HAWK

version 1.0 (June 1, 2023)

https://hawk-sign.info

Joppe W. Bos[1], Olivier Bronchain[1], Léo Ducas[2,3], Serge Fehr[2,3], Yu-Hsuan Huang[2], Thomas Pornin[4], Eamonn W. Postlethwaite[2], Thomas Prest[5], Ludo N. Pulles[2], Wessel van Woerden[6]

[1] NXP Semiconductors,
[2] Centrum Wiskunde & Informatica,
[3] Mathematical Institute, Leiden University,
[4] NCC Group,
[5] PQShield,
[6] Institut de Mathématiques de Bordeaux.

1 Introduction

HAWK is a signature scheme inspired by the introduction of the lattice isomorphism problem (LIP) to signatures [DvW22], and this specification document is based on the article that first described a practical variant [DPPvW22b]. The move from the framework of [DvW22] towards a practical variant in [DPPvW22b] is achieved by introducing module structure and using simpler sampling procedures to create signatures. The HAWK signature scheme specified in this document introduces further simplifications and optimisations compared to the signature scheme introduced in [DPPvW22b], which we will refer to as HAWK-AC22. In particular, the distribution used to sample private keys in HAWK is simplified compared to that in HAWK-AC22.

The practical security of HAWK is determined by extensive lattice reduction experiments along with a detailed cryptanalysis, and the SUF-CMA security of HAWK reduces to a lattice problem called the one more (approximate) shortest vector problem, or omSVP. The problem of recovering the secret key directly from the public key is an instance of the search module lattice isomorphism problem, or smLIP. Compared to HAWK-AC22 the experimental cryptanalysis has been extended and the reduction to omSVP has been modularised, extended to the qROM and its loss has been made explicit. Throughout we work under the premise that our formal reductions and problems give us confidence in the robustness of our design, and our cryptanalysis gives us confidence in the robustness of our parameters.

We provide two parameter sets, HAWK-512 and HAWK-1024, that our analysis suggests satisfy NIST-I and NIST-V security levels respectively. We also provide a challenge parameter set, HAWK-256 to act as a cryptanalytic target.

For didactic purposes we provide an implementation of HAWK in Python 3 that follows the specification, except for in one explicitly commented place. We also provide an optimised C implementation of the exact specification of HAWK. Our C implementation is lightweight, fast, isochronous and without floating points.

We discuss the advantages and limitations of HAWK in Section 1.1.

A note on naming. The name HAWK is similar to another signature scheme: FALCON, which was selected by NIST for standardisation. Initially, beyond sharing the hash-and-sign design of FALCON, there were no obvious similarities. However, as part of our key generation we need to solve an NTRU equation in a similar setting to FALCON and our HAWK-AC22 implementation reused much of the FALCON code. We therefore, despite the different underlying hard problems, named this scheme similarly in homage.

1.0.1 Overview of this specification

Technical preliminaries are given in Section 2. The specification of HAWK is given in Section 3. Implementation and performance details are given in Section 4. In Section 5 we detail our cryptanalytic model and experimental results for HAWK, followed by our estimated security strengths. In Section 6 we detail our formal security claims and give the omSVP reduction. We also discuss the omSVP and smLIP assumptions, as well as our design decisions with respect to security. To aid the ease of use of this document the preliminaries of Section 2 should be read by all, then

- implementors need only read Sections 3 and 4,
- cryptanalysts need only read Section 5, as well as possibly Section 6 for a discussion of omSVP and smLIP,
- theoretical cryptographers need only read Section 2 and Section 6 for the definitions of omSVP, smLIP and reductions.

Table 1: Speed of HAWK. Bracketed values require more RAM, see Table 3.

	HAWK-512	HAWK-1024
Speed on x86 "Coffee Lake" with AVX2 (clock cycles)		
Key pair generation	8.43×10^6	4.37×10^7
Signature generation	8.54×10^4 (4.37×10^4)	1.81×10^5 (8.54×10^4)
Signature verification	1.48×10^5 (1.24×10^5)	3.03×10^5 (2.55×10^5)
Speed on ARM Cortex M4 (clock cycles)		
Key pair generation	5.23×10^7	2.26×10^8
Signature generation	2.80×10^6 (1.16×10^6)	1.42×10^6 (1.23×10^6)
Signature verification	1.42×10^6 (1.23×10^6)	3.01×10^6 (2.61×10^6)

Table 2: Key and signature sizes for HAWK in bytes.

	HAWK-512	HAWK-1024
Private key size	184	360
Public key size	1024	2440
Signature size	555	1221

1.1 Advantages and Limitations

1.1.1 Advantages

Speed. The procedures for generating and verifying signatures are fast on all devices, including low end devices, see Table 1.

Compactness. The keys and signature sizes are all rather small, see Table 2.

Memory usage. HAWK has a small memory footprint. For example, HAWK-512 requires no more than 14 kiB of RAM for any algorithm, including the faster variants of signing and verification.[1] If the keys can be generated externally and hardcoded onto the device, then HAWK-512 and HAWK-1024 can sign and verify using only 6 kiB and 11 kiB of RAM respectively. This makes HAWK a good candidate for many embedded platforms based on ARM Cortex-M0(+) cores: products in this range include, for example, the LPC800 series by NXP, STM32F0 by ST, or the XMC1000 by Infineon (16 KiB of SRAM).

Well suited for various hardware. HAWK is free of floating-point arithmetic. Therefore, no floating point (double precision) unit is required. This enables one to run HAWK on many (constrained) devices not equipped with such an FPU.

[1] A kibibyte (kiB) is 1024 bytes

Table 3: RAM usage of HAWK in bytes. Bracketed values correspond to Table 1.

	HAWK-512	HAWK-1024
Key pair generation	14336	27648
Signature generation	4096 (5272)	7168 (9512)
Signature verification	6144 (8768)	11264 (16512)

Simplicity of distributions. Key generation in HAWK requires samples from a centred binomial distribution, which is simple to sample from using a source of uniform bits. Signing requires discrete Gaussian sampling with a fixed width from two cosets of the integer lattice $\mathbb{Z}$, which is easily achieved using two fixed precomputed tables of sufficient precision.

Isochronous. Any function within HAWK that depends on secret data has a running time independent of said data.

Worst-case running time. With high probability a signature is generated without any internal restarts. In particular, the signing procedures of HAWK-512 and HAWK-1024 restart in approximately 1/200000 and 1/400000 cases respectively.

BUFF transform. Applying the (full) BUFF transform [CDF+20, Fig. 6][2] to HAWK requires implementors to append a single hash digest to the signature and subsequently check its value in verification. This hash digest is already internally computed, and the signing of HAWK follows the necessary design.

1.1.2 Limitations

No efficient masking (yet). Despite the simplicity of requiring only two fixed discrete Gaussian distributions to be sampled during signing, it is an open research problem to efficiently mask the table based method we use. On a positive note, besides this component it is known how to efficiently mask the remainder of the design of HAWK.

Security assumptions (direct key recovery). Recovering the private key from the public key in HAWK is equivalent to solving an instance of smLIP over the integer lattice $\mathbb{Z}^{2n}$ that follows a distribution implicit in our key generation. While there exists research into the lattice isomorphism problem over the integer lattice [GS02, Szy03, BM21, BGPSD23, DPPvW22b, Duc23], the use of the problem in constructive cryptography, especially with module structure, is relatively new, and would benefit from further attention.

Security assumptions (SUF-CMA). The SUF-CMA security of HAWK reduces to a new problem: omSVP [DPPvW22b, App. B]. We pick the parameters of HAWK to satisfy our reduction, so that breaking the SUF-CMA security of HAWK breaks the corresponding omSVP instance with an explicit loss. This reduction is unidirectional given preimage resistance for some hash, i.e. we do not know how to break the SUF-CMA security of HAWK by solving omSVP.

Like smLIP, the use of omSVP in cryptography is new, and would benefit from further attention. Furthermore, while we set the parameters of HAWK to satisfy our reduction, the parametrisation of omSVP that we reduce to has some trivial solutions. To remove these trivial solutions would require us to sample larger keys in our key generation procedure, which we choose not to do. This approach is indicative of our methodology; we use our reduction to omSVP to provide confidence in the robustness of the *design* of HAWK and our practical cryptanalysis to provide confidence in the robustness of our *parameters*.

Finally, we do not know how to publicly simulate the signature distribution of HAWK, à la [GPV08]. Instead, our reduction to omSVP provides a well defined target for the applicability of learning style attacks [NR09, DN12] to HAWK, which requires further attention.

[2] Note that the conference version [CDF+21] numbers its figures differently.

Deviation from (previous) theoretical guarantees. To sign with a narrower discrete Gaussian and reduce the number of cosets required to be sampled from, HAWK deviates from the EUF-CMA proof structure of the signature scheme given in [DvW22, Sec. 6] and therefore requires the omSVP problem for security, rather than the ΔLIP assumption from that work. Also, the choice to use a centred binomial distribution to simplify the generation of public keys, while having no practical effect on security according to our analysis, does not fit the notion of average case distribution given in [DvW22, Sec. 3].

The final three limitations are discussed in more detail in Sections 5 and 6.

2 Preliminaries

Vectors and matrices. We denote (column) vectors in lowercase as e.g. $\mathbf{v}$ and the number of elements in such a vector as $\mathsf{len}(\mathbf{v})$. A vector $\mathbf{v}$ is formed of elements $v_0, \ldots, v_{\mathsf{len}(\mathbf{v})-1}$, that is, we index from 0. We denote matrices in uppercase as e.g. $\mathbf{B}$ and index from 0, so that if $\mathbf{B}$ has dimensions $r \times s$ the top right entry is $b_{0,0}$ and the bottom right entry is $b_{r-1,s-1}$. For any object X with 0 and 1 elements and positive integer n we define $\mathbf{I}_n(X)$ as the identity matrix in $X^{n\times n}$.

Distributions. We consider two families of discrete distributions.

The first is the centred binomial distribution of parameter $\eta \in \mathbb{Z}_{\geq 1}$. If $X \sim \mathrm{Bin}(\eta)$ then

$$\mathrm{Supp}(X) = \{-\eta, -\eta+1, \ldots, \eta\}, \quad \Pr[X = x] = \frac{1}{2^{2\eta}} \cdot \binom{2\eta}{x+\eta} \tag{1}$$

for $x \in \mathrm{Supp}(X)$. It has $\mathbb{E}[X] = 0$ and $\mathbb{V}[X] = \eta/2$. Sampling $X \sim \mathrm{Bin}(\eta)$ is easily achieved as

$$\left(\sum_{i=1}^{2\eta} x_i\right) - \eta, \tag{2}$$

for $x_1, \ldots, x_{2\eta}$ sampled as i.i.d. bits, i.e. from the uniform distribution over $\{0, 1\}$.

The second is the discrete Gaussian distribution on $2\mathbb{Z} + c$ of parameters $\sigma \in \mathbb{R}_{>0}$ and $c \in \mathbb{R}$. Let

$$\rho_\sigma \colon \mathbb{R} \to \mathbb{R},\ x \mapsto \exp(-x^2/2\sigma^2). \tag{3}$$

If $X \sim D_{2\mathbb{Z}+c,\sigma}$ then

$$\mathrm{Supp}(X) = 2\mathbb{Z} + c, \quad \Pr[X = x] = \frac{1}{\sum_{z\in 2\mathbb{Z}+c} \rho_\sigma(z)} \cdot \rho_\sigma(x), \tag{4}$$

for $x \in \mathrm{Supp}(X)$. We consider only $c \in \{0, 1\}$ in HAWK and for these two values of c, it holds $\mathbb{E}[X] = 0$. For σ large enough, $\mathbb{V}[X] \in \sigma^2 \cdot (1 + O(e^{-\sigma^2}))$ [MR04].

For higher dimensions $n \in \mathbb{Z}_{\geq 2}$, a sample $\mathbf{x} \in \mathbb{R}^n$ from the discrete Gaussian distribution on $2\mathbb{Z}^n + \mathbf{c}$ of parameters $\sigma \in \mathbb{R}_{>0}$ and $\mathbf{c} \in \mathbb{R}^n$ is obtained by sampling each $x_i \leftarrow D_{2\mathbb{Z}+c_i,\sigma}$ for $0 \leq i < n$ independently and combining these into a vector $\mathbf{x} \in \mathbb{R}^n$.

Rényi divergence. For two distributions P, Q with supports $\mathrm{Supp}(P) \subseteq \mathrm{Supp}(Q)$ we define the Rényi divergence (RD) of order $a \in (1, \infty)$ as

$$R_a(P \,\|\, Q) = \left(\sum_{x\in \mathrm{Supp}(P)} \frac{P(x)^a}{Q(x)^{a-1}}\right)^{1/(a-1)}. \tag{5}$$

It is not symmetric in its arguments, and does not verify a triangle inequality. When the Rényi divergence is finite, which it will be for our applications, we think of it as a value $1 + \delta$ for $\delta \geq 0$; the smaller δ is the closer the two distributions. Note that for any P, Q on which the Rényi divergence is finite, $R_a(P \,\|\, Q)$ is non decreasing as a function of a, i.e. a larger order a defines a stricter notion of closeness.

We recall three properties of $R_a(P \,\|\, Q)$ and a useful result [BLL+15, Pre17]. For two distributions P, Q and two families $(P_i)_i, (Q_i)_i$ we have

- (data processing inequality) for any function f, $R_a\big(P^f \,\|\, Q^f\big) \leq R_a(P \,\|\, Q)$,

- (multiplicativity) $R_a(\prod_i P_i \,\|\, \prod_i Q_i) = \prod_i R_a(P_i \,\|\, Q_i)$,
- (probability preservation) for event $E \subseteq \mathrm{Supp}(P)$, $Q(E) \geq P(E)^{a/(a-1)}/R_a(P \,\|\, Q)$.

Intuitively the data processing inequality says that new distributions P^f, Q^f formed by "processing" the output of P, Q with some function f can never increase the divergence. We often consider P as an inexact realisation of some ideal distribution Q and the event E as some adversary winning a search game; in this setting the probability preservation inequality allows us to bound above the probability of an adversary winning the search game using distribution P. We also consider the order a as a parameter that we may optimise over.

Finally, we note that [Pre17, (4)] gives us the following useful condition on the Rényi divergence. If a search game is λ-bit secure when instantiated with Q, the game samples Q at most q_s times, and for an order $a \geq 2\lambda + 1$

$$R_a(P \,\|\, Q) \leq 1 + \frac{1}{4q_s}\,, \tag{6}$$

then the search game is at least $(\lambda - 1)$-bit secure when instantiated with P with the same sample bound q_s.

Number fields. We make use of number fields and their rings of integers. In particular for $n = 2^m$ with $m \in \{8, 9, 10\}$ we consider the power of two cyclotomic number field $K_n = \mathbb{Q}[X]/(X^n + 1)$ and its ring of integers $R_n = \mathbb{Z}[X]/(X^n + 1)$. For a polynomial $f \in K_n$ we let

$$f = f[0] + f[1]X + \cdots + f[n-1]X^{n-1},$$

and define a bijection that maps it to the column vector of its coefficients

$$\mathsf{vec}\colon K_n \to \mathbb{Q}^n,\ f \mapsto (f[0], \ldots, f[n-1]). \tag{7}$$

Polynomials are thus also sequences of their n coefficients, in ascending degree order.

We also define

$$\mathsf{rot}\colon K_n \to \mathbb{Q}^{n\times n},\ f \mapsto (\mathsf{vec}(f), \mathsf{vec}(fX), \ldots, \mathsf{vec}(fX^{n-1})), \tag{8}$$

sometimes called the negacyclic matrix of f. We extend vec and rot to K_n^r and $K_n^{r\times r}$ as

$$\mathsf{vec}\colon K_n^r \to \mathbb{Q}^{nr},\ \mathbf{f} = (f_0, \ldots, f_{r-1}) \mapsto (\mathsf{vec}(f_0), \ldots, \mathsf{vec}(f_{r-1})) \tag{9}$$

and

$$\mathsf{rot}\colon K_n^{r\times r} \to \mathbb{Q}^{nr\times nr},\ \mathbf{B} \mapsto \begin{pmatrix} \mathsf{rot}(b_{0,0}) & \cdots & \mathsf{rot}(b_{0,r-1}) \\ \vdots & \ddots & \vdots \\ \mathsf{rot}(b_{r-1,0}) & \cdots & \mathsf{rot}(b_{r-1,r-1}) \end{pmatrix}, \tag{10}$$

where context informs the correct version of vec or rot to use. These number fields are CM-fields and therefore have an involutive automorphism representing complex conjugation; the Hermitian adjoint. We denote by

$$f^\star = f[0] - f[n-1]X - \cdots - f[1]X^{n-1} \tag{11}$$

the Hermitian adjoint of f. If $\mathbf{B} \in K_n^{r\times r}$ then $\mathbf{B}^\star$ is the matrix obtained by applying the Hermitian adjoint to each entry of $\mathbf{B}^\mathsf{t}$, the transpose matrix of $\mathbf{B}$.

Inner product and norm. We define an inner product over K_n as

$$\langle \cdot, \cdot \rangle \colon K_n \times K_n \to \mathbb{Q},\ (f, g) \mapsto \frac{1}{n}\mathrm{Tr}(f^\star g)\,, \tag{12}$$

where $\mathrm{Tr}(\cdot)$ is the algebraic trace over the field extension $K_n/\mathbb{Q}$. Due to the normalisation factor $\frac{1}{n}$, if $f, g \in K_n$ then $\langle f, g\rangle$ equals the Euclidean inner product of $\mathsf{vec}(f)$ and $\mathsf{vec}(g)$. We define two norms,

$$\|\cdot\| \colon K_n \to \mathbb{Q},\ f \mapsto \sqrt{\langle f, f\rangle}, \tag{13}$$

$$\|\cdot\|_\infty \colon K_n \to \mathbb{Q},\ f \mapsto \max_{0 \le i < n}(|f[i]|). \tag{14}$$

Note that these norms coincide with the "Euclidean" ℓ^2 and "infinity" ℓ^∞ norms of $\mathsf{vec}(f)$ respectively. In particular, we may calculate

$$\|f\|^2 = f[0]^2 + \cdots + f[n-1]^2. \tag{15}$$

We extend the inner product in (12) to K_n^r via summation as

$$\langle \cdot, \cdot \rangle \colon K_n^r \times K_n^r \to \mathbb{Q},\ (\mathbf{f}, \mathbf{g}) \mapsto \langle f_0, g_0\rangle + \cdots + \langle f_{r-1}, g_{r-1}\rangle. \tag{16}$$

Similarly, we extend the norms in Eqs. (13) and (14) to K_n^r as

$$\|\cdot\| \colon K_n^r \to \mathbb{Q},\ \mathbf{f} \mapsto \sqrt{\|f_0\|^2 + \cdots + \|f_{r-1}\|^2}, \tag{17}$$

$$\|\cdot\|_\infty \colon K_n^r \to \mathbb{Q},\ \mathbf{f} \mapsto \max_{0 \le i < r}(\|f_i\|_\infty). \tag{18}$$

These extended norms coincide with the ℓ^2 and ℓ^∞ norms of $\mathsf{vec}(\mathbf{f})$ in $\mathbb{Q}^{nr}$. In particular, we may calculate

$$\|\mathbf{f}\|^2 = f_0[0]^2 + \cdots + f_0[n-1]^2 + \cdots + f_{r-1}[0]^2 + \cdots f_{r-1}[n-1]^2. \tag{19}$$

Q-inner product and Q-norm. For $\mathbf{B} \in K_n^{r \times r}$ such that the columns of $\mathbf{B}$ are K_n-linearly independent, we consider the Hermitian form associated to the self adjoint matrix $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$. We now define the $\mathbf{Q}$-inner product, or the inner product with respect to this form, as

$$\langle \cdot, \cdot \rangle_{\mathbf{Q}} \colon K_n^r \times K_n^r \to \mathbb{Q},\ (\mathbf{f}, \mathbf{g}) \mapsto \frac{1}{n}\mathrm{Tr}(\mathbf{f}^\star\mathbf{Q}\mathbf{g})\,. \tag{20}$$

Note that (20) specialises to (16) whenever $\mathbf{Q} = \mathbf{I}_r(K_n)$.

From the $\mathbf{Q}$-inner product one may naturally define the $\mathbf{Q}$-norm as

$$\|\cdot\|_{\mathbf{Q}} \colon K_n^r \to \mathbb{Q},\ \mathbf{f} \mapsto \sqrt{\langle \mathbf{f}, \mathbf{f}\rangle_{\mathbf{Q}}}. \tag{21}$$

We also have the useful identities, when $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$ for $\mathbf{B}$ with K_n-linearly independent columns and $\mathbf{f}, \mathbf{g} \in K_n^r$

$$\langle \mathbf{B}\mathbf{f}, \mathbf{B}\mathbf{g}\rangle = \langle \mathbf{f}, \mathbf{g}\rangle_{\mathbf{Q}},\ \|\mathbf{B}\mathbf{f}\| = \|\mathbf{f}\|_{\mathbf{Q}}. \tag{22}$$

These allow us to translate between the $\mathbf{Q}$-norm and the (identity) norm if we know such a $\mathbf{B}$.

Moving from structured to unstructured. At various points we will implicitly consider "structured" objects, e.g. matrices in $R_n^{2\times 2}$, as corresponding "unstructured" objects, e.g. matrices in $\mathbb{Z}^{2n\times 2n}$, via vec and rot. This is particularly the case in our experimental cryptanalysis, where we do not know how to exploit this structure and our tools work on matrices over the rationals or integers. This approach is also used in the implementation. We note here that for $\mathbf{B} \in K_n^{r\times s}$ and $\mathbf{C} \in K_n^{s\times t}$ we have

$$\mathbf{BC} = \mathsf{rot}^{-1}(\mathsf{rot}(\mathbf{B})\,\mathsf{rot}(\mathbf{C})),$$

where the inverse rot function is well defined because the product $\mathsf{rot}(\mathbf{B})\,\mathsf{rot}(\mathbf{C})$ will consist of $r \times t$ negacyclic blocks of size $n \times n$. In the case where $t = 1$ this collapses to $\mathbf{Bc} = \mathsf{vec}^{-1}(\mathsf{rot}(\mathbf{B})\,\mathsf{vec}(\mathbf{c}))$. Also, for some Hermitian matrix $\mathbf{Q} \in K_n^{r\times r}$ and $\mathbf{f}, \mathbf{g} \in K_n^r$ we have

$$\langle \mathbf{f}, \mathbf{g} \rangle_{\mathbf{Q}} = \mathsf{vec}(\mathbf{f})^{\mathsf{t}}\,\mathsf{rot}(\mathbf{Q})\,\mathsf{vec}(\mathbf{g}).$$

In short, to compute products and norms, we may pass from structured objects to unstructured objects.

However, in the case of performing lattice reduction on $\mathsf{rot}(\mathbf{B})$ with $\mathbf{B} \in K_n^r$, the (unstructured) reduced basis $\mathbf{C} = \mathsf{rot}(\mathbf{B})\mathbf{U} \in \mathbb{Q}^{nr}$ for $\mathbf{U} \in \mathsf{GL}_{nr}(\mathbb{Z})$ is generally not of a "structured" form, i.e. not of the form $\mathsf{rot}(\mathbf{B}')$ for some $\mathbf{B}' \in K_n^r$. We discuss this more in Section 5.

2.1 High level algorithmic description of HAWK

Here we give a high level description of the algorithms of HAWK. The high level descriptions of key generation, signature generation and signature verification for HAWK can be found in Algorithms 1, 2 and 3 respectively. These descriptions of the algorithms leave many subroutines undefined, and are not intended to be implemented; instead implementors are encouraged to read Sections 3 and 4. We assume n is a power of two and we assume the parameters $(\eta, \mathsf{saltlen}_{\mathsf{bits}}, \sigma_{\mathsf{sign}}, \sigma_{\mathsf{verify}})$, which depend on n, are known. We do not include subroutines for encoding and decoding data, but we do indicate where checks regarding encoding take place. Finally, we often compute the output of a function H on some data. In the full specification H is realised as SHAKE256 and varying amounts of output data are used depending on n and the point of use in a given algorithm. In this high level description we do not state the output length used, except when necessary, or specify how an object input to H may be considered as a bitstring.

Algorithm Sketch 1 High level HawkKeyGen

Ensure: $\mathbf{B} \in \mathsf{GL}_2(R_n)$ and $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$

1: Sample coefficients of $f, g \in R_n$ i.i.d. from $\mathrm{Bin}(\eta)$
2: **if** f-g-conditions(f, g) is false **then restart**
3: $r \leftarrow \mathsf{NTRUSolve}(f, g)$
4: **if** r is $\perp$ **then restart**
5: $(F, G) \leftarrow r$
6: $\mathbf{B} \leftarrow \begin{pmatrix} f & F \\ g & G \end{pmatrix}$, $\mathbf{Q} \leftarrow \mathbf{B}^\star\mathbf{B}$
7: **if** KGen-encoding$(\mathbf{Q}, \mathbf{B})$ is false **then restart**
8: hpub $\leftarrow \mathsf{H}(\mathbf{Q})$
9: **return** $(\mathsf{pk}, \mathsf{sk}) \leftarrow (\mathbf{Q}, (\mathbf{B}, \mathsf{hpub}))$

Key generation. Key generation in HAWK consists of sampling a secret unimodular matrix $\mathbf{B} \in \mathsf{GL}_2(R_n)$ from a distribution that is implicitly determined by a centred binomial distribution and the algorithm NTRUSolve.

First, two polynomials $f, g \in R_n$ have their coefficients i.i.d. sampled from $\mathrm{Bin}(\eta)$, where η depends on the security parameter n. The conditions that make up f-g-conditions have several purposes, such as allowing for efficient constant time procedures within the algorithm NTRUSolve that follows, and to abide by the results of our practical cryptanalysis. The algorithm NTRUSolve outputs $F, G \in R_n$ such that $fG - gF = 1_{R_n}$, and such that F, G have somewhat small entries. We restart if NTRUSolve fails; this can happen if no such (F, G) exist, or because NTRUSolve fails to find them. The public key $\mathbf{Q}$ is then the Hermitian matrix $\mathbf{B}^\star\mathbf{B}$.

The conditions that make up KGen-encoding check whether the keys can be properly encoded. Finally, we compute hpub, a hash of the public key, which is included in the secret key.

The recovery of sk from pk is an instance of the search module lattice isomorphism problem from a distribution determined by HawkKeyGen, see Section 6. We approach this cryptanalytic task experimentally via lattice reduction in Section 5. Intuitively, knowledge of $\mathbf{B}$ allows one to efficiently find short vectors in various cosets of $\mathbb{Z}^{2n}$, whereas knowledge of $\mathbf{Q}$ allows one to calculate the lengths of elements.

Algorithm Sketch 2 High level HawkSign

Require: A message m and secret key $\mathsf{sk} = (\mathbf{B}, \mathsf{hpub})$

Ensure: A signature sig formed of a uniform salt $\mathsf{salt} \in \{0,1\}^{\mathsf{saltlen_{bits}}}$ and $s_1 \in R_n$

1: $M \leftarrow \mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$
2: $\mathsf{salt} \leftarrow \mathsf{Rnd}(\mathsf{saltlen_{bits}})$
3: $\mathbf{h} \leftarrow \mathsf{H}(M \parallel \mathsf{salt})$
4: $\mathbf{t} \leftarrow \mathbf{B} \cdot \mathbf{h} \bmod 2$
5: $\mathbf{x} \leftarrow D_{2\mathbb{Z}^{2n}+\mathbf{t}, 2\sigma_{\mathsf{sign}}}$
6: **if** $\|\mathbf{x}\|^2 > 4 \cdot \sigma^2_{\mathsf{verify}} \cdot 2n$ **then restart**
7: $\mathbf{w} \leftarrow \mathbf{B}^{-1}\mathbf{x}$
8: **if** sym-break$(\mathbf{w})$ is false **then** $\mathbf{w} = -\mathbf{w}$
9: $\mathbf{s} \leftarrow \frac{1}{2}(\mathbf{h} - \mathbf{w})$
10: $s_1 \leftarrow \mathsf{Compress}(\mathbf{s})$
11: **if** sig-encoding(salt, s_1) is false **then restart**
12: **return** $\mathsf{sig} \leftarrow (\mathsf{salt}, s_1)$

Signing. Signing in HAWK consists of using the message to determine a target coset $2R_n^2 + \mathbf{h}$ of $2R_n^2$. A random salt is used to make this choice of target coset non deterministic. A signature is the salt and (a compressed version of) a vector $\mathbf{s}$. This signature vector $\mathbf{s}$ is related via $\mathbf{h}$ to a vector $\mathbf{w}$ which is in the target coset and short under $\|\cdot\|_{\mathbf{Q}}$.

First two hashes are calculated; the first over the message m and hpub, then a uniform salt salt is sampled, which is hashed with the output of the first hash. The function $\mathsf{Rnd}(r)$, on Line 2 of Algorithm 2, returns a uniform bitstring of length r. The function is called with input $\mathsf{saltlen_{bits}}$, which depends on n. This design that hashes twice is chosen so that HAWK can be easily adapted to make use of the BUFF transform [CDF+20]. The output of the second hash has length $2n$ and is interpreted as a binary vector in R_n^2.

We then sample a vector $\mathbf{x}$ from the discrete Gaussian distribution over $2\mathbb{Z}^{2n} + \mathbf{t}$ for $\mathbf{t} = \mathsf{vec}(\mathbf{Bh})$ with the width parameter $2\sigma_{\mathsf{sign}}$. Note that for this purpose, it is only required to know the value of $\mathbf{t}$ modulo 2. We then interpret the sampled $\mathbf{x}$ as an element of R_n^2. Note that $\mathbf{w} = \mathbf{B}^{-1}\mathbf{x} \in 2R_n^2 + \mathbf{h}$ is in the target coset and that $\|\mathbf{w}\|_{\mathbf{Q}} = \|\mathbf{x}\|$. If $\|\mathbf{x}\|$ is too long the signature would fail in verification, so we restart.

The condition sym-break prevents a simple weak forgery attack. Since $\mathbf{h} - \mathbf{w} \in 2R_n^2$, we have $\mathbf{s} \in R_n^2$ and that $\mathbf{w}$ can be publicly recomputed from $\mathbf{s}$ and $\mathbf{h}$. The salt salt and a compressed version of $\mathbf{s}$ form the signature. This compression operation does not have

any secret information as input. Finally, sig-encoding checks whether the signature can be properly encoded.

A strong signature forgery requires one to find a short vector $\mathbf{w} \in 2R_n^2 + \mathbf{h}$ such that $\|\mathbf{w}\|_{\mathbf{Q}} \leq 2 \cdot \sigma_{\mathsf{verify}}\sqrt{2n}$ holds for uniform $\mathbf{h} \in R_n^2$. Equivalently, one must find a vector $\mathbf{s} \in R_n^2$ that satisfies $\|\mathbf{h} - 2\mathbf{s}\|_{\mathbf{Q}} \leq 2 \cdot \sigma_{\mathsf{verify}}\sqrt{2n}$. Both of these conditions can be stated equivalently under $\|\cdot\|$ as finding a short vector in the coset $2R_n^2 + \mathbf{Bh}$ or a close vector in $2R_n^2$ to $\mathbf{Bh}$. We approach this cryptanalytic task via estimating the complexity of lattice reduction based approaches to finding close vectors in Section 5.

Algorithm Sketch 3 High level HawkVerify

Require: A message m, a public key $\mathsf{pk} = \mathbf{Q}$, and a signature $\mathsf{sig} = (\mathsf{salt}, s_1)$
Ensure: A bit determining whether sig is a valid signature on m

1: $\mathsf{hpub} \leftarrow \mathsf{H}(\mathbf{Q})$
2: $M \leftarrow \mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$
3: $\mathbf{h} \leftarrow \mathsf{H}(M \parallel \mathsf{salt})$
4: $\mathbf{s} \leftarrow \mathsf{Decompress}(s_1, \mathbf{h}, \mathbf{Q})$
5: $\mathbf{w} \leftarrow \mathbf{h} - 2\mathbf{s}$
6: **if** $\mathsf{len}_{\mathsf{bits}}(\mathsf{salt}) = \mathsf{saltlen}_{\mathsf{bits}}$ and $\mathbf{s} \in R_n^2$ and $\mathsf{sym\text{-}break}(\mathbf{w})$ and $\|\mathbf{w}\|_{\mathbf{Q}}^2 \leq 4 \cdot \sigma_{\mathsf{verify}}^2 \cdot 2n$ **then**
7: **return** 1
8: **else**
9: **return** 0

Verification. Verification in HAWK recomputes values internal to Algorithm 2 from the signature sig, the message m and pk. Ultimately it recomputes $\mathbf{h}$, then $\mathbf{s}$ via Decompress, and then $\mathbf{w}$. A series of conditions are then checked. Since both Compress and Decompress are public functions they do not affect security, and in Sections 5 and 6 we argue about a version of HAWK that does not perform these operations – i.e. we consider both Compress and Decompress to be the identity function on their first argument.

However, Compress and Decompress can affect correctness, whenever

$$\mathsf{Decompress}(\mathsf{Compress}(\mathbf{s}), \mathbf{h}, \mathbf{Q}) \neq \mathbf{s}. \tag{23}$$

By carefully selecting parameters we make sure this happens with a negligible probability.

3 Specifications

Here we give the specification for HAWK. In Section 3.1 we give more details regarding key generation, signing and verification. In Section 3.2, and in particular in Table 4, we give parameters for HAWK. In Section 3.3 we detail our encoding and decoding procedures. Finally, in Sections 3.4, 3.5 and 3.6 we detail key generation, signature generation and verification, respectively.

3.1 Overview

HAWK is a signature scheme with the following characteristics:

- Private keys, public keys and signatures use fixed size encodings. For public keys and signatures, a compression mechanism is used with variable output sizes, but padding is applied so that a fixed size specified in the scheme parameters is achieved. Key pair generation and signature generation are restarted if the public key or signature would exceed the target size.
- The signed data is processed once with SHAKE256; there is no random or key-dependent preamble, so that streamed processing of large messages with minimal RAM usage is possible, and the bulk of the data can be processed before knowing the public or private key (and even before knowing which HAWK parameter set is to be used).

Notation. We collect some additional notation that will be used throughout the formal specification.

The notation $x[i]$ is used to denote the i^{th} element of any sequence x of values. Indices always start at 0, i.e the first element of x is $x[0]$. The notation $x[j{:}k]$ stands for the subsequence of x for indices i such that $j \leq i < k$, i.e. the start index is included, but the end index is excluded, which corresponds to the notation used in several programming languages, such as Python, Go or Rust. Recall that the number of elements in a sequence x is denoted $\mathsf{len}(x)$. For additional clarity, when x is a sequence of bits, we may write $\mathsf{len}_{\mathsf{bits}}(x)$ to denote the number of bits in x.

To go between bit sequences (of arbitrary lengths) and numbers, we define the EncodeInt and DecodeInt functions.

- For an integer $k \geq 0$ and a k-bit sequence $x_0 x_1 \ldots x_{k-1}$, we define

$$\mathsf{DecodeInt}(x_0 x_1 \ldots x_{k-1}, k) = \sum_{i=0}^{k-1} x_i 2^i.$$

- For integers $k \geq 0$ and $0 \leq x < 2^k$, $\mathsf{EncodeInt}(x, k)$ is defined as the k-bit sequence $x_0 x_1 \ldots x_{k-1}$ for which $x = \mathsf{DecodeInt}(x_0 x_1 \ldots x_{k-1}, k)$ holds.

For $k = 0$ we define "$\mathsf{EncodeInt}(0, 0)$" to be the empty sequence. Moreover, for an integer x with $0 \leq x < 2^k$, we define $\mathsf{rev}_k(x)$ as the integer associated to the bit-reversal of $\mathsf{EncodeInt}(x, k)$, i.e.,

$$\mathsf{rev}_k\left(\sum_{j=0}^{k-1} x_j 2^j\right) = \sum_{j=0}^{k-1} x_j 2^{k-1-j}.$$

A cryptographically secure random source that outputs uniformly random bits is denoted Rnd. For an integer k, the notation $\mathsf{Rnd}(k)$ specifies the process of obtaining k new, freshly generated random bits.

3.1.1 Key Pairs

HAWK is defined over a degree n, which is a power of two (equal to 256, 512 or 1024). Most computations are performed on polynomials with integer coefficients, taken modulo $X^n + 1$. A HAWK private key is a randomly generated basis for the lattice $\mathbb{Z}^{2n}$, consisting of four polynomials f, g, F and G, where f and g have small coefficients and together they satisfy the NTRU equation

$$fG - gF = 1 \pmod{X^n + 1}, \tag{24}$$

i.e. the NTRU modulus is one. The lattice secret basis $\mathbf{B}$ and its inverse $\mathbf{B}^{-1}$ are

$$\mathbf{B} = \begin{pmatrix} f & F \\ g & G \end{pmatrix}, \qquad \mathbf{B}^{-1} = \begin{pmatrix} G & -F \\ -g & f \end{pmatrix}. \tag{25}$$

The public key is $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$, which is explicitly

$$\mathbf{Q} = \begin{pmatrix} q_{00} & q_{01} \\ q_{10} & q_{11} \end{pmatrix} = \begin{pmatrix} f^\star f + g^\star g & f^\star F + g^\star G \\ F^\star f + G^\star g & F^\star F + G^\star G \end{pmatrix}. \tag{26}$$

The matrix $\mathbf{Q}$ is self adjoint: $q_{00} = q_{00}^\star$, $q_{11} = q_{11}^\star$, and $q_{10} = q_{01}^\star$. Moreover, a consequence of the NTRU equation is that $q_{00}q_{11} - q_{01}q_{10} = 1$ holds. We can thus always reconstruct the whole of $\mathbf{Q}$ from q_{00} and q_{01} only.

The generation of a new key pair is the following process.

- Generate kgseed, a random seed, from a cryptographically secure source of random bits.
- Sample the coefficients of f and g in a deterministic manner seeded with kgseed from the centred binomial distribution with the appropriate parameter.
- Verify a few conditions on f and g, e.g. that they both have odd parity and that the ℓ^2 norm of (f, g) is not too small. If the conditions are not fulfilled, restart with a new seed.
- Find polynomials F and G that complete the basis, i.e. such that $fG - gF = 1$. Because there may not exist any solution (F, G), this step might fail. In that case, we restart with a new seed.
- Next, a well chosen multiple of (f, g) is subtracted from (F, G) to make (F, G) shorter. Optimised implementation methods, that combine completion and reduction, have been published previously [PP19, Por23]. This is the most expensive part of the key pair generation.
- The public key may also fail to be encodable within the target public key size defined in the scheme parameters; in that case, restart with a new seed.

The key pair generation process is detailed in Section 3.4. The actual encoding format of the private key only contains parts of the private elements, namely:

- kgseed (from which f and g can be regenerated using the specified deterministic process),
- F mod 2 and G mod 2 (i.e. the least significant bit of each coefficient of F and G), and
- hpub, a hash of the public key.

The public key itself encodes q_{00} and q_{01}, which are enough to represent the entire matrix $\mathbf{Q}$. Moreover, q_{00} being itself self adjoint, only half of its coefficients need to be encoded.

3.1.2 Signature Generation

Signature generation proceeds with the following steps.

- Decode the elements of the private key.
- Compute f, g from kgseed.
- Compute the hash $M = \mathsf{SHAKE256}(\mathsf{m} \parallel \mathsf{hpub})$. Recall hpub is part of the private key.
- Generate salt, a random salt value.
- Compute the binary vector $\mathbf{h} = \mathsf{SHAKE256}(M \parallel \mathsf{salt})$ of size $2n$ bits (the two coordinates of $\mathbf{h}$ are the polynomials h_0 and h_1, whose coefficients have value 0 or 1).
- Sample the vector $\mathbf{x}$ (of dimension $2n$) from a discrete Gaussian distribution, centred on zero and of width $2\sigma_{\mathsf{sign}}$, over the lattice coset $\mathbb{Z}^{2n} + \mathbf{Bh}$. Note that here we are forming $\mathbf{B}$ from the recomputed (f, g) and (F mod 2, G mod 2), which are sufficient to determine this coset. If the ℓ^2 norm of $\mathbf{x}$ exceeds a given threshold, restart with a new salt.
- Compute $\mathbf{w} = \mathbf{B}^{-1}\mathbf{x}$ and discard w_0. In Algorithm 15 only w_1 is calculated.
- Check whether w_1 fulfils a symmetry breaking condition (sym-break). If not, replace w_1 by $-w_1$. This step prevents a simple weak forgery attack.
- Compute $s_1 = (h_1 - w_1)/2$. Note that we could have equivalently computed $\mathbf{s} = (\mathbf{h} - \mathbf{w})/2$ and then discarded s_0. As such, this represents the application of Compress from the high level description in Algorithm 2.
- Encode salt and s_1. The encoding fails when a coefficient of s_1 is too large or the encoded object becomes too large for the fixed target signature size. If this happens, restart signing with a new salt.

The signature generation process is fully specified in Section 3.5. Including only the second entry of the vector $\mathbf{s} = (s_0, s_1)$ in the signature encoding format makes signatures smaller, and also speeds up signature generation since only s_1 has to be computed. However, it then requires the signature verifier to recompute s_0 from the signed data, the public key and s_1.

3.1.3 Signature Verification

The signature verification process is the following.

- Decode the public key $\mathbf{Q}$ and the two signature elements salt and s_1 from their respective formats.
- As in signature generation, recompute M as a hash over input data (m) and the hashed public key (hpub). Then recompute $\mathbf{h}$ as a hash over M and the salt (salt).
- Compute $w_1 = h_1 - 2 \cdot s_1$. This equals w_1 from the discussion in Section 3.1.2 above.
- Verify that sym-break(w_1) is fulfilled.
- Recompute s_0 using

$$s_0 = \left\lceil \frac{h_0}{2} + \frac{q_{01}}{q_{00}} \left(\frac{h_1}{2} - s_1 \right) \right\rfloor . \tag{27}$$

Table 4: Parameter sets for HAWK. Note that key and signature sizes are specified in bytes.

Name	HAWK-256	HAWK-512	HAWK-1024
Targeted security	Challenge	NIST-I	NIST-V
Bit security λ	64	128	256
Private key size $\mathsf{privlen}_{\mathsf{bits}}/8$ (bytes)	96	184	360
Public key size $\mathsf{publen}_{\mathsf{bits}}/8$ (bytes)	450	1024	2440
Signature size $\mathsf{siglen}_{\mathsf{bits}}/8$ (bytes)	249	555	1221
Degree n	256	512	1024
Transcript size limit q_s	2^{32}	2^{64}	2^{64}
Centred binomial η for sampling (f, g)	2	4	8
Signature std. dev. σ_{sign}	1.010	1.278	1.299
Verification std. dev. σ_{verify}	1.042	1.425	1.571
Key recovery std. dev. σ_{krsec}	1.042	1.425	1.974
Salt length $\mathsf{saltlen}_{\mathsf{bits}}$	112	192	320
Keygen. seed length $\mathsf{kgseedlen}_{\mathsf{bits}}$	128	192	320
Hashed pubkey length $\mathsf{hpublen}_{\mathsf{bits}}$	128	256	512
Bit lengths for q_{00} (low_{00}, high_{00})	5, 9	5, 9	6, 10
Bit lengths for q_{01} (low_{01}, high_{01})	8, 11	9, 12	10, 14
Bit length for q_{11} (high_{11})	13	15	17
Bit length for s_0 (high_{s0})	12	13	14
Bit lengths for s_1 (low_{s1}, high_{s1})	5, 9	5, 9	6, 10
Signature decompression bound β_0	1/250	1/1000	1/3000

This recomputation will be performed with integer computations using a fixed-point interpretation of 32-bit integers. This represents Decompress from the high level description in Algorithm 3.

- Compute $w_0 = h_0 - 2 \cdot s_0$. In the implementation this step and the one above both happen internally in RebuildS0 in Algorithm 20.
- Let $\mathbf{w} = (w_0, w_1)$. Verify that $\|\mathbf{w}\|_{\mathbf{Q}}$ is not too long.

Signature verification is detailed in Section 3.6. Compared to the high level description of Algorithm 3 the length of the decoded salt and $\mathbf{s} \in R_n^2$ are not checked explicitly. Instead, the decoding procedure only gives a salt of the desired length from the signature and an $s_1 \in R_n$ by construction of the encoding format. By design, s_0 is rebuilt as an integer by (27), so $\mathbf{s} \in R_n^2$ is automatically satisfied.

The scheme parameters and keys are chosen such that the probability, that (27) fails to recover s_0, is negligible.

3.2 Parameters

HAWK is defined for three sets of parameters, shown in Table 4. The name of a HAWK parameter set is derived from the degree n, which is taken to be a power of two.

The HAWK-256 parameter set is a "challenge" because its security level, estimated at "around 64 bits", is too low for submission to NIST. However, it is a good cryptanalytic target, as breaking it is not immediate and a forgery or private key recovery would provide interesting data. At most 2^{32} signatures may be "safely" generated within the target security level of HAWK-256, as opposed to 2^{64} for HAWK-512 and HAWK-1024.

HAWK-512 and HAWK-1024 are meant to fulfil NIST-I and NIST-V security levels, respectively.

3.3 Encoding and Decoding

Encoding and decoding rules are used in HAWK for representing keys and signatures, but also internally for injecting integers as input to SHAKE256, and interpreting the SHAKE256 output as a sequence of 64-bit integers.

3.3.1 Bit Ordering

All input and output operations are described as working over sequences of bits. Such a sequence is written in left-to-right order. The "$\|$" operator denotes concatenation. Most software implementations work over *bytes* (octets); in that case, it is assumed that bits have been grouped into bytes in a way compatible with algorithm *h2b* from section B.1 of FIPS 202 [FIP15], which itself follows the rules from [Kec11]: if eight successive bits are $x_0x_1x_2\ldots x_7$, then they encode the numerical byte value $x = \sum_{i=0}^{7} x_i 2^i$. We denote the mapping of a byte into 8 bits by $\mathsf{EncodeInt}(x, 8) = x_0 \,\|\, x_1 \,\|\, x_2 \,\|\, \ldots \,\|\, x_7$.

A potential source of confusion is the terminology of shift operators available in many programming languages: if some bits in a sequence should be moved *leftward*, then this corresponds to a *right shift* on the integer value corresponding to this sequence through the mapping embodied by EncodeInt and DecodeInt. Within this specification, we do not use the "left" and "right" directions; such operations on integers are expressed as multiplications or divisions by powers of two.

3.3.2 SHAKE256w and SHAKE256x4

SHAKE256, as specified in FIPS 202 [FIP15], outputs a sequence of bits. We define SHAKE256w as SHAKE256, with the output bits grouped into 64-bit words interpreted as integers. Specifically, for a binary input m, we define $\mathsf{SHAKE256w}(m)$ such that, for any $i \geq 0$,

$$\mathsf{SHAKE256w}(m)[i] = \sum_{j=0}^{63} \mathsf{SHAKE256}(m)[64i + j] \cdot 2^j \tag{28}$$

$$= \mathsf{DecodeInt}(\mathsf{SHAKE256}(m)[64i : 64i + 64]). \tag{29}$$

Internally, SHAKE256 implementations that do not use the "bit interleaving" strategy usually keep the sponge state as an array of 25 64-bit words, and the output of SHAKE256 is really a little-endian encoding of some of these state words; the word output stream defined here is then exactly the sequence of these words.

We then define the SHAKE256x4 function, which is four SHAKE256w instances working on almost the same input, and whose respective outputs are interleaved with word granularity. Namely, for a binary input m,

$$\mathsf{SHAKE256x4}(m)[4i + j] = \mathsf{SHAKE256w}(m \,\|\, \mathsf{EncodeInt}(j, 8))[i]$$

for all $i \geq 0$, and $0 \leq j \leq 3$.

An implementation of HAWK running on an architecture offering SIMD opcodes may choose to run some of these internal SHAKE256 instances in parallel; the interleaving naturally corresponds to the effect of running four parallel instances with 256-bit SIMD registers, each register containing one SHAKE256 state 64-bit word for each of the four SHAKE256 instances. However, it is also possible to run the SHAKE256 instances sequentially, reusing the same memory space for the successive SHAKE256 states; all usages of SHAKE256x4 output within HAWK have been designed specifically to allow such RAM efficient usage on small embedded systems with severe RAM size constraints.

3.3.3 Low Bit Extraction

Given a polynomial u with integer coefficients, $u \bmod 2$ denotes the polynomial such that $(u \bmod 2)[i] = u[i] \bmod 2$ for $0 \leq i < n$; i.e. each coefficient of $u \bmod 2$ is the least significant bit (with value 0 or 1) of the corresponding coefficient of u.

Since polynomials are also sequences of their coefficients, a binary polynomial $u \bmod 2$ is also a sequence of n bits ($u[0] \bmod 2$ to $u[n-1] \bmod 2$) and can be used as such in encoding and decoding procedures.

3.3.4 Private Key Encoding

A HAWK private key, mathematically, consists of four polynomials f, g, F and G. As will be detailed in Section 3.4, f and g are generated from a random seed (kgseed), and only $F \bmod 2$ and $G \bmod 2$ are needed for signature generation. The encoding of the private key is thus the concatenation of kgseed, $F \bmod 2$, $G \bmod 2$, and hpub, the latter being a hash of the public key. This is expressed in EncodePrivate (Algorithm 4) and DecodePrivate (Algorithm 5). Note that the lengths of kgseed and hpub are fixed: for a given HAWK parameter set, they are defined in the scheme parameters (Table 4).

Algorithm 4 EncodePrivate: Private key encoding

Require: Private seed kgseed, polynomials $(F \bmod 2, G \bmod 2)$, hashed public key hpub
Ensure: Encoded private key
1: **return** kgseed $\|$ $(F \bmod 2)$ $\|$ $(G \bmod 2)$ $\|$ hpub

Algorithm 5 DecodePrivate: Private key decoding

Require: Encoded private key priv
Ensure: Private seed kgseed, polynomials $(F \bmod 2, G \bmod 2)$, and hashed public key hpub
1: kgseed $\leftarrow$ priv$[0 : \mathsf{kgseedlen}_{\mathsf{bits}}]$
2: $F \bmod 2 \leftarrow$ priv$[\mathsf{kgseedlen}_{\mathsf{bits}} : \mathsf{kgseedlen}_{\mathsf{bits}} + n]$
3: $G \bmod 2 \leftarrow$ priv$[\mathsf{kgseedlen}_{\mathsf{bits}} + n : \mathsf{kgseedlen}_{\mathsf{bits}} + 2n]$
4: hpub $\leftarrow$ priv$[\mathsf{kgseedlen}_{\mathsf{bits}} + 2n : \mathsf{kgseedlen}_{\mathsf{bits}} + 2n + \mathsf{hpublen}_{\mathsf{bits}}]$
5: **return** (kgseed, $F \bmod 2, G \bmod 2$, hpub)

3.3.5 Golomb–Rice Compression

Polynomials in HAWK public keys and signatures use a variable length encoding with Golomb–Rice compression of coefficients. Each integer element is split into a sign bit, a fixed length part comprising the low order bits of the value, and a variable length part that encodes the high order bits of the value. For a sequence of k values to encode, the k sign bits are first produced, followed by the k low parts of the values, then the k high parts. In all uses of this encoding in HAWK, k is a multiple of 8, ensuring that the sign bit chunk and the low part chunk start on byte boundaries.

The CompressGR function (Algorithm 6) takes as input a sequence of integers x (of length k, such that $k \equiv 0 \bmod 8$), as well as two sizes, low and high; the coefficients of x must be such that $-2^{\mathsf{high}} \leq x[i] < 2^{\mathsf{high}}$ for all $0 \leq i < k$.

In plain words, for each integer value $z = x[i]$ we perform the following:

- A sign bit is emitted, of value 0 if $z \geq 0$, or 1 if $z < 0$. In the latter case, z is replaced with its ones' complement $-z - 1$ (in a typical software implementation,

Algorithm 6 CompressGR: Golomb–Rice compression

Require: Integer sequence $x[0{:}k]$, sizes low and high
Ensure: A compressed encoding y (sequence of bits), or $\perp$ on error

```
Set y to an empty sequence of bits.
Set v to an empty sequence of integers.
for i = 0 to k − 1 do
    s ← 1 if x[i] < 0, or 0 if x[i] ≥ 0
    y ← y ‖ s
    v ← v ‖ x[i] − s(2x[i] + 1)
    if v[i] ≥ 2^high then
        return ⊥
for i = 0 to k − 1 do
    y ← y ‖ EncodeInt(v[i] mod 2^low, low)
for i = 0 to k − 1 do
    y ← y ‖ EncodeInt(0, ⌊v[i]/2^low⌋) ‖ 1
return y
```

this is equivalent to a XOR between z and $-s$, where s is the sign bit). Note that now $z \geq 0$.

- The lowest bits of z are emitted as is; this is the fixed size part. Then z is scaled down by low bits (floored division by 2^{low}).
- Finally, exactly z bits of value 0 are emitted, followed by a bit of value 1 which marks the end of the encoding for that integer. This is the variable size part.

In all usages of CompressGR in HAWK, the maximum size is such that $\mathsf{high} \leq \mathsf{low} + 4$. As a consequence the variable size part of an encoded integer cannot be longer than 16 bits (including the terminating 1).

Algorithm 6 returns a failure indicator ($\perp$) if the input is invalid, i.e. at least one of the input integers is not in the expected range. This check may be omitted if the input is known to be in the proper range (e.g. because it has already been tested beforehand).

Decompression is performed with DecompressGR (Algorithm 7). Decompression can return a failure (denoted $\perp$) if the input is invalid. On success, the resulting integer sequence is returned, as well as the actual encoded length (in bits); it is up to the caller to verify that subsequent bits (e.g. padding), if any, have the expected value.

3.3.6 Public Key Encoding

A HAWK public key consists of four polynomials: q_{00}, q_{01}, q_{10} and q_{11}. As explained in Section 3.1.1 we only need to encode q_{00} and q_{01} in the public key. Moreover, q_{00} is self adjoint, so only its first $n/2$ coefficients must be serialized. The EncodePublic function (Algorithm 8) returns the encoded format for a public key.

All coefficients of q_{00} are lower than $2^{\mathsf{high}_{00}}$, except the first coefficient, $q_{00}[0]$, which may be larger, and is thus treated specially by being first scaled down by a few bits; its low bits are then appended to the encoding of q_{00}, followed by up to 7 extra bits (of value 0) to ensure that the encoding of q_{01} starts on a byte boundary. Padding bits (also of value 0) are finally placed after q_{01} so that the total encoded size matches the expected value. The key pair generation process should ensure that the encoded public key can indeed fit in the target size. If not, the key should be discarded, and a new one generated.

The corresponding decoding algorithm is DecodePublic (Algorithm 9). It is noteworthy that DecodePublic enforces that the exact input length should match the public key length

Algorithm 7 DecompressGR: Golomb–Rice decompression

Require: Bit sequence y, output length $k \equiv 0 \bmod 8$, sizes low and high
Ensure: Integers $x[0{:}k]$ and encoded length j, or $\perp$ (failure)

if $\mathsf{len}_{\mathsf{bits}}(y) < k(\mathsf{low}+2)$ **then**
  return $\perp$
for $i = 0$ to $k-1$ **do**
  $x[i] \leftarrow \mathsf{DecodeInt}(y[i \cdot \mathsf{low} + k : (i+1) \cdot \mathsf{low} + k])$
$j \leftarrow k(\mathsf{low}+1)$
for $i = 0$ to $k-1$ **do**
  $z \leftarrow -1$
  repeat
    $z \leftarrow z + 1$
    if $j \geq \mathsf{len}_{\mathsf{bits}}(y)$ or $z \geq 2^{\mathsf{high}-\mathsf{low}}$ **then**
      return $\perp$
    $t \leftarrow y[j]$
    $j \leftarrow j + 1$
  until $t = 1$
  $x[i] \leftarrow x[i] + z \cdot 2^{\mathsf{low}}$
for $i = 0$ to $k-1$ **do**
  $x[i] \leftarrow x[i] - y[i](2x[i]+1)$ ▷ Application of the sign bit.
return (x, j)

Algorithm 8 EncodePublic: Public key encoding

Require: Public polynomials q_{00} and q_{01}
Ensure: Encoded public key, or $\perp$ on error

if $q_{00}[0] < -2^{15}$ or $q_{00}[0] \geq 2^{15}$ **then**
  return $\perp$
$v \leftarrow 16 - \mathsf{high}_{00}$
$q'_{00} \leftarrow q_{00}$
$q'_{00}[0] \leftarrow \lfloor q_{00}[0]/2^v \rfloor$
$y_{00} \leftarrow \mathsf{CompressGR}(q'_{00}[0 : n/2], \mathsf{low}_{00}, \mathsf{high}_{00})$ ▷ Only $n/2$ coefficients.
if $y_{00} = \perp$ **then**
  return $\perp$
$y_{00} \leftarrow y_{00} \parallel \mathsf{EncodeInt}(q_{00}[0] \bmod 2^v, v)$
while $\mathsf{len}_{\mathsf{bits}}(y_{00}) \not\equiv 0 \bmod 8$ **do**
  $y_{00} \leftarrow y_{00} \parallel 0$ ▷ Padding to the next byte boundary.
$y_{01} \leftarrow \mathsf{CompressGR}(q_{01}, \mathsf{low}_{01}, \mathsf{high}_{01})$ ▷ All n coefficients are encoded.
if $y_{01} = \perp$ **then**
  return $\perp$
$y \leftarrow y_{00} \parallel y_{01}$
if $\mathsf{len}_{\mathsf{bits}}(y) > \mathsf{publen}_{\mathsf{bits}}$ **then**
  return $\perp$
while $\mathsf{len}_{\mathsf{bits}}(y) < \mathsf{publen}_{\mathsf{bits}}$ **do**
  $y \leftarrow y \parallel 0$ ▷ Padding to $\mathsf{publen}_{\mathsf{bits}}$ bits.
return y

$\mathsf{publen}_{\mathsf{bits}}$, as defined in Table 4, and all padding bits (both between q_{00} and q_{01}, and after q_{01}) should be zero. A given public key has a single valid encoding, and none other shall be accepted by DecodePublic.

Algorithm 9 DecodePublic: Public key decoding

Ensure: Encoded public key y
Require: Public polynomials q_{00} and q_{01}, or $\perp$ on error.
1: **if** $\mathsf{len}_{\mathsf{bits}}(y) \neq \mathsf{publen}_{\mathsf{bits}}$ **then**
2: **return** $\perp$
3: $v \leftarrow 16 - \mathsf{high}_{00}$
4: $r_{00} \leftarrow \mathsf{DecompressGR}(y, n/2, \mathsf{low}_{00}, \mathsf{high}_{00})$
5: **if** $r_{00} = \perp$ **then**
6: **return** $\perp$
7: $(q_{00}, j) \leftarrow r_{00}$
8: **if** $\mathsf{len}_{\mathsf{bits}}(y) < j + v$ **then**
9: **return** $\perp$
10: $q_{00}[0] \leftarrow 2^v q_{00}[0] + \mathsf{DecodeInt}(y[j : j + v])$
11: $j \leftarrow j + v$
12: **while** $j \not\equiv 0 \bmod 8$ **do**
13: **if** $j \geq \mathsf{len}_{\mathsf{bits}}(y)$ or $y[j] \neq 0$ **then**
14: **return** $\perp$
15: $j \leftarrow j + 1$
16: $q_{00}[n/2] \leftarrow 0$
17: **for** $i = n/2 + 1$ to $n - 1$ **do**
18: $q_{00}[i] \leftarrow -q_{00}[n - i]$
19: $r_{01} \leftarrow \mathsf{DecompressGR}(y[j : \mathsf{len}_{\mathsf{bits}}(y)], n, \mathsf{low}_{01}, \mathsf{high}_{01})$
20: **if** $r_{01} = \perp$ **then**
21: **return** $\perp$
22: $(q_{01}, j') \leftarrow r_{01}$
23: $j \leftarrow j + j'$
24: **while** $j < \mathsf{len}_{\mathsf{bits}}(y)$ **do**
25: **if** $y[j] \neq 0$ **then**
26: **return** $\perp$
27: $j \leftarrow j + 1$
28: **return** (q_{00}, q_{01})

3.3.7 Signature Encoding

A HAWK signature consists of a salt value (salt) and a polynomial s_1. These two elements are encoded and concatenated in the function EncodeSignature (Algorithm 10) to yield the signature. Again, zeros are padded to produce a fixed length signature. If the signature would be bigger than the fixed length, encoding fails. However, the failure is resolved in signing by restarting the signing process, which is a randomised process. The parameters for HAWK have been chosen so that such failures are rare, see Section 3.5.

The corresponding decoding algorithm is DecodeSignature (Algorithm 11). Similarly to public keys, decoding enforces a signature to be of size $\mathsf{siglen}_{\mathsf{bits}}$ from Table 4 and all padding bits at the end should be 0.

Algorithm 10 EncodeSignature: Signature encoding

Require: Salt salt and signature polynomial s_1
Ensure: Encoded signature, or $\perp$ on error
1: $y \leftarrow \mathsf{CompressGR}(s_1, \mathsf{low}_{s1}, \mathsf{high}_{s1})$
2: **if** $y = \perp$ or $\mathsf{len}_{\mathsf{bits}}(y) > \mathsf{siglen}_{\mathsf{bits}} - \mathsf{saltlen}_{\mathsf{bits}}$ **then**
3: **return** $\perp$
4: **while** $\mathsf{len}_{\mathsf{bits}}(y) < \mathsf{siglen}_{\mathsf{bits}} - \mathsf{saltlen}_{\mathsf{bits}}$ **do**
5: $y \leftarrow y \parallel 0$
6: **return** $\mathsf{salt} \parallel y$

Algorithm 11 DecodeSignature: Signature decoding

Require: Encoded signature y
Ensure: Salt salt and polynomial s_1, or $\perp$ on error.
1: **if** $\mathsf{len}_{\mathsf{bits}}(y) \neq \mathsf{siglen}_{\mathsf{bits}}$ **then**
2: **return** $\perp$
3: $\mathsf{salt} \leftarrow y[0 : \mathsf{saltlen}_{\mathsf{bits}}]$
4: $r \leftarrow \mathsf{DecompressGR}(y[\mathsf{saltlen}_{\mathsf{bits}} : \mathsf{len}_{\mathsf{bits}}(y)], n, \mathsf{low}_{s1}, \mathsf{high}_{s1})$
5: **if** $r = \perp$ **then**
6: **return** $\perp$
7: $(s_1, j) \leftarrow r$
8: $j \leftarrow j + \mathsf{saltlen}_{\mathsf{bits}}$
9: **while** $j < \mathsf{len}_{\mathsf{bits}}(y)$ **do**
10: **if** $y[j] \neq 0$ **then**
11: **return** $\perp$
12: $j \leftarrow j + 1$
13: **return** (salt, s_1)

3.4 Key Pair Generation

The HAWK key pair generation process consists of trying random candidates for two polynomials (f, g) with a given distribution, then for each suitable candidate, solving the NTRU equation $fG - gF = 1$. If an appropriate solution is found, and the resulting public key can be encoded within the size defined in the HAWK parameters, then that key pair is kept; otherwise, the candidate (f, g) is discarded, and the process loops.

The polynomials f and g are sampled from a centred binomial distribution using pseudorandom bits obtained from a SHAKE256x4 instance initialised over a random seed (kgseed). This process is implemented by the function Regeneratefg (Algorithm 12). For a given seed value, the function is deterministic and the output (f, g) is fully specified, so kgseed is sufficient for the private key storage.

Algorithm 12 Regeneratefg: Regenerate (f, g)

Require: Key generation seed kgseed
Ensure: Polynomials (f, g)

$b \leftarrow n/64$ ▷ $b = 4$, 8 or 16, depending on n.
$y \leftarrow \mathsf{SHAKE256x4}(\mathsf{kgseed})[0 : 2bn]$ ▷ b bits for each coefficient of f and g.
for $i = 0$ to $n - 1$ **do**
    $f[i] \leftarrow \left(\sum_{j=0}^{b-1} y[ib + j]\right) - b/2$ ▷ centred binomial with $\eta = b/2$.
for $i = 0$ to $n - 1$ **do**
    $g[i] \leftarrow \left(\sum_{j=0}^{b-1} y[(i + n)b + j]\right) - b/2$
return (f, g)

The key pair generation process is defined in the function HawkKeyGen (Algorithm 13), which makes use of the following two helper functions.

- IsInvertible(u, p) returns true if the integer polynomial u is invertible modulo $X^n + 1$ and modulo the provided prime p (i.e. considering the coefficients of u as elements in the field $\mathbb{Z}/p\mathbb{Z}$), and false otherwise. When $p = 2$, this simplifies to a parity check: IsInvertible$(u, 2)$ returns true if and only if $\sum_{i=0}^{n-1} u[i] \equiv 1 \pmod 2$. In HawkKeyGen, this function is also used for two specific 31-bit moduli amenable to efficient implementations using the NTT.

- NTRUSolve(f, g, q), for polynomials (f, g) and integer q, return polynomials (F, G) such that $fG - gF = q$, and (F, G) have somewhat small coefficients. A solution does not necessarily exist, and when solutions do exist NTRUSolve may fail to find one, in which case the function returns $\perp$. Note that NTRUSolve may succeed, but (F, G) may not be short enough, in which case HawkKeyGen also restarts. In [PP19], the algorithms TowerSolverR and TowerSolverI are presented, which can both be used to implement NTRUSolve.

Implementation strategies for these helper functions are discussed in more detail in Section 4.1.

In summary, a HAWK private key describes four secret integer polynomials f, g, F and G (taken modulo $X^n + 1$). The vector (f, g) is of small norm and satisfies

$$\|(f, g)\|_\infty \leq n/128, \text{ and } \|(f, g)\|^2 > 2n\sigma_{\mathsf{krsec}}^2. \tag{30}$$

Then, NTRUSolve comes up with (F, G) such that

$$\|(F, G)\|_\infty \leq 127, \text{ and } fG - gF = 1. \tag{31}$$

Algorithm 13 HawkKeyGen: HAWK key pair generation

Require: Cryptographically secure source of random bits
Ensure: New key pair $(\mathsf{priv}, \mathsf{pub})$

$\mathsf{kgseed} \leftarrow \mathsf{Rnd}(\mathsf{kgseedlen}_{\mathsf{bits}})$ ▷ $\mathsf{kgseedlen}_{\mathsf{bits}}$ is defined in Table 4.
$(f, g) \leftarrow \mathsf{Regeneratefg}(\mathsf{kgseed})$
if $\mathsf{IsInvertible}(f, 2) \neq \mathsf{true}$ or $\mathsf{IsInvertible}(g, 2) \neq \mathsf{true}$ **then**
    restart
if $\|(f, g)\|^2 \leq 2n\sigma_{\mathsf{krsec}}^2$ **then**
    restart
$q_{00} \leftarrow ff^\star + gg^\star$
$(p_1, p_2) \leftarrow (2147473409, 2147389441)$
if $\mathsf{IsInvertible}(q_{00}, p_1) \neq \mathsf{true}$ or $\mathsf{IsInvertible}(q_{00}, p_2) \neq \mathsf{true}$ **then**
    restart
if $(1/q_{00})[0] \geq \beta_0$ **then** ▷ Inverse over $\mathbb{Q}[X]/(X^n + 1)$.
    restart
$r \leftarrow \mathsf{NTRUSolve}(f, g, 1)$
if $r = \perp$ **then**
    restart
$(F, G) \leftarrow r$
if $\|(F, G)\|_\infty > 127$ **then**
    restart
$q_{01} \leftarrow Ff^\star + Gg^\star$
$q_{11} \leftarrow FF^\star + GG^\star$
if $|q_{11}[i]| \geq 2^{\mathsf{high}_{11}}$ for any $i > 0$ **then**
    restart
$\mathsf{pub} \leftarrow \mathsf{EncodePublic}(q_{00}, q_{01})$
if $\mathsf{pub} = \perp$ **then**
    restart
$\mathsf{hpub} \leftarrow \mathsf{SHAKE256}(\mathsf{pub})$ ▷ $\mathsf{hpublen}_{\mathsf{bits}}$ is defined in Table 4.
$\mathsf{priv} \leftarrow \mathsf{EncodePrivate}(\mathsf{kgseed}, F \bmod 2, G \bmod 2, \mathsf{hpub})$
return $(\mathsf{priv}, \mathsf{pub})$

A HAWK public key describes the four elements q_{00}, q_{01}, q_{10} and q_{11} that make up $\mathbf{Q}$. The key generation HawkKeyGen ensures that

$$-2^{15} \leq q_{00}[0] < +2^{15}, \tag{32}$$

$$-2^{\mathsf{high}_{00}} < q_{00}[i] < +2^{\mathsf{high}_{00}} \qquad \text{for } i > 0, \tag{33}$$

$$-2^{\mathsf{high}_{01}} < q_{01}[i] < +2^{\mathsf{high}_{01}} \qquad \text{for all } i, \tag{34}$$

$$-2^{\mathsf{high}_{11}} < q_{11}[i] < +2^{\mathsf{high}_{11}} \qquad \text{for } i > 0. \tag{35}$$

3.5 Signature Generation

As specified in Section 3.1.2, generating a signature requires to sample from a discrete Gaussian with support $2\mathbb{Z}$ or $2\mathbb{Z}+1$. First, we detail the discrete Gaussian sampler used in signature generation. Then, we explain all the steps done in signature generation.

3.5.1 Discrete Gaussian Sampling

Signature generation nominally entails sampling a vector $\mathbf{x}$ from a discrete Gaussian distribution over the lattice coset $2\mathbb{Z}^{2n} + \mathbf{t}$ (with the coefficients of $\mathbf{t}$ taking values 0 or 1), centred on zero and with width $2\sigma_{\mathsf{sign}}$.

Recall the discrete Gaussian distribution given in (4). For any $b \in \{0, 1\}$ and $z \in \mathbb{R}$, let $P_b(z)$ denote the probability that $|X| \geq z$ when $X \sim D_{2\mathbb{Z}+b, 2\sigma_{\mathsf{sign}}}$. From P_0 and P_1, we then define two tables of integers, indexed by an integer $k \in \mathbb{Z}_{\geq 0}$, that use an up-scaling by a factor 2^{78}:

$$T_0[k] = \lfloor 2^{78} \cdot P_0(2 + 2k) \rfloor, \tag{36}$$

$$T_1[k] = \lfloor 2^{78} \cdot P_1(3 + 2k) \rfloor. \tag{37}$$

These integers are provided in hexadecimal format in Table 5.

Using tables T_0 and T_1, we define the function SamplerSign (Algorithm 14) which, given an initial seeding input seed (a sequence of bits) and a centre vector $\mathbf{t}$ (whose coefficients are 0 or 1), samples a vector $\mathbf{x}$ from a distribution that approximates $D_{2\mathbb{Z}^{2n}+\mathbf{t}, 2\sigma_{\mathsf{sign}}}$. We discuss the conditions we require on this approximate distribution more in Section 6.7. Here $\mathbf{x}$ is considered to be a sequence of $2n$ integers.

The following notes apply to Algorithm 14:

- SHAKE256x4 is used to produce pseudorandom 64-bit words. Since each sampled value consumes 80 bits, five words are needed for every four output values. A total of $5n/2$ words ($160n$ bits) is thus needed for the $2n$ output values.
- SHAKE256x4 runs four SHAKE256 instances in parallel. The indexing used above ensures that each output value uses bits from only one of these four instances, so that the SHAKE256 instances may be evaluated sequentially on RAM-starved architectures. The j index is really a designator of the relevant SHAKE256 instance within SHAKE256x4. The first SHAKE256 instance ($j = 0$) produces words used for output values 0, 1, 2, 3, then 16, 17, 18, 19, and so on; the second SHAKE256 instance ($j = 1$) produces words used for output values 4, 5, 6, 7, then 20, 21, 22, 23, and so on; the third SHAKE256 instance ($j = 2$) produces words used for output values 8, 9, 10, 11, then 24, 25, 26, 27, and so on; and the fourth SHAKE256 instance ($j = 3$) produces words used for output values 12, 13, 14, 15, then 28, 29, 30, 31, and so on.
- For each group of five successive words q_0 to q_4 produced by a single SHAKE256 instance, we can sample four values. Each value needs a 78-bit integer z and a sign bit. Words q_0 to q_3 contribute the low 63 bits of each z, and the sign bits; word q_4 contains the high 15 bits of each z. Four of the bits of q_4 are discarded.

Table 5: Cumulative distribution tables for sampling used in signature generation.

Name	T_0	T_1
HAWK-256	0x26B871FBD58485D45050	0x13459408A4B181C718B1
	0x07C054114F1DC2FA7AC9	0x027D614569CC54722DC9
	0x00A242F74ADDA0B5AE61	0x0020951C5CDCBAFF49A3
	0x0005252E2152AB5D758B	0x0000A3460C30AC398322
	0x00000FDE62196C1718FC	0x000001355A8330C44097
	0x000000127325DDF8CEBA	0x00000000DC8DE401FD12
	0x0000000008100822C548	0x00000000003B0FFB28F0
	0x00000000000152A6E9AE	0x00000000000005EFCD99
	0x0000000000000014DA4A	0x0000000000000003953
	0x000000000000000007B	0x0000000000000000000
HAWK-512	0x2C058C27920A04F8F267	0x1AFCBC689D9213449DC9
	0x0E9A1C4FF17C204AA058	0x06EBFB908C81FCE3524F
	0x02DBDE63263BE0098FFD	0x01064EBEFD8FF4F07378
	0x005156AEDFB0876A3BD8	0x0015C628BC6B23887196
	0x0005061E21D588CC61CC	0x0000FF769211F07B326F
	0x00002BA568D92EEC18E7	0x00000668F461693DFF8F
	0x000000CF0F8687D3B009	0x0000001670DB65964485
	0x0000000216A0C344EB45	0x000000002AB6E11C2552
	0x0000000002EDF0B98A84	0x00000000002C253C7E81
	0x0000000000023AF3B2E7	0x00000000000018C14ABF
	0x00000000000000EBCC6A	0x0000000000000007876E
	0x000000000000000034CF	0x000000000000000013D
	0x0000000000000000006	0x0000000000000000000
HAWK-1024	0x2C583AAA2EB76504E560	0x1B7F01AE2B17728DF2DE
	0x0F1D70E1C03E49BB683E	0x07506A00B82C69624C93
	0x031955CDA662EF2D1C48	0x01252685DB30348656A4
	0x005E31E874B355421BB7	0x001A430192770E205503
	0x000657C0676C029895A7	0x00015353BD4091AA96DB
	0x00003D4D67696E51F820	0x000009915A53D8667BEE
	0x0000014A1A8A93F20738	0x000000266700301 60D5F
	0x00000003DAF47E8DFB21	0x00000000557CD1C5F797
	0x0000000006634617B3FF	0x00000000006965E15B13
	0x000000000005DBEFB646	0x00000000000047E9AB38
	0x00000000000002F93038	0x0000000000000001B2445
	0x0000000000000000D5A7	0x00000000000000005AA
	0x0000000000000000021	0x0000000000000000000

Algorithm 14 SamplerSign: Gaussian sampling in HAWK signature generation

Require: Seeding input seed, centre vector **t**
Ensure: **x**

$y \leftarrow \mathsf{SHAKE256x4}(s)[0 : 5n/2]$ ▷ $5n/2$ 64-bit words = $160n$ bits
for $j = 0$ to 3 **do**
  for $i = 0$ to $n/8 - 1$ **do**
    for $k = 0$ to 3 **do**
      $r \leftarrow 16i + 4j + k$
      $a \leftarrow y[j + 4(5i + k)]$
      $b \leftarrow \lfloor y[j + 4(5i + 4)]/2^{16k} \rfloor \bmod 2^{15}$
      $c \leftarrow (a \bmod 2^{63}) + 2^{63}b$
      $(v_0, v_1) \leftarrow (0, 0)$
      $z \leftarrow 0$
      while $T_0[z] \neq 0$ and $T_1[z] \neq 0$ **do**
        if $c < T_0[z]$ **then**
          $v_0 \leftarrow v_0 + 1$
        if $c < T_1[z]$ **then**
          $v_1 \leftarrow v_1 + 1$
        $z \leftarrow z + 1$
      if $\mathbf{t}[r] = 0$ **then**
        $v \leftarrow 2v_0$
      else
        $v \leftarrow 2v_1 + 1$
      if $a \geq 2^{63}$ **then**
        $v \leftarrow -v$
      $\mathbf{d}[r] \leftarrow v$
return **d**

- The actual sampling is done by counting how many elements of T_0 and T_1 are strictly greater than z; thus, lower z values yield larger outputs. Ultimately, only one of the two results v_0 and v_1 is retained, depending on the corresponding bit in $\mathbf{t}$; Algorithm 14 still uses *both* tables because that corresponds to what a constant-time implementation of HAWK should do: the $\mathbf{t}$ vector is secret, and thus the memory access pattern should not depend on whether $\mathbf{t}[r]$ is 0 or 1. Similarly, all table elements should be used each time (no early abort), and the application of the sign bit (conditional replacement of v with $-v$) should be done in a constant-time way.
- The number of iterations of the inner "while" loop depends on the scheme parameters. The T_0 and T_1 tables formally contain an infinite number of entries, but they only contain zeros beyond a certain index that depends on the scheme parameters. As seen in Table 5, with the currently defined parameter sets, only up to at most 13 iterations are needed.

3.5.2 Generating signatures

Using SamplerSign, we can now define the HawkSign function (Algorithm 15), which generates a HAWK signature on an arbitrary message. HawkSign also uses the sym-break function, defined as follows, for a given polynomial w:

- If there exists an index i such that $w[i] > 0$, and $w[j] = 0$ for all $j < i$, then $\mathsf{sym\text{-}break}(w) = \mathsf{true}$.
- Otherwise, $\mathsf{sym\text{-}break}(w) = \mathsf{false}$.

In other words, $\mathsf{sym\text{-}break}(w)$ is true only if w is not zero, and its first non-zero coefficient is positive.

Algorithm 15 HawkSign: HAWK signature generation

Require: Message m, private key priv, secure random source
Ensure: Signature sig

$(\mathsf{kgseed}, F \bmod 2, G \bmod 2, \mathsf{hpub}) \leftarrow \mathsf{DecodePrivate}(\mathsf{priv})$
$(f, g) \leftarrow \mathsf{Regeneratefg}(\mathsf{kgseed})$
$M \leftarrow \mathsf{SHAKE256}(\mathsf{m} \parallel \mathsf{hpub})[0:512]$ ▷ M has size 64 bytes.
$a \leftarrow 0$
loop
  $\mathsf{salt} \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{kgseed} \parallel \mathsf{EncodeInt}(a, 32) \parallel \mathsf{Rnd}(\mathsf{saltlen}_{\mathsf{bits}}))[0:\mathsf{saltlen}_{\mathsf{bits}}]$
  $(h_0, h_1) \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{salt})[0:2n]$
  $(t_0, t_1) \leftarrow ((h_0 f + h_1 F) \bmod 2, (h_0 g + h_1 G) \bmod 2)$
  $\mathsf{seed} \leftarrow M \parallel \mathsf{kgseed} \parallel \mathsf{EncodeInt}(a+1, 32) \parallel \mathsf{Rnd}(320)$
  $(x_0, x_1) \leftarrow \mathsf{SamplerSign}(\mathsf{seed}, (t_0, t_1))$
  $a \leftarrow a + 2$
  if $\|(x_0, x_1)\|^2 > 8n\sigma^2_{\mathsf{verify}}$ **then**
    continue loop
  $w_1 \leftarrow f d_1 - g d_0$
  if $\mathsf{sym\text{-}break}(w_1) = \mathsf{false}$ **then**
    $w_1 \leftarrow -w_1$
  $s_1 \leftarrow (h_1 - w_1)/2$
  $\mathsf{sig} \leftarrow \mathsf{EncodeSignature}(\mathsf{salt}, s_1)$
  if $\mathsf{sig} \neq \perp$ **then**
    return sig

The following notes apply to HawkSign:

- The message (m) is used once, at the start of a SHAKE256 instance, and is followed by the public key hash. This allows streamed processing, i.e. data can be signed as it flows without buffering the entire message. Moreover, this initial hashing process is independent of the signing key or any randomly chosen value.
- The signature salt (salt) and the sampling seed (seed) are both obtained from a SHAKE256 hash, computed over the concatenation of the input message (m), the private key seed (kgseed), an attempt counter (a or $a+1$), and some random bits. This is a *derandomisation* mechanism that provides extra protection in case the random source happens not to be cryptographically strong in a practical situation; in the extreme case where Rnd is fully non-random and outputs only zeros, then this step degrades to deterministic signatures, but it is still safe. It is nonetheless recommended to inject true random bits, or at least some varying data (even if not unpredictable), since that improves resistance to some active physical attacks (fault attacks). Also note that a cryptographically strong random source, indistinguishable from true randomness, is still a strict requirement for key pair generation (HawkKeyGen).
- Conversely to the derandomisation process explained above, it is possible to generate safe and interoperable signatures by generating the salt and the random bits used in SamplerSign directly from a cryptographically strong random source, thus bypassing all use of SHAKE256 on private data. Such alternative implementations can be useful on constrained devices for which SHAKE256 is expensive but a cheaper safe RNG is available, e.g. CTR-DRBG built over a hardware AES-256 implementation.
- Since h_0 and h_1 are binary polynomials, only $F \bmod 2$ and $G \bmod 2$ are needed, not the full F and G.
- The computation of sym-break does not need to be performed in constant-time since w_1 is a public value, and for any sampled (x_0, x_1), the opposite $(-x_0, -x_1)$ (that would lead to the opposite result for sym-break) could have been obtained with the same probability.
- We note that if w_1 is the zero polynomial, then both sym-break(w_1) and sym-break$(-w_1)$ are false. In this case, a valid signature may fail to pass verification, see Algorithm 20. However, for $w_1 = 0$ to be generated in Algorithm 15 it must be that $h_1 = 0$, since $w_1 \in 2R_n^2 + h_1$. We therefore assume this event happens with probability approximately 2^{-n} and disregard it.
- The whole process is organised as a loop because it can fail for three possible reasons:
  - The sampled (x_0, x_1) has a larger than acceptable ℓ^2 norm; this may happen with probability less than 2^{-17} for HAWK-512 and less than 2^{-121} for HAWK-1024 [DPPvW22b, Sec. 5.1].
  - One of the coefficients of s_1 is lower than $-2^{\mathsf{high}_{s1}}$ or greater than $2^{\mathsf{high}_{s1}} - 1$, making it incompatible with CompressGR. This is extremely rare.
  - The signature encoding (EncodeSignature) yields an output larger than the size defined in the scheme parameters. This is uncommon, because the target sizes have been set using the average encoded signature size (before padding) plus at least 4.5 times the standard deviation; however, it does happen in practice, on an occasional basis.

  Most signature attempts still succeed the first time. Retries do not have a significant impact on average performance.

  In particular, over ten million HAWK-512 signatures there were two that failed because (x_0, x_1) was too long, and fifty-one that failed because the signature encoding was too

large. Similarly, for HAWK-1024 there were no failures due to length and twenty-five due to the signature encoding being too large. The second failure condition was never observed. Hence the approximate restart probabilities for HAWK-512 and HAWK-1024 are 5.3×10^{-6} and 2.5×10^{-6} respectively, and these are dominated by encoding failures.

- The polynomial s_1 is only half of the (formal) "uncompressed signature", which is a vector $\mathbf{s} = (s_0, s_1)$. The value of s_0 is reconstructed by the signature verification algorithm. That reconstruction can fail, with a very low probability; private keys are generated with an explicit test against the bound β_0 in Algorithm Algorithm 13 precisely so that this failure probability is as low as possible. For HAWK-512 and HAWK-1024, the failure probability is indeed negligible, i.e. it is estimated as less than 2^{-105} and 2^{-315} respectively [DPPvW22b]. For HAWK-256, the failure probability may be up to about 2^{-40}, which is considered tolerable for a "challenge" scheme.

3.6 Signature Verification

The formal ("uncompressed") signature is a vector $\mathbf{s} = (s_0, s_1)$. However, the encoded signature contains only s_1 and the signature generation process did not even compute s_0. Thus, the first step of the verification process is to rebuild s_0 using (27):

$$s_0 = \left\lceil \frac{h_0}{2} + \frac{q_{01}}{q_{00}} \left(\frac{h_1}{2} - s_1 \right) \right\rfloor ,$$

where the rounding is performed coefficientwise.

We define here an implementation of the above equation that uses only integer computations (interpreted as scaled-up versions of approximations of the actual real coefficients). In many usage contexts of signatures, other implementation strategies may be used, e.g. leveraging floating-point hardware.[3] However, there are some contexts, in particular related to consensus protocols, where it is important that all implementations fully agree on whether a given signature is valid for a given message and public key pair; in that case, applications should use only implementations that can be proven to return the same output status as the one specified here.

For the specification of the FFT, InvFFT and RebuildS0 functions (Algorithms 16 to 18), we use the following conventions:

- Polynomial coefficients, in both normal and FFT representations, are stored as 32-bit signed integers, using two's complement for negative values, and truncation for oversize values. The truncation of an integer z to a signed 32-bit value is mathematically equivalent to $((z + 2^{31}) \bmod 2^{32}) - 2^{31}$. For valid keys and signatures, no such truncation happens (values do not overflow), but invalid, maliciously crafted key pairs may induce such a condition.

- The same rules apply to temporary variables with a lowercase name (such as "$x_{1,\mathsf{re}}$").

- Temporary variables with an uppercase name, such as "T_{re}", use 64-bit signed integers.

In the fixed-point notation, a real value x is represented by an integer, which is a rounded version of x multiplied by a power of two. The scaling factor varies depending on the context (so the "fixed-point" terminology is slightly abusive), but not depending on the actual values, thus avoiding all the complexity of floating-point numbers. In practical

[3] Since signature verification uses only public data, the tricky question of whether a given floating-point hardware or software implementation is constant-time does not arise.

implementations, all operations are on integers with a few extra shift operations that use shift counts independent of the data.

Let $\delta = e^{2i\pi/2048}$. We then define the following table of (precomputed) complex numbers, whose real and imaginary parts are both integers, for index $k = 0$ to 1023:

$$\Delta[k] = \left\lceil 2^{31}\Re(\delta^{\mathsf{rev}_{10}(k)}) \right\rfloor + i \left\lceil 2^{31}\Im(\delta^{\mathsf{rev}_{10}(k)}) \right\rfloor$$

The first two entries ($\Delta[0]$ and $\Delta[1]$) are not actually used. All other entries have real and imaginary integers that are strictly lower than 2^{31} in absolute value, so these can fit in a signed 32-bit representation.

For a polynomial a with real coefficients, its FFT representation $\hat{a}$ is the set of values $a(\zeta)$ for all complex roots ζ of $X^n + 1$. Since a is real, $a(\bar{\zeta})$ is the conjugate of $a(\zeta)$ when $\bar{\zeta}$ is the complex conjugate of ζ, and $\bar{\zeta}$ is another root of $X^n + 1$. Thus, it is sufficient to keep $n/2$ complex values in $\hat{a}$. The FFT function (Algorithm 16) computes $\hat{a}$ from a; in the $\hat{a}$ representation, the real and imaginary parts of $a(\zeta)$ are stored at indices v and $v + n/2$, for some integer $v < n/2$.

Algorithm 16 FFT: Conversion to FFT representation (fixed point)

Require: Polynomial a (fixed-point)
Ensure: FFT representation $\hat{a}$ (fixed-point)

$\hat{a} \leftarrow a$
$t \leftarrow n/2$
$m \leftarrow 2$
while $m < n$ **do**
  $v_0 \leftarrow 0$
  for $u = 0$ to $m/2 - 1$ **do**
    $\varepsilon_{\mathsf{re}} \leftarrow \Re(\Delta[u + m])$
    $\varepsilon_{\mathsf{im}} \leftarrow \Im(\Delta[u + m])$
    for $v = v_0$ to $v_0 + t/2 - 1$ **do**
      $x_{1,\mathsf{re}} \leftarrow \hat{a}[v]$
      $x_{1,\mathsf{im}} \leftarrow \hat{a}[v + n/2]$
      $x_{2,\mathsf{re}} \leftarrow \hat{a}[v + t/2]$
      $x_{2,\mathsf{im}} \leftarrow \hat{a}[v + t/2 + n/2]$
      $T_{\mathsf{re}} \leftarrow x_{2,\mathsf{re}}\varepsilon_{\mathsf{re}} - x_{2,\mathsf{im}}\varepsilon_{\mathsf{im}}$
      $T_{\mathsf{im}} \leftarrow x_{2,\mathsf{re}}\varepsilon_{\mathsf{im}} + x_{2,\mathsf{im}}\varepsilon_{\mathsf{re}}$
      $\hat{a}[v] \leftarrow \lfloor (2^{31}x_{1,\mathsf{re}} + T_{\mathsf{re}})/2^{32} \rfloor$
      $\hat{a}[v + n/2] \leftarrow \lfloor (2^{31}x_{1,\mathsf{im}} + T_{\mathsf{im}})/2^{32} \rfloor$
      $\hat{a}[v + t/2] \leftarrow \lfloor (2^{31}x_{1,\mathsf{re}} - T_{\mathsf{re}})/2^{32} \rfloor$
      $\hat{a}[v + t/2 + n/2] \leftarrow \lfloor (2^{31}x_{1,\mathsf{im}} - T_{\mathsf{im}})/2^{32} \rfloor$
    $v_0 \leftarrow v_0 + t$
  $t \leftarrow t/2$
  $m \leftarrow 2m$
return $\hat{a}$

In Algorithm 16, the operations on integers have been expressed in a detailed way, i.e. real and imaginary parts of the complex values are treated as separate variables. The divisions by 2^{32} are rounded toward minus infinity, which is what an "arithmetic right shift" operator computes in most programming languages.[4]

[4]This is guaranteed in Java, C#, Go, Rust... C and C++ are, formally, an exception, since right-shifting a signed negative value yields an implementation-defined result; however, in practice, all software platforms use two's complement for negative values, and the right-shift operator extends the sign bit.

The inverse FFT is given by the function InvFFT (Algorithm 17). This algorithm uses the inverses of the roots of $X^n + 1$, as stored in the Δ table. Since all these roots lie on the unit circle, their inverse is obtained by complex conjugation, which is easily obtained by negating the imaginary part.

Algorithm 17 InvFFT: Conversion back from FFT representation (fixed point)

Require: FFT representation $\hat{a}$ (fixed-point)
Ensure: Polynomial a (fixed-point)

1: $a \leftarrow \hat{a}$
2: $t \leftarrow 2$
3: $m \leftarrow n/2$
4: **while** $m > 1$ **do**
5: $\quad v_0 \leftarrow 0$
6: $\quad$ **for** $u = 0$ to $m/2 - 1$ **do**
7: $\quad\quad \eta_{\mathsf{re}} \leftarrow \Re(\Delta[u + m])$
8: $\quad\quad \eta_{\mathsf{im}} \leftarrow -\Im(\Delta[u + m])$
9: $\quad\quad$ **for** $v = v_0$ to $v_0 + t/2 - 1$ **do**
10: $\quad\quad\quad x_{1,\mathsf{re}} \leftarrow a[v]$
11: $\quad\quad\quad x_{1,\mathsf{im}} \leftarrow a[v + n/2]$
12: $\quad\quad\quad x_{2,\mathsf{re}} \leftarrow a[v + t/2]$
13: $\quad\quad\quad x_{2,\mathsf{im}} \leftarrow a[v + t/2 + n/2]$
14: $\quad\quad\quad t_{1,\mathsf{re}} \leftarrow x_{1,\mathsf{re}} + x_{2,\mathsf{re}}$
15: $\quad\quad\quad t_{1,\mathsf{im}} \leftarrow x_{1,\mathsf{im}} + x_{2,\mathsf{im}}$
16: $\quad\quad\quad t_{2,\mathsf{re}} \leftarrow x_{1,\mathsf{re}} - x_{2,\mathsf{re}}$
17: $\quad\quad\quad t_{2,\mathsf{im}} \leftarrow x_{1,\mathsf{im}} - x_{2,\mathsf{im}}$
18: $\quad\quad\quad a[v] \leftarrow \lfloor t_{1,\mathsf{re}}/2 \rfloor$
19: $\quad\quad\quad a[v + n/2] \leftarrow \lfloor t_{1,\mathsf{im}}/2 \rfloor$
20: $\quad\quad\quad a[v + t/2] \leftarrow \lfloor (t_{2,\mathsf{re}}\eta_{\mathsf{re}} - t_{2,\mathsf{im}}\eta_{\mathsf{im}})/2^{32} \rfloor$
21: $\quad\quad\quad a[v + t/2 + n/2] \leftarrow \lfloor (t_{2,\mathsf{re}}\eta_{\mathsf{im}} + t_{2,\mathsf{im}}\eta_{\mathsf{re}})/2^{32} \rfloor$
22: $\quad\quad v_0 \leftarrow v_0 + t$
23: $\quad t \leftarrow 2t$
24: $\quad m \leftarrow m/2$
25: **return** a

Using FFT and InvFFT, we can define RebuildS0 (Algorithm 18) which, given the public key polynomials q_{00} and q_{01}, and the polynomials $w_1 = h_1 - 2s_1$ and h_0, recomputes the corresponding s_0 polynomial, and returns $w_0 = h_0 - 2s_0$. Algorithm 18 uses the function sgn, which returns the "sign bit" of an integer: $\mathsf{sgn}(z) = 1$ if $z < 0$, or 0 if $z \geq 0$.

In Algorithm 18, the local variables X_{re} and X_{im} should use a 64-bit integer type, but all other intermediate values will fit in signed 32-bit variables. Moreover, all divisions in this algorithm are by powers of two and can therefore be implemented with bit shifts, except for Lines 21 and 22, which require "true" divisions. The tests on the operands ensure that:

- the dividend (X_{re} or X_{im}) is non-negative and less than 2^{62},
- the divisor (v) is positive and less than 2^{30},
- the quotient is strictly lower than 2^{32}.

The division is therefore well-defined, and can use a hardware division opcode that takes a 64-bit unsigned dividend and a 32-bit unsigned divisor (such as the one-operand variant of `div` on 32-bit x86) without triggering a CPU exception through a division by zero or an overflow.

Algorithm 18 RebuildS0: Rebuild s_0 from public key and signature

Require: Public polynomials q_{00} and q_{01}, w_1, h_0
Ensure: w_0, or $\perp$ on error

$c_{w1} \leftarrow 2^{29-(1+\mathsf{high}_{s1})}$
$c_{q00} \leftarrow 2^{29-\mathsf{high}_{00}}$
$c_{q01} \leftarrow 2^{29-\mathsf{high}_{01}}$
$c_{s0} \leftarrow (2 \cdot c_{w1} \cdot c_{q01})/(n \cdot c_{q00})$
$\hat{w}_1 \leftarrow \mathsf{FFT}(c_{w1} \cdot w_1)$
$z_{00} \leftarrow q_{00}$
if $z_{00}[0] < 0$ **then**
    return $\perp$
$z_{00}[0] \leftarrow 0$
$\hat{q}_{00} \leftarrow \mathsf{FFT}(c_{q00} \cdot z_{00})$
$\hat{q}_{01} \leftarrow \mathsf{FFT}(c_{q01} \cdot q_{01})$
$\alpha \leftarrow (2 \cdot c_{q00} \cdot q_{00}[0])/n$
for $u = 0$ to $n/2 - 1$ **do**
    $X_{\mathsf{re}} \leftarrow \hat{q}_{01}[u]\hat{w}_1[u] - \hat{q}_{01}[u + n/2]\hat{w}_1[u + n/2]$
    $X_{\mathsf{im}} \leftarrow \hat{q}_{01}[u]\hat{w}_1[u + n/2] + \hat{q}_{01}[u + n/2]\hat{w}_1[u]$
    $(X_{\mathsf{re}}, z_{\mathsf{re}}) \leftarrow (|X_{\mathsf{re}}|, \mathsf{sgn}(X_{\mathsf{re}}))$
    $(X_{\mathsf{im}}, z_{\mathsf{im}}) \leftarrow (|X_{\mathsf{im}}|, \mathsf{sgn}(X_{\mathsf{im}}))$
    $v \leftarrow \alpha + \hat{q}_{00}[u]$
    if $v \leq 0$ or $v \geq 2^{30}$ or $X_{\mathsf{re}} \geq 2^{32}v$ or $X_{\mathsf{im}} \geq 2^{32}v$ **then**
        return $\perp$
    $y_{\mathsf{re}} \leftarrow \lfloor X_{\mathsf{re}}/v \rfloor$
    $y_{\mathsf{im}} \leftarrow \lfloor X_{\mathsf{im}}/v \rfloor$
    $\hat{q}_{01}[u] \leftarrow y_{\mathsf{re}} - 2 \cdot z_{\mathsf{re}} \cdot y_{\mathsf{re}}$
    $\hat{q}_{01}[u + n/2] \leftarrow y_{\mathsf{im}} - 2 \cdot z_{\mathsf{im}} \cdot y_{\mathsf{im}}$
$t \leftarrow \mathsf{InvFFT}(\hat{q}_{01})$
for $u = 0$ to $n - 1$ **do**
    $v \leftarrow c_{s0} \cdot h_0[u] + t[u]$
    $z \leftarrow \lfloor (v + c_{s0})/(2 \cdot c_{s0}) \rfloor$ ▷ z is the rebuilt coefficient $s_0[u]$.
    if $z < -2^{\mathsf{high}_{s0}}$ or $z \geq 2^{\mathsf{high}_{s0}}$ **then**
        return $\perp$
    $w_0[u] \leftarrow h_0[u] - 2 \cdot z$
return w_0

The verification algorithm entails computing $\|\mathbf{w}\|_{\mathbf{Q}}^2$. We can write the following equation:

$$n\|\mathbf{w}\|_{\mathbf{Q}}^2 = \mathrm{Tr}(q_{00}w_0w_0^\star + q_{10}w_0w_1^\star + q_{01}w_0^\star w_1 + q_{11}w_1w_1^\star)$$

Using the known relations between the elements of $\mathbf{Q}$, this can be rewritten as:

$$d = w_1/q_{00} \tag{38}$$

$$e = w_0 + q_{01}d \tag{39}$$

$$n\|\mathbf{w}\|_{\mathbf{Q}}^2 = \mathrm{Tr}(q_{00}ee^\star + dw_1^\star) \tag{40}$$

The final value of $\|\mathbf{w}\|_{\mathbf{Q}}^2$ is a non-negative integer; we can thus perform its computation modulo sufficiently many small distinct primes, and rebuild the integer result with the CRT. It can be shown that thanks to the bounds on q_{00}, q_{01} and q_{11} (enforced during key generation with HawkKeyGen) and on s_0 and s_1 (enforced by RebuildS0 and DecodeSignature, respectively), the result cannot be larger than $15 \cdot 2^{58}$. It is therefore sufficient to perform the computation modulo the two primes $p_1 = 2147473409$ and $p_2 = 2147389441$, since $p_1p_2 > 15 \cdot 2^{58}$. In fact, since the target norm is upper bounded in valid signatures by a value which is much lower than both p_1 and p_2, we can avoid the CRT computation: for a valid signature, the computation of $n\|\mathbf{w}\|_{\mathbf{Q}}^2$ must yield the same integer result (when normalised in the 0 to $p-1$ range) for both moduli. For $p = p_1$ or $p = p_2$, the following properties hold:

- p is prime and $\mathbb{Z}/p\mathbb{Z}$ (the integers modulo p) is a finite field.
- $p-1$ is a multiple of 2048, allowing use of the number theoretic transform (NTT) to speed up multiplications and divisions of polynomials modulo X^n+1 and modulo p.
- p is slightly lower than 2^{31}: multiplication modulo p can be implemented with a Montgomery multiplication, implementable with 32-bit integers in an efficient and portable way.
- q_{00} was verified during key pair generation, to be invertible modulo X^n+1 and modulo p. Therefore, the division by q_{00} in NTT representation is well-defined and always works (no division by zero).

When p is prime and $p-1$ is a multiple of $2n$, the polynomial X^n+1 splits completely over $\mathbb{Z}/p\mathbb{Z}$, and the NTT representation of a polynomial u is the sequence of values $u(\mu) \bmod p$ for all n roots μ of X^n+1 modulo p. In NTT representation, additions, subtractions, multiplications and divisions of polynomials are simply performed by applying the operation coefficientwise. Conversion to and from NTT representation can be done in $O(n \log n)$ operations (see Section 4.1). Two other useful properties of the NTT are noteworthy:

- The NTT representation of $u^\star$ contains the same values as the NTT representation of u, in a different order. If the NTT uses the classic "bit-reversal" order, then the coefficients of the NTT representation of $u^\star$ simply use the reverse order of the coefficients of the NTT representation of u.
- For a polynomial u, we have that $\mathrm{Tr}(u) \bmod p$ is congruent to the sum of the coefficients of the NTT representation of u, modulo p.

We suppose that we have a function called NTT, such that $\mathsf{NTT}(u,p)$ is the NTT representation of u modulo p; we also use a function NTTadj such that $\mathsf{NTTadj}(\mathsf{NTT}(u,p)) = \mathsf{NTT}(u^\star,p)$. Using these functions, PolyQnorm (Algorithm 19), given q_{00}, q_{01}, w_0, w_1, and a modulus p, returns $n\|\mathbf{w}\|_{\mathbf{Q}}^2 \bmod p$. It does not matter, for PolyQnorm, which ordering convention NTT uses, as long as NTTadj uses the same convention.

Algorithm 19 PolyQnorm: polynomial Q-norm evaluation (modular)

Require: Polynomials q_{00}, q_{01}, w_0, w_1; modulus p
Ensure: $n\|\mathbf{w}\|_{\mathbf{Q}}^2 \bmod p$

$\bar{q}_{00} \leftarrow \mathsf{NTT}(q_{00}, p)$
$\bar{q}_{01} \leftarrow \mathsf{NTT}(q_{01}, p)$
$\bar{w}_0 \leftarrow \mathsf{NTT}(w_0, p)$
$\bar{w}_1 \leftarrow \mathsf{NTT}(w_1, p)$
$\bar{d} \leftarrow \bar{w}_1 / \bar{q}_{00}$
$\bar{e} \leftarrow \bar{w}_0 + \bar{d} \cdot \bar{q}_{01}$
$\bar{c} \leftarrow \bar{q}_{00} \cdot \bar{e} \cdot \mathsf{NTTadj}(\bar{e}) + \bar{d} \cdot \mathsf{NTTadj}(\bar{w}_1)$
$r \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $r \leftarrow r + \bar{c}[i]$
return $r \bmod p$

In Algorithm 19, all operations performed on NTT representations are computed coefficientwise and modulo p. The final result (r) is also computed modulo p normalised into the 0 to $p - 1$ range.

We can now express the HawkVerify function (Algorithm 20), which performs the signature verification and returns true for a valid (message, public key, signature) triplet, or false otherwise.

Algorithm 20 HawkVerify: HAWK signature verification

Require: Message m, public key pub, signature sig
Ensure: true if valid, false otherwise

$r \leftarrow \mathsf{DecodeSignature}(\mathsf{sig})$
if $r = \perp$ **then**
    return false
$(\mathsf{salt}, s_1) \leftarrow r$
$r \leftarrow \mathsf{DecodePublic}(\mathsf{pub})$
if $r = \perp$ **then**
    return false
$(q_{00}, q_{01}) \leftarrow r$
$\mathsf{hpub} \leftarrow \mathsf{SHAKE256}(\mathsf{pub})$
$M \leftarrow \mathsf{SHAKE256}(\mathsf{m} \parallel \mathsf{hpub})$
$(h_0, h_1) \leftarrow \mathsf{SHAKE256}(M \parallel \mathsf{salt})[0 : 2n]$
$w_1 \leftarrow h_1 - 2 \cdot s_1$
if $\mathsf{sym\text{-}break}(w_1) = \mathsf{false}$ **then**
    return false
$w_0 \leftarrow \mathsf{RebuildS0}(q_{00}, q_{01}, w_1, h_0)$
if $w_0 = \perp$ **then**
    return false
$r_1 \leftarrow \mathsf{PolyQnorm}(q_{00}, q_{01}, w_0, w_1, p_1)$
$r_2 \leftarrow \mathsf{PolyQnorm}(q_{00}, q_{01}, w_0, w_1, p_2)$
if $r_1 \neq r_2$ or $r_1 \not\equiv 0 \bmod n$ **then**
    return false
$r_1 \leftarrow r_1/n$
if $r_1 > 8n \cdot \sigma^2_{\mathsf{verify}}$ **then**
    return false
return true

4 Implementations

In this section, we give recommendations for implementors to end up with a fast implementation that uses little RAM. In comparison to Section 3, these recommendations do not need to be followed, in constract to e.g. the fixed-point FFT specified in Section 3.6.

In Section 4.1, we highlight some techniques that were used in the provided implementations. These allow for fast implementations. Timings, and the method to obtain them, are provided in Section 4.2. There, we also present two trade-offs between time and memory that are possible in signing and verification.

4.1 Software Implementation Techniques

4.1.1 NTT

The NTT transform is a useful tool for performing fast computations over polynomials defined modulo $X^n + 1$. The HAWK signature verification was specified in Section 3.6 using that transform, but without prescribing a specific method of computing the NTT or its inverse, nor a specific order of NTT coefficients. Algorithms 21 and 22 show the classic "bit-reversal" methods for computing the NTT and its inverse, respectively. These algorithms expect the following:

- All computations are done in the finite field $\mathbb{Z}/p\mathbb{Z}$ of integers modulo p for a prime $p \geq 3$ such that $p \equiv 1 \bmod 2n$.
- Let γ be a primitive $2n$-th root of unity modulo p. The tables Γ and Γ^{-1} contain some precomputed powers of γ in a specific order:

$$\Gamma[i] \equiv \gamma^{\mathsf{rev}_{\mathsf{logn}}(i)} \bmod p,$$
$$\Gamma^{-1}[i] \equiv \gamma^{-\mathsf{rev}_{\mathsf{logn}}(i)} \bmod p,$$

  where the bit-reversal function (cf. Section 3.1) is over $\mathsf{logn} = \log_2(n)$ bits.

Algorithm 21 NTT: Conversion to NTT representation

Require: Polynomial u (modulo $X^n + 1$ and modulo p)
Ensure: NTT representation of u (in place)

$t \leftarrow n$
$m \leftarrow 1$
while $m < n$ **do**
  $t \leftarrow t/2$
  for $i = 0$ to $m - 1$ **do**
    $s \leftarrow \Gamma[i + m]$
    for $j = 0$ to $t - 1$ **do**
      $u_0 \leftarrow u[2ti + j]$
      $u_1 \leftarrow u[2ti + t + j]$
      $u[2ti + j] \leftarrow u_0 + su_1$
      $u[2ti + t + j] \leftarrow u_0 - su_1$
  $m \leftarrow 2m$
return u

There are several possible variants for the implementation of NTT and iNTT; for instance, the halvings (division by 2) in iNTT may be delayed and performed with a single pass after completion of the outer loop.

Algorithm 22 iNTT: Inverse NTT transform

Require: NTT representation of polynomial u
Ensure: Plain representation of u
1: $t \leftarrow 1$
2: $m \leftarrow m$
3: **while** $m < n$ **do**
4: $\quad m \leftarrow m/2$
5: $\quad$ **for** $i = 0$ to $m - 1$ **do**
6: $\quad\quad s \leftarrow \Gamma^{-1}[i + m]$
7: $\quad\quad$ **for** $j = 0$ to $t - 1$ **do**
8: $\quad\quad\quad u_0 \leftarrow u[2ti + j]$
9: $\quad\quad\quad u_1 \leftarrow u[2ti + t + j]$
10: $\quad\quad\quad u[2ti + j] \leftarrow (u_0 + u_1)/2$
11: $\quad\quad\quad u[2ti + t + j] \leftarrow s(u_0 - u_1)/2$
12: $\quad t \leftarrow 2t$
13: **return** u

A noteworthy point is that the Γ and Γ^{-1} tables for degree n are strict prefixes of the corresponding tables for degree $2n$; in other words, a precomputed table for degree $n = 1024$ can also be used for computing NTTs with the same prime modulus for lower degrees (512, 256, ...). Moreover, Γ^{-1} can be recomputed from Γ using the fact that $\gamma^n = -1$, thus $\gamma^{-i} = -\gamma^{n-i}$.

4.1.2 Key Pair Generation

The HAWK key pair generation (Algorithm 13) consists of several steps, some of which are amenable to various implementation techniques. In the HAWK reference and optimised implementations, we use the techniques and implementation from [Por23]. The following points are noteworthy:

- The polynomial $q_{00} = ff^\star + gg^\star$ is computed modulo the prime integer $p_1 = 2147473409$. Since the coefficients of f and g are at most 8 in absolute value, it is guaranteed that the coefficients of q_{00} cannot exceed $128n$, which is lower than $p_1/2$. Similar considerations apply for the computation of q_{01} and q_{11}.
- Invertibility of q_{00} modulo p_1 is tested by verifying that none of the coefficients of $\mathsf{NTT}(q_{00})$ are zero. The same test is applied modulo p_2.
- The verification that $(1/q_{00})[0] \geq \beta_0$ is performed by computing the $1/q_{00}$ polynomial (over the rationals, not modulo a prime p) with a fixed-point FFT, which is also used within NTRUSolve. Note that this check is meant to ensure that the probability of algorithm RebuildS0 failing to recompute the s_0 part of a signature during verification is negligible. As such, there are no great precision requirements on this computation, and an approximate fixed-point FFT is good enough.
- NTRUSolve uses the approach of [PP19]. The field norm is applied repeatedly to halve the degrees of the involved polynomial until solving the NTRU equation becomes a simple extended GCD computation over big integers. The full degree solution is then obtained by unwinding the halvings, and applying Babai's round-off algorithm at each step. Computations with the NTT modulo small 31-bit primes p_i, are used extensively. The Babai rounding uses a FFT with fixed-point 64-bit numbers.

Algorithm HawkKeyGen enforces the check of invertibility modulo the specific primes p_1 and p_2 so that signature verification may use the same moduli; proper interoperability

of HAWK implementations requires that key pair generators and signature verifiers use the same primes. However, nothing forces the key pair generation to use 31-bit moduli for its internal computations. For instance, hardware platforms with large 64-bit multipliers might get performance improvements with 63-bit moduli.

4.1.3 Signature Generation

After obtaining the initial hash h_0, h_1 from the message, public key and a salt, the polynomials (t_0, t_1) are computed by multiplying with $\mathbf{B}$ on the left. These computations can be performed modulo 2, since we are ultimately interested only in the least significant bit of the resulting polynomial coefficients. This is why only $F \pmod 2$ and $G \pmod 2$ are stored in the private key, not the full F and G. It is possible to perform the computations of t_0 and t_1 modulo a large enough prime that supports the NTT. However, it appears to be more efficient, especially relatively to RAM usage, to avoid the NTT and directly use binary polynomials, i.e. polynomials with coefficients in $\mathbb{Z}/2\mathbb{Z}$. A product of binary polynomials of degree less than n can be performed with three products of binary polynomials of degree less than $n/2$, and a few polynomial additions (which are just bitwise XORs, since we work here in $(\mathbb{Z}/2\mathbb{Z})[X]$), using the classic Karatsuba–Ofman method [KO62]. For instance, if u and v are binary polynomials of degree less than n, then we can break each in its low and high halves:

$$\begin{aligned} u_0 &= u[0{:}n/2], \\ u_1 &= u[n/2{:}n], \\ v_0 &= v[0{:}n/2], \\ v_1 &= v[n/2{:}n], \end{aligned}$$

and we can then compute the product uv as:

$$\begin{aligned} uv &= (u_0 + X^{n/2}u_1)(v_0 + X^{n/2}v_1) \\ &\equiv (u_0v_0) + X^n(u_1v_1) + X^{n/2}((u_0 + u_1)(v_0 + v_1) + u_0v_0 + u_1v_1) \pmod 2. \end{aligned}$$

Applying this method recursively, the polynomial product boils down to products of small polynomials. Some software platforms provide a "carryless multiplication" opcode (e.g. `pclmul` on x86) which can perform these products; otherwise, integer multiplications can be used (separating the data bits with enough zeros to prevent carries from propagating). Once the product of two binary polynomials has been computed (with a result over $2n - 1$ bits), reduction modulo $X^n + 1$ is a simpler matter of bitwise XORing the high half into the low half.

The SamplerSign function uses secret data and thus should be implemented in a way that does not leak information about that data through side channels. Algorithm 14 shows a design that can at least be constant-time, i.e. avoiding leaks leveraging timing measurements:

- The input random 78-bit value c is compared to the elements of the T_0 and T_1 tables; for each table, the count of table elements greater than c is computed. Although only one of the two counts is ultimately retained, both are systematically computed to avoid leaking the corresponding bit of $\mathbf{t}$.
- Similarly, all table elements are always used, even though the values in each table are monotonically decreasing. An early exit strategy **must not** be used if the signature generation time is potentially observable by outsiders.
- Each comparison between c and a value $T_0[z]$ is done by taking the sign bit of $c - T_0[z]$. Values consist of 78 bits, which exceeds the 64-bit maximum size that

can be expected from portable C systems; the split of c into a 63-bit low part and a 15-bit high part is meant to support manual carry propagation in a portable way.

As was pointed out in Section 3.5, the SHAKE256x4 output words are used in such an order that every value is being sampled from the output of a single SHAKE256 instances among the four instances that SHAKE256x4 formally runs in parallel. This allows RAM-constrained implementations to run the four SHAKE256 instances sequentially, reusing the same space for the internal SHAKE256 state (of size about 200 bytes), while platforms with SIMD opcodes can run two or four SHAKE256 instances truly in parallel, and apply the T_0 and T_1 tables also in parallel. On x86 CPUs with AVX2 opcodes, the four SHAKE256 instances in SHAKE256x4 can indeed run in parallel.

Once the (d_0, d_1) vector has been sampled, the HawkSign algorithm needs to compute $w = fd_1 - gd_0$, potentially negate w, then produce $s_1 = (h_1 - w)/2$ (the division by 2 is exact, so the coefficients of s_1 are integers). For these computations, the NTT can be used with a small prime modulus; the elements of s_1 are normally distributed around zero with a standard deviation less than 400, thus a modulus $p \geq 400 \times 32$ covers all values up to 16 times the standard deviation, and the probability of a value of s_1 being incorrectly computed when working modulo such a p is less than 2^{-189}, hence negligible. For software implementations, moduli $p = 12289$ and $p = 18433$ are especially appropriate, since they are compatible with the use of the NTT for all specified HAWK parameter sets, and they fit on 15 bits, which is convenient for both small microcontrollers, and for powerful CPUs with SIMD opcodes. The reference and optimised implementations of HAWK use $p = 18433$; integers modulo p use Montgomery representation with $R = 2^{32}$ i.e. an integer value x is represented by $2^{32}x \bmod p$, normalised to the 1 to p range (this specific representation allows one to avoid the conditional subtraction of the modulus that usually follows the Montgomery reduction of a product). Other moduli are possible; if multiplication of small integers is expensive on a given platform, then $p = 65537$ is a good choice, since it allows a multiplicationless reduction.

4.1.4 Signature Verification

Signature verification starts with reconstruction of the s_0 polynomial (Algorithm RebuildS0), followed by a computation of $\|\mathbf{h}-2\mathbf{s}\|^2_{\mathbf{Q}}$. The first step involves computing an approximation of a polynomial over the rationals, and rounding its coefficients to the nearest integers. Such a rounding may exercise threshold conditions: if the value to round is very close to, for instance, 1/2, then two different implementations using distinct approximation methods may round such a value differently (one rounding the value to 0, the other rounding the value to 1) and come up with distinct signature verification outcomes (the signature would be deemed valid by the implementation that rounded "correctly", and invalid by the other implementation). The key pair generation enforces a maximum bound on $(1/q_{00})[0]$ precisely so that properly generated signatures cause signature verifiers to rebuild s_0 reliably and correctly, because s_0 has coefficients far enough from the threshold values. Nevertheless, it is conceptually feasible for a malicious signer to craft a key pair and a signature value that would be arbitrarily close to such thresholds. This can be an issue in some usage contexts, in particular distributed consensus protocols that require all parties to agree on the validity of a given signature for a given message and public key.

In order to avoid such issues, implementations of RebuildS0 **should** follow the steps of RebuildS0 (Algorithm 18) exactly. A fixed point approximation of values is used, with 32-bit integers and floored rounding of divisions. This corresponds to the simplest implementation on typical software platforms (using two's complement for negative integers, and arithmetic shifts for division by powers of two).

The second part of HawkVerify, i.e. the computation of $\|\mathbf{h} - 2\mathbf{s}\|^2_{\mathbf{Q}}$, is specified in Algorithms 19 and 20 with polynomials modulo two specific 31-bit primes p_1 and p_2. Other

moduli p_i are usable, provided that q_{00} is invertible modulo $X^n + 1$ and p_i; key pair generation ensures that this is true for the specified p_1 and p_2, but any implementation using another modulus must account for the possibility that one of the modular divisions in PolyQnorm fails because the divisor turns out to be zero. On 64-bit platforms, one may get an improved performance by using a single modulus larger than $15 \cdot 2^{58}$. An alternative implementation strategy for this step is to use floating-point operations: since the squared norm is an integer, an approximate computation is good enough.

4.2 Benchmarks

We measured the speed of the HAWK implementations on two platforms: a recent x86 CPU (using the optimised AVX2 implementation), and an embedded ARM Cortex M4 CPU (using the reference implementation):

- x86 system: Intel Core i5-8259U ("Coffee Lake") running at 2.3 GHz, with Linux (Ubuntu 22.04) in 64-bit mode (`x86_64`). Frequency scaling ("TurboBoost") is disabled. A single CPU core is used, on an otherwise idle machine. Compiler is Clang-14.0.0, with optimisation flags "`-O2`". AVX2 support is enabled on all functions that use AVX2 intrinsics through function attributes.

- ARM Cortex M4 system: STM32F4 "discovery" board (STM32F407VG-DISC1), running at 24 MHz; memory access times, for both RAM and ROM (Flash), have no extra wait states (local caches are not used as such frequencies). Compiler is GCC-10.3.1, with optimisation flags "`-O2 -mthumb -mcpu=cortex-m4`". The Keccak-f permutation which is at the core of SHAKE256 is implemented in assembly. Apart from this, the plain C reference code is used.

The measured performance figures are shown on Table 6.

Speed Measurements. All timings are averages expressed in clock cycles. The code is isochronous in the sense that execution time variations cannot be correlated with secret information; however, execution time still varies, in particular in key pair generation (for restarts when a candidate (f, g) pair is unsuitable), but also for all public key and signature encoding and decoding routines.

RAM Usage Estimation. The shown RAM usage is an estimate for embedded systems such as the ARM Cortex M4. Each of the key pair generation, signature generation and signature verification functions expects the caller to provide a temporary buffer in which most of the computations happen, and output values are also returned in that buffer. The stack space is used only for a small number of index values and pointers, as well as up to two concurrently running SHAKE256 contexts (208 bytes each), and an additional buffer for h_1 (in signature generation, up to 128 bytes) or (h_0, h_1) (in signature verification, up to 256 bytes). The implementation does not use recursion. Since the code is written in C, the C compiler may allocate space for its own purposes in mostly uncontrollable ways, and tends to be somewhat generous in that respect. Moreover, for improved API clarity and flexibility, some extra call layers have been added, which increases the amount of such compiler allocated stack space. To account for stack usage in a hypothetical streamlined implementation, we add 1024 bytes to the size of the temporary buffer. On x86, stack usage is greater (mostly because of the use of four parallel SHAKE256 instances in SHAKE256x4, and also because all registers are twice larger) but that matters less in practice, since systems using 64-bit CPUs have large amounts of available RAM, and the whole computation still fits entirely within the L1 cache.

Table 6: HAWK performance on x86 ("Coffee Lake") and ARM Cortex M4. Speed is expressed in clock cycles, and RAM usage in bytes.

	HAWK-256	HAWK-512	HAWK-1024
Targeted security	Challenge	NIST-1	NIST-5
Speed on x86 "Coffee Lake" with AVX2 (clock cycles)			
Key pair generation	1877122	8432840	43660958
Signature generation	54361	85372	180816
Signature generation (fast)	25979	43686	85381
Signature verification	73260	148224	302861
Signature verification (fast)	61125	124286	255312
Speed on ARM Cortex M4 (clock cycles)			
Key pair generation	18717711	52316870	225658496
Signature generation	1585967	2801495	6179673
Signature generation (fast)	581590	1161174	2494513
Signature verification	702905	1418539	3006983
Signature verification (fast)	607064	1230466	2609859
RAM usage (bytes)			
Key pair generation	7680	14336	27648
Signature generation	2560	4096	7168
Signature generation (fast)	3152	5272	9512
Signature verification	3584	6144	11264
Signature verification (fast)	4928	8768	16512

Fast Signature Generation. Speed and RAM usage for "normal" signature generation correspond to the HAWK scheme exactly as specified in this document. The "fast" signature generation is modified in two aspects:

- In the normal signature generation, first the (encoded) private key is decoded, by the function DecodePrivate, into kgseed, $F \bmod 2$, $G \bmod 2$ and hpub, and then f, g are generated from kgseed in the function Regeneratefg. The fast signature generation, however, takes $f, g, F \bmod 2, G \bmod 2$ and hpub as input, removing the CPU cost of DecodePrivate and Regeneratefg from the signing process. This does increase the RAM requirement for storing the decoded private key.
- The random source used for SamplerSign is no longer SHAKE256x4, but a fast RNG (based on AES-128 on x86, or on ChaCha8 on the Cortex M4).

As was noted in Section 3.5, using an alternative RNG for signature generation does not break interoperability; in fact, outsiders cannot detect whether SHAKE256x4 or another RNG was used for producing any given signature. The measurements provided here are meant only as an illustration of the fact that SHAKE256 related costs are an important part of the runtime cost of signature generation (up to 2/3 of the overall cost on the ARM Cortex M4); thus, substantial performance improvements can be expected if the platform provides a hardware accelerated SHAKE256 implementation, or if the RNG is switched to another secure but faster primitive, e.g. AES-256 or TurboSHAKE [BDH+23].

Fast Signature Verification. The cost of signature verification depends on the amount of available RAM. When only a minimal amount of temporary space is provided, the public key and signature values are decoded several times from their respective Golomb–Rice representations. When more space is available, such decoding is done only once, and the decoded polynomials s_1, q_{00} and q_{01} are stored in that extra space, thereby avoiding the cost of repeated decompression. It is easy for an implementation to opportunistically choose the compact or fast modes, depending on the size of the caller provided temporary buffer. Table 6 illustrates this trade-off: the fast signature verification is about 15% faster (depending on degree and architecture), but uses up to 50% more RAM.

5 Cryptanalysis

For the cryptanalytic ideas behind the modelling of HAWK we refer back to [DvW22, DPPvW22b]. In this chapter we explain, reproduce and extend the practical cryptanalysis from [DPPvW22b] and verify some additional changes, such as the switch to the centred binomial distribution for key generation. The cryptanalysis of HAWK is separated into two sections. In Section 5.1 we consider the problem of directly recovering the secret key from the public key; this is an instance of breaking the module search lattice isomorphism problem and we approach it using lattice reduction. We examine a threshold phenomenon on the lengths of vectors found during lattice reduction that determines a lower bound on the lengths of f and g required for security. In Section 5.2 we consider strong forgery attacks via mapping them to lattice reduction tasks. We end with tables reporting the estimated BKZ blocksizes required for the above two cryptanalytic tasks, also taking into account the dimensions for free techniques of [Duc18].

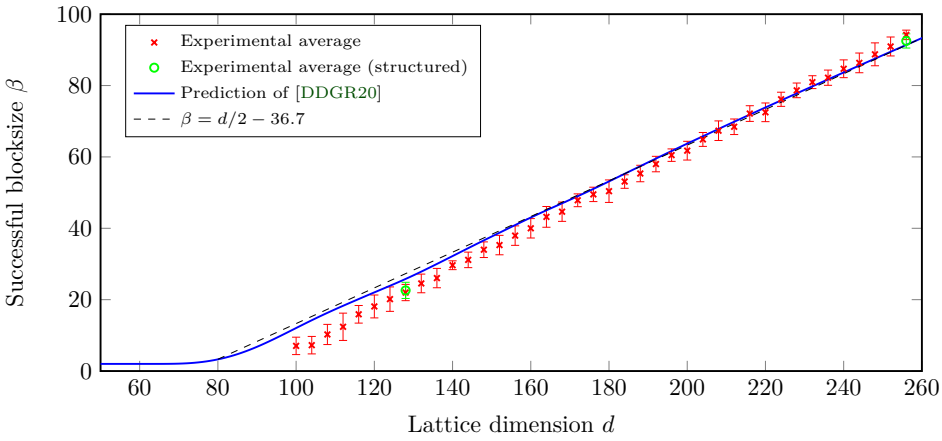
In each section we provide a model for the attack strategy, and for Section 5.1 we provide experimental evidence. For functions relating to our model, see https://github.com/hawk-sign/aux/blob/main/code/find_params.sage. The experiments are conducted over two types of instances. When n is a power of two, we are able to generate HAWK public keys as detailed in Section 3.4, minus the conditions that determine whether encoding will be successful or not, as these are not defined except for HAWK-$\{256, 512, 1024\}$. When n is not a power of two, we generate instances following the method of [DvW22, Alg. 1], i.e. the NTRU structure is lacking. Since the original specification of HAWK in [DPPvW22b] we have altered the key generation procedure to use a centred binomial distribution rather than a discrete Gaussian, as our practical cryptanalysis suggested that only the lengths of f and g were important. Therefore the distribution internal to [DvW22, Alg. 1] in our experiments will be the centred binomial distribution by default, with an explicitly indicated fallback to the discrete Gaussian distribution when we need more control over the width of the distribution. Finally, our experiments treat the rank two module basis given by $\mathbf{B}$ in HAWK as the rank $2n$ integer basis given by $\mathsf{rot}(\mathbf{B})$; equivalently, we treat the module gram matrix $\mathbf{Q}$ as $\mathsf{rot}(\mathbf{Q})$.

5.1 Secret key recovery

We first note that any unimodular matrix $\mathbf{V} \in \mathsf{GL}_{2n}(\mathbb{Z})$ such that $\mathbf{V}^t\mathbf{V} = \mathbf{Q}$ allows one to sign as if one has the secret key $\mathbf{B}$; run an unstructured version of the signature algorithm with $\mathbf{V}$ in place of $\mathbf{B}$. This is equivalent to finding a $\mathbf{U} = \mathbf{V}^{-1} \in \mathsf{GL}_{2n}(\mathbb{Z})$, such that $\mathbf{U}^t\mathbf{Q}\mathbf{U} = \mathbf{I}_{2n}$, i.e. reducing any basis corresponding to $\mathbf{Q}$ to an orthonormal basis. In particular, we consider finding a single vector of length one as a success.

Recovering unusual short vectors. We first study the BKZ blocksize required to recover a vector of length one, as a function of the dimension of the instance, in experiments where the centred binomial distribution has a 'large' parameter η, i.e. when the initial basis is badly reduced. This is an instance of the shortest vectors problem, where we have (up to sign) $d = 2n$ unusually short vectors of length 1. When performing lattice reduction on $\mathbf{Q}$, at a certain point the length of the shortest vector in the basis drops suddenly from what lattice reduction heuristics would suggest, from say $\sigma_{\mathsf{krsec}}(d) \cdot \sqrt{d}$, to length one. This threshold phenomenon is due to a projection of a unit vector being found in a terminal block during BKZ reduction. Asymptotically, the required blocksize that triggers this behaviour is given by $\beta = d/2 + o(d)$, when $\sigma_{\mathsf{krsec}} = \Theta(\sqrt{d})$ [ADPS16]. For a more precise concrete estimate of β and σ_{krsec}, we use the estimator from [DDGR20]: it relies on the BKZ simulator of Chen and Nguyen [CN11], a probabilistic model for the projected length of unusual short vectors, and accounts for the number of such short vectors. In Fig. 1 we

show experimental evidence that the blocksize estimates are closely followed in practice, both for the unstructured and for the structured cases.



We ran progressive BKZ (one tour per blocksize) over $\mathbb{Z}^d$ using an input form generated with $\eta = 20$ and report the average successful β that recovered a length one vector over 20 instances. We used the BKZ simulator and probabilistic model of [DDGR20], accounting for the d target solutions.

Figure 1: Blocksize required to recover a shortest vector via lattice reduction as a function of dimension d.

A threshold phenomenon on basis lengths. To keep our public key $\mathbf{Q}$ small we want to choose the centred binomial parameter η as small as possible. However, if η is too small then $\mathbf{Q}$ is already well reduced, which may make it easier to recover the full orthonormal basis. We consider, for some fixed dimensions, the key recovery experiment by running BKZ on public keys $\mathbf{Q}$ of different reduction quality, to observe how the hardness of this lattice reduction task behaves as a function of η. To make these experiments more granular we considered public keys where the sampling of f and g is performed using a discrete Gaussian distribution for smaller increments of standard deviation $\widetilde{\sigma_{\mathsf{kgen}}}$, as in [DPPvW22b, Fig. 4]. We clarify that here we report the *standard deviations* $\widetilde{\sigma_{\mathsf{kgen}}}$, which significantly differ from the small *widths* σ_{kgen} of the discrete Gaussian distributions. Concretely, this is because $\sigma_{\mathsf{kgen}} \geq \eta_\varepsilon(\mathbb{Z}^d)$ only for large ε in these experiments, see Definition 1. This approach also allows us to observe whether in these experiments there is a difference in practical hardness between the centred binomial distribution and the discrete Gaussian distribution of equal standard deviation. The results are shown in Fig. 2 and show a certain threshold behaviour, where at some point, fixing the dimension, the required blocksize stops increasing and stays the same for growing $\widetilde{\sigma_{\mathsf{kgen}}}$. Based on this cryptanalysis we want to pick our public key around this threshold value of $\widetilde{\sigma_{\mathsf{kgen}}}$ to minimise its size while maintaining the maximal hardness of the problem. We argue that this maximum hardness, even though the actual threshold might be lower, is attained when $\widetilde{\sigma_{\mathsf{kgen}}} > \sigma_{\mathsf{krsec}}$. During the usual progressive lattice reduction attack vectors of this length are by definition found *before* recovering a unit vector. So even if such short vectors are given away one would still require the last BKZ tour(s), or large SVP call in the last block, to actually recover a unit vector. We therefore also plot our simulation of σ_{krsec} on Fig. 2, noting that for the smaller dimensions these simulations deviate from our experimentally observed values of σ_{krsec}. From about successful blocksize 40 onwards, or $d = 170$, the estimates for σ_{krsec} grows slowly with dimension, matching the experimentally observed growth.

To verify that the maximum hardness is achieved at σ_{krsec} we compare the successful

blocksize, for several fixed dimensions, when $\widetilde{\sigma_{\mathsf{kgen}}}$ is large and when $\widetilde{\sigma_{\mathsf{kgen}}} = \sigma_{\mathsf{krsec}}(d)$ is on the estimated threshold. In particular, for these experiments, we fix the length of the secret basis vectors we sample in the key generation from below by $\sigma_{\mathsf{krsec}}\sqrt{d}$, and resample them otherwise. For the structured case this is comparable to the condition $\|(f,g)\| \geq \sigma_{\mathsf{krsec}}\sqrt{d}$ that is used in HAWK. Fig. 3 shows that the successful blocksizes are comparable in both cases, with an almost identical average, and only slightly bigger negative deviations in the threshold case.

We thus conclude that if $\|(f,g)\| \geq \sigma_{\mathsf{krsec}}\sqrt{d}$ then to recover the secret key an adversary must run BKZ with the average blocksize β suggested in Section 5.1, specifically Fig. 1.

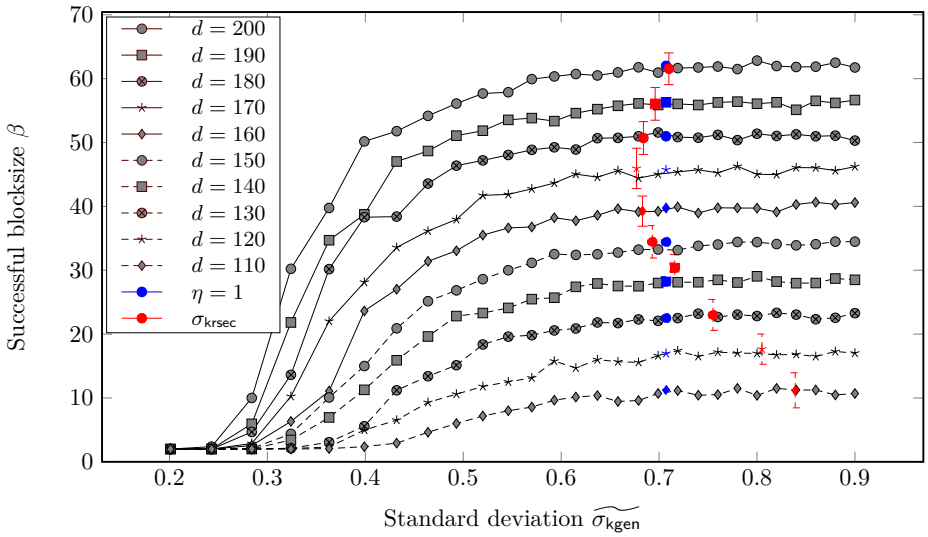
Prior occurrences of the same phenomenon. This threshold phenomenon is not specific to our cryptanalysis of this lattice isomorphism problem, but is merely a reminiscence of a well know phenomenon in the pure geometric formalism of LIP. It is implicitly present in the cryptanalysis of SIS and LWE, i.e. lattice reduction problems in q-ary lattices. For example, when attacking LWE, one usually considers "forgetting" some LWE samples to trade dimension for volume and thereby optimise the attack parameters. A geometric interpretation given in [DDGR20] is that the attacker is implicitly projecting the problem against a q-vector, and this is advantageous whenever q is smaller than the first basis vector length achieved by BKZ with a certain blocksize on a random lattice of the same volume and dimension. That is: the public description of the lattice freely gives mildly short vectors that are helpful. A similar exploit was considered in the cryptanalysis of DILITHIUM [LDK+22], though it was ineffective in their context. It was recently successfully applied [DEP23] against a SIS type problem with low modulus q, breaking a proposed variation of FALCON [ETWY22].

Our parametrisation strategy is precisely meant to avoid this pitfall. The short vectors given to the adversary are no shorter than what he would find during a sufficiently strong lattice reduction to attack the scheme.

A note on tours in progressive BKZ. Throughout our experiments we use one tour per blocksize in our progressive BKZ. It is possible to construct more optimal approaches, see [AWHT16]. However, the approach of running some number $\tau \geq 1$ tours for each blocksize, while offering improvements for small dimensions d, quickly becomes a worse strategy. For example, by simulation [DDGR20], at the HAWK-256 challenge parameters with $\tau \in \{1,2,4,8\}$, the estimated required blocksize decreases by one for each doubling of τ. This is an unfavourable trade; twice as many tours required for a BKZ run that costs more than half as much. More generally, one may vary the tours in `key_recovery_beta_ssec`.

5.2 Strong forgery

A strong forgery is when an adversary is able to produce a valid signature on a message for which it does not have any signatures. In particular, given a point $\mathbf{h} \in \mathbb{Z}^{2n}$ and $\mathbf{Q}$ one must find an $\mathbf{s} \in \mathbb{Z}^{2n}$ such that $\|\mathbf{h} - 2\mathbf{s}\|_{\mathbf{Q}}^2 \leq 8n\sigma_{\mathsf{verify}}^2$. This is equivalent to being given a point $\mathbf{t} = \mathbf{B} \cdot \mathbf{h}$ and $\mathbf{B}$ and having to find some $\mathbf{y} \in \mathbb{Z}^{2n}$ such that $\|\mathbf{t} - 2\mathbf{y}\| \leq 2\sqrt{2n}\sigma_{\mathsf{verify}}$. This is an approximate CVP instance over a rotation of $2\mathbb{Z}^{2n}$. In particular if we measure the approximation factor γ with respect to the normalised volume we have $\|\mathbf{t} - 2\mathbf{y}\| \leq \gamma \cdot \mathrm{vol}(2\mathbb{Z}^{2n})^{1/(2n)} = 2\gamma$ for $\gamma = \sqrt{2n}\sigma_{\mathsf{verify}}$. We can therefore appeal to the nearest colattice algorithm [EK20]. In particular [EK20, Thm. 4.3] implies a heuristic algorithm that allows one to solve the above approximate CVP with approximation factor $\gamma = \mathrm{gh}(\beta)^{(n-1)/(\beta-1)}$ where $\mathrm{gh}(\beta)$ is the expected first minimum of a random lattice with unit volume. This is achieved by the application of the DBKZ algorithm [MW16] with blocksize β and an exact CVP oracle for rank β lattices. In particular, one picks β minimal such that $\mathrm{gh}(\beta)^{(n-1)/(\beta-1)} \leq \sqrt{2n}\sigma_{\mathsf{verify}}$.



We ran progressive BKZ (one tour per blocksize) over $\mathbb{Z}^d$ using an input form generated with various σ_{kgen} and report the average successful β that recovered a length one vector over 40 instances. To allow us to vary σ_{kgen} gradually we sampled from a discrete Gaussian distribution, except for the centered binomial case ($\eta = 1$) at $\sigma_{\mathsf{kgen}} = \sqrt{1/2}$. Note that the range of σ_{kgen} includes values "below smoothing", for which the actual standard deviation $\widetilde{\sigma_{\mathsf{kgen}}}$ can be significantly lower than the Gaussian width parameter σ_{kgen}.

Figure 2: Blocksize required to recover a shortest vector via lattice reduction as a function of the standard deviation $\widetilde{\sigma_{\mathsf{kgen}}}$.

Table 7: Estimated BKZ blocksizes for cryptanalytic tasks using the BKZ simulator of [DDGR20].

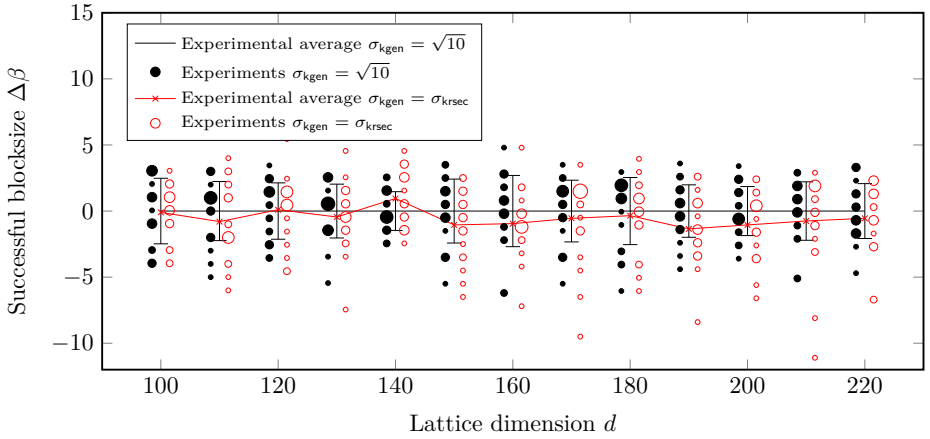
	β_{key}	β_{key} (d4f)	β_{forge}	β_{forge} (d4f)
HAWK-256	211	187	211	187
HAWK-512	452	412	452	412
HAWK-1024	940	873	1009	938

Crucial to this argument is that DBKZ with blocksize β heuristically finds vectors of length $\mathrm{gh}(\beta)^{(n-1)/(\beta-1)} \cdot \mathrm{vol}(2\mathbb{Z}^{2n})^{1/(2n)}$. When we use simulation for this approach we estimate the blocksize required in progressive BKZ with one tour per blocksize that finds vectors of this length. We report this β below and disregard the cost of the exact CVP oracle in rank β lattices.

5.3 Estimated blocksizes

In Table 7 we report the "raw" estimated blocksizes that simulation via [DDGR20] suggests are necessary for the cryptanalytic tasks of Sections 5.1 and 5.2, as well as the respective blocksizes once the dimensions for free technique has been taken into account [Duc18]. See functions `key_recovery_beta_ssec` and `approx_SVP_beta`.

We assume a single tour at blocksize β costs at least n calls to an SVP oracle of rank β for HAWK-n. From https://github.com/jschanck/eprint-2019-1161/blob/main/data/cost-estimate-list_decoding-classical.csv we have an estimated gate count for realising the SVP oracle of rank β via sieving [AGPS20], specifically the fastest known classical sieve [BDGL16]. Taking the lowest blocksizes in the rows associated to HAWK-512



We ran progressive BKZ (one tour per blocksize) over $\mathbb{Z}^d$ using an input form generated with $\sigma_{\mathsf{kgen}} = \sqrt{10}$ or with $\sigma_{\mathsf{kgen}} = \sigma_{\mathsf{krsec}}$ and report the (average) successful β that recovered a length one vector over 20 instances. In order to sample at σ_{krsec} we sampled from a discrete Gaussian distribution instead of the binomial one. We normalised the blocksize with respect to the average successful blocksize when $\sigma_{\mathsf{kgen}} = \sqrt{10}$.

Figure 3: Blocksize required to recover a shortest vector via lattice reduction as a function of dimension d for $\sigma_{\mathsf{kgen}} = \sqrt{10}$ and $\sigma_{\mathsf{kgen}} = \sigma_{\mathsf{krsec}}$.

Table 8: Estimated BKZ blocksizes for cryptanalytic tasks using the GSA-interesect method of [AGVW17] for β_{key} and the nearest colattice approach [EK20] for β_{forge}.

	β_{key}	β_{key} (d4f)	β_{forge}	β_{forge} (d4f)
HAWK-256	220	195	205	181
HAWK-512	456	416	438	399
HAWK-1024	933	866	974	905

and HAWK-1024, and lowering them to the nearest multiple of eight to appeal to the concrete values from [AGPS20], we have estimated gate counts of 2^{141} and 2^{278}. Applying a factor of n to this cost gives us 2^{150} and 2^{288}. This still ignores other factors, such as the progressivity factor $C^2 \approx 2^5$ discussed in [SAB+22, Sec. 5.2.1], and the hidden overhead of the Becker–Ducas–Gama–Laarhoven sieves of a similar order of magnitude studied in [Duc22].

The above analysis only considers a pure BKZ attack, while a typical optimisation consists of applying a final SVP call in a slightly larger blocksize on the terminating block (a distinction sometime referred to as the uSVP versus BDD attack), to amortise the cost of $2n - \beta$ many SVP calls inside BKZ. This approach however consumes even more memory (which is already above 2^{100} bits for HAWK-512), for a limited gain on time (at most nC, and significantly less in practice).

While we expect simulation to give more accurate blocksizes, for reference we also provide in Table 8 the estimated blocksizes not using simulation, similar to the analysis of FALCON [PFH+22, Sec. 2.5.1]. However, in our version of this analysis we do not simplify some exponents, see https://github.com/hawk-sign/aux/blob/main/code/falcon.sage for a comparison of FALCON-$\{512, 1024\}$ and HAWK-$\{512, 1024\}$ in various analyses. We also do not apply the overconservative "double" dimensions for free in key recovery, as explained in [DPPvW22b, App. D]. If we apply the same conservative analysis as for Table 7 above to HAWK-512 and HAWK-1024 we calculate gate costs of 2^{145} and 2^{286}.

We might also consider the methodologies of [PFH+22, Sec. 2.5.1] and [LDK+22, Tab. 4] in deciding what is an acceptable blocksize, after taking dimensions for free into account, to achieve a particular security level. For example, FALCON considers $\beta = 374$ and $\beta = 869$ to be sufficient to achieve NIST-I and NIST-V, whereas DILITHIUM considers $\beta = 394$ and $\beta = 818$ sufficient for NIST-II and NIST-V. We have arrived at our blocksizes via different analyses; here we are solely considering the conversion from blocksize to security level.

We see that the minimum blocksizes, after dimensions for free have been applied, required to perform key recovery or forge across Tables 7 and 8 are 399 and 866 for HAWK-512 and HAWK-1024 respectively. While the lattice dimensions are slightly larger in DILITHIUM, by no more than a factor of two, all of our estimated blocksizes are at least as large for equal (or smaller) security levels. In the comparison to FALCON we see that HAWK-512 has higher blocksizes for all cryptanalytic tasks, and β_{key} (d4f) for HAWK-1024 is three smaller than for FALCON-1024. Given the margin expressed for the FALCON-1024 parameters in [PFH+22, Sec. 2.5.1] this small decrease is acceptable.

6 Formal Security

In this section we provide formal justification for the security of HAWK. We start in Section 6.1 with local preliminaries. Then in Section 6.2 we formally introduce omSVP. In Sections 6.3 and 6.4 we provide reductions in the (quantum) random oracle model from HAWK to omSVP. We then introduce the search module lattice isomorphism problem, which is the problem upon which direct key recovery relies. Finally, we discuss these problems with respect to the parameters of HAWK and our design decisions.

6.1 Preliminaries

For a statement P we let $[\![P]\!]$ equal 1 if P is true, and 0 otherwise. We now define the smoothing parameter [MR04] of the integer lattice with respect to width parameter σ; recall (3).

Definition 1. For $\varepsilon > 0$ the smoothing parameter $\eta_\varepsilon(\mathbb{Z}^n)$ is the minimal σ such that $\rho_{1/2\pi\sigma}(\mathbb{Z}^n \setminus \{\mathbf{0}\}) \leq \varepsilon$.

Note η_ε is often defined with respect to a width parameter $s = \sqrt{2\pi}\sigma$ and therefore its value in Definition 1 is a factor of $\sqrt{2\pi}$ smaller than when defined with respect to s, and that $\mathbb{Z}^n$ is a self dual lattice. Also, $\eta_\varepsilon(\mathbb{Z}^n)$ is continuous and strictly increasing as ε decreases, with $\lim_{\varepsilon\to 0}(\eta_\varepsilon(\mathbb{Z}^n)) = \infty$. The smoothing parameter $\eta_\varepsilon(\mathbb{Z}^{2n})$ implicitly determines the Rényi divergence of the two distributions we consider in Lemma 1 and also determines the upper bounds on the probabilities in Lemma 2. In each case a smaller ε is better for our reduction, but requires a larger σ by the discussion above.

We fix the section $s\colon R_n/2R_n \to R_n$ of the natural surjection $R_n \to R_n/2R_n$ that has $s(a + 2R_n) = a_0 + a_1X + \cdots a_{n-1}X^{n-1}$ with $(a_0, \ldots, a_{n-1}) \in \{0, 1\}^n$. We thus consider reduction mod 2 in R_n as a map $R_n \to R_n/2R_n \to R_n$ via s. In order to abstract away the technical details of sym-break we consider the function $\langle\,\cdot\,\rangle : R_n^2 \to R_n^2$ defined by

$$\mathbf{w} \mapsto \langle \mathbf{w} \rangle = \begin{cases} -\mathbf{w} \text{ if } \mathsf{sym\text{-}break}(\mathbf{w}) = 1, \\ \mathbf{w} \text{ otherwise,} \end{cases}$$

where sym-break is as specified in Section 3.5.2. It is convenient to think of $\langle \mathbf{w} \rangle$ as a representation of the equivalence class $\{\mathbf{w}, -\mathbf{w}\}$, with the representation being unique if $\mathbf{w} \notin R_n \times \{0\}$. Indeed, what will be relevant is that

$$\langle \mathbf{w} \rangle = \langle -\mathbf{w} \rangle \in \{\mathbf{w}, -\mathbf{w}\} \quad \forall\, \mathbf{w} \notin R_n \times \{0\}\,, \tag{41}$$

while $\langle \mathbf{w} \rangle = \mathbf{w}$ for $\mathbf{w} \in R_n \times \{0\}$.

Definition 2. Given $\sigma > 0$ and $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$ for some $\mathbf{B} \in \mathsf{GL}_2(R_n)$, let

$$\rho_{\mathbf{Q},\sigma}\colon R_n^2 \to \mathbb{R},\ \mathbf{x} \mapsto \exp(-\|\mathbf{x}\|_{\mathbf{Q}}^2/2\sigma^2).$$

Note here that for $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$, we have $\rho_{\mathbf{Q},\sigma}(\mathbf{x}) = \rho_\sigma(\mathbf{B}\cdot\mathbf{x})$ by (22).

Definition 3. Let $\mathbf{B} \in \mathsf{GL}_2(R_n)$ and $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$. For every $\sigma > 0$ and $\mathbf{h} \in (R_n/2R_n)^2 \subset R_n^2$ we consider the following $\mathbf{Q}$ dependent distributions.

- $D_{\mathbf{Q},\sigma}$, the discrete Gaussian distribution over R_n^2 under $\|\cdot\|_{\mathbf{Q}}$ given by

$$D_{\mathbf{Q},\sigma}\colon R_n^2 \to \mathbb{R},\ \mathbf{x} \mapsto \rho_{\mathbf{Q},\sigma}(\mathbf{x}) / \sum_{\mathbf{y}\in R_n^2} \rho_{\mathbf{Q},\sigma}(\mathbf{y}).$$

- $\tilde{D}_{\mathbf{Q},\sigma}[\mathbf{h}]$, the discrete Gaussian distribution over $\mathbf{h}+2R_n^2 \subset R_n^2$ under $\|\cdot\|_{\mathbf{Q}}$ given by
$$\tilde{D}_{\mathbf{Q},\sigma}[\mathbf{h}] : \mathbf{h}+2R_n^2 \to \mathbb{R},\ \mathbf{x} \mapsto \rho_{\mathbf{Q},\sigma}(\mathbf{x}) / \sum_{\mathbf{y}\in\mathbf{h}+2R_n^2} \rho_{\mathbf{Q},\sigma}(\mathbf{y}).$$
- $\tilde{D}_{\mathbf{Q},\sigma}$, the distribution implied by sampling a uniform $\mathbf{h} \leftarrow (R_n/2R_n)^2$ and returning a sample from $\tilde{D}_{\mathbf{Q},\sigma}[\mathbf{h}]$.

Given $\mathbf{B}$ we may sample the above distributions by sampling according to $D_{\mathbf{I}_2(K_n),\sigma}$, in the coset defined by $\mathbf{t} = \mathbf{B}\mathbf{h}$ if required, and multiplying by $\mathbf{B}^{-1}$ as in HAWK signing, e.g. Algorithm 2. Note that for every $\mathbf{h} \in (R_n/2R_n)^2$
$$\forall\ \mathbf{w} \in \mathrm{Supp}\Big(\widetilde{D}_{\mathbf{Q},\sigma}[\mathbf{h}]\Big) = \mathbf{h}+2R_n^2,\ \mathbf{w} \bmod 2 = \mathbf{h}. \tag{42}$$

We call the maximum probability of any element in the support of a distribution the guessing probability of the distribution. As a direct consequence of [DPPvW22b, Lemma 3], the guessing probability of $\mathbf{w} \bmod 2$ for $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma}$ with $\sigma \geq \eta_\varepsilon(\mathbb{Z}^{2n})$ has the following upper bound
$$\mathrm{guess}(\mathbf{w} \bmod 2) = \max_{\mathbf{h}^\circ \in (R_n/2R_n)^2} \Pr[\mathbf{w} \bmod 2 = \mathbf{h}^\circ] \leq 2^{-2n} \cdot \frac{1+\varepsilon}{1-\varepsilon}. \tag{43}$$

We will also make use of the Rényi divergence between $\tilde{D}_{\mathbf{Q},2\sigma}$ and $D_{\mathbf{Q},2\sigma}$, which we compute in the following lemma.

Lemma 1. *Let $\sigma > 0$, $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$ for some $\mathbf{B} \in \mathsf{GL}_2(R_n)$ and $a \in (1,\infty)$ be an order. Furthermore, denote by*
$$\alpha = \frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_\sigma(\mathbb{Z})},\ \textit{and}\ \beta = \frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_\sigma\left(\mathbb{Z}+\frac{1}{2}\right)},$$
then
$$R_a\big(\tilde{D}_{\mathbf{Q},2\sigma} \,\|\, D_{\mathbf{Q},2\sigma}\big) = \left(\frac{\alpha^{a-1}+\beta^{a-1}}{2}\right)^{\frac{2n}{a-1}}.$$

Proof. Since each $\mathbf{w} \in R_n^2$ lies in $2R_n^2+\mathbf{h}$ for exactly one $\mathbf{h} \in (R_n/2R_n)^2$ we have
$$\tilde{D}_{\mathbf{Q},2\sigma}(\mathbf{w}) = 2^{-2n}\rho_{\mathbf{Q},2\sigma}(\mathbf{w})/\rho_{\mathbf{Q},2\sigma}(2R_n^2+\mathbf{h}).$$
Similarly we have
$$D_{\mathbf{Q},2\sigma}(\mathbf{w}) = \rho_{\mathbf{Q},2\sigma}(\mathbf{w})/\rho_{\mathbf{Q},2\sigma}(R_n^2).$$
Hence, by definition of the Rényi divergence,
$$\begin{aligned} R_a(\tilde{D}_{\mathbf{Q},2\sigma} \,\|\, D_{\mathbf{Q},2\sigma})^{a-1} &= \sum_{\mathbf{h}\in(R_n/2R_n)^2} \sum_{\mathbf{w}\in 2R_n^2+\mathbf{h}} \tilde{D}_{\mathbf{Q},2\sigma}(\mathbf{w})^a / D_{\mathbf{Q},2\sigma}(\mathbf{w})^{a-1} \\ &= \sum_{\mathbf{h}\in(R_n/2R_n)^2} 2^{-2n} \cdot \left(\frac{\rho_{\mathbf{Q},2\sigma}(R_n^2)}{2^{2n}\rho_{\mathbf{Q},2\sigma}(2R_n^2+\mathbf{h})}\right)^{a-1} \sum_{\mathbf{w}\in 2R_n^2+\mathbf{h}} \frac{\rho_{\mathbf{Q},2\sigma}(\mathbf{w})}{\rho_{\mathbf{Q},2\sigma}(2R_n^2+\mathbf{h})}. \end{aligned}$$
Note that $\sum_{\mathbf{w}} \rho_{\mathbf{Q},2\sigma}(\mathbf{w})/\rho_{\mathbf{Q},2\sigma}(2R_n^2+\mathbf{h}) = 1$, simplifying the equation to
$$\begin{aligned} R_a\big(\tilde{D}_{\mathbf{Q},2\sigma} \,\|\, D_{\mathbf{Q},2\sigma}\big)^{a-1} &= 2^{-2n} \sum_{\mathbf{h}\in(R_n/2R_n)^2} \left(\frac{\rho_{\mathbf{Q},2\sigma}(R_n^2)}{2^{2n}\rho_{\mathbf{Q},2\sigma}(2R_n^2+\mathbf{h})}\right)^{a-1} \\ &= 2^{-2n} \sum_{\mathbf{h}\in(R_n/2R_n)^2} \left(\frac{\rho_{2\sigma}(R_n^2)}{2^{2n}\rho_{2\sigma}(2R_n^2+\mathbf{B}\cdot\mathbf{h})}\right)^{a-1}, \end{aligned}$$

where the second equality relies on the factor that $\mathbf{B} \in \mathsf{GL}_2(R_n)$ and therefore that $\mathbf{B}R_n^2 = R_n^2$. We also note that $\left\{2R_n^2 + \mathbf{Bh} \bmod 2 \colon \mathbf{h} \in (R_n/2R_n)^2\right\} = \left\{2R_n^2 + \mathbf{t} \colon \mathbf{t} \in (R_n/2R_n)^2\right\}$. Therefore, we may sum over $\mathbf{t} \in (R_n/2R_n)^2$. We also here pass to the unstructured setting, noting that $\rho_{2\sigma}(R_n^2) = \rho_{2\sigma}(2\mathbb{Z}^{2n})$ and that, since $\mathbb{Z}^{2n}$ has an orthogonal basis $\mathbf{I}_{2n}(\mathbb{Z})$, $\rho_{2\sigma}(2\mathbb{Z}^{2n}) = \rho_{2\sigma}(2\mathbb{Z})^{2n}$. Similarly, we have $\rho_{2\sigma}(2R_n^2+\mathbf{t}) = \rho_{2\sigma}(2\mathbb{Z}^{2n}+\mathbf{t}) = \prod_i \rho_{2\sigma}(2\mathbb{Z}+t_i)$, with $\mathbf{t} \in \{0,1\}^{2n}$ after the first equality. Thus,

$$\begin{aligned} R_a\big(\tilde{D}_{\mathbf{Q},2\sigma} \,\|\, D_{\mathbf{Q},2\sigma}\big)^{a-1} &= 2^{-2n} \sum_{\mathbf{t}\in\{0,1\}^{2n}} \left(\frac{\rho_{2\sigma}(\mathbb{Z}^{2n})}{2^{2n}\rho_{2\sigma}(2\mathbb{Z}^{2n}+\mathbf{t})}\right)^{a-1} \\ &= 2^{-2n} \sum_{\mathbf{t}\in\{0,1\}^{2n}} \prod_{i=0}^{2n-1} \left(\frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_{2\sigma}(2\mathbb{Z}+t_i)}\right)^{a-1}. \end{aligned}$$

By swapping these (finite) sums and products, and considering the contribution of a single t_i, half of which are zero and half of which are one over $\mathbf{t} \in \{0,1\}^{2n}$ for any fixed index i, we have

$$\begin{aligned} R_a\big(\tilde{D}_{\mathbf{Q},2\sigma} \,\|\, D_{\mathbf{Q},2\sigma}\big)^{a-1} &= \prod_{i=0}^{2n-1} \frac{1}{2} \cdot \left[\left(\frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_{2\sigma}(2\mathbb{Z})}\right)^{a-1} + \left(\frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_{2\sigma}(2\mathbb{Z}+1)}\right)^{a-1}\right] \\ &= \left[\frac{1}{2} \cdot \left(\left(\frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_{2\sigma}(2\mathbb{Z})}\right)^{a-1} + \left(\frac{\rho_{2\sigma}(\mathbb{Z})}{2\rho_{2\sigma}(2\mathbb{Z}+1)}\right)^{a-1}\right)\right]^{2n} \end{aligned}$$

As $\rho_{2\sigma}(2\mathbb{Z}+b) = \rho_\sigma(\mathbb{Z}+b/2)$, one can substitute α and β in the above. □

As σ grows the Rényi divergence decreases towards one. When σ is not too big, α and β in the above lemma can be efficiently numerically computed to any desired precision via a tailcut.

6.1.1 Roots of Unity

The $2n$ roots of unity of R_n are denoted by μ_K and given by $\{1, X, X^2, \ldots, X^{2n-1}\}$. The following lemma is extracted from the proof of [DPPvW22a, Lem. 10] and a replica (up to a variable name change) of [FH23, Lem. 1]; see the latter for a detailed proof.

Lemma 2. *Let n be a power of 2, $\varepsilon > 0$, $\sigma \geq \eta_\varepsilon(\mathbb{Z}^{2n})$, and $\mathbf{h}^\circ \in (R_n/2R_n)^2$. Consider $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma}$ and set $\mathbf{h} = \mathbf{w} \bmod 2$. Then*

$$\Pr\left[\exists\, \alpha \in \mu_K \setminus \{\pm 1\} :\ \tfrac{1}{2}(\mathbf{h} + \alpha\mathbf{w}) \in R_n^2\right] \leq 2^{-n} \cdot \frac{1+\varepsilon}{1-\varepsilon}\ , \text{ and} \tag{44}$$

$$\Pr\left[\exists\, \alpha \in \mu_K :\ \tfrac{1}{2}(\mathbf{h}^\circ + \alpha\mathbf{w}) \in R_n^2\right] \leq n \cdot 2^{-2n} \cdot \frac{1+\varepsilon}{1-\varepsilon}. \tag{45}$$

6.1.2 Adaptive Reprogramming Lemma

The following reprogramming lemma is adapted from [GHHM21, Thm. 1], with the overall loss slightly improved. Intuitively, it states that, if the location x of a reprogramming of the random oracle is hard to guess prior to when it is taking place, then such a reprogramming is hard to notice.

Lemma 3 (Slight modification of [GHHM21, Thm. 1])**.** *Let $\mathsf{H} : \mathcal{X} \to \mathcal{Y}$ be a random oracle, $\varepsilon > 0$ and Ω be a family of distributions on $\mathcal{X}$ where every $\mathcal{D} \in \Omega$ has guessing probability guess$(\mathcal{D}) = \max_{x^\circ} \Pr_{x\leftarrow\mathcal{D}}[x = x^\circ] \leq \gamma$. Define the reprogramming oracle Repro_b for $b \in \{0,1\}$ that, on input (a suitable representation of) $\mathcal{D} \in \Omega$, functions as*

$\mathsf{Repro}_0(\mathcal{D})$		$\mathsf{Repro}_1(\mathcal{D})$	
1 :	$x \leftarrow \mathcal{D}$	1 :	$x \leftarrow \mathcal{D}$
2 :	$y = \mathsf{H}(x)$	2 :	$\mathsf{H}(x) = y \leftarrow \mathcal{Y}$
3 :	**return** (x, y)	3 :	**return** (x, y)

Suppose $\mathcal{A}^{\mathsf{Repro}_b,\mathsf{H}}$ *for* $b \in \{0,1\}$ *makes at most* q_r *queries to the reprogramming oracle* Repro_b*, and at most* q_h *quantum queries to* H *before the last reprogramming query. Then,*

$$\left| \Pr\left[1 \leftarrow \mathcal{A}^{\mathsf{Repro}_0,\mathsf{H}}\right] - \Pr\left[1 \leftarrow \mathcal{A}^{\mathsf{Repro}_1,\mathsf{H}}\right] \right| \leq 2q_r\sqrt{(q_h + q_r)\cdot\gamma}.$$

We refer readers to [FH23, App. A] for a detailed proof, which proves the same statement up to a change of variable names.

6.1.3 SUF-CMA security

For a signature scheme $\Pi = (\mathsf{Sign}, \mathsf{Vf}, \mathsf{KGen})$ we define the game (Q)ROM-SUF-CMA$_{\Pi,\mathcal{A}}$ in Fig. 4. For a signature scheme to exhibit strong unforgeability under chosen message attacks no (quantum) probabilistic polynomial time adversary against (Q)ROM-SUF-CMA$_{\Pi,\mathcal{A}}$ should win with non negligible probability. The prefix (Q)ROM indicates that the adversary gets (quantum) access to the random oracle H.

(Q)ROM-SUF-CMA$_{\Pi,\mathcal{A}}(1^\lambda)$

1 : $\mathcal{L}_{\mathsf{Sign}} \leftarrow \emptyset$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$
3 : $(\mathsf{m}^\star, \mathsf{sig}^\star) \leftarrow \mathcal{A}^{\mathsf{OSign},\mathsf{H}}(1^\lambda, \mathsf{pk})$
4 : **return** $[\![\mathsf{Vf}^{\mathsf{H}}_{\mathsf{pk}}(\mathsf{m}^\star, \mathsf{sig}^\star) = 1 \wedge (\mathsf{m}^\star, \mathsf{sig}^\star) \notin \mathcal{L}_{\mathsf{Sign}}]\!]$

$\mathsf{OSign}(\mathsf{m})$

1 : $\mathsf{sig} \leftarrow \mathsf{Sign}^{\mathsf{H}}_{\mathsf{sk}}(\mathsf{m})$
2 : $\mathcal{L}_{\mathsf{Sign}} \leftarrow \mathcal{L}_{\mathsf{Sign}} \cup \{(\mathsf{m}, \mathsf{sig})\}$
3 : **return** sig

Figure 4: The (Q)ROM-SUF-CMA game.

Definition 4 ((Q)ROM-SUF-CMA security)**.** Let $\Pi = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ be a signature scheme. Let t, ε, q_s, q_h be functions of λ. We say that Π is $(t, \varepsilon, q_s, q_h)$-(Q)ROM-SUF-CMA secure, or strongly unforgeable under chosen message attacks in the (quantum) random oracle model, if for any adversary $\mathcal{A}$ running in time at most t, making at most q_s queries to its signing oracle, and making at most q_h queries to its random oracle, the probability $\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi,\mathcal{A}}(\lambda) := \Pr[\text{(Q)ROM-SUF-CMA}_{\Pi,\mathcal{A}}(1^\lambda) = 1] \leq \varepsilon(\lambda)$ for all λ.

In the above definition we have (mostly) dropped the dependence of $\mathcal{A}$ and t, ε, q_s, q_h on λ. The superscript in $\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi,\mathcal{A}}(\lambda)$ does not read (Q)ROM-SUF-CMA, and instead only SUF-CMA, for typographical aesthetics. Also, outside of definitions, we drop the dependence on the security parameter; $\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi,\mathcal{A}}$.

6.2 The one more shortest vector problem

Informally an average case omSVP instance samples a $\mathbf{Q}$ from a distribution over some $\mathcal{H}^{>0}_r(K)$ and gives Gaussian samples according to this $\mathbf{Q}$. Here, for a CM-field K and

$\mathrm{SAMPLE}_{\mathsf{ac\text{-}omSVP},\mathcal{A}}(1^\lambda)$

1 : $\mathcal{L}_{\mathsf{samples}} \leftarrow \{\mathbf{0}\}$
2 : $(\mathbf{Q}, \mu_K, L, \sigma) \leftarrow \mathsf{Init}(1^\lambda)$
3 : $\mathbf{w}^\star \leftarrow \mathcal{A}^{\mathsf{samp}}(1^\lambda, \mathbf{Q}, \mu_K, L, \sigma)$
4 : **return** $[\![\mathbf{w}^\star \in R^r \wedge \|\mathbf{w}^\star\|_{\mathbf{Q}} \leq L \wedge \mathbf{w}^\star \notin \mathcal{L}_{\mathsf{samples}}]\!]$

samp

1 : $\mathbf{w} \leftarrow D_{\mathbf{Q},\sigma}$
2 : $\mathcal{L}_{\mathsf{samples}} \leftarrow \mathcal{L}_{\mathsf{samples}} \cup \{\alpha\mathbf{w}\}_{\alpha \in \mu_K}$
3 : **return** $\mathbf{w}$

Figure 5: The SAMPLE game.

$r \in \mathbb{Z}_{\geq 1}$, we define $\mathcal{H}_r^{>0}(K) \subset K^{r\times r}$ as the $\mathbf{Q} \in K^{r\times r}$ such that $\mathbf{Q} = \mathbf{Q}^\star$ and $\mathrm{Tr}(\mathbf{v}^\star\mathbf{Q}\mathbf{v}) > 0$ for all $\mathbf{v} \in K^r \setminus \{\mathbf{0}\}$, i.e. matrices that represent Hermitian positive definite forms. If one can find some non zero $\mathbf{x}$ in the ring of integers R of K that is sufficiently short with respect to $\mathbf{Q}$, and that is in some sense non trivially new, then one solves the problem. We show that for certain parameters, if one can forge a signature against HAWK then in the programmable quantum random oracle model one can solve an instance of this problem.

Definition 5 (Average case omSVP)**.** An average case omSVP instance is the pair $\mathsf{ac\text{-}omSVP} = (\mathsf{Init}, \mathsf{samp})$. Init returns a form $\mathbf{Q}$ sampled from some distribution over some $\mathcal{H}_r^{>0}(K)$, the roots of unity μ_K for K, a length bound L, and a Gaussian parameter σ, where r, K, L, σ are functions of λ. Each call to samp returns a sample from $D_{\mathbf{Q},\sigma}$.

The adversary in Fig. 5 wins whenever it can use the output of Init and the samples it receives from samp to return some non trivial new element of R^r that is short enough.

Definition 6 (SAMPLE security)**.** Let $\mathsf{ac\text{-}omSVP} = (\mathsf{Init}, \mathsf{samp})$ be an average case omSVP instance. Let t, ε, q_o be functions of λ. We say that ac-omSVP is (t, ε, q_o)-SAMPLE secure, if for any adversary $\mathcal{A}$ running in time at most t, and making at most q_o queries to samp, the probability $\mathsf{Adv}^{\mathrm{SAMPLE}}_{\mathsf{ac\text{-}omSVP},\mathcal{A}}(\lambda) := \Pr[\mathrm{SAMPLE}_{\mathsf{ac\text{-}omSVP},\mathcal{A}}(1^\lambda) = 1] \leq \varepsilon(\lambda)$ for all λ.

6.3 Quantum security of HAWK

Let $\eta, \sigma_{\mathsf{sign}}, \sigma_{\mathsf{verify}}$, and $\mathsf{saltlen}_{\mathsf{bits}}$ be the parameters as specified in Section 3.2. Let ε be minimal such that $\sigma_{\mathsf{sign}} \geq \eta_\varepsilon(\mathbb{Z}^{2n})$. We write HawkKeyGen for the key generation procedure of Algorithm 1. It produces a key pair $(\mathbf{Q}, (\mathbf{B}, \mathsf{hpub}))$ with $\mathbf{B} \in \mathsf{GL}_2(R_n)$ and $\mathbf{Q} = \mathbf{B}^\star\mathbf{B}$. We first analyse a simplified variant of HAWK with signing $\mathsf{Sign}_{\mathsf{sk}}$ and verification $\mathsf{Vf}_{\mathsf{pk}}$ as in Fig. 6. Hence, define

$$\Pi_{\mathsf{HAWK}} = (\mathsf{HawkKeyGen}, \mathsf{Sign}, \mathsf{Vf}). \tag{46}$$

We discuss these simplifications below, and in Section 6.3.3 consider HAWK without these simplifications, as in Section 2.1.

$\mathsf{Sign}_{(\mathbf{B},\mathsf{hpub})}(M)$

1 : $\mathsf{salt} \leftarrow \mathsf{Rnd}(\mathsf{saltlen}_{\mathsf{bits}})$
2 : $\mathbf{h} = \mathsf{H}(M \parallel \mathsf{salt})$
3 : $\mathbf{w} \leftarrow \widetilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}[\mathbf{h}]$
4 : $\mathbf{s} := \frac{1}{2}(\mathbf{h} + \langle\mathbf{w}\rangle) \in R_n^2$
5 : **return** $\mathsf{sig} := (\mathsf{salt}, \mathbf{s})$

$\mathsf{Vf}_{\mathbf{Q}}(M, \mathsf{sig})$

1 : $(\mathsf{salt}, \mathbf{s}) := \mathsf{sig}$
2 : $\mathbf{h} := \mathsf{H}(M \parallel \mathsf{salt})$
3 : $\mathbf{w} := 2\mathbf{s} - \mathbf{h}$
4 : **check** $\mathsf{sig} \in \{0,1\}^{\mathsf{saltlen}_{\mathsf{bits}}} \times R_n^2$
5 : **check** $\mathbf{w} = \langle\mathbf{w}\rangle$ and $\mathbf{w} \notin R_n \times \{0\}$
6 : **check** $\|\mathbf{w}\|_{\mathbf{Q}} \leq 2\sigma_{\mathsf{ver}} \cdot \sqrt{2n}$
7 : **return** 1 if all checks pass

Figure 6: Simplified HAWK.

The description in Fig. 6 matches the specification of HAWK (see Section 2.1 and in particular Algorithms 2 and 3) up to some small changes.

Several operations look different, but are equivalent. In particular, the sampling of $\mathbf{w}$ in both descriptions coincides, and $\mathbf{s} := \frac{1}{2}(\mathbf{h} + \langle \mathbf{w} \rangle)$ captures the sign flip of $\mathbf{w}$ when $\mathsf{sym\text{-}break}\mathbf{w}$ is false. Finally, whenever the check $\mathsf{sym\text{-}break}(\mathbf{h} - 2\mathbf{s})$ in Algorithm 2 is unsatisfied, it is because either $\mathbf{h} - 2\mathbf{s} = -\mathbf{w} \in R_n \times \{0\}$ and therefore $\mathbf{w} \in R_n \times \{0\}$ also, or the first nonzero coefficient of the second component of $-\mathbf{w}$ is negative, in which case $\langle \mathbf{w} \rangle = -\mathbf{w} \neq \mathbf{w}$. Hence the check $\mathbf{w} = \langle \mathbf{w} \rangle$ and $\mathbf{w} \notin R_n \times \{0\}$ in Vf is equivalent to checking $\mathsf{sym\text{-}break}(\mathbf{h} - 2\mathbf{s})$ as in Algorithm 2.

We now enumerate the simplifications, where we describe what happens in Section 2.1 but not in Fig. 6. First, in Algorithm 2 signing is restarted whenever $\mathbf{w}$ is too long or when various encoding requirements are not satisfied. Second, a hash $M = \mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$ is computed initially in Algorithm 2. Finally, signatures are compressed in Algorithm 2, and decompressed in Algorithm 3.

Consider a (Q)ROM-SUF-CMA attacker $\mathcal{A}^{\mathsf{OSign},\mathsf{H}}(\mathsf{pk})$ against Π_{HAWK}, which on input the public key makes at most q_h (quantum) queries to the random oracle H and at most q_s queries to the signing oracle OSign, and eventually outputs a message forgery pair $(M^\star, \mathsf{sig}^\star)$ with $\mathsf{sig}^\star = (\mathsf{salt}^\star, \mathbf{s}^\star) \in \{0,1\}^{\mathsf{saltlen}_{\mathsf{bits}}} \times R_n^2$. Without loss of generality, we assume $\mathcal{A}$ makes exactly (q_s, q_h) queries to $(\mathsf{OSign}, \mathsf{H})$ respectively.[5] The goal is to turn $\mathcal{A}$ into an algorithm $\mathcal{B}$ that solves an instance of $\mathsf{ac\text{-}omSVP}$, in which $\mathsf{Init}(1^\lambda)$ produces $\sigma = 2\sigma_{\mathsf{sign}}$, $L = 2\sigma_{\mathsf{ver}} \cdot \sqrt{2n}$, μ_K as in Section 6.1.1, and $\mathbf{Q}$ sampled via $(\mathbf{Q}, \mathsf{sk}) \leftarrow \mathsf{HawkKeyGen}$. From here we take it as understood that $\mathsf{ac\text{-}omSVP} = (\mathsf{Init}, \mathsf{samp})$ is as described above.

Theorem 1 (Quantum security of (simplified) Π_{HAWK})**.** *Let $\mathcal{A}$ be an adversary against the* (Q)ROM-SUF-CMA *game making at most q_s queries to* OSign *and at most q_h quantum queries to* H*. Then there exists an adversary $\mathcal{B}$ against the* SAMPLE *game making $q_o = q_s$ queries to* samp*, with running time* $\mathsf{TIME}(\mathcal{B}) = \mathsf{TIME}(\mathcal{A}) + \mathsf{Overhead}(q_s, q_h)$*. This overhead consists of simulating q_h hash queries to* H *and q_s queries to* Sim *(specified in Fig. 7). Furthermore, for each $a \in (1, \infty)$ we have*

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} \leq \left[R_a\big(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \,\|\, D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\big)^{q_s} \cdot \begin{pmatrix} \mathsf{Adv}^{\text{SAMPLE}}_{\mathsf{ac\text{-}omSVP},\mathcal{B}} + O\big(q_h^2 \cdot n \cdot q_s / 2^{2n}\big) \\ + q_s\left(2^{-n} + (q_s - 1) \cdot n \cdot 2^{-2n}\right) \cdot \dfrac{1+\varepsilon}{1-\varepsilon} \end{pmatrix} \right]^{1-1/a} + 2q_s\sqrt{q_h + q_s} \cdot 2^{-\mathsf{saltlen}_{\mathsf{bits}}/2}.$$

The constant hidden by the O is independent of the choice of a.

6.3.1 Simulating the signing queries

We first show we can replace the signing oracle OSign that $\mathcal{A}$ has access to with a simulator Sim that does not know the secret key sk, but instead may sample the $\mathbf{Q}$ dependent distribution $D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$, and that can reprogram the random oracle H; see Fig. 7 (right) below. This, in essence, means that Sim can be replaced by adversary $\mathcal{B}$ against $\mathsf{ac\text{-}omSVP}$ that has access to the samp oracle and can reprogram H. As an intermediate step we consider $\mathsf{Trans}_{\mathsf{sk}}$ as specified in Fig. 7 (left). Letting SUF-CMA-Trans and SUF-CMA-Sim denote the games where the OSign oracle has been replaced by Trans and Sim respectively, we describe the difference in advantage of $\mathcal{A}$ and in particular show

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} \approx \mathsf{Adv}^{\text{SUF-CMA-}\mathsf{Trans}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} \approx \mathsf{Adv}^{\text{SUF-CMA-}\mathsf{Sim}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}}.$$

Note that via OSign the adversary $\mathcal{A}$ depends on sk, and similarly for $\mathcal{A}$ against Trans, which we have highlighted with a subscript. For convenience we may later omit this subscript.

[5] Otherwise, we let $\mathcal{A}$ make dummy queries to H and OSign respectively, with the dummy queries to OSign being on messages different from M^*, so that they do not affect the freshness of a forgery.

$\mathsf{Trans}_{(\mathbf{B},\mathsf{hpub})}(M)$	$\mathsf{Sim}^{D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}}(M)$
1 : $\mathsf{salt} \leftarrow \mathsf{Rnd}(\mathsf{saltlen}_{\mathsf{bits}})$	1 : $\mathsf{salt} \leftarrow \mathsf{Rnd}(\mathsf{saltlen}_{\mathsf{bits}})$
2 : $\mathbf{h} \leftarrow (R_n/2R_n)^2$	2 : $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$
3 : $\mathsf{H}(M \parallel \mathsf{salt}) := \mathbf{h}$	3 : $\mathbf{h} := \mathbf{w} \bmod 2$
4 : $\mathbf{w} \leftarrow \tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}[\mathbf{h}]$	4 : $\mathsf{H}(M \parallel \mathsf{salt}) := \mathbf{h}$
5 : $\mathbf{s} := \frac{1}{2}(\mathbf{h} + \langle \mathbf{w} \rangle) \in R_n^2$	5 : $\mathbf{s} := \frac{1}{2}(\mathbf{h} + \langle \mathbf{w} \rangle) \in R_n^2$
6 : **return** $\mathsf{sig} := (\mathsf{salt}, \mathbf{s})$	6 : **return** $\mathsf{sig} := (\mathsf{salt}, \mathbf{s})$

Figure 7: Oracles $\mathsf{Trans}_{\mathsf{sk}}$ and $\mathsf{Sim}^{D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}}$.

The only difference between OSign and $\mathsf{Trans}_{\mathsf{sk}}$ is that when the former calls $\mathsf{Sign}_{\mathsf{sk}}$ it computes $\mathbf{h} := \mathsf{H}(M \parallel \mathsf{salt})$, while the latter instead reprogrammes $\mathsf{H}(M \parallel \mathsf{salt}) := \mathbf{h} \leftarrow (R_n/2R_n)^2$. We note that, since $\mathsf{Trans}_{\mathsf{sk}}$ has $\mathbf{B}$ it is able to sample $\mathbf{w}$ from $\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}[\mathbf{h}]$.

Considering Lemma 3, the guessing probability of an input $M \parallel \mathsf{salt}$ is at most $2^{-\mathsf{saltlen}_{\mathsf{bits}}}$ for any distribution over M and therefore, letting $\gamma = 2^{-\mathsf{saltlen}_{\mathsf{bits}}}$ and $q_r = q_s$, we have

$$\left| \mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} - \mathsf{Adv}^{\text{SUF-CMA-Trans}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} \right| \leq 2q_s\sqrt{q_h + q_s} \cdot 2^{-\mathsf{saltlen}_{\mathsf{bits}}/2}, \tag{47}$$

where it is understood that the verification Vf^{H} is performed using the possibly reprogrammed H.

We now hop from Trans to Sim. First note that in Trans after line 4, one may redefine $\mathbf{h} := \mathbf{w} \bmod 2$ and $\mathsf{H}(M \parallel \mathsf{salt}) := \mathbf{h}$ without affecting the execution, since $\mathbf{w} \bmod 2 = \mathbf{h}$ holds when $\mathbf{w} \leftarrow \tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}[\mathbf{h}]$ by (42). After this, the only difference between Trans and Sim is that in the former $\mathbf{w}$ is sampled from $\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ and in the latter by $D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$. Therefore, as there are at most q_s independent samples drawn from one of the two distributions, the probability preservation of the Rényi divergence implies for every $a \in (1, \infty)$ that

$$\mathsf{Adv}^{\text{SUF-CMA-Sim}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} \geq \frac{\left(\mathsf{Adv}^{\text{SUF-CMA-Trans}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}}\right)^{a/(a-1)}}{R_a\left(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \parallel D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\right)^{q_s}}, \tag{48}$$

holds, where $R_a\left(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \parallel D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\right)$ can be computed by Lemma 1. We thus conclude that the *validity* of a forgery is preserved when replacing the signing oracle OSign by Sim up to the additive factor of (47) and the multiplicative factor of (48).

Furthermore, the *freshness* of a signature forgery is also preserved, in that we can assume without loss of generality that $\mathcal{A}$ never outputs a forgery (M^*, sig^*) that matches the response of a signing query.

6.3.2 Extracting a fresh short vector

Consider the algorithm $\mathcal{E}^{\mathsf{H}}$ that on input a message forgery pair $(M^\star, \mathsf{sig}^\star)$ computes

$$\mathbf{h}^\star := \mathsf{H}(M^\star \parallel \mathsf{salt}^\star), \text{ and } \mathbf{w}^\star := 2\mathbf{s}^\star - \mathbf{h}^\star \tag{49}$$

and outputs $\mathbf{w}^\star$. Slightly abusing notation, we define the algorithm $\mathcal{B}^{\mathsf{samp}} := \mathcal{E}^{\mathsf{H}} \circ \mathcal{A}^{\mathsf{Sim},\mathsf{H}}$. We take it as understood that when $\mathcal{A}$ makes a query to Sim then $\mathcal{B}$ simulates this query; in particular it obtains $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ as required by Sim by calling its samp oracle. Furthermore, $\mathcal{B}$ locally simulates the random oracle H. It follows that if $\mathcal{A}^{\mathsf{Sim},\mathsf{H}}$ succeeds in producing a valid forgery then the $\mathbf{w}^\star$ output by $\mathcal{B}$ is a short non-zero vector, i.e. $0 < \|\mathbf{w}^\star\|_{\mathbf{Q}} \leq 2\sigma_{\mathsf{verify}} \cdot \sqrt{2n} = L$. That $\mathbf{w}^\star$ is non zero follows from the check on

line 5 of Vf in Fig. 6. It remains to show that $\mathbf{w}^\star$ is a fresh ac-omSVP solution as well, i.e. $\mathbf{w}^\star \notin \mathcal{L}_{\mathsf{samples}}$; recall $\mathcal{L}_{\mathsf{samples}}$ is defined in Fig. 5.

To lower bound the probability that $\mathbf{w}^\star$ is such a fresh ac-omSVP solution, assume that (M^*, sig^*) is a valid and fresh signature forgery such that $\mathbf{w}^\star = \alpha \mathbf{w}_j$ for some $(j, \alpha) \in [q_s] \times \mu_K$, i.e. that $\mathbf{w}^\star$ is not fresh. Note that we have $\mathbf{w}^\star \neq \mathbf{0}$. Let $(M_i, \mathsf{salt}_i, \mathbf{h}_i, \mathbf{s}_i, \mathbf{w}_i)$ be the values taken by $(M, \mathsf{salt}, \mathbf{h}, \mathbf{s}, \mathbf{w})$ in the i^{th} query of $\mathcal{A}$ to Sim, and let $\mathsf{sig}^\star = (\mathsf{salt}^\star, \mathbf{s}^\star)$ be the signature output by $\mathcal{A}$. We distinguish between the following two cases.

First, $M^\star \parallel \mathsf{salt}^\star = M_i \parallel \mathsf{salt}_i$ for some $i \in [q_s]$, where we consider i to be maximal such that the equality holds.[6] Then $\mathbf{h}^\star = \mathbf{h}_i$, and so

$$R_n^2 \ni \mathbf{s}^\star = \frac{1}{2}(\mathbf{h}^\star + \mathbf{w}^\star) = \frac{1}{2}(\mathbf{h}_i + \alpha \mathbf{w}_j).$$

If $i \neq j$ then for any fixed choice of $\mathbf{h}_i$, the probability over the choice of $\mathbf{w}_j$ of there being an $\alpha \in \mu_K$ as above, is at most $n \cdot 2^{-2n} \cdot \frac{1+\varepsilon}{1-\varepsilon}$ by (45). There are q_s choices for i and $q_s - 1$ choices for $j \neq i$. On the other hand, if $i = j$ then we get that

$$R_n^2 \ni \mathbf{s}^* = \frac{1}{2}(\mathbf{h}^* + \mathbf{w}^*) = \frac{1}{2}(\mathbf{h}_i + \alpha \mathbf{w}_i).$$

Furthermore, when $i = j$ we have $\alpha \neq \pm 1$. Indeed, if $\alpha \in \{-1, 1\}$ then $\langle \mathbf{w}_i \rangle = \langle \mathbf{w}^\star \rangle = \mathbf{w}^\star$, where the second equality holds by the validity of sig^* and the first follows from $\alpha \mathbf{w}_i = \mathbf{w}^\star \notin R_n \times \{0\}$ and (41). Therefore $\mathbf{s}^\star = \frac{1}{2}(\mathbf{h}_i + \langle \mathbf{w}_i \rangle) = \mathbf{s}_i$ which contradicts the freshness of sig^*. The probability over the choice of $\mathbf{w}_i$ of there being an $\alpha \in \mu_K \setminus \{\pm 1\}$ as above is at most $2^{-n} \cdot \frac{1+\varepsilon}{1-\varepsilon}$ by (44). There are q_s choices for $i = j$.

Second, $M^\star \parallel \mathsf{salt}^\star \neq M_i \parallel \mathsf{salt}_i$ for every $i \in [q_s]$. In this case we have that

$$R_n^2 \ni \mathbf{s}^\star = \frac{1}{2}(\mathbf{h}^\star + \mathbf{w}^\star) = \frac{1}{2}(\mathbf{h}^\star + \alpha \mathbf{w}_j)\ ,$$

and so $\mathsf{H}(M^\star \parallel \mathsf{salt}^\star) = \mathbf{h}^\star = \alpha \mathbf{w}_j \bmod 2$; furthermore, H has not been reprogrammed throughout the execution at the location $M^\star \parallel \mathsf{salt}^\star$. Hence

$$\mathsf{H}_{\mathsf{init}}(M^\star \parallel \mathsf{salt}^\star) = \mathsf{H}(M^\star \parallel \mathsf{salt}^\star) \in \{\alpha \mathbf{w}_j \bmod 2 \colon (j, \alpha) \in [q_s] \times \mu_K\} =: S, \tag{50}$$

where $\mathsf{H}_{\mathsf{init}}$ is the initial H without any reprogramming. Thus, parsing $\mathcal{A}^{\mathsf{Sim},\mathsf{H}}$ as $\mathcal{C}^{\mathsf{H}_{\mathsf{init}}}$, which runs the calls to Sim (for arbitrary but fixed samples $\mathbf{w}_1, \ldots, \mathbf{w}_{q_s}$ of $D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$) and the reprogramming of H internally, we obtain an algorithm that finds a preimage under $\mathsf{H}_{\mathsf{init}}$ of an element in S, making q_h queries to H_{init}. Note that $\#S \leq nq_s$, so such an algorithm can succeed with probability at most $O\big(q_h^2 \cdot n \cdot q_s / 2^{2n}\big)$ via the standard preimage finding bound. Hence by combining the two cases to get a non fresh forgery,

$$\mathsf{Adv}^{\text{SAMPLE}}_{\text{ac-omSVP},\mathcal{B}} \geq \mathsf{Adv}^{\text{SUF-CMA-Sim}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} - \frac{1}{2^n}\left(O\left(\frac{q_h^2 \cdot n \cdot q_s}{2^n}\right) + \left(q_s + \frac{q_s^2 \cdot n}{2^n}\right) \cdot \frac{1+\varepsilon}{1-\varepsilon}\right) \tag{51}$$

Theorem 1 is now readily proven by combining Eqs. (47), (48) and (51).

6.3.3 Lifting to Unsimplified HAWK

We now argue in a black box manner that the security of Π_{HAWK} is essentially the same as that of the following scheme:

$$\Pi^{\mathsf{full}}_{\mathsf{HAWK}} := (\mathsf{HawkKeyGen}, \mathsf{HawkSign}, \mathsf{HawkVerify}),$$

[6] If i is not the largest, it can be $(M^*, \mathsf{salt}^*) = (M_i, \mathsf{salt}_i)$ yet $\mathbf{h}^* \neq \mathbf{h}_i$ because $\mathbf{h}^*$ is computed via the possibly reprogrammed H.

where the signing and verification are now as specified in Section 2.1. In particular a (Q)ROM-SUF-CMA adversary against this game has a signing oracle $\mathsf{OSign}^{\mathsf{full}}$ that restarts on the same conditions as $\mathsf{HawkSign}$.

The signature scheme Π_{HAWK} lacks the following steps compared to $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$. First, a single evocation of $\mathsf{HawkSign}$ restarts with probability at most pr when certain conditions fail. We define

$$\mathsf{pr} = \max_{\mathsf{sk}} \left[\Pr[\mathsf{HawkSign}_{\mathsf{sk}} \text{ restarts when } \mathbf{h} = \mathbf{h}_{\mathsf{max}}]\right] \tag{52}$$

where we quantify over all possible sk output by $\mathsf{HawkKeyGen}$ and $\mathbf{h}_{\mathsf{max}}$ is the realisation of $\mathbf{h}$ that maximises the probability of a restart for a given sk. This is conservative in the sense that we are assuming an adversary can determine $\mathbf{h}_{\mathsf{max}}$ without knowing sk and can always choose a message M such that $\mathbf{h}_{\mathsf{max}}$ is internally sampled, despite the salt. We note that the public key and choice of H plays no role in signing once $\mathbf{h}$ is chosen.

Given access to OSign one may adaptively query it κ times for some positive integer κ, and choose uniformly from the signatures that would not cause $\mathsf{HawkSign}$ to restart. Note that whether a signature output by OSign should cause a restart is a publicly and efficiently checkable condition. One can then simulate a call to $\mathsf{OSign}^{\mathsf{full}}$ except if all κ queries to OSign would cause a restart in $\mathsf{HawkSign}$. All κ queries would cause a restart with probability bounded above by pr^{κ}, and therefore by a union bound one may simulate q_s queries to $\mathsf{OSign}^{\mathsf{full}}$ using $\kappa \cdot q_s$ queries to OSign, except with probability $q_s \cdot \mathsf{pr}^{\kappa}$.

Second, in both $\mathsf{HawkSign}$ and $\mathsf{HawkVerify}$ an initial hash $M := \mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$ is computed. As long as no restart occurs, a successful forgery against $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$ implies either a collision for H, which, via the standard collision resistance bound [AS04, Zha15], is found with probability $O((q_h + \kappa \cdot q_s)^3/\#\mathcal{M})$ where $\mathcal{M}$ is the codomain for the initial hash $\mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$, or a successful forgery against Π_{HAWK}.

Finally, in $\mathsf{HawkSign}$ the vector $\mathbf{s}$ is compressed for $\mathsf{HawkVerify}$ to decompress later. Since the compression and decompression are public, a forgery attack against such a compressed signature scheme can always be transformed to one against the uncompressed counterpart.

Accounting for the above differences, for every attacker $\mathcal{A}$ against $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$ making (q_s, q_h) respective queries to $(\mathsf{OSign}^{\mathsf{full}}, \mathsf{H})$, there is an attacker $\mathcal{A}'$ against Π_{HAWK} as in (46), making $(\kappa \cdot q_s, q_h)$ respective queries to $(\mathsf{OSign}, \mathsf{H})$, such that

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi^{\mathsf{full}}_{\mathsf{HAWK}},\mathcal{A}} \leq \mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}'} + q_s \cdot \mathsf{pr}^{\kappa} + O\left((q_h + \kappa \cdot q_s)^3/\#\mathcal{M}\right), \tag{53}$$

This concludes the security of $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$.

Theorem 2 (Quantum security of $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$)**.** *Let $\mathcal{A}$ be an adversary against the* (Q)ROM-SUF-CMA *game making at most q_s queries to* OSign *and at most q_h quantum queries to* H *respectively. Then there exists an algorithm $\mathcal{B}$ making $q_o = \kappa \cdot q_s$ queries to* samp*, with running time* $\mathsf{TIME}(\mathcal{B}) = \mathsf{TIME}(\mathcal{A}) + \mathsf{Overhead}(\kappa \cdot q_s, q_h)$*. This overhead consists of simulating q_h hash queries to* H *and $\kappa \cdot q_s$ queries to* Sim *(specified in Fig. 7). Furthermore, for each $a \in (1, \infty)$ and $\kappa \in \mathbb{Z}_{>0}$ we have*

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi^{\mathsf{full}}_{\mathsf{HAWK}},\mathcal{A}} \leq \left[R_a\left(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \parallel D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\right)^{q_s} \cdot \begin{pmatrix} \mathsf{Adv}^{\text{SAMPLE}}_{\mathsf{ac\text{-}omSVP},\mathcal{B}} + O\left(q_h^2 \cdot n \cdot \kappa \cdot q_s/2^{2n}\right) \\ + \kappa \cdot q_s \left(2^{-n} + (\kappa \cdot q_s - 1) \cdot n \cdot 2^{-2n}\right) \cdot \dfrac{1+\varepsilon}{1-\varepsilon} \end{pmatrix}\right]^{1-1/a}$$
$$+ 2\kappa \cdot q_s \sqrt{q_h + \kappa \cdot q_s} \cdot 2^{-\mathsf{saltlen}_{\mathsf{bits}}/2} + q_s \cdot \mathsf{pr}^{\kappa} + O\left((q_h + \kappa \cdot q_s)^3/\#\mathcal{M}\right),$$

where $\mathcal{M}$ is the codomain of $\mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$ *and* pr *is an upper bound on the probability that* $\mathsf{HawkSign}$ *restarts. The constants hidden by the O are independent of a and κ.*

6.4 Classical Security

As our proof is modular, one may substitute parts of the proof of Theorem 2 to obtain better bounds when considering $\mathcal{A}$ that only makes classical queries to H. In (47), where the closeness between Sign and Trans is examined, we may substitute

$$\left|\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}} - \mathsf{Adv}^{\text{SUF-CMA-Trans}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}}\right| \leq 2q_s(q_h+q_s)\cdot 2^{-\mathsf{saltlen}_{\mathsf{bits}}}. \tag{54}$$

Moreover, to control the event (50) of finding a preimage of at most $n \cdot q_s$ elements, classically we have

$$\Pr\left[\mathsf{H}_{\mathsf{init}}(M^\star, \mathsf{salt}^\star) \in S\right] \leq (q_h+1)\cdot n \cdot q_s/2^{2n}.$$

Finally, the probability of finding a collision in (53) can be replaced with $O((q_h+\kappa\cdot q_s)^2/\#\mathcal{M})$, thus

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi^{\mathsf{full}}_{\mathsf{HAWK}},\mathcal{A}} \leq \mathsf{Adv}^{\text{SUF-CMA}}_{\Pi_{\mathsf{HAWK}},\mathcal{A}'} + q_s\cdot \mathsf{pr}^\kappa + O\left((q_h+\kappa\cdot q_s)^2/\#\mathcal{M}\right).$$

We obtain the classical security of HAWK as follows.

Theorem 3 (Classical security of $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$)**.** *Let $\mathcal{A}$ be an adversary against the* (Q)ROM-SUF-CMA *game making at most q_s queries to OSign and at most q_h classical queries to H respectively. Then there exists an algorithm $\mathcal{B}$ making $q_o = \kappa\cdot q_s$ queries to samp, with running time $\mathsf{TIME}(\mathcal{B}) = \mathsf{TIME}(\mathcal{A}) + \mathsf{Overhead}(\kappa\cdot q_s, q_h)$. This overhead consists of simulating q_h hash queries to H and $\kappa\cdot q_s$ queries to Sim (specified in Fig. 7). Furthermore, for each $a\in(1,\infty)$ and $\kappa\in\mathbb{Z}_{>0}$ we have*

$$\begin{aligned}\mathsf{Adv}^{\text{SUF-CMA}}_{\Pi^{\mathsf{full}}_{\mathsf{HAWK}},\mathcal{A}} \leq &\left[R_a\left(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \,\|\, D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\right)^{q_s}\cdot\left(\begin{array}{l}\mathsf{Adv}^{\text{SAMPLE}}_{\mathsf{ac\text{-}omSVP},\mathcal{B}} + (q_h+1)\cdot n\cdot\kappa\cdot q_s/2^{2n}\\ +\kappa\cdot q_s\left(2^{-n}+(\kappa\cdot q_s-1)\cdot n\cdot 2^{-2n}\right)\cdot\dfrac{1+\varepsilon}{1-\varepsilon}\end{array}\right)\right]^{1-1/a}\\ &+2\kappa\cdot q_s(q_h+\kappa\cdot q_s)\cdot 2^{-\mathsf{saltlen}_{\mathsf{bits}}} + q_s\cdot\mathsf{pr}^\kappa + O\left((q_h+\kappa\cdot q_s)^2/\#\mathcal{M}\right),\end{aligned}$$

where $\mathcal{M}$ is the codomain for the initial hash $\mathsf{H}(\mathsf{m}\,\|\,\mathsf{hpub})$ and pr is an upper bound on the probability that $\mathsf{HawkSign}$ restarts. The constants hidden by the O are independent of a and κ.

6.5 To table based sampling HAWK

Theorem 2 is not quite the end of our reduction. There is one final leap; from $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$ to the HAWK specified in Section 3, namely to a HAWK that samples from a table based approximation of $\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$. This table based approximation is achieved by sampling one entry at a time, as described in Section 3.5.1 and in particular in Algorithm 14.

We first consider the Rényi divergence between $D_{2\mathbb{Z},2\sigma_{\mathsf{sign}}}$ and $D_{2\mathbb{Z}+1,2\sigma_{\mathsf{sign}}}$ and their table based approximations. Then, in Theorem 4 we collect the results of this section and those above to finally relate HAWK as specified to omSVP.

Table based sampling. Let D'_b be the distribution of a single coefficient $\mathbf{d}[r]$ for $0\leq r<2n$ in Algorithm 14, given that $\mathbf{t}[r]=b$ for $b\in\{0,1\}$. Internally, the distribution D'_b is determined by the values in T_b as part of Table 5, and has support some subset of $2\mathbb{Z}+b$.

First, we consider two distributions, P_1 and P_2, where P_1 is the distribution of all randomness that is involved in the signing of at most q_s signatures with the table based sampler, which are requested by an adversary against HAWK, and P_2 is the distribution

when using exact discrete Gaussian samplers. Let E be the event of the adversary providing a forgery in either case. If $P_1(E) \geq 2^{-\lambda}$ then, by the probability preservation property of the Rényi divergence [Pre17, Sec. 3.3],[7] for all $a \in (1, \infty)$ we have

$$P_2(E) \geq P_1(E)^{a/(a-1)} / R_a(P_1 \,\|\, P_2) \geq P_1(E) \cdot \frac{1}{R_a(P_1 \,\|\, P_2) \cdot 2^{\lambda/(a-1)}}. \tag{55}$$

Note that if $P_1(E) < 2^{-\lambda}$ then we are done; our adversary against table based sampling HAWK outputs a forgery with small enough probability.

It is now left to determine $R_a(P_1 \,\|\, P_2)$ for a well chosen a. For sampling one coefficient from either D'_b or $D_{2\mathbb{Z}+b,2\sigma_{\mathsf{sign}}}$ the Rényi divergence can be computed numerically. Then, the multiplicativity and data processing inequality give us

$$R_a(P_1 \,\|\, P_2) \leq \left(\max_{b \in \{0,1\}} R_a\big(D'_b \,\|\, D_{2\mathbb{Z}+b,2\sigma_{\mathsf{sign}}}\big) \right)^{2nq_s}, \tag{56}$$

as there are at most q_s signature queries, each requiring $2n$ discrete Gaussian samples.

In the auxiliary data[8] the Rényi divergences are calculated for HAWK-$\{256, 512, 1024\}$. These show that for HAWK-512 and HAWK-1024 the Rényi divergence for both $b \in \{0, 1\}$ is smaller than $1 + 2^{-78}$ at an order $a = 513$, which implies at most one bit of security being lost in this reduction by (6). In fact, the script shows that the security loss is even smaller than one bit by considering higher orders a. In particular, for HAWK-512, at order $a = 37286$ we obtain $P_2(E) \geq P_1(E)/1.07$, and for HAWK-1024 at order $a = 4181$ we obtain $P_2(E) \geq P_1(E)/1.17$.

Total security loss when reducing omSVP to specified HAWK. We now relate the security of HAWK as specified in Section 3 to omSVP. In particular when the sampling discussed in Section 2.1 is implemented via Section 3.5.1 we define

$$\Pi^{\mathsf{tables}}_{\mathsf{HAWK}} = (\mathsf{HawkKeyGen}, \mathsf{HawkSign}, \mathsf{HawkVerify}),$$

Specifically, $\Pi^{\mathsf{tables}}_{\mathsf{HAWK}}$ is identical to $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$ except HawkSign is now Algorithm 15, which calls a sampler that uses precomputed tables, whereas $\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$ sampled from $\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}[\mathbf{h}]$.

Theorem 4. *Let $\lambda \in \{128, 256\}$, $q_s \leq 2^{64}$ and $q_h \leq 2^{\lambda}$. Let $\mathcal{A}$ be a (quantum) adversary for the* (Q)ROM-SUF-CMA *game against $\Pi^{\mathsf{tables}}_{\mathsf{HAWK}}$ that runs in time at most $t_{\mathcal{A}} \geq 1$, makes (q_s, q_h) queries to* (OSign, H) *respectively, and has success probability*

$$\varepsilon_{\mathcal{A}} = \mathsf{Adv}^{\text{SUF-CMA}}_{\Pi^{\mathsf{tables}}_{\mathsf{HAWK}},\mathcal{A}}(\lambda),$$

such that $t_{\mathcal{A}}/\varepsilon_{\mathcal{A}} < 2^{\lambda}$. In particular $\lambda = 128$ corresponds to HAWK-512 *and $\lambda = 256$ corresponds to* HAWK-1024. *Moreover, assume* $\mathsf{saltlen}_{\mathsf{bits}}$ *and $\#\mathcal{M}$ are taken arbitrarily large (in particular, at least as large as listed in Table 4).*

Then, there exists a (quantum) adversary $\mathcal{B}$ for the SAMPLE *game with the appropriate* $\mathsf{Init}(1^{\lambda})$ *running in time at most $t_{\mathcal{B}} = t_{\mathcal{A}} + \mathsf{Overhead}(\kappa \cdot q_s, q_h)$. This overhead consists of making $\kappa \cdot q_s$ queries to* samp *and simulating q_h queries to* H. *The success probability of $\mathcal{B}$ satisfies*

$$\mathsf{Adv}^{\text{SAMPLE}}_{\mathsf{ac\text{-}omSVP},\mathcal{B}}(\lambda) > \frac{1}{2}\varepsilon_{\mathcal{A}}.$$

Proof. Note $\varepsilon_{\mathcal{A}} > 2^{-\lambda}$ as $t_{\mathcal{A}} \geq 1$. We make the following five hops

1. Table based sampling ($\Pi^{\mathsf{tables}}_{\mathsf{HAWK}}$) against (Q)ROM-SUF-CMA,

[7] See https://tprest.github.io/pdf/pub/renyi.pdf for the most recent version.

[8] See https://github.com/hawk-sign/aux/blob/main/code/generate_C_tables.py for the computation.

2. Exact sampling ($\Pi^{\mathsf{full}}_{\mathsf{HAWK}}$) against (Q)ROM-SUF-CMA,
3. Simplified signing (Π_{HAWK}) against (Q)ROM-SUF-CMA,
4. Reprogramming H with uniform targets $\mathbf{h}$ ($\mathcal{A}^{\mathsf{Trans},\mathsf{H}}$),
5. Reprogramming H with targets $\mathbf{h} = \mathbf{w} \bmod 2$ with $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ ($\mathcal{A}^{\mathsf{Sim},\mathsf{H}}$),
6. Using oracle calls to samp, against SAMPLE ($\mathcal{B}^{\mathsf{samp}}$).

For $i \in \{2,3,4,5\}$, we use ε_i to mean the advantage of $\mathcal{A}$ transformed to the ith game.

Transforming $\mathcal{A}$ from 1 to 2, using Eqs. (55) and (56) and the computation from the auxiliary data, we obtain

$$\varepsilon_2 \geq \varepsilon_{\mathcal{A}}/1.17. \tag{57}$$

We now estimate $\mathsf{pr} \leq \frac{1}{2}$, and discuss this more in Section 6.7. Let $\kappa = 67 + \lambda$ and assume $\#\mathcal{M}$ is large enough such that $O\left((q_h + \kappa \cdot q_s)^3/\#\mathcal{M}\right) \leq 2^{-\lambda}/16$ in (53). This allows us to bound the additive loss in (53) above by $2^{-\lambda}/8 + 2^{-\lambda}/16 = \frac{3}{16}2^{-\lambda}$, giving

$$\varepsilon_3 \geq \varepsilon_2 - \frac{3}{16}2^{-\lambda} \geq 0.60\,\varepsilon_{\mathcal{A}}. \tag{58}$$

Now to hop from 3 to 4, we use (47). We assume $\mathsf{saltlen}_{\mathsf{bits}}$ is sufficiently large. In particular, taking $\mathsf{saltlen}_{\mathsf{bits}}$ large enough such that the term in (47) is at most $2^{-\lambda}/16$ (requiring it to be larger than in Table 4), gives

$$\varepsilon_4 \geq \varepsilon_3 - 2^{-\lambda}/16 \geq \varepsilon_2 - \frac{1}{4}2^{-\lambda}. \tag{59}$$

By Lemma 1 we compute the concrete security loss from 4 to 5. The function `security_loss_cosets` of the `generate_C_tables.py` script[9] computes the minimal security loss in (48). Here (59) and our starting assumption on $\varepsilon_{\mathcal{A}}$ imply $\varepsilon_4 \geq 2^{-(\lambda+10)}$ holds, which allows us the following multiplicative loss from the Rényi divergence

$$\varepsilon_5 \geq \varepsilon_4 \cdot \left(\varepsilon_4^{1/(a-1)}/R_a\big(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \,\|\, D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\big)^{q_s}\right) \geq \varepsilon_4/\left(2^{\frac{\lambda+10}{a-1}} R_a\big(\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}} \,\|\, D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}\big)^{q_s}\right).$$

In particular, we calculated for HAWK-512 that $\varepsilon_5 \geq \varepsilon_4/1.04$ holds by taking the Rényi divergence at order $a = 5051$, while for HAWK-1024 we get $\varepsilon_5 \geq \varepsilon_4/1.03$ by taking the Rényi divergence at order $a = 14428$. Taking the worst, from (59) we obtain

$$\varepsilon_5 \geq 0.60\,\varepsilon_{\mathcal{A}}/1.04 \geq 0.57\,\varepsilon_{\mathcal{A}}. \tag{60}$$

For the final hop we use (51). We use $\varepsilon < \frac{1}{2}$, $q_s \leq 2^{64} \leq 2^{n/4}$ and $q_h \leq 2^{\lambda} \leq 2^{n/4}$, yielding the negligible additive loss of at most $(1 + 3q_s)/2^n$. This gives us

$$\varepsilon_{\mathcal{B}} \geq \frac{1}{2}\,\varepsilon_{\mathcal{A}}. \tag{61}$$

Hence the success probability of $\mathcal{B}$ decreases by at most a factor of two. □

Let $\lambda \in \{128, 256\}$. Theorem 4 tells us that if there is an adversary $\mathcal{A}$ against HAWK-(4λ) with $t_{\mathcal{A}}/\varepsilon_{\mathcal{A}} < 2^{\lambda}$ that makes (q_s, q_h) queries to $(\mathsf{OSign}, \mathsf{H})$, then there exists a $(t_{\mathcal{B}}, \varepsilon_{\mathcal{B}})$ adversary against the relevant ac-omSVP instance that makes $\kappa \cdot q_s$ queries to samp and has $t_{\mathcal{B}}/\varepsilon_{\mathcal{B}} \leq 2t_{\mathcal{B}}/\varepsilon_{\mathcal{A}}$.

It remains to consider $t_{\mathcal{B}}$. We see that $t_{\mathcal{B}}/t_{\mathcal{A}} = 1 + \mathsf{Overhead}(\kappa \cdot q_s, q_h)/t_{\mathcal{A}}$. In the worst case, where $\mathcal{A}$ makes no queries to H and performs no computation, while maximising q_s, this quotient is $1 + \kappa = 1 + \lambda + 67 \leq 324 < 2^9$.

Therefore $t_{\mathcal{B}}/\varepsilon_{\mathcal{B}} \leq 2t_{\mathcal{B}}/\varepsilon_{\mathcal{A}} < 2^{10}t_{\mathcal{A}}/\varepsilon_{\mathcal{A}} < 2^{\lambda+10}$. The following corollary takes the contrapositive of the above statement.

[9] See https://github.com/hawk-sign/aux.

Corollary 1. *Let* $\lambda \in \{128, 256\}$, $n = 4\lambda$, $\kappa = \lambda + 67$, $q_s \leq 2^{64}$ *and* $q_h \leq 2^\lambda$. *Consider the* ac-omSVP *instance where* $\mathsf{Init}(1^\lambda)$ *returns* $\mathbf{Q}$ *sampled from* HawkKeyGen, μ_K *as in* Section 6.1.1, $L = 2\sqrt{2n} \cdot \sigma_{\mathsf{verify}}$ *and* $\sigma = 2\sigma_{\mathsf{sign}}$. *If, for all* $t \geq 1$ *there is no* (t, ε)-SAMPLE *adversary against* ac-omSVP *that makes at most* $q_o = \kappa \cdot q_s$ *queries to* samp *and has* $t/\varepsilon < 2^{\lambda+10}$, *then there is no* (t, ε)-(Q)ROM-SUF-CMA *adversary against* $\Pi^{\mathsf{tables}}_{\mathsf{HAWK}}$ *that makes* (q_s, q_h) *queries to* $(\mathsf{OSign}, \mathsf{H})$ *such that* $t/\varepsilon < 2^\lambda$.

Proof. This follows from the discussion above. □

6.6 Search module lattice isomorphism problem

The search module lattice isomorphism problem was formalised in [DPPvW22b, Sec. 6.1] building upon the general lattice isomorphism formalism for cryptography proposed in [DvW22]. Similar problems were considered before during the cryptanalytic effort against early NTRU based signature schemes [GS02, Szy03].

Intuitively it is a problem where one is given two matrices over a field K from the same equivalence class under some relation, and must find an invertible matrix over the ring of integers R that relates them.

Recall, for a CM-field K and $r \in \mathbb{Z}_{\geq 1}$ we define $\mathcal{H}^{>0}_r(K) \subset K^{r\times r}$ as the set of all $\mathbf{Q} \in K^{r\times r}$ such that $\mathbf{Q} = \mathbf{Q}^\star$ and $\mathrm{Tr}(\mathbf{v}^\star\mathbf{Q}\mathbf{v}) > 0$ for all $\mathbf{v} \in K^r \setminus \{\mathbf{0}\}$, i.e. matrices that represent Hermitian positive definite forms. We then define an equivalence relation over $\mathcal{H}^{>0}_r(K)$ where $\mathbf{Q} \sim \mathbf{Q}'$ if and only if there exists $\mathbf{U} \in \mathsf{GL}_r(R)$ such that $\mathbf{Q}' = \mathbf{U}^\star\mathbf{Q}\mathbf{U}$ and write $[\mathbf{Q}]$ for the equivalence class of $\mathbf{Q}$. This is a well defined equivalence relation as $\mathsf{GL}_r(K)$ is a group and therefore has an identity, inverses and is closed under composition.

Note that in [DPPvW22b] a different but equivalent definition of $\sim$ is given that requires $\mathbf{U}$ to be in the special linear group. This simplifies proofs regarding average case instances of module lattice isomorphism problems. The simplification of key generation in HAWK compared to HAWK-AC22 no longer allows these average case distributions so we simplify our presentation here, see Section 6.7 for more details. We now give the definition of worst case smLIP.

Definition 7 (Worst case smLIP)**.** Given a CM-field K, its ring of integers R and $\mathbf{Q} \in \mathcal{H}^{>0}_r(K)$, an instance of the worst case search module lattice isomorphism problem $\mathsf{wc{-}smLIP}^{\mathbf{Q}}_{K,r}$ is given by any $\mathbf{Q}' \in [\mathbf{Q}]$. A solution is $\mathbf{U} \in \mathsf{GL}_r(R)$ such that $\mathbf{Q}' = \mathbf{U}^\star\mathbf{Q}\mathbf{U}$.

Note that K and r alone do not define the problem, instances are within an equivalence class determined by $\mathbf{Q}$. The specific equivalence class we work in for HAWK is $[\mathbf{I}_2(K)]$ for K some power of two cyclotomic, i.e. $\mathbf{Q} = \mathbf{I}_2(K)$ in the above definition. Furthermore, the $\mathbf{Q}'$ we are given in HAWK are not worst case, they follow the distribution of public keys output by key generation. While this is in some formal sense an 'average case' distribution, it differs from the average case distributions defined in [DvW22, DPPvW22b]. Recall that (after passing to the unstructured setting) finding $\mathbf{U} \in \mathsf{GL}_{2n}(\mathbb{Z})$ that solves the corresponding search lattice isomorphism problem instance is precisely the cryptanalytic experiment we perform in Section 5.1.

6.7 Discussion

Changes to key generation and smLIP. An algorithmic difference between HAWK as specified in Section 3 and HAWK-AC22 is the change in sampling during key generation. In HAWK-AC22 the entries of f, g are sampled from a discrete Gaussian over $\mathbb{Z}$ with a given width, whereas in HAWK this is replaced by a centred binomial distribution of similar width. The experiments of Section 5.1 suggest this simplification makes no difference to the practical hardness of secret key recovery via lattice reduction. However, it does not allow us to appeal to the definitions of average case (module) search lattice isomorphism

instances given in [DvW22, Sec. 3.1] and [DPPvW22b, Sec. 6.1]. In the below we follow the notation of [DPPvW22b] which specialises to that of [DvW22] when $K = \mathbb{Q}$ and their $s = \sqrt{2\pi}\sigma$.

The average case distributions are defined algorithmically within an equivalence class $[\mathbf{Q}]$ for some $\mathbf{Q} \in \mathcal{H}_r^{>0}(K)$ and a width parameter σ. We denote an average case distribution by $\mathsf{AC}_\sigma([\mathbf{Q}])$ and the algorithm that samples it by ac_σ.

Distributions $\mathsf{AC}_\sigma([\mathbf{Q}])$ have the useful property that, given any $\mathbf{Q}' \in [\mathbf{Q}]$ and if σ is large enough, then on input $\mathbf{Q}'$ the algorithm ac_σ outputs a sample according to $\mathsf{AC}_\sigma([\mathbf{Q}])$, along with an isomorphism between the sample and $\mathbf{Q}'$. Put simply, provided σ is chosen large enough one can use any representative element of the equivalence class $[\mathbf{Q}]$ to sample from $\mathsf{AC}_\sigma([\mathbf{Q}])$ via ac_σ.

This effectively allows a rerandomisation of worst case instances to average case instances, and therefore worst case to average case reductions when σ is large, see [DvW22, Lem. 3.9] and [DPPvW22b, Lem. 5]. A seemingly necessary property of the distributions sampled internally in ac_σ to produce an output according to $\mathsf{AC}_\sigma([\mathbf{Q}])$ is that they are radial; the probability mass of a sampled element depends only on its ℓ^2 norm. This is not true of the centred binomial distribution, for example if $X \sim \mathrm{Bin}(\eta)^4$ for $\eta \geq 2$ then $(2, 0, 0, 0)$ has the same length as $(1, 1, 1, 1)$ but a different probability mass.

While we therefore cannot claim any worst case to average case reduction for the problem of recovering the secret key of HAWK from the public key, we note that this was already the case for the concrete parameters of HAWK-AC22, where the discrete Gaussian was used in key generation. This is because the width σ_{kgen} used in HAWK-AC22 was too small and did not satisfy [DPPvW22b, Lem. 5].

Discussion of omSVP parametrisation. The ac-omSVP instances implicit in our reduction from HAWK to omSVP are trivially solvable. In particular, without making any queries to the samp oracle, an adversary can submit $\mathbf{w}^\star = (1\ 0)$. In this case $\|\mathbf{w}^\star\|_{\mathbf{Q}} = \|(f, g)\| \approx \sqrt{2n}\sigma_{\mathsf{krsec}} \leq 2\sqrt{2n}\sigma_{\mathsf{verify}} = L$. This is a consequence of us choosing σ_{krsec} as small as our practical cryptanalysis will allow us, see Section 5.1, and not being able to choose σ_{verify} small enough to mitigate this without unacceptably increasing the restart probability. This is an instance of us choosing to follow practical cryptanalysis for setting parameters, and using our theoretical reduction to convince us of the soundness of our design; we could easily fix this by increasing η but this would also increase the size of our public keys. We note that, while this decision makes our reduction vacuous, we do not know practically how to use these public key vectors to create forgeries against HAWK. More generally, one can consider an adversary that performs lattice reduction on the form $\mathbf{Q}$ and possibly combines the result with samples from the samp oracle.

Before we discuss further, we introduce a lemma that gives a concentration result (from above and below) on the lengths of vectors sampled from a discrete Gaussian distribution whenever ε is a negligible function of the dimension of the lattice. We specialise it to $\mathbb{Z}^n$ and to the width σ which is a factor $\sqrt{2\pi}$ smaller than in [BF11].

Lemma 4 ([BF11, Prop. 7])**.** *Let $\sigma \geq \eta_\varepsilon(\mathbb{Z}^n)$ for ε a negligible function of n. For any constant $\alpha > 0$ and $\mathbf{w} \leftarrow D_{\mathbb{Z}^n,\sigma}$*

$$\Pr\left[(1-\alpha)\sigma\sqrt{n} \leq \|\mathbf{w}\| \leq (1+\alpha)\sigma\sqrt{n}\right] \geq 1 - \mathsf{negl}(n)$$

In particular, $\|\mathbf{w}\|^2$ for $\mathbf{w} \leftarrow D_{\mathbb{Z}^n,\sigma}$ is concentrated around $n\sigma^2$. We could rewrite the above as $\|\mathbf{w}\|_{\mathbf{Q}}^2$ for $\mathbf{w} \leftarrow D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ is concentrated around $8n\sigma_{\mathsf{sign}}^2$ to make it more relevant to HAWK.

Asymptotically, whenever we satisfy the conditions of Lemma 4 if we are able to parametrise such that $2\sigma_{\mathsf{verify}} < \sigma_{\mathsf{krsec}}$ and our threshold behaviour on lengths during lattice reduction is accurate, then no vectors found during lattice reduction will be short enough

to act as omSVP solutions until vectors of length one begin being discovered. One can also consider combining vectors discovered during lattice reduction on $\mathbf{Q}$ with $\mathbf{w}$ received from samp. This was conceptualised as a weak forgery attack in [DPPvW22b, Sec. 4.3].

Another parametrisation to note is $\sigma_{\mathsf{verify}} > \sqrt{2}\sigma_{\mathsf{sign}}$. In this instance $\mathbf{w}_1, \mathbf{w}_2$ from samp are such that $\|\mathbf{w}_1 + \mathbf{w}_2\|_{\mathbf{Q}}$ will be short enough to be a omSVP solution with overwhelming probability. We assume that $\mathbf{w}_1 + \mathbf{w}_2 \notin \mathcal{L}_{\mathsf{samples}}$. This again does not immediately lead to a forgery for HAWK, but if $\mathbf{h}_1, \mathbf{h}_2 \in (R_n/2R_n)^2$ are such that $\mathbf{w}_i \in \mathbf{h}_i + 2R_n$ then finding a message and salt such that $\mathsf{H}(\mathsf{m} \parallel \mathsf{salt}) \in \{\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_1 + \mathbf{h}_2\}$ will lead to a forgery. Querying many $\mathbf{w}_i$ from samp may ease forgery, as one effectively has quadratically many targets $\mathbf{h}_i + \mathbf{h}_j$ with $i \neq j$ to find preimages of. Our parameter sets are such that $\sigma_{\mathsf{verify}} < \sqrt{2}\sigma_{\mathsf{sign}}$ by some margin.

In a more idealised setting, we are effectively asking a omSVP adversary to perform a relaxed notion of lattice sieving for just one vector. The standard heuristic from that literature is that the vectors $\mathbf{w}$ from samp are i.i.d. uniform points on a sphere of radius $2\sqrt{2n}\sigma_{\mathsf{sign}}$ [NV08]. Rather than asking for enough distinct shorter combinations of such $\mathbf{w}$ to recurse a sieving operation, the SAMPLE game asks for just one combination of $\mathbf{w}$, provided it does not fall into $\mathcal{L}_{\mathsf{samples}}$, that can actually be slightly longer, by a factor of $\sigma_{\mathsf{verify}}/\sigma_{\mathsf{sign}} \geq 1$. Optimising k-sieving techniques [BLS16, HK17] for this setting where a single relaxed combination is sought may be an interesting approach. Also, whether having access to (lattice reduction on) $\mathbf{Q}$ can help an adversary in this setting, or whether one can reduce to a variant of omSVP where the adversary is not given $\mathbf{Q}$ by showing something equivalent can be obtained efficiently by using the samp oracle, are other interesting avenues for further work.

Finally, the applicability of learning style attacks [NR09, DN12] to the particular ac-omSVP instances relevant to HAWK should be considered. Lemma 1 tells us that $\tilde{D}_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ and $D_{\mathbf{Q},2\sigma_{\mathsf{sign}}}$ are close, which relate to $\mathbf{w}$ sampled internally during HAWK signing and the output of the samp oracle in the SAMPLE game. Recall that $\mathbf{w}$ is public in HAWK signing since it can be computed from $\mathbf{s}$ and $\mathbf{h}$. Therefore, if one can mount such a learning style attack against HAWK signatures, then one can mount such a learning attack against the SAMPLE game using $\mathbf{w}$ from samp with a very similar probability.

Altogether now, parameter selection. We now describe our rationale for parameter selection, which one can see algorithmically as the function `find_params`.[10]

We begin with the selection of σ_{sign}. We need $\sigma_{\mathsf{sign}} \geq \eta_\varepsilon(\mathbb{Z}^{2n})$ for a suitable ε to appeal to Lemma 2, and also to ensure the Rényi divergence of Lemma 1 is small enough. Ultimately "small enough" is quantified by what are acceptable sizes for the respective losses in the reductions of Sections 6.3 to 6.5 and the effect on the practical cryptanalysis of Section 5, but we find setting $\varepsilon = 1/\sqrt{q_s \cdot \lambda}$ as in FALCON to be a reasonable value. Increasing ε, and therefore decreasing σ_{sign}, decreases the signature sizes.

From σ_{sign} we must select σ_{verify} that is small enough that forgery is not too easy, but also large enough that not too many signatures are rejected due to $\|\mathbf{w}\|_{\mathbf{Q}}$ being too long. In particular we take $\sigma_{\mathsf{verify}} \in (\sigma_{\mathsf{sign}}, \sigma_{\mathsf{krsec}}]$. If $\sigma_{\mathsf{verify}} > \sigma_{\mathsf{krsec}}$ then in our simulation model, e.g. Table 7, the blocksizes required for forgery would become smaller than those for secret key recovery. Though it is always true formally that forgery is no harder than secret key recovery, here we are trying to practically ensure that one must recover the secret key to forge. The value we choose for σ_{verify} in this range is maximal such that the weak forgery attack considered in [DPPvW22b, Sec. 4.3] is sufficiently hard. Determining more accurately the value of that attack, and in particular showing that it is ineffective, would allow us to increase σ_{verify}. However, we note that in practice the vast majority of restarts in signing are caused because signatures cannot be properly encoded, so *decreasing* σ_{verify}, which can only improve security, would not penalise us much.

[10] https://github.com/hawk-sign/aux/blob/main/code/find_params.sage

The value σ_{krsec} is not chosen by us, but determined by simulation and our practical cryptanalysis. To ensure that HawkKeyGen does not resample f, g too many times, the distribution that the entries of these polynomials come from must be wide enough to satisfy $\|(f, g)\|^2 > 2n\sigma_{\mathsf{krsec}}^2$ often enough. Thankfully, the centred binomials with $\eta = 4$ and $\eta = 8$ are sufficient. In particular, they have standard deviations $\sqrt{2}$ and 2 respectively which are either close to or above σ_{krsec} for HAWK-512 and HAWK-1024.

We now discuss our choice of $\mathsf{saltlen}_{\mathsf{bits}}$ and the codomain of the hash $M \leftarrow \mathsf{H}(\mathsf{m} \parallel \mathsf{hpub})$ in Algorithm 15. We choose $\mathsf{saltlen}_{\mathsf{bits}} = \lambda + \log_2(q_s)$ for the practical reason that the probability of a collision in $\mathsf{H}(\mathsf{m} \parallel \mathsf{salt})$ over q_s queries is at most $2^{-\lambda}$. However, in light of Eqs. (47) and (54), this is not sufficient either classically or quantumly to make this leap in the reduction non vacuous. Increasing $\mathsf{saltlen}_{\mathsf{bits}}$ by b bits to account for this would require b bits more randomness per signature attempt and increase the signature size by b bits. Similarly, we realise $M \leftarrow \mathsf{SHAKE256}(\mathsf{m} \parallel \mathsf{hpub})[0 : 512]$. Depending on the number of quantum queries to H allowed, this may make (53) vacuous.

We note that $\mathsf{saltlen}_{\mathsf{bits}}$ is chosen as in FALCON, and that the codomain of the initial hash is chosen as in DILITHIUM. Both are easy to change if required, with little effect on the efficiency of HAWK.

Finally, in Theorem 4 we concretised the restart probability pr defined in (52) as at most one half. It seems that this value may be possible to upper bound pr by considering the restrictions put on f, g, F, G in HawkKeyGen and distribution on lengths of $\mathbf{x}$ in HawkSign. The smaller we are able to make this upper bound, the smaller the parameter κ in our reduction may be taken.

References

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe, *Post-quantum key exchange - A new hope*, USENIX Security 2016 (Thorsten Holz and Stefan Savage, eds.), USENIX Association, August 2016, pp. 327–343.

[AGPS20] Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck, *Estimating quantum speedups for lattice sieves*, ASIACRYPT 2020, Part II (Shiho Moriai and Huaxiong Wang, eds.), LNCS, vol. 12492, Springer, Heidelberg, December 2020, pp. 583–613.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer, *Revisiting the expected cost of solving uSVP and applications to LWE*, ASIACRYPT 2017, Part I (Tsuyoshi Takagi and Thomas Peyrin, eds.), LNCS, vol. 10624, Springer, Heidelberg, December 2017, pp. 297–322.

[AS04] Scott Aaronson and Yaoyun Shi, *Quantum lower bounds for the collision and the element distinctness problems*, Journal of the ACM (JACM) **51** (2004), no. 4, 595–605.

[AWHT16] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi, *Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator*, EUROCRYPT 2016, Part I (Marc Fischlin and Jean-Sébastien Coron, eds.), LNCS, vol. 9665, Springer, Heidelberg, May 2016, pp. 789–819.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven, *New directions in nearest neighbor searching with applications to lattice sieving*, 27th SODA (Robert Krauthgamer, ed.), ACM-SIAM, January 2016, pp. 10–24.

[BDH+23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier, *TurboSHAKE*, Cryptology ePrint Archive, Report 2023/342, 2023, https://eprint.iacr.org/2023/342.

[BF11] Dan Boneh and David Mandell Freeman, *Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures*, PKC 2011 (Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, eds.), LNCS, vol. 6571, Springer, Heidelberg, March 2011, pp. 1–16.

[BGPSD23] Huck Bennett, Atul Ganju, Pura Peetathawatchai, and Noah Stephens-Davidowitz, *Just how hard are rotations of $\mathbb{Z}^n$? algorithms and cryptography with the simplest lattice*, Advances in Cryptology – EUROCRYPT 2023 (Carmit Hazay and Martijn Stam, eds.), 2023, pp. 252–281.

[BLL+15] Shi Bai, Adeline Langlois, Tancrède Lepoint, Damien Stehlé, and Ron Steinfeld, *Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance*, ASIACRYPT 2015, Part I (Tetsu Iwata and Jung Hee Cheon, eds.), LNCS, vol. 9452, Springer, Heidelberg, November / December 2015, pp. 3–24.

[BLS16] Shi Bai, Thijs Laarhoven, and Damien Stehlé, *Tuple lattice sieving*, LMS Journal of Computation and Mathematics **19** (2016), no. A, 146–162.

[BM21] Tamar Lichter Blanks and Stephen D. Miller, *Generating cryptographically-strong random lattice bases and recognizing rotations of $\mathbb{Z}^n$*, Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021 (Jung Hee

Cheon and Jean-Pierre Tillich, eds.), Springer, Heidelberg, 2021, pp. 319–338.

[CDF+20] Cas Cremers, Samed Düzlü, Rune Fiedler, Marc Fischlin, and Christian Janson, *BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures*, Cryptology ePrint Archive, Report 2020/1525, 2020, https://eprint.iacr.org/2020/1525.

[CDF+21] ———, *BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures*, 2021 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, May 2021, pp. 1696–1714.

[CN11] Yuanmi Chen and Phong Q. Nguyen, *BKZ 2.0: Better lattice security estimates*, ASIACRYPT 2011 (Dong Hoon Lee and Xiaoyun Wang, eds.), LNCS, vol. 7073, Springer, Heidelberg, December 2011, pp. 1–20.

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi, *LWE with side information: Attacks and concrete security estimation*, CRYPTO 2020, Part II (Daniele Micciancio and Thomas Ristenpart, eds.), LNCS, vol. 12171, Springer, Heidelberg, August 2020, pp. 329–358.

[DEP23] Léo Ducas, Thomas Espitau, and Eamonn W. Postlethwaite, *Finding short integer solutions when the modulus is small*, To appear at CRYPTO'23, 2023.

[DN12] Léo Ducas and Phong Q. Nguyen, *Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures*, ASIACRYPT 2012 (Xiaoyun Wang and Kazue Sako, eds.), LNCS, vol. 7658, Springer, Heidelberg, December 2012, pp. 433–450.

[DPPvW22a] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel van Woerden, *Hawk: Module LIP makes lattice signatures fast, compact and simple*, Cryptology ePrint Archive, Report 2022/1155, 2022, https://eprint.iacr.org/2022/1155.

[DPPvW22b] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden, *Hawk: Module LIP makes lattice signatures fast, compact and simple*, ASIACRYPT 2022, Part IV (Shweta Agrawal and Dongdai Lin, eds.), LNCS, vol. 13794, Springer, Heidelberg, December 2022, pp. 65–94.

[Duc18] Léo Ducas, *Shortest vector from lattice sieving: A few dimensions for free*, EUROCRYPT 2018, Part I (Jesper Buus Nielsen and Vincent Rijmen, eds.), LNCS, vol. 10820, Springer, Heidelberg, April / May 2018, pp. 125–145.

[Duc22] Léo Ducas, *Estimating the hidden overheads in the bdgl lattice sieving algorithm*, Post-Quantum Cryptography (Cham) (Jung Hee Cheon and Thomas Johansson, eds.), Springer International Publishing, 2022, pp. 480–497.

[Duc23] Léo Ducas, *Provable lattice reduction of $\mathbb{Z}^n$ with blocksize $n/2$*, Cryptology ePrint Archive, Paper 2023/447, 2023, https://eprint.iacr.org/2023/447.

[DvW22] Léo Ducas and Wessel P. J. van Woerden, *On the lattice isomorphism problem, quadratic forms, remarkable lattices, and cryptography*, EUROCRYPT 2022, Part III (Orr Dunkelman and Stefan Dziembowski, eds.), LNCS, vol. 13277, Springer, Heidelberg, May / June 2022, pp. 643–673.

[EK20] Thomas Espitau and Paul Kirchner, *The nearest-colattice algorithm: time-approximation tradeoff for approx-CVP*, The Open Book Series **4** (2020), no. 1, 251–266, ANTS XIV.

[ETWY22] Thomas Espitau, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu, *Shorter hash-and-sign lattice-based signatures*, CRYPTO 2022, Part II (Yevgeniy Dodis and Thomas Shrimpton, eds.), LNCS, vol. 13508, Springer, Heidelberg, August 2022, pp. 245–275.

[FH23] Serge Fehr and Yu-Hsuan Huang, *On the quantum security of hawk*, Cryptology ePrint Archive, Paper 2023/711, 2023, https://eprint.iacr.org/2023/711.

[FIP15] *SHA-3 standard: Permutation-based hash and extendable-output functions*, National Institute of Standards and Technology, NIST FIPS PUB 202, U.S. Department of Commerce, August 2015.

[GHHM21] Alex B. Grilo, Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz, *Tight adaptive reprogramming in the QROM*, ASIACRYPT 2021, Part I (Mehdi Tibouchi and Huaxiong Wang, eds.), LNCS, vol. 13090, Springer, Heidelberg, December 2021, pp. 637–667.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan, *Trapdoors for hard lattices and new cryptographic constructions*, 40th ACM STOC (Richard E. Ladner and Cynthia Dwork, eds.), ACM Press, May 2008, pp. 197–206.

[GS02] Craig Gentry and Michael Szydlo, *Cryptanalysis of the revised NTRU signature scheme*, EUROCRYPT 2002 (Lars R. Knudsen, ed.), LNCS, vol. 2332, Springer, Heidelberg, April / May 2002, pp. 299–320.

[HK17] Gottfried Herold and Elena Kirshanova, *Improved algorithms for the approximate k-list problem in euclidean norm*, PKC 2017, Part I (Serge Fehr, ed.), LNCS, vol. 10174, Springer, Heidelberg, March 2017, pp. 16–40.

[Kec11] *Keccak implementation overview*, Submitted to NIST's SHA-3 competition, September 2011, https://keccak.team/obsolete/Keccak-implementation-3.1.pdf.

[KO62] Anatoly Karatsuba and Yuri Ofman, *Multiplication of many-digital numbers by automatic computers*, Proceedings of the USSR Academy of Sciences **145** (1962), no. 2, 293–294.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai, *CRYSTALS-DILITHIUM*, Tech. report, National Institute of Standards and Technology, 2022, available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[MR04] Daniele Micciancio and Oded Regev, *Worst-case to average-case reductions based on Gaussian measures*, 45th FOCS, IEEE Computer Society Press, October 2004, pp. 372–381.

[MW16] Daniele Micciancio and Michael Walter, *Practical, predictable lattice basis reduction*, EUROCRYPT 2016, Part I (Marc Fischlin and Jean-Sébastien Coron, eds.), LNCS, vol. 9665, Springer, Heidelberg, May 2016, pp. 820–849.

[NR09] Phong Q. Nguyen and Oded Regev, *Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures*, Journal of Cryptology **22** (2009), no. 2, 139–160.

[NV08] Phong Q. Nguyen and Thomas Vidick, *Sieve algorithms for the shortest vector problem are practical*, J. Mathematical Cryptology **2** (2008), no. 2, 181–207.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang, *FALCON*, Tech. report, National Institute of Standards and Technology, 2022, available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Por23] Thomas Pornin, *Improved key pair generation for falcon, BAT and hawk*, Cryptology ePrint Archive, Report 2023/290, 2023, https://eprint.iacr.org/2023/290.

[PP19] Thomas Pornin and Thomas Prest, *More efficient algorithms for the NTRU key generation using the field norm*, PKC 2019, Part II (Dongdai Lin and Kazue Sako, eds.), LNCS, vol. 11443, Springer, Heidelberg, April 2019, pp. 504–533.

[Pre17] Thomas Prest, *Sharper bounds in lattice-based cryptography using the Rényi divergence*, ASIACRYPT 2017, Part I (Tsuyoshi Takagi and Thomas Peyrin, eds.), LNCS, vol. 10624, Springer, Heidelberg, December 2017, pp. 347–374.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding, *CRYSTALS-KYBER*, Tech. report, National Institute of Standards and Technology, 2022, available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Szy03] Michael Szydlo, *Hypercubic lattice reduction and analysis of GGH and NTRU signatures*, EUROCRYPT 2003 (Eli Biham, ed.), LNCS, vol. 2656, Springer, Heidelberg, May 2003, pp. 433–448.

[Zha15] Mark Zhandry, *A note on the quantum collision and set equality problems*, Quantum Information and Computation **15** (2015), no. 7-8, 557–567.