odd, then the smaller of the two is subtracted from the larger, and the result, now even, is divided by 2. Corresponding operations are applied on u_0, v_0, u_1 and v_1 to maintain the invariants. Note that computations on u_0 and u_1 are done modulo g, while computations on v_0 and v_1 are done modulo f.
- Algorithm stops when $a = b$, at which point the common value is the GCD of f and g.

If the GCD is 1, then a solution $(F, G) = (qv_0, qu_0)$ can be returned. Otherwise, the Falcon reference implementation rejects the (f, g) pair. Note that the (rare) case of a GCD equal to q itself is also rejected; as noted above, this does not induce any particular algorithm weakness.

The description above is a bit-by-bit algorithm. However, it can be seen that most of the decisions are taken only on the low bits and high bits of a and b. It is thus possible to group updates of a, b and other values by groups of, say, 31 bits, yielding much better performance.

Iterative Version

Each recursion depth involves receiving (f, g) from the upper level, and saving them for the duration of the recursive call. Since degrees are halved and coefficients double in size at each level, the storage space for such an (f, g) pair is mostly constant, around 13000 bits per depth. For $n = 1024$, depth goes to 10, inducing a space requirement of at least 130000 bits, or 16 kB, just for that storage. In order to reduce space requirements, the Falcon reference implementation recomputes (f, g) dynamically from start when needed. Measures indicate a relatively low CPU overhead (about 15%).

A side-effect of this recomputation is that each recursion level has nothing to save. The algorithm thus becomes iterative.

Babai's Reduction

When candidates F and G have been assembled, they must be reduced against the current f and g. Reduction is performed as successive approximate reductions, that are computed with the FFT:

- Coefficients of f, g, F and G are converted to floating-point values, yielding $\dot{f}$, $\dot{g}$, $\dot{F}$ and $\dot{G}$. Scaling is applied so that the maximum coefficient of $\dot{F}$ and $\dot{G}$ is about 2^{30} times the maximum coefficient of $\dot{f}$ and $\dot{g}$; scaling also ensures that all values fit in the exponent range of floating-point values.
- An integer polynomial k is computed as:

  $$k = \left\lfloor \frac{\dot{F}\dot{f}^\star + \dot{G}\dot{g}^\star}{\dot{f}\dot{f}^\star + \dot{g}\dot{g}^\star} \right\rceil \tag{4.6}$$

  This computation is typically performed in FFT representation, where multiplication and division of polynomials are easy. Rounding to integers, though, must be done in coefficient representation.
- kf and kg are subtracted from F and G, respectively. Note that this operation must be exact, and is performed on the integer values, not the floating-point approximations. At high degree (i.e. low recursion depth), RNS and NTT are used: the more efficient multiplications in NTT offset the extra cost for converting values to RNS and back.

This process reduces the maximum sizes of coefficients of F and G by about 30 bits at each iteration; it is applied repeatedly as long as it works, i.e. the maximum size is indeed reduced. A failure is reported if the final maximum size of F and G coefficients does not fit the target size, i.e. the size of the buffers allocated for these values.

4.5 Performances

The Falcon reference implementation achieves the following performance on an Intel® Core® i5-8259U CPU ("Coffee Lake" core, clocked at 2.3 GHz):

degree	keygen (ms)	keygen (RAM)	sign/s	vrfy/s	pub length	sig length
512	8.64	14336	5948.1	27933.0	897	666
1024	27.45	28672	2913.0	13650.0	1793	1 280

The following notes apply:

- For this test, in order to obtain stable benchmarks, CPU frequency scaling ("TurboBoost") has been disabled. This CPU can nominally scale its frequency up to 3.9 GHz (for short durations), for a corresponding increase in performance. In particular, since all operations at degree 512 fit in L1 cache (both code and data), one may expect performance to be proportional to frequency, up to about 10000 signatures per second at the maximum frequency. The figures shown above are for *sustained* workloads in which signatures are repeatedly computed over prolonged periods of time.
- RAM usage for key pair generation is expressed in bytes. It includes temporary buffers for all intermediate values, including the floating-point polynomials used for Babai's reduction.
- Public key length and average signature length are expressed in bytes. The size of public keys includes a one-byte header that identifies the degree and modulus. For signatures, compression and padding is used, thus leading to a fixed signature length.

- Signature generation time does not include the LDL tree building, which is done when the private key is loaded. These figures thus correspond to batch usage, when many values must be signed with a given key. This matches, for instance, the use case of a busy TLS server. If, in a specific scenario, keys are used only once, then the LDL tree building cost must be added to each signature attempt; this almost doubles the signature cost, but reduces RAM usage.

- The implementation used for this benchmark is fully constant-time. It uses AVX2 and FMA opcodes for improved performance. Compiler is Clang 10.0, with optimization flags: `-O3 -march=skylake`.

Bibliography

[ABD16] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 153–178, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany. 16

[ADH$^+$19] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 717–746, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. 16

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343, Austin, TX, USA, August 10–12, 2016. USENIX Association. 15

[Bab85] L Babai. On lovasz' lattice reduction and the nearest lattice point problem. In *Proceedings on STACS 85 2Nd Annual Symposium on Theoretical Aspects of Computer Science*, New York, NY, USA, 1985. Springer-Verlag New York, Inc. 10

[Bab86] László Babai. On lovasz' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1), 1986. 10

[BCD$^+$16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1006–1018, Vienna, Austria, October 24–28, 2016. ACM Press. 12

[BDF$^+$11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany. 9, 11, 19

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24, Arlington, VA, USA, January 10–12, 2016. ACM-SIAM. 15, 16

[BLL$^+$15] Shi Bai, Adeline Langlois, Tancrède Lepoint, Damien Stehlé, and Ron Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance.

In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 3–24, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany. 32

[BM19] Alexandra Boldyreva and Daniele Micciancio, editors. *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. 64, 66

[CD20] André Chailloux and Thomas Debris-Alazard. Tight and optimal reductions for signatures based on average trapdoor preimage sampleable functions and applications to code-based signatures. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 453–479, Edinburgh, UK, May 4–7, 2020. Springer, Heidelberg, Germany. 19

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-SVP. In Coron and Nielsen [CN17], pages 324–348. 16

[CGM19] Yilei Chen, Nicholas Genise, and Pratyay Mukherjee. Approximate trapdoors for lattices and smaller hash-and-sign signatures. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 3–32, Kobe, Japan, December 8–12, 2019. Springer, Heidelberg, Germany. 14

[CJL16] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without a low level encoding of zero. Cryptology ePrint Archive, Report 2016/139, 2016. `http://eprint.iacr.org/2016/139`. 16

[CN17] Jean-Sébastien Coron and Jesper Buus Nielsen, editors. *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. 64, 65

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir transformation in the quantum random-oracle model. In Boldyreva and Micciancio [BM19], pages 356–383. 19

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 22–41, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany. 6, 9, 11, 13, 18, 20

[dLP16] Rafaël del Pino, Vadim Lyubashevsky, and David Pointcheval. The whole is less than the sum of its parts: Constructing more efficient lattice-based AKEs. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 273–291, Amalfi, Italy, August 31 – September 2, 2016. Springer, Heidelberg, Germany. 9, 20

[DN12] Léo Ducas and Phong Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 433–450, Beijing, China, December 2–6, 2012. Springer, Heidelberg, Germany. 6, 10, 14

[DP16] Léo Ducas and Thomas Prest. Fast fourier orthogonalization. In Sergei A. Abramov, Eugene V. Zima, and Xiao-Shan Gao, editors, *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*, pages 191–198. ACM, 2016. 6, 14

[Duc18] Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 125–145, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. 15, 16

[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, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 34–63, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany. 7

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 112–131, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany. 6, 10

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206, Victoria, BC, Canada, May 17–20, 2008. ACM Press. 5, 6, 9, 10, 11

[HHP$^+$03] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: Digital signatures using the NTRU lattice. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 122–140, San Francisco, CA, USA, April 13–17, 2003. Springer, Heidelberg, Germany. 6, 10

[How07] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany. 16

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 53–71, Paris, France, April 15–17 2020. Springer, Heidelberg, Germany. 7, 20

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998. 12

[JNRV20] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. Implementing grover oracles for quantum key search on AES and LowMC. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 280–310, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany. 16

[KF17] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In Coron and Nielsen [CN17], pages 3–26. 16

[Kle00] Philip N. Klein. Finding the closest lattice vector when it's unusually close. In David B. Shmoys, editor, *11th SODA*, pages 937–941, San Francisco, CA, USA, January 9–11, 2000. ACM-SIAM. 9, 10, 13, 14

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844. 7

[KRVV19] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete gaussian sampling. A case study on falcon. *DAC*, 2019. 7

[Laa16] Thijs Laarhoven. *Search problems in cryptography: from fingerprinting to lattice sieving*. PhD thesis, Department of Mathematics and Computer Science, 2 2016. Proefschrift. 15

[LAZ18] Xingye Lu, Man Ho Au, and Zhenfei Zhang. Raptor: A practical lattice-based (linkable) ring signature. Cryptology ePrint Archive, Report 2018/857, 2018. https://eprint.iacr.org/2018/857. 7

[LDK$^+$19] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. 19

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In Pointcheval and Johansson [PJ12], pages 738–755. 19

[LZ19] Qipeng Liu and Mark Zhandry. Revisiting post-quantum Fiat-Shamir. In Boldyreva and Micciancio [BM19], pages 326–355. 19

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In Pointcheval and Johansson [PJ12], pages 700–718. 13, 14

[MW16] Daniele Micciancio and Michael Walter. Practical, predictable lattice basis reduction. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 820–849, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany. 15

[MW17] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 455–485, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany. 17

[NIS15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015. http://dx.doi.org/10.6028/NIST.FIPS.202. 31

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 7, 8, 17, 23, 32, 51

[NR06] Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 271–288, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany. 6, 10, 14

[OSHG19] Tobias Oder, Julian Speith, Kira Höltgen, and Tim Güneysu. Towards practical microcontroller implementation of the signature scheme Falcon. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, pages 65–80, Chongqing, China, May 8–10 2019. Springer, Heidelberg, Germany. 7

[Pei10] Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany. 9, 13, 14

[PFH$^+$17] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`. 8

[PFH$^+$19] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`. 8

[PJ12] David Pointcheval and Thomas Johansson, editors. *EUROCRYPT 2012*, volume 7237 of *LNCS*, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany. 66

[Por19] Thomas Pornin. New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893, 2019. `https://eprint.iacr.org/2019/893`. 7, 20

[PP19] Thomas Pornin and Thomas Prest. More efficient algorithms for the NTRU key generation using the field norm. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 504–533, Beijing, China, April 14–17, 2019. Springer, Heidelberg, Germany. 7, 32, 35

[Pre15] Thomas Prest. *Gaussian Sampling in Lattice-Based Cryptography*. Theses, École Normale Supérieure, December 2015. 13

[Pre17] Thomas Prest. Sharper bounds in lattice-based cryptography using the Rényi divergence. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 347–374, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany. 14, 17, 19, 32

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in TLS 1.3: A performance study. In *NDSS 2020*, San Diego, CA, USA, February 23-26, 2020. The Internet Society. 7

[SS11] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 27–47, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany. 6, 12

[ZSS20] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: fast, compact, and constant-time discrete gaussian sampler over integers. *IEEE Trans. Computers*, 69(1):126–137, 2020. 7, 42