

HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures

Authors	Jung Hee Cheon (Seoul National University / CryptoLab, Inc.) Hyeongmin Choe (University of Luxembourg) Julien Devevey (ANSSI) Tim Güneysu (Ruhr University Bochum / DFKI) Dongyeon Hong (Samsung Electronics) Minhyeok Kang (CryptoLab, Inc.) Taekyung Kim (CryptoLab, Inc.) Jeongbeen Ko (CryptoLab, Inc.) Markus Krausz (Ruhr University Bochum) Georg Land (Intel) Marc Möller (Ruhr University Bochum) Junbum Shin (CryptoLab, Inc.) Damien Stehlé (CryptoLab, Inc.) MinJune Yi
Contact	<code>taekyung.kim@cryptolab.co.kr</code>
Homepage	<code>kpqc.cryptolab.co.kr</code>
Reference Code	<code>github.com/CryptoLabInc/HAETAE</code> Integrity Hash (SHA-256): <code>fcdc0a71 f72117f4 396eaa77 b5b50a6a</code> <code>4dcf3410 0372b967 84381fc8 f51ff0a1</code>

January 30, 2026

Abstract

We present HAETAE (Hyperball bimodal module rejection signature scheme), a new lattice-based signature scheme. Like the NIST-selected Dilithium signature scheme, HAETAE is based on the Fiat-Shamir with Aborts paradigm, but our design choices target an improved complexity/compactness compromise that is highly relevant for many space-limited application scenarios. We primarily focus on reducing signature and verification key sizes so that signatures fit into one TCP or UDP datagram while preserving a high level of security against a variety of attacks. As a result, our scheme has signature and verification key sizes up to 39% and 25% smaller, respectively, compared than Dilithium. We provide a portable, constant-time reference implementation together with an optimized implementation using AVX2 instructions and an implementation with reduced stack size for the Cortex-M4. Moreover, we describe how to efficiently protect HAETAE against implementation attacks such as side-channel analysis, making it an attractive candidate for use in IoT and other embedded systems.

1 Introduction

We introduce HAETAE¹, a new post-quantum digital signature scheme, whose security is based on the hardness of the module versions of the lattice problems LWE and SIS. The scheme design follows the “Fiat-Shamir with Aborts” paradigm [19, 20], which relies on rejection sampling: rejection sampling is used to transform a signature trial whose distribution depends on sensitive information, into a signature whose distribution can be publicly simulated. Our scheme is in part inspired from CRYSTALS-Dilithium [10], a post-quantum “Fiat-Shamir with Aborts” signature scheme which was selected for standardization by the American National Institute of Standards and Technology (NIST). HAETAE differs from Dilithium in two major aspects: (i) we use a bimodal distribution for the rejection sampling, like in the BLISS signature scheme [9], instead of a “unimodal” distribution like Dilithium, (ii) we sample from and reject to hyperball uniform distributions, instead of discrete hypercube uniform distributions. This last aspect also departs from BLISS, which relies on discrete Gaussian distributions, and follows a suggestion from [8], which studied rejection sampling in lattice-based signatures following the “Fiat-Shamir with Aborts” paradigm.

1.1 Design rationale

A brief recap on Fiat-Shamir with Aborts. The Fiat-Shamir with Aborts paradigm was introduced in lattice-based cryptography in [19, 20]. The verification key is a pair of matrices $(\mathbf{A}, \mathbf{T} = \mathbf{A}\mathbf{S} \bmod q)$, where \mathbf{A} is a uniform matrix modulo some integer q and \mathbf{S} is a small-magnitude matrix that makes up the secret key. A signature for a message M is comprised of an integer vector \mathbf{z} of the form $\mathbf{y} + \mathbf{S}\mathbf{c}$, for some random small-magnitude \mathbf{y} and some small-magnitude challenge $\mathbf{c} = H(\mathbf{A}\mathbf{y} \bmod q, M)$. Rejection sampling is then used to ensure that the distribution of the signature becomes independent from the secret key. Finally, the verification algorithm checks that the vector \mathbf{z} is short and that $\mathbf{c} = H(\mathbf{A}\mathbf{z} - \mathbf{T}\mathbf{c} \bmod q, M)$.

Improving compactness. As analyzed in [8], The choice of the distributions to sample from and reject to has a major impact on the signature size. Dilithium relies on discrete uniform distributions in hypercubes, which makes the scheme easier to implement. However, such distributions are far from optimal in terms of resulting signature sizes. We choose a different trade-off: by losing a little on ease of implementation, we obtain more compact signatures.

Uniform distributions in hyperballs. A possibility would be to consider Gaussian distributions, which are superior to uniform distributions in hypercubes, in terms of resulting signature compactness (see, e.g., [8]). However, this choice has two downsides. First, the rejection step involves the computation of a transcendental function on an input that depends on the secret key. This is cumbersome to implement and sensitive to side-channel attacks [13]. Second, since the final signature follows a Gaussian distribution there is a nonzero probability that the final signature is too large and does not pass the verification. The signer must realise that and reject the signature, making the expected

¹The haetae is a mythical Korean lion-like creature with the innate ability to distinguish right from wrong.

number of rejects slightly grow in practice. Uniform distributions over hyperballs have been put forward in [8] as an alternative choice of distributions leading to signatures with compactness between those obtained with Gaussians and those obtained with hypercube uniforms. Compared to Gaussians, they do not suffer from the afore-mentioned downsides: the rejection step is simply checking whether Euclidean norms are sufficiently small; and as there is no tail, there is no need for an extra rejection step to ensure that verification will pass. HAETAE showcases that this provides an interesting simplicity/compactness compromise.

Bimodal distributions. A modification of Lyubashevsky’s signatures was introduced in [9]. It allows for the use of bimodal distributions in the signature generation. The signature is now of the form $\mathbf{y} + (-1)^b \mathbf{S}\mathbf{c}$, where \mathbf{y} is sampled from a fixed distribution and $b \in \{0, 1\}$ is sampled uniformly. The signature is then rejected to a given secret-independent target distribution. To make sure that the verification test passes, computations are performed modulo $2q$ and key generation forces the equality $\mathbf{A}\mathbf{S} = q\mathbf{Id}$. It turns out that this modification can lead to more compact signatures than the unimodal setup. In [9], the authors relied on discrete Gaussian distributions. We instead use uniform distributions over hyperballs: like for Gaussians, switching from unimodal to bimodal for hyperball-uniforms leads to more compact signatures.

Flexible design by working with modules. The original design for BLISS [9] relies on Ring-LWE and Ring-SIS, and a variant of the key generation algorithm relied on ratios of polynomials, *à la* NTRU. This setup forces to choose a working polynomial ring for any desired security level. In order to offer more flexibility without losing in terms of implementation efficiency, we choose to rely on module lattices, like Dilithium, with a fixed working polynomial ring $\mathcal{R} = \mathbb{Z}[x]/(x^{256} + 1)$ across all security levels. In our instantiations, we target the NIST PQC security levels 2, 3 and 5. Varying the security and updating the parameters is easily achievable and we provide a security estimator that is able to help one reach a given target security.

Secret key rejection sampling. We introduce a new rejection procedure in the key generation algorithm to minimize the magnitude of the secret key when multiplied by the challenge. This facilitates rejection sampling in the signing algorithm and leads to smaller signatures. The key generation rejection is also designed to be efficient and simple to implement. It significantly improves over a procedure with a similar objective in the key generation of BLISS.

A compact verification key. The flexibility provided by modules allows us to reduce the verification key size. Instead of taking the challenge \mathbf{c} as a vector over \mathcal{R} , we choose it in \mathcal{R} : the main condition on the challenge is that it has high min-entropy, which is already the case for binary vectors over \mathcal{R} . As a result, the secret \mathbf{S} can be chosen as a vector over \mathcal{R} rather than a matrix. The key-pair equation $\mathbf{A}\mathbf{S} = q\mathbf{Id}$ then becomes $\mathbf{A}\mathbf{s} = q\mathbf{j}$, where \mathbf{j} is the vector starting with 1 and then continuing with 0’s. To further compress the verification key, we use verification key truncation adopted from Dilithium by taking into account the residue modulo 2. Our key generation algorithm just creates an MLWE sample $(\mathbf{A}_{\text{gen}}, \mathbf{b} - \mathbf{a} = \mathbf{A}_{\text{gen}}\mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}})$ modulo q , where \mathbf{a} is uniform random over \mathcal{R}_q^k . By truncating \mathbf{b} as $\mathbf{b} = \mathbf{b}_1 + \mathbf{b}_0$, we define a $k \times (k + \ell)$ matrix

\mathbf{A} as $\mathbf{A} = (-2(\mathbf{a} - \mathbf{b}_1) + q\mathbf{j} \mid 2\mathbf{A}_{\text{gen}} \mid 2\mathbf{Id}_k) \bmod 2q$. The key-pair equation is satisfied for $\mathbf{s} = (1 \mid \mathbf{s}_{\text{gen}} \mid \mathbf{e}_{\text{gen}} - \mathbf{b}_0)$. The verification key consists of $(\mathbf{A}_{\text{gen}}, \mathbf{a}, \mathbf{b}_1)$. As $(\mathbf{a} \mid \mathbf{A}_{\text{gen}})$ is uniformly distributed, we can generate it from a seed using an extendable output function, and the verification key is reduced to the seed and the vector \mathbf{b}_1 . If we had kept the original key-pair equation $\mathbf{AS} = q\mathbf{Id}$, then the appropriately modified variant of our key-generation algorithm would have led to a verification key that is a matrix (with a seed) rather than a vector (with a seed).

Compression techniques to lower the signature size. We use two techniques to compress the signatures. First, as the verification key \mathbf{A} is in (almost)-HNF, we can use the Bai-Galbraith technique [2]. Namely, the second part of the signature, which is multiplied by $2\mathbf{Id}$ in the challenge computation and verification algorithm, can be aggressively compressed by cutting its low bits. This requires in turn modifying the computation of the challenge \mathbf{c} and the verification algorithm, in order to account for this precision loss. Usually, this is done by keeping only the high bits of $\mathbf{A}\mathbf{y}$ in the computation of the challenge. However, as we multiply everything by 2, we do not keep the lowest bit of those high bits and keep the (overall) least significant bit instead. As in Dilithium, our decomposition of bits technique is a Euclidean division with a centered remainder, and we choose a representative range for modular integers that starts slightly below zero to further reduce the support of the high bits. The second compression technique, suggested in [15] in the context of lattice-based hash-and-sign signatures, concerns the choice of the binary representation of the signature. As the largest part of it consists in a vector that is far from being uniform, we can choose some entropic coding to obtain a signature size close to its entropy. In particular, as in [15], we choose the efficient range Asymmetric Numeral System to encode our signature, as it allows us to encode the whole signature and not lose a fraction of a bit per vector coordinate, like with Huffman coding. We can further apply the two techniques to the hint vector \mathbf{h} , which is also a part of the signature, to reduce the signature sizes.

Efficient choice of modulus. We choose the prime q to be a good prime in the sense that the ring operations can be implemented efficiently and that the decomposition of bits algorithms, are correctly operated. For ring operations, we use the Number Theoretic Transform (NTT) with a fully splitting polynomial ring. The polynomial ring \mathcal{R} fully splits modulo q when the multiplicative group \mathbb{Z}_q^\times has an element of order 512, or equivalently when $q = 1 \bmod 512$. We choose $q = 64513$, which indeed satisfies this property. Interestingly, it fits in 16 bits, which allows dense storing on embedded devices. Furthermore, it is close to the next power of two, which is convenient for the sampling of uniform integers modulo q .

Fixed-point algorithm for hyperball sampling. Unlike uniform Gaussian sampling or uniform hypercube sampling, uniform hyperball sampling has not been considered in the cryptographic protocols before the suggestion of [8]. To narrow the gap between the hyperball uniforms sampled in the real and the ideal world, we discretize the hyperball and bound the numerical error and their effect by analyzing their propagation. This leads to a fixed-point hyperball sampling algorithm and, therefore, the fixed-point implementation of the whole signing process.

Deterministic and randomized version. HAETAE can be set in a deterministic or randomized mode. We focus on the deterministic version, but we also give the randomized version. Note that in the randomized version, a significant part of the signing algorithm can be executed off-line as it does not depend on the message.

1.2 Advantages and limitations

1.2.1 Advantages

- Our scheme relies on the difficulty of hard lattice problems, which have been well-studied for a long time.
- Signature sizes are 29% to 39% smaller than those of Dilithium at comparable security levels, and verification keys are 20% to 25% smaller.
- Implementation-wise, while our design rationale departs from Dilithium’s, the scheme remains implementation-friendly. In particular,
 - the rejection step only involves computations of Euclidean norms,
 - the whole signing process can be implemented with fixed-point arithmetic,
 - a significant message-independent part of signing can be performed “off-line”, for the randomized version of the scheme.

Comparison with hash-and-sign lattice signatures. In terms of ease of implementation, our scheme favorably compares to lattice signatures based on the hash and sign paradigm such as Falcon [16]. The efforts for making it masking-friendly, namely Mitaka [14], were recently broken [21]. HAETAE, Falcon and Mitaka all three rely on some form of Gaussian sampling, which are typically difficult to implement and protect against side-channel attacks. Falcon makes sequential calls to a Gaussian sampler over \mathbb{Z} with arbitrary centers. Mitaka also relies on an integer Gaussian sampler with arbitrary centers, but the calls to it can be massively parallelized. It also uses a continuous Gaussian sampler, which is arguably simpler. HAETAE, however, only relies on a (zero-centered) continuous Gaussian sampler, used to sample uniformly in hyperballs. The calls to it can also be massively parallelized. This difference makes HAETAE possible to have a fixed-point signing algorithm and easier masking. Further, in the randomized version of the signature scheme, these samples can be computed offline as they are independent of the message to be signed. The online tasks are far simpler than those of Falcon and Mitaka. Finally, we note that key generation is much simpler for HAETAE than in Falcon and Mitaka.

1.2.2 Limitations

- The key generation algorithm restarts if the secret key does not satisfy the key rejection condition. This makes the key generation algorithm of HAETAE slower than Dilithium’s.
- While HAETAE is simpler from an implementation perspective, its verification key and signature sizes are larger than Falcon’s.

2 Preliminaries

Before introducing specific results adapted to the setting in HAETAE in Section 3 and the HAETAE scheme itself in Section 4, we start by defining notations used throughout this paper and recapitulate relevant fundamental works.

2.1 Notations

Matrices are denoted in bold font and upper case letters (e.g., \mathbf{A}), while vectors are denoted in bold font and lowercase letters (e.g., \mathbf{y} or \mathbf{z}_1). The i -th component of a vector is denoted with subscript i (e.g., y_i for the i -th component of \mathbf{y}).

Every vector is a column vector. We denote concatenation between vectors by putting the rows below as (\mathbf{u}, \mathbf{v}) and the columns on the right as $(\mathbf{u}|\mathbf{v})$. We naturally extend the latter notation to concatenations between matrices and vectors (e.g., $(\mathbf{A}|\mathbf{b})$ or $(\mathbf{A}|\mathbf{B})$).

We let $\lfloor y \rfloor$ be a rounding of $y \in \mathbb{R}$ to the nearest integer. We naturally extend the rounding notation to vectors and polynomials by applying it component-wise.

We let $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ be a polynomial ring where n is a power of 2 integer and for any positive integer q the quotient ring $\mathcal{R}_q = \mathbb{Z}[x]/(q, x^n + 1) = \mathbb{Z}_q[x]/(x^n + 1)$. We abuse notations and identify \mathcal{R}_2 with the set of elements in \mathcal{R} with binary coefficients. We also let $\mathcal{R}_{\mathbb{R}} = \mathbb{R}[x]/(x^n + 1)$ be a polynomial ring over real numbers. For an integer η , we let S_η denote the set of polynomials of degree less than n with coefficients in $[-\eta, \eta] \cap \mathbb{Z}$. Given $\mathbf{y} = (\sum_{0 \leq i < n} y_i x^i, \dots, \sum_{0 \leq i < n} y_{nk-n+i} x^i)^\top \in \mathcal{R}^k$ (or $\mathcal{R}_{\mathbb{R}}^k$), we define its ℓ_2 -norm as the ℓ_2 -norm of the corresponding “flattened” vector $\|\mathbf{y}\|_2 = \|(y_0, \dots, y_{nk-1})^\top\|_2$.

Let $\mathcal{B}_{\mathcal{R},m}(r, \mathbf{c}) = \{\mathbf{x} \in \mathcal{R}_{\mathbb{R}}^m \mid \|\mathbf{x} - \mathbf{c}\|_2 \leq r\}$ denote the continuous hyperball with center $\mathbf{c} \in \mathcal{R}^m$ and radius $r > 0$ in dimension $m > 0$. When $\mathbf{c} = \mathbf{0}$, we omit it. Let $\mathcal{B}_{(1/N)\mathcal{R},m}(r, \mathbf{c}) = (1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r, \mathbf{c})$ denote the discretized hyperball with radius $r > 0$ and center $\mathbf{c} \in \mathcal{R}^m$ in dimension $m > 0$ with respect to a positive integer N . When $\mathbf{c} = \mathbf{0}$, we omit it. Given a measurable set $X \subseteq \mathcal{R}^m$ of finite volume, we let $U(X)$ denote the continuous uniform distribution over X . It admits $\mathbf{x} \mapsto \chi_X(\mathbf{x})/\text{Vol}(X)$ as a probability density, where χ_X is the indicator function of X and $\text{Vol}(X)$ is the volume of the set X . For the normal distribution over \mathbb{R} centered at μ with standard deviation σ , we use the notation $\mathcal{N}(\mu, \sigma)$.

For a positive integer α , we define $r \bmod^\pm \alpha$ as the unique integer r' in the range $[-\alpha/2, \alpha/2)$ satisfying the relation $r = r' \bmod \alpha$. We also define $r \bmod^+ \alpha$ as the unique integer r' in the range $[0, \alpha)$ that satisfies $r = r' \bmod \alpha$. We denote the least significant bit of an integer r with $\text{LSB}(r)$. We naturally extend this to integer polynomials and vectors of integer polynomials, by applying it component-wise.

For a sequence of real numbers a_0, \dots, a_n , we denote the i -th maximum as $\max_j^{i\text{-th}} a_j$.

2.2 Signatures

We briefly recall the formalism of digital signatures.

Definition 1 (Digital Signature). A signature scheme is a tuple of PPT algorithms (KeyGen, Sign, Verify) with the following specifications:

- KeyGen : $1^\lambda \rightarrow (\text{vk}, \text{sk})$ outputs a verification key vk and a signing key sk ;

- $\text{Sign} : (\text{sk}, \mu) \rightarrow \sigma$ takes as inputs a signing key sk and a message μ and outputs a signature σ ;
- $\text{Verify} : (\text{vk}, \mu, \sigma) \rightarrow b \in \{0, 1\}$ is a deterministic algorithm that takes as inputs a verification key vk , a message μ , and a signature σ and outputs a bit $b \in \{0, 1\}$.

Let $\gamma > 0$. We say that it is γ -correct if for any pair (vk, sk) in the range of KeyGen and μ ,

$$\Pr[\text{Verify}(\text{vk}, \mu, \text{Sign}(\text{sk}, \mu)) = 1] \geq \gamma,$$

where the probability is taken over the random coins of the signing algorithm. We say that it is correct in the (Q)ROM if the above holds when the probability is also taken over the randomness of the random oracle modeling the hash function used in the scheme.

We also give two security notions, namely the existential unforgeability under chosen message attacks, and under no-message attacks.

Definition 2 (Security). Let $T, \delta \geq 0$. A signature scheme $\text{sig} = (\text{KeyGen}, \text{Sign}, \text{Verify})$ is said to be (T, δ) -UF-CMA secure in the QROM if for any quantum adversary \mathcal{A} with runtime $\leq T$ given (classical) access to the signing oracle and (quantum) access to a random oracle H , it holds that

$$\Pr_{(\text{vk}, \text{sk})} [\text{Verify}(\text{vk}, \mu^*, \sigma^*) = 1 | (\mu^*, \sigma^*) \leftarrow \mathcal{A}^{H, \text{Sign}}(\text{vk})] \leq \delta,$$

where the randomness is taken over the random coins of \mathcal{A} and $(\text{vk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda)$. The adversary should also not have issued a sign query for μ^* . The above probability of forging a signature is called the advantage of \mathcal{A} and denoted by $\text{Adv}_{\text{sig}}^{\text{UF-CMA}}(\mathcal{A})$. If \mathcal{A} does not output anything, then it automatically fails.

Existential unforgeability against no-message attack, denoted by UF-NMA is defined similarly except that the adversary is not allowed to query any signature per message. Strong existential unforgeability, denoted by sUF-CMA, allows an adversary to query signatures for its target message, as long as it does not output a queried signature.

2.3 Lattice Assumptions

We first recall the well-known lattice assumptions MLWE and MSIS on algebraic lattices.

Definition 3 (Decision-MLWE $_{n,q,k,\ell,\eta}$). For positive integers q, k, ℓ, η and the dimension n of \mathcal{R} , we say that the advantage of an adversary \mathcal{A}_1 solving the decision-MLWE $_{n,q,k,\ell,\eta}$ problem is

$$\text{Adv}_{n,q,k,\ell,\eta}^{\text{MLWE}}(\mathcal{A}_1) = \left| \Pr \left[b = 1 \mid \mathbf{A} \leftarrow \mathcal{R}_q^{k \times \ell}; \mathbf{b} \leftarrow \mathcal{R}_q^k; b \leftarrow \mathcal{A}_1(\mathbf{A}, \mathbf{b}) \right] - \Pr \left[b = 1 \mid \begin{array}{l} \mathbf{A} \leftarrow \mathcal{R}_q^{k \times \ell}; (\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k; \\ b \leftarrow \mathcal{A}_1(\mathbf{A}, \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2) \end{array} \right] \right|.$$

Definition 4 (Search-MSIS $_{n,q,k,\ell,\beta}$). For positive integers q, k, ℓ , a positive real number β and the dimension n of \mathcal{R} , we say that the advantage of an adversary \mathcal{A}_2 solving the search-MSIS $_{n,q,k,\ell,\beta}$ problem is

$$\text{Adv}_{n,q,k,\ell,\beta}^{\text{MSIS}}(\mathcal{A}_2) = \Pr \left[\begin{array}{l} 0 < \|\mathbf{y}\|_2 < \beta \wedge \\ (\mathbf{A} \mid \mathbf{Id}_k) \cdot \mathbf{y} = 0 \pmod{q} \end{array} \mid \mathbf{A} \leftarrow \mathcal{R}_q^{k \times \ell}; \mathbf{y} \in \mathcal{R}_q^{k+\ell} \leftarrow \mathcal{A}_2(\mathbf{A}) \right].$$

Moreover, we finally introduce a variant of the SelfTargetMSIS problem introduced in Dilithium [10], which corresponds to our setting.

Definition 5 (BimodalSelfTargetMSIS _{H,n,q,k,ℓ,β}). Let $H : \{0,1\}^* \times \mathcal{M} \rightarrow \mathcal{R}_2$ be a cryptographic hash function, where $\mathcal{M} \subseteq \{0,1\}^*$ is a message space. Let $q, k, \ell > 0$, $\beta \geq 0$ and the dimension n of \mathcal{R} . An adversary \mathcal{A}_3 solving the search-BimodalSelfTargetMSIS _{H,n,q,k,ℓ,β} problem with respect to $\mathbf{j} \in \mathcal{R}_2^k \setminus \{0\}$ has an advantage of

$$\text{Adv}_{H,n,q,k,\ell,\beta}^{\text{BimodalSelfTargetMSIS}}(\mathcal{A}_3) = \Pr \left[\begin{array}{c} 0 < \|\mathbf{y}\|_2 < \beta \wedge \\ H(\mathbf{A}\mathbf{y} - qc\mathbf{j} \bmod 2q, \mu) = c \\ (\mathbf{A}_0 | \mathbf{b}) \leftarrow \mathcal{R}_q^{k \times \ell}; \\ \mathbf{A} = (2\mathbf{b} + q\mathbf{j} | 2\mathbf{A}_0 | 2\mathbf{Id}_k) \bmod 2q; \\ (\mathbf{y}, c, \mu) \in \mathcal{R}_q^{k+\ell} \times \mathcal{R}_2 \times \mathcal{M} \leftarrow \mathcal{A}_3^{H(\cdot)}(\mathbf{A}) \end{array} \right].$$

In the ROM (resp. QROM), the adversary is given classical (resp. quantum) access to H .

The following classical reduction from MSIS to BimodalSelfTargetMSIS is very similar to the reduction from MSIS to SelfTargetMSIS introduced in [10] and is similarly non-tight. As this latter reduction, it cannot be straightforwardly extended to a reduction in the QROM, since it relies on the forking lemma.

Theorem 6 (Classical Reduction from MSIS to BimodalSelfTargetMSIS). *Let $q > 0$ be an odd modulus, $H : \{0,1\}^* \times \mathcal{M} \rightarrow \mathcal{R}_2$ be a cryptographic hash function modeled as a random oracle and that every polynomial-time classical algorithm has a negligible advantage against MSIS _{n,q,k,ℓ,β} . Then every polynomial-time classical algorithm has negligible advantage against BimodalSelfTargetMSIS _{$n,q,k,\ell,\beta/2$} .*

Proof sketch. Consider a BimodalSelfTargetMSIS _{$n,q,k,\ell,\beta/2$} classical algorithm \mathcal{A} that is polynomial-time and has classical access to H . If $\mathcal{A}^{H(\cdot)}(\mathbf{A})$ makes Q hash queries $H(\mathbf{w}_i, \mu_i)$ for $i = 1, \dots, Q$ and outputs a solution (\mathbf{y}, c, μ_j) for some $j \in [Q]$, then we can construct an adversary \mathcal{A}' for MSIS _{n,q,k,ℓ,β} as follows.

The adversary \mathcal{A}' can first rewind \mathcal{A} to the point at which the j -th query was made and reprogram the hash as $H(\mathbf{w}_j, \mu_j) = c' (\neq c)$. Then, with probability approximately $1/Q$, algorithm \mathcal{A} will produce another solution (\mathbf{y}', c', μ_j) . We then have

$$\begin{cases} \mathbf{A}\mathbf{y} - qc\mathbf{j} = \mathbf{z}_j = \mathbf{A}\mathbf{y}' - qc'\mathbf{j} \bmod 2q, \\ \|\mathbf{y}\|_2, \|\mathbf{y}'\|_2 < \beta/2. \end{cases}$$

As q is odd, we have $\mathbf{A}(\mathbf{y} - \mathbf{y}') = (c - c')\mathbf{j} \bmod 2$. The fact that $c' \neq c$ implies that the latter is non-zero modulo 2, and hence so is $\mathbf{y} - \mathbf{y}'$ over the integers. As it also satisfies $(\mathbf{b} | \mathbf{A}_0 | \mathbf{Id}_k) \cdot (\mathbf{y} - \mathbf{y}') = 0 \bmod q$ and $\|\mathbf{y} - \mathbf{y}'\| < \beta$, it provides a MSIS _{n,q,k,ℓ,β} solution for the matrix $(\mathbf{b} | \mathbf{A}_0 | \mathbf{Id}_k)$, where the submatrix $(-\mathbf{b} | \mathbf{A}_0) \in \mathcal{R}_q^{k \times \ell}$ is uniform. \square

2.4 Sampling from the Continuous Hyperball-uniform

In order to sample in practice from hyperball uniform, we rely on the following result.

$\mathbf{y} \leftarrow U(\mathcal{B}_{\mathcal{R},k}(r)):$ 1: $y_i \leftarrow \mathcal{N}(0, 1)$ for $i = 0, \dots, nk + 1$ 2: $L \leftarrow \ (y_0, \dots, y_{nk+1})^\top\ _2$ 3: $\mathbf{y} \leftarrow r/L \cdot (\sum_{i=0}^{n-1} y_i x^i, \dots, \sum_{i=nk-n}^{nk-1} y_i x^i)^\top$ 4: return \mathbf{y}	$\triangleright \mathbf{y} \in \mathcal{R}_{\mathbb{R}}^k$
---	--

Figure 1: Hyperball uniform sampling

Lemma 7 ([23]). *The distribution of the output of the algorithm in Figure 1 is $U(\mathcal{B}_{\mathcal{R},k}(r))$.*

Sampling from continuous hyperball-uniform can be done using the algorithm in Figure 1 due to Lemma 7. However, to allow side-channel secure implementations of HAETAE, we sample from discrete hyperball-uniform. We delay to Section 3.2 the analysis of a discretized version which turns discrete Gaussian samples to discrete hyperball-uniform distribution.

2.5 Signature Encoding via Range Asymmetric Numeral System

A HAETAE signature is essentially a vector \mathbf{z} , that is compressed into \mathbf{z}_2 with smaller dimension and a hint \mathbf{h} , that are then encoded. While Huffman coding would be applied on each coordinate at a time, an arithmetic coding encodes the entire vector coordinates in a single number. In contrast to Huffman coding, arithmetic coding gets close to entropy also for alphabets, where the probabilities of the symbols are not powers of two. We recall a recent type of entropy coding, named range Asymmetric Numeral systems (rANS) [11], that encodes the state in a natural number and thus allows faster implementations. The rANS encoding technique was recently used in [15] and we adapt it to hyperball uniform distributions. As a stream variant, rANS can be implemented with finite precision integer arithmetic by using renormalization.

Definition 8 (Range Asymmetric Numeral System (rANS) Coding). Let $t > 0$ and $S \subseteq [0, 2^t - 1]$. Let $g : [0, 2^t - 1] \rightarrow \mathbb{Z} \cap (0, 2^t]$ such that $\sum_{x \in S} g(x) \leq 2^t$ and $g(x) = 0$ for all $x \notin S$. We define the following:

- $\text{CDF} : S \rightarrow \mathbb{Z}$, defined as $\text{CDF}(s) = \sum_{y=0}^{s-1} g(y)$.
- $\text{symbol} : \mathbb{Z} \rightarrow S$, where $\text{symbol}(y)$ is defined as $s \in S$ satisfying $\text{CDF}(s) \leq y < \text{CDF}(s + 1)$.
- $C : \mathbb{Z} \times S \rightarrow \mathbb{Z}$, defined as

$$C(x, s) = \left\lfloor \frac{x}{g(s)} \right\rfloor \cdot 2^t + (x \bmod^+ g(s)) + \text{CDF}(s).$$

Then, we define the rANS encoding/decoding for the set S and frequency $g/2^t$ as in Figure 2.

Lemma 9 (Adapted from [11]). *The rANS coding is correct, and the size of the rANS code is asymptotically equal to Shannon entropy of the symbols. That is, for any choice*

Encode($(s_1, \dots, s_m) \in S^m$):

```

1:  $x_0 = 0$ 
2: for  $i = 0, \dots, m - 1$  do
3:    $x_{i+1} = C(x_i, s_{i+1})$ 
4: return  $x_m$ 

```

Decode($x \in \mathbb{Z}$):

```

1:  $y_0 = x$ 
2:  $i = 0$ 
3: while  $y_i > 0$  do
4:    $s'_{i+1} = \text{symbol}(y_i \bmod^+ 2^t)$ 
5:    $y_{i+1} = \lfloor y_i / 2^t \rfloor \cdot g(s'_{i+1}) + (y_i \bmod^+ 2^t) - \text{CDF}(s'_{i+1})$ 
6:    $i++$ 
7:  $m = i - 1$ 
8: return  $(s'_m, \dots, s'_1) \in S^m$ 

```

Figure 2: rANS encoding and decoding procedures

of $\mathbf{s} = (s_1, \dots, s_m) \in S^m$, $\text{Decode}(\text{Encode}(\mathbf{s})) = \mathbf{s}$. Moreover, for any positive x and any probability distribution p over S , it holds that

$$\sum_{s \in S} p(s) \log_2(C(x, s)) \leq \log_2(x) + \sum_{s \in S} p(s) \log_2 \left(\frac{g(s)}{2^t} \right) + \frac{2^t}{x}.$$

Finally, the cost in bitlength of encoding the first symbol is $\leq t$, i.e., for any $s \in S$, we have $\log_2(C(0, s)) \leq t$.

We determine the frequency of the symbols experimentally, by executing the signature computation and collecting several million samples. Finally, we apply some rounding strategy in order to heuristically minimize the empirical entropy $\sum_{s \in S} p(s) \log(g(s)/2^n)$.

3 HAETAE-specific Results

While our scheme is reminiscent of Dilithium, the bimodal setting hinders the use of some of its base components. In this section, we describe parts that are specifically adapted to HAETAE. First, the key generation algorithm departs from known key generation algorithms for BLISS, as we work in the module setting. Second, we study the precision needed when discretizing the hyperball sampler from Section 2.4 to enable fixed-point arithmetic. Then, we explain how challenges are computed in HAETAE. Next, we describe the rejection sampling procedure and estimate its expected number of iterations depending on the fixed-point arithmetic precision. Finally, we explain how to split the coordinates of a signature vector into high and low bits, allowing for signature compression via low bits drop. This order is consistent with the order in which those results are used during signing.

3.1 Key Generation

When using bimodal rejection sampling, the verification step relies on a specific key pair $(\mathbf{A}, \mathbf{s}) \in \mathcal{R}_p^{k \times (k+\ell)} \times \mathcal{R}_p^{k+\ell}$ such that the bimodal centers $(-1)^b \mathbf{A} \mathbf{s} \bmod p$ ($b = 0, 1$) are the same, regardless of the bit b . To generate such a pair, following [9], we choose $p = 2q$ and aim at $\mathbf{A} \mathbf{s} = q\mathbf{j} \bmod 2q$ for $\mathbf{j} = (1, 0, \dots, 0)^\top$.

3.1.1 Key Generation and Encoding

To build such a key pair (\mathbf{A}, \mathbf{s}) , we do as follows. We first generate an MLWE sample $\mathbf{b} = \mathbf{A}_{\text{gen}} \mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}} \bmod q$, where $\mathbf{A}_{\text{gen}} \leftarrow U(\mathcal{R}_q^{k \times (\ell-1)})$ and $(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \leftarrow U(S_\eta^{\ell-1} \times S_\eta^k)$. We then define $\mathbf{A} = (-2\mathbf{b} + q\mathbf{j} \mid 2\mathbf{A}_{\text{gen}} \mid 2\mathbf{I}_k) \bmod 2q$ as well as $\mathbf{s}^\top = (1 \mid \mathbf{s}_{\text{gen}}^\top \mid \mathbf{e}_{\text{gen}}^\top)$. This is a valid verification key pair for HAETAE, but the choice of even modulus $2q$ makes it hard to truncate the least significant bits of \mathbf{b} as in Dilithium.

To enable the verification key truncation, we modify the key generation algorithm, as follows. We use an extra randomness $\mathbf{a}_{\text{gen}} \leftarrow U(\mathcal{R}_q^k)$ and let $\mathbf{b} - \mathbf{a}_{\text{gen}} = \mathbf{A}_{\text{gen}} \mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}} \bmod q$. For any decomposition $\mathbf{b} = \mathbf{b}_1 + \mathbf{b}_0$, we then define $\mathbf{A} = (2(\mathbf{a}_{\text{gen}} - \mathbf{b}_1) + q\mathbf{j} \mid 2\mathbf{A}_{\text{gen}} \mid 2\mathbf{I}_k)$ as well as $\mathbf{s}^\top = (1 \mid \mathbf{s}_{\text{gen}}^\top \mid (\mathbf{e}_{\text{gen}} - \mathbf{b}_0)^\top)$. One sees that $\mathbf{A} \mathbf{s} = q\mathbf{j} \bmod 2q$. In practice, the verification key is then comprised of \mathbf{b}_1 and the seed that allows generating \mathbf{A}_{gen} and \mathbf{a}_{gen} . The secret key is the seed used to generate \mathbf{s} and $(\mathbf{A}_{\text{gen}}, \mathbf{a}_{\text{gen}})$.

It remains to choose the decomposition of \mathbf{b} , that we see as an nk -dimensional vector with coordinates in $[0, q-1]$. We set the coordinates of \mathbf{b}_1 as follows. If some coordinate of \mathbf{b} is even, then we take the same value for the corresponding coordinate of \mathbf{b}_1 . Else, we take the rounding of this coordinate to the nearest multiple of 4 as value for \mathbf{b}_1 . Next we set $\mathbf{b}_0 = \mathbf{b} - \mathbf{b}_1$ and we note that coordinates of \mathbf{b}_0 lie in $[-1, 1]$, i.e., $\mathbf{b}_0 \in S_1^k$. We can then write $\mathbf{b} = \mathbf{b}_0 + 2\mathbf{b}'_1$, where \mathbf{b}'_1 is encoded using $\lceil \log_2(q) - 1 \rceil$ bits per coordinate, i.e. one less bit than \mathbf{b} . This is computed coordinate-wise with $\mathbf{b}_0 = (-1)^{\lfloor \mathbf{b}/2 \rfloor \bmod 2} \mathbf{b} \bmod 2$. In all of the following, we let $(\text{LowBits}^{\text{vk}}(\mathbf{b}), \text{HighBits}^{\text{vk}}(\mathbf{b}))$ denote $(\mathbf{b}_0, \mathbf{b}_1)$.

When \mathbf{b} is uniform, we notice that the coordinates of \mathbf{b}_0 roughly follow a (centered) binomial law with parameters $(2, 1/2)$, which experimentally leads to smaller choices for γ , which we discuss and introduce below.

Note that the truncation reduces each coefficient of \mathbf{b} by 1 bit. So the verification key becomes shorter, but not significantly. Thus, we use the truncation for lower security

levels and keep the no-truncation version for the highest level. In the following, we refer to the truncated version as $d = 1$ and the non-truncated version as $d = 0$, where d is the vk truncation bit.

3.1.2 Rejection Sampling on the Key

A critical step of our scheme is bounding $\|cs\|_2$, where \mathbf{s} is generated as before and $c \in \mathcal{R}$ is a polynomial with coefficients in $\{0, 1\}$ and has less than or equal to τ nonzero coefficients. The lower this bound is, the smaller the signature is, which in turn leads to the harder forging. In the key generation algorithm, we apply the following rejection condition for some heuristic value γ , bounding $\|cs\|_2 \leq \gamma\sqrt{\tau}$:

$$\mathcal{N}(\mathbf{s}) := \tau \cdot \sum_{i=1}^m \max_{0 \leq j < 2n}^{i\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2 + r \cdot \max_{0 \leq j < 2n}^{(m+1)\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2 \leq \gamma^2 n,$$

where $m = \lfloor n/\tau \rfloor$, $r = n \bmod \tau$, ω_j 's are the primitive $2n$ -th roots of unity ($1 \leq j \leq n$). Note that $\mathbf{s}(\omega_j)$ is defined as $(s_1(\omega_j), \dots, s_{k+\ell}(\omega_j)) \in \mathbb{C}^{k+\ell}$ given the secret key $\mathbf{s} = (s_1, \dots, s_{k+\ell}) \in \mathcal{R}^{k+\ell}$. Below, we prove that the left hand side is a bound on $\frac{n}{\tau} \cdot \|cs\|_2^2$ and that this condition leads to asserting $\|cs\|_2 \leq \gamma\sqrt{\tau}$.

Lemma 10. *For any challenge $c \in \{0, 1\}^n$ with Hamming weight τ and a secret $\mathbf{s} \in S_\eta^{k+\ell}$, the value $\|cs\|_2^2$ is upper bounded by*

$$\frac{\tau}{n} \left(\tau \cdot \sum_{i=1}^m \max_{0 \leq j < 2n}^{i\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2 + r \cdot \max_{0 \leq j < 2n}^{(m+1)\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2 \right),$$

where $m = \lfloor n/\tau \rfloor$, $r = n \bmod \tau$, and ω_j 's are the primitive $2n$ -th roots of unity.

Proof. We first rewrite $\|cs\|_2^2$ as:

$$\|cs\|_2^2 = \frac{\sum_j |c(\omega_j)|^2 \cdot \|\mathbf{s}(\omega_j)\|_2^2}{n},$$

where $\mathbf{s}(\omega_j) = (s_1(\omega_j), \dots, s_{k+\ell}(\omega_j))$. We have that $\sum_{j=1}^n |c(\omega_j)|^2 = n\tau$ and $|c(\omega_j)|^2 = |\omega_{j,1} + \dots + \omega_{j,\tau}|^2 \leq \tau^2$. We can bound $\sum_{j=1}^n |c(\omega_j)|^2 \cdot \|\mathbf{s}(\omega_j)\|_2^2$ by rearranging the order. Let $m = \lfloor n/\tau \rfloor$ and $r = n \bmod \tau$. Then m is the maximum number of $|c(\omega_j)|^2$'s that can be τ^2 . By sorting $\|\mathbf{s}(\omega_j)\|_2$ in a decreasing order,

$$\|\mathbf{s}(\omega_{\sigma(1)})\|_2 \geq \|\mathbf{s}(\omega_{\sigma(2)})\|_2 \geq \dots \geq \|\mathbf{s}(\omega_{\sigma(n)})\|_2,$$

where σ is a permutation for the indices, we have

$$\sum_{j=1}^n |c(\omega_j)|^2 \cdot \|\mathbf{s}(\omega_j)\|_2^2 \leq \sum_{j=1}^m |c(\omega_{\sigma(j)})|^2 \cdot \|\mathbf{s}(\omega_{\sigma(j)})\|_2^2 + \sum_{j=m+1}^n |c(\omega_{\sigma(j)})|^2 \cdot \|\mathbf{s}(\omega_{\sigma(m+1)})\|_2^2.$$

Then it reaches the maximum when the m largest $\|\mathbf{s}(\omega_j)\|_2^2$'s are multiplied with τ^2 's, i.e.,

$$\begin{aligned} \sum_{j=1}^n |c(\omega_j)|^2 \cdot \|\mathbf{s}(\omega_j)\|_2^2 &\leq \sum_{j=1}^m \tau^2 \cdot \|\mathbf{s}(\omega_{\sigma(j)})\|_2^2 + \left(\sum_{j=1}^n |c(\omega_j)|^2 - m\tau^2 \right) \cdot \|\mathbf{s}(\omega_{\sigma(m+1)})\|_2^2 \\ &= \tau^2 \cdot \sum_{j=1}^m \|\mathbf{s}(\omega_{\sigma(j)})\|_2^2 + r \cdot \tau \cdot \|\mathbf{s}(\omega_{\sigma(m+1)})\|_2^2. \end{aligned}$$

This concludes the proof. \square

3.2 Sampling in a Discrete Hyperball

In order to generate a hyperball uniform sample \mathbf{y} , we apply a rounding-and-reject strategy to the discretization of the continuous hyperball uniform sampling from Figure 1, which allows to generate rightly distributed samples. Our approach in sampling is to avoid the use of floating point arithmetic for two reasons: First, many microarchitectures do not provide floating-point units and even if so, the execution time of floating-point instructions may be data-dependent and thus unsuitable [1] for a constant-time implementation. Floating-point computation would also prohibit a masked implementation, that is protected against power side-channel attacks, because known masking techniques are only applicable to integers. And second, the required precision is higher than achievable even in IEEE double. In order to do so, we replace the continuous Gaussian sampler from Lemma 7 and instead use discrete Gaussian distributions, as we know that they approximate well continuous Gaussian distribution for large standard deviation.

Discretizing the Output. Once we obtain an “hyperball” sample, we choose to round it. Then, if the resulting sample lies too close to the border of the hyperball, we reject it. This ensures that for any possible sample, they have the same amount of pre-rounding predecessors. This also decreases the precision but the output is now discrete in a hyperball with a somewhat-smaller radius. We simply increase the starting radius to compensate.

```

 $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},m}(B))$ :
1:  $\mathbf{y} \leftarrow U(\mathcal{B}_{\mathcal{R},m}(NB + \sqrt{mn}/2))$    $\triangleright$  continuous sampling in Figure 1
2: if  $\|\lfloor \mathbf{y} \rfloor\|_2 \leq NB$  then
3:   return  $\lfloor \mathbf{y} \rfloor / N$ 
4: else, restart  $\triangleright \mathbf{y} \in \mathcal{B}_{(1/N)\mathcal{R},m}(B) \subset (1/N)\mathcal{R}^m$ 

```

Figure 3: Discrete hyperball uniform sampling

We study in the following lemma the rejection probability of this step.

Lemma 11. *Let n be the degree of \mathcal{R} , $M_0 \geq 1$, $B, m, N > 0$. At each iteration, the algorithm from Figure 3 succeeds with probability $\geq 1/M_0$ and the distribution of the output is $U(\mathcal{B}_{(1/N)\mathcal{R},m}(B))$ if we set*

$$N \geq \frac{\sqrt{mn}}{2B} \cdot \frac{M_0^{1/(mn)} + 1}{M_0^{1/(mn)} - 1}.$$

The proof of this lemma can be found in Appendix B.

3.3 Challenge Sampling

Challenges in HAETAE are polynomials $c \in \mathcal{R}$ with binary coefficients and exactly τ of them are nonzero. As simply hashing the message and the commitment only yields a binary

```

SampleBinaryChallenge $\tau$ ( $\rho$ )
// for HAETAE-2 or HAETAE-3
1: Initialize  $\mathbf{c} = c_0c_1 \dots c_{255} = 00 \dots 0$ 
2: for  $i = 256 - \tau$  to 255 do
3:    $j \leftarrow \{0, \dots, i\}$ 
4:    $c_i = c_j$ 
5:    $c_j = 1$ 
6: Return  $\mathbf{c}$ 
// for HAETAE-5
1: Initialize  $\mathbf{c} = c_0c_1 \dots c_{255} = H(\rho)$ 
2: if  $\text{wt}(c) > 128$  then
3:    $c = c \otimes 11 \dots 1$ 
4: else if  $\text{wt}(c) = 128$  then
5:    $c = c \otimes c_0c_0 \dots c_0$ 
6: Return  $\mathbf{c}$ 

```

Figure 4: Challenge sampling algorithm

string ρ , we now explain how to format it to get such a challenge. Since $n = 256$ across all three parameter sets, the challenge space has size $\binom{n}{\tau}$ exceeding the required entropy 2^{192} and 2^{225} for HAETAE-2 and HAETAE-3, respectively. To sample such challenges we rely on the (binary version of) `SampleInBall` algorithm from Dilithium, which we specify in the first half of Figure 4.

For HAETAE-5, however, we require 255 bits of entropy for the challenge space, which cannot be reached with the fixed Hamming weights for $n = 256$. To achieve it, we replace the challenge space by a set containing exactly half of the bitstrings of length 256. Specifically, we choose a set containing all elements of Hamming weight strictly less than 128 and half of the elements of Hamming weight 128, using the following algorithm. Given a 256-bits hash with Hamming weight w , do the following. If $w < 128$, we do nothing, and if $w > 128$, we flip all the bits. If $w = 128$, we decide whether to flip or not, depending on the first bit. Exactly half of all binary polynomials are reachable this way, which means that the challenge set has size 2^{255} as desired. The algorithm is specified in the second half of Figure 4.

As a side note, this means that the hash function with which we instantiate the Fiat-Shamir transform is the composition of these two steps, hashing and formatting. Looking ahead, this corresponds to steps 6 and 7 of Figure 8. Contrary to Dilithium, we do not stray away from the Fiat-Shamir transform and include the challenge c in the signature as it is no bigger than ρ when encoded.

3.4 Bimodal Hyperball Rejection Sampling

Recently, Devevey et al. [8] conducted a study of rejection sampling in the context of lattice-based Fiat-Shamir with Aborts signatures. They observe that (continuous) uniform distributions over hyperballs can be used to obtain compact signatures, with a relatively simple rejection procedure. To make masking easier, HAETAE uses (discretized) uniform

distributions over hyperballs, in the bimodal context. The proof of the following lemma is available in Appendix B.

Lemma 12 (Bimodal Hyperball Rejection Sampling). *Let n be the degree of \mathcal{R} , $c > 1$, $r, t, m > 0$, and $B \geq \sqrt{B'^2 + t^2}$. Define $M = 2(B/B')^{mn}$ and set*

$$N \geq \frac{1}{c^{1/(mn)} - 1} \frac{\sqrt{mn}}{2} \left(\frac{c^{1/(mn)}}{B'} + \frac{1}{B} \right).$$

Let $\mathbf{v} \in \mathcal{R}^m \cap \mathcal{B}_{(1/N)\mathcal{R},m}(t)$. Let $p : \mathbb{R}^m \rightarrow \{0, 1/2, 1\}$ be defined as follows

$$p(\mathbf{z}) = \begin{cases} 0 & \text{if } \|\mathbf{z}\| \geq B', \\ 1/2 & \text{else if } \|\mathbf{z} - \mathbf{v}\| < B \wedge \|\mathbf{z} + \mathbf{v}\| < B, \\ 1 & \text{otherwise.} \end{cases}$$

Then there exists $M' \leq cM$ such that the output distributions of the two algorithms from Figure 6 are identical.

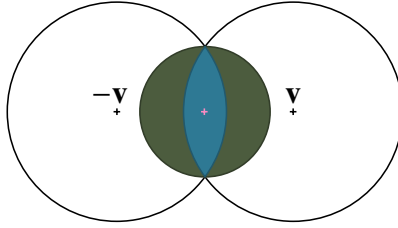


Figure 5: The HAETAE eyes

Figure 5 illustrates (the continuous version) of the rejection sampling that we consider. The black empty circles have radii equal to B and the green circle has radius B' . We sample a vector \mathbf{z} uniformly inside one of the black circles (with probability $1/2$ for each) and keep \mathbf{z} with $p(\mathbf{z}) = 1/2$ if \mathbf{z} lies in the blue zone, with probability $p(\mathbf{z}) = 1$ if it lies in the green zone, and with probability $p(\mathbf{z}) = 0$ everywhere else.

We now have all necessary ingredients in Figures 1, 3, 5, and 6 to make sure the resulting distribution of \mathbf{z} is indeed uniform over the discretized hyperball. Thanks to Lemma 11 and Lemma 12, we already know the level of precision required for \mathbf{y} to maintain the provable security of HAETAE.

$\mathcal{A}(\mathbf{v}) :$	$\mathcal{B} :$
1: $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},m}(B))$	1: $\mathbf{z} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},m}(B'))$
2: $\mathbf{b} \leftarrow U(\{0, 1\})$	2: return \mathbf{z} with probability $1/M'$, else \perp
3: $\mathbf{z} \leftarrow \mathbf{y} + (-1)^{\mathbf{b}}\mathbf{v}$	
4: return \mathbf{z} with probability $p(\mathbf{z})$, else \perp	

Figure 6: Bimodal hyperball rejection sampling

3.5 High and Low Bits

Recall that a HAETAE signature is principally a vector \mathbf{z} , whose lower part is replaced with a (smaller) hint. HAETAE makes use of two different high and low bits decompositions: one helps encoding a signature while the other is used when computing a hint. Following [15], the first is helpful in the sense that if we correctly choose the number of low bits, they will be distributed almost uniformly and can then be excluded from the encoding step. The high bits on the other hand, will then follow a distribution with a very small variance and we apply the rANS encoding on them only, making it much more efficient as the size of the alphabet greatly shrunk.

The second decomposition allows to reduce the alphabet size of the resulting hint, and thus to reduce the size of its encoding.

We use the following base method of decomposing an element in high and low bits. We first recall the Euclidean division with a centered remainder.

Lemma 13. *Let $a \geq 0$ and $b > 0$. It holds that*

$$a = \left\lfloor \frac{a + b/2}{b} \right\rfloor \cdot b + (a \bmod^\pm b),$$

and this writing as $a = bq + r$ with $r \in [-b/2, b/2)$ is unique.

We define our decomposition for compressing the upper part of the signature.

Definition 14 (High and low bits). Let $r \in \mathbb{Z}$ and α be a power of two integer. Define $r_1 = \lfloor (r + \alpha/2)/\alpha \rfloor$ and $r_0 = r \bmod^\pm \alpha$. Finally, define the tuple:

$$(\text{LowBits}(r, \alpha), \text{HighBits}(r, \alpha)) = (r_0, r_1).$$

We extend these definitions to vectors by applying them component-wise. We state that this decomposition lets us recover the original element and bound the components of the decomposition in Lemma 15. The proof is available in Appendix B.

Lemma 15. *Let α be a power of two. Let $q > 2$ be a prime with $\alpha \mid 2(q - 1)$ and $r \in \mathbb{Z}$. Then it holds that*

$$\begin{aligned} r &= \alpha \cdot \text{HighBits}(r, \alpha) + \text{LowBits}(r, \alpha), \\ \text{LowBits}(r, \alpha) &\in [-\alpha/2, \alpha/2), \\ r \in [0, 2q - 1] &\implies \text{HighBits}(r, \alpha) \in [0, (2q - 1)/\alpha]. \end{aligned}$$

We define $\text{HighBits}^{z1}(r) = \text{HighBits}(r, 256)$ and $\text{LowBits}^{z1}(r) = \text{LowBits}(r, 256)$.

3.5.1 High and Low Bits for h

In order to produce the hint that we send instead of the lower part of \mathbf{z} , we could use the previous bit decomposition. However, as noted in [10, Appendix B] in a preliminary version, a slight modification allows to further reduce the entropy of the hint.

The idea is to pack the high bits in the range $[0, 2(q - 1)/\alpha_h)$. This is possible if we use the range $[-\alpha_h/2 - 2, 0)$ to represent the integers that are close to $2q - 1$.

Definition 16 (High and low bits for h). Let $r \in \mathbb{Z}$. Let q be a prime and $\alpha_h | 2(q-1)$ be a power of two. Let $m = 2(q-1)/\alpha_h$, $r_1 = \text{HighBits}(r \bmod^+ 2q, \alpha_h)$, and $r_0 = \text{LowBits}(r \bmod^+ 2q, \alpha_h)$. If $r_1 = m$, let $(r'_0, r'_1) = (r_0 - 2, 0)$. Else, $(r'_0, r'_1) = (r_0, r_1)$. We define:

$$(\text{LowBits}^h(r), \text{HighBits}^h(r)) = (r'_0, r'_1).$$

As before, we extend these definitions to vectors by applying them component-wise. We state that this decomposition lets us recover the original element and bound the decomposition components.

Lemma 17. *Let $r \in \mathbb{Z}$. Let q be a prime, $\alpha_h | 2(q-1)$ be a power of two and define $m = 2(q-1)/\alpha_h$. It holds that*

$$\begin{aligned} r &= \alpha_h \cdot \text{HighBits}^h(r) + \text{LowBits}^h(r) \pmod{2q}, \\ \text{LowBits}^h(r) &\in [-\alpha/2 - 2, \alpha/2), \\ \text{HighBits}^h(r) &\in [0, m - 1]. \end{aligned}$$

The proof of Lemma 17 is available in Appendix B.

4 The HAETAE Signature Scheme

In this section, we describe three different versions of HAETAE. As a warm-up, we give an uncompressed, un-truncated version of HAETAE, implementing the Fiat-Shamir with aborts paradigm in the bimodal hyperball-uniform setting. We then give the full description of optimized and deterministic HAETAE as we implemented it. Finally, we discuss the parts of the signing algorithm which can be pre-computed.

4.1 Uncompressed Description

As a first approach, we give a high-level, uncompressed, description of our signature scheme in Figure 7. In all of the following sections, we let $\mathbf{j} = (1, 0, \dots, 0) \in \mathcal{R}^k$, as well as k, ℓ be two dimensions, $N > 0$ the fix-point precision and $\tau > 0$ the challenge min-entropy parameter. The parameters $B, B',$ and B'' refer to the radii of hyperballs. Let q be an odd prime and $\alpha_n | 2(q-1)$ is a power of two. We recall the key rejection function based on Lemma 10:

$$\mathcal{N} : \mathbf{s} \mapsto \tau \cdot \sum_{i=1}^m \max_j^{i\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2 + r \cdot \max_j^{(m+1)\text{-th}} \|\mathbf{s}(\omega_j)\|_2^2.$$

With the parameter γ , we bound $\mathcal{N}(\mathbf{s}) \leq \gamma^2 n$, which ensures that $\|\mathbf{c}\mathbf{s}\|_2 \leq \gamma\sqrt{\tau}$ for all $c \in \mathcal{R}_2$ satisfying $\text{wt}(c) \leq \tau$. The key generation algorithm is a simplified version from Section 3.1, which removes the verification key truncation, for conceptual simplicity.

Effectively, this requires computing a complex (fix-point) 512-point FFT for each polynomial in \mathbf{s} , where the input is padded with zeros. The absolute value of the FFT outputs is accumulated point-wise, which yields the vector that is the input to the above function f . In fact, this can be optimized this by replacing the 512-point FFT by a 256-point one, where the j -th input coefficient x_j is multiplied by $e^{-\frac{i\pi}{256}j}$, which is functionally equivalent.

4.2 Specification of HAETAE

We now give the full description of the signature scheme HAETAE in Figure 8 with the following building blocks:

- Hash function H_{gen} for generating the seeds and hashing the messages,
- Hash function H for signing, returning a seed ρ for sampling a challenge,
- Extendable output function expandA for deriving \mathbf{a}_{gen} and \mathbf{A}_{gen} from $\text{seed}_{\mathbf{A}}$,
- Extendable output function expandS for deriving $(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \in S_{\eta}^{\ell-1} \times S_{\eta}^k$ from seed_{sk} and $\text{counter}_{\text{sk}}$,
- Extendable output function expandYbb for deriving \mathbf{y}, b and b' from seed_{ybb} and counter ,

KeyGen(1^λ):

- 1: $(\mathbf{A}_{\text{gen}}) \leftarrow \mathcal{R}_q^{k \times (\ell-1)}$ and $(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \leftarrow S_\eta^{\ell-1} \times S_\eta^k$
- 2: $\mathbf{b} = \mathbf{A}_{\text{gen}} \cdot \mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}} \in \mathcal{R}_q^k$
- 3: $\mathbf{A} = (-2\mathbf{b} + q\mathbf{j} \mid 2\mathbf{A}_{\text{gen}} \mid 2\mathbf{Id}_k) \bmod 2q$
- 4: $\mathbf{s} = (1, \mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}})$
- 5: **if** $\mathcal{N}(\mathbf{s}) > \gamma^2 n$ **then** restart
- 6: **return** $\text{sk} = (\mathbf{A}, \mathbf{s})$, $\text{vk} = \mathbf{A}$

Sign(sk, M):

- 1: $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R}, (k+\ell)}(B))$
- 2: $\mathbf{w} \leftarrow \mathbf{A}[\mathbf{y}]$
- 3: $c = H(\mathbf{w}, M) \in \mathcal{R}_2$
- 4: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c \mathbf{s}$ for $b \leftarrow U(\{0, 1\})$
- 5: **if** $\|\mathbf{z}\|_2 \geq B'$ **then** restart
- 6: **else if** $\|2\mathbf{z} - \mathbf{y}\|_2 < B$ **then** restart with probability $1/2$
- 7: **return** $\sigma = (\lfloor \mathbf{z} \rfloor, c)$

Verify($\text{vk}, M, \sigma = (\mathbf{z}, c)$):

- 1: $\tilde{\mathbf{w}} = \mathbf{A}\mathbf{z} - qc\mathbf{j} \bmod 2q$
- 2: **return** $(c = H(\tilde{\mathbf{w}}, M)) \wedge \left(\|\mathbf{z}\| < B + \frac{\sqrt{n(k+\ell)}}{2} \right)$

Figure 7: Uncompressed description of HAETAE .

The above building blocks can be implemented with symmetric primitives.

Note that at Step 6 of the Verify algorithm, the division by 2 is well-defined as the operand is even.

Random Signatures, Randomness Generation, and Seed. Note that, in Figure 8, the key generation uses some randomness for sampling **seed**. Thus, for implementation validation purposes, we can separate the algorithms into internal and external algorithms, respectively. A 256-bit seed can be input to the internal algorithm for KeyGen, which will be **seed** itself.

On the other hand, the signing process uses no further randomness and is strictly deterministic with respect to sk and M . However, for extended use cases, we can easily add some randomness by using an extra seed when generating the 512-bit seed seed_{ybb} , i.e., $\text{seed}_{ybb} = H(K, \text{seed}_{\text{sign}}, \mu)$ for a 256-bit seed $\text{seed}_{\text{sign}}$ for signing. Thus, a 256-bit seed $\text{seed}_{\text{sign}}$ can be an additional input to the internal Sign algorithm.

The external algorithms then need to wrap the internal algorithms and should securely sample the seeds.

4.3 Theoretical Analysis

In this section, we prove the theoretical correctness and security of HAETAE.

4.3.1 Correctness and Runtime

Correctness. Before showing the correctness of the HAETAE scheme, we prove the following intermediary result.

Lemma 18. *We borrow the notations from Figure 8. If we run $\text{Verify}(\text{vk}, M, \sigma)$ on the signature σ returned by $\text{Sign}(\text{sk}, M)$ for an arbitrary message M and an arbitrary key-pair (sk, vk) returned by $\text{KeyGen}(1^\lambda)$, then the following relations hold:*

- 1) $\mathbf{w}_1 = \text{HighBits}^h(\mathbf{w})$,
- 2) $w' \mathbf{j} = \text{LSB}(\lfloor y_0 \rfloor) \cdot \mathbf{j} = \text{LSB}(\mathbf{w}) = \text{LSB}(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rfloor)$,
- 3) $2\lfloor \mathbf{z}_2 \rfloor - 2\tilde{\mathbf{z}}_2 = \text{LowBits}^h(\mathbf{w}) - \text{LSB}(\mathbf{w})$ assuming $B' + \alpha_h/4 + 1 \leq B'' < q/2$.

Proof. Let $m = 2(q - 1)/\alpha_h$. Let us prove the first statement. By definition of \mathbf{h} , it holds that $\mathbf{w}_1 = \text{HighBits}^h(\mathbf{w}) \bmod m$. However, the latter part of the equality already lies in $[0, m - 1]$ by Lemma 17. The first part lies in the same range as we reduce $\bmod^+ m$. Hence, the equality stands over \mathbb{Z} too.

We move on to the second statement. By considering only the first component of $\mathbf{z} = \mathbf{y} + (-1)^b c \mathbf{s}$, we obtain, modulo 2:

$$\tilde{z}_0 = \lfloor z_0 \rfloor = \lfloor y_0 \rfloor + (-1)^b c = \lfloor y_0 \rfloor + c.$$

Moreover, considering everywhere a 2 appears in the definition of \mathbf{A} , we obtain that

$$\mathbf{w} = \mathbf{A}_1 \lfloor \mathbf{z}_1 \rfloor - qc \mathbf{j} = (\lfloor z_0 \rfloor - c) \mathbf{j} \bmod 2.$$

For the last statement, let us use the two preceding results. In particular, we note

$$\mathbf{w}_1 \cdot \alpha_h + w' \mathbf{j} = \mathbf{w} - \text{LowBits}^h(\mathbf{w}) + \text{LSB}(\mathbf{w}).$$

We note that the last two elements have same parity, as the former one has the same parity as $\text{LowBits}(\mathbf{w}, \alpha_h)$. By Lemma 17 their sum has infinite norm $\leq \alpha_h/2 + 2$. Hence from its definition, it holds that

$$2\tilde{\mathbf{z}}_2 = 2\lfloor \mathbf{z}_2 \rfloor - \text{LowBits}^h(\mathbf{w}) + \text{LSB}(\mathbf{w}) \bmod^\pm 2q.$$

Finally, this holds over the integers as the right-hand side has infinite norm at most $2B' + \alpha_h/2 + 2 < q$. \square

Theorem 19 (Completeness). *Assume that $B'' = B' + \sqrt{n(k + \ell)}/2 + \sqrt{nk} \cdot (\alpha_h/4 + 1) < q/2$. Then the signature schemes of Figure 8 is complete, i.e., for every message M and every key-pair (sk, vk) returned by $\text{KeyGen}(1^\lambda)$, we have:*

$$\text{Verify}(\text{vk}, M, \text{Sign}(\text{sk}, M)) = 1.$$

Proof. We use the notations of the algorithms. The first and second equations from Lemma 18 state that $\rho = \tilde{\rho}$ and thus

$$c = \text{SampleBinaryChallenge}_\tau(\tilde{\rho}).$$

On the other hand, we use the last equation from the same lemma to bound the size of $\tilde{\mathbf{z}}$. We have:

$$\begin{aligned}
\|\tilde{\mathbf{z}}\| &\leq \|\mathbf{z}\| + \|\mathbf{z} - \lfloor \mathbf{z} \rfloor\| + \|\lfloor \mathbf{z} \rfloor - \tilde{\mathbf{z}}\| \\
&\leq B' + \sqrt{n(k + \ell)} \cdot \|\mathbf{z} - \lfloor \mathbf{z} \rfloor\|_\infty + \|\lfloor \mathbf{z}_2 \rfloor - \tilde{\mathbf{z}}_2\| \\
&\leq B' + \frac{\sqrt{n(k + \ell)}}{2} + \sqrt{nk} \cdot \|\text{LowBits}^h(\mathbf{w})\|_\infty \\
&\leq B' + \frac{\sqrt{n(k + \ell)}}{2} + \sqrt{nk} \cdot \left(\frac{\alpha_h}{4} + 1\right).
\end{aligned}$$

The definition of B'' implies that the scheme is correct. □

Runtime. A non-trivial task is analyzing the runtime of Fiat-Shamir with aborts schemes. As shown in [7, Section 6.1], degenerate cases exist where the signing algorithm does not terminate. In particular, it is impossible, even in the ROM, to show that the generic signing algorithm has polynomial expected runtime. However, in the case of HAETAE, all starting points \mathbf{y} lying in the Euclidean ball of radius $B' - \gamma\sqrt{\tau}$ have probability at least $1/2$ of being accepted in the end, whatever shift $\pm sc$ is added, whatever instance of the hash function H is chosen. This gives rise to the following two points.

- With M' defined as in Lemma 12, the probability of doing more than i iterations is bounded from above by $(1 - 1/M')^i + 2^{-\alpha} M'^3$, where α is the commitment min-entropy.
- The signing algorithm has a finite expected runtime.

These are insufficient to conclude that the number of iterations is M' on average, but this is nonetheless what happens for our choice of parameters.

4.3.2 Theoretical Security

We analyze the security of HAETAE by following standard arguments and proof techniques. We start by detailing the underlying canonical identification scheme that makes up HAETAE and give their properties in Lemma 20. We then show that HAETAE is UF-NMA secure in Lemma 21. Finally, we put everything together and give the security bound in Theorem 22 based on the reduction from [3]. Reminders on canonical identification schemes and the Fiat-Shamir with aborts transform can be found in Appendix A.

In Figure 10, we first describe in Figure 10a an “uncompressed” version of the identification scheme, as well as the HAETAE canonical identification scheme in Figure 10b. The latter uses the **Compress** and **Decompress** functions, as described in Figure 9. We omit the instance generator for both CID, as it is exactly the key generation algorithm from HAETAE, except that do not do truncation for simplicity. Adding it can be done by noting that \mathbf{a}_{gen} gives uniform low bits that would otherwise be missing.

We sum up the properties of the CID from Figure 10b.

Lemma 20. *The HAETAE canonical identification scheme from Figure 10b satisfies the following properties.*

Commitment Min-entropy. *The commitment (w_0, \mathbf{w}_1) has min-entropy $\geq n$.*

paHVZK. *Assuming $B^2 \geq B'^2 + \gamma^2 \tau$, the HAETAE CID satisfies the paHVZK property with the simulator described in Figure 11.*

Computational Unique Response. *Let \mathcal{A} be an adversary against the CUR property of the scheme. There exist two adversaries \mathcal{B} and \mathcal{B}' with roughly the same runtime such that:*

$$\text{Adv}_{\text{HAETAE}}^{\text{CUR}}(\mathcal{A}) \leq \text{Adv}_{n,q,k,\ell-1,\eta}^{\text{MLWE}}(\mathcal{B}) + \text{Adv}_{n,q,k,\ell,2B''}^{\text{MSIS}}(\mathcal{B}').$$

Proof. Let us prove each point.

Commitment Min-entropy. Recall that $w_0 = \text{LSB}(\lfloor y_0 \rfloor)$ is a binary polynomial of degree $\leq n$. As the coefficients of y_0 are large and distributed following the distribution of a subset of a uniform-hyperball, we expect the coefficients of w_0 to be roughly uniform, hence the lower bound on the commitment min-entropy.

paHVZK. We first prove that the CID from Figure 10a satisfies paHVZK. Consider the simulator described in Figure 11.

The distribution of real non-aborting transcripts and the output distribution of $\text{Sim}_{\perp}(\mathbf{vk}, c)$ are identical under the condition on the radii. Indeed, by conditioning the distribution of the output of algorithms \mathcal{A} and \mathcal{B} from Lemma 12 on not being \perp , the conditional (on \mathbf{A} and c) distributions of \mathbf{z} in the two cases are identical, and we note that $(\mathbf{w}, c, \mathbf{z})$ is a deterministic function of $(\mathbf{A}, c, \mathbf{z})$.

Finally, given an uncompressed transcript $(\mathbf{w}, c, \mathbf{z})$ for the Figure 10a CID, we note that $((\text{HighBits}^h(\mathbf{w}), \text{LSB}(w_0)), c, \text{Compress}(\mathbf{z}, c))$ is a valid transcript for the HAETAE CID. Indeed, note that $w_0 = \lfloor y_0 \rfloor \bmod 2$ by definition of \mathbf{A} . Moreover, applying this deterministic transform turns the distribution of real transcripts for the uncompressed CID to the distribution of real transcripts for the compressed CID. By an immediate reduction, the HAETAE CID also satisfies paHVZK.

CUR. Given $((\mathbf{w}_1, w_0), c, \sigma, \sigma')$ such that $\mathbf{A}\mathbf{z} - qc\mathbf{j} = \mathbf{w}_1\alpha_h + w_0\mathbf{j} = \mathbf{A}\mathbf{z}' - qc\mathbf{j} \bmod 2q$, we get $\mathbf{A}(\mathbf{z} - \mathbf{z}') = 0 \bmod 2q$, which we can reduce $\bmod q$:

$$(-\mathbf{b}|\mathbf{A}_{\text{gen}}|\mathbf{Id}_k)(\mathbf{z} - \mathbf{z}') = 0 \bmod q.$$

We prove that $\mathbf{z} \neq \mathbf{z}' \bmod q$. Let $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$ and $\mathbf{z}' = (\mathbf{z}'_1, \mathbf{z}'_2)$. Let $\sigma = (x, \mathbf{v}, h)$ and $\sigma' = (x', \mathbf{v}', h')$. If $\mathbf{z}_1 = \mathbf{z}'_1$ then $x = x'$ and $\mathbf{v} = \mathbf{v}'$ due to the encoding being unique. This means that $h \neq h'$, which in turn will give two different values for \mathbf{z}_2 and \mathbf{z}'_2 as the other values composing \mathbf{z}_2 and \mathbf{z}'_2 are identical.

However, $(\mathbf{A}_{\text{gen}}, \mathbf{b})$ is a $\text{MLWE}_{n,q,k,\ell-1,\eta}$ instance instead of being uniform. Both adversaries \mathcal{B} and \mathcal{B}' can run \mathcal{A} by simulating a random oracle in the case of \mathcal{B} and by forwarding the queries to its own oracle in the case of \mathcal{B}' . Then, if \mathcal{A} was successful, adversary \mathcal{B} outputs “MLWE” while \mathcal{B}' computes the MSIS solution as described above, and if \mathcal{A} is unsuccessful, adversary \mathcal{B} outputs “unif” while \mathcal{B}' aborts. By the triangular inequality, we get the desired inequality. \square

We reduce the UF-NMA security of the signature scheme to the BimodalSelfTargetMSIS problem.

Lemma 21 (UF-NMA security). *Let \mathcal{A} be an adversary against the UF-NMA security of HAETAE. There exist two adversaries \mathcal{B} and \mathcal{B}' with essentially the same runtime as \mathcal{A} such that:*

$$\text{Adv}_{\text{HAETAE}}^{\text{UF-NMA}}(\mathcal{A}) \leq \text{Adv}_{n,q,k,\ell-1,\eta}^{\text{MLWE}}(\mathcal{B}) + \text{Adv}_{H,n,q,k,\ell,B''}^{\text{BimodalSelfTargetMSIS}}(\mathcal{B}').$$

Proof. Let us first describe \mathcal{B}' . On input the public matrix \mathbf{A} , it calls \mathcal{A} on it. The adversary \mathcal{B}' defines the random oracle $H' : (\mathbf{w}_1, w_0, \mu) \mapsto H(\mathbf{w}_1 \alpha_h + w_0 \mathbf{j} \bmod 2q, \mu)$, to take care of the compression that does not appear in the BimodalSelfTargetMSIS problem. This can be simulated by \mathcal{B} by applying the right function on the queries of \mathcal{A} and forwarding it to its own random oracle H .

When \mathcal{A} outputs a valid forgery $\sigma^* = ((x, \mathbf{v}, h), c)$ for some message μ^* , adversary \mathcal{B}' runs $\text{Decompress}(\mathbf{A}, (x, \mathbf{v}, h), c)$ to get a vector \mathbf{z} such that $\|\mathbf{z}\| \leq B''$ and $H'(\text{HighBits}^h(\mathbf{w}^*), \text{LSB}(w_0^*), \mu) = c$, where we let $\mathbf{w}^* = \mathbf{A}\mathbf{z} - qc\mathbf{j} \bmod 2q$. By construction $\mathbf{w}^* = \text{HighBits}^h(\mathbf{w}^*)\alpha_h + \text{LSB}(w_0^*)\mathbf{j} \bmod 2q$, meaning that $c = H(\mathbf{w}^*, \mu)$. Then (\mathbf{z}, c, μ^*) is exactly a solution to the $\text{BimodalSelfTargetMSIS}_{H,n,q,k,\ell,B''}$ problem.

Note that \mathcal{A} is called on a “lossy” instance of HAETAE, where the verification key \mathbf{b} does not have a signing key associated and is actually uniform. This difference in setting exactly corresponds to the $\text{MLWE}_{n,q,k,\ell-1,\eta}$ problem. As such the design of \mathcal{B} is as follows: on input $(\mathbf{A}_{\text{gen}}, \mathbf{b})$, it puts together the verification key \mathbf{A} and calls \mathcal{A} on it. It can simulate a random oracle for it, and outputs “MLWE” if \mathcal{A} is successful at forging, “unif” otherwise. \square

We combine the previous results with Theorem 27 to get the following security bound for HAETAE.

Theorem 22. *Let \mathcal{A} be an adversary against the UF-CMA security of HAETAE making Q_s signature queries and Q_h hash queries. Let α be the commitment min-entropy of HAETAE. Let $B^2 \geq B'^2 + \gamma^2\tau$. There exist two adversaries \mathcal{B} and \mathcal{B}' such that*

$$\begin{aligned} \text{Adv}_{\text{HAETAE}}^{\text{UF-CMA}}(\mathcal{A}) &\leq \text{Adv}_{n,q,k,\ell-1,\eta}^{\text{MLWE}}(\mathcal{B}) + \text{Adv}_{H,n,q,k,\ell,B''}^{\text{BimodalSelfTargetMSIS}}(\mathcal{B}') \\ &\quad + \frac{2^{-\alpha/2+1}Q_s}{1-\beta} \sqrt{Q_h + 1 + \frac{Q_s}{1-\beta}} + 2^{-\alpha/2+1}(Q_h + 1) \sqrt{\frac{Q_s}{1-\beta}}. \end{aligned} \tag{4.1}$$

If \mathcal{A} is an adversary against the sUF-CMA security of HAETAE, then there exist two more adversaries \mathcal{B}'' and \mathcal{B}''' such that the previous bound holds by adding the extra term $\text{Adv}_{n,q,k,\ell-1,\eta}^{\text{MLWE}}(\mathcal{B}'') + \text{Adv}_{n,q,k,\ell,2B''}^{\text{MSIS}}(\mathcal{B}''')$.

4.4 HAETAE with Pre-computation

We observe that in the randomized signing process of HAETAE, many operations do not depend on the message M , and some do not even depend on the signing key. This enables efficient “offline” procedures, i.e., precomputations that speed up the “online” phase.

Specifically, there are two levels of offline signing that can be applied to randomized HAETAE:

1. **Generic.** If neither the message M nor the signing key is chosen in advance, it is still possible to perform hyperball sampling. This removes the most time-consuming operation from the online phase.
2. **Designated signing key.** Here, only the message M is unknown during offline signing, while the signing key is fixed. This allows us to perform even more pre-computations by using only the verification key, as shown in Figure 12. Most notably, there is no longer a matrix-vector multiplication in the online phase.

We showcase the offline and online parts of the (randomized) version of HAETAE in Figure 12.

4.5 Parameter Sets

Parameter Sets		HAETAE-2	HAETAE-3	HAETAE-5
Security		120	180	260
n	Degree of \mathcal{R} (2.1)	256	256	256
(k, ℓ)	Dimensions of $\mathbf{z}_2, \mathbf{z}_1$ (4.2)	(2,4)	(3,6)	(4,7)
q	Modulus for MLWE & MSIS (2.3)	64513	64513	64513
η	Range of sk coefficients (2.1)	1	1	1
τ	Weight of c (3.3)	58	80	128
γ	sk rejection parameter (3.1)	48.858	57.707	55.13
	Resulting key acceptance rate (3.1)	0.1	0.1	0.1
d	Truncated bits of vk (3.1)	1	1	0
M	Expected # of repetitions (3.4)	6.0	5.0	6.0
B	y radius (3.4)	9846.02	18314.98	22343.66
B'	Rejection radius (3.4)	9838.98	18307.70	22334.95
B''	Verify radius (4.2)	12777.52	21906.65	24441.49
α	\mathbf{z}_1 compression factor (3.5)	256	256	256
α_h	\mathbf{h} compression factor (3.5)	512	512	256
Forgery: SIS Hardness (Core-SVP)				
BKZ block-size b (GSA)		409 (333)	617 (512)	878 (735)
Classical Core-SVP		119 (97)	180 (149)	256 (214)
Quantum Core-SVP		105 (85)	158 (131)	225 (188)
Key-recovery: LWE Hardness (Core-SVP and refined)				
BKZ block-size b (GSA)		428	810	988
Classical Core-SVP		125	236	288
Quantum Core-SVP		109	208	253
BKZ block-size b (simulation)		439	834	1019
\log_2 Classical Gates		159	270	322
\log_2 Classical Memory		99	177	214

Table 1: HAETAE parameters sets. Hardness is measured with the Core-SVP methodology and a refined analysis is given for LWE. The numbers in parenthesis for SIS are for the strong unforgeability property.

To choose parameters reaching the desired NIST security levels, we estimated the costs of practical attacks, as in Dilithium, Falcon, and many other NIST-submitted schemes. In particular, our methodology is directly inspired from the one used in Dilithium, and we sum it up in the following.

Looking at Equation 4.1, we note that we have highly underestimated the commitment min-entropy of the scheme in Lemma 20 as we ignored its biggest part \mathbf{w}_1 , and we choose to ignore terms on the second line as we are confident that they are small enough. We first evaluate the cost of attacks on MLWE, i.e. key-recovery attacks. Second, to evaluate the cost of forgery attacks, i.e. attacks on BimodalSelfTargetMSIS, we use the fact that the only known way to solve BimodalSelfTargetMSIS is to solve MSIS. Heuristically, the hash function is not aware of the algebraic structure of its input, and the random oracle assumption that c is uniform and independent from the input is sound. Thus, an adversary has no choice but to choose some \mathbf{w} , hash its high and low bits along with some message, and try to compute a short preimage of $\mathbf{w} - qc\mathbf{j} \bmod 2q$. If the adversary succeeds, the preimage is in particular an $\text{MSIS}_{n,q,k,\ell+1,\sqrt{B''^2+1}}$ solution for the matrix $[\mathbf{w}|\mathbf{A}]$. Finally, we evaluate the cost of MSIS with a bound twice as loose to evaluate the strong unforgeability of the HAETAE scheme. Our cryptanalysis of these problems follows the approach taken in Dilithium. We provide a modification of their security estimation script², where we also updated the cost of quantum attacks following recent works [5]. These estimations follow the CoreSVP approach. In the case of MLWE, we also give refined estimates, computed in a branch³ of the leaky LWE estimator from [6].

We propose three different parameter sets with varying security levels, where we prioritize low signature and verification key sizes over faster execution time. The parameter choices are versatile, adaptable and allow size vs. speed trade-offs at consistent security levels. For example at cost of larger signatures, a smaller repetition rate M is possible and thus a faster signing process. This versatility is a notable advantage over Falcon and Mitaka.

Like in Dilithium, our modulus q is constant over the parameter sets and allows an optimized NTT implementation shared for all sets. With only 16-bit in size, our modulus also allows storing coefficients memory-efficiently without compression.

²Available in the submission package.

³<https://github.com/jdevevey/refined-haetae>

KeyGen(1^λ):

- 1: $\text{seed} \leftarrow \{0, 1\}^{\rho_0}$ ▷ KeyGen for $d = 1$
- 2: $(\text{seed}_A, \text{seed}_{sk}, K) = H_{\text{gen}}(\text{seed})$
- 3: $(\mathbf{a}_{\text{gen}} | \mathbf{A}_{\text{gen}}) := \text{expandA}(\text{seed}_A) \in \mathcal{R}_q^{k \times \ell}$
- 4: $\text{counter}_{sk} = 0$
- 5: $(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) := \text{expandS}(\text{seed}_{sk}, \text{counter}_{sk})$
- 6: $\mathbf{b} = \mathbf{a}_{\text{gen}} + \mathbf{A}_{\text{gen}} \cdot \mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}} \in \mathcal{R}_q^k$
- 7: $(\mathbf{b}_0, \mathbf{b}_1) = (\text{LowBits}^{\text{vk}}(\mathbf{b}), \text{HighBits}^{\text{vk}}(\mathbf{b}))$
- 8: $\mathbf{A} = (2(\mathbf{a}_{\text{gen}} - 2\mathbf{b}_1) + q\mathbf{j} | 2\mathbf{A}_{\text{gen}} | 2\mathbf{Id}_k) \bmod 2q$
- 9: $\mathbf{s} = (1, \mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}} - \mathbf{b}_0)$
- 10: **if** $\mathcal{N}(\mathbf{s}) > \gamma^2 n$ **then** $\text{counter}_{sk}++$ and **Go to 5**
- 11: **return** $\text{sk} = (\mathbf{s}, K)$, $\text{vk} = (\text{seed}_A, \mathbf{b}_1)$

Sign(sk, M):

- 1: $\mu = H_{\text{gen}}(\text{seed}_A, \mathbf{b}_1, M)$
- 2: $\text{seed}_{ybb} = H_{\text{gen}}(K, \mu)$
- 3: $\text{counter} = 0$
- 4: $(\mathbf{y}, b, b') := \text{expandYbb}(\text{seed}_{ybb}, \text{counter})$
- 5: $\mathbf{w} \leftarrow \mathbf{A}[\mathbf{y}]$
- 6: $\rho = H(\text{HighBits}^h(\mathbf{w}), \text{LSB}(\lfloor y_0 \rfloor), \mu)$
- 7: $c = \text{SampleBinaryChallenge}_\tau(\rho)$
- 8: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c \mathbf{s}$
- 9: $\mathbf{h} = \text{HighBits}^h(\mathbf{w}) - \text{HighBits}^h(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rfloor) \bmod^{+ \frac{2(q-1)}{\alpha_h}}$
- 10: **if** $\|\mathbf{z}\|_2 \geq B'$ **then**
- 11: $\text{counter}++$ and **Go to 4**
- 12: **else if** $\|2\mathbf{z} - \mathbf{y}\|_2 < B \wedge b' = 0$ **then**
- 13: $\text{counter}++$ and **Go to 4**
- 14: **else**
- 15: $x = \text{Encode}(\text{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor))$
- 16: $\mathbf{v} = \text{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor)$
- 17: **return** $\sigma = (x, \mathbf{v}, \text{Encode}(\mathbf{h}), c)$

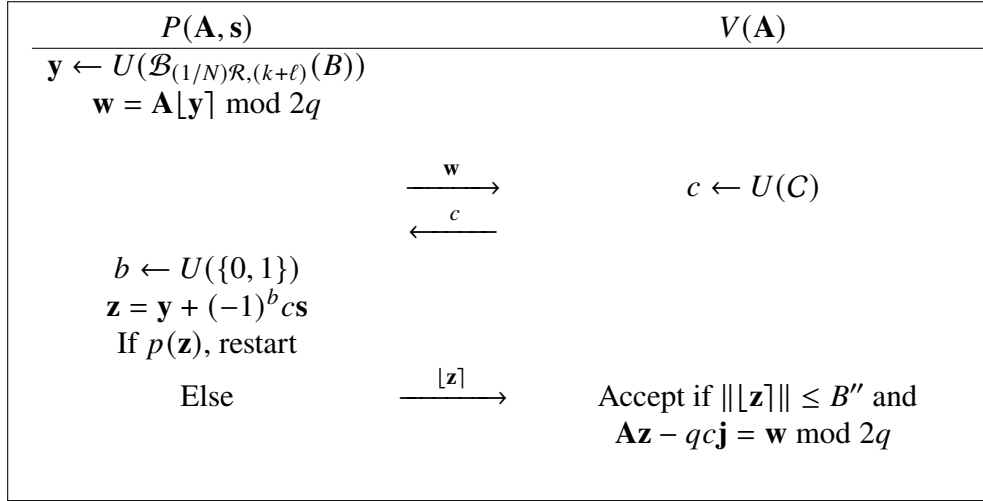
Verify($\text{vk}, M, \sigma = (x, \mathbf{v}, h, c)$):

- 1: $\tilde{\mathbf{z}}_1 = \text{Decode}(x) \cdot 256 + \mathbf{v}$ and $\tilde{\mathbf{h}} = \text{Decode}(h)$
- 2: $(\mathbf{a}_{\text{gen}} | \mathbf{A}_{\text{gen}}) = \text{expandA}(\text{seed}_A)$
- 3: $\mathbf{A}_1 = (2(\mathbf{a}_{\text{gen}} - 2\mathbf{b}_1) + q\mathbf{j} | 2\mathbf{A}_{\text{gen}}) \bmod 2q$
- 4: $\mathbf{w}_1 = \tilde{\mathbf{h}} + \text{HighBits}^h(\mathbf{A}_1 \tilde{\mathbf{z}}_1 - qc\mathbf{j}) \bmod^{+ \frac{2(q-1)}{\alpha_h}}$
- 5: $w' = \text{LSB}(\tilde{z}_0 - c)$
- 6: $\tilde{\mathbf{z}}_2 = (\mathbf{w}_1 \cdot \alpha_h + w'\mathbf{j} - (\mathbf{A}_1 \tilde{\mathbf{z}}_1 - qc\mathbf{j})) / 2 \bmod^{\pm} q$
- 7: $\tilde{\mathbf{z}} = (\tilde{\mathbf{z}}_1, \tilde{\mathbf{z}}_2)$
- 8: $\tilde{\mu} = H_{\text{gen}}(\text{seed}_A, \mathbf{b}_1, M)$
- 9: $\tilde{\rho} = H(\mathbf{w}_1, w', \tilde{\mu})$
- 10: **return** $(c = \text{SampleBinaryChallenge}_\tau(\tilde{\rho})) \wedge (\|\tilde{\mathbf{z}}\| < B'')$

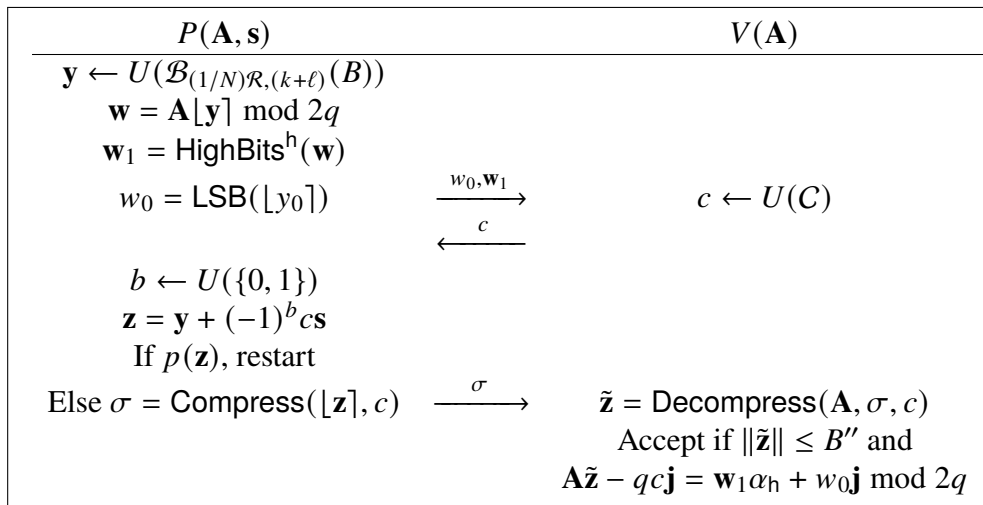
Figure 8: Full description of deterministic HAETAE. The KeyGen algorithm is slightly different for $d = 0$ (HAETAE-260), which do not truncate \mathbf{b} . See Section 3.1.1 for details.

Compress(\mathbf{z}, c):	Decompress($\mathbf{A}, (x, \mathbf{v}, h), c$):
1: if $\mathbf{z} = \perp$ then	1: if $(x, \mathbf{v}, h) = (\perp, \perp, \perp)$ then
2: return (\perp, \perp, \perp)	2: return \perp
3: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c \mathbf{s}$	3: $\tilde{\mathbf{z}}_1 = \text{Decode}(x) \cdot 256 + \mathbf{v}$
4: $\mathbf{h} = \text{HighBits}^h(\mathbf{w})$ $-\text{HighBits}^h(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rfloor) \bmod^{+} \frac{2(q-1)}{\alpha_h}$	4: $\tilde{\mathbf{h}} = \text{Decode}(h)$
5: $x = \text{Encode}(\text{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor))$	5: $\mathbf{A} = (\mathbf{A}_1 \mid 2\mathbf{I}) \bmod 2q$
6: $\mathbf{v} = \text{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor)$	6: $\mathbf{w}_1 = \text{HighBits}^h(\mathbf{A}_1 \tilde{\mathbf{z}}_1 - qc\mathbf{j})$ $+\tilde{\mathbf{h}} \bmod^{+} \frac{2(q-1)}{\alpha_h}$
7: return $(x, \mathbf{v}, \text{Encode}(\mathbf{h}))$	7: $w' = \text{LSB}(\tilde{z}_0 - c)$
	8: $\tilde{\mathbf{w}} = \mathbf{A}_1 \tilde{\mathbf{z}}_1 - qc\mathbf{j}$
	9: $\tilde{\mathbf{z}}_2 = (\mathbf{w}_1 \cdot \alpha_h + w'\mathbf{j} - \tilde{\mathbf{w}}) / 2 \bmod^{\pm} q$
	10: return $\tilde{\mathbf{z}} = (\tilde{\mathbf{z}}_1, \tilde{\mathbf{z}}_2)$

Figure 9: Compression and decompression algorithms.



(a) Uncompressed HAETAE



(b) Compressed HAETAE

Figure 10: Underlying canonical identification schemes.

$\text{Sim}_{\perp}(\text{vk}, c) :$ 1: $\mathbf{z} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R}, k+\ell}(B'))$ 2: $\mathbf{w} \leftarrow \mathbf{A}[\mathbf{z}] - qc\mathbf{j}$ 3: return $(\mathbf{w}, c, \mathbf{z})$
--

Figure 11: Uncompressed HAETAE simulator.

$\text{Sign}^{\text{offline}}(\text{vk}) :$ 1: $(\mathbf{a}_{\text{gen}} \mathbf{A}_{\text{gen}}) = \text{expandA}(\text{seed}_{\mathbf{A}})$ 2: $\mathbf{A}_1 = (2(\mathbf{a}_{\text{gen}} - 2\mathbf{b}_1) + q\mathbf{j} 2\mathbf{A}_{\text{gen}}) \bmod 2q$ 3: $\text{List} = ()$ 4: for iter in $[L]$ do 5: $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R}, (k+\ell)}(B))$ 6: $\mathbf{w} = \mathbf{A}[\mathbf{y}]$ 7: $\mathbf{w}_1 = \text{HighBits}^h(\mathbf{w})$ 8: $\text{List.append}(\mathbf{y}, \mathbf{w}, \mathbf{w}_1, \text{LSB}(\lfloor y_0 \rfloor))$ 9: return List
$\text{Sign}^{\text{online}}(\text{sk}, \text{List}, M) :$ 1: $\mu = H_{\text{gen}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_1, M)$ 2: $\text{tuple} = (\mathbf{y}, \mathbf{w}, \text{tuple}_3, \text{tuple}_4) \leftarrow \text{List}$ 3: $\text{List.delete}(\text{tuple})$ 4: $c = \text{SampleBinaryChallenge}_{\tau}(H(\text{tuple}_3, \text{tuple}_4, \mu))$ 5: $(b, b') \leftarrow \{0, 1\}^2$ 6: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c\mathbf{s}$ 7: $\mathbf{h} = \text{tuple}_3 - \text{HighBits}^h(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rfloor) \bmod^{+} \frac{2(q-1)}{\alpha_h}$ 8: if $\ \mathbf{z}\ _2 \geq B'$ then Go to 2 9: else if $\ 2\mathbf{z} - \mathbf{y}\ _2 < B \wedge b' = 0$ then Go to 2 10: else 11: $x = \text{Encode}(\text{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor))$ 12: $\mathbf{v} = \text{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rfloor)$ 13: return $\sigma = (x, \mathbf{v}, \text{Encode}(\mathbf{h}), c)$

Figure 12: Randomized, on/off-line signing.

5 Implementation Details

In this section, we detail how to efficiently implement HAETAE.

We start this section with an implementation-oriented specification. Specifically, Figure 13 demonstrates how to implement the key generation, Figure 14 the signature generation and Figure 15 the signature verification. These illustrate the use of the CRT and NTT for efficient polynomial arithmetic. The most notable remark from an implementation point of view is that \mathbf{b} can be transmitted in NTT domain if no rounding is applied, and most arithmetic is carried out modulo q , and recovering the values modulo $2q$ is only required for computing the low and high bits. Relevant for signing and verification is the unpacking routine for $\hat{\mathbf{A}}$, depicted in Algorithm 1.

5.1 Hyperball Sampler

Essentially, the hyperball sampling procedure consists of four steps:

1. Sample $n(k + \ell) + 2$ discrete Gaussians with $\sigma = 2^{76}$, sum up their squares, and drop two samples eventually.
2. Compute the inverse of the square root of the sum of squares, multiply the result by $B_0 + \sqrt{nm}/(2N)$.
3. Multiply every sample from Step 1 by the result of Step 2.
4. Check the ℓ_2 norm of the resulting vector, start from Step 1 if this is bigger than B_0N .

In the following, we explain how the Gaussian sampling and the approximation of the inverse of the square root can be implemented efficiently. Besides, we choose to generate each of the $k + \ell$ polynomials independently, which helps parallelizing the randomness generation for implementations that use vectorization and hardware implementations. Then, for the first two polynomials, we generate one more Gaussian sample each, which is never stored but included in the sum of squared samples.

5.1.1 Discrete Gaussian Sampling

As we will lose precision when computing the inverse square root of a Gaussian sample, we require a Gaussian sampler with high fix-point precision. This is achieved by sampling over \mathbb{Z} with a large standard deviation and then scaling the resulting sample to our convenience. We use [22, Algorithm 12] to sample from a discrete Gaussian distribution with $\sigma = 2^{76}$, $k = 2^{72}$.

In essence, we start by sampling a discrete Gaussian x with $\sigma = 16$ using a CDT and a uniform $y \in \{0, \dots, 2^{72} - 1\}$ and set the Gaussian sample candidate as $r = x2^{72} + y$. Subsequently, this candidate is accepted with probability $\exp(-y(y + x2^{73})/2^{153})$. Fortunately, we achieve a very low rejection rate of less than 5 %.

Specifically, the CDT we use has 64 entries and uses a precision of 16 bit. Then, to compute the sample candidate's square and the input to the exponential, we first compute r^2 and round the result to 76-bit precision, which is accumulated later if the sample is accepted. Subsequently, $r^2 - 2^{76}x^2$ yields the input to the exponential.

KeyGen(1^λ) for $d > 0$

```

1: seed  $\leftarrow \{0, 1\}^{\rho_0}$ 
2: (seedA, seedsk,  $K$ )  $:= H_{\text{gen}}(\text{seed})$ 
3: ( $\mathbf{a}_{\text{gen}}, \widehat{\mathbf{A}}_{\text{gen}}$ )  $:= \text{expandA}_d(\text{seed}_A)$   $\triangleright (\mathbf{a}_{\text{gen}}, \widehat{\mathbf{A}}_{\text{gen}}) \in \mathcal{R}_q^k \times \mathcal{R}_q^{k \times \ell - 1}$ 
4: (countersk, flag)  $:= (0, \text{true})$ 
5: while flag do
6:   ( $\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}$ )  $:= \text{expandS}(\text{seed}_{\text{sk}}, \text{counter}_{\text{sk}})$   $\triangleright (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \in \mathcal{S}_\eta^{\ell-1} \times \mathcal{S}_\eta^k$ 
7:    $\mathbf{b} := \mathbf{a}_{\text{gen}} + \text{NTT}^{-1}(\widehat{\mathbf{A}}_{\text{gen}} \circ \text{NTT}(\mathbf{s}_{\text{gen}})) + \mathbf{e}_{\text{gen}} \bmod q$   $\triangleright \mathbf{b} \in \mathcal{R}_q^k$ 
8:   ( $\mathbf{b}_0, \mathbf{b}_1$ )  $:= (\text{LowBits}^{\text{vk}}(\mathbf{b}), \text{HighBits}^{\text{vk}}(\mathbf{b}))$ 
9:   ( $\mathbf{s}_1, \mathbf{s}_2$ )  $:= (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}} - \mathbf{b}_0)$ 
10:  countersk  $:= \text{counter}_{\text{sk}} + 1$ 
11:  if  $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \leq \gamma^2 n$  then
12:    flag  $:= \text{false}$ 
13:  $tr := H(\text{seed}_A, \mathbf{b}_1)$ 
14: return sk = ( $\mathbf{s}_1, \mathbf{s}_2, K, tr, \text{seed}_A, \mathbf{b}_1$ ), vk = ( $\text{seed}_A, \mathbf{b}_1$ )

```

KeyGen(1^λ) for $d = 0$

```

1: seed  $\leftarrow \{0, 1\}^{\rho_0}$ 
2: (seedA, seedsk,  $K$ )  $:= H_{\text{gen}}(\text{seed})$ 
3: (countersk, flag)  $:= (0, \text{true})$ 
4: while flag do
5:   ( $\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}$ )  $:= \text{expandS}(\text{seed}_{\text{sk}}, \text{counter}_{\text{sk}})$   $\triangleright (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \in \mathcal{S}_\eta^{\ell-1} \times \mathcal{S}_\eta^k$ 
6:   ( $\mathbf{s}_1, \mathbf{s}_2$ )  $:= (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}})$ 
7:   countersk  $:= \text{counter}_{\text{sk}} + 1$ 
8:   if  $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \leq \gamma^2 n$  then
9:     flag  $:= \text{false}$ 
10:   $\widehat{\mathbf{A}}_{\text{gen}} := \text{expandA}_d(\text{seed}_A)$   $\triangleright \widehat{\mathbf{A}}_{\text{gen}} \in \mathcal{R}_q^{k \times \ell - 1}$ 
11:   $\widehat{\mathbf{b}} := -2 \left( \widehat{\mathbf{A}}_{\text{gen}} \circ \text{NTT}(\mathbf{s}_{\text{gen}}) + \text{NTT}(\mathbf{e}_{\text{gen}}) \right) \bmod q$   $\triangleright \widehat{\mathbf{b}} \in \mathcal{R}_q^k$ 
12:   $tr := H(\text{seed}_A, \widehat{\mathbf{b}})$ 
13: return sk = ( $\mathbf{s}_1, \mathbf{s}_2, K, tr, \text{seed}_A, \widehat{\mathbf{b}}$ ), vk = ( $\text{seed}_A, \widehat{\mathbf{b}}$ )

```

Figure 13: Implementation specification (deterministic version) of HAETAE key generation

Approximating the Exponential. For this, we need to approximate the exponential function e^{-x} by a polynomial $f(x)$ on the closed interval $[c - \frac{w}{2}, c + \frac{w}{2}]$, with center c and width w . We first determine an upper bound for the polynomial order required to approximate e^{-x} , given an upper bound for the absolute error. We obtain $f(x)$ by truncating the expansion of e^{-x} into a series of Chebyshev polynomials of the first kind $T_n(x)$ with linearly transformed input, as this is known to yield small absolute approximation errors for a given polynomial order. We find:

$$e^{-x} = -e^{-c} + 2e^{-c} \sum_{n=0}^{\infty} (-1)^n I_n\left(\frac{w}{2}\right) T_n\left(\frac{x-c}{w/2}\right) \quad x \in [c - \frac{w}{2}, c + \frac{w}{2}]$$

Sign(sk, M)

- 1: $(s_1, s_2, K, tr, \text{seed}_A, \psi) := \text{sk}$
- 2: $\hat{\mathbf{A}} := \text{unpackA}_d(\text{seed}_A, \psi)$ ▷ Algorithm 1, $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$
- 3: $\mu := H_{\text{gen}}(tr, M)$
- 4: $\text{seed}_{ybb} := H_{\text{gen}}(K, \mu)$
- 5: $(\kappa, \sigma) := (0, \perp)$
- 6: **while** $\sigma = \perp$ **do** ▷ pre-compute $(\hat{s}_1, \hat{s}_2) := (\text{NTT}(s_1), \text{NTT}(s_2))$
- 7: $(y_1, y_2, b, b', \kappa) := \text{expandYbb}(\text{seed}_{ybb}, \kappa)$ ▷ $(y_1, y_2) \in (1/N)\mathcal{R}^\ell \times (1/N)\mathcal{R}^k$
- 8: $\mathbf{w} := \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\lfloor y_1 \rfloor)) + 2 \cdot \lfloor y_2 \rfloor \bmod q$ ▷ $\mathbf{w} \in \mathcal{R}_q^k$
- 9: $\mathbf{w}' := \text{fromCRT}(\mathbf{w}, \lfloor y_{1,1} \rfloor)$ ▷ Algorithm 2, $\mathbf{w}' \in \mathcal{R}_{2q}^k$
- 10: $\mathbf{w}'_1 := \text{HighBits}^h(\mathbf{w}')$
- 11: $\rho = H(\mathbf{w}'_1, \text{LSB}(\lfloor y_{1,1} \rfloor), \mu)$
- 12: $c = \text{SampleBinaryChallenge}_\tau(\rho)$
- 13: $\hat{c} := \text{NTT}(c)$
- 14: $z_{1,1} := y_{1,1} + (-1)^b \cdot c$ ▷ $(z_1, z_2) \in (1/N)\mathcal{R}^\ell \times (1/N)\mathcal{R}^k$
- 15: $(z_1)_{2..\ell} := (y_1)_{2..\ell} + (-1)^b \text{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$
- 16: $z_2 := y_2 + (-1)^b \text{NTT}^{-1}(\hat{c} \circ \hat{s}_2)$
- 17: **if** $\|(z_1, z_2)\|_2 < B'$ **and** $(\|2(z_1, z_2) - (y_1, y_2)\|_2 > B \text{ or } b' = 1)$ **then**
▷ Check this condition in constant time.
- 18: $\mathbf{h} := \mathbf{w}'_1 - \text{HighBits}^h(\mathbf{w}' - 2 \lfloor z_2 \rfloor) \bmod^{+ \frac{2(q-1)}{\alpha_h}}$
- 19: $\sigma := \text{packSig}(\text{HighBits}^{z_1}(\lfloor z_1 \rfloor), \text{LowBits}^{z_1}(\lfloor z_1 \rfloor), \mathbf{h}, c)$
▷ Section 5.2 (can fail and return \perp)

Figure 14: Implementation specification (deterministic version) of HAETAE signing.

Verify(vk, M, σ)

- 1: $(\text{seed}_A, \psi) := \text{vk}$
- 2: $\hat{\mathbf{A}} := \text{unpackA}_d(\text{seed}_A, \psi)$ ▷ Algorithm 1, $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$
- 3: $(\text{HighBits}^{z_1}(\lfloor z_1 \rfloor), \text{LowBits}^{z_1}(\lfloor z_1 \rfloor), \mathbf{h}, c) := \text{unpackSig}(\sigma)$
▷ Section 5.2 (can fail and cause a rejection)
- 4: $\tilde{z}_1 := \text{HighBits}^{z_1}(\lfloor z_1 \rfloor) \cdot 256 + \text{LowBits}^{z_1}(\lfloor z_1 \rfloor)$
- 5: $w' := \text{LSB}(\tilde{z}_{1,1} - c)$
- 6: $\tilde{\mathbf{w}} := \hat{\mathbf{A}} \circ \text{NTT}(\tilde{z}_1) \bmod q$
- 7: $\tilde{\mathbf{w}}' := \text{fromCRT}(\tilde{\mathbf{w}}, w')$ ▷ Algorithm 2
- 8: $\tilde{\mathbf{w}}'_1 := \tilde{\mathbf{h}} + \text{HighBits}^h(\tilde{\mathbf{w}}') \bmod^{+ \frac{2(q-1)}{\alpha_h}}$
- 9: $\tilde{z}_2 := [\tilde{\mathbf{w}}'_1 \cdot \alpha_h + w' \mathbf{j} - \tilde{\mathbf{w}}' \bmod 2q] / 2$ ▷ addition with w' only for first vector element
- 10: $\tilde{\mu} = H_{\text{gen}}(\text{seed}_A, \psi, M)$
- 11: **Return** $(c = \text{SampleBinaryChallenge}_\tau(H(\tilde{\mathbf{w}}'_1, w', \tilde{\mu}))) \wedge (\|(\tilde{z}_1, \tilde{z}_2)\| < B'')$

Figure 15: Implementation specification (deterministic version) of HAETAE verification

where $I_n(z)$ are modified Bessel functions of the first kind, which rapidly converge to zero for growing n . We recall $\|T_n(x)\| \leq 1$ for $\|x\| \leq 1$. For intervals $[0, w]$ with not too large widths we find $2e^{-c} I_{m+1}(\frac{w}{2})$ to be a useful estimate of the maximum absolute error, when


```

expandYbb(seedybb,  $\kappa$ ):
1:  $(\mathbf{y}_1, \mathbf{y}_2) := \perp$   $\triangleright (\mathbf{y}_1, \mathbf{y}_2) \in (1/N)\mathcal{R}^\ell \times (1/N)\mathcal{R}^k$ 
2: while  $(\mathbf{y}_1, \mathbf{y}_2) = \perp$  do
3:    $\mathbf{t}_1 := \text{sampleGauss}(\text{SHAKE256}(\text{seed}_{ybb}, \kappa), n + 1)$   $\triangleright$  subsubsection 5.1.1
4:    $\mathbf{t}_2 := \text{sampleGauss}(\text{SHAKE256}(\text{seed}_{ybb}, \kappa + 1), n + 1)$   $\triangleright$  subsubsection 5.1.1
5:   for  $i := 3$  to  $k + \ell$  do
6:      $\mathbf{t}_i := \text{sampleGauss}(\text{SHAKE256}(\text{seed}_{ybb}, \kappa + i), n)$   $\triangleright$  subsubsection 5.1.1
7:    $s := \sum_{i=1}^{n+1} t_{1,i}^2 + \sum_{i=1}^{n+1} t_{2,i}^2 + \sum_{i=3}^{k+\ell} \sum_{j=1}^n t_{i,j}^2$ 
8:   drop  $t_{1,n}$  and  $t_{2,n}$ 
9:   approximate  $1/\sqrt{s}$   $\triangleright$  subsubsection 5.1.2
10:  for  $i := 0$  to  $k + \ell - 1$  do
11:    for  $j := 0$  to  $n - 1$  do
12:       $t_{i,j} := \left\lceil t_{i,j} \cdot \left( BN + \frac{\sqrt{nm}}{2} \right) \cdot \frac{1}{\sqrt{s}} \right\rceil$   $\triangleright$  round to  $\log_2 N$  fix-point bits
13:   $\kappa := \kappa + k + \ell$ 
14:  if  $\sum_{i=0}^{k+\ell-1} \sum_{j=0}^{n-1} t_{i,j}^2 \leq (BN)^2$  then
15:    arrange  $\mathbf{t}_1, \dots, \mathbf{t}_\ell$  as  $\mathbf{y}_1$ 
16:    arrange  $\mathbf{t}_{\ell+1}, \dots, \mathbf{t}_{k+\ell}$  as  $\mathbf{y}_2$ 
17:  sample  $b, b'$  as the first two bits from the output of  $\text{SHAKE256}(\text{seed}_{ybb}, \kappa)$ 
18:   $\kappa := \kappa + 1$ 
19: return  $(\mathbf{y}_1, \mathbf{y}_2, b, b', \kappa)$ 

```

Figure 16: Deterministic hyperball sampling.

truncating the series at order $m > 1$. This relation allows us to directly limit m according to the interval to cover and the maximum permissible error.

We then determine the polynomial $f(x)$ of at most order m by using the Chebyshev approximation formula, which has been shown to result in a nearly optimal approximation polynomial in the case of the exponential function [18]. The number of fraction bits is chosen to match the error. The numerical evaluation is performed in fixed-point arithmetic using the Horner's scheme and multiplying with shifts to retain significant bits. When shifting right, we round half up, which retains about one additional bit of accuracy when compared to truncation.

Barthe et al. [4] introduced the GALACTICS toolbox to derive suitable polynomials approximating e^{-x} . They numerically evaluate and modify trial polynomials, minimizing the *relative* error, until an acceptable level is reached. The polynomials are evaluated using a Horner's scheme, similar to this work, but rely on truncation. When comparing to polynomials derived using the GALACTICS toolbox, our approximation has a slightly smaller absolute error for intervals of interest in this work, while maintaining the same polynomial order and constant time properties. This holds even when introducing rounding to the GALACTICS evaluation of polynomials. Moreover, our approach is somewhat less heuristic than the GALACTICS method. Practically, as can be seen in Listing 1, the approximation consists of six signed 48-bit multiplications with subsequent rounding (smulh48), several constant shifts with rounding and constant additions.

Listing 1: Fix-point approximation of the exponential function with 48 bit of precision.

```

static uint64_t approx_exp(const uint64_t x) {
    int64_t result;
    result = -0x0000B6C6340925AELL;
    result = ((smulh48(result, x) + (1LL << 2)) >> 3)
        + 0x0000B4BD4DF85227LL;
    result = ((smulh48(result, x) + (1LL << 2)) >> 3)
        - 0x0000887F727491E2LL;
    result = ((smulh48(result, x) + (1LL << 1)) >> 2)
        + 0x0000AAAA643C7E8DLL;
    result = ((smulh48(result, x) + (1LL << 1)) >> 2)
        - 0x0000AAAAA98179E6LL;
    result = ((smulh48(result, x) + 1LL) >> 1)
        + 0x0000FFFFFFFFB2E7ALL;
    result = ((smulh48(result, x) + 1LL) >> 1)
        - 0x0000FFFFFFFFF85FLL;
    result = ((smulh48(result, x)))
        + 0x0000FFFFFFFFF85FLL;
    return result;
}

```

Finalization. If the sample is accepted eventually, it is (implicitly) scaled by the factor 2^{-76} to obtain a continuous sample from the standard normal distribution. Moreover, we only need to store the upper 64 bits of the sample and round off the rest.

In summary, each Gaussian sample candidate requires **72 bit** randomness for the lower part of the candidate (y), **16 bit** randomness for the CDT sampling, and **48 bit** randomness for rejecting the candidate conditionally according to the output of the exponential. This results in a vast randomness demand per hyperball sample, and explains the dominance of hashing in the cycle count performance.

5.1.2 Approximating the Inverse of the Square Root

To turn the vector of standard normal distributed variates into a hyperball sample candidate, we must compute its norm. For this, we accumulate all squared samples and approximate the inverse of the square root of the accumulated value. The approximation result is then multiplied by the constant $r' + \sqrt{nm}/(2N)$, which yields the scaling factor that is multiplied to each Gaussian sample. For the inverse square root, we deploy Newton's method, which is a well-known technique for that purpose. However, Newton's method requires a starting approximation that is, with each iteration, turned into a better approximation. Fortunately, we know that the sum of $nm + 2$ independent squared standard normal variables follows a χ^2 distribution with expected value $nm + 2$. Hence, the starting approximation can be fixed and precomputed as $1/\sqrt{nm + 2}$. The number of iterations for a targeted precision can be determined experimentally. Therefore, we performed the approximation for the first input values that have negligible probabilities either for the cumulative distribution function of $\chi^2(nm + 2)$ or its survival function, and checked how many iterations are required to still reach reasonable precision.

Scheme	cut_h	$offset_h$	$ \{S_h(n)\} $	cut_{z_1}	$ \{S_{z_1}(n)\} $	$base_h$	$base_{z_1}$
HAETAE-120	6	239	13	6	13	7	132
HAETAE-180	8	235	17	8	17	127	376
HAETAE-260	16	471	33	9	19	358	501

Table 2: Symbol mapping and encoding size parameters.

5.2 Signature Packing and Sizes

The last step of the signature generation is to compress and pack the elements of the signature. A packed HAETAE signature consists of the challenge c , the low bits of \mathbf{z}_1 (LN coefficients), the high bits of \mathbf{z}_1 and \mathbf{h} (KN coefficients). Because the distributions of the values in the high bits of \mathbf{z}_1 and the coefficients in \mathbf{h} are both very dense, we can compress both polynomial vectors with encoding. Before compressing the values, we map them to a smaller symbol space and thereby reject the very unlikely values and the corresponding signatures. For \mathbf{h} we cut out most of the values in the middle of the range, for HAETAE-120 this reduces the size of the symbol space from 252 to 13.

$$S_h(n) = \begin{cases} n, & \text{for } 0 \leq n \leq cut_h \\ \perp, & \text{for } cut_h < n \leq cut_h + offset_h \\ n - offset_h, & \text{for } cut_h + offset_h < n \end{cases}$$

For the high bits of \mathbf{z}_1 we tail-cut the distribution left and right of the center at 0, and then shift the remaining values to the non-negative range beginning at 0. For HAETAE-120 this reduces the size of the symbol space from 37 to 13.

$$S_{z_1}(n) = \begin{cases} \perp, & \text{for } n < -cut_{z_1} \\ n + cut_{z_1}, & \text{for } -cut_{z_1} \leq n \leq cut_{z_1} \\ \perp, & \text{for } cut_{z_1} < n \end{cases}$$

The parameters for these mappings are defined in Table 2. At the signature verification, the mapping must be reverted after decoding the compressed symbols.

The reason for these mappings is mainly to get significantly smaller precomputation tables for the rANS encoding and decoding. Also, all symbols can now be represented with 8-bits, which simplifies the rANS implementation. Furthermore, for the high bits of \mathbf{z}_1 , a mapping to non-negative values is necessary to be able to use rANS encoding. The effect on the resulting signature size is insignificant.

The size of the compressed high bits of \mathbf{z}_1 and \mathbf{h} varies and must be included in the signature, to allow a correct unpacking and decoding. The size of one compressed polynomial vector is often more than 255 bytes, and can thus not be expressed by one byte. Its variance however, is limited, and thus we encode the size the compressed high bits of \mathbf{z}_1 and \mathbf{h} as positive offset to a fixed base value. This unsigned offset value fits into one byte in most of the cases, if not, the signature gets rejected. The base values can be found in Table 2.

The final signature is then built as following: The first 32 bytes contain the seed for the challenge polynomial c . Following, we have LN bytes for the low bits of \mathbf{z}_1 . The next first byte consists of the offset to the base size for the encoding of the high bits of \mathbf{z}_1 and the second byte is the offset for \mathbf{h} . Then we have the encoding of the high bits of \mathbf{z}_1 and directly

Scheme	Lvl.	vk	Signature	Sum	Secret key
HAETAE-120	2	992	1,474	2,466	1,408
HAETAE-180	3	1,472	2,349	3,821	2,112
HAETAE-260	5	2,080	2,948	5,028	2,752
Dilithium-2	2	1,312	2,420	3,732	2,528
Dilithium-3	3	1,952	3,293	5,245	4,000
Dilithium-5	5	2,592	4,595	7,187	4,864
Falcon-512	1	897	666	1,563	1,281
Falcon-1024	5	1,792	1,280	3,072	2,305

Table 3: NIST security level, signature and key sizes (bytes) of HAETAE, Dilithium, and Falcon.

afterwards the encoding of \mathbf{h} , both with varying sizes, which are indicated beforehand. Lastly, the signature is padded with zero bytes to reach the fixed signature size, if any bytes remain. Signatures that would exceed the fixed limit get rejected.

To prevent signature forgeries, during signature unpacking and decoding multiple sanity checks have to be performed: the zero padding must be correct, the decoding must not fail and decode the expected number of coefficients while using exactly the amount of bytes indicated with the offset. Furthermore, rANS decoding must end with the fixed predefined start value to be unique. Our rANS encoding is based on an implementation by Fabian Giesen [17].

To set the fixed signature size as reported in Table 3, we evaluated the distribution empirically and determined a threshold that requires a rejection in less than 0.1% of the cases.

In Table 3 we compare the signature and key sizes of HAETAE, Dilithium, and Falcon. The verification keys in HAETAE are 20% (HAETAE-260) to 25% (HAETAE-120 and HAETAE-180) smaller, than their counterparts in Dilithium. The advantage of the hyperball sampling manifests itself in the signature sizes, HAETAE has 29% to 39% smaller signatures than Dilithium. Less relevant are the secret key sizes, that are almost half the size in HAETAE compared to Dilithium. A direct comparison to Falcon for the same claimed security level is only possible for the highest parameter set, Falcon-1024 has a signature of less than half the size compared to HAETAE-260, and its verification key is about 14% smaller.

5.3 Performance Reference Implementation

We developed an unoptimized, portable and constant-time implementation in C for HAETAE and report median and average cycle counts of one thousand executions for each parameter set in Table 4. Due to the key and signature rejection steps, the median and average values for key generation and signing respectively differ clearly, whereas the two values are much closer for the verification.

For a fair comparison, we also performed measurements on the same system with identical settings of the reference implementation of Dilithium⁴ and the implementation

⁴<https://github.com/pq-crystals/dilithium/tree/master/ref>

Scheme		KeyGen	Sign	Verify
HAETAE-120	<i>med</i>	1,403,402	6,039,674	376,486
	<i>ave</i>	1,827,567	9,458,682	376,631
HAETAE-180	<i>med</i>	2,368,038	9,161,312	691,652
	<i>ave</i>	3,448,185	11,611,868	692,014
HAETAE-260	<i>med</i>	3,101,280	11,444,678	895,098
	<i>ave</i>	4,088,383	17,229,712	896,622
Dilithium-2	<i>med</i>	343,222	1,191,218	376,008
	<i>ave</i>	343,639	1,527,406	376,543
Dilithium-3	<i>med</i>	630,170	2,061,816	612,538
	<i>ave</i>	630,607	2,603,237	612,852
Dilithium-5	<i>med</i>	945,776	2,522,834	987,154
	<i>ave</i>	949,662	3,080,734	988,250
Falcon-512	<i>med</i>	53,778,476	17,332,716	103,056
	<i>ave</i>	60,301,272	17,335,484	103,184
Falcon-1024	<i>med</i>	154,298,384	38,014,050	224,378
	<i>ave</i>	178,516,059	38,009,559	224,840

Table 4: Reference implementation speeds. Median and average cycle counts of 1000 executions for HAETAE, Dilithium, and Falcon. Cycle counts were obtained on one core of an Intel Core i7-10700k, with TurboBoost and hyperthreading disabled.

with emulated floating-point operations, and thus also fully portable, of Falcon⁵, as given in Table 4. The performance of the signature verification for HAETAE is very close to Dilithium throughout the parameter sets. HAETAE-180 verification is 13% slower than its counter-part, HAETAE-260 on the other hand, is 9% faster than the respective Dilithium parameter set. For key generation and signature computation, our current implementation of HAETAE is clearly slower than Dilithium. We measure a slowdown of factors three to five. In comparison to Falcon, however, HAETAE has 38-50 times faster key generation and around three times faster signing speed. For the verification, Falcon outperforms both Dilithium and HAETAE by roughly a factor of four.

A closer look at the key generation reveals that the complex Fast Fourier Transformation, that is required for the rejection step, is with 53% by far the most expensive operation and a sensible target for optimized implementations.

Profiling the signature computation reveals that the slowdown compared to Dilithium is mainly caused by the sampling from a hyperball, where about 80% of the computation time is spent. The hyperball sampling itself is dominated by the generation of randomness, which we derive from the extendable output function SHAKE256 [12], which is also used in the Dilithium implementation. Almost 60% of the signature computation time is spent in SHAKE256.

Based on the profiling and benchmarking of subcomponents, we estimate the performance of a randomized HAETAE implementation with pre-computation. The generic version, which is independent of the key, would already achieve a speedup of a factor five for its online signing, because the expensive hyperball sampling can be done

⁵<https://falcon-sign.info/falcon-round3.zip>

offline. For the pre-computation variant with a designated signing key, additionally, a lot of matrix-vector multiplications and therefore most of the transformations from and to the NTT domain, can be precomputed. We estimate about 12% of the full deterministic signing running time, for the online signing in this case.

References

- [1] Andryscio, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: 2015 IEEE Symposium on Security and Privacy. pp. 623–639. IEEE (2015)
- [2] Bai, S., Galbraith, S.D.: An improved compression technique for signatures based on learning with errors. In: Benaloh, J. (ed.) CT-RSA 2014. LNCS, vol. 8366, pp. 28–47. Springer, Cham (Feb 2014). https://doi.org/10.1007/978-3-319-04852-9_2
- [3] Barbosa, M., Barthe, G., Doczkal, C., Don, J., Fehr, S., Grégoire, B., Huang, Y.H., Hülsing, A., Lee, Y., Wu, X.: Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part V. LNCS, vol. 14085, pp. 358–389. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38554-4_12
- [4] Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Rossi, M., Tibouchi, M.: GALACTICS: Gaussian sampling for lattice-based constant- time implementation of cryptographic signatures, revisited. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2147–2164. ACM Press (Nov 2019). <https://doi.org/10.1145/3319535.3363223>
- [5] Chailloux, A., Loyer, J.: Lattice sieving via quantum random walks. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part IV. LNCS, vol. 13093, pp. 63–91. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92068-5_3
- [6] Dachman-Soled, D., Ducas, L., Gong, H., Rossi, M.: LWE with side information: Attacks and concrete security estimation. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 329–358. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56880-1_12
- [7] Devevey, J., Fallahpour, P., Passelègue, A., Stehlé, D.: A detailed analysis of Fiat-Shamir with aborts. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part V. LNCS, vol. 14085, pp. 327–357. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38554-4_11
- [8] Devevey, J., Fawzi, O., Passelègue, A., Stehlé, D.: On rejection sampling in Lyubashevsky’s signature scheme. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part IV. LNCS, vol. 13794, pp. 34–64. Springer, Cham (Dec 2022). https://doi.org/10.1007/978-3-031-22972-5_2
- [9] Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal Gaussians. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 40–56. Springer, Berlin, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40041-4_3
- [10] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A lattice-based digital signature scheme. IACR TCHES **2018**(1), 238–268 (2018). <https://doi.org/10.13154/tches.v2018.i1.238-268>, <https://tches.iacr.org/index.php/TCHES/article/view/839>

- [11] Duda, J.: Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding (2014), arXiv preprint, available at <https://arxiv.org/abs/1311.2540>
- [12] Dworkin, M.J.: SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS 202 (2015)
- [13] Espitau, T., Fouque, P.A., Gérard, B., Tibouchi, M.: Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1857–1874. ACM Press (Oct / Nov 2017). <https://doi.org/10.1145/3133956.3134028>
- [14] Espitau, T., Fouque, P.A., Gérard, F., Rossi, M., Takahashi, A., Tibouchi, M., Wallet, A., Yu, Y.: Mitaka: A simpler, parallelizable, maskable variant of falcon. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part III. LNCS, vol. 13277, pp. 222–253. Springer, Cham (May / Jun 2022). https://doi.org/10.1007/978-3-031-07082-2_9
- [15] Espitau, T., Tibouchi, M., Wallet, A., Yu, Y.: Shorter hash-and-sign lattice-based signatures. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part II. LNCS, vol. 13508, pp. 245–275. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15979-4_9
- [16] Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al.: Falcon: Fast-fourier lattice-based compact signatures over NTRU. Submission to the NIST’s post-quantum cryptography standardization process **36**(5) (2018)
- [17] Giesen, F.: Interleaved entropy coders. arXiv preprint arXiv:1402.3392 (2014)
- [18] Li, R.: Near optimality of chebyshev interpolation for elementary function computations. IEEE Trans. Computers **53**(6), 678–687 (2004). <https://doi.org/10.1109/TC.2004.15>, <https://doi.org/10.1109/TC.2004.15>
- [19] Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 598–616. Springer, Berlin, Heidelberg (Dec 2009). https://doi.org/10.1007/978-3-642-10366-7_35
- [20] Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 738–755. Springer, Berlin, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_43
- [21] Prest, T.: A key-recovery attack against mitaka in the t -probing model. In: Boldyreva, A., Kolesnikov, V. (eds.) PKC 2023, Part I. LNCS, vol. 13940, pp. 205–220. Springer, Cham (May 2023). https://doi.org/10.1007/978-3-031-31368-4_8
- [22] Rossi, M.: Extended Security of Lattice-Based Cryptography. Ph.D. thesis, École Normale Supérieure de Paris (2020), <https://www.di.ens.fr/~mrossi/docs/thesis.pdf>

- [23] Voelker, A.R., Gosmann, J., Stewart, T.C.: Efficiently sampling vectors and coordinates from the n -sphere and n -ball. Centre for Theoretical Neuroscience-Technical Report (01 2017). <https://doi.org/10.13140/RG.2.2.15829.01767/1>

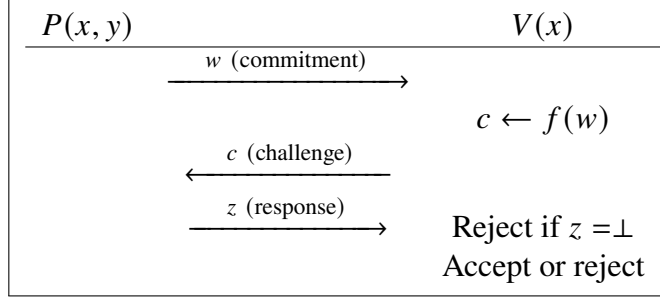


Figure 17: Interaction between P and V . We let $\langle P(x, y) \leftrightarrow V(x) \rangle_{f(\cdot)}$ denote the transcript (w, c, z) .

A Cryptographic Reminders

In this section, we recall the definition of a canonical identification scheme, which is the main building brick of HAETAE and the Fiat-Shamir with aborts transform, which allows one to turn a canonical identification scheme into a signature scheme.

A.1 Canonical Identification Schemes

We start by defining a canonical identification scheme.

Definition 23 (Canonical Identification Scheme with Aborts (CID)). A canonical identification scheme ID for a relation R is a three rounds interactive protocol between a prover P and a verifier V , as defined in Figure 17. The challenge is generated from $f(w)$ set as the distribution $U(C)$ for some challenge set C . The prover holds a pair $(x, y) \in R$ while the verifier only has x , where the pair (x, y) was generated by a PPT algorithm Gen , called an instance generator. The event “ $z = \perp$ ” is called an abort, and its probability β is called the probability of aborting.

The canonical identification schemes considered in this work are such that given c and z , there is only one w such that V accepts, which can be efficiently computed from c and z .

We need the following flavor of zero knowledge, denoted perfect accepting honest verifier zero-knowledge.

Definition 24 (Perfect Accepting Honest Verifier Zero-Knowledge). Let ID be a canonical identification scheme. It satisfies paHVZK if there exists a PPT simulator Sim such that the output distribution of (w, c, z) resulting from the interaction $\langle P(x, y) \leftrightarrow V(x) \rangle_{U(C)}$ conditioned on $z \neq \perp$ and (w', c', z') generated by $\text{Sim}(x)$ for any (x, y) generated by $\text{Gen}(1^\lambda)$ are identical.

In the case of the HAETAE CID, the challenge c can be sampled uniformly from the challenge space C and passed over as input to the simulator Sim .

To avoid using the same commitment twice, we require the min-entropy of the commitment to be high.

KeyGen(1^λ):	Sign(sk, μ):	Verify(vk, μ , σ):
1: $(x, y) \leftarrow \text{Gen}(1^\lambda)$	1: While $z = \perp$	1: Parse $\sigma = (c, z)$
2: $(\text{vk}, \text{sk}) = (x, (x, y))$	2: $(w, c, z) \leftarrow \langle P(x, y) \leftrightarrow V(x) \rangle_{H(\cdot, \mu)}$	2: Recover w from σ
3: return (vk, sk)	3: return $\sigma = (c, z)$	3: return $c = H(w, \mu)$

Figure 18: Fiat-Shamir with Aborts tranform FS[ID, H].

Definition 25 (Commitment Min-Entropy). For $\alpha \geq 0$, we say that an identification scheme ID with instance generator Gen has commitment min-entropy α if $H_\infty[w|(w, c, z) \leftarrow \langle P(x, y) \leftrightarrow V(x) \rangle_{U(C)}] \geq \alpha$, for all $(x, y) \leftarrow \text{Gen}(1^\lambda)$.

Finally we need the notion of computational unique response to enable the strong unforgeability property of the final signature scheme.

Definition 26 (Computational Unique Response). Let ID be a canonical identification scheme with instance generator Gen. The advantage $\text{Adv}_{\text{ID}}^{\text{CUR}}(\mathcal{A})$ of an adversary \mathcal{A} against the CUR property of ID is its probability of computing (w, c, z, z') such that $V(x)$ accepts both (w, c, z) and (w, c, z') and $z \neq z'$ on input x generated from $(x, y) \leftarrow \text{Gen}(1^\lambda)$.

A.2 The Fiat-Shamir Transform

We now briefly recall how to turn a canonical identification scheme into a full-fledge signature scheme. To do so, the signing scheme runs the interactive protocol and outputs the transcript minus the commitment. The challenge is however generated by hashing the message along with the commitment to prevent an adversary from tampering with it.

We recall the assumptions necessary to show the unforgeability of the resulting signature in the following theorem. The following theorem works in the case of quantum adversaries working in the QROM, but tighter statements can be obtained when restricting to classical adversaries in the ROM.

Theorem 27 (Adapted from [3, Theorem 2]). *Let ID be a canonical identification scheme with α commitment min-entropy and that satisfies paHVZK with probability of aborting β . For any quantum adversary \mathcal{A} against the UF-CMA security of FS[ID, H] in the QROM making Q_s signature queries and Q_h hash queries, there exists an adversary \mathcal{B} against the UF-NMA security of FS[ID, H] in the QROM such that:*

$$\text{Adv}_{\text{FS}[\text{ID}, H]}^{\text{UF-CMA}}(\mathcal{A}) \leq \text{Adv}_{\text{FS}[\text{ID}, H]}^{\text{UF-NMA}}(\mathcal{B}) + \frac{2^{-\alpha/2+1} Q_s}{1 - \beta} \sqrt{Q_h + 1 + \frac{Q_s}{1 - \beta}} + 2^{-\alpha/2+1} (Q_h + 1) \sqrt{\frac{Q_s}{1 - \beta}}.$$

Furthermore, if \mathcal{A} is an adversary against the sUF-CMA security, there exists an adversary \mathcal{B}' against the CUR property of ID such that the previous bound holds by adding $\text{Adv}_{\text{ID}}^{\text{CUR}}(\mathcal{B}')$ on the right-hand side.

The strong unforgeability statement comes from the fact that in the last game of their proof, the reduction fails if and only if the forgery uses the same commitment (and thus reprogrammed challenge) than the signature query. As this requires a different answer, this corresponds to an attack against the computational unique response property of the canonical identification scheme.

B Additional Proofs

B.1 Useful Lemma

We will rely on the following claim.

Lemma 28. *Let n be the degree of \mathcal{R} . Let $m, N, r > 0$ and $\mathbf{v} \in \mathcal{R}^m$. Then the following statements hold:*

1. $|(1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)| = |\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr)|$,
2. $|\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r, \mathbf{v})| = |\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)|$,
3. $\text{Vol}(\mathcal{B}_{\mathcal{R},m}(r - \frac{\sqrt{mn}}{2})) \leq |\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)| \leq \text{Vol}(\mathcal{B}_{\mathcal{R},m}(r + \frac{\sqrt{mn}}{2}))$.

Proof. For the first statement, note that we only scaled $(1/N)\mathcal{R}^m$ and $\mathcal{B}_{\mathcal{R},m}(r)$ by a factor N . For the second statement, note that the translation $\mathbf{x} \mapsto \mathbf{x} - \mathbf{v}$ maps \mathcal{R}^m to \mathcal{R}^m .

We now prove the third statement. For $x \in \mathcal{R}^m$, we define $T_{\mathbf{x}}$ as the hypercube of $\mathcal{R}_{\mathbb{R}}^m$ centered in \mathbf{x} with side-length 1. Observe that the $T_{\mathbf{x}}$'s tile the whole space when \mathbf{x} ranges over \mathcal{R}^m (the way boundaries are handled does not matter for the proof). Also, each of those tiles has volume 1. As any element in $T_{\mathbf{x}}$ is at Euclidean distance at most $\sqrt{mn}/2$ from \mathbf{x} , the following inclusions hold:

$$\mathcal{B}_{\mathcal{R},m}\left(r - \frac{\sqrt{mn}}{2}\right) \subseteq \bigcup_{\mathbf{x} \in \mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)} T_{\mathbf{x}} \subseteq \mathcal{B}_{\mathcal{R},m}\left(r + \frac{\sqrt{mn}}{2}\right).$$

Taking the volumes gives the result. \square

B.2 Proof of Lemma 11

Proof. To ease the notation, let us use $B = r'$. Let $\mathbf{y} \in \mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2)$ and set $\mathbf{z} = \lfloor \mathbf{y} \rfloor$. Note that \mathbf{z} is sampled (before the rejection step) with probability

$$\frac{\text{Vol}(T_{\mathbf{z}} \cap \mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))}{\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr'))},$$

where $T_{\mathbf{z}}$ is the hypercube of $\mathcal{R}_{\mathbb{R}}^m$ centered in \mathbf{z} with side-length 1. By the triangle inequality, this probability is equal to $1/\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))$ when $\mathbf{z} \in \mathcal{B}_{\mathcal{R},m}(Nr')$. Hence the distribution of the output is exactly $U(\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr'))$, as each element is sampled with equal probability and as the algorithm almost surely terminates (its runtime follows a geometric law of parameter the rejection probability).

It remains to consider the acceptance probability.

$$\frac{\sum_{\mathbf{y} \in \mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr')} \text{Vol}(T_{\mathbf{y}} \cap \mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))}{\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))}.$$

By the triangle inequality and Lemma 28, it is

$$\frac{|\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr')|}{\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))} \geq \left(\frac{Nr' - \sqrt{mn}/2}{Nr' + \sqrt{mn}/2} \right)^{mn}.$$

Note that by our choice of N , this is $\geq 1/M_0$. \square

B.3 Proof of Lemma 12

Proof. Figure 6 is the bimodal rejection sampling algorithm applied to the source distribution $U((1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r'))$ and target distribution $U((1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r))$ (see, e.g., [8]). It then suffices that the support of the bimodal shift of the source distribution by \mathbf{v} contains the support of the target distribution. It is implied by $r' \geq \sqrt{r^2 + t^2}$.

We now consider the number of expected iterations, i.e., the maximum ratio between the two distributions. To guide the intuition, note that if we were to use continuous distributions, the acceptance probability $1/M'$ would be bounded by $1/M$. In our case, the acceptance probability can be bounded as follows (using Lemma 28):

$$\begin{aligned} \frac{1}{M'} &= \frac{|(1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)|}{2|(1/N)\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r')|} = \frac{|\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr)|}{2|\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(Nr')|} \\ &\geq \frac{\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr - \sqrt{mn}/2))}{2\text{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2))} \\ &= \frac{1}{2} \left(\frac{Nr - \sqrt{mn}/2}{Nr' + \sqrt{mn}/2} \right)^{mn}. \end{aligned}$$

It now suffices to bound the latter term from below by $1/(cM) = 1/(2c(r'/r)^{mn})$. This inequality is equivalent to:

$$c \geq \frac{1}{2} \cdot \left(\frac{r}{r - \sqrt{mn}/(2N)} \right)^{mn} \cdot \left(\frac{r' + \sqrt{mn}/(2N)}{r'} \right)^{mn},$$

and to:

$$N \geq \frac{1}{c^{1/(mn)} - 1} \cdot \frac{\sqrt{mn}}{2} \left(\frac{c^{1/(mn)}}{r} + \frac{1}{r'} \right),$$

which allows to complete the proof. \square

B.4 Proof of Lemma 15

Proof. By Lemma 13, there exists a unique representation

$$r = \lfloor (r + \alpha/2)/\alpha \rfloor \alpha + (r \bmod^\pm \alpha).$$

By identifying $\text{HighBits}(r, \alpha)$ and $\text{LowBits}(r, \alpha)$ in the above equation, we obtain the first result.

By definition of $\bmod^\pm \alpha$, we have the second range.

Finally, since $r \mapsto \lfloor (r + \alpha/2)/\alpha \rfloor$ is a non-decreasing function, it is sufficient to show that $\lfloor (2q - 1 + \alpha/2)/\alpha \rfloor \leq \lfloor (2q - 1)/\alpha \rfloor$. We have $(2q - 1 + \alpha/2) \leq \lfloor (2q - 1)/\alpha \rfloor \alpha + \alpha - 1$ by assumption on q . Dividing by α and taking the floor yields the result. \square

B.5 Proof of Lemma 17

Proof. Let $r \in [0, 2q - 1]$. Let r_0, r_1, r'_0 , and r'_1 defined as in Definition 16. If $r'_0 = r_0$ and $r'_1 = r_1$, the equality $r'_0 + r'_1 \cdot \alpha_h = r_0 + r_1 \cdot \alpha_h \bmod 2q$ holds vacuously.

If not, then $r'_0 = r_0 - 2$ and $r'_1 = r_1 - 2(q - 1)/\alpha_h$ and $r'_0 + r'_1\alpha_h = r_0 + r_1\alpha_h - 2q$. By Lemma 15, we get the first equality.

The second property stems from the second property in Lemma 15. The modifications to r_0 make r'_0 lie in the range $[-\alpha_h/2 - 2, \alpha_h/2)$.

The last property stems from the third property in Lemma 15 and the fact that if $r_1 = m$, then we have $r'_1 = 0$.

□

C Additional Implementation Specification

Algorithm 1 describes how to implement the unpacking of $\hat{\mathbf{A}}$, and in Algorithm 2 we demonstrate how to apply the CRT.

Algorithm 1 Unpacking routine for $\hat{\mathbf{A}}$.

```

unpackAd(seedA,  $\psi$ )
1: if  $d = 0$  then
2:    $\hat{\mathbf{A}}_{\text{gen}} := \text{expandA}_d(\text{seed}_A)$ 
3:    $\hat{\mathbf{b}} := \psi$ 
4: else
5:    $(\mathbf{a}_{\text{gen}}, \hat{\mathbf{A}}_{\text{gen}}) := \text{expandA}_d(\text{seed}_A)$ 
6:    $\hat{\mathbf{b}} := 2 \cdot \text{NTT}(\mathbf{a}_{\text{gen}} - \psi \cdot 2^d) \bmod q$ 
7: return  $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times \ell} := (\hat{\mathbf{b}} \mid 2 \cdot \hat{\mathbf{A}}_{\text{gen}}) \bmod q$ 

```

Algorithm 2 Mapping from $(\mathcal{R}_q^k, \mathcal{R}_q)$ to \mathcal{R}_{2q}^k

```

fromCRT( $\mathbf{w}, x$ )
1: parse  $\mathbf{w}$  as vector of integers  $\bar{\mathbf{w}}$  of size  $kn$ 
2: parse  $x$  as vector of integers  $\bar{x}$  of size  $n$ 
3: for  $i := 0$  to  $n - 1$  do
4:   if  $\text{LSB}(\bar{x}_i) = \text{LSB}(\bar{\mathbf{w}}_i)$  then                                 $\triangleright$  Implement in constant time.
5:      $\bar{\mathbf{w}}'_i := \bar{\mathbf{w}}_i$ 
6:   else
7:      $\bar{\mathbf{w}}'_i := \bar{\mathbf{w}}_i + q$ 
8: for  $j := 1$  to  $k - 1$  do
9:   for  $i := 0$  to  $n - 1$  do
10:    if  $\text{LSB}(\bar{\mathbf{w}}_{nj+i}) = 0$  then                                 $\triangleright$  Implement in constant time.
11:       $\bar{\mathbf{w}}'_{nj+i} := \bar{\mathbf{w}}_{nj+i}$ 
12:    else
13:       $\bar{\mathbf{w}}'_{nj+i} := \bar{\mathbf{w}}_{nj+i} + q$ 
14: arrange  $\bar{\mathbf{w}}'$  to  $\mathbf{w}'$ , an element in  $\mathcal{R}_{2q}^k$ 
15: return  $\mathbf{w}'$ 

```
