EagleSign : A new post-quantum ElGamal-like signature over lattices

Djiby Sow[(1)], sowdjibab@yahoo.fr, djiby.sow@ucad.edu.sn
Abiodoun Clement Hounkpevi[(1)], abiodounkpevi@gmail.com
Sidoine Djimnaibeye[(1)], dthekplus@gmail.com
Michel Seck[(2)], michelseck2@gmail.com

[1] Cheikh Anta Diop University Dakar Senegal
[2] Ecole Polytechnique Thies Senegal

Table of Contents

Introduction

EagleSin : *In this document we present EagleSign signature which can be seen as a variant of Elgamal signature over structured lattices. It is more simple and faster than Falcon and Dilithium signatures proposed by NIST for standardization. The sizes of EagleSign are similar to those of Dilithium. In the particular case of recommended parameters, EagleSin is really more small and it saves* 1000 *bytes relatively to Dilithium when computing (bitsize of the public key) + (bitsize of the signature algorithm).*

Given the recent advancements in quantum computing and the fact that the classical Integer Factorization Problem and the Discrete Logarithm Problem are not secure against quantum computers [69], the scientific community want to design cryptosystems and protocols that resist to attacks by quantum technologies.

For this reason, the National Institute of Standards and Technology (NIST), by a call for submissions [52], propose the transition to quantum-resistant cryptography. Many algorithms for public-key encryption, key encapsulation mechanism, and digital signature were proposed throughout 3 rounds. Many authors have worked on the categorization (according to the family of underlying problem) and the performance analysis of the schemes proposed to NIST [21, 30, 50, 53]. There were 3 evaluation criteria for the case of digital signature schemes: (1) security (Zero knowledge property, security proof in ROM/QROM, Side Channel Attacks mitigation, hardness of the underlying problem), (2) cost and performance, and (3) algorithm and implementation characteristics on software and hardware.
In July 2022, at the end of the 3rd round, regarding the post-quantum digital signatures, there were 3 candidates proposed for NIST standardization: one MLWE-based signature (CRYSTALS-Dilithium), one NTRU-based signature (FALCON) and one hash-based signature (Sphincs+).

Summary of (Module) Falcon: Falcon and its generalization ModFalcon are based on the framework for lattice-based signature schemes proposed by Gentry, Peikert and Vaikuntanathan : hash-and-sign paradigm upon collision-resistant preimage sampleable function [32]. The underlying hard problem in Falcon is NTRU-SIS (Short Integer Solution problem over NTRU public key) together with the "Fast Fourier sampling (FFT)" as a trapdoor sampler. In the ring $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^n+1)}$, the NTRU public key of Falcon is $h = f^{-1}g \mod q, q = 12289, n = 512, 1024$ where f, g are small and sparse polynomials in R_q. The NTRU-SIS hardness is based on the difficulty of recovering the polynomials f and g given the polynomial ring element h. In quantum or classical world, no efficient attack is currently known to break the computational NTRU-SIS or the Decisional Small Polynomial Ratio (DSPR) assumption of NTRU whenever f and g are suitably chosen. In Falcon, after computing f and g from an appropriate distribution, the key generation algorithm computes F and G such that $fG-gF = q \mod X^n+1$. The polynomials f, g, F, and G are stored in the private key sk. To sign a message m, Falcon uses a hash function H, a private key sk, a salt $r, |r| = 64$ and a FFT sampler to compute short vectors s_1, s_2 that satisfy the equation: $s_1 + s_2h = H(r, m)$. Falcon is the most compact (most small size) signature among those proposed to NIST competition but it is based directly on cyclotomic ring and does not allow various security levels.
ModFalcon is introduced by Chuengsatiansup, Prest, Stehlé, Wallet and Xagawa (ASIACCS '20) and it generalizes Falcon to modules where the public key is $\mathbf{H} = \mathbf{F}^{-1}\mathbf{G} \mod q$ where $\mathbf{F}$,(resp: $\mathbf{G}$) is $m \times m$ (resp: $m \times k$) matrix with short entries in R_q. In [25], they instantiated a particular case where $k = 1, q = 12289$, and $n = 256$. Moreover, in the IBE scheme (IACR ePrint 2019/1468) the authors Cheon, Kim, Kim and Son chose $m = 1$. ModFalcon allows an intermediate security level that is missing in Falcon signature.

Fiat-Shamir Transformation: The Fiat-Shamir transformation was proposed by Fiat and Shamir [29] as a framework that allows to derivate a signature from an Identification Protocol (ID) by removing the interaction in ID throughout a hash function.

Summary of Dilithium (hight level description): Crystals Dilithium is a Fiat-Shamir signature with aborts over lattices based on MLWE and MSIS hard problems which is based on Vadim Lyubashesky previous works in 2009 and 2012 [46,47]. In Dilithium, the security of the public keys is based on MLWE and the security of the signature against forgery is based on MSIS and SelfTargetMSIS problems. The public key with MLWE over $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^n+1)}, q = 2^{23} - 2^{13} + 1, n = 256$ is $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ where $\mathbf{A} \in R_q^{k\times l}$ is a public matrix generated uniformly at random and the secrets $sk = (\mathbf{s}_1, \mathbf{s}_2) \in R_q^l \times \in R_q^k$ are generated uniformly at random such that $|\mathbf{s}_1|_\infty, |\mathbf{s}_2|_\infty \leq \eta$ (a short integer). To sign a message m, Dilithium uses a hash function H, the private key sk to compute an ephemeral public key $\mathbf{d} = \mathbf{A}\mathbf{y}$ (together with an ephemeral secret key $\mathbf{y}$), a sparse challenge $c = H(\mathbf{d}, m)$ and sets $\sigma = (\mathbf{z}, c, \mathbf{h})$ as signature where $\mathbf{z} = c\mathbf{s}_1 + \mathbf{y} \in R_q^l$ and $\mathbf{h}$ is a hint vector. To protect $\mathbf{z}$, a "while loop" for rejection sampling containing few steps is included in the process before a valid signature with zero knowledge property is obtained. For this, a counter is incremented in every loop to generate a different ephemeral secret key $\mathbf{y}$ in each iteration. To reduce the size of the signature a special technique based on rounding and hight bits is used. Dilithium has two variants according to the way the ephemeral secret key $\mathbf{y}$ is generated (deterministic or probabilistic).
Recently many other signatures based on NTRU/MNTRU and RLWE/MLWE were proposed [17,55].

Summary of EagleSign (hight level description): EagleSign is a signature without aborts over lattices. We denote by $q = 12289, n \in \{512, 1024\},\ S_\eta = \{u \in R_q / |u|_\infty \leq \eta\}$ the polynomials in $\mathcal{R}_q$ whose $l\infty$ norm is tightly upper-bounded by η.
The public key over $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^n+1)}$ (where q is a prime) is $\mathbf{E} \in R_q^{k\times l}$ where $\mathbf{E} = (\mathbf{A}\mathbf{F}^{-1} + \mathbf{D})\mathbf{G}^{-1}$, $\mathbf{A} \in R_q^{k\times l}$ is a public matrix generated uniformly at random and the secrets $\mathbf{F} \in S_{\eta_F}^{l\times l}, \mathbf{G} \in S_{\eta_G}^{l\times l}$ (resp: $\mathbf{D} \in S_{\eta_D}^{k\times l}$) are invertible matrices of small polynomials generated uniformly at random (resp: matrix of small polynomials generated uniformly at random). Note that $\mathbf{F}$ or $\mathbf{G}$ can be a constant or a polynomial suitably chosen. The secret key is then $sk = (\mathbf{F}, \mathbf{G}, \mathbf{D}) \in S_{\eta_F}^{l\times l} \times S_{\eta_G}^{l\times l} \times S_{\eta_G}^{k\times l}$.
Note that, to sign a message M, EagleSign:

- uses two hash functions H, G (H is modeled as a random oracle in ROM security proof) and a private key sk to compute an ephemeral public key $\mathbf{P} = \mathbf{A}\mathbf{F}^{-1}\mathbf{Y}_1 + \mathbf{Y}_2 \in R_q^{k\times m}$ (together with an ephemeral secret key $(\mathbf{Y}_1, \mathbf{Y}_2) \in S_{\eta_{y_1}}^{l\times m} \times S_{\eta_{y_2}}^{k\times m}$), a challenge $\mathbf{C} \in S_{\eta_c}^{l\times m}$ derived from $H(M, r)$ where $r =: G(\mathbf{P})$
- and sets $\sigma = (r, \mathbf{Z}, \mathbf{W})$ where $\mathbf{Z} = \mathbf{G}\mathbf{U} \mod q$, $\mathbf{U} = \mathbf{Y}_1 + \mathbf{F}\mathbf{C} \mod q \in R_q^{l\times m}$ and $\mathbf{W} = \mathbf{Y}_2 - \mathbf{D}\mathbf{U} \mod q = (\mathbf{Y}_2 - \mathbf{D}\mathbf{Y}_1) - \mathbf{D}\mathbf{F}\mathbf{C} \mod q \in R_q^{k\times m}$.

In practice, $\eta_c = \eta_{y_1} = 1$, and we choose $S_{\eta_c} = B_\tau$, $S_{\eta_{y_1}} = B_t$ and $\mathbf{C} \in B_\tau^{l\times m}$, $\mathbf{Y}_1 \in B_t^{l\times m}$ where $B_\tau = \{f \in R_q / f = \sum_{i=0}^{i=n-1} f_i X^i, f_i \in \{-1, 0, 1\}\ |f|_1 = \sum_{i=0}^{i=n-1} |f_i| = \tau\}$ is the ball of sparse ternary polynomials with hamming weight τ.

The two components of our longterm public key $\mathbf{E} = (\mathbf{AF}^{-1} + \mathbf{D})\mathbf{G}^{-1} \in R_q^{k\times l}$ and ephemeral public key $\mathbf{P} = \mathbf{AF}^{-1}\mathbf{Y}_1 + \mathbf{Y}_2 \in R_q^{k\times m}$ are a mix of MNTRU and MLWE. Most of the known techniques to break RLWE and NTRU can not trivially be generalized to our public key. We hope that using together MNTRU and MLWE in the same public key allows to make more complex the algebraic and geometric properties of the underlying lattice and we thus think that we are moving away a little from strong structured lattices.

In the signature, the zero-knowledge property ensures that the signing process does not reveal any information about the secret key associated to the public key used in the verification process. In order to obtain the zero-knowledge property without multiple rejections sampling used in lattices based signatures, we introduce two additional masks: an additive mask $\mathbf{Y}_1$ and a multiplicative mask $\mathbf{G}$ to obtain the new signature $\mathbf{Z} = \mathbf{G}(\mathbf{Y}_1 + \mathbf{FC}) \mod q, \mathbf{W} = \mathbf{Y}_2 - \mathbf{D}(\mathbf{Y}_1 + \mathbf{DFC})$ mod q where $\mathbf{G}, \mathbf{F}, \mathbf{D}$ are the longterm secrets, $(\mathbf{Y}_1, \mathbf{Y}_2)$ is the ephemeral secret for probabilistic signature.

EagleSign do not use the auxiliary functions of Crystals Dilithium such as HighBits, MakeHint, UseHint, Power2Round, Decompose and SelfTargetMSIS therefore the corresponding pseudo-code can be more simple and compact.
Since the ephemeral public key $\mathbf{P} = \mathbf{AF}^{-1}\mathbf{Y}_1 + \mathbf{Y}_2$ is integrally recovered during the verification process, then we don't need to use the SelfTargetMSIS problem of Dilithium in the security proof.

The security in ROM follows from the general framework using the forking lemma. EagleSign allows more flexibility to upgrade easily the security level in the future. We prove that our signature is secure in ROM by forking lemma and we verify with Crystal tool of Dilithium (for MSIS) and the lattice-estimator for security of Albrecht, et *al.* [2–4] (for LWE) that EagleSign reach the 3 fundamental NIST security levels only with $F = 1, m = 1$ and $k, l \in \{1, 2\}$. We have the following sizes and security results according to NIST security level for each variant for instantiation. The table 2 presents the code efficiency of EagleSign level 3 and 5 based on our specific processor characteristics. The Level 2 and 3+ are not yet implemented because of lack of time thus the corresponding efficiency characteristic is not available.

A comparison between EagleSign, Falcon and Dilithium is done in the section 'performance' below and we remark that EagleSign is more faster and simple than Falcon and Dilithium. For recommended parameters, the sizes of EagleSign are more small than those of Dilithium but the sizes of EaglaSign are similar to those of Dilithium for level 2 and 5.

Organization of the paper: This paper is organized as follows.

Table 1. Parameters Selection for Size and NIST Security Levels

EagleSign				
NIST security level	2	3	3+	5
$F = 1, m = 1$ and $k, l \in \{1, 2\}$				
	Medium	Recomm I	Recomm II	High I
(k, l)	$(2, 1)$	$(1, 1)$	$(2, 2)$	$(1, 2)$
$q = 12289, n =$	512	1024	512	1024
$\eta_c = 1, \mathbf{c} \in B_\tau^l, \tau =$	18	38	38	38
Strong Unforgeable signature:				
$\beta = \max(2\delta', 2\delta'), (\delta, \delta'),$	$(948, 1012)$	$(178, 242)$	$(432, 248)$	$(208, 240)$
BKZ block-size b to break SIS	607	867	748	869
Best Known Classical bit-cost	177	253	218	253
Best Known Quantum bit-cost	160	229	198	230
Best Plausible bit-cost	125	179	155	180
Ephemeral secret recovery $(\mathbf{Y}_1, \mathbf{Y}_2)$				
$\eta_{y_1} = 1, \mathbf{Y}_1 \in B_t^l, (t, \eta_{y_2}) =$	(140,64)	(140,64)	(90,32)	(86,32)
BKZ block-size b to break LWE	439	713	764	870
Best Known Classical bit-cost	128	208	223	254
Best Known Quantum bit-cost	116	188	202	230
Best Plausible bit-cost	91	147	158	180
Longterm secret recovery $(\mathbf{G}, \mathbf{D})$				
(η_G, η_D)	$(6, 6)$	$(1, 1)$	$(2, 1)$	$(1, 1)$
BKZ block-size b to break LWE	444	696	741	1649
Best Known Classical bit-cost	129	203	216	481
Best Known Quantum bit-cost	117	184	196	436
Best Plausible bit-cost	92	144	153	342
Size in bytes:				
signature size $(r, \mathbf{Z}, \mathbf{W})$	2144	2336	2464	3488
public key size $(\rho, \mathbf{E})$	1824	1824	3616	3616

Table 2. EagleSign Performances for NIST Security Levels 3 and 5

EagleSign Performance (12th Gen Intel Core i7-1260P $\times$ 16, RAM 16GB)		
NIST security level	3	5
$F = 1, m = 1$ and $k, l \in \{1, 2\}$		
(k, l)	$(1, 1)$	$(1, 2)$
τ	38	18
(n, q)	$(1024, 12289)$	$(1024, 12289)$
$\beta = \max(2\delta', 2\delta'), (\delta, \delta'),$	$(178, 242)$	$(208, 240)$
$\eta_{y_1} = 1, (t, \eta_{y_2})$	(140,64)	(86,32)
(η_G, η_D)	$(1, 1)$	$(1, 1)$
Reference Implementation		
Gen median cycles	1001330	3345416
Gen average cycles	1020723	3443617
Sign median cycles	1274146	2351304
Sign average cycles	1283454	2358603
Verif median cycles	946459	1594736
Verif average cycles	955956	1602340
Optimized Implementation		
Gen median cycles	978368	3212708
Gen average cycles	1000579	3287036
Sign median cycles	1286413	2251036
Sign average cycles	1241245	2259111
Verif median cycles	917213	1503004
Verif average cycles	927108	1512331

– In Section 1, we recall some useful nations and we define basic operations and maps.
– In Section 2, we propose the specification of EagleSign.
– The Section 3 is devoted to security analysis and parameters selection.
– In Section 4, we study the performance (sizes and cycles) according various security levels.
– In Section 5, we explain at high level how the reference and optimized implementations were done.
– And finally, in Section 6, we summarize the limitations and advantages of EagleSign.

NIST Requirements
As Falcon, here we propose a mapping of the requirements by NIST in June 2022 (Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process) to the appropriate sections of the current document.

– The complete specification as per [NIST Call 2022 [54], Section 2.B.1] can be found in Section 2
– The security analysis of EagleSign as per [NIST Call 2022 [54], Section 2.B.4], the study of known cryptographic attacks against the scheme of as per [NIST Call 2022 [54], Section 2.B.5], and the set of parameters corresponding to the security levels 2, 3, 3+, and 5 [NIST Call 2022 [54], Section 4.A.5] are contained in Section 3. We use the lattice-estimator for the security of the longterm public key and the ephemeral public key based on a mix of MLWE and MNTRU problems. We use the tool of Dilithium to estimate the security for unforgeability relatively to MSIS problem.
– A performance analysis and a comparison with Dilithium, as per [NIST Call 2022 [54], Section 2.B.2], is provided in Section 4.
– A summary of the reference implementation and the optimized implementation as per [NIST Call 2022 [54], Section 2.C.1] can be founded in Section 5.
– Based on a comparison with Falcon and Dilithium, a statement of the advantages and limitations as per [NIST Call 2022 [54], Section 2.B.6] can be found in Section 6.

The following requirements in [NIST Call 2022 [54]] are in EagleSign submission package:

– a cover sheet as per [NIST Call 2022 [54], Section 2.A],
– a reference implementation and an optimized implementation as per [NIST Call 2022 [54], Section 2.C.1] and Known Answer Test values as per [NIST Call 2022 [54], Section 2.B.2],
– all signed statements of intellectual property, as required by [NIST Call 2022 [54], Section 2.D].

1 Preliminaries

1.1 Notations and elementary operations

In this subsection we use the same notations than Falcon, Bliss and Dilithium.

- The underlying rings of our signatures are $\mathcal{R} = \mathbb{Z}[x]/(X^n + 1), \mathcal{R}_q = \mathbb{Z}_q[x]/(X^n + 1)$ where q is prime, $q = 12289, n = 512, 1024$.
- Regular font letters denote polynomials in R or R_q or elements in $\mathbb{Z}$ and $\mathbb{Z}_q$, bold lower-case letters represent column vectors of length l in in R^l or R_q^l and bold upper-case letters are matrices in $R^{k\times l}$ or $R_q^{k\times l}$ thus for $v, \mathbf{v}, \mathbf{V}$ the notation says that v is a scalar or a polynomial, $\mathbf{v}$ is a vector, and $\mathbf{V}$ is a matrix. For a vector $\mathbf{v}$ (resp: matrix $\mathbf{V}$), we denote by $\mathbf{v}^T$ (resp: $\mathbf{V}^T$) its transpose.
- For an odd positive integer p , we define $r = z \mod {}^{\pm}p$, the centred reduction modulo p, to be the unique element r in the range $\frac{p-1}{2} \leq r \leq \frac{p-1}{2}$ such that $r \cong z \mod p$. We consider that $\mathbb{Z}_p = \{-\frac{p-1}{2}, \dots, -1, 0, 1, \dots, \frac{p-1}{2}\}$, thus $r = z \mod {}^{\pm}p = z \mod p$ to simplify the notation throughout equations.
- S_η is the set of small polynomials which means that the element of S_η are polynomials with coefficients are in the interval $[-\eta, +\eta]$ and $B_\tau = \{f \in R_q / f = \sum_{i=0}^{i=n-1} f_i X^i, f_i \in \{-1, 0, 1\}\ |f|_1 = \sum_{i=0}^{i=n-1} |f_i| = \tau\}$ is the ball of sparse ternary polynomials. The entropy of B_τ is $\log \#B_\tau$ where $\#B_\tau = 2^\tau \binom{n}{\tau}$. The value of τ will be chosen such that the entropy of B_τ is greater than the security level.
- For $f = \sum_{i=0}^{i=n-1} f_i X^i \in R_q, -\frac{p-1}{2}, \leq f_i \leq \frac{p-1}{2}$, we denote $|f|_\infty = \max_i |f_i|$. We have $|fg|_\infty \leq |f|_1 |g|_\infty$.
- For $\mathbf{v} = (v_0, \dots, v_{k-1})^T \in\in R_q^k$, we denote $|\mathbf{v}|_\infty = \max_i |v_i|_\infty$.
- The coefficients of the polynomials in $\mathcal{R}_q$ are in $[-(q-1)/2; (q-1)/2]$.

1.2 Hashing

Hashing to a Ball: We hash in the ball B_τ defined above as follows. As Dilithium, we use two steps.
Step 1: In this step, a 2nd pre-image resistant cryptographic hash function maps $\{0,1\}^\star$ onto the domain $\{0,1\}^N$ where $N = 512$ or 1024;
Step 2: the previous step is followed by and eXtendable Output Function (XOF) (modelled here with SHAKE) that maps the output of the first stage to an element of B_τ with the following algorithm :

- Initialize $c = c_0 c_1 \dots c_{N-1} = 0 \dots 0$
- for $i = N - \tau$ to N
  - $b \xleftarrow{\$} \{0, 1, \dots, i\}$ with XOF
  - $c_i := c_j$
  - $s \xleftarrow{\$} \{0, 1\}$ with XOF

- $c_b := 1 - 2s$

– return c

Note that c is a random N-vector with τ ± 1 's and $N - \tau$ 0's using the input seed ρ to generate the randomness needed to compute b and s with an XOF.

1.3 Signature and its security model

A Randomized (deterministic) signature scheme consists of a triplet of polynomial-time algorithms (Genkey, Sig, Ver).

1. **Key Generation (Genkey)**: with input a security parameter K the key generation algorithm outputs a keypair (PK, SK) where PK, SK are related to each other throughout a hard mathematical problem (HMP).
2. **Signature algorithm (Sig)**:
   – Sig takes the security parameter K as input and produces a random r (skip in case of deterministic signature);
   – With input (SK, m, r) the signing algorithm Sig produces a signature σ.
3. **Verification (Ver)**: With input (m, σ, PK) the verification algorithm returns 1 if the signature is valid and 0 otherwise.

Security : When designing a signature scheme, we need to have in mind the following 4 fundamentals properties:

– (1) the signer should be able to make the verifier accept the proof if he really knows the secret key corresponding to the public key.
– (2) if the protocol succeeds (Ver outputs 1), then the verifier is convinced that the signer knows the secret key corresponding to the public key.
– (3) the verifier does not learn any information about the secret itself even if he sees many signatures (Zero-knowledge property).
– (4) nobody can forge a signature (which means that nobody is able to produce a valid signature without knowing the secret key)

Goldwasser, Micali and Rivest (in 1988) in [33], introduce the basic security notion for signatures called "existential unforgeability with respect to adaptive chosen- message attacks".

sEUF-CMA: Strong Unforgeability against Adaptive Chosen Message Attacks

For this, a reduction algorithm $\mathcal{R}$ and an attacker $\mathcal{A}$, simulate a the following game.

1. **Key generation**: $\mathcal{R}$ runs the algorithm Genkey with a security parameter K as input, to obtain the public key PK and the secret key SK, and gives PK to the attacker $\mathcal{A}$.

2. **The Queries of the adversary**: $\mathcal{A}$ may request a signature on any message $m \in \mathcal{M}$ (multiple adaptive requests of the message are allowed) and $\mathcal{R}$ will respond with (m, σ), without using the secret key but where $Ver(PK, m, \sigma) = 1$. The signatures already outputted by the oracle signature to the queries of the $\mathcal{A}$ are stored in a list $List(\mathcal{S})$.
3. **Strong forgery**: Eventually, $\mathcal{A}$ will output a pair (m, σ) and is said to win the game if $Ver(PK, m, \sigma) = 1$ and if $(m, \sigma) \notin List(\mathcal{S})$ (this last condition force the attacker $\mathcal{A}$ to output his own forgery (note that in this case of strong unforgeable it is allowed to the adversary to output $(m'', \sigma'') \notin List(\mathcal{S})$ assuming that $List(\mathcal{S})$ contains already signatures of the form (m'', σ''') with $\sigma''' \neq \sigma''$.

The probability that $\mathcal{A}$ wins in the above game is denoted $Adv\mathcal{A}$.

A signature scheme (Genkey; Sig;Ver) is strongly existentially unforgeable with respect to adaptive chosen message attacks if for all probabilistic polynomial time attacker $\mathcal{A}$, $Adv\mathcal{A}$ is negligible in the security parameter K.

1.4 Hard problems over lattices

Definition 1 *(LWE). The learning with errors problem*
Consider the following equations $b_i = \boldsymbol{a}_i \boldsymbol{s}^t + e_i \mod q$ *for* $1 \leq i \leq k$ *where the* $\boldsymbol{a}_i, \boldsymbol{s} \in \mathbb{Z}_q^n$ *are chosen uniformly at random and the* e_i *(called the errors) are drawn from error distribution* χ.

- *Computational LWE: Given samples* $(\boldsymbol{a}_i, b_i)_i$ *compute* $\boldsymbol{s}$
- *Decisional LWE: Given samples* $(\boldsymbol{a}_i, b_i)_i$, *distinguish them from random samples in* $\mathbb{Z}_q^n \times \mathbb{Z}_q$

Definition 2 *(l_∞-SIS). The short integer solution (Homogenus/Inhomogenus) problem*
Consider the following equation $\boldsymbol{t} = \boldsymbol{sB} \mod q$ *where* $\boldsymbol{B} \in \mathbb{Z}_q^{n \times m},\ m \geq n+1$ *is chosen uniformly at random and* $\boldsymbol{s} \in \mathbb{Z}_q^n$ *(called short vector) verify the upper bound* $|\boldsymbol{s}|_\infty \leq \beta \leq q-1$ *for some* $\beta \in \mathbb{R}$.

Computational l_∞*-*$SIS_{q,n,m,\beta}$*: Given* $(\boldsymbol{t}, \boldsymbol{B})$, *compute an appropriate* $\boldsymbol{s}$.

2 Description of EagleSign

In this section, we give the description of the two variants of our signature.

2.1 EagleSign 1 (General case)

The general case of our signature can be summarized at high level as follows.

1. **Ring :** $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^n+1)}$, $S_\eta = \{u \in R_q / |u|_\infty \leq \eta\}$ the polynomials in $\mathcal{R}_q$ whose $l\infty$ norm is tightly upper-bounded by η

2. Public and private keys:
   Keygen : it takes the security level and a system of parameters as inputs
   - $\mathbf{A} \in R_q^{k\times l}$ is a public matrix generated uniformly at random
   - $\mathbf{F}, \mathbf{G} \in S_{\eta_g}^{l\times l}$ are secret invertible matrices of small polynomials (generated uniformly at random).
   - $\mathbf{D} \in S_{\eta_d}^{k\times l}$ is a secret matrix of small polynomials (generated uniformly at random).
   - $\mathbf{E} := (\mathbf{A}\mathbf{F}^{-1} + \mathbf{D})\mathbf{G}^{-1} \in R_q^{k\times l}$
   - $pk := (\mathbf{A}, \mathbf{E})$ is the (longterm) public key.
   - $sk := (\mathbf{F}, \mathbf{G}, \mathbf{D})$ is the (longterm) private key.
   - Output (pk, sk)
3. Signature
   Sig(M, $sk = (\mathbf{F}, \mathbf{G}, \mathbf{D})$)
   - $(\mathbf{Y}_1, \mathbf{Y}_2) \in S_{\eta_{y_1}}^{l\times m} \times S_{\eta_{y_2}}^{k\times m})$ is the ephemeral secret key;
   - $\mathbf{P} := \mathbf{A}\mathbf{F}^{-1}\mathbf{Y}_1 + \mathbf{Y}_2 \in R_q^{k\times m}$ is ephemeral public key;
   - $r := G(\mathbf{P})$;
   - $\mathbf{C} \in S_{\eta_c}^{l\times m} := H(M, r)$;
   - $\mathbf{Z} := \mathbf{G}\mathbf{U} \mod q$, $\mathbf{U} := \mathbf{Y}_1 + \mathbf{F}\mathbf{C} \mod q \in R_q^{l\times m}$;
   - $\mathbf{W} := \mathbf{Y}_2 - \mathbf{D}\mathbf{U} \mod q := (\mathbf{Y}_2 - \mathbf{D}\mathbf{Y}_1) - \mathbf{D}\mathbf{F}\mathbf{C} \mod q \in R_q^{k\times m}$;
   - Output the signature $\sigma = (r, \mathbf{Z}, \mathbf{W})$
4. Verification
   Ver($\sigma = (r, \mathbf{Z}, \mathbf{W})$, $pk = (\mathbf{A}, \mathbf{E})$)
   - $\mathbf{C} \in S_{\eta_c}^{l\times m} =: H(M, r)$
   - $\mathbf{V} = \mathbf{E}\mathbf{Z} - \mathbf{A}\mathbf{C} + \mathbf{W} \mod q$
   - Reject if some appropriate upper-bounds of the norms of $\mathbf{Z}, \mathbf{W}$ are not verified
   - Reject if $\mathbf{C} \neq H(M, G(\mathbf{V}))$
   - Otherwise accept

2.2 EagleSign 2 (particular case that is implemented)

In this subsection, we propose the three following detailed algorithms for our signature in case $\mathbf{F} = 1, m = 1, k, l \in \{1, 2, \ldots\}$. We use the following function and notations:

1. The function GenMatrixUnifSmallPolyn, with input a (seed, k,l), generates uniformly at random an element in the set $S_\eta^{k\times l}$ for $k, l = 1, 2, \ldots$;
2. The transformation GenMatrixUnifPolyn maps a uniform seed $\rho \in \{0,1\}^{256}$ to a matrix $A \in \mathcal{R}_q^{k\times l}$ (for $k, l = 1, 2, \ldots$) in NTT domain representation;

3. The function GenVectorUnifSmallPolyn, with input a seed, generates uniformly at random an element in the set S_η^l for $l = 1, 2, \ldots$;
4. The function GenVectorUnifSparsePolyn, with input a seed, generates uniformly at random an element $\mathbf{y_1}$ in the set (of ternary sparse polynomials with hamming weight t) B_t^l for $l = 1, 2, \ldots$.
5. The function CRH (resp. CRH1) is a collision resistant hash used in our signature scheme and mapping to $\{0,1\}^{384}$ (resp. $\{0,1\}^{256}$).
6. The function G is a multi-collision resistant hash used in our signature scheme and mapping to $\{0,1\}^{256}$.
7. $H : \{0,1\}^\star \to B_\tau^l$ is a cryptographic hash function used to generate $\mathbf{c} \in B_\tau^l$.
8. The function GenRandoms is interpreted as SHAKE-256 in our implementation.
9. We consider the following bounds to make sure that each output of the signature is short enough :
   $\delta = l \times \eta_G \times (t+\tau)$, $\delta' = \eta_{y_2} + l \times \eta_D \times (t+\tau)$ and $\beta = \max(2\delta', 2\delta) \leq (q-1)/16$.

Note that the description of these previous functions is given is the section 5.5.

Algorithm 1 : EagleSign Key generation algorithm

Require: the security parameter 1^n
1: $\beta \leftarrow \{0,1\}^{256}$;
2: $(\beta_1, \beta_2, \rho, \mathrm{key}) := \mathrm{GenRandoms}(\beta)$ ▷ $(\beta_1, \beta_2, \rho, \mathrm{key}) \in (\{0,1\}^n)^{3+1}$
3: $\beta_1 = \mathrm{Hash}(\beta_1)$, ▷ we use SHAKE-256 for Hash to renew β_1
4: $\mathbf{G} := \mathrm{GenMatrixUnifSmallPolyn}(\beta_1, l, l)$ ▷ $\mathbf{G} \in S_{\eta_G}^{l \times l}$
5: **if** $\mathbf{G}$ is not invertible in $\mathcal{R}_q$ **then**
6: Go to step (3);
7: **end if**
8: $\mathbf{D} := \mathrm{GenMatrixUnifSmallPolyn}(\beta_2, k, l)$; ▷ $\mathbf{D} \in S_{\eta_D}^{k \times l}$,
9: $\mathbf{A} := \mathrm{GenMatrixUnifPolyn}(\rho)$; ▷ $\mathbf{A} \in \mathcal{R}_q^{k \times l}$
10: $\mathbf{E} := (\mathbf{A} + \mathbf{D})\mathbf{G}^{-1} \mod q$;
11: $\mathrm{tr} := \mathrm{CRH1}(\rho, \mathbf{E})$; ▷ $\mathrm{tr} \in \{0,1\}^{256}$
12: $\mathrm{sk} := (\rho, \mathrm{tr}, \mathbf{G}, \mathbf{D}, \mathrm{key})$; ▷ the longterm private key
13: $\mathrm{pk} := (\rho, \mathbf{E})$; ▷ the longterm public key
14: **return** $(\mathrm{pk}, \mathrm{sk})$

Remark: The parameter $'\mathrm{key}'$ is only used in case of deterministic signature.

Algorithm 2 : EagleSign Signature algorithm

Require: a message M, a secret key $\mathrm{sk} = (\rho, \mathrm{tr}, \mathbf{G}, \mathbf{D}, \mathrm{key})$

1: $\mu \in \{0,1\}^{384} := \mathrm{CRH}(\mathrm{tr}, M)$;
2: $\lambda \leftarrow \{0,1\}^{384}$; $\triangleright$ See Remark bellow
3: $\mathbf{y}_1 \leftarrow S^l_{\eta_{y_1}} := \mathrm{GenVectorSparsePoly}(\lambda, 0)$; $\triangleright$ See Remark bellow
4: $\mathbf{y}_2 \leftarrow S^k_{\eta_{y_2}} := \mathrm{GenVectorUnifSmallPoly}(\lambda, l)$;
5: $\mathbf{A} \leftarrow \mathcal{R}_q^{k\times l} := \mathrm{GenMatrixUnifPolyn}(\rho)$;
6: $\mathbf{p} := \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2 \mod q \in R_q^k$;
7: $r := G(\mathbf{p})$;
8: $\mathbf{c} \in B_\tau^l := H(\mu, r)$; $\triangleright$ H is instantiated as SHAKE
9: $\mathbf{u} := \mathbf{y}_1 + \mathbf{c} \mod q$; $\triangleright$ Note that $\mathbf{y}_1$ and $\mathbf{c}$ are generated in the same way
10: $\mathbf{z} := \mathbf{G}\mathbf{u} \mod q$; $\triangleright$ $\mathbf{z} = \mathbf{G}(\mathbf{y}_1 + \mathbf{c}) \mod q \in \mathcal{R}_q$
11: $\mathbf{w} := \mathbf{y}_2 - \mathbf{D}\mathbf{u} \mod q$; $\triangleright$ $\mathbf{w} = \mathbf{y}_2 - \mathbf{D}(\mathbf{y}_1 + \mathbf{c}) \mod q \in \mathcal{R}_q^k$
$\triangleright$ Note that $|\mathbf{z}|_\infty \leq \delta$ and $|\mathbf{w}|_\infty \leq \delta'$
12: $\triangleright$ Validity of the signature (**optional**) to defeat fault signature attacks for steps 13, 14, 15 and 16
13: $\mathbf{v} := \mathbf{E}\mathbf{z} - \mathbf{A}\mathbf{c} + \mathbf{w} \mod q$;
14: **if** $\mathbf{p} \neq \mathbf{v}$ **then**
15: Aborts
16: **end if**
17: **return** $\sigma := (r, \mathbf{z}, \mathbf{w})$ as signature

Remark:
– In case of probabilistic signature λ is a random and in case of deterministic signature $\lambda = (\mu,' \mathrm{key}')$.
– When $\eta_{y_1} = 1$, we choose $\mathbf{y}_1 \in B_t^l$ where B_t is defined in preliminary section.

Algorithm 3 : EagleSign Verification algorithm

Require: signature $\sigma = (r, \mathbf{z}, \mathbf{w})$, public key $(\rho, \mathbf{E})$ and bounds δ, δ'

1: $\mathrm{tr} \in \{0,1\}^{256} := \mathrm{CRH1}(\rho, \mathbf{E})$;
2: $\mu \in \{0,1\}^{384} := \mathrm{CRH}(\mathrm{tr}, M)$;
3: $\mathbf{c} \in B_\tau^l := H(\mu, r)$,
4: $\mathbf{A} \leftarrow \mathcal{R}_q^{k\times l} := \mathrm{GenMatrixUnifPoly}(\rho)$;
5: $\mathbf{v} := \mathbf{E}\mathbf{z} - \mathbf{A}\mathbf{c} + \mathbf{w} \mod q$;
6: $r' = G(\mathbf{v})$;
7: **if** $||\mathbf{z}||_\infty > \delta$ or $||\mathbf{w}||_\infty > \delta'$ or $\mathbf{c} \neq H(\mu, r'))$ **then**
8: **return** 0
9: **else**
10: **return** 1
11: **end if**

Correctness of the signature: Easy to verify.

3 Security analysis

3.1 Security proof in the Random Oracle Model (ROM)

In this subsection, we adapt to lattices, the tools, techniques and frameworks for security proof developed by Pointcheval *et al.* [60] for Elgamal-like signatures (DSA, KCDSA, Schnorr,...) where the underlying hard problem was the discrete logarithm problem. To design a security proof in ROM for Eagle Sign, we use the following steps.

1. Protection against secret key recovery: we need to prove that recovering the private key from the public key is equivalent to solving hard instance in a specified lattice problem namely the MLWE problem in our case.
2. Simulation of the random oracle H: the cryptographic hash function H of the signature is considered to be an ideal random function that the attacker can query as an oracle. For each new query of the attacker, the simulator chooses uniformly at random a value in the output set of the real hash function and sends it as response. This answer needs to be independent from previous query/response pairs stored in a data base L_H by the simulator. If a query is replayed by the attacker, the simulator finds the correct answer in L_H.
3. Simulation of the signature: without the private key and by controlling the ideal hash function H, the simulator design a signature algorithm able to produce valid signatures in polynomial time with a hight probability.

   In our simulation, as proved by Pointcheval *et al* [60] for classical DSA, it is not necessary to consider the second hash function G as a random oracle thus the use of random oracles is minimizing. G will be just considered as a multi-collision-resistance function: G is said j-collision-resistant, if it is hard to find $(u_1, \ldots, u_j)$ pairwise distinct elements such that $G(u_1) = \ldots = G(u_j)$.

4. Signature forgery: Using an adaptively chosen-message attack against the legitimated signer, the attacker produces a valid signature forgery with Q_H queries to the ideal hash function H and Q_S queries to the oracle signature. For each new query to H, L_H is updated with the corresponding query/response pair. To be a real attack, it is assumed the valid signature of the attacker has not been sent as a call to the signature oracle.
5. Solving a MSIS problem using signature forgery (with he following steps):
   - Since the attacker don't control the ideal random function H, from a signature forgery of the attacker, the "forking lemma" is used to show that, she can construct two signatures with the same fixed values $(M/\mu, r)$ but H produce different responses $\mathbf{c}$ and $\mathbf{c}'$ (which really means that different ideal random functions are used; this scenario is possible since the attacker don't control H in ROM).
   - The previous scenario produces collusions throughout G from the positive answer of the verification process.
   - Two valid forged signatures (with collusion) are used to show how to compute a short non-zero vector as a solution of a MSIS problem with l_∞ norm.

Theorem 1. *Assume that an attacker $\mathcal{A}$ produce an existential forgery of the Eagle Sign after Q_H calls to H and Q_S calls to the simulator for signature, under an adaptively chosen message attack with probability ϵ, then by forking lemma, the MSIS-l_∞ problem $(\mathbf{E}|\mathbf{A}|\mathbf{I}_k)\mathbf{x}^T = 0$ can be solved in polynomial time for $||\mathbf{x}||_\infty \leq \beta \leq \frac{q-1}{16}$ where $\mathbf{E}$ is the public key of EagleSign. Note that the probabilities are taken over random tapes, random oracles, messages and public/private keys (long-term and ephemeral).*

Proof. **A) Protection against secret key forgery**:
By the construction of the longterm public key $\mathbf{E} = (\mathbf{A}+\mathbf{D})\mathbf{G}^{-1} \mod q \Leftrightarrow \mathbf{A} = \mathbf{EG} - \mathbf{D}$ and the ephemeral public key $\mathbf{p} = \mathbf{Ay}_1 + \mathbf{y}_2 \mod q$, it is clear that the secret key forgery is equivalent to MLWE.

B) Simulation of the signature:
We need to simulate the signature without the private key with the ideal hash function under control.

- input a message M;
- generate randomly $\mathbf{z}, \mathbf{w}$ such that $||\mathbf{z}||_{l_\infty} \leq \delta$ and $||\mathbf{w}||_{l_\infty} \leq \delta'$;
- $\mathrm{tr} = \mathrm{CRH1}(\rho, \mathbf{E})$;
- $\mu = \mathrm{CRH}(\mathrm{tr}, M)$;
- generate randomly $\mathbf{c} \in B_\tau^l$;
- with the above choice compute $\mathbf{v} = \mathbf{Ez} - \mathbf{Ac} + \mathbf{w} \mod q$:
- compute $r = G(\mathbf{v})$;
- define $\mathbf{c} = H(\mu, r) \in B_\tau^l$ and update the data base L_H of the oracle hash function with the query/response $(M/\mu, r)/\mathbf{c}$;
- Output the signature $(M, r, \mathbf{z}, \mathbf{w})$.

The simulation of the signature is indistinguishable.

D) Forking for solving the MSIS problem:
If the attacker $\mathcal{A}$ output a valid signature $((M/\mu, r, \mathbf{c}), \mathbf{z}, \mathbf{w})$ where c can be found in L_H with the prefix $(M/\mu, r)$ with probability ϵ for a new message M with less than Q_H calls to the hash function H, then by forking technique we obtain two valid signatures of the same message M and fixed values namely $(M/\mu, r, \mathbf{c}), \mathbf{z}, \mathbf{w}$ and $(M/\mu, r', \mathbf{c}'), \mathbf{z}', \mathbf{w}'$ with $r = r'$ and $\mathbf{c} \neq \mathbf{c}'$. From $r = r'$, we deduce $v = v'$ with a high probability, thus $\mathbf{Ez} - \mathbf{Ac} + \mathbf{w} \mod q = \mathbf{Ez}' - \mathbf{Ac}' + \mathbf{w}' \mod q$. Hence, we have $(\mathbf{E}|\mathbf{A}|I_k))(\mathbf{z} - \mathbf{z}', \mathbf{c}' - \mathbf{c}, \mathbf{w} - \mathbf{w}')^T = 0$. Now, put $\mathbf{x} = (\mathbf{z} - \mathbf{z}', \mathbf{c}' - \mathbf{c}, \mathbf{w} - \mathbf{w}')$, since $\mathbf{c}' - \mathbf{c} \neq 0$ and $||\mathbf{x}||_\infty \leq \beta$ then we see that $\mathbf{x}$ is a nonzero short solution of the MSIS problem.

NB: Since the ephemeral public key $\mathbf{p} = \mathbf{Ay}_1 + \mathbf{y}_2$ is integrally recovered during the verification process, then we don't need to use the SelfTargetMSIS problem of Dilithium in our security proof.

3.2 Security proof in the Quantum Random Oracle Model (QROM)

Our signature is secure in ROM and is a signature scheme without aborts, and for the shake of completeness, a complete proof in QROM will be designed later. Note that the authors of Dilithium say the following: ”In our opinion, evidence is certainly mounting that the distinction between signatures secure in the ROM and QROM will soon become treated in the same way as the distinction between schemes secure in the standard model and ROM – there will be some theoretical differences, but security in practice will be the same”.

3.3 Selection of the parameters according different security levels

For a complete study of the estimation of the security level of LWE and NTRU -like schemes proposed at NIST, one can see the recent work of Albrecht, Curtis, Deo, Davidson, Player, Postlethwaite, Virdia, Wunderer in [2] : Estimate all the LWE and NTRU schemes (PQC-Forum January 2018). In their paper [2], the authors point out the sources of divergence (instantiation of the SVP oracle in BKZ by sieving method or enumeration method, treatment of polynomial factor) in estimated security level of the ideal lattice-based schemes proposed to NIST. Many techniques for improving lattice-based cryptanalysis where proposed recently [1, 6, 8, 24, 26, 39, 40, 49, 56, 65–67, 73, 75]. Moreover, vulnerabilities in ideal lattice-based schemes where pointed out by many authors [8, 14, 21, 26]. Based on these results, some authors claim that the security of lattice-based cryptography over the rings is not well understood (see Bernstein *et al.* in NTRU LPRime [52]). Nevertheless, currently, as far as we know, these algebraic structure does not figure into the cost of the best known attacks on NTRU-RLWE-like schemes and in general, no algorithm is known that can exploit enough the ring structure and that is thus working more efficiently on ideal-lattices than classical lattices [1, 7, 14, 52]. Therefore, we can analyse the hardness of our signature over standard lattices.
For recent advances and background for solving uSVP and similar problems, we refer to [1–3, 5, 6, 10]. Recall that BKZ lattice reduction algorithm (which is a blockwise variant of the LLL algorithm) proceeds by sublattice reduction using a SVP oracle in a smaller dimension b. With BKZ, the best known classical algorithm (respectively: quantum sieving algorithm) [1, 10, 24, 39] for the primal/dual attack [1, 6, 16] with block size b of MLWE or MNTRU-like schemes, have costs of $2^{0.292b}$ (respectively: $2^{0.265b}$ with Grover speedups [34]). Therefore, currently (June in 2023, as far as we know), we must at least use $2^{0.265b}$ (or the "paranoid" lower bound $2^{0.2075b}$ given in [1, 2]) to compute the security level.
To estimate the security level, we use the lattice estimator of Albrecht *et al.* [2–4] (lattice-estimator-main with Sagemath and python) to estimate the security of the longterm public key and the ephemeral public key. We use the tool of Crystal Dilithium to estimate the security of MSIS for unforgeability.
The following algorithms 3 are covered by the estimator that we have used in EagleSign security: meet-in-the-middle exhaustive search, coded-BKW, dual-lattice

Table 3. Parameters Selection for NIST Security Levels

EagleSign				
NIST security level	2	3	3+	5
$F = 1, m = 1$ and $k, l \in \{1, 2\}$				
	Medium	Recomm I	Recomm II	High I
(k, l)	$(2, 1)$	$(1, 1)$	$(2, 2)$	$(1, 2)$
$q = 12289, n =$	512	1024	512	1024
$\eta_c = 1, \mathbf{c} \in B_\tau^l, \tau =$	18	38	18	18
Strong Unforgeable signature:				
$\beta = \max(2\delta', 2\delta'), (\delta, \delta'),$	$(948, 1012)$	$(178, 242)$	$(432, 248)$	$(208, 240)$
BKZ block-size b to break SIS	607	867	748	869
Best Known Classical bit-cost	177	253	218	253
Best Known Quantum bit-cost	160	229	198	230
Best Plausible bit-cost	125	179	155	180
Ephemeral secret recovery $(\mathbf{Y}_1, \mathbf{Y}_2)$				
$\eta_{y_1} = 1, \mathbf{Y}_1 \in B_t^l, (t, \eta_{y_2}) =$	(140,64)	(140,64)	(90,32)	(86,32)
BKZ block-size b to break LWE	439	713	764	870
Best Known Classical bit-cost	128	208	223	254
Best Known Quantum bit-cost	116	188	202	230
Best Plausible bit-cost	91	147	158	180
Longterm secret recovery $(\mathbf{G}, \mathbf{D})$				
(η_G, η_D)	$(6, 6)$	$(1, 1)$	$(2, 1)$	$(1, 1)$
BKZ block-size b to break LWE	444	696	741	1649
Best Known Classical bit-cost	129	203	216	481
Best Known Quantum bit-cost	117	184	196	436
Best Plausible bit-cost	92	144	153	342

attack and small/sparse secret variant, lattice-reduction and enumeration, primal attack via uSVP [16], Arora-Ge algorithm [14] using Gröbner bases.

We provide the following examples for security level 3 (with $p = N = 1024, q = 12289$) on how to find the values in the previous table.

Python Code for security level relatively to various attacks with lattice-estimator

Nist security level 3: Longterm secret key recovery (G, D) from $E = (A + D)G^{-1}$ based on a mix of MNTRU and MLEW with uniform secret G and error D

```
from estimator import *
from estimator.lwe_parameters import *
from estimator.nd import *
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
print("EagleSign Security estimate")
print("Nist security level 3 :")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
print("Ring dimension p=1024, underlying field modulus q=12289
    ")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
print("Longterm secret key recovery (G,D) from E=(A+D)G^{-1}")
print("To estimate the security level, E=(A+D)G^{-1} is viewed
     (as usual) as a LWE instance where G is the secret and D
    is the error")
print("Uniform Distribution  for G and D")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
p=1024
q=12289
k=1
l=1
etag=1
etad=1
EagleSign3LPk = LWEParameters(n=p*l,
q=q,
Xs=NoiseDistribution.Uniform(-etag,etag),
Xe=NoiseDistribution.Uniform(-etad,etad),
m=k*p,
tag="EagleSign3LPk")
print("p:",p, ", q:", q,", k:", k, ", l:", l,   ", etag:",
    etag, ", etad:", etad)
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")

r=LWE.estimate(EagleSign3LPk)
```

The previous code produces the following output

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
EagleSign Security estimate
Nist security level 3 :
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Ring dimension p=1024, underlying field modulus q=12289
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Longterm secret key recovery (G,D) from E=(A+D)G^{-1}
To estimate the security level, E=(A+D)G^{-1} is viewed
(as usual) as a LWE instance where G is the secret and D
 is the error Uniform Distribution  for G and D
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
p: 1024 , q: 12289 , k: 1 , l: 1 , etag: 1 , etad: 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bkw  :: rop: ≈2^274.8, m: ≈2^261.5, mem: ≈2^262.5, b: 19,
        t1: 0, t2: 21, ℓ: 18, #cod: 900, #top: 0, #test: 128,
        tag: coded-bkw

usvp :: rop: ≈2^230.1, red: ≈2^230.1, δ: 1.002634, β: 712,
        d: 1896, tag: usvp

bdd  :: rop: ≈2^226.5, red: ≈2^225.6, svp: ≈2^225.5, β: 696,
        η: 732, d: 1874, tag: bdd

bdd_hybrid :: rop: ≈2^226.6, red: ≈2^225.6, svp: ≈2^225.5,
        β: 696, η: 732, ζ: 0, |S|: 1, d: 1900, prob: 1, ↻: 1,
        tag: hybrid

bdd_mitm_hybrid :: rop: ≈2^340.8, red: ≈2^340.0, svp: ≈2^339.5,
        β: 711, η: 2, ζ: 262, |S|: ≈2^415.3, d: 1653,
        prob: ≈2^-108.3, ↻: ≈2^110.5, tag: hybrid

dual :: rop: ≈2^239.5, mem: ≈2^151.0, m: 920, β: 742, d: 1944,
        ↻: 1, tag: dual

dual_hybrid :: rop: ≈2^228.1, mem: ≈2^223.7, m: 883, β: 701,
        d: 1856, ↻: 1, ζ: 51, tag: dual_hybrid
```

Nist security level 3: Ephemeral secret key recovery (y_1, y_2) from $P = A.y_1 + y_2$ based on MLEW with a sparse secret y_1 and uniform error y_2

```
from estimator import *
from estimator.lwe_parameters import *
from estimator.nd import *
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
print("EagleSign Security estimate")
print("Nist security level 3")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
print("Ring dimension p=1024, underlying field modulus q=12289
    ")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
```

```
print("Ephemeral secret key recovery (y_1,y_2) from P=A.y_1+
    y_2")
print("To estimate the security level, P=A.y_1+y_2 is viewed
    as a LWE instance where y1 is the secret and y2 is the
    error ")
print("Uniform Distribution  for y_2 and sparse distribution
    for y_1")
print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
p=1024
q=12289
k=1
l=1
t=140
etay2=64
etay1=1
EagleSign3EPk = LWEParameters(n=p*l,
q=q,
Xs=NoiseDistribution.SparseTernary(p,t/2,t/2),
Xe=NoiseDistribution.Uniform(-etay2,etay2),
m=k*p,
tag="EagleSign3EPk")
print("p:",p, ", q:", q,", k:", k, ", l:", l,  ", t:", t,  ",
    etay1:", etay1, ", etay2:", etay2)

r=LWE.estimate(EagleSign3EPk)
```

The previous code produces the following output

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 EagleSign Security estimate
 Nist security level 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Ring dimension p=1024, underlying field modulus q=12289
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Ephemeral secret key recovery (y_1,y_2) from P=A.y_1+y_2
To estimate the security level, P=A.y_1+y_2 is viewed  as
 a LWE instance where y1 is the secret and y2 is the error
Uniform Distribution  for y_2 and sparse distribution for
y_1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
p: 1024 , q: 12289 , k: 1 , l: 1 , t: 140 , etay1: 1 ,
etay2: 64
Algorithm functools.partial(<function dual_hybrid at
0x7f98644e4700>, red_cost_model=<estimator.reduction.
MATZOV object at 0x7f98645f0730>, mitm_optimization=True)
on LWEParameters(n=1024, q=12289, Xs=D(σ=0.37), Xe=D(σ=37.24),
m=1024, tag='EagleSign3EPk') failed with β = 79 > d = 65

bkw  :: rop: ≈2^245.5, m: ≈2^233.1, mem: ≈2^234.1, b: 17,
```

```
         t1: 0, t2: 9, ℓ: 16, #cod: 757, #top: 0, #test: 268,
         tag: coded−bkw

usvp  :: rop: ≈2^301.7, red: ≈2^301.7, δ: 1.002088, β: 972,
         d: 1503, tag: usvp

bdd   :: rop: ≈2^315.6, red: ≈2^315.5, svp: ≈2^310.3, β: 1026,
         η: 1035, d: 1241, tag: bdd

bdd_hybrid  :: rop: ≈2^313.9, red: ≈2^311.9, svp: ≈2^313.5,
         β: 739, η: 780, ζ: 287, |S|: 1, d: 1655, prob: ≈2^−72.4,
         ↻: ≈2^74.6, tag: hybrid

bdd_mitm_hybrid  :: rop: ≈2^427.2, red: ≈2^427.2, svp: ≈2^420.2,
         β: 1007, η: 2, ζ: 512, |S|: ≈2^566.6, d: 1537,
         prob: ≈2^−113.5, ↻: ≈2^115.7, tag: hybrid

dual  :: rop: ≈2^320.9, mem: ≈2^208.0, m: 551, β: 1037, d: 1575,
         ↻: 1, tag: dual

dual_hybrid  :: rop: ≈2^261.7, mem: ≈2^228.4, m: 459, β: 713,
         d: 1095, ↻: ≈2^31.0, ζ: 388, h1: 23, tag: dual_hybrid
```

NIST security level 3: Unforgeability Security Analysis based on MSIS problem upper bounded by β with l_∞ norm

```python
from MSIS_security import MSIS_summarize_attacks, MSISParameterSet
class UniformEagleSignParameterSet(object):
  def __init__(self, n, k, l, etay2, etay1, t, etag, etad, tau, q):
    self.n = n
    self.k = k
    self.l = l
    self.etay1 = etay1
    self.etay2 = etay2
    self.etag = etag
    self.etad = etad
    self.tau = tau
    self.t = t
    self.q = q

    delta = l*(t+tau)*etag
    deltaprime = l*(t+tau)*etad + etay2
    self.beta = max(2*deltaprime, 2*delta)


def EagleSign_to_MSIS(dps):
```

```
    return MSISParameterSet(dps.n, dps.k + dps.l + dps.l, dps.k,
        dps.beta, dps.q, norm="linf")

if __name__ == "__main__":
    sheme = "Uniform EagleSign Recommended I"
    param = UniformEagleSignParameterSet(
    1024, 1, 1, 64, 1, 140, 1, 1, 38, 12289)
    print("\n"+scheme)
    print(param.__dict__)
    print("")
    print("=== STRONG UF")
    v = MSIS_summarize_attacks(EagleSign_to_MSIS(param))
```

Here is the output of the previous code:

```
Uniform EagleSign Recommended I
{'n': 1024, 'k': 1, 'l': 1, 'etay1': 1, 'etay2': 64, 'etag':
    1, 'etad': 1, 'tau': 38, 't': 140, 'q': 12289, 'beta':
    484}

=== STRONG UF
Attack uses block-size 867 and 3072 dimensions, with 0 q-
    vectors
log2(epsilon) = -179.58, log2 nvector per run 179.92
shortest vector used has length l=11617.36, q=12289, `l<q'= 1
SIS & 3072 & 867 & 253 & 229 & 179
```

3.4 Constant time implementation

As Dilithium we do not use branch depending on secret data and also we do not use access memory locations that depend on secret data. Moreover, for modular reductions $\mod q$, we do not use the '%' operator of the C programming language, instead we use Montgomery reductions. We do not use also rejection sampling in the signature and verification algorithm.

4 Performance: sizes and cycles

In the table 4, we give the sizes and the cycles for the 3 algorithms of our signature.

The table 5 presents, in the case of reference implementation, a comparison between code efficiency of EagleSign and Dilithium level 3 and 5 based on our specific processor characteristics.

With this comparison, for security level 3, we see that our signature algorithm is twice faster than those of Dilithium. The verification algorithm of EagleSign and Dilithium have similar performance but the key generation algorithm of Dilithium is 1.5 times faster than those of EagleSign.

Sizes of the Public key and the Signature

Table 4. Performances of Implementation of EagleSign for NIST Security Levels 3 and 5

EagleSign				
NIST security level	2	3	3+	5
$F = 1, m = 1$ and $k, l \in \{1, 2\}$				
	Medium	Recomm I	Recomm II	High I
(k, l)	$(2, 1)$	$(1, 1)$	$(2, 2)$	$(1, 2)$
$q = 12289, N =$	512	1024	512	1024
τ	18	38	18	18
$\beta = \max(2\delta', 2\delta'), (\delta, \delta'),$	$(948, 1012)$	$(178, 242)$	$(432, 248)$	$(208, 240)$
$\eta_{y_1} = 1, (t, \eta_{y_2})$	(140,64)	(140,64)	(90,32)	(86,32)
(η_G, η_D)	$(6, 6)$	$(1, 1)$	$(2, 1)$	$(1, 1)$
Size in bytes:				
signature size $(r, \mathbf{Z}, \mathbf{W})$	2144	2336	2464	3488
public key size $(\rho, \mathbf{E})$	1824	1824	3616	3616
EagleSign Performance (12th Gen Intel Core i7-1260P × 16, RAM 16GB)				
Reference Implementation				
Gen median cycles		1001330		3345416
Gen average cycles		1020723		3443617
Sign median cycles		1274146		2351304
Sign average cycles		1283454		2358603
Verif median cycles		946459		1594736
Verif average cycles		955956		1602340
Optimized Implementation				
Gen median cycles		978368		3212708
Gen average cycles		1000579		3287036
Sign median cycles		1286413		2251036
Sign average cycles		1241245		2259111
Verif median cycles		917213		1503004
Verif average cycles		927108		1512331

Table 5. Comparison of Implementation Performances of EagleSign and Dilithium for NIST Security Levels 3 and 5 (Processor: 12th Gen Intel Core i7-1260P $\times$ 16, RAM 16GB)

NIST security level	3	5
EagleSign Performance (12th Gen Intel Core i7-1260P $\times$ 16, RAM 16GB)		
Gen median cycles	1001330	3345416
Gen average cycles	1020723	3443617
Sign median cycles	1274146	2351304
Sign average cycles	1283454	2358603
Verif median cycles	946459	1594736
Verif average cycles	955956	1602340
Dilithium Performance (12th Gen Intel Core i7-1260P $\times$ 16, RAM 16GB)		
Gen median cycles	617942	933226
Gen average cycles	618503	934927
Sign median cycles	2061071	2622695
Sign average cycles	2519785	3142267
Verif median cycles	617642	1023610
Verif average cycles	621859	1030807

The signature, the public key of EagleSign are respectively $\sigma = (r, \mathbf{z}, \mathbf{w})$, $pk = (\rho, \mathbf{E})$ where $\mathbf{z} := \mathbf{G}\mathbf{u} \mod q$, $\mathbf{w} := \mathbf{y}_2 - \mathbf{D}\mathbf{u} \mod q$, $\mathbf{u} := \mathbf{y}_1 + \mathbf{c} \mod q$, $\mathbf{y}_1 \in B_t^l$, $\mathbf{c} \in B_\tau^l, \mathbf{y}_2 \in S_{\eta_{y_2}}^k, \mathbf{D} \in S_{\eta_D}^{k\times l}, \mathbf{D} \in S_{\eta_G}^{l\times l}$ and $\mathbf{E} = (\mathbf{A} + \mathbf{D})\mathbf{G}^{-1} \mod q \in R_q$, then $|\sigma| = 32 + N \times (l \times \log_2(1 + 2 \times \delta) + k \times \log_2(1 + 2 \times \delta'))/8$ bytes and $|pk| = 32 + N \times (k \times l \times \log_2(q))/8$ bytes, where $\delta = l \times \eta_G \times (t + \tau)$, $\delta' = \eta_{y_2} + l \times \eta_D \times (t + \tau)$.

Table 6. Comparison of The Sizes of EagleSign and Dilithium

NIST security level	2	3	5
EagleSign Size in bytes			
signature size $(r, \mathbf{Z}, \mathbf{W})$	2144	2336	3488
public key size $(\rho, \mathbf{E})$	1824	1824	3616
Dilithium Size in bytes			
signature size $(c, \mathbf{z}, \mathbf{h})$	2420	3293	4595
public key size $(\rho, \mathbf{t})$	1312	1952	2592

From table 6, we see that the sizes of EagleSign are more small than those of Dilithium and in the particular case of recommended parameters, we save 1000 bytes relatively to Dilithium. We use the following python code to compute the sizes of the signature

```python
if __name__ == "__main__":

  # Parameters definition in the format
  # (n, q, k, l, eta_y1, eta_y2, eta_g, eta_d, t, tau)
  # per Nist Security Levels
  params = {
    2: (512, 12289, 2, 1, 1, 64, 6, 6, 140, 18),
    3: (1024, 12289, 1, 1, 1, 64, 1, 1, 140, 38),
    "3+": (512, 12289, 2, 2, 1, 32, 2, 1, 90, 18),
    5: (1024, 12289, 1, 2, 1, 32, 1, 1, 86, 18),
  }

  # Computing the sizes for each level
  print("Eagle\t|\tdelta\t|\tlog_delta|\tdelta_p\t|\tlog_delta_p|\t|Sig|\t|\t|Pk|")
  print("--------------------------------------------------\
--------------------------------------------------\
---------------")
  for idx in params.keys():
    param = params[idx]

    delta = param[3]*param[6]*(param[8]+param[9])
    delta_p = param[3]*param[7]*(param[8]+param[9]) + param[5]

    # Computing logdelta = bitLength(1+2*delta) and
    # logdelta_prime = bitLength(1+2*delta_p)
    logdelta = int(2*delta+1).bit_length()
    logdelta_p = int(2*delta_p+1).bit_length()

    # Computing |Sig| and |Pk|
    sigma = 32 + (param[3]*logdelta + param[2]*logdelta_p)*param[0]/8
    pk = 32 + (param[2]*param[3]*int(param[1]).bit_length())*param[0]/8
    print("{}\t|\t{}\t|\t{}\t |\t{}\t|\t{}\t   |\t{}\t|\t{}".format(idx, delta, logdelta, delta_p, logdelta_p, int(sigma), int(pk)))
    print("--------------------------------------------------\
--------------------------------------------------\
---------------")
```

5 Reference and optimized implementations

5.1 Bit/Byte Packing

In this section, we will explain the process of converting vectors and matrices into byte strings and vis-versa. The procedure used in our implementation is similar to the one used in Dilithium round 3. For completeness purpose, we will describe it in this section. The general rule that we will follow is that if the range of an element x consists exclusively of non-negative integers, then we simply pack the integer x. If x is from a range $[-a, b]$ that may contain some negative integers, then we pack the positive integer $b - x$.

Let's start with a single polynomial of N coefficients $N \in \{512, 1024\}$ where each coefficient is an integer which can be encoded on b bits. Then each set of 8 coefficients can be encoded on $8 * b/8 = b$ bytes.

In the case of EagleSign signature (r, Z, W), r is a byte array and does not need any conversion. Z is a vector of l elements $l \in \{1, 2\}$ where each polynomial's coefficient can be encoded on 9 bits for NIST Level 3 and 5. This means that each set of 8 coefficients of Z polynomials can be encoded on 9 bytes string as shown in figure 1. Similarly, each set of 8 coefficients of W polynomials can be encoded on 9 bytes string for both NIST levels 3 and 5.

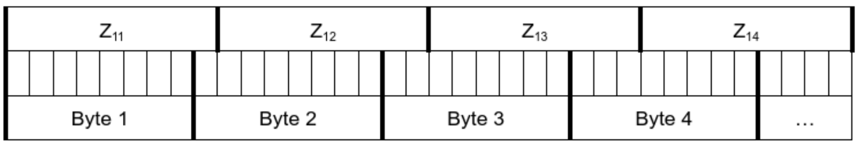

Fig. 1. Bit-Packing Z's polynomials' coefficient Z_{11} Z_{12}, Z_{13}, ... for Nist Levels 3 and 5

The previous described procedure has also been used to pack and unpack different other parameters including the matrix E in the public key as well as D and G in the private key.

In the following subsection, , we have provided a python code that we wrote in order to generate the set of instructions in C language to convert a list of 8 different b-bits coefficients into a bytes string for any odd integer b. When b is even and $b/2$ is odd, the same code can be customized to generate the set of instructions in C language to convert a list of 4 different b-bits coefficients into a bytes string.

5.2 Bit-Packing: Python Code for generating Bit-Packing instructions in C

```python
import numpy as np
import pandas as pd

if __name__ == "__main__":
  D = 9  # Change this value according to your need
```

```
dterm = "COEFF_BIT_SIZE"

X = [[i]*D for i in range(8)]
Y = [[-1]*8 for i in range(D)]

Z = [-1]*(8*D)

l = 0
for i in range(8):
  for j in range(D):
    Z[l] = X[i][j]
    l += 1

l = 0
for i in range(D):
  for j in range(8):
    Y[i][j] = Z[l]
    l += 1

ta = []
tb = []
for y in Y:
  y = pd.Series(y)
  c = dict(y.value_counts())
  ta.append(c)
  for key in c.keys():
    tb.append({key: c[key]})

print("\nunsigned int i;\nint16_t t[8];\nfor (i = 0; i < N /
    8; ++i)\n{\n")

for i in range(8):
  print(
  "    t[{0}] = (1 << ({1} - 1)) - a->coeffs[8 * i + {0}];".
  format(i, dterm))

print()

cp = 0
cp_key = 0
it = 0

for y in Y:
  y = pd.Series(y)
  c = dict(y.value_counts())
  init = 0
  sorted_ = list(c.keys())
  sorted_.sort()
  for key in sorted_:
    cp = cp % D
```

```
      if init == 0:
        print("    r[{} * i + {}] = t[{}]{};".format(D,
        it, key, " >> {}".format(cp) if cp else ""))

        init += c[key]
      else:
        print("    r[{} * i + {}] {}= t[{}]{};".format(D, it,
        "|" if init else "", key,
        " << {}".format(init) if init else ""))

        init += c[key]

      if (cp_key == key):
        cp += c[key]
      else:
        cp = c[key]

      cp_key = key

    it += 1

  print("\n}")
```

The out of the previous code is presented in the next code. Note that the output generated depends on three (04) parameters : N, r, a and $COEFF\_BIT\_SIZE$. r is the output byte array, a is the input polynomial, N is the number of components in polynomial a, $N \in \{512, 1024\}$ and $COEFF\_BIT\_SIZE$ is the coefficients' bits size.

```
unsigned int i;
int16_t t[8];
for (i = 0; i < N / 8; ++i)
{
  t[0] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 0];
  t[1] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 1];
  t[2] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 2];
  t[3] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 3];
  t[4] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 4];
  t[5] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 5];
  t[6] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 6];
  t[7] = (1 << (COEFF_BIT_SIZE - 1)) - a->coeffs[8 * i + 7];

  r[9 * i + 0] = t[0];
  r[9 * i + 1] = t[0] >> 8;
  r[9 * i + 1] |= t[1] << 1;
  r[9 * i + 2] = t[1] >> 7;
  r[9 * i + 2] |= t[2] << 2;
  r[9 * i + 3] = t[2] >> 6;
  r[9 * i + 3] |= t[3] << 3;
  r[9 * i + 4] = t[3] >> 5;
```

```
  r[9 * i + 4] |= t[4] << 4;
  r[9 * i + 5] = t[4] >> 4;
  r[9 * i + 5] |= t[5] << 5;
  r[9 * i + 6] = t[5] >> 3;
  r[9 * i + 6] |= t[6] << 6;
  r[9 * i + 7] = t[6] >> 2;
  r[9 * i + 7] |= t[7] << 7;
  r[9 * i + 8] = t[7] >> 1;
}
```

5.3 Bit-Unpacking: Python Code for generating Bit-Unpacking instructions in C

```
import numpy as np
import pandas as pd

if __name__=="__main__":
  D = 9  # Change this value according to your need
  Ty = "int16_t"
  dterm = "COEFF_BIT_SIZE"

  X = [[i] * D for i in range(8)]
  Y = [[-1] * 8 for i in range(D)]

  Z = [-1] * (8 * D)

  l = 0
  for i in range(8):
    for j in range(D):
      Z[l] = X[i][j]
      l += 1

  l = 0
  for i in range(D):
    for j in range(8):
      Y[i][j] = Z[l]
      l += 1

  ta = []
  tb = []
  for y in Y:
    y = pd.Series(y)
    c = dict(y.value_counts())
    ta.append(c)
    for key in c.keys():
      tb.append({key: c[key]})

  cp = 0
  cp_key = 0
  it = 0
```

```
  print("\nunsigned int i;\nfor (i = 0; i < N / 8; ++i)\n{\n")
  for y in Y:
    y = pd.Series(y)
    c = dict(y.value_counts())
    init = 0
    sorted_ = list(c.keys())
    sorted_.sort()
    for key in sorted_:
      cp = cp % D
      if init == 0:
        print("    r->coeffs[8 * i + {}] {}= {}a[{} * i + {}]{};".format(key, "|" if cp else "",
        "(%s)" % (Ty) if cp else "", D, it,
        " << {}".format(cp) if cp else ""))

        init += c[key]
      else:
        print("    r->coeffs[8 * i + {}] &= {};\n".format(cp_key, hex((2 << (D - 1)) - 1)))
        print("    r->coeffs[8 * i + {}] = a[{} * i + {}]{};".format(key, D, it,
        " >> {}".format(init) if init else ""))
        init += c[key]

      if (cp_key == key):
        cp += c[key]
      else:
        cp = c[key]

      cp_key = key

    it += 1
  print("    r->coeffs[8 * i + {}] &= {};\n".format(cp_key, hex((2 << (D - 1)) - 1)))

  for i in range(8):
    print("    r->coeffs[8 * i + {0}] = (1 << ({1} - 1)) - r->coeffs[8 * i + {0}];".format(i, dterm))

  print("\n}")
```

The out of the previous code is presented in the next code. Note that the output generated depends on three (04) parameters : N, r, a and $COEFF\_BIT\_SIZE$. a is the input byte array, r is the output polynomial, N is the number of components in polynomial r, $N \in \{512, 1024\}$ and $COEFF\_BIT\_SIZE$ is the coefficients' bits size.

```
unsigned int i;
for (i = 0; i < N / 8; ++i)
```

```
{
  r->coeffs[8 * i + 0] = a[9 * i + 0];
  r->coeffs[8 * i + 0] |= (int16_t)a[9 * i + 1] << 8;
  r->coeffs[8 * i + 0] &= 0x1ff;

  r->coeffs[8 * i + 1] = a[9 * i + 1] >> 1;
  r->coeffs[8 * i + 1] |= (int16_t)a[9 * i + 2] << 7;
  r->coeffs[8 * i + 1] &= 0x1ff;

  r->coeffs[8 * i + 2] = a[9 * i + 2] >> 2;
  r->coeffs[8 * i + 2] |= (int16_t)a[9 * i + 3] << 6;
  r->coeffs[8 * i + 2] &= 0x1ff;

  r->coeffs[8 * i + 3] = a[9 * i + 3] >> 3;
  r->coeffs[8 * i + 3] |= (int16_t)a[9 * i + 4] << 5;
  r->coeffs[8 * i + 3] &= 0x1ff;

  r->coeffs[8 * i + 4] = a[9 * i + 4] >> 4;
  r->coeffs[8 * i + 4] |= (int16_t)a[9 * i + 5] << 4;
  r->coeffs[8 * i + 4] &= 0x1ff;

  r->coeffs[8 * i + 5] = a[9 * i + 5] >> 5;
  r->coeffs[8 * i + 5] |= (int16_t)a[9 * i + 6] << 3;
  r->coeffs[8 * i + 5] &= 0x1ff;

  r->coeffs[8 * i + 6] = a[9 * i + 6] >> 6;
  r->coeffs[8 * i + 6] |= (int16_t)a[9 * i + 7] << 2;
  r->coeffs[8 * i + 6] &= 0x1ff;

  r->coeffs[8 * i + 7] = a[9 * i + 7] >> 7;
  r->coeffs[8 * i + 7] |= (int16_t)a[9 * i + 8] << 1;
  r->coeffs[8 * i + 7] &= 0x1ff;

  r->coeffs[8 * i + 0] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 0];
  r->coeffs[8 * i + 1] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 1];
  r->coeffs[8 * i + 2] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 2];
  r->coeffs[8 * i + 3] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 3];
  r->coeffs[8 * i + 4] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 4];
  r->coeffs[8 * i + 5] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 5];
  r->coeffs[8 * i + 6] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 6];
  r->coeffs[8 * i + 7] = (1 << (COEFF_BIT_SIZE - 1)) - r->
    coeffs[8 * i + 7];

```

```
}
```

5.4 NTT transformation

The NTT transformation is particularly advantageous when dealing with large polynomials or performing polynomial multiplications and convolutions. Unlike the traditional polynomial multiplication algorithms, such as the schoolbook method or Karatsuba algorithm, the NTT algorithm reduces the complexity from $O(n^2)$ to $O(n \log n)$. This speedup becomes especially pronounced as the polynomial size grows, making it an appealing choice for high-performance computing applications.

In EagleSign Nist Level 3 and 5, the NTT transformation allows for faster implementations of public key, signature and verification operations over over the ring $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^n+1)}$, $q = 12289, N = 1024$ by speeding the polynomials multiplications and divisions operations. Our NTT implementations over the aforementioned ring follows the implementation proposed by Falcon since we use the same field than Falcon. The implementation of our signature in case $N = 512$ is not finished yet.

5.5 Hashing and Sampling techniques, special functions

Sampling y_2 : The function GenVectorUnifSmallPoly($\lambda_2 = (\lambda, l)$) maps (λ_2) to $y_2 \in S^k_{\eta_{y_2}}$. We compute independently the k components of y_2. Note that these components are polynomials in $S_{\eta_{y_2}}$. For the i-th polynomial, $0 \leq i < k$, it absorbs the 48 bytes of λ_2 concatenated with the 2 bytes representing $l + i$ in little endian byte order into SHAKE-256.

Sampling invertible $\mathbf{G}$: The function GenMatrixUnifSmallPolyn(β_1, l, l) maps (β_1, l, l) to $\mathbf{G} \in S^{l\times l}_{\eta_G}$. We compute independently the $l \times l$ components of $\mathbf{G}$. For each polynomial $\mathbf{G}_{(i,j)}$, $0 \leq i, j < l$, it absorbs the 48 bytes of β_1 concatenated with the 2 bytes representing $i \times l + j$ in little endian byte order into SHAKE-256. If $\mathbf{G}$ is not invertible, we renew the seed β_1 by computing $\beta_1 =$ SHAKE-256(β_1) until $\mathbf{G}$ is invertible. Remark that this algorithm terminates quickly since the ring $R_q, q = 12289$ contains enough invertible polynomials.

Sampling $\mathbf{D}$: The function GenMatrixUnifSmallPolyn(β_2, k, l) maps (β_2, k, l) to $\mathbf{D} \in S^{k\times l}_{\eta_D}$. We compute independently the $k \times l$ components of $\mathbf{D}$. For each polynomial $\mathbf{D}_{(i,j)}$, $0 \leq i < k, 0 \leq j < l$, it absorbs the 48 bytes of β_2 concatenated with the 2 bytes representing $i \times l + j$ in little endian byte order into SHAKE-256.

Sampling $\mathbf{y}_1 \in B^l_t$: The function GenVectorSparsePoly(ρ) maps ρ to $\mathbf{y}_1 \in B^l_t$. The seed ρ is defined by $\rho = (\lambda, 0)$ where λ is generated randomly. We compute independently the l components of $\mathbf{y}_1$. For each polynomial $\mathbf{y}_{1,i}$, $0 \leq i < l$, it absorbs the 48 bytes of λ concatenated with the 2 bytes representing i in little endian byte order into XOF interpreted as SHAKE/STREAM-128 of the FIPS202 standard. The output of the XOF is used to generate $\mathbf{y}_{1,i} = e$ in a Ball as follows:

- Initialize $e = e_0 e_1 \dots e_{N-1} = 0 \dots 0$
- for $i = N - t$ to N
  - $b \xleftarrow{\$} \{0, 1, \dots, i\}$ with XOF
  - $e_i := e_b$
  - $s \xleftarrow{\$} \{0, 1\}$ with XOF
  - $e_b := 1 - 2s$
- return e

Note that in the expression $\mathbf{y}_{1,i} = e$, e is used to simplify the notation in the previous algorithm.

Computing $\mathbf{c} = H(\mu, r) \in B_\tau^l$: The cryptographic Hash function H maps (μ, r) to $\mathbf{c} \in B_\tau^l$. For this purpose we first extract 384 bits of the output of SHAKE-256 onto the input μ, r in this order as a seed seed$_c$. We then compute independently the l components of $\mathbf{c}$. For each polynomial $\mathbf{c}_i$, $0 \le i < l$, we absorbs the 48 bytes of seed$_c$ concatenated with the 2 bytes representing i in little endian byte order into XOF interpreted as SHAKE/STREAM-128 of the FIPS202 standard. The output of the XOF is used to generate $\mathbf{c}_i = d$ in a Ball as follows:

- Initialize $d = d_0 d_1 \dots d_{N-1} = 0 \dots 0$
- for $i = N - \tau$ to N
  - $b \xleftarrow{\$} \{0, 1, \dots, i\}$ with XOF
  - $d_i := d_b$
  - $s \xleftarrow{\$} \{0, 1\}$ with XOF
  - $d_b := 1 - 2s$
- return d

Note that in the expression $\mathbf{c}_i = d$, d is used to simplify the notation in the previous algorithm.

Sampling the Matrix A : The function GenMatrixUnifPolyn maps a uniform seed $\rho \in \{0, 1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times l}, q = 11289, N \in \{512, 1024\}$ in NTT domain representation. $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$. We computes independently the components $\hat{\mathbf{a}}_{i,j} \in R_q$ of $\hat{\mathbf{A}}$. We use SHAKE-128 to compute the coefficient $\hat{\mathbf{a}}_{i,j}$ by absorbing the 32 bytes of ρ followed by 2 bytes representing $0 \le 2^8 \times i + j < 2^{16}$ in little-endian byte order. The output stream of SHAKE-128 is interpreted as a sequence of integers between 0 and $2^{14} - 1$, where 14 is the bit-size of prime $q = 12289$ which is used. To obtain such result, we set the highest bit of every second byte to zero and interpreting blocks of 2 consecutive bytes in little endian byte order. In practice, the two consecutive bytes b_0 , b_1 are used to get the integer $0 \le t = b_1' \times 2^8 + b_0 \le 2^{14} - 1$ where b_1' is the logical AND of b_1 and $2^6 - 1$. Another method is to compute t as the logical

AND of $t' = b_1 \times 2^8 + b_0$ and $2^{14} - 1$. Finally, GenMatrixUnifPolyn performs rejection sampling on these 14-bit integers t to sample the N coefficients between 0 and $q - 1$.

Collision resistant hash (CRH1, CRH) The function CRH1 and CRH are collision resistant hash functions. For this purpose 256 and 384 bits of the output of SHAKE-256 are used respectively for CRH1 and CRH. Note that we can easily choose and integrate other hash functions.
CRH1 is called on the public Key $(\rho, \mathbf{E})$ to compute tr. For this reason, it takes as input the byte string obtained from packing ρ and $\mathbf{E}$ in this order and the result is absorbed into SHAKE-256 and the first 32 output bytes are used as the resulting hash.

CRH on the other hand is called on the input $tr||M$ to compute μ. Here the concatenation of the hash tr and the message string M are absorbed into SHAKE-256 and the first 48 output bytes are used as the resulting hash.

Collision resistant hash (G) The function G is a collision resistant hash function. For this purpose 256 bits of the output of SHAKE-256 is used. G is called the input P to compute r in the signature and on V to compute r' in the verification algorithm. Note that we can easily choose and integrate other hash function.

NB: EagleSign is more simple than Dilithium because it does not use the auxiliary functions of Dilithium such as HighBits, MakeHint, UseHint, Power2Round, Decompose and SelfTargetMSIS.

5.6 Optimized Implementation

In our optimized implementation, the main function that we have optimized include : Adding, subtracting and multiplying polynomials since they are the key basic operations that we have used over the ring $R_q = \dfrac{\mathbb{Z}_q(X)}{(X^N+1)}$, $q = 12289, N = 1024$ which concerns EagleSign Nist Level 3 and 5. Our optimized implementation follows the one in Dilithium and we have used ChatGPT sometimes.

6 Advantages and Limitations

Advantages: EagleSign is more simple and faster than Falcon and Dilithium. The sizes are similar to those of Dilithium, but for recommended parameters, the sizes of EagleSign are more small than those of Dilithium.
Limitations: It has the same limitations as any lattices based digital signature regarding the long term security.

References

1. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. *Post-quantum key exchange - A new hope.* In T. Holz and S. Savage, editors, Proceedings of the 25th USENIX Security Symposium, pages 327-343. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/techniqueal-sessions/presentation/alkim.
2. M. R. Albrecht, B. R. Curtis, A. Deo, A. Davidson, R. Player, E. Postlethwaite, F. Virdia, T. Wunderer, *Estimate all the LWE and NTRU schemes!* https://estimate-all-the-lwe- ntru-schemes.github.io/paper. pdf. NIST Call for transision to quantum-resistant cryptography (November 2017)
3. Martin Albrecht. *Security estimates for the learning with errors problem*, 2017. Version 2017-09-27, https://bitbucket.org/malb/lwe-estimator. 21
4. Martin R. Albrecht, Rachel Player and Sam Scott. *On the concrete hardness of Learning with Errors.* Journal of Mathematical Cryptology. Volume 9, Issue 3, Pages 169–203, ISSN (Online) 1862-2984, ISSN (Print) 1862-2976 DOI: 10.1515/jmc-2015-0016, October 2015
5. Martin Albrecht and Amit Deo. *Large modulus Ring-LWE $\geq$ Module-LWE*, 2017.To appear. https: //eprint.iacr.org/2017/612. 22
6. Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. *Revisiting the expected cost of solving uSVP and applications to LWE.* In Advances in Cryptology -ASIACRYPT 2017 -23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I, pages 297 322, 2017
7. M. R. Albrecht, C. Cid, J.C. Faugere, and L. Perret. *Algebraic algorithms for LWE.* Cryptology ePrint Archive, Report 2014/1018, 2014. http://eprint.iacr.org/2014/1018
8. Albrecht M., Bai S., Ducas L. *A Subfield Lattice Attack on Overstretched NTRU Assumptions.* In: Robshaw M., Katz J. (eds) Advances in Cryptology - CRYPTO 2016. Lecture Notes in Computer Science, vol 9814. Springer, Berlin, Heidelberg, pp 153-178.
9. M. R. Albrecht, R. Player, and S. Scott. *On the concrete hardness of learning with errors.* J. Mathematical Cryptology, 9(3):169-203, 2015. URL: http://www.degruyter.com/view/j/jmc.2015. 9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.
10. Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi. *Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator.* In Marc Fischlin and Jean-Sébastien Coron, editors, Advances in Cryptology-EUROCRYPT 2016, volume 9665 of LNCS, pages 789-819. Springer, 2016. https://eprint.iacr.org/2016/146. 20
11. Aharonov, D., Regev, O.: *A lattice problem in quantum NP.* In: FOCS, pp. 210-219 (2003).
12. Ajtai, M.: *The shortest vector problem in L_2 is NP-hard for randomized reductions.* In: STOC, pp. 10-19 (1998).
13. Ambainis, A.: *Quantum walk algorithm for element distinctness.* In: FOCS, pp. 22-31 (2003).
14. S. Arora and Rong Ge. *New algorithms for learning in presence of errors.* In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, ICALP 2011, Part I, volume 6755 of LNCS, pages 403-415. Springer, Heidelberg, July 2011.

15. S. Bai, D. Stehlé and W. Wen. *Improved Reduction from the Bounded Distance Decoding Problem to the Unique Shortest Vector Problem in Lattices.* In Springer Proc. of ICALP'2016, pp. 76:1-76:12.
16. S. Bai and S. D. Galbraith. *Lattice decoding attacks on binary LWE* In Willy Susilo and Yi Mu, editors, ACISP 14, volume 8544 of LNCS, pages 322-337. Springer, Heidelberg, July 2014
17. Shi Bai, Austin Beard, Floyd Johnson, Sulani K. B. Vidhanalage, Tran Ngo. *Fiat-Shamir Signatures Based on Module-NTRU.* In Khoa Nguyen, Guomin Yang, Fuchun Guo, Willy Susilo, editors, Information Security and Privacy - 27th Australasian Conference, ACISP 2022, Wollongong, NSW, Australia, November 28-30, 2022, Proceedings. Volume 13494 of Lecture Notes in Computer Science, pages 289-308, Springer, 2022. [doi]
18. A. Becker, L. Ducas, N. Gama, and T. Laarhoven. *New directions in nearest neighbor searching with applications to lattice sieving.* Robert Krauthgamer, editor. Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, DIOP 2016, Arlington, VA, USA, January 10-12, 2016. SIAM, 2016, pages 10-24. https://eprint.iacr.org/2015/1128.
19. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. *Relations among Notions of Security for Public-Key Encryption Schemes.* In Proc. of CRYPTO '98, LNCS 1462, pages 26-45. Springer-Verlag, Berlin, 1998
20. M. Bellare and P. Rogaway. *Random Oracles Are Practical : a Paradigm for Designing Efficient Protocols.* In Proc. of the 1st CCS, pages 62-73. ACM Press, New York, 1993
21. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. *NTRU Prime.* In Jan Camenisch and Carlisle Adams, editors, Selected Areas in Cryptography - SAC 2017, LNCS, to appear. Springer, 2017. http://ntruprime.cr.yp.to/papers.html
22. R. Canetti, O. Goldreich and S. Halevi, *The random oracle methodology*, revisited, STOC'98, ACM, 1998.
23. Hao Chen, Kristin Lauter, and Katherine E. Stange. *Vulnerable Galois RLWE families and improved attacks.* IACR Cryptology ePrint Archive, 2016. *https://eprint.iacr.org/2016/193.*
24. Yuanmi Chen and Phong Q. Nguyen. *BKZ 2.0: Better lattice security estimates.* In Dong Hoon Lee and Xiaoyun Wang, editors, Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings, volume 7073 of LNCS, pp. 1-20. Springer.T 97, volume 1233 of Lecture Notes in Comput. Sci., pp 52-61. Springer, Berlin, 1997.
25. Chuengsatiansup, Chitchanok et al. "ModFalcon: Compact Signatures Based On Module-NTRU Lattices." Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (2020): n. pag.
26. Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. *Recovering Short Generators of Principal Ideals in Cyclotomic Rings* Marc Fischlin and Jean-Sébastien Coron (Eds.). In Advances in Cryptology Eurocrypt May 2016, Lecture Notes in Computer Science, Springer-Verlag,Proceedings, Part II, pp. pp 559-585.
27. D. Dadush, O. Regev, and N. Stephens-Davidowitz. *On the closest vector problem with a distance guarantee.* In Proc. of CCC, pages 98-109. IEEE Computer Society Press, 2014.
28. T. El Gamal. *A public key cryptosystem and signature scheme based on discrete logarithms.* IEEE Trans. Inform. Theory, 31:469-472, 1985

29. A. Fiat, A. Shamir, *How to prove yourself: practical solutions to identification and signature problems*, Advances in Cryptology—Proceedings of Crypto '86, LNCS, vol. 263, Springer, 1987, pp. 186–194.
30. R. Fujita. *Table of underlying problems of the NIST candidate algorithms.* Available at https://groups.google.com/a/list.nist.gov/d/ msg/pqc-forum/1lDNio0sKq4/7zXvtfdZBQAJ, 2017
31. Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie *Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems.* Marc Fischlin and Jean-Sébastien Coron (Eds.), In Advances in cryptology Eurocrypt May 2016, Lecture Notes in Computer Science, Springer-Verlag Proceedings, Part II, pp. 528-558.
32. Craig Gentry,Chris Peikert,Vinod Vaikuntanathan, STOC '08: Proceedings of the fortieth annual ACM symposium on Theory of computingMay 2008Pages 197–206https://doi.org/10.1145/1374376.1374407
33. Goldwasser S., Micali S. and Rivest R. , A digital signature scheme secure against adaptive chosen- message attacks, SIAM Journal of computing, 17(2), pp. 281-308, April 1988.
34. Grover, L. K.: *A fast quantum mechanical algorithm for database search.* In: STOC, pp. 212-219 (1996)39.
35. Grover, L. K., Rudolph, T.: *How significant are the known collision and element distinctness quantum algorithms?* Quantum Info. Comput.4 (3), pp. 201-206 (2004).
36. Jung Hee Cheon, Jinhyuck Jeong, Changmin Lee *An Algorithm for NTRU Problems and Cryptanalysis of the GGH Multilinear Map without a Low Level Encoding of Zero.* IACR Cryptology ePrint Archive, *https://eprint.iacr.org/2016/139.pdf.*
37. Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. *Terminating BKZ.* IACR Cryptology ePrint Archive report 2011/198, 2011. https://eprint.iacr.org/2011/198.
38. J. Hoffstein, J. Pipher, and J. H. Silverman. *NTRU: A Ring Based Public Key Cryptosystem in Algorithmic Number Theory*, Lecture Notes in Computer Science 1423, Springer-Verlag, pp. 267-288, 1998.
39. Thijs Laarhoven. *Sieving for shortest vectors in lattices using angular locality-sensitive hashing.* In Rosario Gennaro and Matthew Robshaw, editors, Advances in Cryptology CRYPTO 2015 -35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I, volume 9215 of Lecture Notes in Computer Science, pages 3-22. Springer, 2015. https://eprint.iacr.org/2014/744.pdf.
40. Thijs Laarhoven, Michele Mosca, and Joop van de Pol.*Finding shortest lattice vectors faster using quantum search.* Des. Codes Cryptography, 77(2-3):375-400, 2015.
41. M. Liu, X. Wang, G. Xu, and X. Zheng. *A note on BDD problems with λ2-gap.* Inf. Process. Lett., 114(1-2):9-12, January 2014.
42. Y. K. Liu, V. Lyubashevsky, and D. Micciancio. *On bounded distance decoding for general lattices.* In Proc. of RANDOM, volume 4110 of LNCS, pages 450-461. Springer, 2006.
43. V. Lyubashevsky, C. Peikert, and O. Regev. *On ideal lattices and learning with errors over rings.* In EUROCRYPT 2010, pages 1-23. Springer, 2010.
44. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *A toolkit for ring-LWE cryptography.* In EUROCRYPT 2013, pp. 35-54.

45. V. Lyubashevsky and D. Micciancio. *On bounded distance decoding, unique shortest vectors, and the minimum distance problem.* In Proc. of CRYPTO 2009, pp. 577-594.
46. Vadim Lyubashevsky. *Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures.* In ASIACRYPT, pages 598–616, 2009. 2, 3, 21
47. Vadim Lyubashevsky. *Lattice signatures without trapdoors.* In EUROCRYPT, pages 738–755, 2012. 3, 5, 6, 21, 28
48. Qipeng Liu and Mark Zhandry. *Revisiting post-quantum fiat-shamir.* Cryptology ePrint Archive, Report 2019/262, 2019. https://eprint.iacr.org/ 2019/262. 6
49. Micciancio, D., Voulgaris, P.: *Faster exponential time algorithms for the shortest vector problem.* In DIOP(2010), pp. 1468-1480.
50. D. Moody. *The NIST post quantum cryptography competition.* Available at https://csrc.nist.gov/CSRC/media/Projects/ Post-Quantum-Cryptography/documents/asiacrypt-2017-moody-pqc.pdf, 2017.
51. M. Naor and M. Yung. Public Key Cryptosystems *Provably Secure against Chosen Ciphertext Attacks.* In Proc. of the 22nd ACM STOC, pages 427-437. ACM Press, New York, 1990.
52. NIST Post-Quantum Cryptography- Call for Proposals. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization /Call-for-Proposals. List of First Round candidates available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions
53. National Institute of Standards and Technology. *Performance testing of the NIST candidate algorithms.* Available at https://drive.google.com/ file/d/1g-l0bPa-tReBD0Frgnz9aZXpO06PunUa/view, 2017
54. NIST. Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process, jun 2022 https://csrc.nist.gov/Projects/pqc-dig-sig
55. Hiroki OKADA, Atsushi TAKAYASU, Kazuhide FUKUSHIMA, Shinsaku KIYOMOTO and Tsuyoshi TAKAGI *A Compact Digital Signature Scheme Based on the Module-LWR Problem* Journal: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2021, Volume E104 A, Number 9, Page 1219 DOI: 10.1587/transfun 2020DMP0012
56. Xavier Pujol and Damien Stehlé. *Solving the shortest lattice vector problem in time* $2^{2,465.n}$. IACR Cryptology ePrint Archive, 2009. *https://eprint.iacr.org/2009/605.*
57. C. Peikert. *A useful fact about Ring-LWE that should be known better: it is *at least as hard* to break as NTRU, and likely strictly harder.* Available at http://archive.is/B9KEW.
58. C. Peikert. *Public-key cryptosystems from the worst-case shortest vector problem.* In STOC 2009, pp. 333-342. ACM.
59. C. Peikert and B. Waters. *Lossy trapdoor functions and their applications.* In STOC 2008, pages 187-196, 2008.
60. Pointcheval, D., Stern, J. (1996). Security Proofs for Signature Schemes. In: Maurer, U. (eds) Advances in Cryptology — EUROCRYPT '96. EUROCRYPT 1996. Lecture Notes in Computer Science, vol 1070. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-68339-9_33
61. Regev, O. *On lattices, learning with errors, random linear codes, and cryptography.* In: STOC, pp. 84-93 (2005).

62. O. Regev. *On lattices, learning with errors, random linear codes, and cryptography.* J. ACM, 56(6), 2009.
63. Regev, O.:*Lattices in computer science.* Lecture notes for a course at the Tel Aviv University (2004)78.
64. Regev, O.:*Quantum computation and lattice problems.* SIAM J. Comput. 33 (3), pp. 738-760 (2004).
65. Santha, M.: *Quantum walk based search algorithms.* In: TAMC (2008), pp. 31-46 .
66. Schneider, M.: *Analysis of Gauss-Sieve for solving the shortest vector problem in lattices.* In: WALCOM (2011), pp. 89-97.
67. Schneider, M.: *Sieving for short vectors in ideal lattices.* In: AFRICACRYPT (2013), pp. 375-391.
68. C. P. Schnorr and M. Euchner. *Lattice basis reduction: Improved practical algorithms and solving subset sum problems.* Mathematical Programming, 66(1):181-199, 1994
69. Shor, P.W.:*Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM J. Comput. 26 (5), pp. 1484-1509 (1997).
70. D. Stehlé and R. Steinfeld. *Making NTRU as secure as worst-case problems over ideal lattices.* Draft of full extended version of Eurocrypt 2011 paper, ver. 10, Oct. 2011. Available at *http://web.science.mq.edu.au.*
71. D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa. *Efficient public key encryption based on ideal lattices.* In ASIACRYPT 2009, pp. 617-635. Springer.
72. D. Stehlé and R. Steinfeld. *Making NTRU as secure as worst-case problems over ideal lattices.* In EUROCRYPT 2011, pp. 27-47. Springer.
73. Wang, X., Liu, M., Tian, C., Bi, J.: *Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem.* In: ASIACCS (2011), pp. 1-9.
74. T. Wunderer. *Revisiting the hybrid attack: Improved analysis and refined security estimates.* Cryptology ePrint Archive, Report 2016/733,2016. http://eprint.iacr.org/2016/733
75. Zhang, F., Pan, Y., Hu, G.: *A three-level sieve algorithm for the shortest vector problem.* In: SAC (2013), pp. 29-47.