INFONET JournalClub

Seungmin Kim

03/27/2026

1 Brief Summary & Background

- This document summarizes the Falcon signature scheme specification, which explains how to build a compact and practical post-quantum digital signature system. Before starting research on a FalconVRF, I wanted to organize the overall structure and main ideas of Falcon itself

2 Introduction

- Falcon stands for *Fast Fourier Lattice-based Compact Signatures over NTRU*. Falcon is a lattice-based digital signature scheme designed to achieve two main goals: **small signature size** and **high computational efficiency**.
- At a high level, Falcon follows a hash-and-sign digital signature structure based on the **GPV (Gentry–Peikert–Vaikuntanathan) framework**. it combines the framework with **NTRU lattice structure** and **fast Fourier sampling** so that it can achieve both short signatures and efficient implementation:

$$\text{Falcon} = \text{GPV framework} + \text{NTRU lattices} + \text{Fast Fourier sampling}.$$

2.1 Falcon의 계보

- **NTRUSign** provided very short signatures, but it was vulnerable to secret key recovery attacks because it used a deterministic signing procedure.
- The **GPV framework** replaced this with a probabilistic hash-and-sign structure based on trapdoor sampling, and presented a design principle for lattice-based signatures with provable security.
- Later, through **Provable NTRUSign** and **studies on concrete constructions of GPV IBE**, the combination of GPV and NTRU gradually became more practical.
- Finally, with the introduction of **Fast Fourier Sampling**, Falcon was completed as a practical lattice-based digital signature scheme that achieves both short signature size and high efficiency.

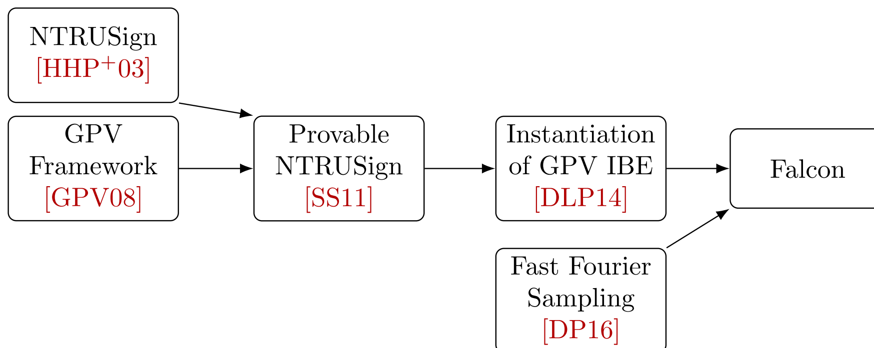

Figure 1: The genealogic tree of Falcon

3 The Design Rationale of Falcon

3.1 Design Principle for Compactness

- The design of Falcon starts from the observation that, when moving to post-quantum digital signatures, **communication complexity can become a bigger problem than computational speed itself**.
- In fact, many post-quantum digital signature schemes are computationally fast, but compared with traditional RSA- or discrete-log-based schemes, they often have much larger **public keys** or **signatures**.
- For this reason, Falcon takes the minimization of the following value as a main design principle:

$$|pk| + |sig|$$

In other words, the central goal of Falcon is to reduce the **sum of the public key size and the signature size**.

Three Design Choices The concrete design of Falcon is based on the following three choices.

- **First, choosing the GPV framework based on hash-and-sign** Lattice-based digital signatures can be broadly divided into the **Fiat–Shamir approach** and the **hash-and-sign approach**. **GPV framework**, which is a representative hash-and-sign method, is known to be secure not only in the classical random oracle model but also in the **quantum random oracle model**.
- **Third, choosing the lattice structure** For implementing the GPV framework in a compact way, the **NTRU lattice** is an almost optimal choice. The NTRU lattice is favorable for keeping both the public key and the signature small, and its structural properties also make it possible to perform many operations about **two orders of magnitude** faster.
- **Fourth, choosing the trapdoor sampler** Finally, Falcon introduces a new **trapdoor sampler**. This sampler is designed to have a speed comparable to the asymptotically fastest general trapdoor samplers, while also providing a security level comparable to the safest samplers.

3.2 The Gentry–Peikert–Vaikuntanathan Framework

- In 2008, Gentry, Peikert, and Vaikuntanathan proposed a general framework for constructing secure lattice-based digital signatures. This so-called **GPV framework** systematically explains *hash-and-sign* lattice-based signatures, and later became the theoretical foundation for many lattice-based signature schemes, including Falcon.
- At a very high level, the goal of the GPV framework is to **generate a short vector corresponding to a given hash value as a signature**. For this purpose, the public key provides a linear relation for verification, while the secret key acts as a trapdoor that helps find a **short solution (short preimage)** satisfying that relation.

The Roles of the Public Key and the Secret Key In the GPV framework, the public key and the secret key can be understood as follows.

- The public key is given as a full-rank matrix $A \in \mathbb{Z}_q^{n \times m}, (m > n)$ that generates a q-ary lattice Λ.
- On the other hand, the secret key includes a matrix $B \in \mathbb{Z}_q^{m \times m}$ that generates $\Lambda_q^\perp$. Here, $\Lambda_q^\perp$ means the lattice that is orthogonal to Λ modulo q.
- That is, for any $x \in \Lambda$ and $y \in \Lambda_q^\perp$, $\langle x, y \rangle = 0 \pmod q$ holds. Equivalently, this can be written as

$$BA^t = 0$$

What is a Signature? Given a message m, in the GPV framework, signing can be understood as the problem of finding a **short preimage** of the hash value $H(m)$.

- The hash function is given as $H : \{0,1\}^* \to \mathbb{Z}_q^n$.
- A signature $s \in \mathbb{Z}_q^m$ must satisfy $sA^t = H(m)$ and at the same time must be sufficiently short.
- Therefore, using only the public key A, the verifier only needs to:
  1. check whether s is short enough, and
  2. check whether $sA^t = H(m)$ actually holds.

How is a Signature Generated? The actual signing process can be understood in the following two steps.

1. First, the signer finds any preimage $c_0 \in \mathbb{Z}_q^m$ satisfying $c_0 A^t = H(m)$. This vector does not need to be short, and since $m > n$, it can be computed relatively easily using linear algebra.
2. Next, using the secret key B, the signer finds a vector $v \in \Lambda_q^\perp$ that is close to c_0. Then, by setting $s = c_0 - v$, we obtain a valid signature.

The reason is that, since $v \in \Lambda_q^\perp$, $vA^t = 0$ holds, and therefore

$$sA^t = (c_0 - v)A^t = c_0 A^t - vA^t = H(m) - 0 = H(m)$$

is satisfied.

Also, if v is close enough to c_0, then their difference s becomes a short vector, so it satisfies the signing requirement. In summary, the main idea of the GPV framework is the following.

> First, find one preimage of the hash value, and then adjust it along the orthogonal lattice direction to obtain a **short solution (short preimage)**.

In other words, the public key is used to verify whether "this vector is a correct solution," while the secret key acts as a trapdoor that provides "the direction that makes the solution sufficiently short."

3.2.1 Difference from Earlier Approaches: From Deterministic Rounding to Randomized Sampling

- The basic idea of the GPV framework is not completely new. In fact, GGH and NTRUSign also used an approach where one first finds a preimage of the hash value and then adjusts it with an appropriate lattice vector to produce a short signature.
- In GGH and NTRUSign, the correction vector is obtained using a deterministic *round-off algorithm.* This method is based on Babai's nearest plane idea: after expressing the preimage c_0 as a real linear combination of the basis B, each coefficient is rounded, and one computes

  $$v \leftarrow \lfloor c_0 B^{-1} \rceil B$$

  in this way.
- The advantage of this method is that it can control the signature size relatively well. However, because the signature always lies in a specific structure determined by the secret basis B, information about the basis may gradually leak when many signatures are observed.
- **The GPV approach** Instead of deterministic rounding, the GPV framework uses **trapdoor sampling** based on Klein's randomized nearest plane algorithm.
- That is, in GPV, the short vector is not chosen as a single "nearest" lattice point. Instead, it is **randomly sampled** according to a suitable distribution.

- As a result, the signature follows a Gaussian distribution over the shifted lattice $c_0 + \Lambda_q^{\perp}$, and it no longer directly reflects a specific structure of the secret basis. This is exactly the key reason why GPV, unlike GGH or NTRUSign, can support a discussion of **provable security**.
- In summary, the difference between GGH and NTRUSign on the one hand, and GPV on the other hand, is not simply whether they can produce a short signature, but **how that short vector is chosen**.
- GGH and NTRUSign suffer from structural information leakage because of deterministic rounding, whereas GPV introduces randomized trapdoor sampling and achieves both **short signatures and security**.
- Falcon can be understood as a scheme built on this GPV line of work, while adopting the more efficient **fast Fourier sampling** technique to achieve practical performance.

3.2.2 Statefulness, Determinization, and Hash Randomization

- When the GPV framework is used as a practical digital signature scheme, there is one important point to be careful about. Two different signatures s and s' for the same hash value $H(m)$ should not both be revealed.
- If multiple signatures for the same hash input are exposed, then a condition needed for the security proof of the GPV framework may fail. Therefore, some additional method is needed to prevent this.

Possible Solutions There are three main ways to solve this problem.

- **1) Stateful approach** The signer keeps a record of all previously signed messages and their corresponding signatures, and makes sure not to sign the same input again. This is theoretically possible, but in real systems it creates the problem of securely storing and synchronizing the state.
- **2) Determinization** Another method is to make the originally randomized signing procedure fully deterministic, so that the same message always produces the same signature. This is also conceptually natural, but it requires all implementations to generate pseudorandomness in exactly the same way.
- **3) Hash randomization** Instead of hashing the message directly, one can prepend a sufficiently long random salt and then hash the result. That is, instead of computing $H(m)$ directly for a message m, one generates a random salt $r \in \{0,1\}^k$ and computes $H(r\|m)$ instead. If k is large enough, then the probability of repeating the same hash input for the same message becomes extremely small.

Falcon's Choice

- Among these three options, Falcon adopts **hash randomization**.
- More specifically, Falcon generates a random salt of length 320 bits and prepends it to the message before hashing.
- This 320-bit length matches the value $\lambda + \log_2(q_s)$ based on the highest NIST security level, $\lambda = 256$, and the maximum number of signing queries per signer, $q_s = 2^{64}$.
- Of course, for lower security levels, this length may be somewhat conservative. However, using the same salt length for all security levels simplifies the API design and implementation, and it also has the advantage that users can process messages in the same way even if they do not know the exact security level of the secret key.
- Therefore, Falcon's choice can be seen as a result of considering both **theoretical security** and **practical ease of implementation**.

3.3 NTRU Lattices

- When implementing the GPV framework in practice, the first important question is **what kind of lattice to use**. This choice directly affects the efficiency, key size, and security of the whole signature scheme.
- For example, if one wants to prioritize security over structural simplicity and avoid adding special algebraic structure as much as possible, then using **standard lattices** without extra algebraic structure can be a natural choice. Schemes such as Frodo follow this approach.
- However, the main design principle of Falcon is **compactness**. In other words, one of its most important goals is to keep the public key and the signature as small as possible.
- For this reason, Falcon adopts the **NTRU lattice** proposed by Hoffstein, Pipher, and Silverman.

Advantages of NTRU Lattices

- NTRU lattices have an additional **ring structure**, and because of this structure, they can be represented much more compactly than standard lattices.
- More specifically, this structure can reduce the public key size by about a factor of $O(n)$, and it also helps speed up several operations by at least a factor of $O(n/\log n)$.
- Furthermore, even within the broader class of lattices over rings, NTRU lattices provide a particularly compact structure. In fact, Falcon's public key can be represented by just a single polynomial of degree at most $n-1$:

$$h \in \mathbb{Z}_q[x]$$

- NTRU lattices also provide practical confidence because they have withstood extensive cryptanalysis for more than about 20 years.
- Falcon makes use of these advantages of NTRU lattices, while choosing its parameters carefully to achieve additional robustness.

3.3.1 Overview of NTRU Lattices

- First, for a power of two $n = 2^\kappa$, define the polynomial $\phi = x^n + 1$, and let $q \in \mathbb{N}^*$ be the modulus.
- The NTRU secret key consists of four polynomials f, g, F, G in the ring $\mathbb{Z}[x]/(\phi)$, and they satisfy the following NTRU equation:

$$fG - gF = q \bmod \phi \tag{1}$$

- If f is invertible modulo q, then the public polynomial can be defined as

$$h \leftarrow g \cdot f^{-1} \bmod q$$

  and in general, h is used as the public key, while f, g, F, G are used as the secret key.
- In this case, the following two matrices generate the same lattice:

$$\begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} f & g \\ F & G \end{bmatrix}.$$

- However, the two representations play different roles. The first matrix is the public-key representation and contains the relatively large polynomial h and the modulus q, while the second matrix provides a secret basis made of small polynomials. Because of this property, efficient computations using the secret basis become possible.

- Also, if f and g are generated with sufficient entropy, then the public key h appears pseudorandom. In other words, it is expected to be difficult to recover the small polynomials f and g that generate h from the public value h alone.
- More specifically, for a given public key h, the problem of finding small polynomials f', g' satisfying
$$h = g'(f')^{-1} \bmod q$$
is still believed to be hard, and this computational difficulty is exactly the core of the NTRU assumption.

3.3.2 Combination with the GPV Framework

In Falcon, this can be interpreted as follows.

- **Public basis** The public basis can be written as $A = [1 \;\; h]$, which is essentially equivalent to knowing the public polynomial h.
- **Secret basis** The secret basis is defined as
$$B = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix} \tag{2}$$
- **Orthogonality** In this setting, A and B have an orthogonal structure, and they satisfy
$$B \cdot A^\star = 0 \bmod q$$
Here, $A^\star$ can be understood as an operation corresponding to a suitable conjugation or transpose.
- **Signature form** A signature on a message m is represented as a pair consisting of a salt r and two polynomials (s_1, s_2), and it must satisfy the following relation:
$$s_1 + s_2 h = H(r\|m),$$
where H is the hash function, and $r\|m$ means the concatenation of the salt and the message.
  - Once s_2 is chosen,
$$s_1 = H(r\|m) - s_2 h$$
determines s_1 automatically.
  - Therefore, in actual transmission, s_1 does not need to be included separately.
  - As a result, the actual Falcon signature can be sent in the form
$$(r, s_2)$$

3.3.3 Quasi-Optimal Parameter Choice

Falcon's trapdoor sampler essentially generates signatures whose size is proportional to the largest length among the orthogonal vectors obtained by applying Gram–Schmidt orthogonalization to the vectors of the secret basis B, namely $\|B\|_{GS}$. Therefore, in order to obtain short and efficient signatures, it is important to make $\|B\|_{GS}$ as small as possible. From this viewpoint, Falcon's parameter choice can be understood as follows.

- **Empirical optimum:** $\|(f, g)\| \approx 1.17\sqrt{q}$
Earlier studies provided empirical and experimental evidence that $\|B\|_{GS}$ is minimized when
$$\|(f, g)\| \approx 1.17\sqrt{q}.$$
Based on this, Falcon generates f and g from a discrete Gaussian distribution centered at 0 over $\mathbb{Z}[x]/(\phi)$, with parameters chosen so that the expected value of $\|(f, g)\|$ is approximately $1.17\sqrt{q}$.

- **Selecting a good basis**
  For generated f and g, the value of $\|B\|_{GS}$ can be computed very efficiently. If this value is larger than
  $$1.17\sqrt{q},$$
  then Falcon generates new f and g and repeats the whole process. In other words, Falcon does not simply use any random NTRU secret key. Instead, it refines the parameters by selecting only those "good" bases whose Gram–Schmidt norm is sufficiently small.

- **Meaning of quasi-optimality**
  This choice can in fact be regarded as quasi-optimal. For any basis B of the form in (2), if f, g, F, G satisfy (1), then
  $$\det(B) = fG - gF = q$$
  holds. Therefore, $\sqrt{q}$ becomes a theoretical lower bound for $\|B\|_{GS}$.

  - The theoretically optimal value is on the order of $\sqrt{q}$.
  - The value actually achieved by Falcon is
    $$\|B\|_{GS} \leq 1.17\sqrt{q}.$$
  - Therefore, Falcon's choice is within only a factor of 1.17 from the theoretical lower bound, which means that its parameter choice is very close to optimal.

3.4 Fast Fourier Sampling

When implementing the GPV framework in practice, the second important component to decide is the *trapdoor sampler*. A trapdoor sampler is an algorithm that takes a matrix A, a trapdoor T, and a target value c as input, and outputs a short vector s satisfying

$$s^t A = c \pmod{q}.$$

this is equivalent to finding a vector $v \in \Lambda_q^{\perp}$ that is close to c_0. Therefore, in this document, the term trapdoor sampler refers to the whole algorithm that performs these two tasks.

When designing or choosing a trapdoor sampler, the following criteria are important.

- **Efficiency**: the time and space complexity required for sampling
- **Quality**: how short the output vector s is, or equivalently, how close v is to c_0

In general, the shorter the output vector is, the better it is for both signature size and security. To satisfy these requirements, Falcon adopts the *fast Fourier nearest plane* approach proposed by Ducas and Prest. The main features of this method are as follows.

- **A structure specialized for lattices over rings.** Fast Fourier nearest plane is a sampling method that directly uses the lattice structure over rings.
- **A recursive computation structure.** Since this algorithm has a recursive structure similar to the fast Fourier transform, it can perform sampling efficiently.
- **Both efficiency and quality.** When randomized, this method can achieve both high-quality sampling that outputs short vectors and fast computational efficiency.
- **Compatibility with NTRU lattices.** This method works well with the NTRU lattice structure, allowing Falcon to implement practical trapdoor sampling while still keeping its compact key and signature structure.

In summary, Falcon's fast Fourier sampling is not just a technique for making sampling faster. It is a key component that simultaneously satisfies compatibility with NTRU lattices, sampling efficiency, and the need for short output vectors.

3.4.1 Choice of the Standard Deviation σ

Another important parameter when using a trapdoor sampler is the standard deviation σ. The choice of σ is directly related to the balance between preventing information leakage and keeping the output vector short. This can be summarized as follows.

- **If σ is too small,** the output distribution of the sampler does not hide the structure of the secret basis B well enough. In particular, in the extreme case $\sigma = 0$, the output becomes close to deterministic, which increases the possibility of leaking secret information through attacks such as learning-based attacks.
- **If σ is too large,** the sampler outputs unnecessarily large vectors. This increases the signature length and, as a result, can reduce both the efficiency and the security of the signature scheme.
- Therefore, σ must be chosen large enough to suppress leakage of the secret basis, but not so large that the output vector becomes too big.

Falcon's fast Fourier sampler is similar to Klein's sampler in several ways, and the choice of σ follows the same principle. Therefore, Falcon sets the standard deviation as

$$\sigma = \eta_\varepsilon(\mathbb{Z}^{2n}) \cdot \|B\|_{GS}.$$

Here, $\eta_\varepsilon(\mathbb{Z}^{2n})$ is the smoothing parameter, and $\|B\|_{GS}$ is the Gram–Schmidt norm of the secret basis B. In other words, Falcon uses a standard deviation proportional to $\|B\|_{GS}$, which reflects the "shape" of the secret basis.

4 Overview of Falcon

The main elements of Falcon are polynomials of degree n with integer coefficients. Here, n is usually a power of two, and the typical choices are 512 or 1024. All computations are performed modulo a monic polynomial ϕ of degree n, and in Falcon this polynomial is always

$$\phi = x^n + 1.$$

The overall structure of Falcon can be summarized as follows.

- **Correspondence between polynomials and matrices**
  Mathematically, Falcon interprets some polynomials as vectors and some as matrices. More specifically, a polynomial f modulo ϕ corresponds to an $n \times n$ square matrix, whose rows are

  $$x^i f \bmod \phi, \qquad i = 0, \ldots, n-1.$$

  Under this interpretation, matrix addition and multiplication correspond exactly to polynomial addition and multiplication modulo ϕ. Therefore, although Falcon is really handling matrices that define a lattice, most operations can be written more simply in terms of polynomial operations.

- **Public key**
  The public key of Falcon is given as a basis of a lattice of dimension $2n$, and it is written as

  $$\begin{bmatrix} -h & I_n \\ qI_n & 0_n \end{bmatrix} \tag{3}$$

  Here, I_n is the n-dimensional identity matrix, 0_n is the zero matrix, and h is a polynomial modulo ϕ that represents an $n \times n$ submatrix in the way explained above. The coefficients of h are integers between 0 and $q-1$, and q is the small prime used in Falcon. In the recommended parameter set,

  $$q = 12289.$$

- **Secret key**
The corresponding secret key is another basis of the same lattice, given in the form

$$\begin{bmatrix} g & -f \\ G & -F \end{bmatrix} \tag{4}$$

where f, g, F, G are short integer polynomials modulo ϕ, and they satisfy the following two relations:

$$h = g/f \bmod \phi \bmod q, \qquad fG - gF = q \bmod \phi. \tag{5}$$

A lattice of this form is called a *complete NTRU lattice*, and the second equation is especially called the *NTRU equation*. One important point is that the relation $h = g/f$ holds modulo q, but the lattice itself and the polynomial coefficients are fundamentally defined over unrestricted integers.

- **Key generation**
In the key generation process, Falcon first generates f and g randomly from a suitable distribution. These polynomials must be short enough, but at the same time they must not be too short. After that, Falcon solves the NTRU equation to find the corresponding F and G. The key format is described in Section 3.4, and the key generation procedure is explained in detail in Section 3.8.

- **Signature generation**
The signature generation process starts by hashing the message together with a random nonce to obtain a polynomial c modulo ϕ. The coefficients of c are mapped uniformly to integers between 0 and $q-1$. This process is explained in Section 3.7. Then, using the secret lattice basis (f, g, F, G), the signer generates a short polynomial pair (s_1, s_2) satisfying

$$s_1 = c - s_2 h \bmod \phi \bmod q. \tag{6}$$

In Falcon, the value actually transmitted as the signature is s_2.

- **Fast sampling**
In general, finding short vectors s_1, s_2 is computationally expensive. Falcon uses the special structure of $\phi = x^n + 1$ to implement this process with a divide-and-conquer algorithm similar to the fast Fourier transform, which greatly speeds up the computation. In addition, to prevent the signature from leaking too much information about the secret key, Falcon adds noise during the sampling process according to a carefully tuned Gaussian distribution.

- **Signature verification**
The signature verification process recomputes s_1 from the hashed message c and the signature s_2, and then checks whether (s_1, s_2) is a sufficiently short vector. Signature verification can be performed entirely using integer arithmetic modulo q.

4.1 Key Pair Generation

4.1.1 Overview

The key pair generation process can be divided into two main steps.

- **Step 1: Solving the NTRU equation**
The first step of key generation is to compute polynomials

$$f, g, F, G \in \mathbb{Z}[x]/(\phi)$$

so that they satisfy the NTRU equation. Here, generating f and g is relatively easy, but efficiently computing F and G that satisfy the equation is the main technical challenge.

To do this, Falcon uses the *tower-of-rings structure*. More specifically, it uses the field norm N to map the original NTRU equation into a smaller ring $\mathbb{Z}[x]/(\phi')$, and repeats this process until the problem is finally reduced to one over $\mathbb{Z}$. Then, after solving it in the integer ring using the extended Euclidean algorithm, Falcon lifts the solution (F, G) back to the original ring $\mathbb{Z}[x]/(\phi)$ by using properties of the norm.

During this process, no modular reduction by q is performed, so in the lower levels of the recursion one may have to deal with polynomials whose coefficients are very large. Therefore, the implementation must be able to handle large integer arithmetic in a stable way.

- **Step 2: Computing the Falcon tree**
  Once suitable f, g, F, G have been generated, the second step preprocesses them into a form suitable for signature generation. In Falcon, this preprocessing result is called the **Falcon tree**. The Falcon tree is a relatively compact data structure that enables fast sampling during signature generation.

  For this, Falcon first defines the following matrix:

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

  Then, for the Gram matrix

  $$\mathcal{G} = BB^{\star},$$

  it computes the LDL* decomposition

  $$\mathcal{G} = LDL^{\star}.$$

  This essentially corresponds to the Gram–Schmidt orthogonalization of B.

  Instead of storing only L, Falcon applies splitting operators in order to make use of the tower-of-rings structure. In other words, Falcon decomposes the diagonal entries of D into smaller subproblems over smaller rings, and recursively performs LDL* decomposition on each subproblem, thereby forming a tree structure. This recursion continues until it reaches the base level, where the coefficients belong to the rational field $\mathbb{Q}$.

  The main technical features of this step are as follows.

  - It uses the tower structure of the ring $\mathbb{Q}[x]/(\phi)$ to break the problem into smaller rings.
  - Since intermediate results are stored in a tree structure, one must carefully manage the fact that elements at different levels belong to different rings.
  - For better performance, the whole process is carried out in the FFT domain.

After these two steps are completed, the leaf nodes of the LDL tree are normalized to obtain the final Falcon tree. The resulting keys are then given by

$$sk = (\widehat{B}, T), \qquad pk = h = gf^{-1} \bmod q.$$

In other words, Falcon key generation can be understood as first constructing the NTRU secret basis through `NTRUGen`, and then computing the Falcon tree suitable for signature generation through `ffLDL`*. This whole process is summarized by the following `Keygen` algorithm.

Algorithm 1 Keygen(ϕ, q)

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

1: $f, g, F, G \leftarrow \text{NTRUGen}(\phi, q)$ ▷ Solve the NTRU equation
2: $B \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$
3: $\widehat{B} \leftarrow \text{FFT}(B)$ ▷ Apply FFT to each of the four components $\{g, -f, G, -F\}$
4: $\mathcal{G} \leftarrow \widehat{B}\widehat{B}^{\star}$
5: $T \leftarrow \text{ffLDL}^{\star}(\mathcal{G})$ ▷ Compute the LDL* tree
6: **for** each leaf $\in T$ **do**
7: $\quad$ $leaf.value \leftarrow \sigma/\sqrt{leaf.value}$ ▷ Normalization step
8: **end for**
9: $sk \leftarrow (\widehat{B}, T)$
10: $h \leftarrow gf^{-1} \bmod q$
11: $pk \leftarrow h$
12: **return** sk, pk

4.2 Signature Generation

4.2.1 Overview

Falcon's signature generation procedure produces a short polynomial pair (s_1, s_2) for a message m such that

$$s_1 + s_2 h = c \pmod q$$

holds. Here, c is a polynomial obtained by hashing the message together with a salt, and h is the public key.

The basic flow of signature generation is as follows.

1. First, a random salt $r \in \{0,1\}^{320}$ is generated uniformly, and the string $(r \| m)$ is hashed to obtain
$$c \in \mathbb{Z}_q[x]/(\phi).$$

2. Next, using the secret key, Falcon computes a preimage t of c. However, simply rounding t coefficient by coefficient to obtain a short vector is not secure, because it may leak information about the secret key.

3. Instead, Falcon uses **fast Fourier sampling**, which is its trapdoor sampler. This algorithm uses the Falcon tree T, which was precomputed during key generation, and performs randomized sampling around t according to a discrete Gaussian distribution.

4. Using the sampled vector z, Falcon computes
$$s = (t - z)\widehat{B},$$
which gives a signature vector following a Gaussian distribution. Then, by applying the inverse FFT, Falcon recovers (s_1, s_2).

5. Finally, Falcon checks whether $\|s\|^2$ is within the allowed bound. If the condition is satisfied, it compresses s_2 and outputs the final signature
$$\text{sig} = (r, s).$$

The most important part of this process is ffSampling, which is a recursive trapdoor sampler that performs adaptive randomized rounding using the Falcon tree. In contrast, the hash computation and compression are relatively simple parts of the implementation.

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

Require: A message m, a secret key sk, a bound $\lfloor\beta^2\rfloor$
Ensure: A signature sig of m

1: $r \leftarrow \{0,1\}^{320}$ uniformly
2: $c \leftarrow \textsc{HashToPoint}(r \| m, q, n)$
3: $t \leftarrow \left(-\frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(F),\ \frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(f)\right)$ ▷ $t = (\mathrm{FFT}(c), \mathrm{FFT}(0)) \cdot \widehat{B}^{-1}$
4: **repeat**
5:     **repeat**
6:         $z \leftarrow \textsc{ffSampling}_n(t, T)$
7:         $s \leftarrow (t - z)\widehat{B}$ ▷ At this point, $s \sim D_{(c,0)+\Lambda(B),\sigma,0}$
8:     **until** $\|s\|^2 \le \lfloor\beta^2\rfloor$
9:     $(s_1, s_2) \leftarrow \mathrm{invFFT}(s)$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
10:     $s \leftarrow \textsc{Compress}(s_2, 8 \cdot \mathit{sbytelen} - 328)$ ▷ Remove 1 byte for header and 40 bytes for r
11: **until** $s \ne \bot$
12: **return** sig $= (r, s)$

4.2.2 Fast Fourier Sampling

Basic Idea

- The goal of `ffSampling` is, given the input
$$t = (t_0, t_1) \in \mathrm{FFT}\big(\mathbb{Q}[x]/(x^n+1)\big)^2$$
and a Falcon tree T, to output
$$z = (z_0, z_1) \in \mathrm{FFT}\big(\mathbb{Z}[x]/(x^n+1)\big)^2.$$
- Here, z is not a simple rounding result. Instead, it is a **value adaptively sampled according to a discrete Gaussian distribution** using the information stored in the Falcon tree.
- In other words, `ffSampling` is Falcon's core sampling procedure that
  - keeps the signature vector sufficiently short, while
  - preventing the secret basis information from being revealed.
- The algorithm works in a **divide-and-conquer** manner.
  - It splits a degree-n problem into two degree-$n/2$ problems,
  - solves them recursively, and then
  - combines the results to obtain a solution for the original degree.

Base Case: $n = 1$

- In the smallest case $n = 1$, the problem can no longer be divided recursively.
- In this case, Falcon uses the value stored in the current node of the Falcon tree,
$$\sigma' = T.\mathrm{value},$$
as the standard deviation, and applies the discrete Gaussian sampler over the integers, `SamplerZ`, separately to t_0 and t_1.
- Since t_i and z_i are essentially a real number and an integer in this case, the algorithm reduces to **one-dimensional Gaussian randomized rounding**.
- More specifically, it computes
$$z_0 \leftarrow \mathrm{SamplerZ}(t_0, \sigma'), \qquad z_1 \leftarrow \mathrm{SamplerZ}(t_1, \sigma')$$
and then returns (z_0, z_1).

Recursive Step

- When $n > 1$, the current node of the Falcon tree consists of

$$(\ell, T_0, T_1) = (T.\text{value},\ T.\text{leftchild},\ T.\text{rightchild}).$$

- Here,
  - ℓ is the nontrivial LDL* component at the current step, and
  - T_0 and T_1 are the Falcon trees for the left and right subproblems, respectively.
- The algorithm first processes the **second component** t_1.
  - It applies `splitfft` to t_1 to decompose it into two degree-$n/2$ problems,
  - recursively runs $\texttt{ffSampling}_{n/2}$ using the right child tree T_1, and
  - combines the result with `mergefft` to obtain z_1.
- Next, it updates the target value of the first component as

$$t_0' = t_0 + (t_1 - z_1) \odot \ell.$$

- The meaning of this equation is the following.
  - The sample z_1 already chosen in the second component
  - affects the conditional distribution of the first component,
  - so t_0 must be adjusted to reflect this.
- Then, for the updated t_0',
  - Falcon applies `splitfft`,
  - recursively calls the sampler using the left child tree T_0, and
  - applies `mergefft` to recover z_0.
- Finally, it returns

$$z = (z_0, z_1).$$

4.2.3 Sampler over the Integers

- Falcon's fast Fourier sampling requires **discrete Gaussian sampling over the integers** at the base level of the recursion.
- For this purpose, Falcon uses `SamplerZ`. This algorithm is designed to generate an integer sample z that is very close to the distribution

$$D_{\mathbb{Z},\sigma',\mu}$$

  for any

$$\sigma' \in [\sigma_{\min}, \sigma_{\max}], \qquad \mu \in \mathbb{R}.$$

- Also, for resistance against timing attacks, it is important to implement this process as *isochronously* as possible.

Algorithm 3 $\texttt{ffSampling}_n(t, T)$

Require: $t = (t_0, t_1) \in \mathrm{FFT}(\mathbb{Q}[x]/(x^n + 1))^2$, Falcon tree T
Ensure: $z = (z_0, z_1) \in \mathrm{FFT}(\mathbb{Z}[x]/(x^n + 1))^2$
1: **if** $n = 1$ **then**
2: $\quad \sigma' \leftarrow T.\text{value}$
3: $\quad z_0 \leftarrow \text{SamplerZ}(t_0, \sigma')$
4: $\quad z_1 \leftarrow \text{SamplerZ}(t_1, \sigma')$
5: $\quad$ **return** $z = (z_0, z_1)$
6: **end if**
7: $(\ell, T_0, T_1) \leftarrow (T.\text{value}, T.\text{leftchild}, T.\text{rightchild})$
8: $t_1 \leftarrow \text{splitfft}(t_1)$
9: $z_1 \leftarrow \text{ffSampling}_{n/2}(t_1, T_1)$
10: $z_1 \leftarrow \text{mergefft}(z_1)$
11: $t'_0 \leftarrow t_0 + (t_1 - z_1) \odot \ell$
12: $t_0 \leftarrow \text{splitfft}(t'_0)$
13: $z_0 \leftarrow \text{ffSampling}_{n/2}(t_0, T_0)$
14: $z_0 \leftarrow \text{mergefft}(z_0)$
15: **return** $z = (z_0, z_1)$

Components

- `SamplerZ` is not a single simple procedure. Instead, it has a **hierarchical structure** made of the following three components.
  - `BaseSampler`
  - `ApproxExp`
  - `BerExp`
- The role of each component is as follows.
  - `BaseSampler`: samples an integer from a fixed reference distribution.
  - `ApproxExp`: approximately computes the exponential probability needed for rejection sampling.
  - `BerExp`: performs Bernoulli accept/reject according to the approximated exponential probability.
- In other words, `SamplerZ` first draws a sample from the reference distribution,
- and then accepts or rejects it with an appropriate probability,
- so that the final output is close to the target distribution.

How It Works

- The overall flow of `SamplerZ` can be understood as follows.
  1. It separates the center μ into its integer part and fractional part.
  2. It samples a value from the reference distribution using `BaseSampler`.
  3. It extends this value to either the positive or negative direction to form a candidate integer z.
  4. It computes an acceptance probability that reflects the difference between the candidate value and the target distribution.
  5. It uses `BerExp` to probabilistically accept or reject the candidate.
  6. Finally, it adds back $\lfloor \mu \rfloor$ to recover a sample centered at μ.

Significance

- In summary, `SamplerZ` is an algorithm that uses a **reference distribution + rejection sampling** structure to generate samples that are very close to

$$D_{\mathbb{Z},\sigma',\mu}.$$

- In Falcon, this procedure is used at the **lowest level** of fast Fourier sampling.
- Therefore, `SamplerZ` can be seen as a key component that
  - guarantees the correctness of the whole signature generation process,
  - does not directly expose the secret basis, and
  - enables secure discrete Gaussian sampling.

4.3 Signature Verification

The signature verification procedure in Falcon is much simpler than key generation or signature generation. Given a public key $pk = h$, a message m, a signature $\mathrm{sig} = (r, s)$, and an acceptance bound $\lfloor \beta^2 \rfloor$, the verifier performs the following four steps.

1. First, the verifier concatenates the salt r and the message m to form $(r\|m)$, and inputs this into `HashToPoint` to compute
$$c \in \mathbb{Z}_q[x]/(\phi).$$
2. Next, the compressed bitstring s contained in the signature is restored by `Decompress` into the polynomial
$$s_2 \in \mathbb{Z}[x]/(\phi).$$
3. Then, using the public key h, the verifier computes
$$s_1 = c - s_2 h \pmod q.$$
4. Finally, the verifier checks the length of the vector (s_1, s_2). If
$$\|(s_1, s_2)\|^2 \le \lfloor \beta^2 \rfloor,$$
then the signature is accepted; otherwise, it is rejected.

In other words, Falcon's verification procedure can be understood as recomputing the hash value c, reconstructing s_1 from the recovered s_2, and then checking whether (s_1, s_2) is sufficiently short.

Algorithm 4 Verify$(m, \mathrm{sig}, pk, \lfloor\beta^2\rfloor)$

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

```
 1: c ← HashToPoint(r ‖ m, q, n)
 2: s2 ← Decompress(s, 8 · sbytelen − 328)
 3: if s2 = ⊥ then
 4:     reject                              ▷ Reject invalid encodings
 5: end if
 6: s1 ← c − s2h mod q                      ▷ s1 should be normalized between ⌈−q/2⌉ and ⌊q/2⌉
 7: if ‖(s1, s2)‖² ≤ ⌊β²⌋ then
 8:     accept
 9: else
10:     reject                              ▷ Reject signatures that are too long
11: end if
```

5 Security

5.1 Known Attacks

Key Recovery Attack The most efficient known attack against Falcon is a key recovery attack based on lattice reduction. This attack starts by considering the lattice generated by the columns of the following matrix:

$$\begin{bmatrix} q & h \\ 0 & 1 \end{bmatrix}.$$

After performing lattice reduction on this basis, one can enumerate lattice points inside a ball of radius $\sqrt{2n}\sigma_{\{f,g\}}$. With significant probability, this reveals the secret vector $\begin{bmatrix} g \\ f \end{bmatrix}$.

Let λ be the $(2n - B)$-th Gram–Schmidt norm. Then, a sieve algorithm on the projected lattice can recover vectors shorter than $\sqrt{4/3}\lambda$. Therefore, if

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

holds, then Babai's nearest plane algorithm can recover the secret key with high probability. For DBKZ, λ is given by

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

so the condition for successful key recovery can be written as

$$\left(\frac{B}{2\pi e}\right)^{1-\frac{n}{B}} \sqrt{q} = \sqrt{\frac{3}{4}B}\sigma_{\{f,g\}} \tag{7}$$

Signature Forgery Attack Signature forgery can be reduced to the problem of finding a lattice point within distance β from an arbitrary point. Using Kannan's embedding, one obtains the following matrix:

$$\begin{bmatrix} q & h & H(r\|m) \\ 0 & 1 & 0 \\ 0 & 0 & K \end{bmatrix}.$$

A sieve algorithm can then find a short vector of the form $(c, *, K)$, and in that case $H(r\|m) - c$ becomes the desired lattice point. When $K \approx \sqrt{q}$, the condition for successful forgery becomes

$$\left(\frac{B}{2\pi e}\right)^{\frac{n}{B}} \sqrt{q} \leq \beta \tag{8}$$

The paper considers this attack to be the optimal forgery attack against Falcon instances, and evaluates the bit security according to the core-SVP methodology as follows:

$$\text{Classical: } \lfloor 0.292B \rfloor, \qquad \text{Quantum: } \lfloor 0.262B \rfloor.$$

Concrete Attack Costs For Falcon-512, the optimal attack corresponds to BKZ with block size $B = 411$, and the sieve dimension is reduced to $B' = 374$. The resulting total classical cost is about 2^{120} operations, and the minimum gate count is conservatively estimated to be at least 2^{143}. For Falcon-1024, the key recovery attack is slightly more efficient, corresponding to $B = 936$ and $B' = 869$. Accordingly, the total classical cost is estimated to be at least 2^{264} operations, with a gate count of at least 2^{287}. The quantum cost is also considered to be much higher than brute-force key search against AES-128 or AES-256.

Hybrid Attack A hybrid attack combines the meet-in-the-middle technique with the key recovery attack. This attack was effective for NTRU schemes using sparse polynomials, but in Falcon the polynomials are not sparse, so its effectiveness is much more limited.

Dense, High-Rank Sublattice Attack Recent studies showed that NTRU-based schemes may become vulnerable when f, g are too small compared with q. However, Falcon is designed to resist such *overstretched NTRU* attacks by choosing f, g not too small and q not excessively large.

Algebraic Attack Falcon has a rich algebraic structure, but with currently known methods, this structure does not lead to an attack that is better than general lattice attacks by a factor of more than n^2. However, there do exist efficient algorithms for finding relatively large elements inside ideals of $\mathbb{Z}[x]/(\phi)$.

5.2 Precision of Floating-Point Computation

A trapdoor sampler generally requires floating-point computation, and Falcon's fast Fourier sampler is no exception. Therefore, an important practical question is: *how much precision is needed to maintain a meaningful security bound*? A naive analysis may suggest that the required precision is about $O(\lambda)$ bits, but this would make signature generation significantly slower.

To analyze this issue, Falcon uses Rényi divergence. As in $[MW17]$, let

$$a \lesssim b$$

mean that

$$a \leq b + o(b).$$

Falcon's fast Fourier sampler works recursively and depends on a total of $2n$ discrete samplers $D_{\mathbb{Z},c_j,\sigma_j}$. Here, assume that each c_j is given with absolute error δ_c, and each σ_j is given with relative error δ_σ. Let D denote the output distribution under infinite precision, and let $\bar{D}$ denote the output distribution under finite precision.

In this setting, one can reuse the precision analysis of Klein's sampler. For any output z that appears with non-negligible probability, the following worst-case bound holds:

$$\left|\log\left(\frac{\bar{D}(z)}{D(z)}\right)\right| \lesssim 2n\left[\frac{\sqrt{154}}{1.312}\delta_c + (2\pi+1)\delta_\sigma\right] \leq 20n(\delta_c + \delta_\sigma). \tag{9}$$

In the average case, the factor $2n$ in (9) can be replaced by $\sqrt{2n}$. Therefore, on average, one may regard the security loss as essentially negligible if

$$\delta_c + \delta_\sigma \leq 2^{-46}.$$

To check how well this condition is satisfied in practice, Falcon compared the values of c_j and σ_j obtained when running the implementation with 200-bit high precision and with standard 53-bit precision. The result was that

$$\delta_c + \delta_\sigma \leq 2^{-40}$$

always held. This is about 6 bits larger than the ideal target 2^{-46}, but the authors judge that this gap is not large enough to create a practical threat.

Therefore, Falcon concludes that 53-bit precision is sufficient for the parameter range required by NIST, namely security level $\lambda \leq 256$ and number of signing queries $q_s \leq 2^{64}$. In other words, the possibility that floating-point precision could leak information about the secret basis during signing is considered much closer to a theoretical possibility than to a practical attack vector.

6 Advantages and Limitations of Falcon

6.1 Advantages

Compactness Falcon's biggest advantage is its compactness. This is natural, because Falcon was originally designed with compactness as a main goal. Stateless hash-based signatures usually have small public keys but large signatures, while some multivariate signatures have very small signatures but large public keys. Lattice-based schemes can offer the advantages of both, and Falcon stands out because it keeps the total size $|pk| + |sig|$ very small.

Fast Signature Generation and Verification Falcon's signature generation and verification procedures are very fast. In particular, verification is highly efficient, and signature generation is also practical enough to produce more than 1000 signatures per second on a moderately powerful computer.

Security in ROM and QROM The theoretical foundation of Falcon, namely the GPV framework, has a security proof in the random oracle model (ROM), and its security was later also studied in the quantum random oracle model (QROM). In contrast, the security of the Fiat–Shamir heuristic in the QROM was established only relatively recently, and only under certain conditions. Therefore, Falcon also has a strength in terms of theoretical security.

Modular Design Falcon has a modular design. At present, it implements the GPV framework over NTRU lattices and uses fast Fourier sampling as the trapdoor sampler, but both of these components could in principle be replaced by other choices. For example, if one uses Klein's sampler instead of the fast Fourier sampler, signature generation becomes much slower, but the implementation becomes simpler, while the black-box security remains the same.

Message-Recovery Mode Falcon can also be implemented in message-recovery mode. In this case, the message m can be recovered from the signature. The signature length becomes about twice as long, but messages below a certain length can be fully recovered. Therefore, in some applications this can provide an even more competitive form of compactness.

Key-Recovery Mode Falcon can also be implemented in key-recovery mode. In this mode, the signature length doubles, but the public key can be reduced to a single hash value. As a result, the total size $|pk| + |sig|$ can be reduced by about 15%.

Extensibility to IBE Falcon can be transformed into Identity-Based Encryption (IBE) in a relatively direct way.

Easy Verification Falcon's verification procedure is very simple. Essentially, one only needs to compute

$$[H(r\|m) - s_2 h] \bmod q,$$

which can be summarized as a few NTT operations and one hash computation.

6.2 Limitations

Delicate Implementation Falcon's biggest drawback is that its implementation is delicate. In particular, the key generation process and fast Fourier sampling are not easy to understand, and they are also difficult to implement correctly. Still, since they rely on lower-level routines of the fast Fourier transform and tree structures, they are not made entirely of completely unfamiliar implementation components.

Dependence on Floating-Point Computation Falcon's signing procedure depends on 53-bit floating-point computation. This is not a serious problem in ordinary software environments, but it can be an important limitation for constrained devices that do not have floating-point hardware.

Side-Channel Resistance In the past, one of Falcon's limitations was that its side-channel resistance was not clearly established. This issue mainly came from the discrete Gaussian sampling over the integers. However, more recent work has proposed constant-time implementations for that step and for the whole scheme, so this problem has been largely reduced. A future challenge is to implement Falcon in a masked form.