EPiC Series in Computing

Volume 106, 2025, Pages 187–198

Proceedings of the 20th Asia Joint Conference on Information Security

Comparison of Post-Quantum Signatures – FALCON vs SOLMAE with Python

Kwangjo Kim

International Research Institute for Cyber Security (IRCS), KAIST and CSU in Ohio
kkj@kaist.ac.kr

Abstract

This paper presents a comparative analysis of two lattice-based post-quantum digital signature schemes: FALCON and SOLMAE. FALCON which was finallty selected by NIST for PQC standardization, represents an efficient realization of the GPV framework over NTRU lattices. SOLMAE, inspired by FALCON, Mitaka, and Antrag, aims to improve implementation simplicity and performance while preserving strong security guarantees.

Adopting a pedagogical approach, we provide algorithmic insights into both schemes and conduct practical evaluations using their Python implementations, focusing on key generation, signing, and verification procedures. Performance comparisons at two NIST security levels (512 and 1024 bits) highlight SOLMAE's potential advantages in simplicity and execution time, suggesting its suitability for deployment in resource-constrained environments.

1 Introduction

In 1999, Shor [1] proposed an efficient randomized algorithm on a hypothetical quantum computer to integer factorization and discrete logarithm problems in a polynomial time. Currently the threat of attacking the legacy (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 [2] and other notorious ICT companies. We are very concerned about so-called *Harvest Now, Decrypt Later* attack [3] 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.

In 2016, NIST initiated Post Quantum Cryptography (PQC) project [4] to solicit, evaluate, and standardize one or more quantum-resistant cryptographic algorithms for KEM and DS. After several evaluation rounds, NIST announced in 2022 the selection of CRYSTALS-Kyber [5] for KEM and CRYSTALS-Dilithium [6], FALCON [7] and SPHINCS+ [8] for digital signatures. In 2024 the FIPS PUB standard of Kyber, Dilithium and SPHINCS+ were released at [9], [10] and [11], respectively. As of writing this paper, the FIPS PUB standard of FALCON is still in process.

SOLMAE [12] is conceived with inspiration from FALCON's design; Some of the new theoretical foundations were influenced by the proposals of Mitaka [13] and Antrag [14]

A. Yamada, H.K. Kim, Y. Wang and T.-T. Tsai (eds.), AsiaJCIS 2025 (EPiC Series in Computing, vol. 106), pp. 187–198

while maintaining the FALCON's security level across NIST's five level (I to V). In order to thoroughly comprehend FALCON and SOLMAE, one needs not only knowledge of algebra and Gaussian sampling techniques, but also a solid understanding of lattice theory and polynomial arithmetic.

Since Python packages implementing both signature schemes are publicly available, this paper deliberately minimizes complex mathematical exposition. Instead, we adopt a pedagogical approach by focusing on practical, script-based explanations. The primary goal is to make the underlying mechanisms of FALCON and SOLMAE accessible to a broader audience—including students, educators, and developers—through clear Python examples that illustrate key generation, signing, and verification procedures.

The organization of this paper is as follows: Section II introduces the notations and definition used throughout the paper. Sections III and IV provide concise overviews of the FALCON and SOLMAE signature schemes, respectively, including their key generation, signing, and verification procedures. In Section V, we present the implementation details of FALCON–512 and FALCON–1024, along with SOLMAE–512 and SOLMAE–1024, based on their Python code. Section VI offers a performance comparison between FALCON and SOLMAE at two security levels by executing each scheme 1,000 times on a random message. Finally, Section VII provides concluding remarks.

2 Notations

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 $\mathbf{B}$ is denoted $\mathbf{B}^{\mathrm{t}}$. The ℓ_2-norm of a vector $\mathbf{x} = (x_1, \ldots, x_d)$ is $\|\mathbf{x}\| = \left(\sum_i |x_i|^2\right)^{1/2}$.

Lattice

A lattice is a discrete subgroup of the set of n-dimensional real numbers, $\mathbb{R}^n$. Equivalently, it is the set of *integer* linear combinations obtained from a basis $\mathbf{B}$ of $\mathbb{R}^n$. The volume (or determinant) of a lattice is $\det \mathbf{B}$ (which is invariant for all bases of the lattice).

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 $\Lambda(\mathbf{B})$ (the 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 let $f \in \mathbb{Z}[x]/(x^d + 1)$ be a polynomial that is invertible modulo q (equivalently, $\det[f]$ is coprime with q). Let $h = g/f \bmod q$ and consider the NTRU lattice associated to h:

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

This lattice has volume q^d. Over $\mathbb{R}[x]/(\phi)$, where ϕ is a monic minimal polynomial,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 $h = g/f \mod q$, 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.

3 FALCON Signature Scheme

3.1 Overview

FALCON follows a framework introduced in 2008 by Gentry, Peikert, and Vaikuntanathan [15] which we call the GPV framework for short over the NTRU lattices and uses a typically hash–and–sign paradigm. Only the signer should be able to *efficiently* compute a lattice point 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 in brief later by restating the original specification [7].

3.2 Key Generation of FALCON

For the class of NTRU lattices, a trapdoor pair is $(h, \mathbf{B}_{f,g})$ where $h = f^{-1}g, \mathbf{B}_{f,g}$ is a trapdoor basis over $\mathcal{L}_{\text{NTRU}}$ and Pornin & Prest [16] 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 $d = 2^n$.

Algorithm 1 describes the pseudo–code for key generation of FALCON. Readers can refer to **Algorithms 5** and **6** in [7] for details on how to perform `NtruGen` and `NtruSolve`, respectively. Additionally, **Algorithms 9** in [7] explain the procedures for ffLDL*.

3.3 Signing of FALCON

Algorithm 2 sketches the pseudo-code for FALCON signing procedure.

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`

$f, g, F, G \leftarrow \texttt{NtruGen}(\phi, q)$ // Solving the NTRU equation
$\mathbf{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$;
$\hat{\mathbf{B}} \leftarrow \mathsf{FFT}(\mathbf{B})$ // Compute FFT for each $\{g, -f, G, -F\}$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^*$;
T $\leftarrow$ ffLDL$^*(\mathbf{G})$ // Compute the LDL* tree
for each leaf of T do
    $leaf.value \leftarrow \sigma/\sqrt{leaf.value}$ // Normalization step
`sk` $\leftarrow (\hat{\mathbf{B}}, \mathrm{T})$;
$h \leftarrow gf^{-1} \bmod q$;
`pk` $\leftarrow h$;
return `sk`, `pk`

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 $\|(\mathbf{s}_1, \mathbf{s}_2)\| \leq \gamma$.

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

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,and 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 [7]) 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.

3.4 `Compress` and `Decompress` Algorithms

The specification [7] of FALCON suggests encoding and decoding algorithms to reduce the size of keys and signatures. For the sake of simplicity, we skip their detailed description here. The `Compress` and `Decompress` techniques are generic and have no impact on the security level.

3.5 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 [12]. Due to the page limit, the pseudo–code of verification procedure of FALCON is skipped.

4 SOLMAE signature Scheme

4.1 Overview

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 simplicity translates into faster operations while preserving signature and verification key sizes, in addition to allowing for additional features absent from FALCON, such as enjoying less expensive masking, and being parallelizable. In 2023, Espitau *et al.* suggested so–called Antrag in order to improve Mitaka without loss of security covering all NIST 5 levels of security using the degree of cyclotomic ring from 512 to 1024 over specific cyclotomic polynomials under the prime modulus but is not limited to the power of 2. Taking all advantages of FALCON, Mitaka and Antrag, SOLMAE is yet another quantum–safe signature based on NTRU trapdoor and achieves *better performance* for the *same security and advantages* as FALCON which focused only on NIST I and V levels of security. More precisely, SOLMAE offers the "best of three worlds" between FALCON, Mitaka and Antrag. For details on SOLMAE, refer to [12]. The main ingredients of SOLMAE are as follows:

- An **optimally tuned key generation algorithm**, enhancing the security of the new sampler to that of FALCON's security level;
- The **hybrid sampler** is a faster, simpler, parallelizable and maskable Gaussian sampler to generate signatures;
- **Easy implementation** by assembling all the advantages of Mitaka and Antrag to make faster and simpler for practical purposes.

4.2 Key Generation of SOLMAE

An important concern here is that not all pairs $(f, g), (F, G)$ gives good trapdoor pairs for `Sample` described as **Algorithm 4** in [12]. 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:

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

where we recall that $h = g/f \bmod q$.

These parameters and a table of their practical values are described more thoroughly in [12]. 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 3 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.

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

4.3 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.

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 as identical to FALCON. A signature is then to be shortened $\mathtt{sig} = (r, \mathtt{Compress}(s_1))$ using the same compression algorithm in FALCON. SOLMAE cannot output two different signatures for a message like FALCON as mentioned before. **Algorithm 4** shows its pseudo–code of signing procedure in SOLMAE.

Algorithm 4 `Sign` of SOLMAE

Input: A message $M \in \{0,1\}^*$, a tuple $\mathtt{sk} = ((f,g),(F,G),(\widetilde{F},\widetilde{G}),\sigma_{\mathtt{sig}},\Sigma_1,\Sigma_2,\eta)$, a rejection parameter $\gamma > 0$.
Output: A pair $(r, \mathtt{Compress}(\mathbf{s}_1))$ with $r \in \{0,1\}^{320}$ and $\|(\mathbf{s}_1,\mathbf{s}_2)\| \le \gamma$.

$r \leftarrow \mathcal{U}(\{0,1\}^{320})$
$\mathbf{c} \leftarrow (0, \mathtt{H}(r||M))$
$\hat{\mathbf{c}} \leftarrow \mathsf{FFT}(\mathbf{c})$
while $\|(\mathsf{FFT}^{-1}(\hat{s}_1), \mathsf{FFT}^{-1}(\hat{s}_2))\|^2 \le \gamma^2$ **do**
  $(\hat{s}_1, \hat{s}_2) \leftarrow \hat{\mathbf{c}} - \mathtt{Sample}(\hat{\mathbf{c}}, \mathtt{sk})$ // $(\mathbf{s}_1,\mathbf{s}_2) \leftarrow D_{\mathcal{L}_{\mathrm{NTRU}},\mathbf{c},\sigma_{\mathtt{sig}}}$
end while
$s_1 \leftarrow \mathsf{FFT}^{-1}(\hat{s}_1)$
$s \leftarrow \mathtt{Compress}(s_1)$
return (r, s)

4.4 Verification of SOLMAE

The verification procedure used in SOLMAE is identical to that of used in FALCON.

5 Python Implementation

Using the Kim's monograph[17], we will discuss how to verify the functionality of both FALCON and SOLMAE by utilizing their Python modules as well as scheme-specific Python implementations. This includes examining the common interface used for key generation, signing, and verification, along with the distinct internal components and optimizations that are unique to each scheme.

5.1 Checking FALCON with Python

`https://github.com/tprest/falcon.py` contains an implementation of the FALCON signature scheme in Python at github repository. This repository contains the following files (roughly in order of dependency):

1. `common.py` contains shared functions and constants
2. `encoding.py` contains compression and decompression
3. `rng.py` implements a ChaCha20-based PRNG, useful for KATs (standalone)
4. `samplerz.py` implements a Gaussian sampler over the integers (standalone)
5. `fft_constants.py` contains precomputed constants used in the FFT
6. `ntt_constants.py` contains precomputed constants used in the NTT
7. `fft.py` implements the FFT over $R[x]/(x^n+1)$
8. `ntt.py` implements the NTT over $Z_q[x]/(x^n+1)$
9. `ntrugen.py` generate polynomials f,g,F,G in $Z[x]/(x^n+1)$ such that $f \cdot G - g \cdot F = q$
10. `ffsampling.py` implements the fast Fourier sampling algorithm
11. `falcon.py` implements FALCON
12. `test.py` implements tests to check that everything is properly implemented

5.1.1 Checking `falcon.py`

This section describes the correctness of executing FALCON–512 and FALCON–1024 from the predetermined polynomials, f, g, F, and G which is provided as `falcon.py` in the FALCON Python package simply. The value of `n` is fixed at 512 or 1024 depending on which version of FALCON you are verifying.

By setting the value of `n` to 512 or 1024, Figs. 1 and 2 present an example of generated key pairs, a random message, its signature in hexadecimal notation, the verification of signature for FALCON–512 and FALCON–1024, respectively.

```
** Testing keygen, sign and verify procedures of Falcon-512
Test Case : 1
== Leading 10 values of private key
f = [1, -3, 0, 4, 0, 5, -3, -4, 4, -2, ...]
g = [-4, -7, 4, -2, 3, 3, -2, 4, -7, -1, ...]
F = [30, -32, -19, 0, -14, 46, -28, -18, 1, 19, ...]
G = [-25, -14, 10, 8, 28, 18, 7, 12, 34, -18, ...]
== Leading 10 values of public key
h = [11496, 8750, 6367, 8513, 9698, 2801, 11184, 7720, 3044, 6551, ...]
Messge  =  b'zyggautvsowlwphccrpgbaxlcj'
Signature = 393d488cde1b60858f3c5c23944a81 ... 00000000000000000000
Length of Signature: 666 Bytes
Verification passed!!
```

Figure 1: An example of `KeyGen`, `Sign` and `Verif` of Falcon-512

```
** Testing keygen, sign and verify procedures of Falcon-1024
Test Case : 1
== Leading 10 values of private key
f = [2, 1, 3, 3, 2, 2, 2, -4, 2, 0, ...]
g = [3, -3, -3, -1, 2, 3, 1, -2, 0, 2, ...]
F = [14, -13, -33, 46, 31, 8, 12, 29, 22, -2, ...]
G = [-58, 19, -8, 0, 6, -11, 2, 1, -1, 20, ...]
== Leading 10 values of public key
h = [3680, 7862, 6250, 919, 1038, 11753, 2971, 2770, 12273, 2831, ...]
Messge  =  b'rtkczofilqmoajhnnjrzycsojd'
Signature = 3a0b6d34bfc1b8baf6c08a409f9bbf ... bc24999fe5424e1a0000
Length of Signature: 1280 Bytes
Verification passed!!
```

Figure 2: An example of `KeyGen`, `Sign` and `Verif` of Falcon-1024

5.2 Checking SOLMAE with Python

SOLMAE Python package is available at web page: `https://solmae-sign.info`. To implement the SOLMAE in Python, the modules used for FALCON such as 1) `common.py` ... 9) `ntrugen.py` are re-used, as their functionalities are also essential for the basic operation of SOLMAE.

The SOLMAE-specific modules are as follows:

1. `params.py` contains security parameters
2. `Unifcrown.py` implements Unifcrown sampler and its test script
3. `Pairgen.py` implements Pairgen and its test script
4. `keygen.py` implements keygen and its test script
5. `PeikertSampler.py` implements Peikert Sampler
6. `N_sampler.py` implements N-sampler
7. `Sampler.py` implements Sampler

8. `solmae.py` implements keygen, sign and verify procedures of SOLMAE–512 or SOLMAE–1024
9. `test.py` contains how to use and to check that everything is properly implemented.(same as FALCON Python Package)

5.3 Checking `solmae.py`

This section describes the `KeyGen`, `Sign`, and `Verif` procedures of SOLMAE–512 and SOLMAE–1024 from the randomly generated private and its corresponding public key.

The value of `SOLMAE_D` in `params.py` is fixed at 512 or 1024 depending on which type of SOLMAE you are verifying. Depending on the value of `SOLMAE_D` in `params.py`, Figs. 3 and 4 present one test of randomly generated key data, a random 512 byte message, its signature in hexadecimal notation, the verification of signature for SOLMAE–512 and SOLMAE–1024, respectively.

```
**Testing of keygen, sign and verify procedures of SOLMAE-512
<< Test Case : 1 >>
==Leading 5 values of keygen function for SOLMAE-512
f            = [-5, 5, 3, 0, 2, ...]
g            = [-2, -1, 0, -5, -3, ...]
F            = [15, -3, 17, -36, -9, ...]
G            = [-29, -4, 22, -4, -37, ...]
h            = [10584, 7983, 3214, 11619, 2601, ...]
f_fft        = [18.000-106.311j, 51.018-54.726j, 98.702-53.016j, ...]
g_fft        = [33.266+53.408j, -56.722+65.946j, -0.474+5.675j, ...]
F_fft        = [491.941+455.316j, 251.152+124.845j, 239.141+322.386j, ...]
G_fft        = [-372.041+103.780j, -185.559-13.212j, 76.266+53.164j, ...]
beta10_fft   = [18.000-106.311j, 51.018-54.726j, 98.702-53.016j,  ...]
beta11_fft   = [33.266+53.408j, -56.722+65.946j, -0.474+5.675j, ...]
beta20_fft   = [491.941+455.316j, 251.152+124.845j, 239.141+322.386j, ...]
beta21_fft   = [-372.041+103.780j, -185.559-13.212j, 76.266+53.164j, ...]
Sigma1       = [0.377, 0.705, 0.776, ...]
Sigma2       = [1.148, 0.914, 0.848, ...]
Message = 3ec29088765e80f921aeb648ea26a2f990774b0f ... c24af424a492e304a
Length of Message:  512
Signature = 3904988187c26069f7c51cca41f625f9f53096c2 ... 5b880000000000000
Length of Sig. =  666 Bytes
Verification passed!!
```

Figure 3: An example of `KeyGen`, `Sign` and `Verif` of SOLMAE–512

6 Comparison of FALCON and SOLMAE

While performance evaluation using Python implementations may not provide an exact measure of the true performance of FALCON and SOLMAE, it offers a rough indication of their relative efficiency. The platform used for our comparison features an Intel Xeon E3-1230 v3 CPU running at 3.3GHz with 32GB RAM under Windows OS. We conducted experiments to measure the average execution time in `msec` of the `KeyGen`, `Sign`, and `Verif` procedures for both FALCON and SOLMAE, iterating 1,000 times on random messages ranging from 512 to 1024 bytes. Tables 1 and 2 summarize the average times in msec for `KeyGen`, `Sign`, and `Verif` procedures for FALCON–512 vs SOLMAE–512 and FALCON–1024 vs SOLMAE–1024 (successful `KeyGen`'s count only by removing deadlock count), respectively. Our experimental results indicate that, in terms of execution time across the three procedures, the two signature schemes exhibit similar performance.

```
**Testing of keygen, sign and verify procedures of SOLMAE-1024
<< Test Case : 1 >>
==Leading 5 values of keygen function for SOLMAE-1024
f            = [-3, -2, -4, -1, 5, ...]
g            = [2, -1, -2, 1, -2, ...]
F            = [-22, 10, -21, -9, 3, ...]
G            = [30, 3, 58, -15, -14, ...]
h            = [6723, 11180, 8130, 1914, 7378, ...]
f_fft        = [-104.280-96.797j, -69.196-56.365j, 34.935-44.066j, ...]
g_fft        = [21.710+25.918j, 67.878+30.724j, -55.365+120.224j, ...]
F_fft        = [-226.288+2151.829j, 35.686+60.839j, -400.363+278.509j, ...]
G_fft        = [53.959-441.838j, -148.546+45.477j, 896.346-688.553j, ...]
beta10_fft   = [-104.280-96.797j, -69.196-56.365j, 34.935-44.066j,  ...]
beta11_fft   = [21.710+25.918j, 67.878+30.724j, -55.365+120.224j, ...]
beta20_fft   = [-226.288+2151.829j, 35.686+60.839j, -400.363+278.509j, ...]
beta21_fft   = [53.959-441.838j, -148.546+45.477j, 896.346-688.553j, ...]
Sigma1       = [0.998, 1.624, 1.045, ...]
Sigma2       = [2.592, 1.891, 2.537, ...]
Message = 9ba020beb78f34e3b1f52596c519c754cd6079ed ... d8ba0867d484b8fc6
Length of Message:  1024
Signature = 3adeee535b7f475a7cdcf9fb6ac0874885711af9 ... 0000000000000000
Length of Sig. =  1375 Bytes
Verification passed!!
```

Figure 4: An example of `KeyGen`, `Sign` and `Verif` of SOLMAE–1024

Table 1: Comparison of FALCON–512 vs SOLMAE–512

	FALCON–512	SOLMAE–512
`KeyGen`	6,029.52	4,100.78
`Sign`	35.38	47.33
`Verif`	9.52	9.88

7 Concluding Remark

In this work, we explored the design, implementation, and performance of two prominent lattice-based post-quantum digital signature schemes: FALCON and SOLMAE. Through Python-based simulations, we demonstrated that both schemes are efficient and secure, with SOLMAE offering notable improvements in implementation simplicity and marginal gains in key generation performance. These findings suggest that SOLMAE may be a strong candidate for deployment in resource-constrained systems. Future work will involve low-level implementations (e.g., in C), as well as comprehensive evaluations of side-channel resistance and hardware-level optimizations to assess practical deployment feasibility.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.

[2] IBM, "Expanding the IBM quantum roadmap to anticipate the future of quantum-centric supercomputing," 2022, https://research.ibm.com/blog/ibm-quantum-roadmap-2025.

[3] Wikipedia, "Harvest now, decrypt later," 2023. [Online]. Available: https://en.wikipedia.org/wiki/\Harvest_now_decrypt_later

[4] NIST, "Post-quantum crytography," 2016, https://csrc.nist.gov/projects/post-quantum-cryptography.

Table 2: Comparison of FALCON–1024 vs SOLMAE–1024

	FALCON–1024	SOLMAE–1024
`KeyGen`	41,816.15	35,492.70
`Sign`	674.81	464.14
`Verif`	593.39	343.99

[5] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, D. Stehle, and J. Ding, "Crystal–kyber," https://pq-crystals.org/kyber/index.shtml.

[6] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehle, and S. Bai, "Crystal–dilithum," https://pq-crystals.org/dilithium/index.shtml.

[7] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info/.

[8] A. Hulsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens, "Sphincs+," https://sphincs.org/.

[9] National Institute of Standards and Technology, "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard," August 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203

[10] ——, "FIPS 204: Module-Lattice-Based Digital Signature Standard," August 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.204

[11] ——, "FIPS 205: Stateless Hash-Based Digital Signature Standard," August 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.205

[12] K. Kim, M. Tibouchi, T. Espitau, A. Takashima, A. Wallet, Y. Yu, S. Guilley, and S. Kim, "Solmae : Algorithm specification," Updated SOLMAE, IRCS Blog, 2023, https://ircs.re.kr/?p=1714.

[13] T. Espitau, P.-A. Fouque, F. Gérard, M. Rossi, A. Takahashi, M. Tibouchi, A. Wallet, and Y. Yu, "Mitaka: A simpler, parallelizable, maskable variant of FALCON," in *Advances in Cryptology – EUROCRYPT 2022*, O. Dunkelman and S. Dziembowski, Eds. Cham: Springer International Publishing, 2022, pp. 222–253.

[14] T. Espitau, T. T. Q. Nguyen, C. Sun, M. Tibouchi, and A. Wallet, "Antrag: Annular ntru trapdoor generation," *Proc. of Asiacrypt2023, Part VII, Guangzhou, China*, pp. 3–32, 2023.

[15] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*. Victoria, British Columbia, Canada: ACM, 2008, pp. 197–206.

[16] T. Pornin and T. Prest, "More efficient algorithms for the ntru key generation using the field norm," in *Public-Key Cryptography – PKC 2019*, D. Lin and K. Sako, Eds. Cham: Springer International Publishing, 2019, pp. 504–533.

[17] K. Kim, *Practical Post-Quantum Signatures: FALCON and SOLMAE with Python*, ser. SpringerBriefs in Information Security and Cryptography. Springer, 2025.