Theoretical and Empirical Analysis of FALCON and SOLMAE using their Python Implementation

Kwangjo Kim

Korea Advanced Institute of Science and Technology(KAIST) and
International Research Institute for Cyber Security(IRCS), Korea
kkj@kaist.ac.kr

Abstract. 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 standard 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 take $\Theta(n \log n)$ in their `KeyGen`, `Sign` and `Verif` procedures simultaneously, but our computer experiments using their Python implementation exhibit empirically that `KeyGen` of FALCON–512 takes longer time than that of SOLMAE–512 by about a second while the other two procedures are running almost the same time. We show a sample execution of FALCON–512 and SOLMAE–512 with their real value are described in detail for the educational purpose to understand FALCON and SOLMAE easily. We also checked the Gaussian randomness of $\mathcal{N}$-`Sampler` and `UnifCrown` samplers used in SOLMAE only.

Keywords: Lattice-based cryptography · Hash-and-sign paradigm · NTRU trapdoors · Discrete Gaussian sampling · Python 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 standard 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–**S**ecure alg**O**rithm for **L**ong–term **M**essage **A**uthentication and **E**ncryption 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 [6].

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. We need to evaluate this features with our Python implementation which all the readers can easily understand and compare them.

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 and evaluate their performance empirically using their Python implementation including Gaussian samplers used in SOLMAE.

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 specification, respectively. In **Section 5**, we discuss the asymptotic computational complexity of FALCON and SOLMAE. In **Section 6**, we analyse the $\mathcal{N}$`-Sampler` and `UnifCrown` sampler used in SOLMAE only and verify its function by the experiment. In **Section 7**, we suggest the practical execution time of `KeyGen`, `Sign` and `Verif` procedures running 3,000 times for FALCON–512 and SOLMAE–512 by their Python implementation. 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. $\mathbf{B}$), vectors in bold lowercase (e.g. $\mathbf{v}$), and scalars - which include polynomials - in italic (e.g. s). We use the row convention for vectors. The transpose of a matrix $\mathbf{B}$ may be noted $\mathbf{B}^t$. It is to be noted that for a polynomial f, we do *not* use f' to denote its derivative in this document.

Quotient rings

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 $j = 1, 2, \cdots d$ are roots of $\phi(x)$. For $f = \Sigma f_i x^i \in K_{\mathbb{R}}$, we define the coefficient representation as $\mathbf{f} = (f_0, f_1, \cdots f_{d-1})$ and Discrete Fourier Transform(DFT) representation $\varphi(f) = (\varphi_1(f), \cdots, \varphi_d(f))$.

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:

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

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. $\mathbf{v}$):

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

Inner product

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

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

$$\|a\| = \sqrt{\langle a, a \rangle} \tag{4}$$

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:

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

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

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

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

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

we get

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

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

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

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:

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

and its lattice version

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

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 $\mathcal{L}_{\mathrm{NTRU}}$ has a basis of the form

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

One checks that $([h], -\mathrm{Id}_d)\cdot \mathbf{B}_{f,g} = 0 \bmod q$, so the verification key is h. The NTRU-search problem is : given $h = g/f \mod q$, 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:

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

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:

$$\mathbf{B} = \mathbf{L} \times \tilde{\mathbf{B}} \tag{11}$$

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:

$$\|\mathbf{B}\|_{GS} = \max_{\tilde{\mathbf{b_i}} \in \tilde{\mathbf{B}}} \|\tilde{\mathbf{b_i}}\| \tag{12}$$

The LDL* decomposition

The LDL* decomposition writes any full-rank Gram matrix as a product LDL*, where $\mathbf{L} \in \mathcal{Q}^{n\times n}$ is lower triangular with 1's on the diagonal, and $\mathbf{D} \in \mathcal{Q}^{n\times n}$ is diagonal. The LDL* decomposition and the GSO are closely related as for a basis $\mathbf{B}$, there exists a unique GSO $\mathbf{B} = \mathbf{L}\cdot\tilde{\mathbf{B}}$, and for a full-rank Gram matrix $\mathbf{G}$, there exists a unique LDL* decomposition $\mathbf{G} = \mathbf{LDL}^*$. If $\mathbf{G} = \mathbf{BB}^*$, 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:

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

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 How FALCON works

A group of top-notch cryptographers, Hoffstein, Pipher and Silverman [8] 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 NTRU [1] 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 [7] 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 [6] 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.

[1] Number Theorists 'R' Us, or Number Theory Research Unit, or N–th degree TRuncated polynomial Ring.

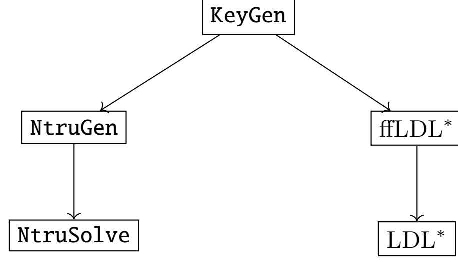


Fig. 1: Flowchart of **KeyGen** for FALCON

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 `sk`, a public key `pk`

```
1: f, g, F, G ← NtruGen ;                              /* Solving the NTRU equation */
2: B ← [g  −f ; G  −F] ;
3: B̂ ← FFT(B) ;                        /* Compute FFT for each {g, −f, G, −F} */
4: G ← B̂ × B̂* ;
5: T ← ffLDL*(G);                                        /* Compute the LDL* tree */
6: for each leaf of T do
7:     leaf.value ← σ/√leaf.value ;                       /* Normalization step */
8: sk ← (B̂, T);
9: h ← g f^{-1} mod q;
10: pk ← h;
11: return sk, 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, **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 **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 $\texttt{sig} = (r, \texttt{Compress}(\mathbf{s}_1))$.

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

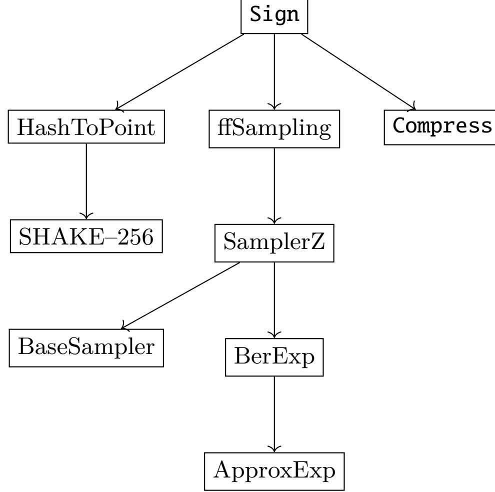


Fig. 2: Flowchart of `Sign` for FALCON.

Algorithm 2: `Sign` of FALCON

Input: A message $M \in \{0,1\}^*$, secret key **sk**, a bound γ.
Output: A pair $(r, \texttt{Compress}(\mathbf{s}_1))$ 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: $\mathbf{t} \leftarrow (-\frac{1}{q}\,\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\,\mathsf{FFT}(c) \odot \mathsf{FFT}(f))$; /* $\mathbf{t} = (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$ */
4: do
5: do
6: $\mathbf{z} \leftarrow \text{ffSampling}_n(\mathbf{t}, \mathrm{T})$;
7: $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$; /* At this point, s follows Gaussian distribution. */
8: while $||s||^2 > \gamma$
9: $(s_1, s_2) \leftarrow \mathsf{FFT}^{-1}(\mathbf{s})$;
10: $s \leftarrow \texttt{Compress}(s_2, 8 \cdot \text{sbytelen} - 328)$; /* Remove 1 byte for the header, and 40 bytes for 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 $\mathbf{c} = (0, \mathtt{H}(r||M))$, 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 $(h, -1) \cdot (\mathbf{c} - \mathbf{v}) = -\mathtt{H}(r||M) - h\mathbf{s}_1 + \mathbf{s}_2 \bmod q = 0$, or equivalently the verifier can compute

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

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 [9].

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:

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

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 $\mathsf{pk} = h$, a bound γ.
Output: Accept or Reject.

1: $\mathbf{s}_1 \leftarrow \texttt{Decompress}(\mathbf{s})$;
2: $\mathbf{c} \leftarrow \texttt{H}(r||M)$;
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**
5: **return** Reject.
end
6: **return** Accept.

4 How SOLMAE works

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 level[2];
– **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 [9]. For details, the readers can refer to [9].

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 [9]. 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:

$$\mathsf{sk} = (\mathbf{b}_1 = (f, g), \mathbf{b}_2 = (F, G), \widetilde{\mathbf{b}}_2 = (\widetilde{F}, \widetilde{G}), \Sigma_1, \Sigma_2, \beta_1, \beta_2),$$
$$\mathsf{pk} = (h, q, \sigma_{\texttt{sig}}, \eta),$$

[2] 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 [9].

Informally, they correspond to the following:

- $(f, g), (F, G)$ is a good basis of the lattice $\mathcal{L}_{\mathrm{NTRU}}$ associated to h, with quality $\mathcal{Q}(f, g) = \alpha$, and $\widetilde{\mathbf{b}}_2$ is the Gram-Schmidt orthogonalization of (F, G) with respect to (f, g);
- $\sigma_{\mathtt{sig}}, \eta$ 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 $K_{\mathbb{R}} \cdot \widetilde{\mathbf{b}}_2$ 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

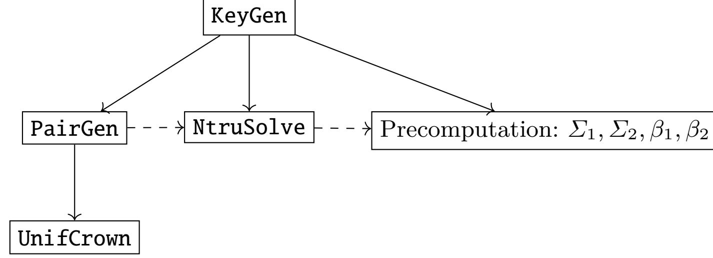


Fig. 3: Flowchart of `KeyGen` of SOLMAE.

Algorithm 4: `KeyGen` of SOLMAE

Input: A modulus q, a target quality parameter $1 < \alpha$, parameters $\sigma_{\mathtt{sig}}, \eta > 0$
Output: A basis $((f, g), (F, G)) \in R^2$ of an NTRU lattice $\mathcal{L}_{\mathrm{NTRU}}$ with $\mathcal{Q}(f, g) = \alpha$;

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

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

1. The `PairGen` algorithm generates d complex numbers $(x_j e^{i\theta_j})_{j\le d/2}, (y_j e^{i\theta_j})_{j\le d/2}$ to act as the FFT representations of two *real* polynomial $f^{\mathbb{R}}, g^{\mathbb{R}}$ 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` ([9]). 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 $\mathcal{L}_{\mathrm{NTRU}}$ 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 $\mathbf{m} = \mathtt{H}(M) \in \mathbb{R}^d$, 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 $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{L}_{\mathrm{NTRU}}$ 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. [3]

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 $\mathtt{sig} = (r, \mathtt{Compress}(s_1))$. 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.

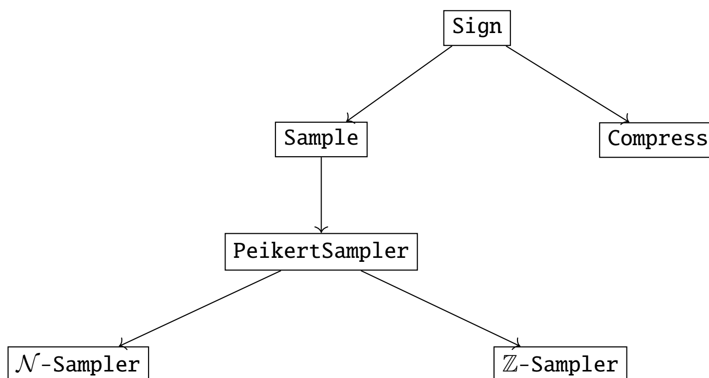


Fig. 4: Flowchart of `Sign` of SOLMAE.

[3] 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: $\texttt{Sign}$ of SOLMAE

Input: A message $M \in \{0,1\}^*$, a tuple $\texttt{sk} = ((f,g),(F,G),(\widetilde{F},\widetilde{G}),\sigma_{\texttt{sig}},\Sigma_1,\Sigma_2,\eta)$, a rejection parameter $\gamma > 0$.

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

1: $r \leftarrow \mathcal{U}(\{0,1\}^{320})$;
2: $\mathbf{c} \leftarrow (0, \texttt{H}(r||M))$;
3: $\hat{\mathbf{c}} \leftarrow \mathsf{FFT}(\mathbf{c})$;
4: **repeat**
  $(\hat{s}_1,\hat{s}_2) \leftarrow \hat{\mathbf{c}} - \texttt{Sample}(\hat{\mathbf{c}},\texttt{sk})$; /* $(\mathbf{s}_1,\mathbf{s}_2) \leftarrow D_{\mathcal{L}_{\texttt{NTRU}},\mathbf{c},\sigma_{\texttt{sig}}}$ */
  until $\|(\mathsf{FFT}^{-1}(\hat{s}_1),\mathsf{FFT}^{-1}(\hat{s}_2))\|^2 \le \gamma^2$;
5: $s_1 \leftarrow \mathsf{FFT}^{-1}(\hat{s}_1)$;
6: $s \leftarrow \texttt{Compress}(s_1)$;
  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 $\Sigma_{k=1}^{\infty} p(1-p)^{k-1}\cdot kt = \frac{t}{p} \approx t$ 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 $\texttt{KeyGen}$ 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	$\mathbf{B} \times \hat{\mathbf{B}}^*$	$\Theta(d \log d)$	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: $f \leftarrow \Sigma_i f_i x^i$;
4: $g \leftarrow \Sigma_i g_i x^i$;
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: $F, G \leftarrow$ `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$_n(\mathbf{t}, T)$	$\Theta(d \log d)$	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`$(\mathbf{s}, 8 \cdot$ `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**
    $x_i, y_i \leftarrow$ `UnifCrown`(R_{-}, R_{+}) ; `/* see Algorithm 9 in [9] */`
    $\theta_x, \theta_y \leftarrow \mathcal{U}(0, 1)$;
    $\varphi_{f,i} \leftarrow |x_i| \cdot e^{2i\pi\theta_x}$;
    $\varphi_{g,i} \leftarrow |y_i| \cdot e^{2i\pi\theta_y}$;
  end

2: $(f^{\mathbb{R}}, g^{\mathbb{R}}) \leftarrow \left(\mathsf{FFT}^{-1}((\varphi_{f,i})_{i \le d/2}), \mathsf{FFT}^{-1}((\varphi_{g,i})_{i \le d/2})\right)$;
3: $(\mathbf{f}, \mathbf{g}) \leftarrow (\lfloor f_i^{\mathbb{R}} \rceil)_{i \le d/2}, (\lfloor g_i^{\mathbb{R}} \rceil)_{i \le d/2}$;

4: $(\varphi(f), \varphi(g)) \leftarrow (\mathsf{FFT}(\mathbf{f}), \mathsf{FFT}(\mathbf{g}))$;
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	$\mathsf{FFT}(\mathbf{c})$	$\Theta(d \log d)$	Step 3 of **Alg.** 5	
3	`Sample`$(\hat{c}, sk)$	$\Theta(d \log d)$	Step 4 of **Alg.** 5	See below †
4	$\mathsf{FFT}^{-1}(\hat{s_1})$	$\Theta(d \log d)$	Step 5 of **Alg.** 5	
5	`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 [9],) there are some polynomial multiplications and additions which take $\Theta(d \log d)$ and calls PeikertSampler(**Algorithm** 5 in[9]) two times. In PeikerSampler, 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 Gaussian Sampler

Gaussian sampler plays a significant role in preventing quantum–secure signature schemes from secret key leakage attacks described in [13]. FALCON and SOLMAE use discrete Gaussian sampling with fixed and variable center values for efficient and secure sampling. We describe the theoretical significance of $\mathcal{N}$-`Sampler` (**Algorithm** 10 in [9]) and and the visual analysis `UnifCrown` sampler (**Algorithm 9** in [9]) used in SOLMAE specification [9].

6.1 $\mathcal{N}$-Sampler

A multivariate normal distribution is a natural distribution that is used in many fields. The process of generating a sample that follows normal distribution is called Gaussian sampling. In SOLMAE, we use an $\mathcal{N}$-Sampler (which is the same as the Gaussian sampler) to generate the FFT representation of d–dimensional standard normal distribution. To generate $\mathbb{R}^d$ vector that follows the multivariate normal distribution of mean $\mathbf{0}$ and variance matrix $\frac{d}{2} \cdot I_d$, we can generate independent $\frac{d}{2}$ many random vectors that follow a bivariate normal distribution with mean $\mathbf{0}$ and variance matrix $\frac{d}{2} \cdot I_2$, then concatenate them together. Also, sampling bivariate normal distribution with mean $\mathbf{0}$ and variance matrix $\frac{d}{2} \cdot I_2$ can be done by using Box–Muller transform [5]. We describe how it works: First, generate u_1 and u_2, two independent random numbers, that follow uniform distribution between 0 and 1. Then, compute R and θ as shown below.

$$R = \sqrt{-d \cdot ln(u_1)}, \quad \Theta = 2\pi \cdot u_2 \tag{14}$$

Finally, calculate X and Y, to convert (R, Θ) into Cartesian coordinates.

$$X = R \cdot \cos(\Theta), \quad Y = R \cdot \sin(\Theta) \tag{15}$$

Theorem 6.1. *(X, Y) in Eq.(15) follows bivariate normal distribution with mean* $\mathbf{0}$ *and variance matrix* $\frac{d}{2} \cdot I_2$.

Proof. By using the random variable transform theorem stated in [10], we show that this theorem holds as follows:

$$\begin{aligned}
pdf_{X,Y}(x, y) &= pdf_{U_1,U_2}(u_1, u_2) \cdot \begin{vmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} \end{vmatrix} \\
&= \begin{vmatrix} -\frac{2x}{d} \cdot e^{-\frac{x^2+y^2}{d}} & -\frac{2y}{d} \cdot e^{-\frac{x^2+y^2}{d}} \\ \frac{1}{2\pi} \cdot \frac{-y}{x^2+y^2} & \frac{1}{2\pi} \cdot \frac{x}{x^2+y^2} \end{vmatrix} (u_1 = e^{-\frac{x^2+y^2}{d}}, u_2 = \frac{1}{2\pi} \tan^{-1}(\frac{y}{x})) \\
&= (2\pi)^{-1} \cdot |\Sigma|^{-1/2} exp(-\frac{1}{2} \mathbf{x}^T \Sigma^{-1} \mathbf{x})(\mathbf{x} = (x, y)^T, \Sigma = \frac{d}{2} \cdot I_2)
\end{aligned}$$

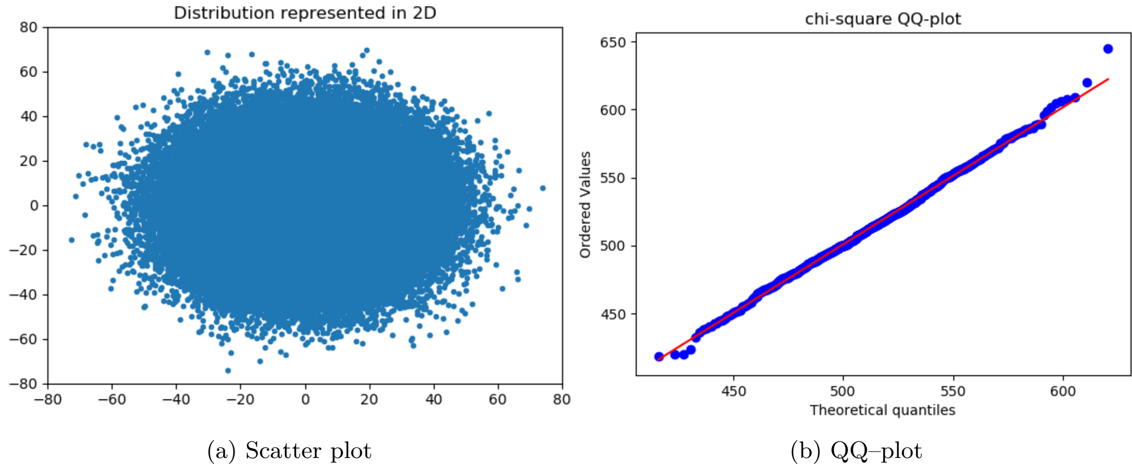
where *pdf* means the *probability density function*. Thus, we see that (X, Y) in Eq.(15) follows the bivariate normal distribution. □

Figure 5 shows 10 bivariate normal samplings using Box–Muller transform generated by Python script.

-1.5600039712701361	-1.1549240059661916
-1.4247377976757196	1.002076190337793
0.9969000368756169	-1.993812973058359
0.7107783282470497	0.0979834381524135
-0.4516874832960174	-0.9235298958094609
0.04449314089974015	1.1053117363335245
-0.9864717691744923	0.020836466309925545
0.887687084897981	-0.010185532828900362
-1.4066801271173832	-0.7906097922917507
0.9722996719071684	-1.6390701046508105

Fig. 5: 10 bivariate normal samplings

To check whether the $\mathcal{N}$-Sampler used in SOLMAE reference implementation generates the multivariate normal distribution of mean zero and variance matrix $\frac{d}{2} \cdot I_d$ properly, we made a checking program in Python script that produces a sample of size 1,000, then plots the random vectors' projections to $\mathbb{R}^2$ and Chi-square QQ-plot using the built-in library provided in Python. Fig. 6 illustrates the 2–dimensional plot of this $\mathcal{N}$-Sampler. Figures 6(a) and 6(b) are its scatter plot and QQ–plot, respectively. From this experiment, we can see that this $\mathcal{N}$-Sampler works properly.



(a) Scatter plot (b) QQ–plot

Fig. 6: Plot of $\mathcal{N}$-`Sampler`

6.2 `UnifCrown` Sampler

`UnifCrown` sampler used in SOLMAE is a method that generates a random vector that follows uniform distribution over $\Omega = \{(x, y) \in \mathbb{R}^2 | R_{min}^2 < x^2 + y^2 < R_{max}^2, x > 0, y > 0\}$ (*i.e.,* the probability density function of random vector (X, Y) is $f_{X,Y}(x, y) = \frac{4}{\pi \cdot (R_{max}^2 - R_{min}^2)} \cdot I_{(R_{min}^2 < x^2 + y^2 < R_{max}^2, x>0, y>0)}$). With some calculations, we can easily see that if U_ρ, U_θ follows uniform distribution over $[0, 1]$, $(X, Y) = \sqrt{R_{min}^2 + U_\rho (R_{max}^2 - R_{min}^2)}(\cos(\frac{\pi}{2} \cdot U_\theta), \sin(\frac{\pi}{2} \cdot U_\theta))$ follows uniform distribution over Ω.

To verify this implementation visually, the scatter plot of `UnifCrown` sampler with 10,000 samples was depicted in Fig. 7. From this, we can see that `UnifCrown` sampler works properly.

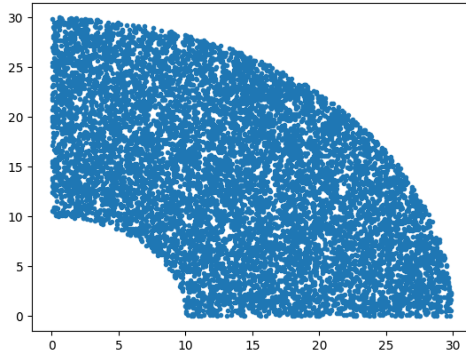


Fig. 7: Scatter plot of `UnifCrown` Sampler

7 Sample Execution and Performance of FALCON–512 and SOLMAE–512

Using Python implementation of SOLMAE in `https://github.com/kjkim0410/SOLMAE_python_512` and FALCON [3], we will describe their practical execution and performance comparison here.

For your clear understanding how FALCON–512 and SOLMAE–512 operate step-by-step, we run the total execution of FALCON–512 and SOLMAE–512 once using the same 512-byte

message and same 40-byte salt which was randomly generated by using `urandom()` in Python `os` module as below:

```
Salt: b730f4e48087d8c5d6dcc085a5ad47437fd4da454c4598142b5284a794660a2cf5322d3425c631c2
Length of Salt: 40

Message: bf467a6d349c6409eba490a9ec34443ad9c009b49a0a0e71974893...d147eb98e818e600e8f6
Length of Message: 512
```

Each algorithm generates a set of secret keys (f, g, F, G) and public key (h) at first using its `KeyGen` procedure. Due to the space problem, we printed out the partial output of the intermediate values such as `HashtoPoint`, s_0 `signature`, s_1 `signature`, `norm of signature` with the `allowable bound of signature`. Also, the values of the message, key, and signal are too big, so they are partially expressed in this paper. The full information of executing FALCON–512 and SOLMAE–512 can refer to `FALCON_512_EX2.txt` and `SOLMAE_512_EX2.txt`, respectively at blog https://ircs.re.kr/?p=1769 for details.

7.1 Sample Execution of FALCON–512

The following is a partial printout of key generation, signification, and verification with FALCON–512.

```
f: [-3, 1, 0, -5, 10, -5, 3, 4, 2, 4, 4, ..., 3, -3, 2, -7, 5, 2, 3, 4, -1, -2]

g: [-3, -3, 4, 5, 5, -6, 10, 1, -4, 3, ..., -9, -3, 1, -2, -7, -3, 7, -2, 0, 2]

F: [23, 10, 2, -16, 14, -26, -20, -1, ..., -7, -14, -24, -21, -23, 18, -1, 44]

G: [-2, 1, 4, 2, -25, 50, 14, 28, -19, ..., -32, -10, -2, 14, 1, -14, -12, 5]

h: [2923, 7873, 9970, 6579, 16, 10828, 337, 8243, ..., 6409, 6857, 2467, 5207]

HashToPoint: [8332, 4711, 5492, 4716, 9558, 8284, ..., 6556, 7525, 11628, 5028]

s0 Signature: [-34, 182,  -7, 82, -86, 113, ..., 204, 212, 349, -65, -89,  -3]

s1 Signature: [-120, -58, 30, 133, 126, 13, ..., -205, -50, 114, -502, 290, 136]

Norm of Signature : 27,222,436
Bound of Signature: 34,034,726

Signature: f8dd47a0abf43635d313e9d5a5dbb7ec2354a805d3...420000000000000000000000
Length of Signature: 666

Verification result = True
```

7.2 Sample Execution of SOLMAE–512

The following is a partial printout of key generation, signification, and verification with SOLMAE–512.

```
f: [0, 1, -3, -6, -3, 2, -4, 1, -1, 0, ..., 2, -3, -5, 0, -5, -5, -1, 1, -4, 3]

g: [5, 3, 5, 2, 0, 4, -10, -6, -3, 1, -2, ..., 2, 4, -2, 4, -3, 3, 3, 2, 2, -3]

F: [-4, 18, 18, -36, 4, -24, 45, 42, 23, 31, ..., 3, 0, 0, 40, 42, -20, -35, 19]
```

```
G: [27, -11, 70, -13, 19, -10, -37, 20, ..., -45, 12, -3, -32, 48, 14, -16, -1]

h: [11703,  2428, 2427, 11947, 9582, 8908, 3567, ..., 6080, 7718, 3106,  11973]

HashToPoint: [8332, 4711, 5492, 4716, 9558, 8284, ..., 6556, 7525, 11628, 5028]

s0 Signature: [74, -6, 253, 187, -285, 210, ..., -225, -195, 364, 325, 138, -44]

s1 Signature: [-50, -47, 198, 13, 218, -201, ...,  114, -234, 5, -119, -41, 170]

Norm of Signature : 30,454,805
Bound of Signature: 33,870,790

Signature: 4ac35f53b674a92dd3ef7f28a9cee2bd2cc48cc8c471d1...9ac80000000000000000
Length of Signature: 666

Verification result = True
```

7.3 Performance Comparison of FALCON–512 and SOLMAE–512

Note that Python code is not so good tool to evaluate the exact performance of FALCON and SOLMAE. However, we can grab a rough idea of their relative performance which one can work fast. The specification of our test platform is Intel Core i7–9700 CPU at 3 GHz clock speed with 16 GRAM. We limited our experiment to the relative performance of `KeyGen`, `Sign`, and `Verif` procedures on FALCON–512 and SOLMAE–512 only. We executed 3 cases of each test which is executed 1,000 times iteration.

Tables 6 and 7 indicates the average time in second of `KeyGen` and `Sign` and `Verif` procedures of FALCON–512 and SOLMAE–512, respectively.

Table 6: Average time of `KeyGen`

	FALCON–512	SOLMAE–512
Test 1	3.6316	2.7346
Test 2	3.6908	2.7633
Test 3	3.7250	2.7306

Table 7: Average time of `Sign` and `Verif`

	Time of `Sign`		Time of `Verif`	
Algo.	FALCON–512	SOLMAE–512	FALCON–512	SOLMAE–512
Test 1	6.4849×10^{-2}	5.9507×10^{-2}	5.4598×10^{-3}	5.4609×10^{-3}
Test 2	6.4664×10^{-2}	5.9432×10^{-2}	5.4359×10^{-3}	5.3684×10^{-3}
Test 3	6.4900×10^{-2}	5.8873×10^{-2}	5.4271×10^{-3}	5.3648×10^{-3}

Our experiments show almost indistinguishable performances between FALCON–512 and SOLMAE–512 by their Python implementation in `Sign` and `Verif` procedures while the `KeyGen` procedure of FALCON takes longer time than that of SOLMAE. We couldn't check the performance of FALCON–1024 and SOLMAE–1024 due to the deadlock issue of our experiment with the limited precision inherited from Python language. The SOLMAE specification [9] states that the `Sign` procedure of SOLMAE takes 2 times faster than that of FALCON by the reference implementation of SOLMAE in C language.

8 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 Python implementation exhibit empirically that `KeyGen` of FALCON–512 only takes longer time than that of SOLMAE–512 by about a second. But we can say that this is not an exact evaluation of their performance by Python implementation.

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

Acknowledgement

This work was partially supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean Government (20ZR1300, Core Technology Research on Trust Data Connectome). Jaehyun Kim and Jueun Jung from Seoul National Unviersity, Korea have partially contributed this work while they were intern researchers at IRCS during their summer break in 2023.

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, 16
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. G. E. P. Box, M.E.M.: A note on the generation of random normal deviatess. Ann. Math. Statist. pp. 610–611 (1958) 15
6. 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
7. 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
8. 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
9. 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, 18
10. Kim, W.: Mathematical Statistics(in Korean). Minyoungsa, Seoul, Korea (2021) 15
11. KpqC: Korean post-quantum crytography (2020), https://kpqc.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
14. NIST: Post-quantum crytography (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