Peregrine: Toward Fastest FALCON Based on GPV Framework*

Eun-Young Seo[1], Young-Sik Kim[1]**, Joon-Woo Lee[2], and Jong-Seon No[3]

[1] Department of Information and Communication Engineering, Chosun University, Gwangju, Republic of Korea
eunyoung00@gmail.com, iamyskim@chosun.ac.kr

[2] School of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea
jwlee2815@cau.ac.kr

[3] Department of Electrical and Computer Engineering, INMC, Seoul National University, Republic of Korea
jsno@snu.ac.kr

Abstract. FALCON and Crystals-Dilithium are the digital signatures algorithms selected as NIST PQC standards at the end of the third round. FALCON has the advantage of the shortest size of the combined public key and signature but has the disadvantage of the relatively long signing time. Since FALCON algorithm is faithfully designed based on theoretical security analysis, the implementation of the algorithms is quite complex and needs considerable complexity. In order to implement the FALCON algorithm, the isochronous discrete Gaussian sampling algorithm should be used to prevent the side-channel attack, which causes a longer signature time. Also, FFT operations with floating-point numbers should be performed in FALCON, and they cause difficulty in applying the masking technique, making it vulnerable to side-channel attacks. We propose the Peregrine signature algorithm by devising two methods to make the signing algorithm of the FALCON scheme efficient. To reduce the signing time, Peregrine replaces the discrete Gaussian sampling algorithm with the sampling algorithm from the centered binomial distribution in the key generation algorithm and the signing algorithm by adjusting the encryption parameters. Also, it replaces the fast Fourier transform (FFT) operations of floating-point numbers with the number theoretic transform (NTT) operations of integers represented in residue number system (RNS), making the scheme faster and easy to be applied with a masking technique to prevent the side channel attack.

Keywords: FALCON · GPV framework · centered binomial distribution · NTRU lattices · post-quantum cryptography.

* This work is submitted to 'Korean Post-Quantum Cryptography Competition' (www.kpqc.or.kr).

** Corresponding author: Young-Sik Kim

1 Introduction

Peregrine can be classified into the lattice-based signature scheme and uses the hash-and-sign scheme proposed by Gentry, Peikert, and Vaikuntanathan (GPV) [1]. It uses the NTRU lattice problem as the trapdoor and the centered binary distribution for the key generation and signing algorithm. In addition, it is designed to avoid the computation of the floating-point numbers in the signing algorithm to increase the convenience of implementation and to prevent the side channel attack.

$$\text{Peregrine} = \text{GPV framework} + \text{NTRU lattices} + \text{Centered Binomial r.v.}$$

1.1 Design Rationale

Hash-and-Sign Signature Scheme FALCON and Crystals-Dilithium are the digital signature algorithms selected as NIST PQC standards for digital signature, and both schemes are lattice-based digital signature algorithms. While Crystals-Dilithium follows the Fiat-Shamir method, FALCON uses the hash-and-sign method with the GPV framework. The lattice-based hash-and-sign method introduced in [1] is designed based on the hardness of the short integer solution (SIS) problem and proved to be secure in the quantum random oracle model. Like FALCON, Peregrine uses a lattice-based hash-and-sign method based on the GPV framework. We briefly describe the GPV framework as follows:

- A full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}, (m > n)$, where the rows of $\mathbf{A}$ are basis of a q-ary lattice Λ, is generated for the public key.
- A matrix $\mathbf{B} \in \mathbb{Z}_q^{m\times m}$, where its rows are orthogonal to the rows of matrix $\mathbf{A}$, is generated for the private key. The rows of matrix $\mathbf{B}$ are basis of a q-ary lattice $\Lambda^{\perp}$, which is an orthogonal lattice of Λ such as $\mathbf{B} \times \mathbf{A^t} = \mathbf{0}$.
- A hash function H is defined as $H : \{0,1\}^* \to \mathbb{Z}_q^n$. For a given message M, $H(M)$ is used for generating a signature of M.
- Find any vector $\mathbf{c}$ satisfying $\mathbf{c} \cdot \mathbf{A}^t = H(M)$.
- A valid short signature $\mathbf{s}$ can be computed from the private key $\mathbf{B}$ by finding a vector $\mathbf{v}$ which exists on the lattice $\Lambda^{\perp}$ such as $\mathbf{v} = \mathbf{v'} \cdot \mathbf{B}$, $\mathbf{v'} \in \mathbb{Z}_q^m$, and close to $\mathbf{c}$. Because $\mathbf{v} \cdot \mathbf{A^t} = \mathbf{0}$, the difference $\mathbf{s} = \mathbf{c} - \mathbf{v}$ also satisfies $\mathbf{s} \cdot \mathbf{A}^t = (\mathbf{c} - \mathbf{v}) \cdot \mathbf{A}^t = \mathbf{c} \cdot \mathbf{A^t} = \mathbf{H(M)}$.

The hash-and-sign method in the GPV framework uses the following core idea: it is difficult to find $\mathbf{s}$ with the small norm satisfying $\mathbf{s} \cdot \mathbf{A}^t = H(M)$, but it is easy for those who know the matrix $\mathbf{B}$, which is used for the secret key. The size of the public and private keys and the their performances are affected by the algebraic structures of the public and secret keys. If we use structured lattices with useful algebraic properties, the key size can be reduced, but the vulnerability can be introduced with the algebraic structures. Like the FALCON, Peregrine is a digital signature constructed from the hash-and-sign method based on this GPV scheme. In the Peregrine, lattices are created using the elements on the polynomial ring in generating public and secret keys, which will be explained in Section 3.

NTRU Lattices In this subsection, we can think about how to create an orthogonal private key $\mathbf{B}$, given a public key $\mathbf{A}$. FALCON uses NTRU lattices and has a compact structure with a short combined length of public key and signature compared to other digital signatures. Peregrine also generates public and secret keys using the same NTRU lattice for this advantage. However, Peregrine provides a more straightforward way to create NTRU lattices. NTRU was first introduced by Hoffstein, Piper, and Silverman [2]. Later, Stehlé and Steinfeld[3] demonstrated that NTRU lattices could be used in the GPV framework in a provably secure manner.

When public and private keys are generated using NTRU lattices, four polynomials $f,\ g,\ F,\ G \in \mathbb{Z}[x]/\phi(x)$ are required. Let $\phi(x) = x^n + 1$, $n = 2^k$, q be positive integers. f and g are randomly generated, and if f satisfies the invertible condition with the modulus of $(\phi,\ q)$, we can calculate

$$h = g \cdot f^{-1} \mod (\phi,\ q)$$

and use it as a public key polynomial. F and G can be found by finding a solution that satisfies the following NTRU equation using the given f and g.

$$f \cdot G - g \cdot F = q \mod \phi. \tag{1}$$

Note that in the NTRU equation in (1), after the the right hand-side is reduced using the modulus ϕ, only q should remain as a constant term. If we find F and G that satisfy this formula, we can use the above polynomials to create a secret key orthogonal to the public key, and we can get the desired result at the time of signing. Using these polynomials, create the matrices $\mathbf{A}$ and $\mathbf{B}$ as follows.

$$\mathbf{A} = \begin{pmatrix} 1, & h \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} g, & -f \\ G, & -F \end{pmatrix}.$$

Then, when $\mathbf{B} \cdot \mathbf{A^t}$ is calculated, it can be confirmed that they are orthogonal to each other as follows.

$$\mathbf{B} \cdot \mathbf{A^t} = \begin{pmatrix} g - h \cdot f, & G - h \cdot F \end{pmatrix} = \begin{pmatrix} 0, & 0 \end{pmatrix} \mod (\phi,\ q).$$

When generating arbitrary polynomials f, g, F, and G that satisfy the NTRU equation (1) in all cases and the generated f is not always invertible. Also, it is recommended to create F and G so that the norm of the generated row vectors of $\mathbf{B}$ is not too large. There is difficulty in generating polynomials that satisfy all these conditions. Specific conditions will be described in Section 3.

Centered Binomial Random Variable LWE, RLWE, and MLWE are problems defined on lattices and are proven to be secure when the random error is sampled from the discrete Gaussian distribution. However, sampling random errors from exact discrete Gaussian distribution is not only complicated to implement but also vulnerable to side-channel attacks. Thus, other than FALCON signature scheme, the lattice-based KEM and signature algorithms finally selected as the NIST PQC standard algorithms replaced the discrete Gaussian distribution with other distributions that can be easily sampled. Crystal-Dilithium signature scheme [20] uses the uniform distribution, and Crystal-Kyber KEM scheme [19] uses the centered binomial distribution. In the case of FALCON, it pursuits the provable security for rigorous security assurance, and thus it implements the isochronous discrete Gaussian distribution based on the works of Zhao, *et al.*, [4] and Howe *et al.*, [5]. However, these implementations are not constant-time implementation, are not easy to be applied with the masking technique, and have complicated implementation processes. Thus, each implementation step can be exposed to the side channel attack.

In FALCON, discrete Gaussian random variables are used in the key generation and signature processes. When FALCON creates polynomials f and g in the key generation process, discrete Gaussian random variables with variance $1.17\sqrt{q/2n}$ are used to make the value of $||\mathbf{B}||_{\mathrm{GS}}$ small. The value is selected experimentally in [6] and [7]. If f and g are generated whose polynomial coefficients have a discrete Gaussian distribution, it has been proven in [3] that h used in the public key is cryptographically secure. Peregrine utilizes centered binomial random variables close to the discrete Gaussian to perform faster with a similar security level. We experimentally confirm the successful derivation of F and G satisfying the NTRU equation (1) using the generated f and g.

In addition, in the signing process of FALCON, the discrete Gaussian random variables are used to introduce randomness for hiding the information of secret key $\mathbf{B}$. Fast Fourier sampling is used for theoretical security assurance, and discrete Gaussian random variables are used to add randomness to each sampling. Peregrine adds randomness to a signature using a centered binomial distribution instead of discrete Gaussian. In addition, it provides diverse parameters for security enhancement.

Signature Procedure without Floating-point Number Operation Peregrine does not use the Gram-Schmidt norm for $\mathbf{B}$ and the fast Fourier sampling when the signing process introduces some randomness. Therefore, it is designed to be able to sign using the residue number system (RNS) and the number theoretic transform (NTT) without using the fast Fourier transform (FFT) as in the FALCON. However, the implementation complexity is designed to be significantly reduced. A detailed explanation of this will be given in Section 2.

1.2 Advantages and Limitations

FALCON, a representative of the lattice-based hash-and-sign method, has the advantage of theoretical security assurance, but the signing speed is relatively slow. Above all, since the implementation complexity is relatively high, there is an entry barrier to use. Peregrine is an algorithm proposed to overcome this shortcoming of the FALCON and has the advantage of being a signature with a structure that is safe from side-channel attacks as it is simple to implement. It is possible to apply techniques such as making. However, the disadvantage is that it does not provide the same security assurance as other algorithms using uniform or centered binary distribution.

2 Preliminaries

The main idea of the lattice-based hash-and-sign signature is to find a point $\mathbf{v}$ on the lattice $\Lambda(\mathbf{B})$ close to $\mathbf{c}$ satisfying $\mathbf{c} \cdot \mathbf{A^t} = \mathbf{H}(\mathbf{m})$. Depending on the finding method, a distinct signature can be constructed. For example, a deterministic method to find $\mathbf{v}$ given $\mathbf{c}$ is Babai's round-off algorithm [8]. Let $\mathbf{v}$ on $\Lambda(\mathbf{B})$ be $\mathbf{v} = \mathbf{v_0} \cdot \mathbf{B}$. We can find a real value vector $\mathbf{c_0}$ satisfying $\mathbf{c_0} \cdot \mathbf{B} = \mathbf{c}$. If we generate signature $\mathbf{s} = \mathbf{c} - \mathbf{v}$ using $\mathbf{v}$ by assigning $\mathbf{v_0}$ to $\lfloor \mathbf{c_0} \rceil$, $\mathbf{s}$ is always inside of the parallelepiped $[-\frac{1}{2}, \frac{1}{2}]^m \times \mathbf{B}$, and thus we can assure the generation of signature with a small size of the norm, which is depicted in Fig. 2. If many signatures are generated and accumulated, the information on $\mathbf{s}$ in the parallelepiped is collected, which is a leak on the lattice $\mathbf{B}$ and can result in the key-recovery attacks [9],[10].

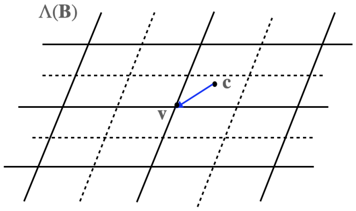

Fig. 1. Babai's round-off algorithm

Babai's nearest plane algorithm can make sure that $\mathbf{s}$ is in the parallelepiped $[-\frac{1}{2}, \frac{1}{2}]^m \times \tilde{\mathbf{B}}$ using the Gram-Schmidt orthogonalized basis $\tilde{\mathbf{B}}$. As in Fig 2, $\mathbf{B}$ is not directly exposed because the parallelepiped indicated by the dots is made of an orthogonal basis. However, since this is also a deterministic algorithm, there is a problem that information about the norm value of row vectors of $\tilde{\mathbf{B}}$ may leak if multiple signatures are generated.

Klein's algorithm[11] is proposed to solve the deterministic algorithm problem. Klein's algorithm is based on combining discrete Gaussian random variables with Babai's nearest plane algorithm. It is a secure algorithm that does not expose the information on $\mathbf{B}$ even if signatures are generated many times. In Fig. 2, for a given $\mathbf{c}$, we select the nearest lattice point $\mathbf{v}$ in the dotted parallelepiped

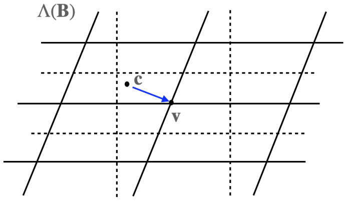

Fig. 2. Babai's the nearest plane algorithm

and then choose $\mathbf{v}'$ of $\mathbf{v}+\boldsymbol{\alpha}$, where the value of $\boldsymbol{\alpha}$ is a discrete Gaussian random variable. The greatest complexity in the signing of FALCON arises from the recursive implementation of Klein's algorithm, which combines Babai's nearest plane algorithm and discrete Gaussian distribution using the fast Fourier transform (FFT).

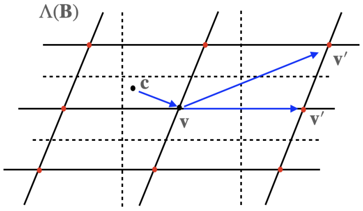

Fig. 3. Klein's algorithm

Contrary to this, Peregrine uses Babai's round-off algorithm to find the closest point $\mathbf{v}$ to $\mathbf{c}$ and add a centered binomial random variable to find $\mathbf{v}' = \mathbf{v}+\boldsymbol{\alpha}$. Using the numerical simulation, we can show the uniform distribution of signature enough to hide the information on the secret key $\mathbf{B}$. Thus, Peregrine can be securely used like Crystal-Kyber, an algorithm based on the centered binomial distribution.

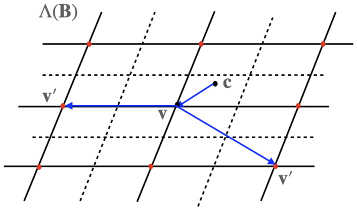

Fig. 4. Peregrine algorithm

3 Specification

Peregrine comprises three phases; key generation, singing, and verification. Each process will be explained in detail in this section, and the specific parameters used will be explained.

3.1 Notation

Cryptographic Parameters The security level and the maximal number of signing queries, indicating the security of signing, are expressed as λ and Q_s, respectively, and Q_s is set to be 2^{64}.

Matrices, Vectors, Polynomials Matrices are written in bold uppercase, and vectors in bold lowercase. Elements of matrices and vectors are polynomials, which are written in italic. We use the notation $\mathbf{B^t}$ to express the transpose of a matrix $\mathbf{B}$.

Number Fields Polynomials of Peregrine are defined in $\mathbb{Z}_q[x]/\phi(x)$, $\mathbb{Z}[x]/\phi(x)$, $\mathbb{Q}[x]/\phi(x)$. Since polynomials defined in various number fields have to be handled depending on many stages of key generations, we have to care about some points in implementation. Since the integer modulus q is the prime 12289, $\mathbb{Z}_q$ becomes a finite field. A polynomial modulus $\phi(x) = x^n + 1$, $n = 2^k$, ($k = 9$ or 10 in Peregrine), is a monic polynomial of $\mathbb{Z}[x]$, irreducible in $\mathbb{Q}[x]$, and has distinct roots over $\mathbb{C}$.

Ring Lattices In Peregrine, we define a full-rank matrix $\mathbf{B} \in (\mathbb{Z}[x]/\phi(x))^{2\times 2}$ and the ring lattice $\Lambda(\mathbf{B})$ generated by $\mathbf{B}$ is the set $\{\mathbf{z} \cdot \mathbf{B} | \mathbf{z} \in (\mathbb{Z}[x]/\phi(x))^2\}$.

Inner product Let $a = \sum_{i=0}^{n-1} a_i \cdot x^i$ and $b = \sum_{i=0}^{n-1} b_i \cdot x^i$, where $a, b \in \mathbb{Q}[x]/\phi(x)$. The inner product of two polynomials over $\mathbb{Q}[x]/\phi(x)$ is defined as

$$< a, b >= \frac{1}{\deg(\phi)} \cdot \sum_{\phi(\zeta)} a(\zeta) \cdot \overline{b(\zeta)}, \tag{2}$$

which coincides with the usual coefficient-wise inner product

$$< a, b >= \sum_{i=0}^{n-1} a_i \cdot b_i$$

for our choice of $\phi(x)$. For polynomial vectors $\mathbf{u} = (u_0, u_1, \cdots, u_m)$ and $\mathbf{v} = (v_0, v_1, \cdots, v_m)$ in $(\mathbb{Q}[x]/\phi(x))^m$, the inner product is defined by $< \mathbf{u}, \mathbf{v} > = \sum_{i=0}^{m} < u_i, v_i >$. When polynomials are in FFT expression, (2) is useful to calculate the inner product with the concept of Hermitian adjoint. We define the Hermitian adjoint of a polynomial a as

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

which satisfies that $a^*(\zeta) = \overline{a(\zeta)}$ for any root ζ of $\phi(x)$. Also, note that a norm of a polynomial is defined as $||a|| = \sqrt{< a, a >}$.

Centered Binomial Distribution A centered binomial distribution with parameter μ has samples in the interval $[-\mu/2, \mu/2]$ and probability mass function $P[X = x] = \frac{\mu!}{((\mu/2)+x)! \cdot ((\mu/2)-x)!} \cdot 2^{-\mu}$, where the parameter μ should be even. The centered binomial random variable, β_μ, with parameter μ can be generated simply from $B_1 - B_2$ where B_1 and B_2 are binomial random variables with $(\mu/2, 1/2)$.

3.2 Specification of Peregrine Key Generation

Among the polynomials required to create public key $\mathbf{A}$ and private key $\mathbf{B}$, we first explain how to randomly generate f and g, and then we will explain how to generate F and G by solving the NTRU equation in (1).

Generation of f and g The key generation of Peregrine starts from generating polynomials f and g in $\mathbb{Z}[x]/\phi(x)$. Let us consider some things in the generation of f and g.

First, with the generated f and g, public key $h = f^{-1} \cdot g \mod (q, \phi)$ is generated. The public key should not be distinguishable from the polynomials with coefficients generated uniformly at random. To ensure that, a random variable with a discrete Gaussian distribution is used in FALCON as demonstrated by [3]. The NTT coefficients of the generated polynomial f should be non-zero to guarantee that f is invertible. Otherwise, it must be regenerated.

Second, the secret basis matrix $\mathbf{B}$ has row vectors of $(g, \ -f)$ and $(G, \ -F)$ as its basis, and the norm values of these bases are small. The smaller the number, the shorter the signature will be. At this point, the value of $||(g, \ -f)||$ can be calculated, but the value of $||(G, \ -F)||$ cannot be accurately known. Instead, we can make a guess: Let $\mathbf{v} = (g, \ -f)$ and $\mathbf{u} = (G, -F)$. If $\mathbf{u}$ is orthogonalized to $\mathbf{v}$, $\mathbf{u'} \leftarrow \mathbf{u} - \lfloor \frac{<\mathbf{u},\mathbf{v}>}{<\mathbf{v},\mathbf{v}>} \rceil \cdot \mathbf{v}$ can be obtained. If we omit the round operation, we can calculate $\mathbf{u'}$ as follows

$$\begin{aligned}\mathbf{u}' &= \mathbf{u} - \frac{<\mathbf{u},\mathbf{v}>}{<\mathbf{v},\mathbf{v}>}\cdot\mathbf{v}\\ &= (G,\ \ -F) - \frac{G\cdot g^* + F\cdot f^*}{g\cdot g^* + f\cdot f^*}\cdot(g,\ \ -f)\\ &= (\frac{(f\cdot G - g\cdot F)\cdot f^*}{g\cdot g^* + f\cdot f^*}, \frac{(f\cdot G - g\cdot F)\cdot g^*}{g\cdot g^* + f\cdot f^*})\\ &= (\frac{q\cdot f^*}{g\cdot g^* + f\cdot f^*}, \frac{q\cdot g^*}{g\cdot g^* + f\cdot f^*}).\end{aligned}$$

Therefore, we can make the secret key good enough by checking in advance whether the value of $||\mathbf{u}'||$ is small enough. By checking the following conditions, we can determine whether to solve the NTRU equation or generate a new random polynomial with f and g.

$$\gamma = \max\{||(g,\ \ -f)||, ||(\frac{q\cdot f^*}{g\cdot g^* + f\cdot f^*}, \frac{q\cdot g^*}{g\cdot g^* + f\cdot f^*})||\} \leq 1.17\sqrt{q}.$$

When generating f and g accordingly, if the centered binomial distribution with $\mu = 26$ is used, we can obtain a distribution very close to the discrete Gaussian distribution using $\sigma = 1.17\sqrt{q/2n}$ originally used in FALCON, and successfully generate random polynomials, f and g, that pass all the conditions described above.

Generation of *F* and *G* Efficiently finding F and G satisfying the NTRU equation (1) based on the polynomials f and g is a complicated problem. In Peregrine, it is implemented as [12] using a tower field following the method used in FALCON. Here, the theoretical background will be omitted, and the specific implementation method will be summarized. For more information, please find the FALCON specification [18].

To solve the given NTRU equation, f and g on $\mathbb{Z}[x]/(x^n+1)$ are denoted as f^n and g^n. Let $f^{\frac{n}{2}}$ and $g^{\frac{n}{2}}$ be polynomials corresponding to elements on $\mathbb{Z}[x]/(x^{\frac{n}{2}}+1)$ of f^n and g^n. These polynomials can be calculated as follows

$$f^{\frac{n}{2}}(x) = f^n(x)\cdot f^n(-x) \mod x^{\frac{n}{2}}+1.$$

In this procedure, the degree of two factored polynomials becomes halved, but their coefficients are continuously increased. Thus, we have to use a prime larger than the maximum polynomial coefficient to preserve the polynomial coefficients. We can reduce the complexity by performing this multiplication in the NTT domain when we obtain $f^{\frac{n}{2}}$. We can obtain $f^{\frac{n}{4}}$ and $g^{\frac{n}{4}}$ in $\mathbb{Z}[x]/(x^{\frac{n}{4}}+1)$ in the same way. At the end of these sequential computations, we can obtain constants f^1 and g^1 in $\mathbb{Z}[x]/(x+1)$. The extended Euclidean algorithm can solve the NTRU problem with these constants when f^1 and g^1 are coprime. If f^1 and g^1 are not coprime, we cannot obtain a solution to the corresponding NTRU problem, and thus we have to regenerate f and g. If we denote the solutions of the NTRU problem with constants as $F^1,\ G^1 \in \mathbb{Z}[x]/(x+1)$, we can obtain $F^2,\ G^2 \in \mathbb{Z}[x]/(x^2+1)$. After successive computations, we can obtain $F^n,\ G^n \in \mathbb{Z}[x]/(x^n+1)$. Specifically, we can obtain $F^n,\ G^n \in \mathbb{Z}[x]/(x^n+1)$ from $F^{\frac{n}{2}},\ G^{\frac{n}{2}} \in \mathbb{Z}[x]/(x^{\frac{n}{2}}+1)$.

$$\begin{aligned} F^n &= F^{\frac{n}{2}}(x^2) \cdot g^{\frac{n}{2}}(x^2)/g(x) = F^{\frac{n}{2}}(x^2) \cdot g^{\frac{n}{2}}(-x) \mod x^n+1 \\ G^n &= G^{\frac{n}{2}}(x^2) \cdot f^{\frac{n}{2}}(x^2)/f(x) = G^{\frac{n}{2}}(x^2) \cdot f^{\frac{n}{2}}(-x) \mod x^n+1. \end{aligned}$$

We additionally need Babai's reduction to $(g^i,\ -f^i)$, $(G^i,\ -F^i)$, $i = 2^k,\ 0 \leq k \leq n$ for each step in the above procedure, since we should minimize the norm of vectors $(G^i,\ -F^i)$. In this Babai's reduction, the polynomials k^i with integer coefficients are obtained and the vector $k^i \cdot (g^i,\ -f^i)$ is subtracted from $(G^i,\ -F^i)$. This procedure is specified in Algorithm 1.

Algorithm 1 BabaiReduction$(f,\ g,\ F,\ G,\ \phi)$

```
Require: Polynomials $f,\ g,\ F,\ G \in \mathbb{Z}[x]/\phi$
Ensure: $(F,\ G)$ is reduced with respect to $(f,\ g)$
while $k \neq 0$ do
    $k \leftarrow \lfloor \frac{F\cdot f^* + G\cdot g^*}{f\cdot f^* + g\cdot g^*} \rceil \in Z[x]/\phi$
    $F \leftarrow F - k \cdot f$
    $G \leftarrow G - k \cdot g$
end while
```

Algorithm 2 shows the recursive algorithm to obtain the solutions of NTRU problem F and G using Algorithm 1 as a subroutine algorithm, which is equivalent to the above procedure. Algorithm 3 shows the generation algorithm for polynomial $f,\ g,\ F,\ G$ including these whole procedure. We can generate the public and secret keys from these polynomials as Algorithm 4.

3.3 Specification of Peregrine Signature

Message Hashing When we sign with the message M, we use the hash value of M, $H(M)$. However, if we sign multiple times using the same $H(M)$ in the GPV framework, the security proof in [1] cannot cover this situation. Therefore, the message is concatenated with a random salt $r \in \{0,1\}^k$, and the hash value $H(M||r)$ is used in the signature to prevent this issue. As in FALCON, we use $k = 320$ to preserve the 256-bit security level. We use SHAKE-256 as our hash function, which is included in FIPS 202 as the approved extendable-output hash function (XOF) [13].

Signature Generation Following the GPV framework, when we have $H(M||r) = c$, we need to find a short vector $\mathbf{s} = (s_1, s_2)$, satisfying $\mathbf{s} \cdot \mathbf{A}^t = c \mod (\phi, q)$, where a public key $\mathbf{A} = (1, h)$ and $s_1,\ s_2,\ c$, and h are in $\mathbb{Z}_q[x]/\phi$. Let us define the vector $\mathbf{c} := (c, 0)$. A simple method for obtaining a short signature $\mathbf{s}$ is as

Algorithm 2 NTRUSolve($f,\ g,\ n,\ q$)

```
Require: Polynomials $f,\ g \in \mathbb{Z}[x]/(x^n+1),\ n = 2^k$
Ensure: Polynomials $F^i,\ G^i$ satisfying (1)
if $n = 1$ then
    Compute $u, v \in \mathbb{Z}$ such that $u \cdot f - v \cdot g = \gcd(f, g)$
    if $\gcd(f, g) \neq 1$ then
        abort and return $\perp$
    end if
    $(F,\ G) \leftarrow (vq, uq)$
else
    $f' \leftarrow f(x) \cdot f(-x) \mod (x^{\frac{n}{2}} + 1)$
    $g' \leftarrow g(x) \cdot g(-x) \mod (x^{\frac{n}{2}} + 1)$
    $(F', G') \leftarrow$NTRUSolve$(f', g', \frac{n}{2}, q)$
    $F \leftarrow F'(x^2) \cdot g(-x)$
    $G \leftarrow G'(x^2) \cdot f(-x)$
    BabaiReduction$(f, g, F, G, x^n + 1)$
end if
```

Algorithm 3 PolyGen(ϕ, q)

```
Require: A monic polynomial $\phi(x) = x^n + 1 \in \mathbb{Z}[x]$ and a modulus $q$
Ensure: Polynomials $f,\ g,\ F,\ G$
$\mu \leftarrow 26$
for $i = 0,\ i \leq n-1,\ i \leftarrow i+1$ do
    $f_i \leftarrow \beta_\mu$
    $g_i \leftarrow \beta_\mu$
end for
$f \leftarrow \sum_{i=0}^{n-1} f_i \cdot x^i$
$g \leftarrow \sum_{i=0}^{n-1} g_i \cdot x^i$
if NTT of $f$ has 0 then
    restart
end if
$\gamma \leftarrow \max\{||(g,\ -f)||, ||(\frac{q \cdot f^*}{g \cdot g^* + f \cdot f^*}, \frac{q \cdot g^*}{g \cdot g^* + f \cdot f^*})||\}$
if $\gamma > 1.17\sqrt{q}$ then
    restart
end if
$(F, G) \leftarrow$NTRUSolve$(f,\ g,\ n,\ q)$
if $(F, G) = \perp$ then
    restart
end if
```

Algorithm 4 KeyGeneration(ϕ, q)

```
Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus $q$
Ensure: A secret key sk, a public key pk
f, g, F, G ← PolyGen($\phi, q$)
$\mathbf{B} \leftarrow \begin{pmatrix} g, & -f \\ G, & -F \end{pmatrix}$
sk←B
$h \leftarrow g \cdot f^{-1} \mod (q,\ \phi)$
pk← $h$
return sk, pk
```

follows. After obtaining a polynomial vector in $\mathbf{t} \in (\mathbb{Q}[x]/\phi)^2$ satisfying $\mathbf{t} \cdot \mathbf{B} = \mathbf{c}$, we obtain $\mathbf{s} = (\mathbf{t} - \mathbf{z}) \cdot \mathbf{B}$. Then, we have

$$(\mathbf{t} - \mathbf{z}) \cdot \mathbf{B} \cdot \mathbf{A}^t = \mathbf{t} \cdot \mathbf{B} \cdot \mathbf{A}^t - \mathbf{z} \cdot \mathbf{B} \cdot \mathbf{A}^t = (c, 0) \cdot (1, h)^t + 0 = c = \mathbf{s} \cdot \mathbf{A}^t.$$

The vector $\mathbf{t} = (t_1, t_2)$ satisfies the following formula.

$$\mathbf{t} \cdot \mathbf{B} = (t_1 \cdot g + t_2 \cdot G, -t_1 \cdot f - t_2 \cdot F) = (c,\ 0). \tag{3}$$

We can obtain $t_1 = -t_2 \cdot F/f$ from (3), and we have

$$\begin{aligned} c &= t_1 \cdot g + t_2 \cdot G \\ &= (-\frac{F}{f} \cdot g + G) \cdot t_2 \\ &= \frac{f \cdot G - g \cdot F}{f} \cdot t_2 = \frac{q}{f} \cdot t_2. \end{aligned}$$

In other words, we have $t_2 = c \cdot f/q \in \mathbb{Q}[x]/\phi$ and $t_1 = -c \cdot F/q \in \mathbb{Q}[x]/\phi$. $c \cdot f \mod \phi$ and $c \cdot F \mod \phi$ are in $\mathbb{Z}[x]/\phi$. All coefficients divided by q are rational numbers. Thus, we can separate each t_l for $l = 1, 2$ as the sum of the integer part I_l and the fraction part R_l, where all coefficients of R_l are in $[-0.5, 0.5)$. We have, $R_l = \sum_{j=0}^{n-1} r_{lj} \cdot x^j,\ l = 1, 2,\ -0.5 < r_{lj} \leq 0.5$ and $I_l = \sum_{j=0}^{n-1} i_{lj} \cdot x^j,\ l = 1, 2,\ i_{lj} \in \mathbb{Z}$. Then, we can represent the vector $\mathbf{t}$ as

$$\mathbf{t} = (t_1, t_2) = (-\frac{c \cdot F}{q},\ \frac{c \cdot f}{q}) = (R_1 + I_1,\ R_2 + I_2).$$

The first trial for minimizing the signature $\mathbf{s}$ is to obtain $\mathbf{s} = (\mathbf{t} - \mathbf{z}) \cdot \mathbf{B}$ where $\mathbf{z} = (I_1, I_2)$. However, this deterministic method has problems with security. Thus, Peregrine generates $\mathbf{s} = (\mathbf{t} - \mathbf{z}') \cdot \mathbf{B}$ where $\mathbf{z}' = (I_1 + J_1,\ I_2 + J_2)$. The vector $(I_1,\ I_2)$ can be easily derived from $\mathbf{t}$. The vector $(J_1,\ J_2) \in (\mathbb{Z}[x]/\phi)^2$ is the polynomial vector with small coefficients where each coefficient is sampled from the centered binomial distribution.

With the above $\mathbf{z}'$, the following equations on $\mathbf{s}$ hold.

$$\begin{aligned}\mathbf{s} &= (\mathbf{t} - \mathbf{z}') \cdot \mathbf{B} \\ &= \mathbf{t} \cdot \mathbf{B} - \mathbf{z}' \cdot \mathbf{B} \\ &= (c, 0) - ((I_1 + J_1) \cdot g + (I_2 + J_2) \cdot G, \ \ - (I_1 + J_1) \cdot f - (I_2 + J_2) \cdot F) \\ &= (c - (I_1 + J_1) \cdot g - (I_2 + J_2) \cdot G, \ (I_1 + J_1) \cdot f + (I_2 + J_2) \cdot F) \mod (\phi, q).\end{aligned}$$

In Peregrine, after we compute $(I_1,\ I_2)$, we can use NTT in the signing procedure without FFT used in FALCON. For obtaining $(I_1,\ I_2)$, we can use the RNS and NTT as in the method in the ModDown algorithm in RNS-CKKS homomorphic encryption [22]. Suppose we set a modulus Q larger than the maximum values of coefficients of input polynomials and the result polynomials. In that case, we can use the RNS system for fast multiplication and division without the modular reduction by a large modulus Q in these operations. Here, we can guess the maximum values with $c \cdot F$ and $c \cdot f$. If the coefficients of $c,\ f,\ F$ are less than q, modulus Q is taken as the product of primes greater than $q^2 \cdot n$ $Q = q_1 \cdot q_2 \cdot q > q^2 \cdot n$, where $q_1, q_2,$ and q are coprime. Then, expressing c, f, and F in RNS can speed up their multiplication. Refer to the appendix for a detailed description of the ModDown algorithm. Algorithm 5 summarizes in Algorithm 5 about finding a value obtained by rounding the quotient of each coefficient divided by q for a polynomial having integer coefficients expressed in RNS form using modulus Q.

```
Algorithm 5 ModDown($f, Q, q$)
Require: A polynomial $f = \sum_{i=0}^{n-1} f_i \cdot x^i \in \mathbb{Z}_Q[x]/\phi$, $Q = q_1 \cdots q_l \cdot q$, a modulus $q$, $f_i$
    is in RNS form with $\mod (q_1, q_2, \cdots, q_l, q_{l+1} = q)$
Ensure: A polynomial $\overline{f} = \sum_{i=0}^{n-1} \overline{f_i} \cdot x^i = \lfloor f/q \rceil \in \mathbb{Z}_{Q/q}[x]/\phi$ in RNS form with
    mod $(q_1, q_2, \cdots, q_l)$
for $i = 0,\ i \leq n-1,\ i \leftarrow i+1$ do
    $b \leftarrow f_i$
    $b' = (b'_1, b'_2, \cdots, b'_l, b'_{l+1}) \leftarrow b + ([\lfloor \frac{q}{2} \rfloor]_{q_1}, [\lfloor \frac{q}{2} \rfloor]_{q_2}, \cdots, [\lfloor \frac{q}{2} \rfloor]_{q_l}, [\lfloor \frac{q}{2} \rfloor]_q)$
    $b'' = (b''_1, b''_2, \cdots, b''_l, b''_{l+1}) \leftarrow ([b'_{l+1}]_{q_1}, [b'_{l+1}]_{q_2}, \cdots, [b'_{l+1}]_{q_l}, [b'_{l+1}]_q)$
    $\overline{f_i} \leftarrow ((b'_1 - b''_1) \cdot [q^{-1}]_{q_1}, \cdots, (b'_l - b''_l) \cdot [q^{-1}]_{q_l})$
end for
```

Algorithm 6 shows the process of finding (I_1, I_2) using Algorithm 5 and generating (J_1, J_2) using the centered binomial distribution to generate signature $\mathbf{s}$. If the size of the norm of the generated signature $\mathbf{s}$ is greater than the acceptance bound, $\lfloor \beta^2 \rfloor$, it is regenerated. In actual signing, $\mathbf{s}$ is not transmitted, but only s_1 is transmitted, which is $s_1 + s_2 \cdot h = c \mod (\phi, q)$. This is because we can calculate the rest even if we know only one of s_1 and s_2.

Signature Verification The verification process is simple. When we know the message M and the public key, we must check that the received signature has a

Algorithm 6 Signature(M, $\mathbf{sk}$, $\lfloor \beta^2 \rfloor$)

```
Require: A message $M$, a secret key $\mathbf{sk}$, a bound $\lfloor \beta^2 \rfloor$
Ensure: A signature $\mathbf{sig}$ of $M$
$r \leftarrow \{0,1\}^{320}$ uniformly
$c \leftarrow H(M||r)$
$\mu \leftarrow 26$
$I_1 \leftarrow$ModDown($-c \cdot F$, $Q$, $q$)
$I_2 \leftarrow$ModDown($c \cdot f$, $Q$, $q$)
while $||\mathbf{s}|| > \lfloor \beta^2 \rfloor$ do
    for $i = 0$, $i \leq n-1$, $i \leftarrow i+1$ do
        $J_{1i} \leftarrow \beta_\mu$
        $J_{2i} \leftarrow \beta_\mu$
    end for
    $J_1 \leftarrow \sum_{i=0}^{n-1} J_{1i} \cdot x^i$
    $J_2 \leftarrow \sum_{i=0}^{n-1} J_{2i} \cdot x^i$
    $s_1 \leftarrow c - (I_1 + J_1) \cdot g - (I_2 + J_2) \cdot G \mod (\phi, q)$
    $s_2 \leftarrow (I_1 + J_1) \cdot f + (I_2 + J_2) \cdot F \mod (\phi, q)$
    $\mathbf{s} \leftarrow (s_1, s_2)$
end while
return $\mathbf{sig} = (r, s_2)$
```

sufficiently small norm value. Since the verifier uses the same hash function as the signer, if we know the message M and the received random salt r, we can restore the polynomial c to be signed. Moreover, since the public key, h is known, using the value of s_2 included in the signature, $s_1 + s_2 \cdot h = c \mod (\phi, q)$ value can be found. In this process, the polynomial product can be easily calculated using NTT. Finally, finding the norm value of $\mathbf{s} = (s_1, s_2)$ found in this way and checking that it is within the acceptance bound is arranged in Algorithm 7.

Algorithm 7 verify(M, $\mathbf{sig}$, $\mathbf{pk}$, $\lfloor \beta^2 \rfloor$)

```
Require: A message $M$, a signature $\mathbf{sig} = (r, s_1)$, a public key $\mathbf{pk} = h \in \mathbb{Z}_q[x]/\phi$, a
    bound $\lfloor \beta^2 \rfloor$
Ensure: Accept or reject
$c \leftarrow H(M||r)$
$s_1 \leftarrow c - s_2 \cdot h \mod (\phi, q)$
if $||(s_1, s_2)|| \leq \lfloor \beta^2 \rfloor$ then
    accept
else
    reject
end if
```

3.4 Parameter Sets

Peregrine is a signature algorithm made with the rationale to reduce the implementation complexity and thus increase the usability for general applications. Therefore, many parameters used in FALCON can be reused as they are, except for those related to random variable generation. The parameters used in Peregrine are summarized in Table 1. When the maximal number of signing queries Q_s is 2^{64}, target security level λ satisfying NIST level I according to [13] is set to 128 and target security level λ that satisfies NIST level V is set to 256. For each security level, the degree of the polynomial ring is 512 and 1024. The modulus $q = 12289 = 12 \cdot 1024 + 1$ is chosen by the smallest prime number that satisfies $2n|(q-1)$ so that multiplication of polynomials can be performed by NTT[14] operation. FALCON showed that there is no security problem because the q value of this size is large enough to respond to hybrid attacks and trivial attacks on SIS and small enough to prevent overstretched NTRU attacks. Under the assumption that discrete Gaussian and fast Fourier sampling algorithms are used in FALCON, signature acceptance bound, $\lfloor \beta^2 \rfloor$, are set according to each security level. Peregrine uses the same parameters, except for some parameters for a discrete Gaussian distribution, which are replaced with new parameters generated from a centered binomial distribution. The length of the public key and signature shown in Table 1 can be obtained when the compression and decompression algorithms used in FALCON (Algorithms 17 and 18) are used, and Peregrine uses the same method.

Table 1. Peregrine recommended parameters

	Peregrine-512	Peregrine-1024
Target NIST level	I	V
Ring degree n	512	$1,024$
Modulus q	$12,289$	$12,289$
CBD μ	26	26
Signature acceptance bound $\lfloor \beta^2 \rfloor$	$34,034,726$	$150,700,176$
Public key bytelength	897	$1,793$
Private key bytelength	$1,281$	$2,305$
Signature bytelength	666	$1,280$

4 Performance Analysis

This section presents the performance measurement results using the reference implementation of Peregrine. The runtime for key generation, sign, and verification of Peregrine are measured for the proposed parameters 512 and 1024. Since the performance presented in this section is measured based on a reference code, optimized measurement results in various computing environments will be added.

4.1 Description of Platform

Runtime measurements are carried out on Apple M1 chip with eight cores and 16GB memory. Peregrine's reference code is implemented in a standard ANSI C-based environment and compiled without any optimization options using the gnu c compiler (gcc).

4.2 Performance of Reference Implementation

When implementing Peregrine, we use RNG and SHAKE 256 codes developed by the third party as it is. Peregrine uses the same NTRU structure of FALCON for key generation and utilizes a centered binomial distribution instead of a discrete Gaussian distribution in FALCON. Unlike FALCON, NTT is mainly used rather than FFT so that floating point operations can be minimized, and overall performance can be improved by replacing them with integer operations. Since integer operations are used in sign and verification, there is no need to include an additional library for floating point operation when implementing SW.

In many applications where only sign and verify are frequently used with a key created once, it can be processed only by integer operations, which can be implemented to operate faster with a more compact code size. In addition, since floating point operation is minimized, it is possible to design smaller and lower power even when implemented with HW. HW implementation issues will be further studied in the future. The runtime of Peregrine's key generation, sign, and verification measured using the reference code is shown in Table 2.

Table 2. Runtimes in milliseconds (ms) of Peregrine using a reference implementation

	Peregrine-512	Peregrine-1024	FALCON-512	FALCON-1024
KeyGen	4.12	15.29	5.64	17.47
Sign	0.05	0.12	0.18	0.34
Verify	0.013	0.028	0.013	0.028

5 Security

Peregrine utilizes randomness generated using a centered binomial distribution in addition to Babai's nearest plane algorithm for the method of generating randomness added from the signature, instead of using the Klein[11] method, which has already proven security in FALCON. Therefore, Peregrine currently does not provide the same security proof level as FALCON. However, the centered binomial distribution is widely used in many lattice-based cryptosystems instead of the discrete Gaussian distribution.

For example, among the algorithms submitted in the NIST PQC standardization, Crystals-Dilithium, which was finally selected as a PQC signature standard, and NewHope [21], which is one of prominent candidates for the second round and embedded in Google Chrome Browser, are the most representative examples. In the case of using a centered binomial distribution, it is more difficult to derive an accurate security proof of the scheme, but practically no problem has been found due to having characteristics similar to discrete Gaussian. Peregrine's security justification will be studied through further research.

Thus, it is assumed that Peregrine's method will not significantly damage FALCON's security level. We will introduce the security level related to key recovery and forgery proven by FALCON.

Key Recovery Attacks The most straightforward way to find out the private key from a given public key is to find all lattice points within a certain radius from the origin by applying lattice reduction to the lattice generated based on the public key. Due to the similarity of parameters between Peregrine and FALCON, the security level of Peregrine can be similarly derived as in FALCON. The key recovery attack complexity can be presented in Table 3, which shows the BKZ blocksize attack complexity calculated by the best-known lattice reduction algorithm, DBKZ[17], and the Core-SVP hardness calculated using the corresponding classical and quantum algorithms.

Table 3. Key-recovery parameters in Peregrine

	Peregrine-512	Peregrine-1024
BKZ blocksize	458	936
Core-SVP hardness in classical algorithm	133	273
Core-SVP hardness in quantum algorithm	121	248

Forgery Attacks Generating a signature for a given message without knowing the private key will have the same difficulty as solving the SIS problem. For any given point, finding all points with a distance less than β and generating a forged signature solve the same problem, which can also be solved by lattice reduction. The key recovery parameters are listed in Table 4 that include BKZ blocksize calculated by DBKZ[17], the Core-SVP hardness calculated using the corresponding classical and quantum algorithms.

Table 4. Forgery parameters in Peregrine

	Peregrine-512	Peregrine-1024
BKZ blocksize	411	850
Core-SVP hardness in classical algorithm	120	248
Core-SVP hardness in quantum algorithm	108	222

6 Conclusion

The proposed Peregrine algorithm is a lattice-based hash-and-sign digital signature. It is an algorithm created to improve the convenience of implementation and speed of signature. Since the goal of FALCON selected in the existing NIST 4 rounds was to implement the proven security perfectly, it has the disadvantages of additional actual calculations, very complex implementation, and a long signature time. Peregrine has the disadvantage that security cannot be proved theoretically yet. However, Peregrine generates a signature through a more straightforward implementation process than FALCON. Since the distribution of the signature generated as a result of the Peregrine signature is difficult to distinguish from the FALCON result, it will be a practical and secure option for post-quantum signatures.

Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00400, Development of Highly Efficient PQC Security and Performance Verification for Constrained Devices) The authors have no competing interests to declare that are relevant to the content of this article.

References

1. C. Gentry, C. Peikert, and V. Vaikuntanathan, “Trapdoors for hard lattices and new cryptographic constructions,” in *Proc. 40th ACM STOC*, pp. 197–206, Victoria, Canada, May, 2008.
2. J. Hoffstein, J. Pipher, and J. H. Silverman, “NTRU: A ring-based public key cryptosystem.” Lecture Notes in Computer Science, Springer, vol. 1423, pp. 267–288, 1998.
3. D. Stehlé and R. Steinfeld, “Making NTRU as secure as worst-case problems over ideal lattices,” in *Proc. EUROCRYPT 2011*, vol. 6632 of LNCS, pp. 27–47, Tallinn, Estonia, May, 2011.
4. R. K. Zhao, R. Steinfeld, and A. Sakzad, “FACCT: fast, compact, and constant-time discrete gaussian sampler over integers,” *IEEE Trans. Computers*, 69(1), pp. 126–137, 2020.

5. J. Howe, T. Prest, T, Ricosset, and M. Rossi, “Isochronous gaussian sampling: From inception to implementation,” in *Proc. PQCrypto 2020*, pp. 53–71, Paris, France, April, 2020.
6. L. Ducas, V. Lyubashevsky, and T. Prest, “Efficient identity-based encryption over NTRU lattices,” in *Proc. ASIACRYPT 2014*, Part II, vol. 8874 of LNCS, pp. 22–41, Kaoshiung, Taiwan, December, 2014.
7. T. Prest, “Gaussian Sampling in Lattice-Based cryptography.” Theses, École Normale Supérieure, December, 2015.
8. L. Babai, “On Lovasz’ lattice reduction and the nearest lattice point problem,” in *Proc. STACS 85 2ND Annual Symposium on Theoretical Aspects of Computer Science*, New York, USA, 1985.
9. P. Q. Nguyen and O. Regev, “Learing a parallelepiped: Cryptanalysis of GGH and NTRU signatures,” in *Proc. EUROCRYPT 2006*, vol. 4004 of LNCS, pp. 271–288, St. Petersburg, Russia, May, 2006.
10. L. Ducas and P. Q. Nguyen, “Learning a zonotope and more: Cryptanalysys of NTRUSign countermeasures,” in *Proc. ASIACRYPT 2012*, vol. 7658 of LNCS, pp. 433–450, Beijing, China, December, 2012.
11. P. N. Klein, “Finding the closest lattice vector when it’s unusually close,” in *Proc. 11th SODA*, pp. 937–941, San Francisco, USA, January, 2000.
12. T. Pornin and T. Prest, “More efficient algorithms for the NTRU key generation using the field norm,” in *Proc. PKC 2019*, Part II, vol. 11443 of LNCS, pp. 504–533, Beijing, China, April, 2019.
13. FIPS 202, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” FIPS 202, Aug. 2015. http://dx.doi.org/10.6028/NIST.FIPS.202
14. J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex.” Fourier series, Mathematics of Computation, vol. 19, no. 90, pp. 297-–301, 1965.
15. T. Prest, “Sharper bounds in lattice-based cryptography using the Rényi divergence,” in *Proc. ASIACRYPT 2017*, Part I, vol. 10624 of LNCS, pp. 347–374, Hong Kong, China, December, 2017.
16. V. Lyubashevsky, “Lattice signatures without trapdoors,” in *Proc. EUROCRYPT 2012*, vol. 7237 of LNCS, pp. 738–755, Cambridge, UK, April, 2012.
17. D. Micciancio and C. Peikert, “Trapdoors for lattices: Simpler, tighter, faster, smaller,” in *Proc. EUROCRYPT 2012*, vol. 7237 of LNCS, pp. 700–718, Cambridge, UK, April, 2012.
18. P.A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, “Falcon: Fast-fourier lattice-based compact signatures over ntru,” NIST PQC Standardization Selected Algorithms 2022, July 2022.
19. R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-KYBER Algorithm Specifications and Supporting Documentation,” NIST PQC Standardization Selected Algorithms 2022, July 2022.
20. S. Bai, L. Ducas, E. Kiltz, T. Leopoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-Dilithium Algorithm Specification and Supporting Documentation,” NIST PQC Standardization Selected Algorithms 2022, July 2022.
21. T. Poppelmann, E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, P. Schwabe, D. Stebila, M.R. Albrecht, E. Orsini, V. Osheter, K.G. Paterson, G. Peer, N.P. Smart, ”NewHope Algorithm Specification and Supporting Documentation,” NIST PQC Standardization Round 2 Submissions, Jan. 2019.

22. J.H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song,“A full RNS variant of approximate homomorphic encryption,” in Proc. SAC 2018, pp. 347-–368, Calgary, Canada, 2018.

A ModDown Algorithm

We will explain the ModDown algorithm in detail. Suppose there is an arbitrary coefficient b. If $Q = \prod_{i=0}^{l} q_i$ and q_i are expressed as prime numbers that are prime numbers, they can be expressed as follows.

$$\begin{aligned} b \mod Q &= (b \mod q_0,\ b \mod q_1,\ \cdots,\ b \mod q_{l-1},\ b \mod q_l) \\ &= ([b]_{q_0},\ [b]_{q_1},\ \cdots,\ [b]_{q_{l-1}},\ [b]_{q_l}) \\ &= (b_0,\ b_1,\ \cdots,\ b_{l-1},\ b_l). \end{aligned}$$

We want to find the value of $\lfloor b/q_l \rceil$ through the ModDown algorithm. Assuming that the original b is less than Q, we can expect that the value of $\lfloor b/q_l \rceil$ will be smaller than the value of Q/q_l, so from q_0 to $q_{l-Upto1}$ can be expressed in RNS using l primes. This is the ModDown algorithm. What we want to find here is $\lfloor b/q_l \rceil$, which can be obtained as follows

$$\begin{aligned} \lfloor b/q_l \rceil &= (b - [b]_{q_l})/q_l + \lfloor ([b]_{q_l})/q_l \rceil \\ &= (b - [b]_{q_l})/q_l + \lfloor ([b]_{q_l} + \frac{q_l}{2})/q_l \rfloor \\ &= (b - [b]_{q_l})/q_l + ([b]_{q_l} + \lfloor \frac{q_l}{2} \rfloor - [[b]_{q_l} + \lfloor \frac{q_l}{2} \rfloor]_{q_l})/q_l \\ &= (b - [b]_{q_l} + [b]_{q_l} + \lfloor \frac{q_l}{2} \rfloor - [[b]_{q_l} + \lfloor \frac{q_l}{2} \rfloor]_{q_l})/q_l \\ &= (b + \lfloor \frac{q_l}{2} \rfloor - [[b]_{q_l} + \lfloor \frac{q_l}{2} \rfloor]_{q_l})/q_l \\ &= (b + \lfloor \frac{q_l}{2} \rfloor - [b + \lfloor \frac{q_l}{2} \rfloor]_{q_l})/q_l. \end{aligned}$$

Looking at the last of the above equations, we have to calculate $b + \lfloor \frac{q_l}{2} \rfloor$, calculate the modulo q_l, find the difference, and then finally calculate the result value as q_l. Therefore, we can obtain the round-off of b/q_l, which can be represented in RNS as follows.

$$\begin{aligned} b' &= b + \lfloor \frac{q_l}{2} \rfloor \\ &= ([b]_{q_0},\ \cdots,\ [b]_{q_{l-1}},\ [b]_{q_l}) + ([\lfloor \frac{q_l}{2} \rfloor]_{q_0},\ \cdots,\ [\lfloor \frac{q_l}{2} \rfloor]_{q_{l-1}},\ [\lfloor \frac{q_l}{2} \rfloor]_{q_l}) \\ &= ([b]_{q_0} + [\lfloor \frac{q_l}{2} \rfloor]_{q_0},\ \cdots,\ [b]_{q_{l-1}} + [\lfloor \frac{q_l}{2} \rfloor]_{q_{l-1}},\ [b]_{q_l} + [\lfloor \frac{q_l}{2} \rfloor]_{q_l}) \\ &= ([b + \lfloor \frac{q_l}{2} \rfloor]_{q_0},\ \cdots,\ [b + \lfloor \frac{q_l}{2} \rfloor]_{q_{l-1}},\ [b + \lfloor \frac{q_l}{2} \rfloor]_{q_l}) \\ &= (b'_0,\ \cdots,\ b'_{l-1},\ b'_l) \end{aligned}$$

and

$$\begin{aligned} b'' &= [b + \lfloor \frac{q_l}{2} \rfloor]_{q_l} = b'_l \\ &= ([b'_l]_{q_0},\ \cdots,\ [b'_l]_{q_{l-1}},\ [b'_l]_{q_l}) \\ &= (b''_0,\ \cdots,\ b''_{l-1},\ b'_l) \end{aligned}$$

In the above expression, $[b'_l]_{q_l}$ does not change even after taking the modulus because b'_l has already reduced with the modulo q_l. Therefore, $b' - b''$ can be found as follows

$$\begin{aligned} \lfloor b/q_l \rceil &= (b' - b'')/q_l \\ &= ((b'_0 - b''_0) \cdot [q_l^{-1}]_{q_0},\ \cdots,\ (b'_{l-1} - b''_{l-1}) \cdot [q_l^{-1}]_{q_{l-1}}, 0). \end{aligned}$$