HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures

Jung Hee Cheon[1,2], Hyeongmin Choe[1], Julien Devevey, Tim Güneysu[3,4], Dongyeon Hong, Markus Krausz[3], Georg Land[3], Marc Möller[3], Junbum Shin[2], Damien Stehlé[2] and MinJune Yi[1]

[1] Seoul National University, Seoul, Republic of Korea
jhcheon@snu.ac.kr,sixtail528@snu.ac.kr,yiminjune@snu.ac.kr

[2] CryptoLab Inc., Seoul, Republic of Korea damien.stehle@cryptolab.co.kr

[3] Ruhr University Bochum, Bochum, Germany firstname.lastname@rub.de,mail@georg.land

[4] DFKI, Bremen, Germany

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.

Keywords: Signature, Fiat-Shamir, Lattice-based Cryptography, Bimodal Distribution

1 Introduction

The rise of quantum computing has brought up – among others – the necessity of new, post-quantum digital signature schemes. In the standardization process of post-quantum cryptography by the American National Institute of Standards and Technology (NIST), the lattice-based schemes Falcon [FHK+18] and Dilithium [DKL+18] have already been announced as future standards, and another 40 new candidates are on-ramp for an additional process. The critical challenge in developing lattice-based digital signatures lies in finding a balance between security and practicality: while developing secure schemes against a wide range of attacks is essential, it is also vital to ensure they are practical for real-world applications. This challenge becomes even more critical with the increasing prevalence of embedded devices and the Internet of Things (IoT). Both technologies have become ubiquitous, from home appliances to medical devices connected to the internet.

In particular, this leads to two practical requirements:

1. The verification key and signature sizes must be as small as possible since both are frequently transmitted. Specifically, it is helpful if the signature is small enough to be sent in only one UDP or TCP datagram, as this minimizes the need for

Table 1: NIST security level, signature size, verification key size, and implementation security, with respect to constant-time and masking of selected signature schemes

Scheme	Lvl.	Sig.	vk	Const.-time.	Maskable
Falcon-512	1	666B	897B	✓ [Por19]	✗ [Pre23]
Dilithium-2	2	2,420B	1,312B	✓ [DKL+18]	✓ [MGTF19]
HAETAE-120	2	1,474B	992B	✓	✓

packet fragmentation. The importance of the signature and verification key sizes for communication protocols has been highlighted already in multiple evaluations [Wes21, PST20, GS23]. Paquin et al. [PST20] observe for TLS, that fragmentation over many packets has a significant performance impact for network links with non-ideal packet loss rates. Benchmarking DNSSEC [GS23] revealed, that the smaller signatures of Falcon lead to faster resolution times in comparison to Dilithium in most scenarios, although the signature computation and verification is much faster with Dilithium compared to Falcon.

2. The secret-dependent operations such as key generation and message signing must be easy to protect against implementation attacks. This is essential in embedded use cases like the IoT, where attackers have physical access and can measure power consumption or electromagnetic emanation [KA21], additionally to the timing behaviour [Sch00], which is also exploitable from remote.

In this context, Falcon fulfills the first requirement very well, but efforts for making it to satisfy the second requirement, namely Mitaka [EFG+22], were recently broken [Pre23]. Dilithium, on the other hand, focuses on being easy to implement and protecting against side-channel attacks. However, this comes at the sacrifice of larger signatures and verification keys, which, for example, do not allow a signature to fit in one UDP datagram. We summarize this discussion in Table 1 and compare the two with HAETAE.

Contribution. We present HAETAE[1], a lattice-based digital signature scheme that improves over Dilithium by up to 39 % smaller signature and key sizes while being similarly easy to protect against physical attacks. Its quantum security is based on the hardness of the module versions of the lattice problems LWE and SIS [BGV12, LS15], in the Quantum Random Oracle Model (QROM). The scheme design follows the "Fiat-Shamir with Aborts" paradigm [Lyu09, Lyu12], 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.

HAETAE is in part inspired from Dilithium, a post-quantum "Fiat-Shamir with Aborts" signature scheme, notably concerning the use of the module LWE and SIS assumptions. HAETAE differs from Dilithium in two major aspects: (i) we use a bimodal distribution for the rejection sampling, like in the BLISS signature scheme [DDLL13], instead of a "unimodal" distribution like Dilithium, (ii) we sample from and reject to hyperball uniform distributions, instead of discrete hypercube uniform distributions. The design departs from BLISS, as HAETAE relies on module lattice problems while BLISS relies either on assumptions on unstructured lattices or the NTRU assumption [HPS98]: this leads us to introduce a new key generation algorithm. A further difference is that BLISS involves on discrete Gaussian distributions whereas HAETAE considers hyperball uniform distributions as suggested in [DFPS22]. This choice allows for simple rejection sampling without transcendental function computation and tail-cutting, while retaining the signature compactness enabled by Gaussian distributions.

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

Table 2: Relative comparison between HAETAE, Dilithium, and Falcon. The security levels are given in the parameter sets instead of their name. The percentages are the ratio of their sizes and the execution times. The execution time is measured as the median cycle counts among 1000 executions, obtained on one core of an Intel Core i7-10700k, with TurboBoost and hyperthreading disabled.

Parameter set	Sig. size	vk size	KeyGen	Sign	Verify
HAETAE-2 / Dilithium-2	61%	76%	409%	507%	100%
HAETAE-3 / Dilithium-3	71%	75%	376%	444%	113%
HAETAE-5 / Dilithium-5	64%	80%	328%	454%	91%
HAETAE-2 / Falcon-1	221%	111%	3%	35%	365%
HAETAE-5 / Falcon-5	230%	116%	2%	30%	399%

HAETAE benefits from several novel improvements in the key generation algorithm. 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. Furthermore, we introduce to the bimodal setting a verification key truncation with the same objective as Dilithium's. A direct adaptation would lead to large bounds for the verification algorithm and degraded security. Instead, we compensate for the verification key truncation by correcting the signing key accordingly. It increases the magnitude of the signing key, but by a much smaller amount than the naive approach.

For the signing algorithm, we adapt Dilithium's signature compression so that it is compatible with our module lattices key generation algorithm, by taking into account the residues modulo 2. Further, we apply the signature encoding technique from [ETWY22] to hyperball uniform distributions. The main novelty in the signing algorithm is a detailed description of a fixed-point arithmetic algorithm for sampling uniformly in a hyperball, which was left open in [DFPS22]. The discretization leads to numerical errors: we bound them and bound their effect on the scheme security.

Implementation and Performance. We propose three parameter sets with NIST security levels 2, 3 and 5. Each parameter set of HAETAE has 20-25% smaller verification key size and 29-39% shorter signatures than its counterpart in Dilithium. Based on our portable and constant-time reference implementation of HAETAE, the verification process is as fast as Dilithium's, while the resulting key generation and signing algorithm are up to five times slower than Dilithium's. Up to 80% of the signing time is consumed by the hyperball sampling. Thus, any improvement to this sampling would contribute greatly to the efficiency of HAETAE, an independent speedup to further optimizations. Nonetheless, our benchmarks indicated signing with HAETAE is still around three times faster than with Falcon (portable Falcon with emulated floating-point operations). We summarize the comparison results in Table 2.

We provide a detailed, implementation-oriented specification using Chinese Remainder Theorem (CRT) and Number-Theoretic Transform (NTT), which enables efficient implementation of HAETAE (Section 5). We additionally developed an optimized version using AVX2 instructions (Section 6), and an implementation for the Cortex-M4 (Section 7), where we explore stack reduction techniques.

Moreover, we observe that masking HAETAE against physical attacks is only slightly more complex than masking Dilithium, based on the similarity of the scheme design and the use of fixed-point arithmetic. One of the conceptual differences between HAETAE and Falcon (and their variants) regarding physical attacks is that HAETAE only needs Gaussian samples for secret-independent centers and standard deviations.

Finally, we note that like other Fiat-Shamir signatures, such as Schnorr signatures [Sch90], the randomized signing of HAETAE can take advantage of pre-computations. By sampling from the hyperball and pre-computing the message-independent components offline, the online signing phase of HAETAE is cut by factor five.

Our code is publicly available.

Related Work. An alternative approach to avoid leakage during the rejection step in Fiat-Shamir signatures based on lattices is to remove it altogether. A first approach is to flood what depends on the signature key in signatures by a much larger quantity. As shown in [ASY22], relying on the Rényi divergence for the security analysis allows to limit the amount of flooding. A concrete instantiation was recently proposed in [dPEK$^+$]. This however results in signature sizes that are much higher than ours. A second approach was recently given in [DPS23], which uses Gaussian convolutions to obtain signatures that can be simulated, without flooding nor rejection sampling. However, signing is more complex as it relies on sampling from large-dimensional integer Gaussian distribution with non-diagonal covariance matrices. Also, the signature sizes provided in [DPS23] are worse than ours for the smallest parameter set, and only marginally smaller for larger parameter sets. An extensive comparison of the recent lattice signatures can be found in Appendix A.

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 $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ 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 $\mathbf{y} = (\sum_{0 \le i < n} y_i\ x^i, \cdots, \sum_{0 \le i < n} y_{nk-n+i}\ x^i)^\top \in \mathcal{R}^k$ (or $\mathcal{R}_{\mathbb{R}}^k$), we define its ℓ_2-norm as the ℓ_2-norm of the corresponding "flattened" vector $\|\mathbf{y}\|_2 = \|(y_0, \cdots, y_{nk-1})^\top\|_2$.

Let $\mathcal{B}_{\mathcal{R},m}(r, \mathbf{c}) = \{\mathbf{x} \in \mathcal{R}_{\mathbb{R}}^m | \|\mathbf{x} - \mathbf{c}\|_2 \le r\}$ 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 $\mathbf{x} \mapsto \chi_X(\mathbf{x})/\mathsf{Vol}(X)$ as a probability density, where χ_X is the indicator function of X and $\mathsf{Vol}(X)$ 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 $\mathsf{LSB}(r)$. We naturally extend this to integer polynomials and vectors of integer polynomials, by applying it component-wise.

2.2 Signatures

We briefly recall the formalism of digital signatures.

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

- $\mathsf{KeyGen} : 1^\lambda \to (\mathsf{vk}, \mathsf{sk})$ outputs a verification key vk and a signing key sk;
- $\mathsf{Sign} : (\mathsf{sk}, \mu) \to \sigma$ takes as inputs a signing key sk and a message μ and outputs a signature σ;
- $\mathsf{Verify} : (\mathsf{vk}, \mu, \sigma) \to b \in \{0, 1\}$ 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 $(\mathsf{vk}, \mathsf{sk})$ in the range of KeyGen and μ,

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

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 $\mathsf{sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is said to be (T, δ)-$\mathsf{UF\text{-}CMA}$ 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

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

where the randomness is taken over the random coins of $\mathcal{A}$ and $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. 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 $\mathsf{Adv}^{\mathsf{UF\text{-}CMA}}_{\mathsf{sig}}(\mathcal{A})$. If $\mathcal{A}$ does not output anything, then it automatically fails.

Existential unforgeability against no-message attack, denoted by $\mathsf{UF\text{-}NMA}$ is defined similarly except that the adversary is not allowed to query any signature per message.

2.3 Lattice Assumptions

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

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

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

Definition 4 (Search-$\mathsf{MSIS}_{n,q,k,\ell,\beta}$)**.** 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 $\mathcal{A}$ solving the search-$\mathsf{MSIS}_{n,q,k,\ell,\beta}$ problem is

$$\mathsf{Adv}^{\mathsf{MSIS}}_{n,q,k,\ell,\beta}(\mathcal{A}) = \Pr\left[\begin{matrix} 0 < \|\mathbf{y}\|_2 < \beta \wedge \\ (\mathbf{A}|\ \mathbf{Id}_k) \cdot \mathbf{y} = 0 \mod q \end{matrix} \ \middle|\ \mathbf{A} \leftarrow \mathcal{R}_q^{k\times\ell}; \mathbf{y} \leftarrow \mathcal{A}(\mathbf{A})\right].$$

Moreover, we finally introduce a variant of the $\mathsf{SelfTargetMSIS}$ problem introduced in $\mathsf{Dilithium}$ [DKL+18], which corresponds to our setting.

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

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

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

The following classical reduction from MSIS to $\mathsf{BimodalSelfTargetMSIS}$ is very similar to the reduction from MSIS to $\mathsf{SelfTargetMSIS}$ introduced in [DKL+18] 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 1 (Classical Reduction from MSIS to $\mathsf{BimodalSelfTargetMSIS}$)**.** *Let $q > 0$ be an odd modulus, $H : \{0,1\}^* \times \mathcal{M} \to \mathcal{R}_2$ be a cryptographic hash function modeled as a random oracle and that every polynomial-time classical algorithm has a negligible advantage against $\mathsf{MSIS}_{n,q,k,\ell,\beta}$. Then every polynomial-time classical algorithm has negligible advantage against $\mathsf{BimodalSelfTargetMSIS}_{n,q,k,\ell,\beta/2}$.*

Proof sketch. Consider a $\mathsf{BimodalSelfTargetMSIS}_{n,q,k,\ell,\beta/2}$ 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 $H(\mathbf{w}_i, M_i)$ for $i = 1, \cdots, Q$ and outputs a solution $(\mathbf{y}, c, M_j)$ for some $j \in [Q]$, then we can construct an adversary $\mathcal{A}'$ for $\mathsf{MSIS}_{n,q,k,\ell,\beta}$ 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 $H(\mathbf{w}_j, M_j) = c'(\neq c)$. Then, with probability approximately $1/Q$, algorithm $\mathcal{A}$ will produce another solution $(\mathbf{y}', c', M_j)$. We then have

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

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 $\mathsf{MSIS}_{n,q,k,\ell,\beta}$ 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. □

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

Figure 1: Hyperball uniform sampling

2.4 Sampling from the Continuous Hyperball-uniform

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

Lemma 1 ([VGS17])**.** *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 1. However, to secure the HAETAE implementation, 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 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) [Dud13], that encodes the state in a natural number and thus allows faster implementations. The rANS encoding technique was recently used in [ETWY22] 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 6 (Range Asymmetric Numeral System (rANS) Coding)**.** Let $t > 0$ and $S \subseteq [0, 2^t - 1]$. Let $g : [0, 2^t - 1] \to \mathbb{Z} \cap (0, 2^t]$ such that $\sum_{x \in S} g(x) \leq 2^t$ and $g(x) = 0$ for all $x \notin S$. We define the following:

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

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

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

Lemma 2 (Adapted from [Dud13])**.** *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, ..., s_m) ∈ S^m):
1: $x_0 = 0$
2: for $i = 0, \cdots, m-1$ do
3:     $x_{i+1} = C(x_i, s_{i+1})$
4: return $x_m$

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

Figure 2: rANS encoding and decoding procedures

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

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

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

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 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 $\mathbf{As} = -\mathbf{As} \bmod p$. To generate such a pair, following [DDLL13], we choose $p = 2q$ and aim at $\mathbf{As} = q\mathbf{j} \bmod 2q$ for $\mathbf{j} = (1, 0, \ldots, 0)^\top$.

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 $\mathbf{b} = \mathbf{A}_{\mathsf{gen}}\mathbf{s}_{\mathsf{gen}} + \mathbf{e}_{\mathsf{gen}} \bmod q$, where $\mathbf{A}_{\mathsf{gen}} \hookleftarrow U(\mathcal{R}_q^{k\times(\ell-1)})$ and $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) \hookleftarrow U(S_\eta^{\ell-1} \times S_\eta^k)$. We then define $\mathbf{A} = (-2\mathbf{b} + q\mathbf{j}|\ 2\mathbf{A}_{\mathsf{gen}}|\ 2\mathbf{Id}_k) \mod 2q$ as well as $\mathbf{s}^\top = (1|\mathbf{s}_{\mathsf{gen}}^\top|\mathbf{e}_{\mathsf{gen}}^\top)$. 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 $\mathsf{Dilithium}$.

To enable the verification key truncation, we modify the key generation algorithm, as follows. We use an extra randomness $\mathbf{a}_{\mathsf{gen}} \hookleftarrow U(\mathcal{R}_q^k)$ and let $\mathbf{b}-\mathbf{a}_{\mathsf{gen}} = \mathbf{A}_{\mathsf{gen}}\mathbf{s}_{\mathsf{gen}}+\mathbf{e}_{\mathsf{gen}} \bmod q$. For any decomposiaiton $\mathbf{b} = \mathbf{b}_1 + \mathbf{b}_0$, we then define $\mathbf{A} = (2(\mathbf{a}_{\mathsf{gen}} - \mathbf{b}_1) + q\mathbf{j}|2\mathbf{A}_{\mathsf{gen}}|2\mathbf{I}_k)$ as well as $\mathbf{s}^\top = (1|\mathbf{s}_{\mathsf{gen}}^\top|(\mathbf{e}_{\mathsf{gen}} - \mathbf{b}_0)^\top)$. 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 $\mathbf{A}_{\mathsf{gen}}$ and $\mathbf{a}_{\mathsf{gen}}$. The secret key is the seed used to generate $\mathbf{s}$ and $(\mathbf{A}_{\mathsf{gen}}, \mathbf{a}_{\mathsf{gen}})$.

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 $(\mathsf{LowBits}^{\mathsf{vk}}(\mathbf{b}), \mathsf{HighBits}^{\mathsf{vk}}(\mathbf{b}))$ 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 coeffficient 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 $\|c\mathbf{s}\|_2$, 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 $\|c\mathbf{s}\|_2 \le \gamma\sqrt{\tau}$:

$$\tau \cdot \sum_{i=1}^{m} \overset{i\text{-th}}{\max_j} \|\mathbf{s}(\omega_j)\|_2^2 + r \cdot \overset{(m+1)\text{-th}}{\max_j} \|\mathbf{s}(\omega_j)\|_2^2 \le \gamma^2 n,$$

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

Lemma 3. *For any challenge $c \in \{0, 1\}^n$ with hamming weight τ and a secret $\mathbf{s} \in S_\eta^{k+\ell}$, the value $\|c\mathbf{s}\|_2^2$ is upper bounded by*

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

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 $\|c\mathbf{s}\|_2^2$ as:

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

where $\mathbf{s}(\omega_j) = (s_1(\omega_j), \cdots, s_{k+\ell}(\omega_j))$. We have that $\sum_{j=1}^n |c(\omega_j)|^2 = n\tau$ and $|c(\omega_j)|^2 = |\omega_{j,1} + \cdots + \omega_{j,\tau}|^2 \le \tau^2$. 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,

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

where σ is a permutation for the indices, we have

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

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

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

This concludes the proof. □

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 [AKM+15] 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 1 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.

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

Lemma 4. *Let n be the degree of $\mathcal{R}$, $M_0 \ge 1$, $B, m, N > 0$. At each iteration, the algorithm from Figure 3 succeeds with probability $\ge 1/M_0$ and the distribution of the output is $U(\mathcal{B}_{(1/N)\mathcal{R},m}(B))$ if we set*

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

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

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

Figure 3: Discrete hyperball uniform sampling

```
SampleBinaryChallenge$_\tau(\rho)$
// for HAETAE-120 or HAETAE-180
1: Initialize $\mathbf{c} = c_0 c_1 \ldots c_{255} = 00\ldots0$
2: for $i = 256 - \tau$ to 255 do
3:     $j \hookleftarrow \{0, \ldots, i\}$
4:     $c_i = c_j$
5:     $c_j = 1$
6: Return $\mathbf{c}$
// for HAETAE-260
1: Initialize $\mathbf{c} = c_0 c_1 \ldots c_{255} = H(\rho)$
2: if $\mathrm{wt}(c) > 128$ then
3:     $c = c \otimes 11 \cdots 1$
4: else if $\mathrm{wt}(c) = 128$ then
5:     $c = c \otimes c_0 c_0 \cdots c_0$
6: Return $\mathbf{c}$
```

Figure 4: Challenge sampling algorithm

3.3 Challenge Sampling

Challenges in HAETAE are polynomials $c \in \mathcal{R}$ with binary coefficients and exactly τ of them are nonzero. 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-120 and HAETAE-180, 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-260, 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 into 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.

3.4 Bimodal Hyperball Rejection Sampling

Recently, Devevey et al. [DFPS22] 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 5 (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 $p : \mathbb{R}^m \to \{0, 1/2, 1\}$ 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.

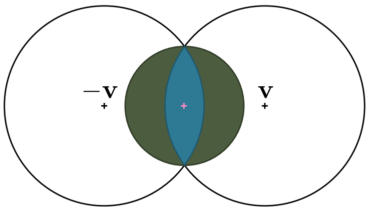


Figure 5: The HAETAE eyes

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 4 and Lemma 5, we already know the level of precision required for $\mathbf{y}$ to maintain the provable security of HAETAE.

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 [ETWY22], 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

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

B :
1: $\mathbf{z} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},m}(B'))$
2: return $\mathbf{z}$ with probability $1/M'$, else $\perp$
```

Figure 6: Bimodal hyperball rejection sampling

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 implements a trick that allow 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 6. *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 7 (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:

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

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 7. The proof is available in Appendix B.

Lemma 7. *Let α be a power of two. Let $q > 2$ be a prime with $\alpha | 2(q-1)$ and $r \in \mathbb{Z}$. Then it holds that*

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

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

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 [DKL+18, 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 $[0, 2(q-1)/\alpha_{\mathsf{h}})$. This is possible if we use the range $[-\alpha_{\mathsf{h}}/2 - 2, 0)$ to represent the integers that are close to $2q - 1$.

Definition 8 (High and low bits for h)**.** Let $r \in \mathbb{Z}$. Let q be a prime and $\alpha_{\mathsf{h}} | 2(q-1)$ be a power of two. Let $m = 2(q-1)/\alpha_{\mathsf{h}}$, $r_1 = \mathsf{HighBits}(r \bmod^{+} 2q, \alpha_{\mathsf{h}})$, and $r_0 = \mathsf{LowBits}(r \bmod^{+} 2q, \alpha_{\mathsf{h}})$. If $r_1 = m$, let $(r_0', r_1') = (r_0 - 2, 0)$. Else, $(r_0', r_1') = (r_0, r_1)$. We define:

$$(\mathsf{LowBits}^{\mathsf{h}}(r), \mathsf{HighBits}^{\mathsf{h}}(r)) \;=\; (r_0', r_1').$$

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

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

The proof of Lemma 8 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 $\mathbf{j} = (1, 0, \ldots, 0) \in \mathcal{R}^k$, 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 $\alpha_{\mathsf{h}} | 2(q-1)$ is a power of two. We define the key rejection function based on Lemma 3:

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

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

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 $\mathbf{a}_{\mathsf{gen}}$ and $\mathbf{A}_{\mathsf{gen}}$ from $\mathsf{seed}_{\mathbf{A}}$,
- Extendable output function expandS for deriving $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) \in S_\eta^{\ell-1} \times S_\eta^k$ from $\mathsf{seed}_{\mathsf{sk}}$ and $\mathsf{counter}_{\mathsf{sk}}$,
- Extendable output function expandYbb for deriving $\mathbf{y}$, b and b' from seed_{ybb} and counter,

The above building blocks can be implemented with symmetric primitives.

Note that at Step 3 of the Verify algorithm, the division by 2 is well-defined as the operand is even.

Lemma 9. *We borrow the notations from Figure 8. If we run $\mathsf{Verify}(\mathsf{vk}, M, \sigma)$ on the signature σ returned by $\mathsf{Sign}(\mathsf{sk}, M)$ for an arbitrary message M and an arbitrary key-pair $(\mathsf{sk}, \mathsf{vk})$ returned by $\mathsf{KeyGen}(1^\lambda)$, then the following relations hold:*

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

Sign(sk, M):
  1: $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},(k+\ell)}(B))$
  2: $\mathbf{w} \leftarrow \mathbf{A}\lfloor \mathbf{y} \rceil$
  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 $\sigma = (\lfloor \mathbf{z} \rceil, c)$

Verify(vk, M, σ = (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

1) $\mathbf{w}_1 = \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w})$,

2) $w'\mathbf{j} = \mathsf{LSB}(\lfloor y_0 \rceil) \cdot \mathbf{j} = \mathsf{LSB}(\mathbf{w}) = \mathsf{LSB}(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rceil)$.

3) $2\lfloor \mathbf{z}_2 \rceil - 2\tilde{\mathbf{z}}_2 = \mathsf{LowBits}^{\mathsf{h}}(\mathbf{w}) - \mathsf{LSB}(\mathbf{w})$ *assuming it holds that* $B' + \alpha_{\mathsf{h}}/4 + 1 \leq B'' < q/2$,

Proof. Let $m = 2(q-1)/\alpha_{\mathsf{h}}$. Let us prove the first statement. By definition of $\mathbf{h}$, it holds that $\mathbf{w}_1 = \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w}) \mod m$. However, the latter part of the equality already lies in $[0, m-1]$ by Lemma 8. 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:

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

This yields the result. Moreover, considering everywhere a 2 appears in the definition of $\mathbf{A}$, we obtain that

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

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

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

We note that the last two elements have same parity, as the former one has the same parity as $\mathsf{LowBits}(\mathbf{w}, \alpha_{\mathsf{h}})$. By Lemma 8 their sum has infinite norm $\leq \alpha_{\mathsf{h}}/2 + 2$. Hence from its definition, it holds that

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

Finally, this holds over the integers as the right-hand side has infinite norm at most $2B' + \alpha_{\mathsf{h}}/2 + 2 < q$. □

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

Sign($\mathsf{sk}, M$):
1: $\mu = H_{\mathsf{gen}}(\mathsf{seed}_{\mathbf{A}}, \mathbf{b}_1, M)$
2: $\mathsf{seed}_{ybb} = H_{\mathsf{gen}}(K, \mu)$
3: $\mathsf{counter} = 0$
4: $(\mathbf{y}, b, b') := \mathsf{expandYbb}(\mathsf{seed}_{ybb}, \mathsf{counter})$
5: $\mathbf{w} \leftarrow \mathbf{A}\lfloor \mathbf{y} \rceil$
6: $\rho = H(\mathsf{HighBits}^{\mathsf{h}}(\mathbf{w}), \mathsf{LSB}(\lfloor y_0 \rceil), \mu)$
7: $c = \mathsf{SampleBinaryChallenge}_\tau(\rho)$
8: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c\mathbf{s}$
9: $\mathbf{h} = \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w}) - \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rceil) \bmod^+ \frac{2(q-1)}{\alpha_{\mathsf{h}}}$
10: if $\|\mathbf{z}\|_2 \geq B'$ then
11:     $\mathsf{counter}{+}{+}$ and Go to 4
12: else if $\|2\mathbf{z} - \mathbf{y}\|_2 < B \ \wedge \ b' = 0$ then
13:     $\mathsf{counter}{+}{+}$ and Go to 4
14: else
15:     $x = \mathsf{Encode}(\mathsf{HighBits}^{z1}(\lfloor \mathbf{z}_1 \rceil))$
16:     $\mathbf{v} = \mathsf{LowBits}^{z1}(\lfloor \mathbf{z}_1 \rceil)$
17:     return $\sigma = (x, \mathbf{v}, \mathsf{Encode}(\mathbf{h}), c)$

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

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.

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

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

Proof. We use the notations of the algorithms. The first and second equations from Lemma 9 state that $\rho = \tilde{\rho}$ and thus

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

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

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

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

□

4.3 Security

When proving security in the Fiat-Shamir with aborts setting in the QROM, on typically relies on the generic reduction from [KLS18]. However, as pointed out in [DFPS23] and [BBD$^+$23], this analysis is flawed. Both works give adaptations to Fiat-Shamir with aborts of the analysis from [GHHM21] of Fiat-Shamir (without aborts). Moreover, the reduction from [KLS18] assumes an arbitrary bound on the number of restarts, which is not the case here. This restriction is waived in both [DFPS23] and [BBD$^+$23].

4.3.1 UF-CMA Security

The security of HAETAE relies on the analysis of [DFPS23], which reduces UF-CMA security to UF-NMA security, where an adversary is not allowed to make signing queries. This analysis requires that the commitment min-entropy is high and the underlying Σ-protocol is Honest-Verifier Zero-Knowledge (HVZK). The latter is proved by providing a simulator for non-aborting transcripts and proving that the distribution of $\lfloor \mathbf{y} \rceil$ has sufficiently large min-entropy.

Commitment Min-entropy. We first claim that the underlying Σ-protocol has large commitment min-entropy. The underlying identification protocol has ε *bits of min-entropy* if for any $(\mathbf{w}, x)$,

$$\Pr_{\mathbf{y}}\left[(\mathsf{HighBits}^{\mathsf{h}}(\mathbf{A}\lfloor \mathbf{y} \rceil), \mathsf{LSB}(\lfloor y_0 \rceil)) = (\mathbf{w}, x)\right] \leq 2^{-\varepsilon},$$

for any $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}$ and $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},(k+\ell)}(B))$. We note that $\mathsf{LSB}(\lfloor y_0 \rceil)$ is a binary vector of length n and is statistically close to uniform. Thus, the inner probability is (very loosely) bounded by 2^{-n} regardless of the choice of $(\mathsf{pk}, \mathsf{sk})$. Hence we obtain at least 256 bits of min-entropy in all of our parameter sets.

HVZK. Next, we show that the underlying Σ-protocol satisfies the HVKZ property. To do so, we follow the strategy from [DFPS23, Section 4.2], which studies the simulation of non-aborting transcripts and switch to computational mode for aborting ones. We propose the following simulator in Figure 9. Here, $p(\mathbf{z})$ is $1/M$ if $\|\mathbf{z}\| \leq r$ and 0 everywhere else.

We first remark that the resulting simulated distribution of $(x, \mathbf{v}, h)$ (possibly $x = \perp$, $\mathbf{v} = \perp$, $h = \perp$) is identical to the real distribution of $(x, \mathbf{v}, h)$. As Lemma 5 states that the

```
Sim(vk, c) :
 1: $\mathbf{z} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},k+\ell}(B))$
 2: $\mathbf{w} \leftarrow (\mathsf{HighBits}^{\mathsf{h}}(\mathbf{A}\lfloor\mathbf{z}\rceil - qc\mathbf{j}), \mathsf{LSB}(c - y_0))$
 3: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$
 4: $x = \mathsf{Encode}(\mathsf{HighBits}^{z1}(\lfloor\mathbf{z}_1\rceil))$
 5: $\mathbf{v} = \mathsf{LowBits}^{z1}(\lfloor\mathbf{z}_1\rceil)$
 6: $h = \mathsf{Encode}(\mathbf{h})$
 7: $\mathbf{u} \leftarrow U(\mathcal{R}_q^k)$
 8: $u_0 \leftarrow U(\mathcal{R}_2)$
 9: $\tilde{\mathbf{w}} \leftarrow (\mathsf{HighBits}^{\mathsf{h}}(2\mathbf{u} + q\mathbf{j}u_0), u_0)$
10: return $(\mathbf{w}, x, \mathbf{v}, h)$ with probability $p(\mathbf{z})$, else $(\tilde{\mathbf{w}}, \perp, \perp, \perp)$
```

Figure 9: HAETAE transcript simulator

real and simulated $\mathbf{z}$ has an identical distribution, the bijective map $(x, \mathbf{v}, h) \mapsto \mathbf{z}$ gives the same result for the transcripts $(x, \mathbf{v}, h)$.

(i) Simulating non-aborting transcripts. In both the real and simulated cases, the non-aborted transcript satisfies $\mathbf{w} = \mathbf{A}\mathbf{z} - qc\mathbf{j} \bmod 2q$ for $\mathbf{z}$ recovered from $(x, \mathbf{v}, h)$. Hence, the statistical distance between the real and simulated transcripts is bounded by Lemma 5.

(ii) Simulating aborting transcripts. As argued in [DFPS23, Section 4.2], in this context, we can use a computational notion of HVZK rather than the usual statistical definition. We introduce an LWE-like assumption which states that it is hard to distinguish $\mathbf{w} = \mathbf{A}\lfloor\mathbf{y}\rceil \bmod 2q$ from a uniform element mod $2q$. This LWE assumption is unusual only in its choice of distribution for the noise and the secret.

These two properties allow us to apply [DFPS23, Theorem 4] to reduce the SUF-CMA security to UF-NMA security.

If one wants to avoid this assumption, it is possible to use the reduction from [BBD+23] by using $\mathcal{A}(0)$ as a simulator. The non-aborting transcripts produced by this simulator have statistical distance 0 with real ones.

UF-NMA security. Finally, we note that the UF-NMA security game is exactly the problem defined in Definition 5, up to replacing the verification key by an uniform matrix (still in HNF form), which can be done under the MLWE assumption.

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 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 yet unchosen 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 precomputations by using only the verifiction key, as shown in Figure 10. 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 10.

```
$\underline{\mathsf{Sign}^{\mathsf{offline}}(\mathsf{vk})}$:
1: $(\mathbf{a}_{\mathsf{gen}} |\ \mathbf{A}_{\mathsf{gen}}) = \mathsf{expandA}(\mathsf{seed}_{\mathbf{A}})$
2: $\mathbf{A}_1 = (2(\mathbf{a}_{\mathsf{gen}} - 2\mathbf{b}_1) + q\mathbf{j} |\ 2\mathbf{A}_{\mathsf{gen}}) \bmod 2q$
3: $\mathsf{List} = ()$
4: for iter in $[L]$ do
5:     $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},(k+\ell)}(B))$
6:     $\mathbf{w} = \mathbf{A}\lfloor \mathbf{y} \rceil$
7:     $\mathbf{w}_1 = \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w})$
8:     $\mathsf{List}.\mathsf{append}(\mathbf{y}, \mathbf{w}, \mathbf{w}_1, \mathsf{LSB}(\lfloor y_0 \rceil))$
9: return List

$\underline{\mathsf{Sign}^{\mathsf{online}}(\mathsf{sk}, \mathsf{List}, M)}$:
1: $\mu = H_{\mathsf{gen}}(\mathsf{seed}_{\mathbf{A}}, \mathbf{b}_1, M)$
2: $\mathsf{tuple} = (\mathbf{y}, \mathbf{w}, \mathsf{tuple}_3, \mathsf{tuple}_4) \leftarrow \mathsf{List}$
3: $\mathsf{List}.\mathsf{delete}(\mathsf{tuple})$
4: $c = \mathsf{SampleBinaryChallenge}_{\tau}(H(\mathsf{tuple}_3, \mathsf{tuple}_4, \mu))$
5: $(b, b') \leftarrow \{0,1\}^2$
6: $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c\mathbf{s}$
7: $\mathbf{h} = \mathsf{tuple}_3 - \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w} - 2\lfloor \mathbf{z}_2 \rceil) \bmod^+ \frac{2(q-1)}{\alpha_{\mathsf{h}}}$
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:     $x = \mathsf{Encode}(\mathsf{HighBits}^{z1}(\lfloor \mathbf{z}_1 \rceil))$
12:     $\mathbf{v} = \mathsf{LowBits}^{z1}(\lfloor \mathbf{z}_1 \rceil)$
13:     return $\sigma = (x, \mathbf{v}, \mathsf{Encode}(\mathbf{h}), c)$
```

Figure 10: Randomized, on/off-line signing. Note that app is the function that appends a tuple to the list.

4.5 Parameter Sets

For setting parameters, we estimated the costs of practical attacks, as in Dilithium, Falcon, and many other NIST-submitted schemes. We consider two kinds of attacks: 1) Key recovery attacks, which amount to solving an MLWE instance; 2) Signature forgery attacks, for which we rely on the BimodalSelfTargetMSIS instance that appears in the security analysis for UF-CMA. We then 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 $\mathbf{w} - qc\mathbf{j} \bmod 2q$, modulo the low bits truncation. If the adversary succeeds, the preimage becomes an MSIS solution.

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.

Table 3: HAETAE parameters sets. Hardness is measured with the Core-SVP methodology.

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	$\mathbf{y}$ radius (3.4)	9846.02	18314.98	22343.66
B'	Rejection radius (3.4)	9838.99	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
BKZ block-size b to break SIS		409 (333)	617 (512)	878 (735)
Best Known Classical bit-cost		119 (97)	180 (149)	256 (214)
Best Known Quantum bit-cost		105 (85)	158 (131)	225 (188)
BKZ block-size b to break LWE		428	810	988
Best Known Classical bit-cost		125	236	288
Best Known Quantum bit-cost		109	208	253

5 Implementation Specification

In this section, we explain how to efficiently implement HAETAE. We furthermore specify implementation aspects, that are relevant for compatibility or security. We start this section with an implementation-oriented specification. Specifically, Figure 11 demonstrates how to implement the key generation, Figure 12 the signature generation and Figure 13 the signature verification. These illustrate the use of the CRT and NTT for efficient polynomial arithmetic. Notably, $\mathbf{b}$ 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.

5.1 Hyperball Sampler

One critical component of HAETAE is the hyperball sampling. 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 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 $B_0 N$.

In the following, we explain how the Gaussian sampling and the inverse square root approximation 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 [Ros20, 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 Cumulative Distribution Table (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.

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

Algorithm 1 Deterministic hyperball sampling.

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

```
KeyGen(1^λ) for d > 0
 1: seed ← $\{0,1\}^{\rho_0}$
 2: $(\mathsf{seed}_{\mathbf{A}}, \mathsf{seed}_{\mathsf{sk}}, K) := H_{\mathsf{gen}}(\mathsf{seed})$
 3: $(\mathbf{a}_{\mathsf{gen}}, \widehat{\mathbf{A}}_{\mathsf{gen}}) := \mathsf{expandA}_d(\mathsf{seed}_{\mathbf{A}})$    ▷ $(\mathbf{a}_{\mathsf{gen}}, \widehat{\mathbf{A}}_{\mathsf{gen}}) \in \mathcal{R}_q^k \times \mathcal{R}_q^{k\times \ell-1}$
 4: $(\mathsf{counter}_{\mathsf{sk}}, \mathsf{flag}) := (0, \mathsf{true})$
 5: while flag do
 6:     $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) := \mathsf{expandS}(\mathsf{seed}_{\mathsf{sk}}, \mathsf{counter}_{\mathsf{sk}})$    ▷ $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) \in \mathcal{S}_\eta^{\ell-1} \times \mathcal{S}_\eta^k$
 7:     $\mathbf{b} := \mathbf{a}_{\mathsf{gen}} + \mathsf{NTT}^{-1}(\widehat{\mathbf{A}}_{\mathsf{gen}} \circ \mathsf{NTT}(\mathbf{s}_{\mathsf{gen}})) + \mathbf{e}_{\mathsf{gen}} \bmod q$    ▷ $\mathbf{b} \in \mathcal{R}_q^k$
 8:     $(\mathbf{b}_0, \mathbf{b}_1) := (\mathsf{LowBits}^{\mathsf{vk}}(\mathbf{b}), \mathsf{HighBits}^{\mathsf{vk}}(\mathbf{b}))$
 9:     $(\mathbf{s}_1, \mathbf{s}_2) := (\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}} - \mathbf{b}_0)$
10:     $\mathsf{counter}_{\mathsf{sk}} := \mathsf{counter}_{\mathsf{sk}} + 1$
11:     if $f(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 n$ then
12:         flag := false
13: $tr := H(\mathsf{seed}_{\mathbf{A}}, \mathbf{b}_1)$
14: return $\mathsf{sk} = (\mathbf{s}_1, \mathbf{s}_2, K, tr, \mathsf{seed}_{\mathbf{A}}, \mathbf{b}_1)$, $\mathsf{vk} = (\mathsf{seed}_{\mathbf{A}}, \mathbf{b}_1)$

KeyGen(1^λ) for d = 0
 1: seed ← $\{0,1\}^{\rho_0}$
 2: $(\mathsf{seed}_{\mathbf{A}}, \mathsf{seed}_{\mathsf{sk}}, K) := H_{\mathsf{gen}}(\mathsf{seed})$
 3: $(\mathsf{counter}_{\mathsf{sk}}, \mathsf{flag}) := (0, \mathsf{true})$
 4: while flag do
 5:     $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) := \mathsf{expandS}(\mathsf{seed}_{\mathsf{sk}}, \mathsf{counter}_{\mathsf{sk}})$    ▷ $(\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}}) \in \mathcal{S}_\eta^{\ell-1} \times \mathcal{S}_\eta^k$
 6:     $(\mathbf{s}_1, \mathbf{s}_2) := (\mathbf{s}_{\mathsf{gen}}, \mathbf{e}_{\mathsf{gen}})$
 7:     $\mathsf{counter}_{\mathsf{sk}} := \mathsf{counter}_{\mathsf{sk}} + 1$
 8:     if $f(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 n$ then
 9:         flag := false
10: $\widehat{\mathbf{A}}_{\mathsf{gen}} := \mathsf{expandA}_d(\mathsf{seed}_{\mathbf{A}})$    ▷ $\widehat{\mathbf{A}}_{\mathsf{gen}} \in \mathcal{R}_q^{k\times \ell-1}$
11: $\widehat{\mathbf{b}} := -2\left(\widehat{\mathbf{A}}_{\mathsf{gen}} \circ \mathsf{NTT}(\mathbf{s}_{\mathsf{gen}}) + \mathsf{NTT}(\mathbf{e}_{\mathsf{gen}})\right) \bmod q$    ▷ $\mathbf{b} \in \mathcal{R}_q^k$
12: $tr := H(\mathsf{seed}_{\mathbf{A}}, \widehat{\mathbf{b}})$
13: return $\mathsf{sk} = (\mathbf{s}_1, \mathbf{s}_2, K, tr, \mathsf{seed}_{\mathbf{A}}, \widehat{\mathbf{b}})$, $\mathsf{vk} = (\mathsf{seed}_{\mathbf{A}}, \widehat{\mathbf{b}})$
```

Figure 11: Implementation specification (deterministic version) of HAETAE key generation

approximation errors for a given polynomial order. We find:

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

where $I_n(z)$ are modified Bessel functions of the first kind, which rapidly converge to zero for growing n. We recall $\|T_n(x)\| \le 1$ for $\|x\| \le 1$. 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 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 [Li04]. 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

```
Sign(sk, $M$)
 1: $(\mathbf{s}_1, \mathbf{s}_2, K, tr, \mathsf{seed}_{\mathbf{A}}, \psi) := \mathsf{sk}$
 2: $\widehat{\mathbf{A}} := \mathsf{unpackA}_d(\mathsf{seed}_{\mathbf{A}}, \psi)$    ▷ Algorithm 2, $\mathbf{A} \in \mathcal{R}_q^{k\times\ell}$
 3: $\mu := H_{\mathsf{gen}}(tr, M)$
 4: $\mathsf{seed}_{ybb} := H_{\mathsf{gen}}(K, \mu)$
 5: $(\kappa, \sigma) := (0, \perp)$
 6: while $\sigma = \perp$ do    ▷ pre-compute $(\widehat{\mathbf{s}}_1, \widehat{\mathbf{s}}_2) := (\mathsf{NTT}(\mathbf{s}_1), \mathsf{NTT}(\mathbf{s}_2))$
 7:     $(\mathbf{y}_1, \mathbf{y}_2, b, b', \kappa) := \mathsf{expandYbb}(\mathsf{seed}_{ybb}, \kappa)$    ▷ $(\mathbf{y}_1, \mathbf{y}_2) \in (1/N)\mathcal{R}^\ell \times (1/N)\mathcal{R}^k$
 8:     $\mathbf{w} := \mathsf{NTT}^{-1}(\widehat{\mathbf{A}} \circ \mathsf{NTT}(\lfloor \mathbf{y}_1 \rceil)) + 2 \cdot \lfloor \mathbf{y}_2 \rceil \bmod q$    ▷ $\mathbf{w} \in \mathcal{R}_q^k$
 9:     $\mathbf{w}' := \mathsf{fromCRT}(\mathbf{w}, \lfloor y_{1,1} \rceil)$    ▷ Algorithm 3, $\mathbf{w}' \in \mathcal{R}_{2q}^k$
10:     $\mathbf{w}'_1 := \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w}')$
11:     $\rho = H(\mathbf{w}'_1, \mathsf{LSB}(\lfloor y_{1,1} \rceil), \mu)$
12:     $c = \mathsf{SampleBinaryChallenge}_\tau(\rho)$
13:     $\widehat{c} := \mathsf{NTT}(c)$
14:     $z_{1,1} := y_{1,1} + (-1)^b \cdot c$    ▷ $(\mathbf{z}_1, \mathbf{z}_2) \in (1/N)\mathcal{R}^\ell \times (1/N)\mathcal{R}^k$
15:     $(\mathbf{z}_1)_{2..\ell} := (\mathbf{y}_1)_{2..\ell} + (-1)^b \mathsf{NTT}^{-1}(\widehat{c} \circ \widehat{\mathbf{s}}_1)$
16:     $\mathbf{z}_2 := \mathbf{y}_2 + (-1)^b \mathsf{NTT}^{-1}(\widehat{c} \circ \widehat{\mathbf{s}}_2)$
17:     if $\|(\mathbf{z}_1, \mathbf{z}_2)\|_2 < B'$ and $(\|2(\mathbf{z}_1, \mathbf{z}_2) - (\mathbf{y}_1, \mathbf{y}_2)\|_2 > B$ or $b' = 1)$ then
                                                    ▷ Check this condition in constant time.
18:         $\mathbf{h} := \mathbf{w}'_1 - \mathsf{HighBits}^{\mathsf{h}}(\mathbf{w}' - 2\lfloor \mathbf{z}_2 \rceil) \bmod^+ \frac{2(q-1)}{\alpha_{\mathsf{h}}}$
19:         $\sigma := \mathsf{packSig}(\mathsf{HighBits}^{z1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LowBits}^{z1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c)$
                                                    ▷ Section 5.2 (can fail and return $\perp$)
```

Figure 12: Implementation specification (deterministic version) of HAETAE signing.

```
Verify(vk, $M$, $\sigma$)
 1: $(\mathsf{seed}_{\mathbf{A}}, \psi) := \mathsf{vk}$
 2: $\widehat{\mathbf{A}} := \mathsf{unpackA}_d(\mathsf{seed}_{\mathbf{A}}, \psi)$    ▷ Algorithm 2, $\mathbf{A} \in \mathcal{R}_q^{k\times\ell}$
 3: $(\mathsf{HighBits}^{z1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LowBits}^{z1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c) := \mathsf{unpackSig}(\sigma)$
                                                    ▷ Section 5.2 (can fail and cause a rejection)
 4: $\tilde{\mathbf{z}}_1 := \mathsf{HighBits}^{z1}(\lfloor \mathbf{z}_1 \rceil) \cdot 256 + \mathsf{LowBits}^{z1}(\lfloor \mathbf{z}_1 \rceil)$
 5: $w' := \mathsf{LSB}(\tilde{z}_{1,1} - c)$
 6: $\tilde{\mathbf{w}} := \widehat{\mathbf{A}} \circ \mathsf{NTT}(\tilde{\mathbf{z}}_1) \bmod q$
 7: $\tilde{\mathbf{w}}' := \mathsf{fromCRT}(\tilde{\mathbf{w}}, w')$    ▷ Algorithm 3
 8: $\tilde{\mathbf{w}}'_1 := \tilde{\mathbf{h}} + \mathsf{HighBits}^{\mathsf{h}}(\tilde{\mathbf{w}}') \bmod^+ \frac{2(q-1)}{\alpha_{\mathsf{h}}}$
 9: $\tilde{\mathbf{z}}_2 := [\tilde{\mathbf{w}}'_1 \cdot \alpha_{\mathsf{h}} + w'\mathbf{j} - \tilde{\mathbf{w}}' \bmod 2q]/2$ ▷ addition with $w'$ only for first vector element
10: $\tilde{\mu} = H_{\mathsf{gen}}(\mathsf{seed}_{\mathbf{A}}, \psi, M)$
11: Return $(c = \mathsf{SampleBinaryChallenge}_\tau(H(\tilde{\mathbf{w}}'_1, w', \tilde{\mu}))) \wedge (\|(\tilde{\mathbf{z}}_1, \tilde{\mathbf{z}}_2)\| < B'')$
```

Figure 13: Implementation specification (deterministic version) of HAETAE verification

compared to truncation.

Barthe et al. [BBE+19] 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) + 0x0000FFFFFFFB2E7ALL;
 result = ((smulh48(result, x) + 1LL) >> 1) - 0x0000FFFFFFFFF85FLL;
 result = ((smulh48(result, x))) + 0x0000FFFFFFFFFFFCLL;
 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 vast a randomness demand per hyperball sample, and explains the dominance of hashing in the cycle count performance.

5.1.2 Approximating the Inverse 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 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.

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 $z1$ (LN

Table 4: Symbol mapping and encoding size parameters.

Scheme	cut_h	$offset_h$	$\lvert\{S_h(n)\}\rvert$	cut_{z1}	$\lvert\{S_{z1}(n)\}\rvert$	$base_h$	$base_{z1}$
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

coefficients), the high bits of $z1$ and h (KN coefficients). Because the distributions of the values in the high bits of $z1$ and the coefficients in h are both very dense, we can compress both polynomial vectors with encoding. Figure 14 displays the frequencies for the possible values for both vectors in HAETAE-120.

Before compressing the values, we map them to a smaller symbol space and thereby reject the very unlikely values and the corresponding signatures. For 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.

$$S_h(n) = \left\{ \begin{array}{lr} n, & \text{for } 0 \leq n \leq cut_h \\ \bot, & \text{for } cut_h < n \leq cut_h + offset_h \\ n - offset_h, & \text{for } cut_h + offset_h < n \end{array} \right\}$$

For the high bits of $z1$ 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.

$$S_{z1}(n) = \left\{ \begin{array}{lr} \bot, & \text{for } n < -cut_{z1} \\ n + cut_{z1}, & \text{for } -cut_{z1} \leq n \leq cut_{z1} \\ \bot, & \text{for } cut_{z1} < n \end{array} \right\}$$

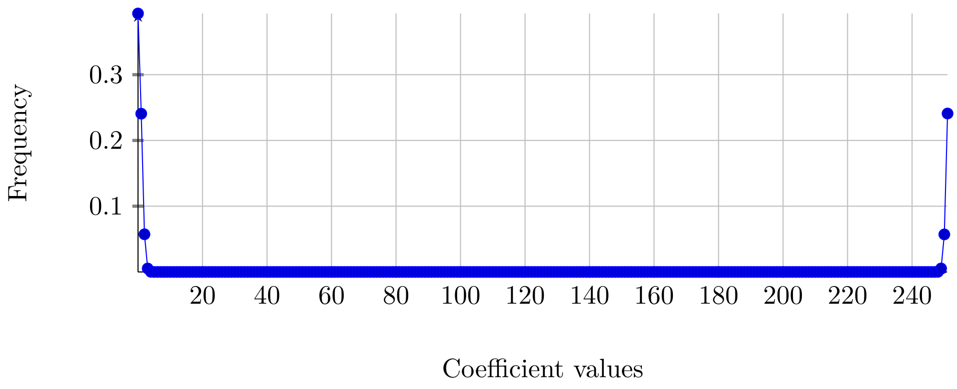
The parameters for these mappings are defined in Table 4. At the signature verification, the mapping must be reverted after decoding the compressed symbols.

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 $z1$, a mapping to non-negative values is necessary to be able to use rANS encoding. The effect on the resulting signature size is insignificant.

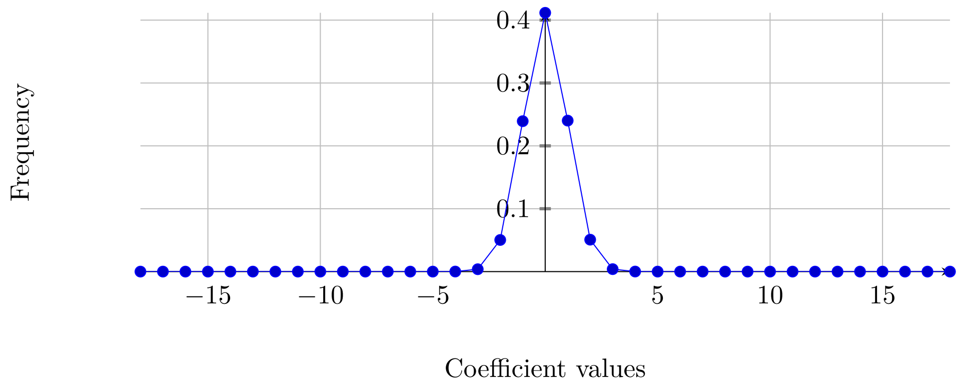
The size of the compressed high bits of $z1$ and h varies and must be included in the signature, to allow a correct unpacking and decoding. The size of one compressed polynomial vector is often more than 255 bytes, and can thus not be expressed by one byte. Its variance however, is limited, and thus we encode the size the compressed high bits of $z1$ and 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 4.

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 $z1$. The next first byte consists of the offset to the base size for the encoding of the high bits of $z1$ and the second byte is the offset for h. Then we have the encoding of the high bits of $z1$ and directly afterwards the encoding of 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 [Gie14].



(a) h



(b) HB(z1)

Figure 14: Distribution of the coefficients of h and HB(z1) in HAETAE-120.

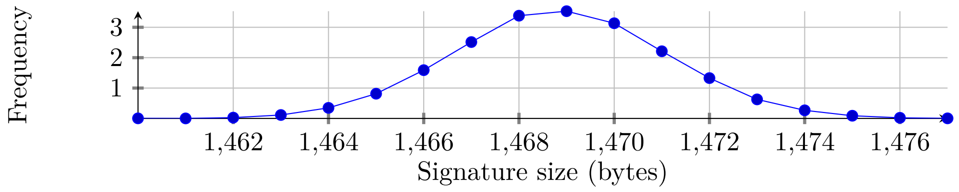
To set the fixed signature size as reported in Table 5, we evaluated the distribution empirically and determined a threshold that requires a rejection in less than 0.1% of the cases. Figure 15 displays the raw signature size distribution of 20000 executions (without the size-based rejection sampling).

In Table 5 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.

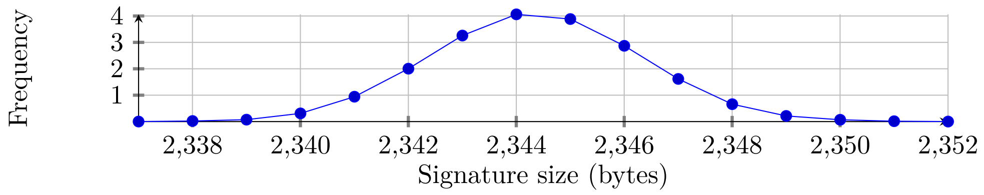
5.3 Performance Reference Implementation

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 6. 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.

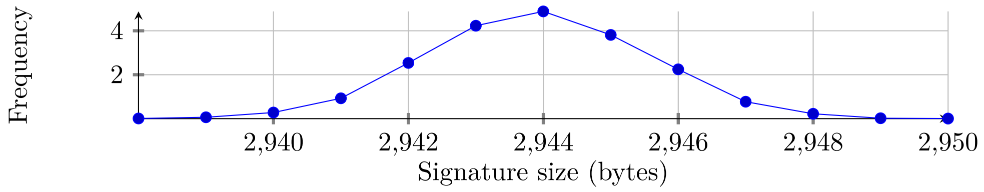
For a fair comparison, we also performed measurements on the same system with



(a) HAETAE-120



(b) HAETAE-180



(c) HAETAE-260

Figure 15: Raw signature size distribution over 20000 executions. We set the bound for the size-based rejection to result in a rejection rate of less than 0.1%.

identical settings of the reference implementation of Dilithium[1] and the implementation with emulated floating-point operations, and thus also fully portable, of Falcon[2], as given in Table 6. 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.

[1]https://github.com/pq-crystals/dilithium/tree/master/ref
[2]https://falcon-sign.info/falcon-round3.zip

Table 5: NIST security level, signature and key sizes (bytes) of HAETAE, Dilithium, and Falcon.

Scheme	Lvl.	vk	Signature	Sum	Secret key
HAETAE-120	2	992	1,474	2,466	1,376
HAETAE-180	3	1,472	2,349	3,821	2,080
HAETAE-260	5	2,080	2,948	5,028	2,720
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 6: 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.

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

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 [Dwo15], 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 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.

6 Optimized Implementation for AVX2

Advanced Vector Extensions 2 (AVX2) is an extension to the x86 instruction set architecture and available in processors since 2011. It provides Single Instruction Multiple Data (SIMD) operations on 256-bit registers, and thus allows to e.g. do an operation on eight 32-bit values in parallel. In this section we explain, how to exploit this parallelization.

The three major components, that significantly determine the computation time of HAETAE are Keccak, the NTT and the hyperball sampling. For the first two components we can fall back to existing optimized code. For the NTT in particular, we can reuse the implementation in Dilithium with only slight adaptions with regard to constants. In the following we demonstrate how to implement the third component, the hyperball sampling efficiently using AVX2 instructions.

6.1 Vectorized Hyperball Sampling

After the parallel generation of the randomness, generally, we have two options to parallelize the hyperball sampling. First, we can sample four different polynomials in parallel, and second, we can generate the Gaussian samples within a polynomial in parallel, since they are generated independently. We opt for the first approach.

As the sampling process is relatively complex, we cannot load input vectors, generate samples from them, and eventually store the samples. Instead, we pass several times over the internal memory state, dividing the procedure into seven separate steps:

1. parsing of the input randomness: separating the three parts for each sample candidate into separate memory locations such that later steps can process them quickly,
2. CDT sampling,
3. constructing the sample candidate, its square, and the input to the exponential approximation,
4. approximate the exponential,
5. generate masks which candidates to reject,
6. accumulate the squares of the non-rejected samples, and
7. storing only the accepted samples into the correct final memory positions.

In the following, we detail Steps 2, 3, and 4.

6.1.1 Parallel CDT Sampling

Although we perform 16-bit CDT sampling, we cannot use the 16-fold parallel `vpcmpgtw` comparison, since it is a *signed* comparison, and use `vpcmpgtd` instead, which operates on eight signed 32-bit integers. As we want to sample four samples in parallel, we store the CDT and the input randomness redundantly, such that we can perform the comparison with all 64 entries with 32 comparisons. However, since AVX2 only offers 16 vector registers, we have to perform this comparison in three chunks.

`vpcmpgtd c, a, b` writes -1 to c if the respective vector element in a is in fact greater than its counterpart in b, and else 0. Thus, we can use `vpaddd` to accumulate these results, but they will be negative. For the final chunk, we use `vpsrlq` to line up the two intermediate results, then add them and negate them.

6.1.2 Multi-precision Squaring

Since the sample candidate is 72 bits long and AVX2 only supports 32-bit multiplication, we perform vectorized multi-precision arithmetic. Therefore, we split the candidate into its low 32 bits, the middle 16 bits, and the upper 31 bits. For the schoolbook multiplication, we perform the six partial multiplications with `vpmuludq` consecutively, such that they can be executed pipelined and in parallel. The subsequent recombination and rounding can be performed with a sequence of 16 instructions of shifts (`vpsrlq`, `vpsllq`), additions (`vpaddq`), and ANDs (`vpand`).

6.1.3 Vectorized Approximation of the Exponential

The exponential approximation as explained in Section 5.1 consists of six signed 48-bit multiplications, which is not supported natively by AVX2. Consequently, we implement this operation with a vectorized multi-precision approach.

More specifically, we know that the first operand of this multiplication is signed, and the second is not. Thus, splitting the second operand into a low and a high half is trivial, but for the signed operand, this requires a slightly more sophisticated approach: Here, the upper half is obtained by an arithmetic right-shift by 24. By shifting this result left again by 24 (shifting in zeros), and subtracting the result from the original value, we obtain the lower half.

Since AVX2 does not offer a signed right shift over 64-bit entries, we generate a mask of sign bits and simulate a signed right shift by performing a bitwise OR. Unfortunately, we require this operation three times during a single signed multiplication operation. Notably, AVX512 offers a signed right shift, which will speed up this operation considerably.

Eventually, we perform a vectorized signed 48-bit multiplication using 32 instructions, out of which 17 are used for the emulation of a signed right shift. Moreover, we use seven variable and three constant vector registers (out of which one is the second input, cf. Listing 1), which leaves six registers for other constants. Apart from the multiplication, we only make use of addition and shifts (`vpaddq`, `vpsrlq`).

6.2 Performance and Comparison AVX2

The impact of the parallelized Keccak can be observed by looking at the cycle costs for unpacking the matrix $\mathbf{A}$, which is between five to seven times faster compared to the reference implementation. For e.g. HAETAE-120, the costs went down from around 132k cycles to 24k cycles. The picture is similar for the hyperball sampling, where we measure a speed-up of factor six to eight. The cycle counts for one function call in HAETAE-120 are around 1640k in the reference and 270k in the optimized implementation. The highly optimized NTT taken from Dilithium, is almost 19 times faster than the one in our portable reference code.

Table 7 provides cycle numbers for the AVX2 optimized implementations of HAETAE, Dilithium and Falcon. Compared to our reference implementation, the signature generation is around five times faster in the optimized implementation. For the signature verification we observe an acceleration of around three to four.

The comparison with Dilithium does not change distinctly with respect to the reference implementations, except for the key generation, where Dilithium experiences a much greater acceleration. Falcon on the other hand is considerably faster at the signature generation with its AVX2 implementation, compared to its portable reference code. Table 6 showed around three times faster signature generation for HAETAE compared to Falcon. Optimized for AVX2 and also using the floating-point unit, Falcon becomes faster than HAETAE.

We note, however, that both Dilithium and Falcon went through a multi-year process of incrementally optimized implementations, whereas this process has just started for HAETAE. Moreover, when we apply the heuristic that sending one byte via internet costs *at least* 1000 cycles [BBC+20, Sec. 5.4], remarkably, HAETAE is already nearly as performant as Dilithium in terms of signing plus sending the signature.

7 Embedded Implementation on Cortex-M4

To evaluate the suitability of HAETAE for embedded environments we developed an implementation for the `STM32f4-Discovery` board, featuring 128 KiB RAM and a Cortex-M4F processor which implements the ARMv7E-M ISA and operates on 32-bit words. We use the PQM4 framework [KRSS19] for development and evaluation, as it is the de facto standard for comparison of post-quantum cryptography schemes on Cortex-M4 processors. The Cortex-M4F, in contrast to the Cortex-M4, features a floating-point unit. Its floating point registers can be used to store and load intermediate values within a single cycle to

Table 7: AVX2 optimized 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.

Scheme		KeyGen	Sign	Verify
HAETAE-120	*med*	882,350	1,323,118	115,638
	ave	1,185,881	1,943,274	115,690
HAETAE-180	*med*	1,654,464	1,844,610	183,920
	ave	2,242,520	2,299,298	184,003
HAETAE-260	*med*	2,199,678	2,069,734	223,852
	ave	2,935,538	3,013,524	223,976
Dilithium-2	*med*	87,020	200,242	92,148
	ave	86,937	252,905	92,190
Dilithium-3	*med*	146,560	334,898	148,810
	ave	146,688	402,012	148,883
Dilithium-5	*med*	233,976	415,228	232,146
	ave	233,895	484,119	232,241
Falcon-512	*med*	24,663,306	863,076	100,540
	ave	26,637,878	863,420	100,709
Falcon-1024	*med*	71,013,520	1,740,188	228,086
	ave	78,797,658	1,740,520	228,326

reduce the pressure on the 13 general purpose registers.

The profiling of the reference implementation already indicates that replacing the portable Keccak implementation with one optimized for the Cortex-M4 is an important and straightforward step towards fast execution time. The two other major components that are highly relevant are the polynomial arithmetic and the Gaussian sampler, both will be discussed in the following.

7.1 Polynomial Arithmetic

In this section, we will address the issue of how to implement the required arithmetic operations on these rings and mappings between them on a Cortex-M4 platform.

7.1.1 Modular Reductions

In modular arithmetic, the Barret reduction [Bar87], the Montgomery modular multiplication [Mon85], and related techniques are indispensable for efficient computation, the first for reducing given numbers, the second for yielding the reduced result of a multiplication with a constant $\bmod q$. The algorithms avoid computationally expensive divisions by q and replace them with a multiplication by a suitably chosen number and division by powers of two, which can be realized with shift operations. Both methods initially reduce the result to the interval $[0, 2q]$ and perform the full reduction with a conditional subtraction, which can be done in constant time. In many cases one can forgo the final reduction for intermediate results, an approach dubbed lazy reduction.

The prime chosen in the HAETAE scheme is $q = 64513 = \texttt{0xFC01}$, the largest unsigned 16-bit prime with a 512^{th} root of unity. Fully reduced elements of $\mathbb{Z}_q$ can be stored efficiently in 16-bit in the bottom or top half of a 32-bit register.

Unfortunately, this does not carry over to arithmetic operations. A lazy reduction or an addition already requires 17 bits to store the result, a combination of a lazily reduced multiplication followed by an addition requires 18 bits. The recent advance of the Plantard multiplication [Pla21] is not useful within this work, as the prime in HAETAE is not compatible. Plantard multiplication requires $q < \frac{2R}{1+\sqrt{5}} \approxeq 0.618\,R$, i.e., $q \leq 40503$ for

$R = 2^{16}$. So the prime of HAETAE is too large for this use-case with Cortex-M4 16-bit DSP-instructions. The same goes for Seiler's variant [Sei18] of signed Montgomery modular multiplication, which is only well-defined for $q < \frac{R}{2}$.

7.1.2 16-bit vs 32-bit

Quite a few post-quantum schemes use primes that are 13-bit values or smaller. In this case, one can both store and manipulate the coefficients graciously and efficiently as two signed 16-bit values packed into one 32-bit register, as the Cortex-M4 offers a wide range of instructions intended for Digital Signal Processor (DSP) applications, like mixed multiplication of upper and lower halves of two registers that can be used for this purpose. In the case of HAETAE, trade-offs need to be found between the compactness of 16-bit storage and doubled speed of access for consecutive coefficients on the one hand, and the required overhead to fully reduce the coefficients before writing them to memory.

If coefficients are written once and afterwards are read repeatedly without alterations, the 16-bit representation can be worthwhile. When polynomials of the public key are expanded, the coefficients are sampled in fully reduced state, we therefore store them in halfwords.

While it is feasible to use a modified Montgomery reduction with unsigned 16-bit integers as input and 17 bits output (or a 16-bit value with overflow flag), there are no corresponding instructions available to exploit this. In contrast, the Montgomery reduction in the Dilithium implementation for the Cortex-M4 uses $R = 2^{32}$ and takes only three instructions. We determined the overhead associated with full reductions required to store coefficients as 16-bit values to be too large to outperform the 32-bit variant for the NTT. The same applies to other polynomial arithmetic operations in HAETAE, besides the expanded polynomials of the public key, we therefore operate on 32-bit coefficients.

7.1.3 NTT

HAETAE, as other lattice-based schemes, extensively employs polynomial multiplication. The NTT is a generalization of the Fast Fourier Transform (FFT) and is the state-of-the-art technique to perform this operation, speeding up the computation considerably, as compared to, e.g., schoolbook multiplication. The addition and multiplication of polynomials transformed into the NTT domain are carried out coefficient-wise, greatly reducing the cost of the latter operation. The overhead to perform the transform and inverse transform, where required, is usually outweighed by the performance gain in the multiplication. HAETAE is specified such that large parts of the public key are expanded directly to the NTT domain.

Fortunately, the closeness to Dilithium, which also uses polynomials of degree $n = 256$, allows us the reuse its highly optimized assembly code for the NTT developed for the Cortex-M4 by Abdulrahman et al. [AHKS22], which improves over previous work [GKOS18]. The code adjustments required for operation in HAETAE are limited to adjusting constants like the prime being used, the root of unity, which is chosen as 426 (the primitive 512^{th} root of unity of $q = 64513$) and the twiddle factors.

Replacing the portable C code from the reference implementation with optimized assembly derived from the Dilithium implementation reduces the cycle count per invocation from 37506 cycles to 8047 cycles, a speed-up by a factor of **4.6**. For the inverse NTT, the cycle count dropped from 43116 to 8369, a speed-up by a factor of **5.1**.

7.2 Gaussian Sampler

The major numerical components of the Gaussian sampler are the CDT sampler for sampling the most significant bits and the fixed-point exponential function used in the rejection step. As both components are called repeatedly, both have been implemented in assembly code.

The CDT sampler accumulates ones and zeros, depending on comparisons of a uniformly sampled 16-bit random value to tabled threshold values. We use the `uadd16`, `usub16` and `sel` SIMD instructions from the Cortex-M4 instruction set to carry out two comparisons and accumulations in parallel. We furthermore optimize the memory access and unroll the loop. By doing so we reduce the instruction count from 800 cyles for the reference implementation to 206 cyles, a speed-up by a factor of 3.9.

The exponential is approximated by a polynomial, which is evaluated using Horner's scheme. The reference implementation of the exponential function uses fixed-point arithmetic with 48 fraction bits. Values are embedded in 64-bit integers and 64x64 to 128-bit multiplication is used. The latter operation has no native support on the Cortex-M4. To circumvent this limitation, the Cortex-M4 implementation splits the value into two signed components at the start of each multiplication, namely the most significant bits `ah = a » 24` and the least significant bits as `al = a - ah`. For the accumulation of the results a 64-bit integer is used again, taking advantage of the `smlal` instruction. This repeated switch between representations allows for efficient computation of the individual Horner's scheme iterations. Whereas the reference implementation of the exponential function takes 1658 cycles to execute, this is reduced to 563 instructions in the optimized Cortex-M4 code, a speed-up by a factor of 2.9.

7.3 Stack Optimization

Besides execution time, also the memory footprint is an important metric for constrained devices. The target device in this work has 128 KiB of RAM available as stack memory. In this context, data structures typically encountered in lattice based cryptography need to be considered as rather large. A single polynomial in HAETAE takes 512 B or 1 KiB of memory to store, depending on whether the data is represented as 16-bit or 32-bit values. So vectors or matrices of polynomials can occupy a considerable share of the available RAM, if stored in their entirety. In this section we explore how to minimize memory use; in some cases significant trade-offs between memory usage and execution speed can be made.

The reference implementation of HAETAE is designed with an emphasis on readability and close similarity to the mathematical specification. This results in top level functions, which consist of long monolithic blocks with many large data structures, which do not necessarily have overlapping lifetimes, but nevertheless occupy stack space for the entire function lifecycle. Most stack memory is required for the signature generation. Due to the unnecessarily high stack usage of the reference implementation, HAETAE-180 and HAETAE-260 do not run on the `STM32f4-Discovery` board without optimizations.

To reduce the memory footprint we executed two strategies. First, we carefully analyzed the liveness of each relevant variable and refactored the monolithic code into subroutines to reduce the scope of variables and thereby the total stack usage. This slightly impacts the readability of the code, but does not affect the performance.

In a second variant we additionally opted for a more aggressive memory reduction, by recomputing polynomials on demand, this obviously comes with performance costs. We adapt the data structures to be primarily polynomial oriented instead of vector and matrix oriented representations. Recomputations are done during public key usage, where we generate each polynomial on demand, and during hyperball sampling, where we sample each polynomial twice, once for the evaluation of the normalization factor and a second time to sample the actual $\mathbf{y}$ values. Since the hyperball sampling is computationally very expensive, this leads to a severe overhead in runtime.

Table 8: Maximum stack size in bytes for Cortex-M4 implementations of HAETAE, Dilithium, and Falcon.

Scheme		KeyGen	Sign	Verify
HAETAE-120	*speed-opt*	19,796	54,564	22,532
	stack-opt	17,364	40,732	22,532
HAETAE-180	*speed-opt*	29,612	69,631	31,020
	stack-opt	22,444	57,116	31,020
HAETAE-260	*speed-opt*	34,108	102,964	36,428
	stack-opt	22,356	68,380	36,428
Dilithium-2		38,408	49,380	36,212
Dilithium-3		60,836	68,836	57,724
Dilithium-5		97,692	115,932	92,788
Falcon-512		18,416	42,508	4,724
Falcon-1024		36,296	82,532	8,820

7.4 Performance and Comparison Cortex-M4

Table 8 shows the maximum stack size of our two Cortex-M4 implementations of HAETAE and values reported in the PQM4 framework about Dilithium and Falcon. With *speed-opt*, we refer to our implementation, that is optimized for the Cortex-M4 and includes multiple stack-size optimizations, but does not trade speed for better memory requirements. *stack-opt* refers to the version, where we additionally exploit speed vs memory trade-offs.

First, we can observe that the memory requirements of HAETAE are small enough to run on the `STM32f4-Discovery` board for all parameter sets, even in the *speed-opt* version. Second, the stack sizes for HAETAE are in the same order of magnitude as Dilithium and Falcon. Compared to *speed-opt* HAETAE, Dilithium requires around two to three times more memory during key generation, and a similar overhead for signature verification. The difference is at most 20% for signature generation, for this operation Dilithium requires less memory than HAETAE for the first two parameter sets.

Our *stack-opt* version reduces the stack-size up to 34% during signature generation and key generation, but does not differ for the verification. However, this comes with higher costs in terms of computation time.

Falcon sticks out for its stack-size below 10 KiB for both parameter sets during verification.

Table 9 shows the cycles spend by our two Cortex-M4 implementations of HAETAE and values reported in the PQM4 framework about Dilithium and Falcon. Our version using aggressive stack reduction techniques based on recomputations does not impact the signature verification time, but almost doubles the computation time for the signature generation. The time overhead for key generation is up to 30%. Similar to our AVX2 optimized implementation, the relative performance comparison of HAETAE to Dilithium and Falcon does not change drastically.

7.5 Security against Physical Attacks

Implementation security is a crucial aspect of making cryptosystems feasible in real-world applications. A significant advantage of HAETAE is that it can be protected against power side-channel attacks efficiently and with reasonable overhead. In this context, we emphasize the similarity of HAETAE to Dilithium. Hence, past works analyzing concrete attacks [BP18, MUTS22], but also countermeasures [MGTF19, ABC$^+$22], mainly apply to HAETAE as well.

While there is no known method to efficiently mask Falcon, Mitaka [EFG$^+$22] was

Table 9: Average cycle counts of 1000 (100 for Falcon) executions on the Cortex-M4 for HAETAE, Dilithium, and Falcon.

Scheme		KeyGen	Sign	Verify
HAETAE-120	*speed-opt*	8,904,552	27,311,965	1,054,478
	stack-opt	10,818,804	51,016,745	1,054,472
HAETAE-180	*speed-opt*	17,666,326	34,466,279	2,026,448
	stack-opt	22,859,766	65,854,630	2,026,454
HAETAE-260	*speed-opt*	22,850,880	50,174,603	2,733,469
	stack-opt	23,213,004	99,471,768	2,733,451
Dilithium-2		1,597,999	4,111,596	1,571,804
Dilithium-3		2,830,024	6,588,465	2,691,283
Dilithium-5		4,826,422	8,779,067	4,705,693
Falcon-512		155,757,768	38,979,435	481,452
Falcon-1024		480,071,949	85,125,001	994,972

designed to be easy to protect against implementation attacks, while still having the advantage of similarly small signatures as Falcon. For Mitaka, the crux regarding side-channel security is sampling Gaussian-distributed values. Together with Mitaka, an efficient, masked algorithm for discrete Gaussian sampling was presented. However, Prest broke its security proof recently [Pre23]. In this respect, HAETAE has the strong advantage that Gaussian sampling only needs to be secured against the much stronger Simple Power Analysis (SPA) attacker model, which allows for simpler countermeasures, while Mitaka's side-channel security will always depend on a masked sampler.

While a fully protected implementation of HAETAE is out of scope for this paper, we briefly sketch its feasibility.

Protecting the Arithmetic. Most notably, HAETAE does not deploy floating-point arithmetic at any point, and only few secrets require fix-point arithmetic. Remarkably, addition of fix-point arithmetic can be masked relatively easy, and HAETAE never requires a multiplication of fix-point values.

During signing, the most critical operation is multiplying the (public) challenge polynomial c with $\mathbf{s}$ and subsequently adding the result to $\mathbf{y}$. Since this operation may leak information about the secret key statistically over many executions, implementers must protect it accordingly. As countermeasures against these so-called Differential Power Analysis (DPA) attacks, masking has been proven effective.

This operation is straightforward to mask at arbitrary order by splitting the secret key polynomials into multiple additive shares in $\mathcal{R}_q$. A masked implementation then stores the NTT of each share of $\mathbf{s}$ and multiplies them to c, obtaining a shared $c\mathbf{s}$. Following this, the inverse NTT is applied share-wise. Since $\mathbf{y}$ is a polynomial vector in $(1/N)\mathcal{R}$, it is not trivially possible to add our shares of $c\mathbf{s} \in \mathcal{R}_q^{k+\ell}$. On the other hand, $\mathbf{y}$ is not a secret-key-dependent value. Therefore, it is not required to be protected against DPA but only against the much stronger attacker model of a SPA. In fact, coefficient-wise shuffling of the addition might be sufficient at this point.

Independent of whether the addition is shuffled or masked, this involves a masking conversion from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^{32}}$. Subsequently, the computation of $2\mathbf{z} - \mathbf{y}$ and the bound checks can be shuffled without applying costly masking.

Protecting the Hyperball Sampler. The same idea applies to the whole hyperball sampling procedure. Since the order of the Gaussian samples is, in principle, irrelevant, they can be generated in random order. This is particularly an advantage for randomized HAETAE.

For the deterministic version, a masked CDT sampler, and a masked approximation of the exponential function are required. The former was shown to be feasible recently by Krausz et *al.* [KLS+23], while the latter is a sequence of multiplications, shifting by a constant amount, and addition by constants, which is expected to be costly but feasible.

It is noteworthy that the random oracle hash (which outputs the challenge) is only required to be protected against SPA as well. Since the input order into the hash function cannot be randomized, the preceding values must still be protected by masking. Therefore, if no masked hyperball sampling has been performed, we propose to perform a shuffled point-wise multiplication of $\mathbf{A}$ and $\mathbf{y}$, directly followed by freshly masking the resulting coefficients. Then, a share-wise inverse NTT and a masking conversion to the Boolean domain will be performed, which enables a secure HighBits operation. For the LSBs of y_0, generating a fresh Boolean masking during the shuffled generation of the hyperball sample's coefficients is sufficient.

8 Conclusion

With HAETAE, we close an important gap between the two state-of-the-art digital signature schemes Dilithium and Falcon. Novel contributions in key generation and rejection sampling allow us to reach *significantly* smaller signature and verification key sizes, while still allowing physical side-channel protected implementations for IoT use-cases. Moreover, our first set of optimized implementations exhibits that our proposed algorithms run in feasible time both on embedded and high-end platforms and compete with existing schemes when considering sending latency.

Acknowledgements

Part of this work was done while Julien Devevey and Damien Stehlé were in École Normale Supérieure de Lyon and Institut Universitaire de France, and Dongyeon Hong and MinJune Yi were in CryptoLab Inc. Julien Devevey and Damien Stehlé were supported by the AMIRAL ANR grant (ANR-21-ASTR-0016), the PEPR quantique France 2030 programme (ANR-22-PETQ-0008) and the PEPR Cyber France 2030 programme (ANR-22-PECY-0003). Tim Güneysu, Markus Krausz, Georg Land and Marc Möller have been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by the German Federal Ministry of Education and Research BMBF through the projects QuantumRISC (16KIS1038) and PQC4Med (16KIS1044).

References

[ABC+22] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Leveling Dilithium against leakage: Revisited sensitivity analysis and improved implementations. Cryptology ePrint Archive, Report 2022/1406, 2022. https://eprint.iacr.org/2022/1406.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 853–871. Springer, Heidelberg, June 2022.

[AKM$^+$15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[ASY22] Shweta Agrawal, Damien Stehlé, and Anshu Yadav. Round-optimal lattice-based threshold signatures, revisited. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *ICALP 2022*, volume 229 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl, July 2022.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, August 1987.

[BBC$^+$20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. Ntru prime: round 3. https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf, 2020.

[BBD$^+$23] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. Cryptology ePrint Archive, Paper 2023/246, 2023. https://eprint.iacr.org/2023/246.

[BBE$^+$19] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. GALACTICS: Gaussian sampling for lattice-based constant- time implementation of cryptographic signatures, revisited. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2147–2164. ACM Press, November 2019.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7267.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, Heidelberg, August 2013.

[DFPS22] Julien Devevey, Omar Fawzi, Alain Passelègue, and Damien Stehlé. On rejection sampling in lyubashevsky's signature scheme. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 34–64. Springer, Heidelberg, December 2022.

[DFPS23] Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of Fiat-Shamir with aborts. Cryptology ePrint Archive, Paper 2023/245, 2023. https://eprint.iacr.org/2023/245.

[DKL$^+$18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839.

[dPEK+] Rafaël del Pino, Thomas Espitau, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, Mélissa Rossi, and Markku-Juhani Saarinen. Raccoon: A side-channel secure signature scheme. Submission to the NIST Additional Post-Quantum Digital Signature Schemes standardization process. Available at https://raccoonfamily.org/wp-content/uploads/2023/07/raccoon.pdf.

[DPS23] Julien Devevey, Alain Passelègue, and Damien Stehlé. G+G: A fiat-shamir lattice signature based on convolved gaussians. *IACR Cryptol. ePrint Arch.*, page 1477, 2023.

[Dud13] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding, 2013. ArXiv preprint, available at https://arxiv.org/abs/1311.2540.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. *FIPS 202*, 2015.

[EFG+22] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 222–253. Springer, Heidelberg, May / June 2022.

[ENST23] Thomas Espitau, Guilhem Niot, Chao Sun, and Mehdi Tibouchi. Square unstructured integer euclidean lattice signature. *Submission to the NIST's post-quantum cryptography standardization process*, 2023.

[ETWY22] Thomas Espitau, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Shorter hash-and-sign lattice-based signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 245–275. Springer, Heidelberg, August 2022.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5), 2018.

[GHHM21] Alex B. Grilo, Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz. Tight adaptive reprogramming in the QROM. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 637–667. Springer, Heidelberg, December 2021.

[Gie14] Fabian Giesen. Interleaved entropy coders. *arXiv preprint arXiv:1402.3392*, 2014.

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of lattice-based signature schemes in embedded systems. In *25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018, Bordeaux, France, December 9-12, 2018*, pages 385–388. IEEE, 2018.

[GS23] Jason Goertzen and Douglas Stebila. Post-quantum signatures in DNSSEC via request-based fragmentation. In Thomas Johansson and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2023, Proceedings*, volume 14154 of *Lecture Notes in Computer Science*, pages 535–564. Springer, 2023.

[HDS23] Abiodoun Clement Hounkpevi, Sidoine Djimnaibeye, and Michel Seck. Eaglesign: A new post-quantum elgamal-like signature over lattices. *Submission to the NIST's post-quantum cryptography standardization process*, 2023.

[HPP$^+$23] Thomas Pornin Huang, Eamonn W Postlethwaite, Thomas Prest, Ludo N Pulles, and Wessel van Woerden. Hawk. version 1.0.1 (july 19, 2023), 2023.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.

[KA21] Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 691–696, 2021.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 552–586. Springer, Heidelberg, April / May 2018.

[KLS$^+$23] Markus Krausz, Georg Land, Florian Stolz, Dennis Naujoks, Jan Richter-Brockmann, Tim Güneysu, and Lucie Johanna Kogelheide. Generic accelerators for costly-to-mask PQC components. *IACR Cryptol. ePrint Arch.*, page 1287, 2023.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.

[Li04] Ren-Cang Li. Near optimality of chebyshev interpolation for elementary function computations. *IEEE Trans. Computers*, 53(6):678–687, 2004.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Heidelberg, December 2009.

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Heidelberg, April 2012.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 344–362. Springer, Heidelberg, June 2019.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MUTS22] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on Dilithium: A small bit-fiddling leak breaks it all. Cryptology ePrint Archive, Report 2022/106, 2022. https://eprint.iacr.org/2022/106.

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021.

[Por19] Thomas Pornin. New efficient, constant-time implementations of falcon. Cryptology ePrint Archive, Paper 2019/893, 2019.

[Pre23] Thomas Prest. A key-recovery attack against mitaka in the t-probing model. Cryptology ePrint Archive, Report 2023/157, 2023. https://eprint.iacr.org/2023/157.

[PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in tls. In *Post-Quantum Cryptography: 11th International Conference, PQCrypto 2020, Paris, France, April 15–17, 2020, Proceedings 11*, pages 72–91. Springer, 2020.

[Ros20] Melissa Rossi. *Extended Security of Lattice-Based Cryptography*. PhD thesis, École Normale Supérieure de Paris, 2020.

[RS23] Keegan Ryan and Adam Suhl. Round 1 (additional signatures) official comment: Eht, 2023.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[Sch00] Werner Schindler. A timing attack against rsa with the chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems—CHES 2000: Second International Workshop Worcester, MA, USA, August 17–18, 2000 Proceedings 2*, pages 109–124. Springer, 2000.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[SSF23] Igor Semaev, Auxiliary Submitter, and Martin Feussner. Digital signature algorithms ehtv3 and ehtv4 submission to nist pqc. *Submission to the NIST's post-quantum cryptography standardization process*, 2023.

[Tib23] Mehdi Tibouchi. Round 1 (additional signatures) official comment: Eaglesign, 2023.

[VGS17] Aaron R Voelker, Jan Gosmann, and Terrence C Stewart. Efficiently sampling vectors and coordinates from the n-sphere and n-ball. *Centre for Theoretical Neuroscience-Technical Report*, 01 2017.

[Wes21] Bas Westerbaan. Sizing Up Post-Quantum Signatures. Technical report, Cloudflare, 2021.

[YJL$^+$23] Yang Yu, Huiwen Jia, Leibo Li, Delong Ran, Zhiyuan Qiu, Shiduo Zhang, Xiuhan Lin, and Xiaoyun Wang. Hufu: Hash-and-sign signatures from powerful gadgets. algorithm specifications and supporting documentation. version 1.1 (september 2, 2023), 2023.

A Recent lattice-based signatures

In this section, we give a size comparison of the recent lattice-based signature schemes. In Table 10, we compare the verification key and signature sizes (in bytes) of some selected lattice signature schemes, including the NIST's on-ramp candidates. Note that we additionally showcase some disadvantages, e.g., some recent analyses reported at pqc-forum (some are verified by the submitters, and others are not) or the use of new family of assumptions, which may degrade the claimed security.

Table 10: Size comparison of recent lattice-based signatures.

Scheme	Levels 1 & 2			Level 3			Level 5			Ref.
	vk	sig.	sum ↑	vk	sig.	sum	vk	sig.	sum	
EHTv4[1]	1,110	369	1,479	—	—	—	1,110	369	1,479	[SSF23]
Falcon [2]	897	666	1,563	—	—	—	897	666	1,563	[FHK+18]
HAWK[3]	1,024	555	1,579	—	—	—	1,024	555	1,579	[HPP+23]
Mitaka [4]	896	713	1,609	—	—	—	896	713	1,609	[EFG+22]
HAETAE	992	1,474	2,466	1,472	2,349	3,821	2,080	2,948	5,028	**ours**
Dilithium	1,312	2,420	3,732	1,952	3,293	5,245	2,592	4,595	7,187	[DKL+18]
EagleSign[5]	1,824	2,144	3,968	2,842	2,336	5,160	3,616	3,488	7,104	[HDS23]
Raccoon	2,256	11,524	13,780	3,160	14,544	17,704	4,064	20,330	24,394	[dPEK+]
SQUIRRELS	682K	1,019	683K	1,600K	1,554	1,602K	—	—	—	[ENST23]
HUFU[6]	1,059K	2,450	1,061K	2,177K	3,540	2,181K	3,573K	4,520	3,578K	[YJL+23]

[1] reported attack [RS23] [2] infeasible to mask, uses floating-point arithmetic [3] new assumptions smLIP, omSVP [4] hard to mask, proposed masked Gaussian sampler recently broken [Pre23] [5] reported attack [Tib23] [6] uses floating-point arithmetic

B Additional Proofs

B.1 Useful Lemma

We will rely on the following claim.

Lemma 10. *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. $\mathsf{Vol}(\mathcal{B}_{\mathcal{R},m}(r - \frac{\sqrt{mn}}{2})) \le |\mathcal{R}^m \cap \mathcal{B}_{\mathcal{R},m}(r)| \le \mathsf{Vol}(\mathcal{B}_{\mathcal{R},m}(r + \frac{\sqrt{mn}}{2}))$.

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:

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

Taking the volumes gives the result. □

B.2 Proof of Lemma 4

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 $\mathbf{z} = \lfloor \mathbf{y} \rceil$. Note that $\mathbf{z}$ is sampled (before the rejection step) with probability

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

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 $1/\mathsf{Vol}(\mathcal{B}_{\mathcal{R},m}(Nr' + \sqrt{mn}/2)$ 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.

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

By the triangle inequality and Lemma 10, it is

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

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

B.3 Proof of Lemma 5

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., [DFPS22]). 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 10):

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

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. □

Algorithm 2 Unpacking routine for $\widehat{\mathbf{A}}$.

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

B.4 Proof of Lemma 7

Proof. By Lemma 6, there exists a unique representation

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

By identifying $\mathsf{HighBits}(r, \alpha)$ and $\mathsf{LowBits}(r, \alpha)$ 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.

□

B.5 Proof of Lemma 8

Proof. Let $r \in [0, 2q - 1]$. Let r_0, r_1, r_0', and r_1' defined as in Definition 8. If $r_0' = r_0$ and $r_1' = r_1$, the equality $r_0' + r_1' \cdot \alpha_{\mathsf{h}} = r_0 + r_1 \cdot \alpha_{\mathsf{h}} \bmod 2q$ holds vacuously.

If not, then $r_0' = r_0 - 2$ and $r_1' = r_1 - 2(q - 1)/\alpha_{\mathsf{h}}$ and $r_0' + r_1'\alpha_{\mathsf{h}} = r_0 + r_1\alpha_{\mathsf{h}} - 2q$. By Lemma 7, we get the first equality.

The second property stems from the second property in Lemma 7. The modifications to r_0 make r_0' lie in the range $[-\alpha_{\mathsf{h}}/2 - 2, \alpha_{\mathsf{h}}/2)$.

The last property stems from the third property in Lemma 7 and the fact that if $r_1 = m$, then we have $r_1' = 0$.

□

C Additional Implementation Specification

Algorithm 2 describes how to implement the unpacking of $\widehat{\mathbf{A}}$, and in Algorithm 3 we demonstrate how to apply the CRT.

D Notes Regarding Hardware Implementations

Hashing and generation of randomness are the most time-consuming operations of HAETAE. Therefore, we assume that hardware implementations will bring significant speedup and can be competitive to Dilithium, particularly through efficient Keccak cores. Furthermore, hardware implementations will benefit significantly from applying the offline approach. Naturally, a module generating hyperball samples can be instantiated and run parallel to the online phase, thus, hiding its latency behind the online phase. Moreover, high-speed

Algorithm 3 Mapping from $(\mathcal{R}_q^k, \mathcal{R}_q)$ to $\mathcal{R}_{2q}^k$

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

applications could adopt the offline approach with designated signing key, including the multiplication of $\mathbf{A}$ and $\mathbf{y}$, to further reduce the latency of the online phase.