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

Specifications v1.1

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

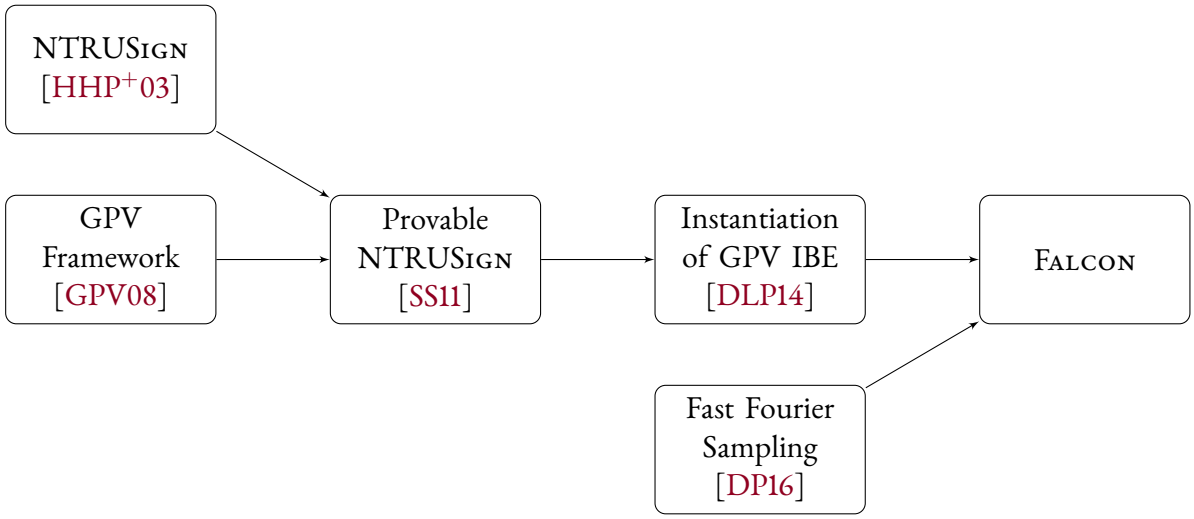

Figure 1.1: The genealogic tree of Falcon

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 flow 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 figure 1.1 shows the genealogic tree of Falcon, the first of the many trees that this document contain.

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.

Constant-time Gaussian sampling. Realising efficient constant-time Gaussian sampling over the integers has long been identified as an important problem. Recent works by Zhao *et al.* [ZSS18] and Karmakar *et al.* [KRVV19], in addition to proposing new techniques to achieve it, have studied the impact induced by integrating their techniques to FALCON. In both works, the running time overhead induced by their techniques did not exceed 33%.

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-M4. Oder *et al.* [OSHG19] have implemented FALCON on an ARM Cortex-M4 microprocessor. The work focuses on reducing the memory footprint so that the scheme can fit in the Cortex-M4 limited RAM. The signing procedure is somewhat slow, and we see it as an interesting open problem to make it faster.

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.

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, in particular Section 4.6.
- 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.

- A statement of the advantages and limitations as per [NIS16, Section 2.B.6] can be found in Section 2.6.
- Set of parameters to address the five security levels required by NIST [NIS16, Section 4.A.5] can be found in Section 3.12.

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

- A cover sheet as per [NIS16, Section 2.A] is present in this 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.
- Signed statements of intellectual property, as required by [NIS16, Section 2.D], will be conveyed to NIST physically by all submitters, patent owners and implementation authors.

1.4 Differences with the Version 1.0 of the Specification

This document is the version 1.1 of the specification of FALCON. The differences with the version 1.0 are the following:

- 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 1.1 of the specification, as well as the reference implementation, 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 competitive from a compactness perspective (Section 3.13);
- We did a few other minor additions which essentially consist of clarifying and detailing a few points.

1.5 Additional Resources

Additional resources can be found on FALCON's official website:

https://falcon-sign.info/

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 [?]. 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: $\mathbf{B} \times \mathbf{A}^t = \mathbf{0}$.
- Given a message m, a signature of m is a short value $\mathbf{s} \in \mathbb{Z}_q^m$ such that $\mathbf{s}\mathbf{A}^t = H(\mathsf{m})$, 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 $\mathbf{s}\mathbf{A}^t = H(\mathsf{m})$.
- Computing a valid signature is more delicate. First, an arbitrary preimage $\mathbf{c}_0 \in \mathbb{Z}_q^m$ is computed, which verifies $\mathbf{c}_0\mathbf{A}^t = H(\mathsf{m})$. As $\mathbf{c}_0$ is not required to be short and $m \geq n$, this can simply be done through 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, $\mathbf{s}\mathbf{A}^t = \mathbf{c}_0\mathbf{A}^t - \mathbf{v}\mathbf{A}^t = \mathbf{c} - \mathbf{0} = H(\mathsf{m})$, and if $\mathbf{c}_0$ and $\mathbf{v}$ are close enough, then $\mathbf{s}$ is short.

In this abstract form, this description of a signature scheme is not specific to the GPV framework: it was first instantiated in the GGH [GGH97] and NTRUSign [HHP+03] signature schemes. However, GGH and NTRUSign suffer of total break attacks, whereas the GPV framework is proven to be secure in the classical and quantum random oracle models assuming the hardness of SIS for some parameters. The reason behind this is somewhat subtle, and lies in the fact that GGH/NTRUSign and the GPV framework have radically different ways of computing $\mathbf{v}$ in the signing procedure.

Computing $\mathbf{v}$ 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 each $\mathbf{s}$ leaks a little information about the basis $\mathbf{B}$. This fact was successfully exploited by several attacks [NR06, DN12] which led to a total break on the schemes.

Computing $\mathbf{v}$ 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 not only impervious to the attacks described hereabove, but is also 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. This is described in [DLP14]. However, this requires de-randomizing the signature procedure (see the paragraph "Statefulness, de-randomization or hash randomization").

2.2.2 Statefulness, de-randomization or hash randomization

In the GPV framework, two different signatures $\mathbf{s}, \mathbf{s}'$ 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 occuring. 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 $\phi \in \mathbb{Z}[x]$ be a monic polynomial, 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 $\|\mathbf{B}\|_{\text{GS}}$, where $\|\mathbf{B}\|_{\text{GS}}$ 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, $\|\mathbf{B}\|_{\text{GS}}$ 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 $\|\mathbf{B}\|_{\text{GS}}$ 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 $\|\mathbf{B}\|_{\text{GS}} \leq 1.17\sqrt{q}$ that we reach in practice is within a factor 1.17 of the theoretic lower bound for $\|\mathbf{B}\|_{\text{GS}}$. 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 $\|\mathbf{B}\|_{\text{GS}}$.

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 $\|\mathbf{B}\|_{GS}$, 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. This approach was generalized in [LW15]. Unfortunately, it is not straightforwardly compatible with NTRU lattices and whether we can reach the same level of compactness as with NTRU lattices is unclear.

4. Ducas and Prest [DP16] proposed 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, and for this reason they dubbed it "fast Fourier nearest plane". This algorithm can be randomized as well: 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 $\mathbf{s}$**	**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 (i.e. the shortest which is known not to leak the secret basis). According to [Pre17], it is sufficient for the security level and number of queries set by NIST to take $\sigma \leq 1.312 \|\mathbf{B}\|_{\text{GS}}$, which in our case translates to $\sigma \leq 1.55\sqrt{q}$.

2.5 Security

2.5.1 Known Attacks

Key Recovery. The most efficient attacks come from lattice reduction. We start by considering the lattice $(\mathbb{Z}[x]/(\phi))^2 \left[\begin{array}{c|c} 0 & q \\ \hline 1 & h \end{array}\right]$. After using lattice reduction on this basis, we enumerate all lattice points in a ball of radius $\sqrt{2n}\sigma'$, centered on the origin. With significant probability, we are therefore able to find $\left[\begin{array}{c|c} g & f \end{array}\right]$. If we use a block-size of B, enumeration takes negligible time if the $(2n-B)$th Gram-Schmidt norm is larger than $\sqrt{3B/4}\sigma'$. For the best known lattice reduction algorithm, DBKZ [MW16], it is

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

It is then easy to deduce B, and to show that $B = n + o(n)$, which is the fastest attack *asymptotically*. 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 is also eased by first carrying out lattice reduction on the original basis. Using an embedding and DBKZ, the success condition is :

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

This is the best attack against our instantiations, even if we have $B = 2n + o(n)$. The security implied is detailed in the following table, using the methodology of New Hope [ADPS16].

n	B	Classical	Quantum
512	392	114	103
1024	921	263	230

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 arithmetics, 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 (nonwithstanding 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 to discard 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:

$$\left|\log\left(\frac{\bar{\mathcal{D}}(\mathbf{z})}{\mathcal{D}(\mathbf{z})}\right)\right| \lesssim 2n\left[\frac{\sqrt{154}}{1.312}\delta_c + (2\pi+1)\delta_\sigma\right] \leq 20n(\delta_c + \delta_\sigma) \tag{2.3}$$

In the average case, the value $2n$ in 2.3 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 Advantages and Limitations of Falcon

This section lists the advantages and limitations of Falcon.

2.6.1 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 [BHH+15, AE17]. Conversely, some multivariate [KPG99, DS05] and code-based [CFS01] signature schemes achieve very small signatures but require large public keys. Lattice-based schemes [DLL+17] can somewhat offer the best of both worlds, but we do not know of any post-quantum signature schemes getting $|\mathsf{pk}| + |\mathsf{sig}|$ 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, and a security proof in the quantum random oracle model was later provided in [BDF+11]. This stands in contrast with schemes using the Fiat-Shamir heuristic, which are notoriously harder to render secure in the QROM [Unr17, KLS18].

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 about two orders of magnitudes slower, but it would be simpler to implement and the security would remain exactly 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 3.13.

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.

2.6.2 Limitations

FALCON also has a few limitations. These limitations are implementation-related and interestingly, they concern only the signer. We list them below.

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 constitue 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.

Unclear side-channel resistance. FALCON relies heavily on discrete Gaussian sampling over the integers. How to implement this securely with respect to timing and side-channel attacks has remained largely unstudied, although this has recently started to change [ZSS18, KRVV19, MW17, RRVV14].

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 fulfill 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 3.12.

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. However, using these endomorphisms to their full potential is not easy, as it entails manipulating individual coefficients of polynomials (or of their Fourier transform) and working with binary trees. We will see that most of the technicalities of FALCON arise from this.

3.3 Notations

Cryptographic parameters. For a cryptographic signature scheme, λ denotes its security level and q_s the maximal number of signature queries which may be made. Following the assumptions of [NIS16], we suppose that $q_s \leq 2^{64}$.

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, $q = 12289$ is prime so $\mathbb{Z}_q$ becomes a finite field. We also denote by $\mathbb{Z}_q^\times$ the group of invertible elements of $\mathbb{Z}_q$, and by φ Euler's totient function: $\varphi(q) = |\mathbb{Z}_q^\times| = q-1$ since q is prime.

Number fields. We denote by ϕ a monic polynomial of $\mathbb{Z}[x]$, irreducible in $\mathbb{Q}[x]$, of degree n and with distinct roots over $\mathbb{C}$. In FALCON, we will always consider that $\phi = x^n + 1$ for $n = 2^\kappa$.

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. $\mathbf{v}$).

Inner product. The inner product over $\mathcal{Q}$ is $\langle a, b \rangle = \frac{1}{\deg(\phi)} \sum_{\phi(\zeta)=0} a(\zeta) \cdot \overline{b(\zeta)}$, and the associated norm is $\|a\| = \sqrt{\langle a, a \rangle}$. We extend this definition to vectors: for $\mathbf{u} = (u_i)_i$ and $\mathbf{v} = (v_i)_i$ in $\mathcal{Q}^m$, we define $\langle \mathbf{u}, \mathbf{v} \rangle$ as $\sum_i \langle u_i, v_i \rangle$. For our choice of ϕ, the inner product coincides with the usual coefficient-wise inner product:

$$\langle a, b \rangle = \sum_{0 \le i < n} a_i b_i; \tag{3.7}$$

Ring Lattices. For the rings $\mathcal{Q} = \mathbb{Q}[x]/(\phi)$ and $\mathcal{Z} = \mathbb{Z}[x]/(\phi)$, positive integers $m \ge n$ 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 $\mathcal{Z}^n \cdot \mathbf{B} = \{\mathbf{zB} | \mathbf{z} \in \mathcal{Z}^n\}$. 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

$$D_{\mathbb{Z},\sigma,\mu}(x) = \frac{\rho_{\sigma,\mu}(x)}{\sum_{z \in \mathbb{Z}} \rho_{\sigma,\mu}(z)}. \tag{3.8}$$

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

Field norm. Let $\mathbb{K}$ be a number field of degree $n = [\mathbb{K} : \mathbb{Q}]$ over $\mathbb{Q}$ and $\mathbb{L}$ be a Galois extension of $\mathbb{K}$. We denote by $\mathrm{Gal}(\mathbb{L}/\mathbb{K})$ the Galois group of $\mathbb{L}/\mathbb{K}$. The field norm $\mathrm{N}_{\mathbb{L}/\mathbb{K}} : \mathbb{L} \to \mathbb{K}$ is a map defined for any $f \in \mathbb{L}$ by the product of the Galois conjugates of f:

$$\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f) = \prod_{\mathrm{g} \in \mathrm{Gal}(\mathbb{L}/\mathbb{K})} \mathrm{g}(f). \tag{3.9}$$

Equivalently, $\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f)$ can be defined as the determinant of the $\mathbb{K}$-linear map $y \in \mathbb{L} \mapsto fy$. One can check that the field norm is a multiplicative morphism.

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

$$\mathbf{B} = \mathbf{L} \times \tilde{\mathbf{B}}, \tag{3.10}$$

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 $\mathbf{b}_i \mathbf{b}_j^\star = 0$ 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:

$$\|\mathbf{B}\|_{\text{GS}} = \max_{\tilde{\mathbf{b}}_i \in \tilde{\mathbf{B}}} \|\tilde{\mathbf{b}}_i\|. \tag{3.11}$$

The LDL* decomposition. Closely related to the GSO is the LDL* decomposition. It 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. It can be computed using Algorithm 8.

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 $\mathbf{G} = \mathbf{BB}^\star$, 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:

$$\left[\mathbf{L} \cdot \tilde{\mathbf{B}} \text{ is the GSO of } \mathbf{B}\right] \Leftrightarrow \left[\mathbf{L} \cdot (\tilde{\mathbf{B}}\tilde{\mathbf{B}}^\star) \cdot \mathbf{L}^\star \text{ is the LDL}^\star \text{ decomposition of } (\mathbf{BB}^\star)\right]. \tag{3.12}$$

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 $\beta > 0$.

For clarity, all the parameters presented 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:

$$fG - gF = q \bmod \phi. \tag{3.13}$$

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:

$$\hat{\mathbf{B}} = \left[\begin{array}{c|c} \mathsf{FFT}(g) & -\mathsf{FFT}(f) \\ \hline \mathsf{FFT}(G) & -\mathsf{FFT}(F) \end{array}\right], \tag{3.14}$$

  $\mathsf{FFT}(a)$ 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 T.value, is a polynomial $\ell \in \mathbb{Q}[x]/(x^n + 1)$ with $n = 2^\kappa$.
  - Its left and right children, noted T.leftchild and 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 the figure 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

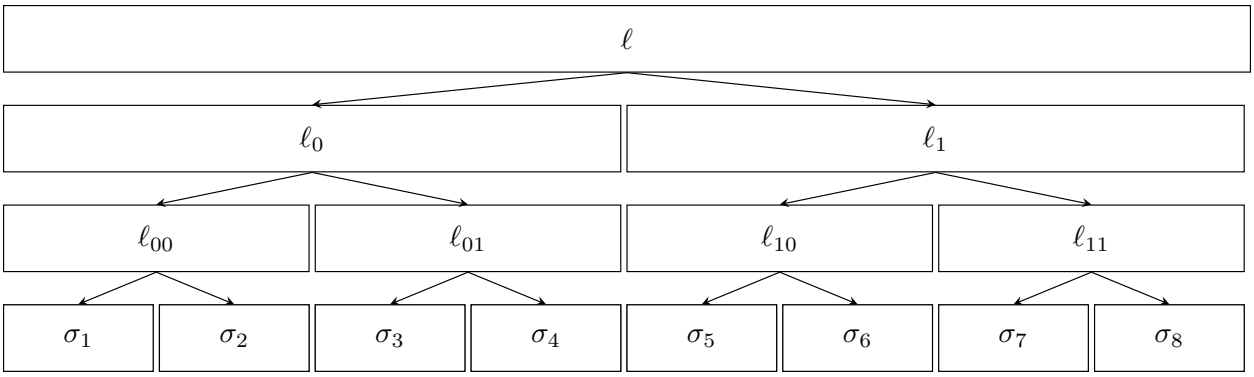

Figure 3.1: A FALCON tree of height 3

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.

3.4.3 Public key

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

$$h = gf^{-1} \bmod (\phi, q). \tag{3.15}$$

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:

$$\mathsf{FFT}_\phi(f) = (f(\zeta))_{\zeta \in \Omega_\phi} \tag{3.16}$$

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 ϕ as in FALCON, we have $\Omega_\phi = \{\zeta^k | k \in \mathbb{Z}_{2n}^\times\}$, with ζ a primitive $2n$-th complex root of 1.

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 $f(x) = f_0(x^2) + xf_1(x^2)$, with $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$. In coefficient representation, such a decomposition is straightforward to write:

$$f_0 = \sum_{0 \leq i < n/2} a_{2i} x^i \quad \text{and} \quad f_1 = \sum_{0 \leq i < n/2} a_{2i+1} x^i \tag{3.17}$$

f is simply split with respect to its even or odd coefficients. We note $(f_0, f_1) = \mathsf{split}(f)$. 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 algorithm 1.

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

Require: $\mathsf{FFT}(f) = (f(\zeta))_\zeta$ for some $f \in \mathbb{Q}[x]/(\phi)$
Ensure: $\mathsf{FFT}(f_0) = (f_0(\zeta'))_{\zeta'}$ and $\mathsf{FFT}(f_1) = (f_1(\zeta'))_{\zeta'}$ for some $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$
Format: All polynomials are in FFT representation.

1: for ζ such that $\phi(\zeta) = 0$ and $\mathrm{Im}(\zeta) > 0$ do
2: $\quad \zeta' \leftarrow \zeta^2$
3: $\quad f_0(\zeta') \leftarrow \frac{1}{2}[f(\zeta) + f(-\zeta)]$
4: $\quad f_1(\zeta') \leftarrow \frac{1}{2\zeta}[f(\zeta) - f(-\zeta)]$
5: return $(\mathsf{FFT}(f_0), \mathsf{FFT}(f_1))$

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: $\mathsf{merge}(f_0, f_1) = f_0(x^2) + xf_1(x^2) \in \mathbb{Q}[x]/(\phi)$. Similarly to split, it is often relevant from an efficiently standpoint to perform merge in the FFT representation. This is done in 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.

Algorithm 2 mergefft(f_0, f_1)

Require: $\mathsf{FFT}(f_0) = (f_0(\zeta'))_{\zeta'}$ and $\mathsf{FFT}(f_1) = (f_1(\zeta'))_{\zeta'}$ for some $f_0, f_1 \in \mathbb{Q}[x]/(\phi')$
Ensure: $\mathsf{FFT}(f) = (f(\zeta))_{\zeta}$ for some $f \in \mathbb{Q}[x]/(\phi)$
Format: All polynomials are in FFT representation.

1: for ζ such that $\phi(\zeta) = 0$ do
2: $\quad \zeta' \leftarrow \zeta^2$
3: $\quad f(\zeta) \leftarrow f_0(\zeta') + \zeta f_1(\zeta')$
4: return $\mathsf{FFT}(f)$

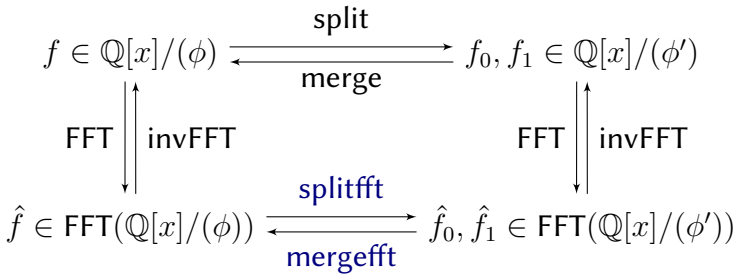

Figure 3.2: Relationship between FFT, invFFT, split, merge, splitfft and mergefft

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 figure 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. As an illustration, we consider the operation:

$$a = bc \tag{3.18}$$

where $a, b, c \in \mathbb{Q}[x]/(\phi)$. For $f \in \mathbb{Q}[x]/(\phi)$, we consider the associated endomorphism $\psi_f : z \in \mathbb{Q}[x]/(\phi) \mapsto fz$. (3.18) 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 (3.18) as:

$$\left[\begin{array}{c|c} a_0 & a_1 \end{array}\right] = \left[\begin{array}{c|c} b_0 & b_1 \end{array}\right] \left[\begin{array}{c|c} c_0 & c_1 \\ \hline xc_1 & c_0 \end{array}\right] \tag{3.19}$$

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 (3.19).

3.6.2 Relationship with the field norm

The splitting and merging operators allow to easily express the field norm for some specific cyclotomic fields. Let $\mathbb{L} = \mathbb{Q}[x]/(\phi), \mathbb{K} = \mathbb{Q}[x]/(\phi')$ and $f \in \mathbb{L}$. Since by definition $\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f) = \det_{\mathbb{K}}(\psi_d)$, we can use (3.19) to compute it explicitly. This yields:

- If $\phi'(x^2) = \phi(x)$, then $\mathrm{N}_{\mathbb{L}/\mathbb{K}}(f) = f_0^2 - x f_1^2$, where $(f_0, f_1) = \mathsf{split}(f)$;

For $f \in \mathbb{L}$ with $\mathbb{L} = \mathbb{Q}[x]/(x^{2^\kappa} + 1)$, we also denote $\mathrm{N}(f) = f_0^2 - x f_1^2 = \mathrm{N}_{\mathbb{L}/\mathbb{K}}(f)$, where $\mathbb{K}$ is the largest strict subfield of $\mathcal{L}$ (see (3.4)). For the values of ϕ considered in this document, this allows to define $\mathrm{N}(f)$ in an unambiguous way.

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.

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 defines the hashing process used in Falcon. It is defined for any $q \leq 2^{16}$.

Possible variants.

- If $q > 2^{16}$, then larger chunks can be extracted from SHAKE-256 at each step.

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 $\leftarrow$ SHAKE-256-Init()
3: SHAKE-256-Inject(ctx, str)
4: $i \leftarrow 0$
5: while $i < n$ do
6: $\quad t \leftarrow$ SHAKE-256-Extract(ctx, 16)
7: $\quad$ if $t < kq$ then
8: $\quad\quad c_i \leftarrow t \bmod q$
9: $\quad\quad i \leftarrow i + 1$
10: return c

- Algorithm 3 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 target values modulo q will be 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 is arguably the most technical part of Falcon to describe. It can be chronologically and conceptually decomposed in two clearly separate parts, which each contain their own technicalities, make use of different mathematical tools and require to address different challenges.

- *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 (3.13) – the NTRU equation. Generating f and g is easy; the hard part is to efficiently compute polynomials F, G such that (3.13) is verified.

  In order to do this, we propose a novel method which exploits the tower-of-rings structure explicited 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 $\mathbb{Z}$ – which amounts to an extended gcd – and use the properties of the norm to lift the solutions (F, G) in the tower of rings, up to the ring $\mathbb{Z}[x]/(\phi)$.

From an implementation point of view, the main technicality of this part is that it requires to handle polynomials with large coefficients (a few thousands of bits per coefficient in the lowest levels of the recursion). This step is specified in Section 3.8.2.

- *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

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right], \tag{3.20}$$

  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 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.

  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 figure 3.3.

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 the NTRU equation: $fG - gF = q \bmod \phi$. This is specified in algorithm 5 (NTRUGen). We provide a general explanation of the algorithm:

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 (steps 1 to 11). It particular, it shall be checked that:

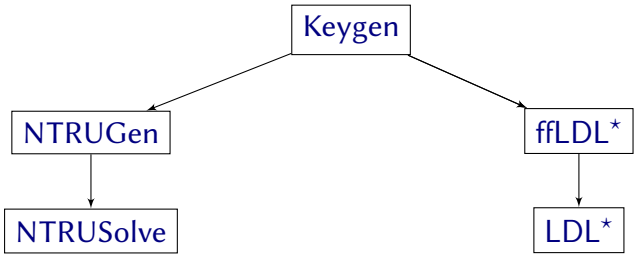

Figure 3.3: Flowchart of the key generation

Algorithm 4 Keygen(ϕ, q)

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

1: $f, g, F, G, \gamma \leftarrow$ NTRUGen(ϕ, q) ▷ Solving the NTRU equation
2: $\mathbf{B} \leftarrow \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$
3: $\hat{\mathbf{B}} \leftarrow \mathsf{FFT}(\mathbf{B})$
4: $\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^\star$
5: $\mathsf{T} \leftarrow$ ffLDL$^\star(\mathbf{G})$ ▷ Computing the LDL* tree
6: $\sigma \leftarrow 1.55\sqrt{q}$ ▷ Normalization step
7: for each leaf leaf of T do
8: leaf.value $\leftarrow \sigma / \sqrt{\text{leaf.value}}$
9: sk $\leftarrow (\hat{\mathbf{B}}, \mathsf{T})$
10: $h \leftarrow gf^{-1} \bmod q$
11: pk $\leftarrow h$
12: return sk, pk

(a) 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 $\mathrm{Res}(f, \phi) \bmod q \neq 0$, where Res denotes the resultant. The NTT can be used to perform this check and then compute h (f is invertible in $\mathbb{Z}_q[x]/(\phi)$ if and only if its NTT representation contains no zero).

(b) The polynomials f, g, F, G allow to generate short signatures. This is the case if and only if $\gamma = \max\left\{\|(g, -f)\|, \left\|\left(\frac{qf^\star}{ff^\star+gg^\star}, \frac{qg^\star}{ff^\star+gg^\star}\right)\right\|\right\}$ is small enough.

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

Solving the NTRU equation

We now explain how to solve (3.13). As said before, we repeatedly use the field norm N (see Section 3.6.2 for explicit formulae) to map f, g to a smaller ring $\mathbb{Z}[x]/(x^{n/2}+1)$, until we reach the ring $\mathbb{Z}$. Solving

Algorithm 5 NTRUGen(ϕ, q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree n, a modulus q
Ensure: Polynomials f, g, F, G

$\sigma' \leftarrow 1.17\sqrt{q/2n}$ ▷ σ' is chosen so that $\mathbb{E}[\|(f,g)\|] = 1.17\sqrt{q}$
for i from 0 to $n-1$ do
    $f_i \leftarrow D_{\mathbb{Z},\sigma',0}$
    $g_i \leftarrow D_{\mathbb{Z},\sigma',0}$
$f \leftarrow \sum_i f_i x^i$ ▷ $f \in \mathbb{Z}[x]/(\phi)$
$g \leftarrow \sum_i g_i x^i$ ▷ $g \in \mathbb{Z}[x]/(\phi)$
if $\mathrm{Res}(f, \phi) \bmod q = 0$ then ▷ Check that f is invertible mod q
    restart
$\gamma \leftarrow \max\left\{\|(g,-f)\|, \left\|\left(\frac{qf^\star}{ff^\star+gg^\star}, \frac{qg^\star}{ff^\star+gg^\star}\right)\right\|\right\}$
if $\gamma > 1.17\sqrt{q}$ then ▷ Check that signatures will be short
    restart
$F, G \leftarrow$ NTRUSolve$_{n,q}(f, g)$ ▷ Computing F, G such that $fG - gF = 1 \bmod \phi$
return f, g, F, G

3.13 then amounts to computing an extended GCD over $\mathbb{Z}$, which is simple. Once this is done, we 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 (3.13).

Algorithm 6 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 (3.13) 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
    $(F, G) \leftarrow (vq, uq)$
    return (F, G)
else
    $f' \leftarrow \mathrm{N}(f)$ ▷ $f', g', F', G' \in \mathbb{Z}[x]/(x^{n/2}+1)$
    $g' \leftarrow \mathrm{N}(g)$ ▷ See Section 3.6.2
    $(F', G') \leftarrow$ NTRUSolve$_{n/2,q}(f', g')$ ▷ Recursive call
    $F \leftarrow F'(x^2)g'(x^2)/g(x)$ ▷ $F, G \in \mathbb{Z}[x]/(x^n+1)$
    $G \leftarrow G'(x^2)g'(x^2)/g(x)$
    Reduce(f, g, F, G) ▷ (F, G) is reduced with respect to (f, g)
return (F, G)

NTRUSolve uses the procedure Reduce 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 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)

1: do
2: $\quad k \leftarrow \left\lfloor \frac{Ff^\star + Gg^\star}{ff^\star + gg^\star} \right\rceil$ ▷ $\frac{Ff^\star + Gg^\star}{ff^\star + gg^\star} \in \mathbb{Q}[x]/(\phi)$ and $k \in \mathbb{Z}[x]/(\phi)$
3: $\quad F \leftarrow F - kf$
4: $\quad G \leftarrow G - kg$
5: 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 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 Algorithm 4 (steps 6-8).

  The polynomials manipulated in Algorithm 9 and its subroutine Algorithm 8 are all in FFT representation. While it is possible to convert these algorithms to the coefficient representation, doing so would be suboptimal from an efficiency viewpoint.

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. Such a decomposition is easy to compute: we recall a method for doing so in Algorithm 8.

   We store the matrix $\mathbf{L}$ in $\mathsf{T}.\mathsf{value}$, which is the value of the root of T. Since $\mathbf{L}$ is a lower triangular matrix with 1's on the diagonal of dimensions – in our case – (2×2), this only amounts to storing one element of $\mathbb{Q}[x]/(\phi)$.

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

$$\left[\begin{array}{c|c} d_0 & d_1 \\ \hline xd_1 & d_0 \end{array}\right] \left(= \left[\begin{array}{c|c} d_0 & d_1 \\ \hline d_1^\star & d_0 \end{array}\right]\right)^{1}. \tag{3.21}$$

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 Algorithm 9.

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

Require: A full-rank autoadjoint matrix $\mathbf{G} = (G_{ij}) \in \mathsf{FFT}(\mathbb{Q}[x]/(\phi))^{n\times n}$
Ensure: The LDL* decomposition $\mathbf{G} = \mathbf{LDL}^\star$ over $\mathsf{FFT}(\mathbb{Q}[x]/(\phi))$
Format: All polynomials are in FFT representation.

1: $\mathbf{L}, \mathbf{D} \leftarrow \mathbf{0}^{n\times n}$
2: for i from 1 to n do
3: $\quad L_{ii} \leftarrow 1$
4: $\quad D_i \leftarrow G_{ii} - \sum_{j<i} L_{ij} \odot L_{ij}^\star \odot D_j$
5: $\quad$ for j from 1 to $i-1$ do
6: $\quad\quad L_{ij} \leftarrow \frac{1}{D_j}\left(G_{ij} - \sum_{k<j} L_{ik} \odot L_{jk}^\star \odot D_k\right)$
7: return $(\mathbf{L}, \mathbf{D})$

3.9 Signature Generation

3.9.1 Overview

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 $\mathbf{t} \leftarrow (c, 0) \cdot \mathbf{B}^{-1}$, to round it coefficient-wise to a vector $\mathbf{z}$ and to output $(s_1, s_2) \leftarrow (\mathbf{t} - \mathbf{z})\mathbf{B}$; one can check that (s_1, s_2) does fill all the requirements to be a legitimate, but this method is known to be insecure and to leak the secret key.

The proper way to generate (s_1, s_2) without leaking the secret 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

[1] The equality in parenthesis is true if and only if d is self-adjoint, i.e. $d^\star = d$.

Algorithm 9 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.

1: $(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^\star(\mathbf{G})$ ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
2: T.value $\leftarrow L_{10}$
3: if $(n = 2)$ then
4: T.leftchild $\leftarrow D_{00}$
5: T.rightchild $\leftarrow D_{11}$
6: return T
7: else
8: $d_{00}, d_{01} \leftarrow \mathsf{splitfft}(D_{00})$
9: $d_{10}, d_{11} \leftarrow \mathsf{splitfft}(D_{11})$
10: $\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline xd_{01} & d_{00} \end{array}\right], \mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline xd_{11} & d_{10} \end{array}\right]$ ▷ $\mathbf{G}_0, \mathbf{G}_1 \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)^{2\times 2}$
11: T.leftchild $\leftarrow \mathsf{ffLDL}^\star(\mathbf{G}_0)$ ▷ Recursive calls
12: T.rightchild $\leftarrow \mathsf{ffLDL}^\star(\mathbf{G}_1)$
13: return T

Fourier sampling. The computation of the falcon tree T by the procedure ffLDL* during the key pair generation was the initialization step of this trapdoor sampler.

The heart of our signature generation, the procedure ffSampling (Algorithm 11), will adaptatively apply a randomized rounding (according to a discrete Gaussian distribution) on the coefficients of $\mathbf{t}$. But it will do so in an adaptative 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 $\mathbf{t} = (t_0, t_1)$, 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 base ring of $\mathbb{Z}[x]/(\phi)$; it then applies itself again recursively with t_0. It is important to notice 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 algorithm 11, 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 secret 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

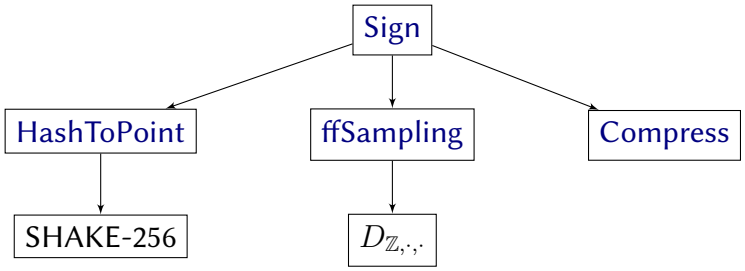

Figure 3.4: Flowchart of the signature

to a point $c \in \mathbb{Z}_q[x]/(\phi)$ as specified by 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 Algorithm 11.

3. s_2 is encoded (compressed) to a bitstring s as specified in Section 3.11.

4. The signature consists of the pair (r, s).

Algorithm 10 Sign $(\mathsf{m}, \mathsf{sk}, \beta)$

Require: A message m, a secret key sk, a bound β
Ensure: A signature sig of m

$\mathsf{r} \leftarrow \{0,1\}^{320}$ uniformly
$c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m}, q, n)$
$\mathbf{t} \leftarrow (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
do
    $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$
    $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
while $\|\mathbf{s}\| > \beta$
$(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
$\mathsf{s} \leftarrow \mathsf{Compress}(s_2)$
return $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$

A note on sampling over $\mathbb{Z}$. Algorithm 11 requires access to an oracle $\mathcal{D}$ for the distribution $D_{\mathbb{Z},\sigma',c'}$, where σ' can be the value of any leaf of the private FALCON tree T, and $c' \in \mathbb{Q}$ is arbitrary. How to implement $\mathcal{D}$ is outside the scope of this specification. It is only required that the Rényi divergence between this oracle and an ideal discrete Gaussian $D_{\mathbb{Z},\sigma',c'}$ verifies $R_{512}(\mathcal{D} \| D_{\mathbb{Z},\sigma',c'}) \leq 1 + 2^{-66}$, for the definition of the Rényi divergence given in e.g. [BLL+15]. We note that several proposals [ZWXZ18, ZSS18, KRVV19, Wal19] for efficient (constant-time) Gaussian sampling over the integers have been made recently.

Our reference implementation uses a Gaussian sampler based on rejection sampling against a bimodal

distribution; it is described in Section 4.4. We note that the range of possible values for the standard deviation in the Gaussian sampler is limited: it is always greater than 1.2, and always lower than 1.9.

3.9.2 Fast Fourier Sampling

This section describes our fast Fourier sampler (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 $O(n/\log n)$ 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.

1: if $n = 1$ then
2: $\quad \sigma' \leftarrow \mathsf{T.value}$
3: $\quad z_0 \leftarrow D_{\mathbb{Z},\, t_0,\, \sigma'}$ ▷ Since $n = 1$, $t_0 = \mathsf{invFFT}(t_0) \in \mathbb{Q}$ and $z_0 = \mathsf{invFFT}(z_0) \in \mathbb{Z}$
4: $\quad z_1 \leftarrow D_{\mathbb{Z},\, t_1,\, \sigma'}$ ▷ Since $n = 1$, $t_1 = \mathsf{invFFT}(t_1) \in \mathbb{Q}$ and $z_1 = \mathsf{invFFT}(z_1) \in \mathbb{Z}$
5: $\quad$ return $\mathbf{z} = (z_0, z_1)$
6: $(\ell, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
7: $\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$ ▷ $\mathbf{t}_1 \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)^2$
8: $\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$ ▷ First recursive call to $\mathsf{ffSampling}_{n/2}$
9: $z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$ ▷ $\mathbf{z}_1 \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^{n/2}+1)\right)^2$
10: $t'_0 \leftarrow t_0 + (t_1 - z_1) \odot \ell$
11: $\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t'_0)$
12: $\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$ ▷ Second recursive call to $\mathsf{ffSampling}_{n/2}$
13: $z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
14: return $\mathbf{z} = (z_0, z_1)$

3.10 Signature Verification

3.10.1 Overview

The signature verification procedure is much simpler than the key pair generation and the signature generation, both to describe and to implement. Given a public key $\mathsf{pk} = h$, a message m, a signature $\mathsf{sig} = (\mathsf{r},\mathsf{s})$ and an acceptance bound β, 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 Algorithm 3.
2. s is decoded (decompressed) to a polynomial $s_2 \in \mathbb{Z}[x]/(\phi)$ as specified in Section 3.11.
3. The value $s_1 = c - s_2 h \bmod q$ is computed.
4. If $\|(s_1, s_2)\| \leq \beta$, then the signature is accepted as valid. Otherwise, it is rejected.

The only subtlety here is that, as recalled in the notations, $\|\|$ denotes the embedding norm and not the coefficient norm. However, it is possible to compute it in linear time. Given two polynomials a and b in $\mathbb{Z}_q[x]/(\phi)$, whose coefficients are denoted a_j and b_j, respectively, the norm $\|(a, b)\|$ is such that:

$$\|(a, b)\|^2 = \sum_{j=0}^{n-1} (a_j^2 + b_j^2) \tag{3.22}$$

3.10.2 Specification

The specification of the signature verification is given in Algorithm 12.

Algorithm 12 Verify (m, sig, pk, β)

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

```
c ← HashToPoint(r||m, q, n)
s_2 ← Decompress(s)
s_1 ← c − s_2 h mod q
if ||(s_1, s_2)|| ≤ β then
    accept
else
    reject
```

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 β^2.

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 $\sigma = 1.55\sqrt{q} \ll q$. 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 $\underbrace{0 \ldots 0}_{k \text{ times}} 1$;
2. The compression of s is the concatenated string $\mathsf{str} \leftarrow (\mathsf{str}_0 \| \mathsf{str}_1 \| \ldots \| \mathsf{str}_{n-1})$.

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 Algorithm 13.

The corresponding decompression algorithm is given in Algorithm 14. For any polynomial $s \in \mathbb{Z}[x]$, it holds that Decompress $\circ$ Compress$(s) = s$.

Algorithm 13 Compress(s)

Require: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]$ of degree $< n$
Ensure: A compressed representation str of s

str $\leftarrow \{\}$ ▷ str is the empty string
for i from 0 to $n-1$ do ▷ At each step, str $\leftarrow$ (str$\|$str$_i$)
    str $\leftarrow$ (str$\|b$), where $b = 1$ if $s_i > 0$, $b = 0$ otherwise
    str $\leftarrow$ (str$\|b_0 b_1 \dots b_6$), where $b_j = \lfloor |s_i|/2^j \rfloor \bmod 2$
    $k \leftarrow \lfloor |s_i|/2^7 \rfloor$
    for j from 1 to k do
        str $\leftarrow$ (str$\|0$)
    str $\leftarrow$ (str$\|1$)
return str

Algorithm 14 Decompress(str)

Ensure: A string str $= (\text{str}[i])_{i=0\dots\ell-1}$ of length ℓ
Require: A polynomial $s = \sum s_i x^i \in \mathbb{Z}[x]$

$j \leftarrow 0$
for i from 0 to $n-1$ do
    $s'_i \leftarrow \sum_{j=0}^{6} 2^j \text{str}[1+j]$ ▷ We recover the lowest bits of $|s_i|$.
    $k \leftarrow 0$ ▷ We recover the highest bits of $|s_i|$.
    while str$[7+k] = 0$ do
        $k \leftarrow k+1$
    $s_i \leftarrow (-1)^{\text{str}[0]+1} \cdot (s'_i + 2^7 k)$ ▷ We recover s_i.
    str $\leftarrow$ str$[9+k \dots \ell-1]$ ▷ We remove the bits of str already read.
return $s = \sum_{i=0}^{n-1} s_i x^i$

3.11.3 Signatures

A FALCON signature consists of two strings r and s. They are normally 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.

s encodes the polynomial s_2 in a sequence of bytes. The first byte has the following format (bits indicated from most to least significant):

```
t c c 0 n n n n
```

with these conventions:

- The leftmost bit t is 0 (in a previous version, t could be set to 1 to indicate the ternary case).
- Bits cc indicates the compression algorithm: 00 is "uncompressed", and 01 is "compressed". Values 10 and 11 are reserved and shall not be used for now.
- The fourth bit is reserved and must be zero.
- Bits nnnn encode a value ℓ such that $n = 2^\ell$. ℓ must be in the allowed range (1 to 10).

Following this header byte is the encoded s_2 value (s string). If the compression algorithm is "uncompressed", then the n coefficients of s_2 follow, in signed (two's complement) big-endian 16-bit encoding. If the algorithm is "compressed", then the compression algorithm described in Section 3.11.2 is applied and yields s as a sequence of bits; extra bits in the final byte (if the length of s is not a multiple of 8) are set to 0.

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:

```
t 0 0 0 n n n n
```

with these conventions:

- The leftmost bit t is 0 (in a previous version, t could be set to 1 to indicate the ternary case).
- The next three bits are reserved and must be zero.
- Bits nnnn encode a value ℓ such that $n = 2^\ell$. 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:

```
t c c g n n n n
```

with these conventions:

- The leftmost bit `t` is 0.
- Bits `cc` indicates the compression algorithm: 00 is "uncompressed", and 01 is "compressed". Values 10 and 11 are reserved and shall not be used for now.
- Bit `g` is 0 if the key includes the polynomial G, or 1 if G is absent.
- Bits `nnnn` encode the value ℓ such that $n = 2^\ell$. ℓ must be in the allowed range (1 to 10).

Following the header byte are the encodings of f, g, F, and optionally G, in that order. When no compression is used (bit `c` is 0), each coordinate is encoded as a 16-bit signed value (two's complement, big-endian convention). When compression is used, each polynomial is compressed with the algorithm described in Section 3.11.2; each of the four polynomial yields a bit sequence which is split into bytes with up to 7 padding bits so that each encoded polynomial starts at a byte boundary.

When G is absent (bit `g` is 1), users must recompute it. This is easily done thanks to the NTRU equation:

$$G = (q + gF)/f \mod \phi \tag{3.23}$$

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

In order to fit the API to be implemented by candidates to the NIST call for post-quantum algorithms, the following choices have been made:

- Private keys use uncompressed format: the API does not provide for keeping a private key length parameter, so the uncompressed format is used because it has a known, fixed length.
- The API assumes that a *signed message* is encoded as a single entity. In the case of FALCON, a signed message is the concatenation of, in that order:

- the signature length, in bytes (encoded over two bytes, big-endian convention);
- the salt r (40 bytes);
- the message itself;
- the signature (s).

3.12 Recommended Parameters

In this section, we specify three set of parameters to address three of the five security levels required by NIST [NIS16, Section 4.A.5]. These can be found in Table 3.1.

Level	**Dimension** n	**Polynomial** ϕ	**Modulus** q	**Acceptance bound** β^2
1 - AES128	512	$x^n + 1$	12289	43533782
4 - SHA384 5 - AES256	1024	$x^n + 1$	12289	87067565

Table 3.1: FALCON security parameters

Acceptance bound. It is important that signers and verifiers agree *exactly* on the acceptance bound β, since signatures may come arbitrarily close to that bound (signers restart the signing process when they exceed it). Banaszczyk [Ban93, Lemma 1.5] provided values for β so that the signing process restarts with negligible probability, however we can relax it to a β so that it restarts with small probability:

$$\beta^2 = \left\lfloor \frac{87067565n}{1024} \right\rfloor. \tag{3.24}$$

3.13 A Note on the Key-Recovery Mode

We mentioned in Section 2.6 that FALCON can be implemented in key-recovery mode. With the same notations as before, here is how it 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 $h = s_2^{-1}(\mathsf{HashToPoint}(\mathsf{r}\|\mathsf{m}, q, n) - s_1)$, 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 $\bmod(\phi, q)$. 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).

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 $\text{rev}(j)$, 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 $64 \times 64 \to 128$ 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 Gaussian Sampler

When sampling a short vector, the inner Gaussian sampler is invoked twice for each leaf of the LDL tree. Each invocation should produce an integer value that follows a Gaussian distribution centered on a value μ and with standard deviation σ. The centers μ change from call to call, and are dynamically computed based on the message to sign, and the values returned by previous calls to the sampler. The values of σ are the leaves of the LDL tree: they depend on the private key, but not on the message; they range between 1.2 and 1.9.

In the FALCON reference code, rejection sampling with regards to a bimodal Gaussian is used:

- The target μ is moved into the $[0..1[$ interval by adding an appropriate integer value, which will be subtracted from the sampling result at the end. For the rest of this description, we assume that $0 \leq \mu < 1$.

- A nonnegative integer z is randomly sampled following a half Gaussian distribution of standard deviation $\sigma_0 = 2$, centered on 0.
- A random bit b is obtained, to compute $z' = b + (2b - 1)z$. The integer z' follows a bimodal Gaussian distribution, and in the range of possible values for z' (depending on b), that distribution is above the target Gaussian of center μ and standard deviation σ.
- Rejection sampling is applied. z' follows the distribution:

$$G(z) = e^{-(z-b)^2/(2\sigma_0^2)} \tag{4.2}$$

  and we target the distribution:

$$S(z) = e^{-(z-\mu)^2/(2\sigma^2)} \tag{4.3}$$

  We thus generate a random bit d, whose value is 1 with probability:

$$\begin{aligned} P(d=1) &= S(z)/G(z) \\ &= e^{(z-b)^2/(2\sigma_0^2)-(z-\mu)^2/(2\sigma^2)} \end{aligned} \tag{4.4}$$

  If bit d is 1, then we return z'; otherwise, we start over.

Random values are obtained from a custom PRNG; the reference code uses ChaCha20, but any PRNG whose output is indistinguishable from random bits can be used. On a recent x86 CPU, it would make sense to use AES in CTR mode, to leverage the very good performance of the AES opcodes implemented by the CPU.

With a careful Rényi argument, the 53-bit precision of floating-point values used in the sampler computations are sufficient to achieve the required security levels.

It is worth noting that the Gaussian sampler in the FALCON reference code is not constant time, therefore it may be a source of leakage for a side-channel attack. Keeping this in mind, since the bimodal Gaussian samples can be drafted off-line and the rejection sampling only leaks inaccurate information about the secret key (which, based on the state of our knowledge, cannot result in a side-channel attack) we did not feel it was necessary to integrate a constant time Gaussian sampler in the FALCON reference code. However, we note that several proposals [ZWXZ18, ZSS18, KRVV19, Wal19] for efficient (constant-time) Gaussian sampling over the integers have been made recently and could be used in the bimodal Gaussian sampler.

Assuming that the bimodal Gaussian sampler output is unpredictable for an attacker (e.g. by using a constant-time algorithm, off-line buffering or time-padding techniques), simple adjustments can be made to obtain an execution time independant of any secret value:

- The rejection probability is proportionnal to $\sum_{z=-\infty}^{\infty} e^{-(z)^2/(2\sigma^2)}$. This sum is actually a theta function and can be approximated by $\sigma\sqrt{2\pi}$. So the probability to return z' is proportional to σ. However, one can easily remove this dependency by replacing the probability $p := S(z)/G(z)$ in (4.4) by $\frac{\sigma_{\min}}{\sigma}p$. This makes the probability independent of σ: here, $\sigma_{\min}$ is a lower bound on σ (say $\sigma_{\min} = 1.2$) which knowledge is considered not sensitive.

- In the FALCON reference code, the $\exp(x)$ implementation is similar to the C standard library which uses a floating-point division. However, the compiler may replace the division operation with its own arithmetic library routine, which may not be constant-time [Sei18]. One can easily avoid this division by using a polynomial evaluation at point x instead [ZSS18].

4.5 Key Pair Generation

4.5.1 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.

4.5.2 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.

4.5.3 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:

$$\begin{array}{rcl} F & = & F'(x^2)g'(x^2)/g(x) \mod \phi \\ G & = & G'(x^2)f'(x^2)/f(x) \mod \phi \end{array} \tag{4.5}$$

   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:

$$\begin{array}{rcl} f'(\omega^2) & = & N(f)(\omega^2) = f(\omega)f(-\omega) \\ F(\omega) & = & F'(\omega^2)g(-\omega) \end{array} \tag{4.6}$$

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:

$$\begin{aligned} a &= fu_0 - gv_0 \\ b &= fu_1 - gv_1 \end{aligned} \tag{4.7}$$

Initial values are:

$$\begin{array}{rcl} a & = & f \\ u_0 & = & 1 \\ v_0 & = & 0 \\ b & = & g \\ u_1 & = & g \\ v_1 & = & f - 1 \end{array} \tag{4.8}$$

- 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:

$$k = \left\lfloor \frac{\dot{F}\dot{f}^\star + \dot{G}\dot{g}^\star}{\dot{f}\dot{f}^\star + \dot{g}\dot{g}^\star} \right\rceil \tag{4.9}$$

  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.

4.6 Performances

The FALCON reference implementation achieves the following performance on an Intel® Core® i7-6567U CPU (clocked at 3.3 GHz):

degree	keygen (ms)	keygen (RAM)	sign/s	vrfy/s	pub length	sig length
512	6.98	14336	6081.9	37175.3	897	617.38
1024	19.64	28672	3072.5	17697.4	1793	1233.29

The following notes apply:

- 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 is used, which makes the size slightly variable; the average is reported here.
- The FALCON reference implementation uses only standard C code, not inline assembly, intrinsics or 128-bit integers. In particular, it is expected that replacing the internal PRNG (a straightforward, portable implementation of ChaCha20) with AES-CTR using the dedicated CPU opcodes will yield a substantial performance improvement. SSE2 and AVX opcodes should help with FFT, and 64-bit multiplications (with 128-bit results) might improve key generation time as well.

- The i7-6567U processor implements dynamic frequency scaling based on load and temperature. As such, measures are not very precise and tend to move by as much as 15% between any two benchmark runs.
- 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. It is expected that this would about double the CPU cost of each signature.

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. 18

[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. 17

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. Cryptology ePrint Archive, Report 2017/933, 2017. `http://eprint.iacr.org/2017/933`. 19

[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. 12

[Bab86] László Babai. On lovasz' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1), 1986. 12

[Ban93] W. Banaszczyk. New bounds in some transference theorems in the geometry of numbers. *Mathematische Annalen*, 296(4):625–636, 1993. 46

[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. 14

[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. 11, 13, 19

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 368–397, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. 19

[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, 39

[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. 18

[CFS01] Nicolas Courtois, Matthieu Finiasz, and Nicolas Sendrier. How to achieve a McEliece-based digital signature scheme. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 157–174, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany. 19

[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. 18

[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. 62, 63

[DLL+17] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. http://eprint.iacr.org/2017/633. 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. 8, 11, 13, 15, 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. 19

[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. 8, 12, 17

[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. 8, 16

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 164–175, New York, NY, USA, June 7–10, 2005. Springer, Heidelberg, Germany. 19

[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. 8, 12

[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. 7, 8, 11, 12, 13, 14

[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. 8, 12

[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. 18

[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. 14

[KF17] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In Coron and Nielsen [CN17], pages 3–26. 18

[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. 11, 12, 16

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 552–586, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. 19

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 206–222, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany. 19

[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. 9, 20, 39, 54

[LAZ18] Xingye Lu, Man Ho Au, and Zhenfei Zhang. Raptor: A practical lattice-based (linkable) ring signature. *IACR Cryptology ePrint Archive*, 2018:857, 2018. 9

[LW15] Vadim Lyubashevsky and Daniel Wichs. Simple lattice trapdoor sampling from a broad class of distributions. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 716–730, Gaithersburg, MD, USA, March 30 – April 1, 2015. Springer, Heidelberg, Germany. 16

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 700–718, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany. 16

[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. 17

[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. 18, 20

[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. 9, 10, 23, 32, 46

[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. 8, 12, 17

[OSHG19] Tobias Oder, Julian Speith, Kira Höltgen, and Tim Güneysu. Towards practical microcontroller implementation of the signature scheme falcon. *The Tenth International Conference on Post-Quantum Cryptography, Chongqing University, Chongqing*, 2019. 9

[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. 11, 16

[PP19] Thomas Pornin and Thomas Prest. More efficient algorithms for the NTRU key generation using the field norm. *IACR Cryptology ePrint Archive*, 2019:15, 2019. 9, 36

[Pre15] Thomas Prest. *Gaussian Sampling in Lattice-Based Cryptography*. Theses, École Normale Supérieure, December 2015. 15

[Pre17] Thomas Prest. Sharper bounds in lattice-based cryptography using the Rényi divergence. In Takagi and Peyrin [TP17], pages 347–374. 17, 18, 32

[RRVV14] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel secure discrete Gaussian sampling. Cryptology ePrint Archive, Report 2014/591, 2014. http://eprint.iacr.org/2014/591. 20

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039. 55

[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. 8, 14, 15

[TP17] Tsuyoshi Takagi and Thomas Peyrin, editors. *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany. 64, 65

[Unr17] Dominique Unruh. Post-quantum security of Fiat-Shamir. In Takagi and Peyrin [TP17], pages 65–95. 19

[Wal19] Michael Walter. Sampling the integers with low relative error. *IACR Cryptology ePrint Archive*, 2019:68, 2019. 39, 54

[ZSS18] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: fast, compact, and constant-time discrete gaussian sampler over integers. *IACR Cryptology ePrint Archive*, 2018:1234, 2018. 9, 20, 39, 54, 55

[ZWXZ18] Zhongxiang Zheng, Xiaoyun Wang, Guangwu Xu, and Chunhuan Zhao. Error estimation of practical convolution discrete gaussian sampling with rejection sampling. Cryptology ePrint Archive, Report 2018/309, 2018. `https://eprint.iacr.org/2018/309`. 39, 54