

Asymptotic Complexity and Performance Comparison of Falcon and SOLMAE using their C Implementation

Kwangjo Kim¹ and YeonJun Kim²

¹ International Research Institute for Cyber Security(IRCS)/KAIST, Korea
kkj@kaist.ac.kr

² LGUPLUS, Korea
cherryk@lguplus.co.kr

Abstract. In 2023, since NIST has recently selected FALCON as one of quantum-resistant digital signatures which uses the hash-and-sign paradigm in the style of Gentry–Peikert–Vaikuntanathan framework and instantiated over NTRU lattices, SOLMAE as a variant of FALCON was submitted to KpqC competition by taking all the pros of FALCON and MITAKA and reducing their cons as much as possible.

In this paper, we suggest the asymptotic computational complexity of FALCON and SOLMAE that take $\Theta(n \log n)$ in their **KeyGen**, **Sign** and **Verif** procedures simultaneously and verify their performance by ANSI C language implementation. Our experiment shows that SOLMAE achieves the same high security and short key and signature sizes as FALCON but faster **Sign** procedure, takes a bit longer **KeyGen** procedure. However, the **Sign** and **Verif** procedures of SOLMAE-512 is about 10 times faster than those of ECDSA P256r1 currently used in TSL or SSL.

Keywords: Lattice-based cryptography · Hash-and-sign paradigm · NTRU trapdoors · Discrete Gaussian sampling · C language implementation

1 Introduction

When Shor [16] has proposed an efficient randomized algorithm on a hypothetical quantum computer in 1999 to integer factorization and discrete logarithm problems in a polynomial time, it was beyond our imagination building for the powerful computing environment at that time. Currently the threat of attacking the current (or classical) secure system by using the quantum computer is expected to be right at our fingertips due to the aggressive road map by IBM quantum computing. We are very concerned about so called *Harvest Now, Decrypt Later* attack [17] which is a surveillance strategy that relies on the acquisition and long-term storage of currently unreadable encrypted data awaiting possible breakthroughs in decryption technology that would render it readable in the future.

Due to the substantial amount of research on quantum computers, large-scale quantum computers if built, can break many public-key cryptosystems based on the number-theoretic hard problems in use. In 2016, NIST [14] has initiated Post Quantum Cryptography(PQC) project to solicit, evaluate, and standardize one or more quantum-resistant cryptographic algorithms for Key Encapsulation Mechanism(KEM) and Digital Signature(DS) worldwide. After several rounds, NIST has finally selected CRYSTALS-Kyber for KEM and CRYSTALS-Dilithium, FALCON, and SPHINCS+ for DS in 2022.

Influenced by this NIST PQC project, Korean cryptographic society led by KpqC task force [11] has called for soliciting Korean PQC candidates by the end of Oct. in 2022. By the due of submission, 7 candidates KEM and 8 candidates DS for KpqC competition were submitted and their details are available at <https://kpqc.or.kr/>.

SOLMAE which stands for an acronym of quantum-Secure algOrithm for Long-term Message Authentication and Encryption was submitted to KpqC Competition as one of DS candidate algorithms which is a lattice-based signature scheme inspired by several pioneering works based on the hash-then-sign signature paradigm proposed by Gentry, Peikert and Vaikuntanathan [5]. SOLMAE is inspired from FALCON’s design. Some of the new theoretical foundations were laid out in the presentation of Mitaka [1] while keeping the security level of FALCON with 5 NIST levels of security I to V. At a high level, SOLMAE removes the inherent technicality of the sampling

procedure, and most of its induced complexity from an implementation standpoint, for *free*, that is with no loss of efficiency. This theoretical simplicity translates into faster operations while preserving signatures and verification key sizes, on top of allowing for additional features absent from FALCON, such as enjoying cheaper masking and being parallelizable.

The companion work of this paper was published at Proc. of ICSICS2023 [9] to discuss with the performance comparison using their Python implementation of both algorithms which can't be accepted their exact performance evaluation in real-time applications. On the other hand, this paper discusses the real performance comparison between 2 algorithms by their ANSI C implementations which is more valuable to the practitioners and easy to grasp their execution times. To the best of our knowledge, there is no the open literature to compare FALCON and SOLMAE directly from the point of their asymptotic complexity and performance. In this paper, after giving a brief description from the specification of FALCON and SOLMAE, we discuss their asymptotic computational complexity of **KeyGen**, **Sign** and **Verif** procedures from the theoretical point of view and suggest our performance evaluation including performance comparison of SOLMAE-512 and ECDSA P256r1.

The organization of this paper is as follows: In **Section 2**, we define our notations and definition used in this paper. In **Sections 3 and 4**, we describe how FALCON and SOLMAE work summarized from their specifications, respectively. In **Section 5**, we discuss the asymptotic computational complexity of FALCON and SOLMAE. **Section 6** suggests our performance analysis of FALCON and SOLMAE in total and commonly-used computations in their C language reference implementation and gives an **exact** comparison the performance of SOLMAE-512 and ECDSA P256r1 at the same level of security. Finally, we will give concluding remarks and challenging issues.

2 Notations and Definition

To keep the consistency to understand FALCON and SOLMAE correctly, we will use the following notations and definitions used their specification throughout this paper.

Matrices, vectors, and scalars

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

Quotient rings

Let \mathbb{Z} and \mathbb{N} denote a set of integers and a set of all numbers starting from 1, respectively. \mathbb{Q} and \mathbb{R} denote a set of rational numbers and a set of real numbers, respectively. For $q \in \mathbb{N}^\times$, we denote by \mathbb{Z}_q the quotient ring $\mathbb{Z}/q\mathbb{Z}$. In FALCON and SOLMAE, an integer modulus $q = 12,289$ is prime, so \mathbb{Z}_q is also a finite field. We denote by \mathbb{Z}_q^\times the group of invertible elements of \mathbb{Z}_q , and by φ Euler's totient function: $\varphi(q) = |\mathbb{Z}_q^\times| = q - 1 = 3 \cdot 2^{12}$ since q is prime. The rings $\mathbb{Q}[x]/(\phi)$, $\mathbb{Z}[x]/(\phi)$, and $\mathbb{R}[x]/(\phi)$ where ϕ is a monic minimal polynomial will be interchangeably written as \mathcal{Q} , \mathcal{Z} , and $K_{\mathbb{R}}$, respectively for the sake of our convenience.

DFT representation

For $d = 2^n$, we use $\phi(x) = x^d + 1$. It is a monic polynomial of $\mathbb{Z}[x]$, irreducible in $\mathbb{Q}[x]$ and with distinct roots over \mathbb{C} . Then $\zeta_j = \exp(i(2j - 1)\pi/d)$ for $j = 1, 2, \dots, d$ are roots of $\phi(x)$. For $f = \sum f_i x^i \in K_{\mathbb{R}}$, we define the coefficient representation as $\mathbf{f} = (f_0, f_1, \dots, f_{d-1})$ and Discrete Fourier Transform(DFT) representation $\varphi(f) = (\varphi_1(f), \dots, \varphi_d(f))$.

Number fields

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

$$a^* = a_0 - \sum_{i=1}^{d-1} a_i x^{d-i} \quad (1)$$

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

$$\mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \Leftrightarrow \mathbf{B}^* = \begin{bmatrix} a^* & c^* \\ b^* & d^* \end{bmatrix} \quad (2)$$

Inner product

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

$$\langle a, b \rangle = \frac{1}{\deg(\phi)} \sum_{0 < i < d} \varphi_i(a) \cdot \overline{\varphi_i(b)} \quad (3)$$

$$\|a\| = \sqrt{\langle a, a \rangle} \quad (4)$$

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

$$\langle a, b \rangle = \sum_{0 \leq i < d} a_i b_i; \quad (5)$$

From an algorithmic point of view, computing the inner product or the norm is most easily done using Eq.(3) if polynomials are in FFT representation, and using Eq.(5) if they are in coefficient representation. By substituting $b = a$ in Eqs (3) and (5), we get

$$\|\varphi(a)\| = \sqrt{d} \cdot \|a\|. \quad (6)$$

where $\|\cdot\|$ is Euclidean norm. Since we know that

$$\|\varphi(a)\| = \sqrt{2} \cdot \|(Re(\varphi_1(a)), Im(\varphi_1(a)), \dots, Re(\varphi_{d/2}(a)), Im(\varphi_{d/2}(a)))\|, \quad (7)$$

we get

$$\|(Re(\varphi_1(a)), Im(\varphi_1(a)), \dots, Re(\varphi_{d/2}(a)), Im(\varphi_{d/2}(a)))\| = \sqrt{\frac{d}{2}} \cdot \|a\|. \quad (8)$$

If $a \in K_{\mathbb{R}}$ follows the d -dimensional standard normal distribution, it is known that

$$(Re(\varphi_1(a)), Im(\varphi_1(a)), \dots, Re(\varphi_{d/2}(a)), Im(\varphi_{d/2}(a))) \text{ follows } \mathcal{N}_{d/2}, \quad (9)$$

where $\mathcal{N}_{d/2}$ denotes continuous Gaussian distribution with zero mean and $\frac{d}{2} \cdot I_d$ (i.e., Identity matrix) variance.

Ring lattices

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

NTRU lattices

Let q be an integer, and $f \in \mathbb{Z}[x]/(x^d + 1)$ such that f is invertible modulo q (equivalently, $\det[f]$ is coprime to q). Let $h = g/f \bmod q$ and consider the NTRU module associated to h :

$$\mathcal{M}_{\text{NTRU}} = \{(u, v) \in K_{\mathbb{R}}^2 : hu - v = 0 \bmod q\},$$

and its lattice version

$$\mathcal{L}_{\text{NTRU}} = \{(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^{2d} : [h]\mathbf{u} - \mathbf{v} = 0 \bmod q\}.$$

This lattice has volume q^d . Over $K_{\mathbb{R}}$, it is generated by (f, g) and any (F, G) such that $fG - gF = q$. For such a pair $(f, g), (F, G)$, this means that $\mathcal{L}_{\text{NTRU}}$ has a basis of the form

$$\mathbf{B}_{f,g} = \begin{bmatrix} [f] & [F] \\ [g] & [G] \end{bmatrix}.$$

One checks that $([h], -\text{Id}_d) \cdot \mathbf{B}_{f,g} = 0 \bmod q$, so the verification key is h . The NTRU-search problem is : given $h = g/f \bmod q$, find any $(f' = x^i f, g' = x^i g)$. In its decision variant, one must distinguish $h = g/f \bmod q$ from a uniformly random $h \in R_q := \mathbb{Z}[x]/(q, x^d + 1) = (\mathbb{Z}/q\mathbb{Z})[x]/(x^d + 1)$. These problems are assumed to be intractable for large d .

Discrete Gaussians

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

$$D_{\mathbb{Z},\sigma,\mu}(x) = \frac{\rho_{\sigma,\mu}(x)}{\sum_{z \in \mathbb{Z}} \rho_{\sigma,\mu}(z)} \quad (10)$$

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

Gram-Schmidt orthogonalization

Any matrix $\mathbf{B} \in \mathcal{Q}^{n \times m}$ can be decomposed as follows:

$$\mathbf{B} = \mathbf{L} \times \tilde{\mathbf{B}} \quad (11)$$

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

$$\|\mathbf{B}\|_{GS} = \max_{\mathbf{b}_i \in \tilde{\mathbf{B}}} \|\tilde{\mathbf{b}}_i\| \quad (12)$$

The LDL* decomposition

The LDL* decomposition writes any full-rank Gram matrix as a product $\mathbf{L}\mathbf{D}\mathbf{L}^*$, where $\mathbf{L} \in \mathcal{Q}^{n \times n}$ is lower triangular with 1's on the diagonal, and $\mathbf{D} \in \mathcal{Q}^{n \times n}$ is diagonal. The LDL* decomposition and the GSO are closely related as for a basis \mathbf{B} , there exists a unique GSO $\mathbf{B} = \mathbf{L} \cdot \tilde{\mathbf{B}}$, and for a full-rank Gram matrix \mathbf{G} , there exists a unique LDL* decomposition $\mathbf{G} = \mathbf{L}\mathbf{D}\mathbf{L}^*$. If $\mathbf{G} = \mathbf{B}\mathbf{B}^*$, then $\mathbf{G} = \mathbf{L} \cdot (\tilde{\mathbf{B}}\tilde{\mathbf{B}}^*) \cdot \mathbf{L}^*$ is a valid LDL* decomposition of \mathbf{G} . As both decompositions are unique, the matrices \mathbf{L} in both cases are actually the same. In a nutshell:

$$[\mathbf{L} \cdot \tilde{\mathbf{B}} \text{ is the GSO of } \mathbf{B}] \Leftrightarrow [\mathbf{L} \cdot (\mathbf{B}\mathbf{B}^*) \cdot \mathbf{L}^* \text{ is the LDL}^* \text{ decomposition of } (\mathbf{B}\mathbf{B}^*)]. \quad (13)$$

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

3 Falcon Algorithm

A group of top-notch cryptographers, Hoffstein, Pipher and Silverman [7] suggested new public-key cryptosystem based on a polynomial ring in 1997 as an alternative to RSA and DH whose difficulties are based on number-theoretic hard problems such as integer factorization and discrete log problem, respectively. They founded the company so-called as NTRU³ Cryptosystem with Lieman and initiated an open-source lattice-based cryptography consisting of two algorithms: NTRUENCRYPT used for encryption/decryption and NTRUSIGN used for digital signatures. Their security relies on the presumed difficulty of factoring certain polynomials in a truncated polynomial ring into a quotient of two polynomials having very small coefficients.

NTRUSIGN was designed based on the GGH signature scheme [6] which was proposed in 1995 based on solving the closest vector problem (CVP) in a lattice and asymptotically is more efficient than RSA in the computation time for encryption, decryption, signing, and verifying are all quadratic in the natural security parameter. The signer demonstrates knowledge of a good basis for the lattice by using it to solve CVP on a point representing the message; the verifier uses a bad basis for the same lattice to verify that the signature under consideration is actually a lattice point and is sufficiently close to the message point.

On the other hand, Min *et al.*[12] suggested weak property of malleability of NTRUSIGN using the annihilating polynomial from a given message and signature pair to generate a valid signature. Nguyen and Regev [13] had cryptanalyzed the original GGH signature scheme including NTRUSIGN in 2006 successfully extracting secret information from many known signatures characterized by multivariate optimization problems. Their experiments showed that 90,000 signatures are sufficient to recover the NTRUSIGN-251 secret key.

In a nutshell, FALCON follows a framework introduced in 2008 by Gentry, Peikert, and Vaikuntanathan [5] which we call the GPV framework for short over the NTRU lattices and uses a typically hash-and-sign paradigm. Their high-level idea is the following:

1. The public key is a long basis of a q -ary lattice.
2. The private key is (essentially) a short basis of the same lattice.
3. In the signing procedure, the signer:
 - (a) generates a random value, $salt$;
 - (b) computes a target $\mathbf{c} = H(M||salt)$, where H is a hash function sending input to a random-looking point (on the grid);
 - (c) uses his knowledge of a short basis to compute a lattice point \mathbf{v} close to the target \mathbf{c} ;
 - (d) outputs $(salt, \mathbf{s})$, where $\mathbf{s} = \mathbf{c} - \mathbf{v}$.
4. The verifier accepts the signature $(salt, \mathbf{s})$ if and only if:
 - (a) the vector \mathbf{s} is short;
 - (b) $H(M||salt) - \mathbf{s}$ is a point on the lattice generated by his public key.

Only the signer should be able to *efficiently* compute v close enough to an arbitrary target. This is a decoding problem that can be solved when a basis of *short* vectors is known. On the other hand, anyone wanting to check the validity of a signature should be able to verify lattice membership. The **KeyGen**, **Sign** and **Verif** procedures for FALCON will be introduced briefly in the later Section by restating the original specification as in [3]. For details, the readers can refer to [3].

3.1 Key Generation of Falcon

For the class of NTRU lattices, a trapdoor pairs is $(h, \mathbf{B}_{f,g})$ where $h = f^{-1}g$, $\mathbf{B}_{f,g}$ is trapdoor basis over $\mathcal{L}_{\text{NTRU}}$ and Pornin & Prest [15] showed that a completion (F, G) can be computed in $O(d \log d)$ time from short polynomials $f, g \in \mathcal{Z}$. In practice, their implementation is as efficient as can be for this technical procedure: it is called **NtruSolve** in FALCON. Their algorithm only depends on the underlying ring and has now a stable version for $\mathbb{Z}[x]/(x^d + 1)$, where $d = 2^n$.

Figure 1 illustrates the flowchart of the key generation procedure for FALCON.

³ Number Theorists ‘R’ Us, or Number Theory Research Unit, or N-th degree TRuncated polynomial Ring.

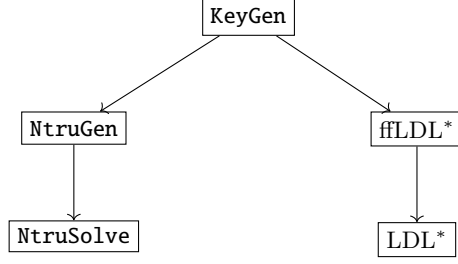


Fig. 1: Flowchart of **KeyGen** for FALCON

Algorithm 1 describes the pseudo-code for key generation of FALCON.

Algorithm 1: KeyGen of FALCON

Input: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q

Output: A secret key \mathbf{sk} , a public key \mathbf{pk}

```

1:  $f, g, F, G \leftarrow \text{NtruGen}$  ; /* Solving the NTRU equation */
2:  $\mathbf{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$  ;
3:  $\hat{\mathbf{B}} \leftarrow \text{FFT}(\mathbf{B})$  ; /* Compute FFT for each  $\{g, -f, G, -F\}$  */
4:  $\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^*$  ;
5:  $\mathbf{T} \leftarrow \text{ffLDL}^*(\mathbf{G})$  ; /* Compute the LDL* tree */
6: for each leaf of  $\mathbf{T}$  do
7:    $\text{leaf.value} \leftarrow \sigma / \sqrt{\text{leaf.value}}$  ; /* Normalization step */
8:  $\mathbf{sk} \leftarrow (\hat{\mathbf{B}}, \mathbf{T})$  ;
9:  $h \leftarrow gf^{-1} \bmod q$  ;
10:  $\mathbf{pk} \leftarrow h$  ;
11: return  $\mathbf{sk}, \mathbf{pk}$  ;

```

3.2 Signing of Falcon

At a high level, the signing procedure in FALCON is at first to compute a hashed value $\mathbf{c} \in \mathbb{Z}_q[x]/(\phi)$ from the message, M and a salt r , then using the secret key, f, g, F, G to generate two short values $(\mathbf{s}_1, \mathbf{s}_2)$ such that $\mathbf{s}_1 + \mathbf{s}_2 h = \mathbf{c} \bmod q$. An interesting feature is that only the *first half* of the signature $(\mathbf{s}_1, \mathbf{s}_2)$ needs to be sent along the message, as long as h is available to the verifier. This comes from the identity $h\mathbf{s}_1 = \mathbf{s}_2 \bmod q$ defining these lattices, as we will see in the **Verif** algorithm description.

The core of FALCON signing is to use **ffSampling** (**Algorithm 11** in [3]) which applies a randomizing rounding according to Gaussian distribution on the coefficient of $\mathbf{t} = (\mathbf{t}_0, \mathbf{t}_1) \in (\mathbb{Q}[x]/(\phi))^2$ stored in the FALCON Tree, \mathbf{T} at the **KeyGen** procedure of FALCON.

This fast Fourier sampling algorithm can be seen as a recursive version of Klein’s well-known trapdoor sampler, but *cannot be computed in parallel* also known as the GPV sampler. Klein’s sampler uses a matrix \mathbf{L} and the norm of Gram-Schmidt vectors as a trapdoor while FALCON are using a tree of non-trivial elements in such matrices. Note that Fouque *et. al.*[4] suggested Gram-Schmidt norm leakage in FALCON by timing side channels in the implementation of the one-dimensional Gaussian samplers.

FALCON cannot output two different signatures for a message. This well-known concern of the GPV framework can be addressed in several ways, for example, making a stateful scheme or by hash randomization. FALCON chose the latter solution for efficiency purposes. In practice, **Sign** adds a random “salt” $r \in \{0, 1\}^k$, where k is large enough that an unfortunate collision of messages is unlikely to happen, that is, it hashes $(r||M)$ instead of M . A signature is then $\mathbf{sig} = (r, \text{Compress}(\mathbf{s}_1))$.

Figure 2 and **Algorithm 2** sketches the signing procedure for FALCON and shows its pseudo-code for FALCON, respectively.

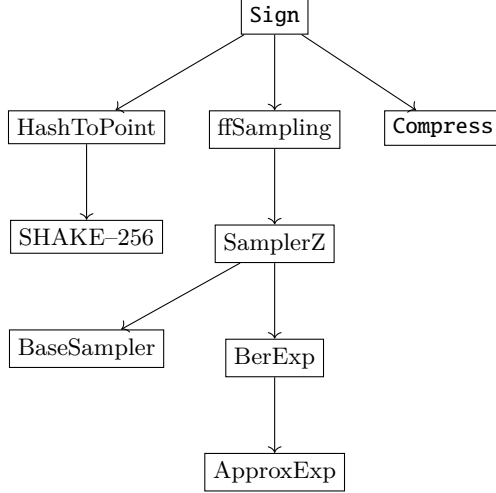


Fig. 2: Flowchart of **Sign** for FALCON.

Algorithm 2: Sign of FALCON

Input: A message $M \in \{0, 1\}^*$, secret key \mathbf{sk} , a bound γ .

Output: A pair $(r, \text{Compress}(\mathbf{s}_1))$ with $r \in \{0, 1\}^{320}$ and $\|(\mathbf{s}_1, \mathbf{s}_2)\| \leq \gamma$.

```

1:  $r \leftarrow \mathcal{U}(\{0, 1\}^{320})$ ;
2:  $\mathbf{c} \leftarrow \text{HashToPoint}(r \| M, q, n)$ ;
3:  $\mathbf{t} \leftarrow (-\frac{1}{q} \text{FFT}(c) \odot \text{FFT}(F), \frac{1}{q} \text{FFT}(c) \odot \text{FFT}(f))$ ;          /*  $\mathbf{t} = (\text{FFT}(c), \text{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$  */
4: do
5:   do
6:      $\mathbf{z} \leftarrow \text{ffSampling}_n(\mathbf{t}, \mathbf{T})$ ;
7:      $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$ ;          /* At this point,  $\mathbf{s}$  follows Gaussian distribution. */
8:     while  $\|\mathbf{s}\|^2 > \gamma$ 
9:        $(s_1, s_2) \leftarrow \text{FFT}^{-1}(\mathbf{s})$ ;
10:     $s \leftarrow \text{Compress}(s_2, 8 \cdot \text{sbytelen} - 328)$ ;          /* Remove 1 byte for the header, and 40
bytes for  $\mathbf{r}$  */
11: while  $(s = \perp)$ 
12: return  $(r, s)$ ;

```

3.3 Verification of Falcon:

The last step of the scheme is thankfully simpler to describe. Upon receiving a signature (r, \mathbf{s}) and message M , the verifier decompresses \mathbf{s} to a polynomial \mathbf{s}_1 and $\mathbf{c} = (0, \mathbf{H}(r \| M))$, then wants to recover the full signature vector $\mathbf{v} = (\mathbf{s}_1, \mathbf{s}_2)$. If \mathbf{v} is a valid signature, the verification identity is $(h, -1) \cdot (\mathbf{c} - \mathbf{v}) = -\mathbf{H}(r \| M) - h\mathbf{s}_1 + \mathbf{s}_2 \bmod q = 0$, or equivalently the verifier can compute

$$\mathbf{s}_2 = \mathbf{H}(r \| M) + h\mathbf{s}_1 \bmod q.$$

This is computed in the ring R_q , and can be done very efficiently for a good choice of modulus q using the Number Theoretic Transform (NTT). FALCON currently follow the standard choice of $q = 12, 289$, as the multiplication in NTT format amounts to d integer multiplications in $\mathbb{Z}/q\mathbb{Z}$. The last step is to check that $\|(\mathbf{s}_1, \mathbf{s}_2)\|^2 \leq \gamma^2$: the signature is only accepted in this case. The rejection bound γ comes from the expected length of vectors outputted by **Sample** described as **Algorithm 4** in [10].

Since they are morally Gaussian, they concentrate around their standard deviation; a “slack” parameter $\tau = 1.042$ is tuned to ensure that 90% of the vectors generated by **Sample** will get through the loop:

$$\gamma = \tau \cdot \sigma_{\text{sig}} \cdot \sqrt{2d}.$$

Algorithm 3 shows the pseudo-code of verification procedure of FALCON.

Algorithm 3: Verif of FALCON

Input: A signature (r, \mathbf{s}) on M , a public key $\mathbf{pk} = h$, a bound γ .

Output: Accept or Reject.

```

1:  $\mathbf{s}_1 \leftarrow \text{Decompress}(\mathbf{s})$ ;
2:  $\mathbf{c} \leftarrow \mathbb{H}(r || M)$ ;
3:  $\mathbf{s}_2 \leftarrow \mathbf{c} + h\mathbf{s}_1 \bmod q$ ;
4: if  $\|(\mathbf{s}_1, \mathbf{s}_2)\|^2 > \gamma^2$  then
 $\S$ : return Reject.
   end
6: return Accept.

```

4 SOLMAE Algorithm

SOLMAE is inspired from FALCON’s design. Some of the new theoretical foundations were laid out in the presentation of Mitaka [1]. At a high level, it removes the inherent technicality of the sampling procedure, and most of its induced complexity from an implementation standpoint, for *free*, that is with no loss of efficiency. This simplicity translates into faster operations while preserving signatures and verification keys sizes, on top of allowing for additional features absent from FALCON, such as enjoying cheaper masking, and being parallelizable. By using the novel compression techniques and tools of [2], SOLMAE can also obtain smaller signatures and verification keys than those already achieved by FALCON. To sum up, SOLMAE aims to achieve *better performances* for the same security and advantages as FALCON.

While its predecessor FALCON could be summed-up as *an efficient instantiation of the GPV framework*, SOLMAE takes it one step further. The main ingredients in SOLMAE are:

- **Hybrid sampler** is a faster, simpler, parallelizable, and maskable Gaussian sampler to generate signatures;
- **Optimally tuned key generation algorithm**, enhancing the security of the used hybrid sampler to that of FALCON’s level⁴;
- **Dedicated compression techniques** to reduce bandwidth consumption even further, at no cost on the security according to our analyses.

The **KeyGen**, **Sign** and **Verif** procedures for SOLMAE will be introduced briefly in the later Section by restating the original specification in [10]. For details, the readers can refer to [10].

4.1 Key Generation of SOLMAE

An important concern here is that not all pair $(f, g), (F, G)$ gives good trapdoor pairs for **Sample** described as **Algorithm 4** in [10]. Schemes such as FALCON and MITAKA solve this technicality essentially by sieving among all possible bases to find the ones that reach an acceptable quality for the **Sample** procedure. This technique is costly, and many tricks were used to achieve an acceptable **KeyGen**. *This sieving routine was bypassed by redesigning completely how good quality bases can be found.* This improves the running time of **KeyGen** and also increases the security offered by **Sample**. In any case, note that **NtruSolve**’s running time largely dominates the overall time for **KeyGen**: this is not avoidable as the basis completion algorithm requires working with quite large integers and relatively high-precision floating-point arithmetic.

At the end of the procedure, the secret key contains not only the secret basis but also the necessary data for **Sign** and **Sample**. This additional information can be represented by elements in $K_{\mathbb{R}}$ and is computed during or at the end of **NtruSolve**. All-in-all, **KeyGen** outputs:

$$\begin{aligned} \mathbf{sk} &= (\mathbf{b}_1 = (f, g), \mathbf{b}_2 = (F, G), \tilde{\mathbf{b}}_2 = (\tilde{F}, \tilde{G}), \Sigma_1, \Sigma_2, \beta_1, \beta_2), \\ \mathbf{pk} &= (h, q, \sigma_{\text{sig}}, \eta), \end{aligned}$$

⁴ This corresponds to NIST-I and NIST-V requirements.

where we recall that $h = g/f \bmod q$. These parameters and a table of their practical values are described more thoroughly in [10].

Informally, they correspond to the following:

- $(f, g), (F, G)$ is a good basis of the lattice $\mathcal{L}_{\text{NTRU}}$ associated to h , with quality $\mathcal{Q}(f, g) = \alpha$, and $\tilde{\mathbf{b}}_2$ is the Gram-Schmidt orthogonalization of (F, G) with respect to (f, g) ;
- $\sigma_{\text{sig}}, \eta$ are respectively the standard deviation for signature vectors, and a tight upper bound on the “smoothing parameter of \mathbb{Z}^d ”;
- $\Sigma_1, \Sigma_2 \in K_{\mathbb{R}}$ represent covariance matrices for two intermediate Gaussian samplings in **Sample**;
- the vectors $\beta_1, \beta_2 \in K_{\mathbb{R}}^2$ represent the orthogonal projections from $K_{\mathbb{R}}^2$ onto $K_{\mathbb{R}} \cdot \mathbf{b}_1$ and $K_{\mathbb{R}} \cdot \tilde{\mathbf{b}}_2$ respectively. In other words, they act as “getCoordinates” for vectors in $K_{\mathbb{R}}^2$. They are used by **Sample** and are precomputed for efficiency.

Algorithm 4 computes the necessary data for signature sampling, then outputs the key pair. Note that **NtruSolve** could also compute the sampling data and the public key, but for clarity, the pseudo-code gives these tasks to **KeyGen** of SOLMAE. Figure 3 sketches the key generation procedure of SOLMAE

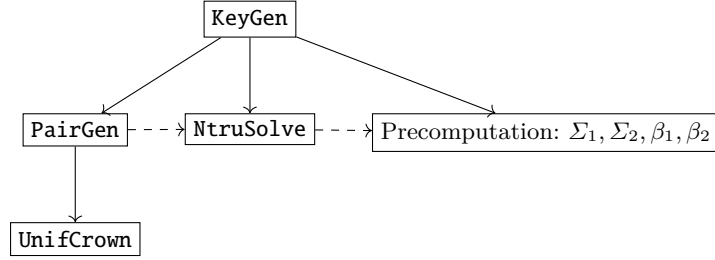


Fig. 3: Flowchart of **KeyGen** of SOLMAE.

Algorithm 4: KeyGen of SOLMAE

Input: A modulus q , a target quality parameter $1 < \alpha$, parameters $\sigma_{\text{sig}}, \eta > 0$

Output: A basis $((f, g), (F, G)) \in R^2$ of an NTRU lattice $\mathcal{L}_{\text{NTRU}}$ with $\mathcal{Q}(f, g) = \alpha$;

```

1: repeat
  |  $\mathbf{b}_1 := (f, g) \leftarrow \text{PairGen}(q, \alpha, R_-, R_+)$ 
  |   until  $f$  is invertible modulo  $q$ ;
  ; /* Secret basis computation between  $R_-$  and  $R_+$  */
2:  $\mathbf{b}_2 := (F, G) \leftarrow \text{NtruSolve}(q, f, g)$ ;
3:  $h \leftarrow g/f \bmod q$ ; /* Public key data computation */
4:  $\gamma \leftarrow 1.1 \cdot \sigma_{\text{sig}} \cdot \sqrt{2d}$ ; /* tolerance for signature length */
5:  $\beta_1 \leftarrow \frac{1}{\langle \mathbf{b}_1, \mathbf{b}_1 \rangle_K} \cdot \mathbf{b}_1$ ; /* Sampling data computation, in Fourier domain */
6:  $\Sigma_1 \leftarrow \sqrt{\frac{\sigma_{\text{sig}}^2}{\langle \mathbf{b}_1, \mathbf{b}_1 \rangle_K} - \eta^2}$ ;
7:  $\tilde{\mathbf{b}}_2 := (\tilde{F}, \tilde{G}) \leftarrow \mathbf{b}_2 - \langle \beta_1, \mathbf{b}_2 \rangle \cdot \mathbf{b}_1$ ;
8:  $\beta_2 \leftarrow \frac{1}{\langle \tilde{\mathbf{b}}_2, \tilde{\mathbf{b}}_2 \rangle_K} \cdot \tilde{\mathbf{b}}_2$ ;
9:  $\Sigma_2 \leftarrow \sqrt{\frac{\sigma_{\text{sig}}^2}{\langle \tilde{\mathbf{b}}_2, \tilde{\mathbf{b}}_2 \rangle_K} - \eta^2}$ ;
10:  $\text{sk} \leftarrow (\mathbf{b}_1, \mathbf{b}_2, \tilde{\mathbf{b}}_2, \Sigma_1, \Sigma_2, \beta_1, \beta_2)$ ;
11:  $\text{pk} \leftarrow (q, h, \sigma_{\text{sig}}, \eta, \gamma)$ ;
12: return  $\text{sk}, \text{pk}$ ;
  
```

The function of two subroutines **PairGen** and **NtruSolve** are described below:

1. The **PairGen** algorithm generates d complex numbers $(x_j e^{i\theta_j})_{j \leq d/2}, (y_j e^{i\theta_j})_{j \leq d/2}$ to act as the FFT representations of two *real* polynomials $f^{\mathbb{R}}, g^{\mathbb{R}}$ in $K_{\mathbb{R}}$. The magnitude of these complex numbers is sampled in a planar annulus whose small and big radii are set to match a target $\mathcal{Q}(f, g)$ with **UnifCrown** ([10]). It then finds close elements $f, g \in \mathcal{Z}$ by round-off, unless maybe the rounding error was too large. When the procedure ends, it outputs a pair (f, g) such that $\mathcal{Q}(f, g) = \alpha$, where α depends on the security level.
2. **NtruSolve** is exactly Pornin & Prest’s algorithm and implementation [15]. It takes as input $(f, g) \in \mathcal{Z}^2$ and a modulus q , and outputs $(F, G) \in \mathcal{Z}^2$ such that $(f, g), (F, G)$ is a basis of $\mathcal{L}_{\text{NTRU}}$ associated to $h = g/f \bmod q$. It does so by solving the Bézout-like equation $fG - gF = q$ in \mathcal{Z} using recursively the tower of subfields for optimal efficiency.

4.2 Signing of SOLMAE

Recall that NTRU lattices live in \mathbb{R}^{2d} . Their structure also helps to simplify the preimage computation. Indeed, the signer only needs to compute $\mathbf{m} = \mathbf{H}(M) \in \mathbb{R}^d$, as then $\mathbf{c} = (0, \mathbf{m})$ is a valid preimage: the corresponding polynomials satisfy $(h, 1) \cdot \mathbf{c} = \mathbf{m}$.

As the same with **Sign** procedure of FALCON, an interesting feature is that only the *first half* of the signature $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{L}_{\text{NTRU}}$ needs to be sent along the message, as long as h is available to the verifier. This comes from the identity $h\mathbf{s}_1 = \mathbf{s}_2 \bmod q$ defining these lattices, as we will see in the **Verif** algorithm description.⁵

Because of their nature as Gaussian integer vectors, signatures can be encoded to reduce the size of their bit-representation. The standard deviation of **Sample** is large enough so that the $\lfloor \log \sqrt{q} \rfloor$ least significant bits of one coordinate are essentially random.

In practice, **Sign** adds a random “salt” $r \in \{0, 1\}^k$, where k is large enough that an unfortunate collision of messages is unlikely to happen, that is, it hashes $(r||M)$ instead of M — our analysis in this regard is identical to FALCON. A signature is then $\mathbf{sig} = (r, \mathbf{Compress}(s_1))$. SOLMAE cannot output two different signatures for a message like FALCON which was mentioned in Section 3.2.

Figure 4 sketches the signing procedure of SOLMAE and **Algorithm 5** shows its pseudo-code.

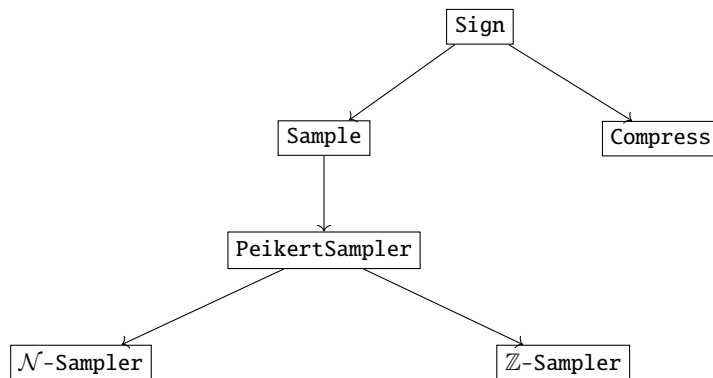


Fig. 4: Flowchart of **Sign** of SOLMAE.

⁵ The same identity can also be used to check the validity of signatures only with a hash of the public key h , requiring this time send both \mathbf{s}_1 and \mathbf{s}_2 , but we will not consider this setting here.

Algorithm 5: Sign of SOLMAE

Input: A message $M \in \{0, 1\}^*$, a tuple $\mathbf{sk} = ((f, g), (F, G), (\tilde{F}, \tilde{G}), \sigma_{\text{sig}}, \Sigma_1, \Sigma_2, \eta)$, a rejection parameter $\gamma > 0$.
Output: A pair $(r, \text{Compress}(s_1))$ with $r \in \{0, 1\}^{320}$ and $\|(\mathbf{s}_1, \mathbf{s}_2)\| \leq \gamma$.
1: $r \leftarrow \mathcal{U}(\{0, 1\}^{320})$;
2: $\mathbf{c} \leftarrow (0, \mathbf{H}(r \| M))$;
3: $\hat{\mathbf{c}} \leftarrow \text{FFT}(\mathbf{c})$;
4: **repeat**
 | $(\hat{s}_1, \hat{s}_2) \leftarrow \hat{\mathbf{c}} - \text{Sample}(\hat{\mathbf{c}}, \mathbf{sk})$; /* $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow D_{\mathcal{L}_{\text{NTRU}, \mathbf{c}, \sigma_{\text{sig}}}}$ */
 until $\|(\text{FFT}^{-1}(\hat{s}_1), \text{FFT}^{-1}(\hat{s}_2))\|^2 \leq \gamma^2$;
5: $s_1 \leftarrow \text{FFT}^{-1}(\hat{s}_1)$;
6: $s \leftarrow \text{Compress}(s_1)$;
return (r, s) ;

4.3 Verification of SOLMAE

This is the same as the Verification of FALCON stated in Section 3.3.

5 Asymptotic Complexity of Falcon and SOLMAE

To the best of our allowable knowledge as of writing this paper, we will suggest the asymptotic computational complexity of FALCON and SOLMAE algorithms with their pseudo-codes described their specifications based on the following assumptions to make our computation work to be simple:

- (i) Multiplication of large integers can be done by integer-type Karatsuba algorithm or Schönhage-Strassen algorithm. However, we assumed multiplication of large integers can be done in $\Theta(1)$.
- (ii) The multiplication and division of polynomials in $\mathbb{Z}[x]/(x^d + 1)$ or $\mathbb{Q}[x]/(x^d + 1)$ are assumed to compute the polynomial-type Karatsuba algorithm or operate pointwise in Fourier domain. It is known that the time complexity of the Karatsuba algorithm and FFT (or FFT^{-1}) are $\Theta(d^{3/2})$ and $\Theta(d \log d)$, respectively. We assume that all polynomial operations are done in the Fourier domain, so polynomial multiplication and division in $\mathbb{Z}[x]/(x^d + 1)$ or $\mathbb{Q}[x]/(x^d + 1)$ takes $\Theta(d \log d)$ time. Since every inverse element of \mathbb{Z}_q is stored in the list and the division of polynomials in $\mathbb{Z}_q[x]/(x^d + 1)$ can be done in the NTT domain, the division of polynomials in $\mathbb{Z}_q[x]/(x^d + 1)$ also takes $\Theta(d \log d)$.
- (iii) Some number of rejection samplings may inevitably happen in FALCON and SOLMAE. If one-loop for rejection sampling takes t times and its probability of the acceptance is p , the expectation value of the total time is $\sum_{k=1}^{\infty} p(1-p)^{k-1} \cdot kt = \frac{t}{p} \approx t$ since the value $1/p$ does not influence our asymptotic analysis due to its fixed constant value. So, we may ignore the number of rejections occurred in the rejection sampling. In fact, our experiment reveals that more or less 5 times rejections have occurred.
- (iv) Ignore some minor operations and trivial computations which do not affect the total asymptotic complexity so much.

5.1 Asymptotic Complexity of Falcon

Using the previous assumption stated in Section 5, Table 1 is the detailed analysis of the asymptotic complexity of KeyGen in FALCON from its algorithm whose total complexity to complete takes $\Theta(d \log d)$.

Table 1: Asymptotic complexity of **KeyGen** in FALCON

No.	Computation	Complexity	Location	Comment(d is degree)
1	NTRUGen(ϕ, q)	$\Theta(d \log d)$	Step 1 of Alg. 1	See below [†]
2	FFT(f)	$\Theta(d \log d)$	Step 3 of Alg. 1	
3	$\mathbf{B} \times \widehat{\mathbf{B}}^*$	$\Theta(d \log d)$	Step 4 of Alg. 1	Polynomial multiplications
4	fFLDL*(\mathbf{G})	$\Theta(d \log d)$	Step 5 of Alg. 1	See below [‡]
5	Normalization	$\Theta(d)$	Step 6–7 of Alg. 1	d leaf nodes in FALCON tree
6	$gf^{-1} \bmod q$	$\Theta(d \log d)$	Step 9 of Alg. 1	See the beginning of Section 5
Total Complexity of KeyGen :		$\Theta(d \log d)$		

[†] In **Algorithm 6**: NTRUGen(), Step 2 and Step 5(or 6) take $\Theta(d)$ and $\Theta(d \log d)$, respectively. Since the recurrence relation of **NtruSolve** is $T(d) = T(d/2) + \Theta(d \log d)$, thus Step 8 in **Algorithm 6** takes $\Theta(d \log d)$.

[‡] **Algorithm 9**: fFLDL*(\mathbf{G}) in [3] recursively calls fFLDL*(\mathbf{G}_0) and fFLDL*(\mathbf{G}_1), and other processes such as LDL* and Splitfft both take $\Theta(d)$, so the recursive formula is $T(d) = 2T(d/2) + \Theta(d)$. From this, we can get $T(d) = \Theta(d \log d)$.

Algorithm 6: NTRUGen(ϕ, q)

Input: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree n , a modulus q

Output: Polynomials f, g, F, G

```

1:  $\sigma \leftarrow 1.17\sqrt{q/2n}$ ;
2: for  $i$  from 0 to  $n - 1$  do
  |  $f_i \leftarrow D_{\mathbb{Z}, \sigma_{\{f, g\}}, 0}$ ;
  |  $g_i \leftarrow D_{\mathbb{Z}, \sigma_{\{f, g\}}, 0}$ ;
  | end
3:  $f \leftarrow \sum_i f_i x^i$ ;
4:  $g \leftarrow \sum_i g_i x^i$ ;
5: if  $NTT(f)$  contains 0 as a coefficient then
  | restart
  | end
6:  $\gamma \leftarrow \max\{\|(g, -f)\|, \|(\frac{qf^*}{ff^*+gg^*}, \frac{qg^*}{ff^*+gg^*})\|\}$ ;
7: if  $\gamma > 1.17\sqrt{q}$  then
  | restart
  | end
  | ;
8:  $F, G \leftarrow \text{NtruSolve}_{n, q}(f, g)$ ;
9: if  $(F, G) = \perp$  then
  | restart
  | end
  | return  $f, g, F, G$ ;

```

Tables 2 and 3 are the asymptotic complexity of **Sign** and **Verif** in FALCON, respectively whose total complexity to complete takes $\Theta(d \log d)$.

Table 2: Asymptotic complexity of **Sign** in FALCON

No.	Computation	Complexity	Location	Comment(d is degree)
1	HashToPoint($r M, q, n$)	$\Theta(d)$	Step 2 of Alg. 2	
2	FFT	$\Theta(d \log d)$	Step 3 of Alg. 2	
3	ffSampling $_n(t, T)$	$\Theta(d \log d)$	Step 6 of Alg. 2	See below †
4	$(\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$	$\Theta(d \log d)$	Step 7 of Alg. 2	Polynomial multiplications
5	$\ \mathbf{s}\ ^2$	$\Theta(d)$	Step 8 of Alg. 2	Calculating norm
6	invFFT	$\Theta(d \log d)$	Step 9 of Alg. 2	
7	Compress	$\Theta(d)$	Step 10 of Alg. 2	See below ‡
Total Complexity of Sign :		$\Theta(d \log d)$		

† ffSampling $_d$ recursively calls ffSampling $_{d/2}$ two times, and other processes such as splitfft and mergefft take $\Theta(d)$. So, the recursive formula is $T(d) = 2T(d/2) + \Theta(d)$. If we solve this, we get $T(d) = \Theta(d \log d)$.

‡ The compression function converts d degree polynomial into string of length $slen(= 666)$. $slen \approx d$, so it is irrelevant to say that the compression function takes $\Theta(d)$.

Table 3: Asymptotic complexity of **Verif** in FALCON

No.	Computation	Complexity	Location	Comment(d is degree)
1	HashToPoint($r m, q, n$)	$\Theta(d)$	Step 1 of Alg. 3	
2	Decompress ($\mathbf{s}, 8 \cdot \text{sbytelen} - 328$)	$\Theta(d)$	Step 2 of Alg. 3	More or less on par with Compress in Table 2
3	$c - s_2 h \bmod q$	$\Theta(d \log d)$	Step 5 of Alg. 3	Polynomial multiplication
4	$\ (s_1, s_2)\ ^2$	$\Theta(d)$	Step 6 of Alg. 3	Calculating norm
Total Complexity of Verif :		$\Theta(d \log d)$		

5.2 Asymptotic Complexity of SOLMAE

Based on the previous assumption stated in Section 5 as the same manner as we analyze the asymptotic complexity of FALCON, Table 4 is the asymptotic complexity of **KeyGen** in SOLMAE whose total complexity to complete takes $\Theta(d \log d)$.

Table 4: Asymptotic complexity of **KeyGen** in SOLMAE

No.	Computation	Complexity	Location	Comment(d is degree)
1	Pairgen	$\Theta(d \log d)$	Step 1 of Alg. 4	See below †
2	NtruSolve(q, f, g)	$\Theta(d \log d)$	Step 2 of Alg. 4	Explained in Table 1
3	$g/f \bmod q$	$\Theta(d \log d)$	Step 3 of Alg. 4	Polynomial operations
4	Key computations	$\Theta(d \log d)$	Step 4-9 of Alg. 4	Polynomial operations
Total Complexity of KeyGen :		$\Theta(d \log d)$		

† In **Algorithm 7:PairGen**, Steps 1,3,and 5 all take $\Theta(d)$ time. Steps 2 and 4 take $\Theta(d \log d)$ time.

Algorithm 7: PairGen

Input: A modulus q , a target quality parameter $1 < \alpha$, two radii parameters $0 < R_- < R_+$
Output: A pair (f, g) with $\mathcal{Q}(f, g) = \alpha$

- 1: **for** $i = 1$ **to** $d/2$ **do**
 - $x_i, y_i \leftarrow \text{UnifCrown}(R_-, R_+)$; /* see Algorithm 9 in [10] */
 - $\theta_x, \theta_y \leftarrow \mathcal{U}(0, 1)$;
 - $\varphi_{f,i} \leftarrow |x_i| \cdot e^{2i\pi\theta_x}$;
 - $\varphi_{g,i} \leftarrow |y_i| \cdot e^{2i\pi\theta_y}$;
- end**
- 2: $(f^{\mathbb{R}}, g^{\mathbb{R}}) \leftarrow (\text{FFT}^{-1}((\varphi_{f,i})_{i \leq d/2}), \text{FFT}^{-1}((\varphi_{g,i})_{i \leq d/2}))$;
- 3: $(\mathbf{f}, \mathbf{g}) \leftarrow (\lfloor f_i^{\mathbb{R}} \rfloor)_{i \leq d/2}, (\lfloor g_i^{\mathbb{R}} \rfloor)_{i \leq d/2}$;
- 4: $(\varphi(f), \varphi(g)) \leftarrow (\text{FFT}(\mathbf{f}), \text{FFT}(\mathbf{g}))$;
- 5: **for** $i = 1$ **to** $d/2$ **do**
 - if** $q/\alpha^2 > |\varphi_i(f)|^2 + |\varphi_i(g)|^2$ **or** $\alpha^2 q < |\varphi_i(f)|^2 + |\varphi_i(g)|^2$ **then**
 - | restart;
 - end**
- end**
- return** (\mathbf{f}, \mathbf{g}) ;

Table 5 is the asymptotic complexity of **Sign** in SOLMAE whose total complexity to complete takes $\Theta(d \log d)$.

Table 5: Asymptotic complexity of **Sign** in SOLMAE

No.	Computation	Complexity	Location	Comment(d is degree)
1	$\mathbf{H}(r \ M)$	$\Theta(d)$	Step 2 of Alg. 5	This is same as HashToPoint()
2	$\text{FFT}(\mathbf{c})$	$\Theta(d \log d)$	Step 3 of Alg. 5	
3	$\text{Sample}(\hat{c}, sk)$	$\Theta(d \log d)$	Step 4 of Alg. 5	See below †
4	$\text{FFT}^{-1}(\hat{s}_1)$	$\Theta(d \log d)$	Step 5 of Alg. 5	
5	$\text{Compress}(s_1)$	$\Theta(d)$	Step 6 of Alg. 5	Explained in Table 2

Total Complexity of **Sign**: $\Theta(d \log d)$

† In **Sample** (**Algorithm 4** in [10],) there are some polynomial multiplications and additions which take $\Theta(d \log d)$ and calls **PeikertSampler** (**Algorithm 5** in [10]) two times. In **PeikertSampler**, Step 1 takes $\Theta(d)$ (Generating normal vector with N-sampler takes $\Theta(d)$ and multiplying Σ takes $\Theta(d)$ since Σ is a diagonal matrix.). Steps 2, 3, and 5 take $\Theta(d \log d)$ since FFT computation is required. Step 4 takes $\Theta(d)$ simply since the loop iterates d times.

The asymptotic complexity of verification in SOLMAE is omitted since the algorithm is identical to verification in FALCON. Our asymptotic analysis discussed here is the first step to estimate the execution time of FALCON and SOLMAE roughly. We can claim that **KeyGen**, **Sign**, **Verif** procedures take $\Theta(d \log d)$ together with FALCON and SOLMAE here. This analysis does imply that FALCON and SOLMAE show the same execution times regardless of its implemented platform.

6 Performance analysis

As mentioned before, we have verified asymptotic complexities of FALCON and SOLMAE in **KeyGen**, **Sign** and **Verif** procedures and suggested that both take asymptotically similar $\Theta(d \log d)$ complexity that is difficult to evaluate their exact performance. Using the reference implementations of FALCON and SOLMAE which will be available over the github in C language under the same platform, we discuss the performance comparison between them and show the specific execution times here.

6.1 Description of platform

Timings below have been collected on two cores of Intel Xeon CPU E5-2640 v3 2.60GHz 4.00GB RAM under Ubuntu-20.04 server using `CFLAGS = -Wall -Wextra -march=native -O3` compiling flag of gcc. All the tests were executed 10,000 times and the average value were taken for the sake of the good evaluation.

6.2 Performance of our reference implementation

For this test, the input messages are chosen 1,024 byte randomly per 10,000 times with each count using different key pairs. FALCON performance numbers are also provided using the `speed` tool included in the official code archive

At first we checked and compared the performance of the FALCON with different security level having 256, 512 (NIST level I) and 1024 (NIST level V) in Fig. 5. The `Sign` procedure consumes more time than `KeyGen` and `Verif` procedures.

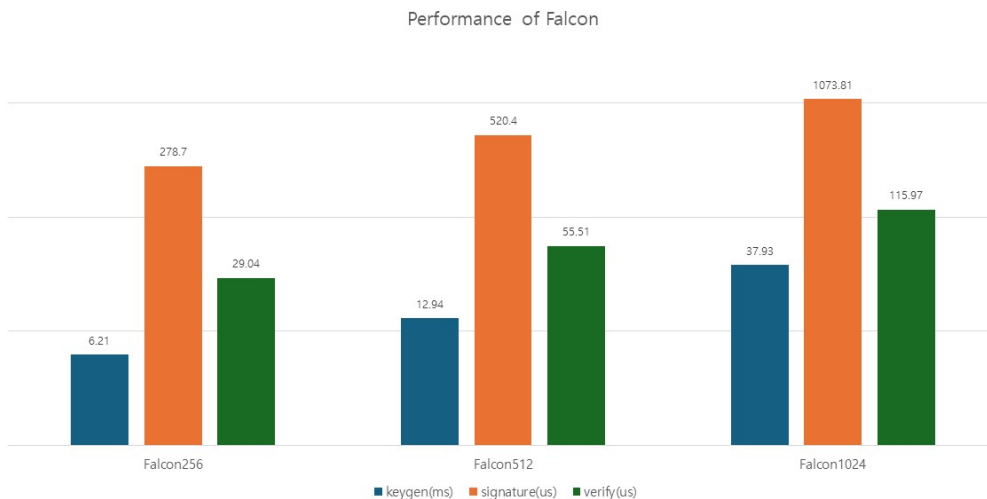


Fig. 5: Performance Comparison of FALCON-{256, 512, 1024}

The average clock cycle and time (μs) during `KeyGen`, `Sign` and `Verif` procedures using SOLMAE-512 and SOLMAE-1024 are shown in Table 6.

From this experiments, we found that SOLMAE without compression and decompression consistently outperforms FALCON in and `Sign` and `Verif` procedures in equal dimension, while its `KeyGen` procedure is slightly slower due to its extra computation of additional data for the private key in advance. The time to execute compression and decompression can be ignored.

The average time in μs of internal common computations such as FFT, point-wise multiplication and Gaussian sampling, *etc.* which are commonly used for SOLMAE-{512 and 1024} also FALCON as well are shown in Table 7.

On the other hand, in Table 8 we made performance comparison of SOLMAE-512 with Shake-128 and ECDSA P256r1 with SHA256 provided by Dreamsecurity Engineer [8] using our publicly available SOLMAE-512 C language reference implementation under their computing platform. Compared with ECDSA, the keygen of SOLMAE-512 is slower than ECDSA P256r1 by online computation⁶. But the signing and verification of SOLMAE-512 is about 10 times more faster

⁶ Note that keygen of SOLMAE can be operated online/offline together in parallel.

Table 6: Performance comparison of SOLMAE and FALCON

		SOLMAE-512	SOLMAE-1024	FALCON-512	FALCON-1024
KeyGen time	Mcycles	58.77	137.76	33.65	98.65
	time (ms)	22.6	52.97	12.94	37.93
pk size	Bytes	896	1,792	896	1,792
Sign time	Kcycles	758.3	1,512.9	1,353.15	2,792.22
	time (μ s)	291.63	581.82	520.4	1,073.81
sgn size	Bytes	666	1,375	666	1,280
Verif time	Kcycles	121.3	289.48	144.31	301.52
	time (μ s)	46.66	111.34	55.51	115.97

Table 7: Average time (μ s) of common computations for SOLMAE- $\{512$ and $1024\}$

	FFT	poly-add	pointwise-mult	FFT-mul-adj	poly-div-FFT	discrete-Gauss	Gaussian- sampling
SOLMAE-512	4.62	0.17	0.21	0.21	0.68	68.85	46.06
SOLMAE-1024	8.94	0.3	0.4	0.38	1.37	139.07	89.99

than those of ECDSA P256r1 currently used in TSL or SSL which makes no speed degradation when we apply quantum-secure SOLMAE for PKI and various security applications.

Table 8: Comparison of SOLMAE and ECDSA

		SOLMAE ECDSA	
Specification		512	P256r1
Size(Bytes)	pk	1,792	65
	sgn	1,375	32
Time	KeyGen (ms)	30.21	2.53
	Sign (μ s)	288.2	2,582.8
	Verif (μ s)	55.6	7,744.7

7 Concluding Remarks

FALCON is claimed to have the advantage of providing short public keys and signatures as well as high-security levels; plagued by a contrived signing algorithm, not very fast for signing and hard to parallelize; very little flexibility in terms of parameter settings. However, SOLMAE has a simple, fast, parallelizable signing algorithm, with flexible parameters with its novel key generation algorithm.

In this paper, after giving a brief description of the specification of FALCON and SOLMAE, we found that their asymptotic computational complexity of **KeyGen**, **Sign** and **Verif** procedures take $\Theta(n \log n)$ simultaneously. Also, our computer experiments using their C language implementation exhibit empirically that **KeyGen** of SOLMAE takes a bit longer time than that of SOLMAE, but executes faster **Sign** procedure. Amazingly enough, the **Sign** and **Verif** procedures of SOLMAE-512 is about 10 times more faster than those of ECDSA P256r1 currently used in TSL or SSL.

Further work such as elaborated analysis of computational complexity on FALCON and SOLMAE asymptotically is left to do next.

References

1. Espitau, T., Fouque, P.A., Gérard, F., Rossi, M., Takahashi, A., Tibouchi, M., Wallet, A., Yu, Y.: Mitaka: a simpler, parallelizable, maskable variant of falcon. *Advances in Cryptology, Proc. of EUROCRYPTO 2022, Part III* pp. 222–253 (2022) [1](#), [8](#)
2. Espitau, T., Tibouchi, M., Wallet, A., Yu, Y.: Shorter hash-and-sign lattice-based signatures. *Advances in Cryptology, Proc. of CRYPTO 2022, Part II* pp. 245–275 (2022) [8](#)
3. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru, <https://falcon-sign.info/> [5](#), [6](#), [12](#)
4. Fouque, P.A., Kirchner, P., Tibouchi, M., Wallet, A., Yu, Y.: Key recovery from gram-schmidt norm leakage in hash-and-sign signatures over ntru lattices. *Cryptology ePrint Archive, Paper 2019/1180* (2019), <https://eprint.iacr.org/2019/1180>, <https://eprint.iacr.org/2019/1180> [6](#)
5. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Ladner, R.E., Dwork, C. (eds.) *40th ACM STOC*. pp. 197–206. ACM Press (May 2008). <https://doi.org/10.1145/1374376.1374407> [1](#), [5](#)
6. Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. *Advances in Cryptology, Proc. of Crypto 1997* pp. 112–131 (1997) [5](#)
7. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998*. *Lecture Notes in Computer Science*, vol. 1423, pp. 267–288. Springer (1998) [5](#)
8. Kim, H.: Personal correspondence (2023), will provide upon request [15](#)
9. Kim, K.: Theoretical and empirical analysis of falcon and solmae using their python implementation. In: *Information Security and Cryptology -ICISC2023*. *Lecture Notes in Computer Science*, vol. 14562. Springer (2024) [2](#)
10. Kim, K., Tibouchi, M., Espitau, T., Takashima, A., Wallet, A., Yu, Y., Guilley, S., Kim, S.: Solmae : Algorithm specification. Updated SOLMAE, IRCS Blog (2023), <https://ircs.re.kr/?p=1714> [7](#), [8](#), [9](#), [10](#), [14](#)
11. KpqC: Korean post-quantum cryptography (2020), <https://kpgc.or.kr/> [1](#)
12. Min, S., Yamamoto, G., Kim, K.: Weak property of malleability in ntrusign. *Proc. of ACISP 2004, LNCS 3108* pp. 379–390 (2004) [5](#)
13. Nguyen, P.Q., Regev, O.: Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures. *Journal of Cryptology* **22**(2), 139–160 (2009) [5](#)
14. NIST: Post-quantum cryptography (2016), <https://csrc.nist.gov/projects/post-quantum-cryptography> [1](#)
15. Pornin, T., Prest, T.: More efficient algorithms for the NTRU key generation using the field norm. In: Lin, D., Sako, K. (eds.) *PKC 2019, Part II*. LNCS, vol. 11443, pp. 504–533. Springer, Heidelberg (Apr 2019). https://doi.org/10.1007/978-3-030-17259-6_17 [5](#), [10](#)
16. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review* **41**(2), 303–332 (1999) [1](#)
17. Wikipedia: Harvest now, decrypt later (2023), https://en.wikipedia.org/wiki/Harvest_now_decrypt_later [1](#)