

Kettle: Short Post-Quantum Threshold Signatures from the HAWK Signature Scheme

Calvin Abou Haidar¹ , Daniel Escudero² , Thomas Espitau³ , Clément Hoffmann¹ ,
Kaoru Takemure^{4,5} , Mehdi Tibouchi¹ , Hernán Darío Vanegas Madrigal⁶ 

¹NTT Social Informatics Laboratories, Tokyo, Japan ²TACEO, Graz, Austria ³PQShield, Paris, France

⁴PQShield, Tokyo, Japan ⁵National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

⁶HashCloak Inc., Ontario, Canada

Abstract

HAWK is a lattice-based hash-and-sign signature scheme that was a third round candidate in the NIST additional call for post-quantum signatures. It has short signatures and keys, as well as fast and portable signing and verification. However, in response to a highly-publicized AI-driven cryptanalytic result that reduced its security (roughly doubling the required dimension to achieve a given security level), the authors decided to withdraw from the competition. The need to increase parameters was seen as limiting the size advantage of HAWK compared to the already selected lattice schemes ML-DSA and Falcon, which was one of its primary selling points alongside speed and the lack of floating-point arithmetic.

In this paper, we identify another attractive aspect of the HAWK design that is not affected by the AI attack and has been overlooked so far: its surprising friendliness to multi-party computation (MPC). Indeed, HAWK signing can be achieved in a generic way from a small collection of basic composable MPC functionalities (like integer multiplication, uniform random generation, etc.) in such a way that instantiating those functionalities using MPC protocols with given security properties (semi-honest vs. malicious, honest vs. dishonest majority, etc.) yields a secure protocol for MPC signing with those same properties.

As a result, we obtain Kettle, a UC-secure threshold signature protocol that outputs specification-compliant HAWK signatures and scales to an arbitrary number of parties. It offers essentially the shortest signature size so far for a lattice-based threshold signature, while achieving significantly lower round complexity and online communication cost than other threshold protocols built from pre-existing, non-threshold lattice-based signatures (like ML-DSA and Falcon): as low as 2 online rounds (a dozen rounds total) and 2 kB online communication per party, depending on the specific setting and optimization choices.

We also provide a proof-of-concept implementation of our approach in the MP-SPDZ framework, which can be instantiated with any number of parties, any threshold, and a variety of security properties.

1 Introduction

Threshold signatures are protocols that make it possible to distribute the ability of signing messages among a collection of users, in such a way that at least t users need to cooperate to sign a message, and fewer of them cannot sign even if they collude and fail to follow the protocol. Such protocols are particularly useful to distribute trust, eliminate single points of failure and mitigate the risks arising from key compromise. Accordingly, they are often regarded as a necessity to securely deploy sensitive real-world systems involving, e.g., cloud key management, HSMs, or blockchain wallets and smart contracts. This has motivated NIST to launch a call [11] for *multi-party threshold schemes* (MPTS), which has in turn spurred a flurry of renewed research activity in the field.

While most threshold signature schemes proposed so far are based on the hardness of discrete logarithms [4, 6, 16, 20, 32] (and related to ECDSA, Schnorr signatures or pairing-based schemes like BLS), recent work has partially shifted towards post-quantum secure protocols, most of them based on lattice assumptions. These can be roughly divided into two types: “ad hoc” protocols on the one hand, that are designed from the ground up as threshold schemes and optimized as such; and on the other hand protocols that thresholdize a pre-existing signature scheme, making it possible to generate in a distributed way signatures that pass a pre-existing verification algorithm. The former type, which includes constructions like Pelican [25], TRaccoon [37], EKT [24], CTZ [13], Ringtail [10] and Olingo [28], tends to lead to more efficient protocols, with fewer rounds and often less communication, since they have the freedom to choose “multi-party computation (MPC)-friendly” operations as part of their signature generation algorithm; but they also tend to suffer from larger signature size, and of course have the drawback of being incompatible with pre-existing deployments (which may furthermore be constrained by interoperability requirements). In contrast, the latter type, which includes thresholdized ML-DSA [36] protocols Quorus [8] and Mithril [12] as well as

the recently proposed Threshold FALCON [26], needs to adapt an existing scheme to the MPC setting, which tends to require substantial compromises in terms of round and communication complexity, but the resulting signatures correspond to an existing scheme, so they tend to be comparatively short and are interoperable for free.

In this paper, we present an instance of the latter approach. We thresholdize the HAWK signature scheme, a lattice-based hash-and-sign construction that had advanced to the third round of the NIST additional call for post-quantum signatures. It has short signatures, and has been praised for being fast as well as easy to implement (not requiring floating point arithmetic or overly complex computations). However, a very recent, high-profile cryptanalytic result attributed to the Anthropic Mythos model [39] has reduced the conjectured hardness of the underlying lattice problem (lattice isomorphism for rank 2 modules over power-of-two cyclotomic rings), requiring twice as large of a dimension to achieve the same security level (so that HAWK-1024, originally expected to reach the 256-bit security level, actually offers 128 bits of security, while a new HAWK-2048 parameter set would be needed for 256-bit security). Even taking this new attack into account, HAWK remains surprisingly competitive, with signatures half as long as ML-DSA while avoiding FALCON’s floating point arithmetic, and achieving faster signing than both. Nevertheless, the scheme can no longer be said to hold a size advantage over FALCON (a condition set by NIST for considering a third lattice-based signature standard), and the authors decided to withdraw from the competition as a result. From the standpoint of MPC, however, we argue that HAWK remains a highly compelling scheme.

We present Kettle,¹ a threshold protocol for HAWK that inherits its short signature size and its simple and efficient verification, while offering a flexible design combining simple MPC functionalities that can be instantiated to achieve a wide range of security properties (semi-honest vs. malicious, honest vs. dishonest majority, etc.), and scales gracefully to an arbitrary number of parties. Kettle also offers an attractive performance profile, particularly in terms of online round and communication complexity, with as few as 2 online rounds and only 2 kB online communication (at the 128-bit security level with HAWK-1024) depending on the specific setting and optimization choices. This compares favorably not only to other threshold signature protocols for pre-existing lattice-based signatures, but even to many purpose-built schemes. An overview of the MPC techniques used to achieve this and build the Kettle protocol is provided later in this introduction (Section 1.2).

Threshold signatures are useful whenever signing authority should not be concentrated in a single device, administrator, or organization: custody of blockchain assets, HSM-backed key management, certification-authority and code-signing keys,

and organizational separation of duties are all natural deployment targets. We expand on these application domains in Section A.

1.1 Related Work

Lattice signatures and multiparty computation. Most practical lattice-based signatures belong broadly to one of two families. The first consists of hash-and-sign constructions in the GPV framework of Gentry, Peikert and Vaikuntanathan [27], including Falcon/FN-DSA. Signing in a typical GPV-style scheme requires sampling a discrete Gaussian over a lattice coset determined by the message and the secret trapdoor. This operation is already among the most intricate parts of a conventional implementation: it may involve recursive lattice decompositions, high-precision or floating-point arithmetic, and careful control of the output distribution. Carrying it out on secret-shared data is considerably more difficult. In particular, generic MPC techniques do not provide an obvious efficient realization of lattice Gaussian sampling, and translating the usual samplers into arithmetic or Boolean circuits results in complex, highly interactive computations.

The second family follows Lyubashevsky’s Fiat–Shamir-with-aborts paradigm [34,35], as instantiated by schemes such as BLISS [21] and Dilithium/ML-DSA [36]. These schemes avoid trapdoor sampling, but use rejection sampling to make the distribution of the response statistically independent of the signing key. Rejection is particularly inconvenient in a multiparty protocol: whenever an attempt is rejected, the parties must discard the computation and restart, including any message-dependent online interaction. Moreover, determining whether an attempt should be accepted may itself require expensive secure operations, such as norm tests, decomposition and rounding, or, in Gaussian-based schemes such as BLISS, comparisons involving secret discrete-Gaussian samples. Thus, neither of the two dominant approaches to lattice signatures translates directly into an efficient MPC signing protocol.

Purpose-built threshold lattice signatures. These difficulties motivated a line of constructions designed from the outset for distributed signing. Many of them resemble Fiat–Shamir-with-aborts signatures algebraically, but replace rejection sampling with *noise flooding*: sufficiently large additional noise statistically hides the contribution of the signing key, eliminating the need to restart the protocol. This approach underlies TRaccoon [37] and related constructions including EKT [24], Ringtail [10] and Olingo [28]. Because their signing algorithms can be chosen to consist largely of linear operations and other MPC-friendly primitives, these schemes achieve low round complexity, in some cases only two signing rounds and a single online round. They can also support features such as identifiable aborts or robust distributed key generation.

The price of avoiding rejection is that the flooding noise must be substantially larger than the masking noise used in

¹“kettle, n.: a usually large group of raptors (such as hawks or vultures) circling high in the sky on an updraft of warm air” (Merriam–Webster).

Table 1: Comparison of lattice-based threshold schemes. The generation of correlated randomness is excluded from the computations. We compare schemes for $N = 1024$ signers. When these numbers are not available, we show the estimates for the largest number of users given by the authors. Numbers for this work correspond to Kettle instantiated with HAWK-1024 (128-bit security level in light of [39]), in the honest-majority setting; upgrading to malicious security adds one online round and negligible communication (Section 5.2).

Scheme	Offline Rounds	Offline Com. Cost	Online Rounds	Online Com. Cost	Final Sig. Size	No. of Signers
Kettle (this work)	10	36.6 MB	2	1.9 kB	1221 B	1024
Threshold FALCON [26]	0		<i>thousands</i>	800.3 MB	666 B	16
Quorus (Threshold MLDSA) [8]	108	330.1 MB	16	1.44 MB	2.4 kB	63
Mithril (Threshold MLDSA) [12]	2	32 B	4	1.05 MB	2.4 kB	6
Pelican [25]	0		4	$(53.8 + 116N)$ kB	26.4 kB	1024
TRaccoon [37]	1	32 B	2	28.2 kB	12.7 kB	1024
Ringtail [10]	1	$(602.0 + 16N)$ kB	1	10.7 kB	13.7 kB	1024
EKT [24]	1	282.3 kB	1	14.4 kB	10.8 kB	1024
Olingo [28]	0		3	852.0 kB	11.3 kB	1024

ordinary Fiat–Shamir-with-aborts signatures. The resulting responses are therefore harder to compress, and signatures are typically much larger than those of ML-DSA and its predecessors. Furthermore, these constructions define new verification algorithms and signature formats. They consequently do not provide interoperability with existing standards or deployments, even when their algebraic structure is closely related to that of standardized lattice signatures.

Threshold ML-DSA. A different line of work attempts to execute the signing algorithm of an existing Fiat–Shamir-with-aborts scheme in MPC. This requires addressing not only rejection sampling, but also several nonlinear operations in ML-DSA, including high-bit extraction, hint generation, modular reduction, norm checks, and the evaluation of the rejection conditions on shared values. Quorus [8] gives a general MPC-based thresholdization of ML-DSA that supports arbitrary numbers of parties and general threshold access structures. Its modularity and scalability come at a relatively high interaction cost: its concrete signing protocol requires 16 online communication rounds, before accounting for repetitions caused by rejected attempts.

Mithril [12] takes a substantially different approach. Rather than evaluating the entire ML-DSA signing algorithm as a generic shared computation, it has the parties work with shares of the signing key and prove that their local contributions were formed correctly. A direct application of conventional secret sharing is problematic because ordinary shares of a short lattice secret need not themselves be short, whereas the relevant lattice proofs and rejection arguments require short witnesses. Mithril resolves this issue using replicated secret sharing, so that the values appearing as witnesses remain suitably structured. This yields a comparatively efficient protocol, requiring three rounds per signing attempt, but restricts the construction to small committees, with concrete instantiations supporting

at most six parties. Moreover, its output distribution is not that of ordinary ML-DSA signing: although the signatures are accepted by the ML-DSA verification algorithm, threshold-generated signatures can be distinguished from signatures produced by the standard signer. Thus, Mithril provides verification compatibility, but not a distribution-preserving threshold realization of ML-DSA.

Threshold hash-and-sign signatures. Distributing a GPV trapdoor was in fact one of the earliest approaches to lattice-based threshold signatures. Bendlin, Krehbiel and Peikert [7] constructed threshold signatures and identity-based encryption from a shared lattice trapdoor. Its signing protocol, however, assumes access to a black-box MPC functionality that samples a discrete Gaussian over an arbitrary lattice coset. No concretely efficient realization of this functionality was provided, and implementing it appears to require essentially all the difficult numerical and recursive operations of GPV sampling inside MPC. The construction is therefore principally a feasibility result rather than a practical threshold-signing protocol.

Very recently, Threshold FALCON [26] provided the first concrete thresholdization of FALCON/FN-DSA. It shows that general-lattice Gaussian sampling can indeed be implemented securely in MPC, but at a very high interaction cost: the protocol requires many thousands of rounds for an individual execution, with batching and amortization reducing the reported cost to roughly one thousand online rounds per signature when many signatures are generated together. This illustrates the central obstacle faced by GPV-style schemes: their compact signatures are obtained through a sampling procedure that is exceptionally difficult to reproduce on shared secrets.

Our work Kettle gives the first efficient threshold instantiation of a lattice-based hash-and-sign signature. The key

enabling feature is specific to HAWK. Unlike conventional GPV schemes, HAWK does not sample a Gaussian over a general secret-dependent lattice coset. It samples each coordinate independently from a one-dimensional discrete Gaussian over $2\mathbb{Z}$ or $2\mathbb{Z} + 1$, with the choice of coset determined by secret bits. Sampling over $\mathbb{Z}$, selecting between the two distributions, and performing the remaining polynomial operations can all be expressed using standard MPC primitives. Kettle therefore retains the short, specification-compliant signatures of HAWK while avoiding both the rejection-heavy computation of threshold ML-DSA and the general-lattice Gaussian sampler responsible for the high round complexity of Threshold FALCON. See the comparison in Table 1. Technically, Kettle only has the second shortest signature size after Threshold FALCON in that table, but that protocol is so far from practical that we think it fair to claim “essentially the shortest” signature size among lattice-based threshold signatures.

1.2 Technical Overview

1.2.1 HAWK Signatures

At a high level, the HAWK signing algorithm proceeds as follows, where $\mathbf{B}$, f and g are all part of the signing key.

1. **Hashing:** Compute a public hash of the message m to be signed, a binary vector $\mathbf{h} \leftarrow H(m)$.
2. **Center Computation:** Compute the center $\mathbf{t} \leftarrow \mathbf{B} \cdot \mathbf{h} \pmod{2}$ (again a binary vector).
3. **Gaussian Sampling:** Sample a Gaussian vector $\mathbf{x}$ coordinate-wise from $\mathcal{D}^{(t_0)} \times \mathcal{D}^{(t_1)} \times \dots$, where $\mathcal{D}^{(b)}$ is a discrete Gaussian on the coset $2\mathbb{Z} + b$ centered at 0. That is, the i -th coefficient of $\mathbf{x}$ is distributed following a Gaussian supported over $2\mathbb{Z} + t_i$.
4. **Final Multiplication:** Let $\mathbf{x}$ be split into two halves $\mathbf{x} = (\mathbf{x}_0 \parallel \mathbf{x}_1)$, and interpret each $\mathbf{x}_i \in \mathbb{Z}_q^n$ as a ring element in a certain cyclotomic ring $\mathcal{R}$. Compute $w_1 \leftarrow f \cdot x_1 - g \cdot x_0$ in the ring $\mathcal{R}$. The signature is then given by $s \leftarrow (h_1 - w_1)/2$ (up to a choice of sign and a rare rejection step that do not matter for our purposes).

1.2.2 Naive Threshold Signing

The basic idea of the Kettle protocol is the following. We start with a secret-shared secret key $[\mathbf{B}]$. Since the message m is public, the parties can locally compute $[\mathbf{t}'] = [\mathbf{B}] \cdot \mathbf{h}$. Now, we actually need to obtain $[\mathbf{t}]$, where $\mathbf{t} = \mathbf{t}' \pmod{2}$. If we use a power of two as the modulus, the parties can locally compute $[\mathbf{t}]_2$ (over the binary domain) by locally reducing their shares modulo 2, but this still falls short, since we need $[\mathbf{t}]$ (over the arithmetic domain) for the computation that follows next.

To address this we employ *daBits* [38], which work as follows. The parties preprocess a series of random bits $\mathbf{r} \in$

$\{0, 1\}^{2n}$ both as $[\mathbf{r}]_2$ (binary sharings) and $[\mathbf{r}]$ (arithmetic sharings). The parties then compute locally and open $\mathbf{c} \leftarrow [\mathbf{t}]_2 \oplus [\mathbf{r}]_2$, followed by $[\mathbf{t}] \leftarrow \mathbf{c} + [\mathbf{r}] - 2 \cdot \mathbf{c} \odot [\mathbf{r}]$ (where $\oplus$ is the coordinate-wise XOR operation on bits, and $\odot$ denotes the Hadamard product of integer vectors, i.e., coordinate-wise multiplication).

With $[\mathbf{t}]$ in hand, the next step is to do the Gaussian sampling. Assume that, from a preprocessing to be specified later, the parties obtain shares of discrete Gaussian vectors $[\mathbf{x}^{(0)}]$ and $[\mathbf{x}^{(1)}]$ such that the coefficients of $\mathbf{x}^{(0)}$ are sampled from $\mathcal{D}^{(0)}$ (supported over $2\mathbb{Z}$) and those of $[\mathbf{x}^{(1)}]$ from $\mathcal{D}^{(1)}$ (supported over $2\mathbb{Z} + 1$). They can multiplex these using $\mathbf{t}$ as

$$[\mathbf{x}] = [\mathbf{x}^{(0)}] + [\mathbf{t}] \odot [\Delta\mathbf{x}],$$

where $\Delta\mathbf{x} = \mathbf{x}^{(1)} - \mathbf{x}^{(0)}$.

The above requires the parties to compute the secure product $[\mathbf{t}] \odot [\Delta\mathbf{x}]$, which could be done with any secure multiplication protocol. What follows then is splitting into two halves $[\mathbf{x}] = ([\mathbf{x}_0] \parallel [\mathbf{x}_1])$, interpreting $([\mathbf{x}_0], [\mathbf{x}_1])$ as *polynomials* $([\mathbf{x}_0], [\mathbf{x}_1])$ over $\mathcal{R}$, and computing $[w] = [f] \cdot [\mathbf{x}_1] - [g] \cdot [\mathbf{x}_0]$. Once again, this requires secure multiplication. After this, the signature can be reconstructed with local computation only.

1.2.3 Optimizing the Naive Baseline

The naive online phase has two costly secure-multiplication steps: the Hadamard product $[\mathbf{t}] \odot [\Delta\mathbf{x}]$ and the polynomial product $[f] \cdot [\mathbf{x}_1] - [g] \cdot [\mathbf{x}_0]$ over $\mathcal{R}$. Neither of them genuinely depends on the message: the only message-dependent quantity in the whole online phase is the public binary vector $\mathbf{c}$ produced by the daBit opening. The optimization below moves *all* secret multiplications into the preprocessing, leaving the online phase with two openings and a single public-coefficient linear combination of stored shares.

Linearizing the final polynomial. Let $\text{rot}(h) \in \mathbb{Z}_q^{n \times n}$ denote the matrix of multiplication by $h \in \mathcal{R}$ in the coefficient basis (see 2.1 for the explicit form). Then the HAWK final polynomial $w = f \cdot x_1 - g \cdot x_0$ has coefficient vector $\mathbf{w} = \text{rot}(f) \cdot \mathbf{x}_1 - \text{rot}(g) \cdot \mathbf{x}_0$, which we write compactly as $\mathbf{w} = \mathbf{M} \cdot \mathbf{x}$ with $\mathbf{x} = (\mathbf{x}_0 \parallel \mathbf{x}_1)$ and

$$\mathbf{M} := [-\text{rot}(g) \mid \text{rot}(f)] \in \mathbb{Z}_q^{n \times 2n}.$$

Since each entry of $\mathbf{M}$ is a signed copy of one coefficient of f or g , we have $[\mathbf{M}]$ “for free” from $[f], [g]$.

Absorbing the multiplex into the mask. Substituting the daBit identity $\mathbf{t} = \mathbf{c} + (\mathbf{1} - 2\mathbf{c}) \odot \mathbf{r}$ into the multiplex $\mathbf{x} = \mathbf{x}^{(0)} + \mathbf{t} \odot \Delta\mathbf{x}$ and rearranging, we have

$$\mathbf{x} = \underbrace{\mathbf{x}^{(0)} + \mathbf{r} \odot \Delta\mathbf{x}}_{:=\mathbf{s}_0} + \underbrace{\mathbf{c} \odot (\mathbf{1} - 2\mathbf{r}) \odot \Delta\mathbf{x}}_{:=\mathbf{s}_1}.$$

The split $\mathbf{x} = \mathbf{s}_0 + \mathbf{c} \odot \mathbf{s}_1$ separates the secret preprocessed randomness $(\mathbf{s}_0, \mathbf{s}_1)$, derived from $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{r}$ from the public binary coefficients $(\mathbf{c}, \text{opened online})$.

Packaging the preprocessing. Composing the two steps,

$$\mathbf{w} = \mathbf{M} \cdot \mathbf{x} = \underbrace{\mathbf{M} \cdot \mathbf{s}_0}_{:= \mathbf{w}_0} + \mathbf{M} \cdot (\mathbf{c} \odot \mathbf{s}_1).$$

The first summand $\mathbf{w}_0$ is fully determined by the preprocessing. The second summand depends on $\mathbf{c}$ but only *linearly*: expanding $(\mathbf{c} \odot \mathbf{s}_1)_i = c_i \cdot (\mathbf{s}_1)_i$, its j -th coordinate is

$$(\mathbf{M} \cdot (\mathbf{c} \odot \mathbf{s}_1))_j = \sum_{i=1}^{2n} \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i \cdot c_i = (\mathbf{V} \cdot \mathbf{c})_j,$$

where $\mathbf{V}_{j,i} := \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i$. Both $\mathbf{w}_0$ and $\mathbf{V}$ are products of secret-key shares and preprocessed randomness; given them, the entire variable part of $\mathbf{w}$ is the linear combination $\mathbf{V} \cdot \mathbf{c}$ with the public coefficients $\mathbf{c}$, computed locally on stored shares.

The optimized online phase. Assuming the preprocessing has stored $([\mathbf{w}_0], [\mathbf{V}], [\mathbf{r}]_2)$, signing reduces to

1. one opening to publish $\mathbf{c} \leftarrow \text{open}([\mathbf{r}]_2 \oplus [\mathbf{r}]_2) \in \mathbb{F}_2^{2n}$;
2. the local computation $[\mathbf{w}] \leftarrow [\mathbf{w}_0] + [\mathbf{V}] \cdot \mathbf{c}$, using only public coefficients $\mathbf{c}$ and stored shares;
3. one opening to recover $\mathbf{w}$, and follow the final *public* steps of HAWK signing

No secret multiplication takes place online. The preprocessing itself still grows quadratically in n (it computes $\Theta(n^2)$ secret products to assemble $[\mathbf{w}_0]$ and $[\mathbf{V}]$); we propose an alternative scheme in Section 5.1 with linear-communication preprocessing, at the cost of one extra online round.

Remark 1 (Preprocessing the salt). In HAWK there is a public salt that gets hashed alongside the message. We treat this as preprocessing, which means an adversary can choose the message after seeing the salt, and hence requires an extra security argument. This is similar to what is done in one-round Threshold MAYO [3]. We discuss this in Section 4.1.

1.2.4 Preprocessing the Gaussians

In the online phase above we assumed that from preprocessing the parties have Gaussian samples $[\mathbf{x}^{(0)}]$ from $\mathcal{D}^{(0)}$ (supported over $2\mathbb{Z}$) and $[\mathbf{x}^{(1)}]$ from $\mathcal{D}^{(1)}$ (supported over $2\mathbb{Z} + 1$). To do this, we directly adapt the discrete Gaussian sampling algorithm from HAWK itself to the multi-party setting.

More precisely, discrete Gaussian sampling in HAWK is carried out by so-called reverse cumulative distribution table (RCDT) sampling. For each of the two distributions $\mathcal{D}^{(b)}$, $b = 0, 1$, one tabulates the probabilities $p_i^{(b)} = \Pr_{x \sim \mathcal{D}^{(b)}}[x \geq i]$ with sufficiently many bits of precision L (in the case of HAWK,

it is shown that $L = 78$ suffices): these are the RCDTs. Then, one can sample from $\mathcal{D}^{(b)}$ by sampling a random integer u of L bits, comparing it to every value $p_i^{(b)}$ in the corresponding RCDT, and counting the number of indices i such that $u \leq p_i^{(b)}$.

In Kettle, we proceed similarly: to generate a shared Gaussian sample from $\mathcal{D}^{(0)}$, say, the parties first sample Boolean shares $[u_i]_2$ of the bits of the L -bit value u (by locally generating uniform bits), and apply a Boolean comparison circuit to these bits to obtain the shared bit $[b_i]_2 = [u \leq p_i^{(0)}]_2$ for each i (i.e., $b_i = 1$ if and only if $u \leq p_i^{(0)}$). By using a comparison circuit of logarithmic depth and linear size such as the one arising from the Brent–Kung adder circuit, this can typically be done in roughly $\log_2 L$ rounds and a total of $2L$ multi-party bit multiplications (i.e., multi-party AND gates). The number of indices i such that $u \leq p_i^{(0)}$, from which the Gaussian sample is easily deduced, can then be obtained by converting these Boolean shares to arithmetic shares $[b_i]$ using daBit openings, and summing them together (which is a local computation).

This description glosses over some minor technical details. First, since $\mathcal{D}^{(b)}$ is supported over $2\mathbb{Z} + b$, a sample $x \sim \mathcal{D}^{(b)}$ can be computed as $x \leftarrow 2y + b$ where y is sampled according to a discrete Gaussian over $\mathbb{Z}$ of half the standard deviation, centered at $-b/2$. This is the approach taken in HAWK (which halves the size of the RCDTs and the number of comparisons) and we can clearly do the same in our setting, as the affine map $y \mapsto 2y - b$ can be computed on a shared value with only local computations. Moreover, it is observed in HAWK that the distribution of x is symmetric around 0, so one can simply sample its absolute value $|x|$ using the same RCDT approach, and randomly flip its sign. This can also be done in our setting, at the cost of one extra offline round for the multiplication by ± 1 , but further halving the size of the RCDT, and hence the communication cost of the comparisons. Alternatively, one can use the full size RCDTs to save an offline round, but increasing the communication.

Incidentally, additional tricks exist that make it possible to reduce the round complexity of the comparisons themselves at the cost of increased communication. In general, regarding the *offline* phase, we prefer lower communication rather than a lower number of rounds, since that phase can be amortized by carrying out the preprocessing for many signatures in advance, so we omit those tricks in our main protocol.

Remark 2 (Rejection of large Gaussian vectors). In standard HAWK, the signature algorithm restarts if the length of the Gaussian vector $\mathbf{x}$ is too large, in the sense that it exceeds the verification bound. Since we simply omit this check in our protocol, signatures have a probability of being invalid. According to [9], the probability that this check triggers for the HAWK-1024 parameters is less than 2^{-121} . Thus our protocol produces an invalid signature with a negligible probability, which does not affect the unforgeability of the scheme.

1.2.5 Preprocessing the Linearization

It remains to explain how the parties precompute, in the $\mathcal{F}_{\text{ABB}}$ -hybrid model, the secret-shared objects $\llbracket \mathbf{w}_0 \rrbracket \in \mathbb{Z}_q^n$ and $\llbracket \mathbf{V} \rrbracket \in \mathbb{Z}_q^{n \times 2n}$ from $\llbracket f \rrbracket, \llbracket g \rrbracket$ and the preprocessed Gaussians $\llbracket \mathbf{x}^{(0)} \rrbracket, \llbracket \mathbf{x}^{(1)} \rrbracket$, together with the daBit mask $(\llbracket \mathbf{r} \rrbracket, \llbracket \mathbf{r} \rrbracket_2)$. The bulk of the cost is $\Theta(n^2)$ secret multiplications: the two halves of $\llbracket \mathbf{w}_0 \rrbracket = \mathbf{M} \cdot \mathbf{s}_0$ are two polynomial products $\llbracket f \rrbracket \cdot \llbracket s_{0,1} \rrbracket$ and $\llbracket g \rrbracket \cdot \llbracket s_{0,0} \rrbracket$ over $\mathcal{R}$, and the $n \times 2n$ matrix $\llbracket \mathbf{V} \rrbracket$ has $2n^2$ entries, each a single secret product $\llbracket \mathbf{M}_{j,i} \rrbracket \cdot \llbracket (\mathbf{s}_1)_i \rrbracket$.

At a high level, Π_{Prep} instantiates exactly the algebra described above. It first samples the public salt and uses the MPC Gaussian sampler to obtain shared vectors $\llbracket \mathbf{x}^{(0)} \rrbracket$ and $\llbracket \mathbf{x}^{(1)} \rrbracket$ for the two possible cosets. It then generates a daBit mask $(\llbracket \mathbf{r} \rrbracket, \llbracket \mathbf{r} \rrbracket_2)$, computes $\llbracket \Delta \mathbf{x} \rrbracket$, and locally/multiplicatively forms the two affine preprocessing vectors

$$\llbracket \mathbf{s}_0 \rrbracket = \llbracket \mathbf{x}^{(0)} \rrbracket + \llbracket \mathbf{r} \rrbracket \odot \llbracket \Delta \mathbf{x} \rrbracket, \quad \llbracket \mathbf{s}_1 \rrbracket = (\mathbf{1} - 2\llbracket \mathbf{r} \rrbracket) \odot \llbracket \Delta \mathbf{x} \rrbracket.$$

Finally, using the shared key polynomials $\llbracket f \rrbracket, \llbracket g \rrbracket$, the parties compute the two polynomial products defining $\llbracket \mathbf{w}_0 \rrbracket = \llbracket \mathbf{M} \mathbf{s}_0 \rrbracket$ and the entrywise products $\llbracket \mathbf{V}_{j,i} \rrbracket = \llbracket \mathbf{M}_{j,i} \rrbracket \cdot \llbracket (\mathbf{s}_1)_i \rrbracket$ for all $(j, i) \in [n] \times [2n]$. The output of preprocessing is therefore the tuple $(\text{salt}, \llbracket \mathbf{w}_0 \rrbracket, \llbracket \mathbf{V} \rrbracket, \llbracket \mathbf{r} \rrbracket_2)$ consumed by the two-round online protocol. We give the full protocol, including the Gaussian sampler and its security analysis, in Section 3.

2 Preliminaries

2.1 The Polynomial Ring $\mathcal{R}$ and Coefficient Matrices

Throughout the paper, n is a power of two and q a prime number. We let

$$\mathcal{R} := \mathbb{Z}_q[X] / \langle X^n + 1 \rangle$$

denote the underlying polynomial ring of HAWK,² and identify $\mathcal{R}$ with $\mathbb{Z}_q^n$ via the natural $\mathbb{Z}_q$ -linear isomorphism $\pi : \mathcal{R} \rightarrow \mathbb{Z}_q^n$ that reads off coefficients, namely $\pi(a_0 + a_1X + \dots + a_{n-1}X^{n-1}) = (a_0, a_1, \dots, a_{n-1})$. We freely identify a polynomial $a \in \mathcal{R}$ with its coefficient vector $\pi(a) \in \mathbb{Z}_q^n$, using bold notation $\mathbf{a}$ when the vector representation is more convenient and the same letter without bold for the polynomial representation.

For every $h \in \mathcal{R}$, multiplication by h is a $\mathbb{Z}_q$ -linear endomorphism of $\mathcal{R}$. We denote its matrix in the coefficient basis by $\text{rot}(h) \in \mathbb{Z}_q^{n \times n}$, which satisfies $\text{rot}(h) \cdot \pi(a) = \pi(h \cdot a)$ for every $a \in \mathcal{R}$. Concretely, when $\mathcal{R} = \mathbb{Z}_q[X] / \langle X^n + 1 \rangle$, the matrix $\text{rot}(h)$ is the *negacyclic Toeplitz* matrix of h : writing $\pi(h) = (h_0, h_1, \dots, h_{n-1})$, its entry in row j and column i is

²While the standard HAWK scheme operates over $\mathbb{Z}[X] / \langle X^n + 1 \rangle$, we use $\mathcal{R}$ to handle secret sharing. The modulus q is chosen to be sufficiently large to prevent overflows during computation, similar to the NTT modulus used in HAWK.

$\text{rot}(h)_{j,i} = h_{j-i}$ when $j \geq i$ and $\text{rot}(h)_{j,i} = -h_{n+j-i}$ when $j < i$, so each entry of $\text{rot}(h)$ is a signed copy of exactly one coefficient of h . The map rot is a $\mathbb{Z}_q$ -algebra homomorphism: it satisfies $\text{rot}(\alpha h + \beta h') = \alpha \text{rot}(h) + \beta \text{rot}(h')$ and $\text{rot}(h \cdot h') = \text{rot}(h) \cdot \text{rot}(h')$, with $\text{rot}(1) = \mathbf{I}_n$.

2.2 Discrete Gaussian Distribution

For a parameter $\sigma > 0$, we define $\rho_\sigma(x) = \exp(-|x|^2 / 2\sigma^2) \in \mathbb{R}$. For any $c \in \mathbb{R}$, the discrete Gaussian distribution over $\mathbb{Z} + c$ with standard deviation σ , denoted by $\mathcal{D}_{\mathbb{Z}+c,\sigma}$, is defined by $\mathcal{D}_{\mathbb{Z}+c,\sigma}(x) = \rho_\sigma(x) / \sum_{z \in \mathbb{Z}+c} \rho_\sigma(z)$ for $x \in \mathbb{Z} + c$. In the signing algorithm of HAWK, we sample a vector $\mathbf{x} \in \mathbb{Z}^{2n}$ from the discrete Gaussian distribution over $2\mathbb{Z}^{2n} + \mathbf{t}$ for $\mathbf{t} \in \{0, 1\}^{2n}$ by sampling each coordinate x_i from $\mathcal{D}_{2\mathbb{Z}+t_i,\sigma}$, where t_i is the i -th entry of $\mathbf{t}$.

2.3 Digital Signature

A digital signature scheme consists of the following three algorithms (KeyGen, Sign, Verify). We suppose that all algorithms implicitly take a public parameter pp as input.

- **KeyGen()** $\rightarrow (\text{vk}, \text{sk})$: The key generation algorithm outputs a verification key vk and a signing key sk .
- **Sign(sk, m)** $\rightarrow \sigma$: The signing algorithm takes as input sk and a message m and outputs a signature σ .
- **Verify(vk, σ, m)** $\rightarrow b$: The verification algorithm takes as input vk , σ , and m and outputs 1 if the signature is valid, and 0 otherwise.

Looking ahead, our threshold signing protocol precomputes a salt salt in the offline phase before m is determined. To deal with this leakage, we consider an explicit-salt variant by adopting the approach used in [3]. Instead of generating a salt within Sign, this variant is given salt as input. We denote this modified signing algorithm by $\text{SignDet}(\text{sk}, m, \text{salt}) \rightarrow (\sigma, \text{aux})$. We additionally consider an auxiliary output aux to handle the additional leakage during our signing algorithm.

Strong Unforgeability. We briefly recall the definition of strong unforgeability under chosen message attack (SUF-CMA). The procedure of the security game is as follows: the challenger generates (vk, sk) by KeyGen and runs the adversary $\mathcal{A}$ by giving it vk . $\mathcal{A}$ can query m to the signing oracle upon which the challenger runs Sign on m and returns σ . The goal of $\mathcal{A}$ is to produce a valid non-trivial forgery. A non-trivial forgery means that the message-signature pair has never been produced by the signing oracle.

We consider a similar security game for the explicit-salt variant. The only difference is that $\mathcal{A}$ can query an adaptively chosen salt along with m to the signing oracle. The winning condition remains identical.

2.4 HAWK Signing Algorithm

We recall the HAWK signature scheme [9] in Algorithm 1, deferred to Section B.

For simplicity, we omit some encodings for the verification key and signatures. Since we consider the key generation by the trusted third party and encodings apply to aggregated signatures, it is easy to apply such encoding to our threshold signature without compromising the security. The key generation algorithm samples the secret key from the centred binomial distribution with parameter $\eta \in \mathbb{Z}_{\geq 1}$. Concretely, a value x is sampled from the support $\{-\eta, -\eta+1, \dots, \eta\}$ with probability $\binom{2\eta}{x+\eta}/2^{2\eta}$. HAWK uses SHAKE256 to instantiate the hash function H . Specifically, it uses H to compute a hash of the verification key $\text{hpub} = H(\text{vk})$, whose length depends on the security level, a message digest $M = H(m, \text{hpub}) \in \{0, 1\}^{512}$ where m is the message to be signed, and a vector $\mathbf{h} = H(M, \text{salt}) \in \{0, 1\}^{2n}$. We write ℓ for the bit length of the salt, whose value depends on the parameter set and for which we refer to [9]. For simplicity, we use the same notation H for both cases since the details of the instantiation are not essential for proving the security of our protocols. HAWK uses two subroutines $\text{NTRUSolve}(f, g, 1)$ and $\text{Decompress}(s_1, \mathbf{h}, \mathbf{Q})$. NTRUSolve is to find a somewhat short vector $(F, G) \in \mathcal{R}^2$ such that $fG - gF = 1$. Decompress recomputes s_0 , a part of the full signature, from s_1 , vk and $\mathbf{h}$, which can be computed from vk and m . We omit the details of these subroutines because they are not involved in our threshold signing protocol. We refer the reader to [9] for details on the mappings of H and those subroutines. The signing algorithm samples $\mathbf{x}$ using RCDT sampling for $\mathcal{D}_{2\mathbb{Z}^{2n} + \mathbf{t}, \sigma}$. For simplicity, we denote this by $\mathbf{x} \xleftarrow{\$} \mathcal{D}^{(t)}$. For $a \in \mathcal{R}$, $\text{sym-break}(a)$ returns true if $a \neq 0$ and the first non-zero coefficient of a is positive.

We also present an explicit-salt variant of HAWK, in which Sign is replaced with SignDet . Instead of sampling salt internally, this variant takes salt as input. Furthermore, it additionally outputs w_1 (computed in line 20 of Sign), which is the value before applying sym-break . The security of the HAWK scheme instantiated with SignDet is analyzed in Appendix F. We show that this variant is SUF-CMA if the standard HAWK is SUF-CMA .

2.5 Secure Multiparty Computation

2.5.1 UC Framework

We use the Universal Composability (UC) framework to state the security of our protocols. In this framework, security is defined by comparing a real-world execution of a protocol with an ideal-world execution in which the desired task is performed by a trusted functionality. A protocol Π UC-realizes an ideal functionality $\mathcal{F}$ if, for every real-world adversary $\mathcal{A}$, there exists an ideal-world simulator $\mathcal{S}$ such that no probabilistic polynomial-time environment $\mathcal{Z}$ can distinguish the

real execution from the ideal execution except with negligible probability. We consider static corruptions, where the adversary chooses the corrupted parties before the protocol begins. The main advantage of working in the UC framework is composability: protocols proven secure in an ideal-hybrid model remain secure when the ideal functionalities are replaced by concrete UC-secure implementations. Accordingly, our signing, preprocessing, Gaussian-sampling, and auxiliary MPC subprotocols are specified through ideal functionalities and then instantiated in the corresponding hybrid models.

2.5.2 Arithmetic Black-Box Model

Our threshold signing protocols are described in the *Arithmetic Black-Box* (ABB) model, a standard MPC abstraction that captures the ability of the parties to distribute, operate on, and reconstruct secret values without committing to any concrete secret-sharing scheme or adversarial setting. A value x stored in the ABB is denoted $\llbracket x \rrbracket$, in line with the standard secret-sharing notation that typically instantiates the model. We refer to the parties as having “shares” of x when x is stored in the ABB.

HAWK signing requires the parties to manipulate both arithmetic values and bits, so we equip the ABB with two storage domains: an *arithmetic* domain over $\mathbb{Z}_q$, where q is a prime, and a *binary* domain over $\mathbb{F}_2$. We write $\llbracket x \rrbracket$ for an arithmetic share of $x \in \mathbb{Z}_q$ and $\llbracket b \rrbracket_2$ for a binary share of $b \in \mathbb{F}_2$.

The basic ABB-style commands, parameterized by a domain $\text{dom} \in \{\text{arith}, \text{bin}\}$, are: input, add, addmult-cons, mult, open, rand, coin, and mod2.³ We do not introduce a stand-alone $\mathcal{F}_{\text{ABB}}$ functionality; instead, the basic commands appear as the foundation of each functionality we define.

Notation and conventions. We omit handle names when convenient and write, e.g., $\llbracket x + y \rrbracket \leftarrow \llbracket x \rrbracket + \llbracket y \rrbracket$ for an add call, $\llbracket r \rrbracket \leftarrow \text{rand}$ for a rand call, and $c \leftarrow \text{coin}$ for a coin call. We extend the notation component-wise to vectors, matrices, and polynomials in $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$ (viewed as length- n coefficient vectors): for a vector $\mathbf{x} \in \mathbb{Z}_q^n$ we write $\llbracket \mathbf{x} \rrbracket$ to mean that each coordinate is stored in the ABB, and similarly for matrices. We write $\mathbf{a} \odot \mathbf{b}$ for the Hadamard (component-wise) product and $\mathbf{A} \cdot \mathbf{b}$ for matrix-vector multiplication; both extend to shared values in the obvious way using the ABB commands above.

2.6 Some Functionalities

2.6.1 The Key-Generation Functionality $\mathcal{F}_{\text{KG}}$

The functionality $\mathcal{F}_{\text{KG}}$ extends the arithmetic black box of Section 2.5.2 with a one-shot (key-gen) command, which runs HAWK.KeyGen (Algorithm 1), stores the four secret-key polynomials f, g, F, G in the arithmetic domain, and sends the

³This mod 2 reduction is free when q is a power of two; otherwise it requires an edaBit and one comparison per value.

verification key $\text{vk} = (\mathbf{Q}, \text{hpub})$ to all parties. Of the ABB commands, only (open) and (coin) produce output visible to the parties; all others merely update the internal store. The full description is deferred to Section C. In this work we do not discuss how to instantiate this functionality. Doing so would require solving the NTRU equation in MPC, which seems to be a challenging problem, as also observed in [26].

2.6.2 The Extended ABB Functionality $\mathcal{F}_{\text{eABB}}$: daBits and LTC

For our threshold signing we will require two extra operations: doubly-authenticated bits (daBits), which consist of a secret pair $(\llbracket r \rrbracket, \llbracket r \rrbracket_2)$ where $r \in \{0, 1\}$, and a less-than comparison (LTC) command, which compares a secret bit-wise value $\llbracket u \rrbracket_2$ with a public constant c , storing the result as a bit $\llbracket b = u < c \rrbracket$. Both commands can be instantiated in the $\mathcal{F}_{\text{KG}}$ -hybrid model using standard techniques. daBits were introduced by Rotaru and Wood [38], and a more efficient variant known as edaBits is given in [23]; for instantiations whose arithmetic domain is $\mathbb{Z}_{2^k}$ we refer to [18]. We summarize both cases in Section D. The (ltc) command is realized by a standard Boolean comparison circuit, whose cost we account for in Section 5.4. The formal description of both commands is deferred to Section C.

3 Offline Phase

We begin by discussing the offline phase of our protocol.

3.1 The Preprocessing Functionality $\mathcal{F}_{\text{Prep}}$

The functionality $\mathcal{F}_{\text{Prep}}$ below captures the required preprocessing for our online phase, which in particular includes the generation of the public salt, but also produces the preprocessed values needed to compute the signature in the online phase, as outlined in the technical overview in Section 1.2.

Concretely, on a (prep) call the functionality fetches the stored key polynomials f, g , samples a fresh salt $\text{salt} \xleftarrow{\$} \{0, 1\}^\ell$, two Gaussian vectors $\mathbf{x}^{(0)} \xleftarrow{\$} (\mathcal{D}^{(0)})^{2n}$ and $\mathbf{x}^{(1)} \xleftarrow{\$} (\mathcal{D}^{(1)})^{2n}$, and a uniform mask $\mathbf{r} \xleftarrow{\$} \{0, 1\}^{2n}$; it forms $\mathbf{s}_0 = \mathbf{x}^{(0)} + \mathbf{r} \odot \Delta \mathbf{x}$ and $\mathbf{s}_1 = (\mathbf{1} - 2\mathbf{r}) \odot \Delta \mathbf{x}$ with $\Delta \mathbf{x} = \mathbf{x}^{(1)} - \mathbf{x}^{(0)}$, stores $\mathbf{w}_0 = \mathbf{M} \cdot \mathbf{s}_0$ and $\mathbf{V}_{j,i} = \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i$ under the supplied handles together with $\mathbf{r}$ in the binary domain, and sends salt to all parties. Here $\mathbf{M} := [-\text{rot}(g) \mid \text{rot}(f)] \in \mathbb{Z}_q^{n \times 2n}$, and $\mathcal{D}^{(0)}, \mathcal{D}^{(1)}$ denote the two coordinate-wise RCDT samplers used in HAWK (see Section 2.4). Each call samples fresh randomness and writes to the handles supplied by the caller, so distinct calls produce independent preprocessing and the functionality keeps no state of its own. The full description is deferred to Section C.

3.2 The Preprocessing Protocol Π_{Prep}

Now we instantiate Functionality $\mathcal{F}_{\text{Prep}}$ via Protocol Π_{Prep} below. This protocol is set in the $\mathcal{F}_{\text{eABB}}$ -hybrid model. The only command that must be instantiated is (prep). For the Gaussians we make use of procedure π_{Gaussian} , described later below in Section 3.3. For the salt, the (coin) command of the ABB is used. Finally, for the preprocessed products, the multiplication command of the ABB is invoked.

Protocol 1: Π_{Prep}

The protocol is set in the $\mathcal{F}_{\text{eABB}}$ -hybrid model.

Input. Arithmetic-domain handles id_f, id_g of a secret-shared signing key $(\llbracket f \rrbracket, \llbracket g \rrbracket)$, already populated in $\mathcal{F}_{\text{eABB}}$ by key-gen, together with fresh output handles $\text{id}_{\mathbf{w}_0}, \text{id}_{\mathbf{V}}, \text{id}_{\mathbf{r}}$.

Output. A public salt salt and three $\mathcal{F}_{\text{eABB}}$ -stored sharings $(\llbracket \mathbf{w}_0 \rrbracket, \llbracket \mathbf{V} \rrbracket, \llbracket \mathbf{r} \rrbracket_2)$ as in $\mathcal{F}_{\text{Prep}}$.

1. Parties call $\text{salt} \leftarrow \text{coin}(\{0, 1\}^\ell)$.
2. Parties call $(\llbracket \mathbf{x}^{(0)} \rrbracket, \llbracket \mathbf{x}^{(1)} \rrbracket) \leftarrow \pi_{\text{Gaussian}}$.
3. Each party locally computes $\llbracket \Delta \mathbf{x} \rrbracket \leftarrow \llbracket \mathbf{x}^{(1)} \rrbracket - \llbracket \mathbf{x}^{(0)} \rrbracket$.
4. For $i \in [2n]$, parties call $(\llbracket r_i \rrbracket, \llbracket r_i \rrbracket_2) \leftarrow \text{daBit}$ on $\mathcal{F}_{\text{eABB}}$. Stack the results into $(\llbracket \mathbf{r} \rrbracket, \llbracket \mathbf{r} \rrbracket_2)$.
5. Parties call $\llbracket \mathbf{m} \rrbracket \leftarrow \llbracket \mathbf{r} \rrbracket \odot \llbracket \Delta \mathbf{x} \rrbracket$.
6. Each party locally computes $\llbracket \mathbf{s}_0 \rrbracket \leftarrow \llbracket \mathbf{x}^{(0)} \rrbracket + \llbracket \mathbf{m} \rrbracket$ and $\llbracket \mathbf{s}_1 \rrbracket \leftarrow \llbracket \Delta \mathbf{x} \rrbracket - 2 \llbracket \mathbf{m} \rrbracket$.
7. Parse $\llbracket \mathbf{s}_0 \rrbracket = (\llbracket \mathbf{s}_{0,0} \rrbracket \parallel \llbracket \mathbf{s}_{0,1} \rrbracket)$ and call the two polynomial multiplications $\llbracket f \cdot \mathbf{s}_{0,1} \rrbracket$ and $\llbracket g \cdot \mathbf{s}_{0,0} \rrbracket$ over $\mathcal{R}$; each party locally sets $\llbracket \mathbf{w}_0 \rrbracket \leftarrow \llbracket f \cdot \mathbf{s}_{0,1} \rrbracket - \llbracket g \cdot \mathbf{s}_{0,0} \rrbracket$.
8. Form $\mathbf{M} = [-\text{rot}(g) \mid \text{rot}(f)]$ entry-wise: for each $(j, i) \in [n] \times [2n]$, the entry $\llbracket \mathbf{M}_{j,i} \rrbracket$ is a signed coordinate of $\llbracket f \rrbracket$ or $\llbracket g \rrbracket$, derived locally by a (signed) permutation of the stored shares. Parties call $\llbracket \mathbf{V}_{j,i} \rrbracket \leftarrow \llbracket \mathbf{M}_{j,i} \rrbracket \cdot \llbracket (\mathbf{s}_1)_i \rrbracket$ for each $(j, i) \in [n] \times [2n]$.
9. Output (salt, $\llbracket \mathbf{w}_0 \rrbracket, \llbracket \mathbf{V} \rrbracket, \llbracket \mathbf{r} \rrbracket_2$).

3.3 The Gaussian Sampling Sub-Procedure

π_{Gaussian}

A core primitive we need is sampling obliviously from certain Gaussian distributions. The first procedure $\pi_{\text{HalfGS}}(T)$ below takes as input a RCDT, and samples from it by obtaining a random integer and determining via a set of comparisons— instantiated using the (ltc) command from $\mathcal{F}_{\text{eABB}}$ —the position of the table this corresponds to.

Procedure 2: $\pi_{\text{HalfGS}}(T)$

1. Sample $\llbracket u \rrbracket_2 \in \mathbb{F}_2^{78}$ uniformly (via 78 independent binary rand calls).

2. For each $i \in \{0, \dots, 12\}$, call $\llbracket b_i \rrbracket_2 \leftarrow \text{lrc}(\llbracket u \rrbracket_2, T[i])$ on $\mathcal{F}_{\text{eABB}}$, all 13 calls in parallel.
3. For each $i \in \{0, \dots, 12\}$, convert $\llbracket b_i \rrbracket_2$ to an arithmetic share $\llbracket b_i \rrbracket$:
 - (a) Call $(\llbracket r_i \rrbracket, \llbracket r_i \rrbracket_2) \leftarrow \text{daBit}$ on $\mathcal{F}_{\text{eABB}}$.
 - (b) Open $d_i \leftarrow \text{open}(\llbracket b_i \rrbracket_2 \oplus \llbracket r_i \rrbracket_2)$.
 - (c) Each party locally sets $\llbracket b_i \rrbracket \leftarrow d_i + \llbracket r_i \rrbracket - 2d_i \cdot \llbracket r_i \rrbracket$.
4. Each party locally sets $\llbracket h \rrbracket \leftarrow \sum_{i=0}^{12} \llbracket b_i \rrbracket$.
5. Return $\llbracket h \rrbracket$.

The procedure above will be ultimately used to sample *half Gaussians*. The procedure π_{RndFlip} below randomly shifts this sampled value to obtain a symmetric sample around 0.

Procedure 3: $\pi_{\text{RndFlip}}(\llbracket x \rrbracket)$

1. Call $(\llbracket \rho \rrbracket, \llbracket \rho \rrbracket_2) \leftarrow \text{daBit}$ on $\mathcal{F}_{\text{eABB}}$ (only the arithmetic share $\llbracket \rho \rrbracket$ is used below).
2. Each party locally sets $\llbracket \sigma \rrbracket \leftarrow 1 - 2 \llbracket \rho \rrbracket$, an arithmetic share of $\sigma \in \{-1, +1\}$ uniform.
3. Call $\llbracket y \rrbracket \leftarrow \text{mult}(\llbracket \sigma \rrbracket, \llbracket x \rrbracket)$.
4. Return $\llbracket y \rrbracket$.

Finally, Procedure π_{Gaussian} below puts together the techniques from above to sample from the distributions required in HAWK.

Procedure 4: π_{Gaussian}

The procedure is parametrized by two public 13-entry RCDTs $T_0, T_1 \in (\mathbb{F}_2^{78})^{13}$.

Output. Arithmetic-domain sharings $\llbracket \mathbf{x}^{(0)} \rrbracket, \llbracket \mathbf{x}^{(1)} \rrbracket \in \mathbb{Z}_q^{2n}$.

1. For each $i \in [2n]$, in parallel:
 - (a) Call $\llbracket h_0^{(i)} \rrbracket \leftarrow \pi_{\text{HalfGS}}(T_0)$ and $\llbracket h_1^{(i)} \rrbracket \leftarrow \pi_{\text{HalfGS}}(T_1)$.
 - (b) Each party locally sets $\llbracket y_0^{(i)} \rrbracket \leftarrow 2 \llbracket h_0^{(i)} \rrbracket$ and $\llbracket y_1^{(i)} \rrbracket \leftarrow 2 \llbracket h_1^{(i)} \rrbracket + 1$.
 - (c) Call $\llbracket x_i^{(0)} \rrbracket \leftarrow \pi_{\text{RndFlip}}(\llbracket y_0^{(i)} \rrbracket)$ and $\llbracket x_i^{(1)} \rrbracket \leftarrow \pi_{\text{RndFlip}}(\llbracket y_1^{(i)} \rrbracket)$.
2. Stack the per-coordinate shares into $\llbracket \mathbf{x}^{(0)} \rrbracket, \llbracket \mathbf{x}^{(1)} \rrbracket$ and store them under the requested handles.

3.4 Security of the Offline Phase

Theorem 1. *Protocol Π_{Prep} UC-realizes the functionality $\mathcal{F}_{\text{Prep}}$ in the $\mathcal{F}_{\text{eABB}}$ -hybrid model against any static adversary*

corrupting an arbitrary strict subset of the parties.

The proof reduces the adversary's view to one public coin (the salt) and the $52n$ daBit-masked bit openings of π_{Gaussian} , each uniform and independent of the value it hides; it is given in Section E.

4 Threshold Signing: Online Phase

Now we are ready to describe the online phase of our protocol. First we describe the functionality to instantiate, $\mathcal{F}_{\text{Sign}}$, in Section 4.1. Then we present our main (online phase) protocol in Section 4.2. Finally, we analyze its security in Section 4.3.

4.1 Threshold Signing Functionality

Functionality $\mathcal{F}_{\text{Sign}}$ below captures the threshold version of HAWK we ultimately aim at instantiating in this work. It contains three commands: (key-gen), which stores internally the key, (sample-salt), which samples a salt, appends it to a queue of unused salts and sends it to all parties, and (sign), which as the name implies generates the signature. The functionality exposes a salt-sampling command because our protocol publishes the salt during preprocessing, before the message is known; this is the only preprocessed value the adversary learns early.

Concretely, (key-gen) runs $\text{HAWK.KeyGen}()$ once, stores $\text{sk} = (\mathbf{B}, \text{hpub})$ internally and sends $\text{vk} = (\mathbf{Q}, \text{hpub})$ to all parties; (sample-salt) samples $\text{salt} \xleftarrow{\$} \{0, 1\}^\ell$, appends it to a queue Q and sends it to all parties; and (sign, m) pops the first salt from Q (aborting if Q is empty), runs $\text{HAWK.SignDet}(\text{sk}, m, \text{salt})$, and sends to all parties the resulting signature σ together with the pre-sym-break polynomial w_1 used internally. The queue is what enforces that each salt serves at most one signature. The full description is deferred to Section C.

Remark 3 (On “explicit-salt HAWK” and comparison to [3]). Note that (1) $\mathcal{F}_{\text{Sign}}$ only executes a signature once the salt has been generated, and (2) $\mathcal{F}_{\text{Sign}}$ destroys a salt once it has been used. We refer to this variant of HAWK as *explicit-salt HAWK*, a name inspired by explicit-salt MAYO from [3]. As in [3], the signing functionality maintains a queue of unused salts, which is what enforces that every salt is used for at most one signature. Our formalism nevertheless differs from theirs in two aspects. First, our variant captures additional leakage during the signing protocol. Specifically, the variant signing algorithm outputs not only a signature but also an auxiliary value. Second, we show that our variant is SUF-CMA , which is a stronger notion than existential unforgeability under CMA considered in [3]. For more details on formalization and security analysis of our variant, refer to Section 2.3 and Section F.

4.2 The Online Signing Protocol Π_{Sign}

Now we propose our protocol for (online) signing, Π_{Sign} . In short the protocol, which is set in the $\mathcal{F}_{\text{Prep}}$ -hybrid model, uses the functionality to generate the key and to produce the necessary preprocessing. Then the ABB is used to execute the online phase sketched in the technical overview (Section 1.2).

Protocol 5: Π_{Sign}

The protocol is set in the $\mathcal{F}_{\text{Prep}}$ -hybrid model.

Setup. The parties have previously invoked $\mathcal{F}_{\text{Prep}}$'s key-gen to populate $\llbracket f \rrbracket, \llbracket g \rrbracket, \llbracket F \rrbracket, \llbracket G \rrbracket$ under known handles, which they assemble into $\llbracket B \rrbracket$. They also hold one or more preprocessing tuples $(\text{salt}, \llbracket w_0 \rrbracket, \llbracket V \rrbracket, \llbracket r \rrbracket_2)$ produced by calls to (prep), which they consume in the order in which they were produced.

Input. A public message m .

Output. A signature $\sigma = (\text{salt}, s_1)$ on m .

1. Let $(\text{salt}, \llbracket w_0 \rrbracket, \llbracket V \rrbracket, \llbracket r \rrbracket_2)$ be the first unused preprocessing tuple, which the parties mark as used.
2. Each party locally computes $M \leftarrow H(m \parallel \text{hpub})$ and $\mathbf{h} = (h_0, h_1) \leftarrow H(M, \text{salt}) \in \mathbb{F}_2^{2n}$, the binary-domain matrix $\llbracket B \rrbracket_2 \leftarrow \text{mod2}(\llbracket B \rrbracket)$, and sets $\llbracket t \rrbracket_2 \leftarrow \llbracket B \rrbracket_2 \cdot \mathbf{h}$.
3. Parties call $\mathbf{c} \leftarrow \text{open}(\llbracket t \rrbracket_2 \oplus \llbracket r \rrbracket_2)$.
4. Each party locally computes

$$\llbracket w \rrbracket \leftarrow \llbracket w_0 \rrbracket + \llbracket V \rrbracket \cdot \mathbf{c}.$$

5. Parties call $\mathbf{w} \leftarrow \text{open}(\llbracket w \rrbracket)$.
6. If $\neg \text{sym-break}(\mathbf{w})$, set $\mathbf{w} \leftarrow -\mathbf{w}$.
7. Each party locally computes $s_1 \leftarrow (h_1 - w)/2$ and outputs $\sigma = (\text{salt}, s_1)$.

Remark 4 (Ordering of signing requests). The queue discipline of $\mathcal{F}_{\text{Sign}}$ requires the parties to agree on the order in which preprocessing tuples are consumed, which formally is an assumption on the environment: all parties receive the same sequence of (prep) calls and signing requests. In deployment it is provided by whichever component already serializes client requests, such as a gateway in front of the signing nodes or a state-machine-replication layer.

4.3 Security of the Online Phase

Theorem 2. *Protocol Π_{Sign} UC-realizes the functionality $\mathcal{F}_{\text{Sign}}$ in the $\mathcal{F}_{\text{Prep}}$ -hybrid model against any static adversary corrupting an arbitrary strict subset of the parties.*

Proof sketch. The protocol performs exactly two openings, and every other step is local. The first opens $\mathbf{c} = \mathbf{t} \oplus \mathbf{r}$, which is uniform because $\mathbf{r}$ is a fresh daBit mask, and the second opens $\mathbf{w}$, which equals the value w_1 that $\mathcal{F}_{\text{Sign}}$ returns to the simulator. Correctness follows from the identities of Section 3.1,

which give $\mathbf{w} = \mathbf{M} \cdot (\mathbf{x}^{(0)} + \mathbf{t} \odot \Delta \mathbf{x})$, together with the fact that $\mathbf{x}^{(0)}$ and $\mathbf{x}^{(1)}$ are sampled independently of $\mathbf{t}$, so the Gaussian vector implicitly used by Π_{Sign} is distributed as a fresh sample from $\mathcal{D}^{(t)}$. The full proof is given in Section E. $\square$

5 Communication/Rounds-Optimizations

5.1 A communication optimized variant

As discussed in Section 1.2, running the signature protocol in only 2 online rounds requires the communication-heavy pre-computation of the matrix $\llbracket V \rrbracket$, which consumes $O(n^2)$ secret multiplications. This cost is analyzed precisely in Section 5.5. We also provide a communication-optimized variant of our signing protocol. This alternative entirely removes the need to compute the expensive quadratic terms, at the cost of one extra online round. Instead of resolving the polynomial products during the offline phase, this variant performs a coefficient-wise selection of the Gaussian samples based on the target centers, followed by an online polynomial multiplication.

The offline phase of this variant follows Π_{Prep} up to step 6, omitting the polynomial products of step 7 and the computation of $\llbracket V \rrbracket$ in step 8; the stored tuple is $(\text{salt}, \llbracket s_0 \rrbracket, \llbracket s_1 \rrbracket, \llbracket r \rrbracket_2)$. We detail the online phase of this communication-optimized variant in $\Pi_{\text{CO-Sign}}$.

Protocol 6: $\Pi_{\text{CO-Sign}}$

Setup. Each party holds, stored in $\mathcal{F}_{\text{Prep}}$, arithmetic-domain shares $\llbracket B \rrbracket$ of the signing matrix and one or more preprocessing tuples $(\text{salt}, \llbracket s_0 \rrbracket, \llbracket s_1 \rrbracket, \llbracket r \rrbracket_2)$, where $\llbracket s_0 \rrbracket$ and $\llbracket s_1 \rrbracket$ are precomputed to allow the linear selection of the Gaussian samples. As in Π_{Sign} , the tuples are consumed in the order in which they were produced.

Input. A public message m .

Output. A signature $\sigma = (\text{salt}, s_1)$ on m , or $\perp$.

1. Each party locally computes $M \leftarrow H(m \parallel \text{hpub})$ and $\mathbf{h} = (h_0, h_1) \leftarrow H(M, \text{salt}) \in \mathbb{F}_2^{2n}$, the binary sharing $\llbracket B \rrbracket_2 \leftarrow \text{mod2}(\llbracket B \rrbracket)$, and $\llbracket t \rrbracket_2 \leftarrow \llbracket B \rrbracket_2 \cdot \mathbf{h}$.
2. — *Round 1: Selection* —
Parties call $\mathbf{c} \leftarrow \text{open}(\llbracket t \rrbracket_2 \oplus \llbracket r \rrbracket_2)$.
3. Each party locally computes the coefficient-wise (Hadamard) selection:

$$\llbracket \mathbf{x} \rrbracket \leftarrow \llbracket s_0 \rrbracket + \mathbf{c} \odot \llbracket s_1 \rrbracket$$

and parses it as $\llbracket \mathbf{x} \rrbracket = (\llbracket x_0 \rrbracket \parallel \llbracket x_1 \rrbracket) \in \mathcal{R}^2$.

4. — *Round 2: Convolution* —
Parties call the secure polynomial multiplications over $\mathcal{R}$:

$$\llbracket v_f \rrbracket \leftarrow \llbracket f \rrbracket \cdot \llbracket x_1 \rrbracket \quad \text{and} \quad \llbracket v_g \rrbracket \leftarrow \llbracket g \rrbracket \cdot \llbracket x_0 \rrbracket$$

5. Each party locally sets $\llbracket w \rrbracket \leftarrow \llbracket v_f \rrbracket - \llbracket v_g \rrbracket$.

6. — *Round 3: Opening* —
Parties call $\mathbf{w} \leftarrow \text{open}(\llbracket \mathbf{w} \rrbracket)$.
7. If $\neg \text{sym-break}(\mathbf{w})$, set $\mathbf{w} \leftarrow -\mathbf{w}$.
8. Each party locally computes $s_1 \leftarrow (h_1 - w)/2$ and outputs $\sigma = (\text{salt}, s_1)$.

If the HAWK public rejection conditions fail when applied to $\mathbf{w}$, the parties output $\perp$, discard the preprocessing tuple, and restart with a fresh call to prep.

Note that this version requires a polynomial multiplication over $\mathcal{R}$ in the online phase. This operation can be achieved in a single round with a communication cost linear in n by using Beaver triples [5] generated directly in the polynomial ring $\mathcal{R}$.

5.2 Fine-grained rounds and communication cost analysis.

As discussed in Section 5.1, the online phase of our signature scheme runs in either 2 or 3 rounds, depending on the chosen communication cost tradeoff. In this section, we evaluate the performance of our protocol. We first break down the round complexity of the offline phase (Section 5.3) and the communication costs of our underlying cryptographic building blocks (Section 5.4). We then aggregate these metrics to derive the total communication cost for both the offline and online phases (Section 5.5). Finally, we explore several tradeoffs that allow further reduction of the offline round complexity at the expense of higher communication overhead (Section 5.6).

Scope. Our analysis evaluates the protocol in the semi-honest model and deliberately omits the raw generation phase of daBits [23, 38] and Beaver triples [5]. Because this cryptographic material is entirely independent of both the signing key and the message, it can be batch-generated arbitrarily far in advance using standard techniques, based either on oblivious transfer [30] or on somewhat homomorphic encryption [19, 31]. Furthermore, upgrading our protocol to malicious security adds only a single online round dedicated to the final MAC-check (and also adds offline rounds), along with a constant, negligible communication overhead.

5.3 Offline Round Complexity

The offline phase primarily consists of evaluating the discrete Gaussian sampler and, optionally, linearizing the final polynomial. Because the sampling applies to $2n$ coefficients for both secret key polynomials (f and g) and spans two candidate centers ($x^{(0)}$ and $x^{(1)}$), the operations below are executed $4n$ times in parallel. The round complexity decomposes as follows:

1. **Boolean Comparison (RCDT):** The evaluation of 13 parallel leq circuits over 78-bit integers. Using a standard

Brent-Kung tree topology, this step takes $\lceil \log_2(78) \rceil = 7$ rounds.

2. **Mask Opening:** One round of communication to publicly reconstruct the masked comparison bits.
3. **Sample Assembly:** One round of arithmetic multiplications to apply the Gaussian sign and assemble the final values $\llbracket \mathbf{x}^{(0)} \rrbracket$ and $\llbracket \mathbf{x}^{(1)} \rrbracket$.
4. **Polynomial Linearization:** One round containing the parallel polynomial products and the $O(n^2)$ scalar multiplications necessary to construct $\llbracket \mathbf{V} \rrbracket$ and $\llbracket \mathbf{w}_0 \rrbracket$. (This step is strictly required for Π_{Sign} and omitted in $\Pi_{\text{CO-Sign}}$).

Due to full vectorization, the offline phase runs in exactly 10 rounds for Π_{Sign} and 9 rounds for $\Pi_{\text{CO-Sign}}$.

5.4 Communication Cost of the Building Blocks

We parameterize the communication costs using the number of parties N and a prime ring modulus q . In our HAWK instantiation, we choose q such that $\lceil \log_2 q \rceil = 15$. The only requirement for q is to be sufficiently big so that no reduction occurs on the manipulated values (HAWK operates over the integers; the specification notes that 15-bit NTT-friendly moduli are suitable for all HAWK parameter sets, including $n = 1024$ [9]), and we choose it prime for Shamir secret sharing efficiency.

5.4.1 Multiplication Cost

Assuming a standard Shamir secret-sharing scheme (or additive secret-sharing over authenticated SPDZ rings), we distinguish two domains:

- **Boolean domain:** Binary values within the sampler cannot be shared natively using Shamir over $\mathbb{F}_2$ if $N > 2$. We operate over an extension field $\mathbb{F}_{2^\kappa}$, which has $2^\kappa - 1$ nonzero evaluation points and hence supports up to $2^\kappa - 1$ parties, so we need $2^\kappa > N$. The communication cost of one Boolean multiplication (an AND gate) is $\approx 2\kappa$ bits per party. For the $N = 1024$ signers of Table 1 this gives $\kappa = 11$; the smaller choice $\kappa = 10$ would support only 1023 parties, and lowers the offline cost of Π_{Sign} to 36.6 MB.
- **Arithmetic domain:** The HAWK operations are executed over $\mathbb{Z}_q$. The communication cost for an arithmetic multiplication (MUL_q) is $2k = 30$ bits per party.

5.4.2 Comparison Cost (leq)

Proposition 1 (Cost of Brent-Kung Comparison). *Let a and b be L -bit integers. The MPC comparison circuit $\text{MPC.Leq}(a, b)$ requires slightly less than $2L$ Boolean multiplications and runs in $\lceil \log_2(L) \rceil$ communication rounds.*

For the HAWK parameters, following the specification, we evaluate the RCDT on $L = 78$ bits, yielding approximately 156 Boolean multiplications per leq invocation.

Remark 5 (On the choice of the modulus q). Both a prime and a power of two are viable choices for q . With q prime an arithmetic share is a single field element (for Shamir it suffices that $q > N$), but the binary domain must be embedded in an unrelated extension $\mathbb{F}_{2^\kappa}$ with $2^\kappa > N$, so every daBit requires a dedicated cross-domain protocol [23, 38]. With $q = 2^k$, reduction modulo 2 is a ring homomorphism and hence local on shares; Shamir then requires a Galois ring $\text{GR}(2^k, \kappa)$ with $2^k > N$ [1], inflating arithmetic shares by a factor κ , but that same reduction maps $\text{GR}(2^k, \kappa)$ onto $\mathbb{F}_{2^\kappa}$ —exactly the extension the binary domain already needs—so a daBit degenerates into sampling a shared random bit and reducing it locally.

Which side wins depends on how arithmetic-heavy the variant is. Π_{Sign} is dominated by the $\Theta(n^2)$ products assembling $\llbracket \mathbf{V} \rrbracket$ and $\llbracket \mathbf{w}_0 \rrbracket$, so the factor κ is expensive and a prime modulus is preferable; $\Pi_{\text{CO-Sign}}$ removes those products, leaving an almost purely Boolean offline phase in which the same factor is negligible while the savings on the $\Theta(n)$ daBits of the RCDT sampler remain. For a few parties the question disappears altogether: replicated sharing over $\mathbb{Z}_{2^k}$ needs no extension, as exploited by the four-party instantiation of Section 6. This comparison is qualitative, since daBit generation is excluded from Section 5.5.

5.5 Total Protocol Communication Cost

Based on the component costs from Section 5.4, we summarize the aggregate operation counts for both variants of the protocol. For an n -dimensional HAWK ring ($n = 1024$ for HAWK-1024), the total footprint is heavily dominated by the offline phase.

We only need to open w_1 , which is half of w . The opening communication cost is therefore n times the size of an arithmetic share.

We derive concrete numbers from these formulas in Table 1. We also provide a small Python script⁴ to compute the offline and online communication cost depending on the number of parties.

5.6 Offline Rounds Tradeoffs

We present three distinct tradeoffs that allow pushing the offline round complexity even lower, at the expense of higher

Table 2: Operation count per signature ($n = 1024$)

Phase	$\Pi_{\text{CO-Sign}}$ (3-Round)	Π_{Sign} (2-Round)
Offline	$4n \times (13\text{MPC.Leq} + 13 \text{ daBit Opens} + 1 \text{ MPC.Mult}_q) + 2n \text{ MPC.Mult}_q \text{ (mask)}$	Same as $\Pi_{\text{CO-Sign}}$ + $4n^2 \text{ MPC.Mult}_q$ (Linearization: $\mathbf{w}_0, \mathbf{V}$)
Online	$2n \text{ Binary Mask Opens} + 2 \text{ Ring Products} + 1 \text{ Signature Open}$	$2n \text{ Binary Mask Opens} + 1 \text{ Signature Open}$

communication or precomputation overhead:

- **Full-Gaussian RCDT:** It is possible to save the round dedicated to the sign application by evaluating the Cumulative Distribution Table of the *full* Gaussian, rather than that of a Half-Gaussian. While this saves one round and removes a scalar multiplication, it doubles the size of the RCDT. Consequently, the number of leq evaluations (and their associated communication cost) jumps from 13 to 26.
- **Higher-Degree Beaver Triples:** By generating degree-3 Beaver triples during the raw preprocessing phase, the comparison circuit can be evaluated with a depth of $\lceil \log_3(78) \rceil = 4$ rounds instead of $\lceil \log_2(78) \rceil = 7$. However, standard techniques for classic Beaver triple generation do not apply to generating higher-degree ones. We therefore need to compute the degree-3 triples from the degree-2 ones, which effectively takes two rounds. In total, this saves one round of offline latency but effectively doubles the communication cost of every leq gate.
- **Cross-Term Precomputation:** We can precompute the correlated algebraic cross-terms to perfectly parallelize the algebraic assembly with the comparison circuit. Applying this technique to the daBits and the Δx terms saves the mask-opening dependency round, at the cost of multiplying the number of required arithmetic multiplications by a factor of exactly 14 (one sign term plus 13 daBit combinations). While this $14\times$ overhead is highly efficient for the sampler itself, attempting to apply this cross-term parallelization to the $\Theta(n^2)$ linearization products of Π_{Sign} makes the communication absolutely prohibitive.

These different tradeoffs can reduce the number of offline rounds down to 7, at the cost of increasing the communication cost of the leq step. Such optimizations are particularly suitable when the leq footprint is small compared to the total offline communication cost, typically in deployments optimized for round complexity involving few parties.

⁴<https://github.com/hdvanegas/kettle>

6 Implementation

To evaluate our algorithm’s real-world performance, we implemented it using the MP-SPDZ framework [29].⁵ Our protocol is expressed entirely in terms of $\mathcal{F}_{\text{eABB}}$, using only generic operations and no scheme-specific optimization, so MP-SPDZ is a natural fit: it runs a single circuit against many interchangeable backends, letting us vary the security setting while keeping the circuit fixed. Our goal is to measure the *relative* cost of Π_{Prep} followed by Π_{Sign} across these backends in a single emulated network setup.

To benchmark our protocol, we choose four different instantiations of $\mathcal{F}_{\text{eABB}}$ that are secure against a malicious adversary: a protocol based on Shamir secret sharing that is secure in the honest majority setting, which we will call “malicious Shamir” [33]; a SPDZ-wise Shamir version that do not require multiplication triples [2, 14], also in the honest majority setting; the MASCOT protocol which is secure in the dishonest majority setting; and the four-party protocol Fantastic Four [17], which achieves robustness, does not require correlated randomness generation, and is secure in the honest majority setting. The first two instantiations let us choose any number of parties and any corruption threshold, as long as we have an honest majority, and the underlying algebraic structure is $\mathbb{F}_p$. The MASCOT protocol, whose computational domain is also $\mathbb{F}_p$, allows us to choose any threshold up to $N - 1$ parties. In contrast, the Fantastic Four protocol runs over $\mathbb{Z}_{2^k}$ and is specifically for four parties with one corruption, exploiting this setting to achieve high efficiency.

This work uses two distinct notions of pre-processing: *MPC pre-processing*, the correlated material required to execute an arithmetic circuit, and *signing pre-processing*, the protocol Π_{Prep} . Our benchmarks exclude the former and include the latter in full.

For protocol execution, we emulate a LAN with a network delay of 0.5 milliseconds and a bandwidth of 1 Gbit/s using the `tc` Linux command. In our implementation, we assume all parties are connected via a point-to-point authenticated private channel, and we do not consider a broadcast channel. We executed the protocols on a Google Cloud `n4a-highcpu-64` instance with 64 vCPU and 128 GiB of RAM.

For each instantiation of the arithmetic black-box, we execute Π_{Prep} followed by the execution of Π_{Sign} among 8 parties with a corruption threshold of 3 parties (except Fantastic Four, which is fixed to 4 parties and 1 corruption). We independently run each parameter combination 30 times. As a result, we report the average execution time and a 95% confidence interval (CI); in every configuration, the half-width of this interval is below 1% of the reported mean, except for MASCOT, which is below 2.7%. The results are shown in Figure 1.

The results show different tradeoffs depending on the concrete realization of $\mathcal{F}_{\text{eABB}}$. First, notice that Fantastic Four is the most efficient protocol in this execution. This protocol

Average running time per protocol

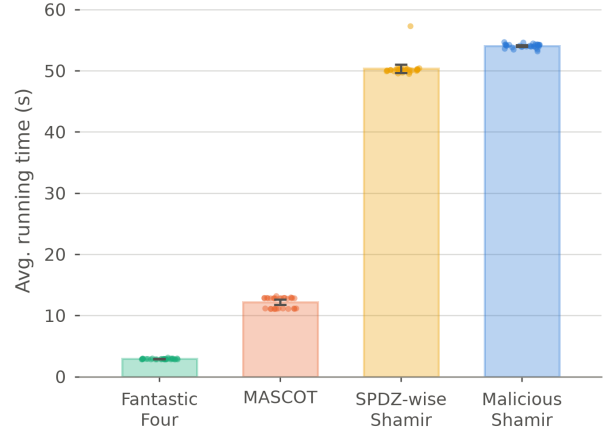


Figure 1: Average runtime for each protocol in an emulated LAN. Each bar shows the average with each protocol execution shown as points and the 95% confidence interval.

has the following advantages: the number of parties and corruptions is fixed, and the protocol exploits this fact to catch the only malicious adversary and achieve robustness without sacrificing performance; also, the computational domain is $\mathbb{Z}_{2^k}$ which offers facilities to compute over mixed arithmetic circuits as in our case. Another advantage of Fantastic Four is that it is not in the online/offline MPC paradigm, which means the protocol does not need to generate correlated material before the signing message is known. However, unlike the other three protocols, Fantastic Four is restricted to four parties, which may be a considerable limitation in some practical settings. Second, without MPC pre-processing, MASCOT surpasses the other two protocols over $\mathbb{F}_p$ even when the latter guarantee security on a dishonest majority, while the former are secure on an honest majority. This is because MASCOT’s online phase is more efficient than Shamir-based protocols. However, Shamir-based protocols outperform MASCOT in the offline phase (which we exclude from the experiments). These results show the versatility of our protocol, as it can fit in multiple setups and real-world applications by carefully analyzing the tradeoffs and the characteristics of each arithmetic black-box instantiation.

On the interpretation of these timings. These numbers are meant to be read horizontally, as a comparison between security settings for one and the same protocol, not as absolute performance figures: MP-SPDZ pays a substantial generic-circuit overhead, so a hand-optimized Kettle would be considerably faster, whereas the schemes of Table 1 report purpose-built implementations. We therefore refrain from concrete timing comparisons with other threshold signatures, and compare only on the implementation-independent metrics of Table 1.

⁵<https://github.com/hdvanegas/kettle>

Acknowledgments

This work was supported in part by JST K Program Grant Number JPMJKP24U3.

References

- [1] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Cham, December 2019. doi:10.1007/978-3-030-36030-6_19.
- [2] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 2021, Part II*, volume 12727 of *LNCS*, pages 122–152. Springer, Cham, June 2021. doi:10.1007/978-3-030-78375-4_6.
- [3] Diego F. Aranha, Giacomo Borin, Sofía Celi, and Guilhem Niot. Optimizing and implementing threshold MAYO. CHES, 2026.
- [4] Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 449–458. ACM Press, October 2008. doi:10.1145/1455770.1455827.
- [5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO’91*, volume 576 of *LNCS*, pages 420–432. Springer, Berlin, Heidelberg, August 1992. doi:10.1007/3-540-46766-1_34.
- [6] Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 517–550. Springer, Cham, August 2022. doi:10.1007/978-3-031-15985-5_18.
- [7] Rikke Bendlin, Sara Krehbiel, and Chris Peikert. How to share a lattice trapdoor: Threshold protocols for signatures and (H)IBE. In Michael J. Jacobson, Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 2013*, volume 7954 of *LNCS*, pages 218–236. Springer, Berlin, Heidelberg, June 2013. doi:10.1007/978-3-642-38980-1_14.
- [8] Alexander Bienstock, Leo de Castro, Daniel Escudero, Antigoni Polychroniadou, and Akira Takahashi. Quorus: Efficient, scalable threshold ML-DSA signatures from MPC. In Elissa Redmiles and Ben Stock, editors, *USENIX Security 2026*. USENIX Association, August 2026.
- [9] Joppe W. Bos, Olivier Bronchain, Léo Ducas, Serge Fehr, Yu-Hsuan Huang, Thomas Pornin, Eamonn W. Postlethwaite, Thomas Prest, Ludo N. Pulles, and Wessel van Woerden. HAWK. Technical report, National Institute of Standards and Technology, 2024. available at <https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures>.
- [10] Cecilia Boschini, Darya Kaviani, Russell W. F. Lai, Giulio Malavolta, Akira Takahashi, and Mehdi Tibouchi. Ringtail: Practical two-round threshold signatures from learning with errors. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 149–164. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00070.
- [11] Luis Brandao and Rene Peralta. NIST first call for multiparty threshold schemes. Technical report, National Institute for Standards and Technology, January 2026. URL: <https://csrc.nist.gov/pubs/ir/8214/c/final>.
- [12] Sofia Celi, Rafael del Pino, Thomas Espitau, Guilhem Niot, and Thomas Prest. Efficient threshold ML-DSA. In Elissa Redmiles and Ben Stock, editors, *USENIX Security 2026*. USENIX Association, August 2026.
- [13] Rutchathon Chairattana-Apirom, Stefano Tessaro, and Chenzhi Zhu. Partially non-interactive two-round lattice-based threshold signatures. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part IV*, volume 15487 of *LNCS*, pages 268–302. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0894-2_9.
- [14] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Cham, August 2018. doi:10.1007/978-3-319-96878-0_2.
- [15] Jean-Sébastien Coron. On the exact security of full domain hash. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 229–235. Springer, Berlin, Heidelberg, August 2000. doi:10.1007/3-540-44598-6_14.

- [16] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully adaptive Schnorr threshold signatures. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 678–709. Springer, Cham, August 2023. doi:10.1007/978-3-031-38557-5_22.
- [17] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2183–2200. USENIX Association, August 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov>.
- [18] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00078.
- [19] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012. doi:10.1007/978-3-642-32009-5_38.
- [20] Manu Drijvers, Krasa Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igor Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00050.
- [21] Léo Ducas, Alain Durmus, Tancrede Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, Berlin, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_3.
- [22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 65–94. Springer, Cham, December 2022. doi:10.1007/978-3-031-22972-5_3.
- [23] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020. doi:10.1007/978-3-031-38557-5_29.
- [24] Thomas Espitau, Shuichi Katsumata, and Kaoru Take-mure. Two-round threshold signature from algebraic one-more learning with errors. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 387–424. Springer, Cham, August 2024. doi:10.1007/978-3-031-68394-7_13.
- [25] Thomas Espitau, Guilhem Niot, and Thomas Prest. Flood and submerge: Distributed key generation and robust threshold signature from lattices. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 425–458. Springer, Cham, August 2024. doi:10.1007/978-3-031-68394-7_14.
- [26] Radhika Garg, Daniel Escudero, Antigoni Polychroniadou, Akira Takahashi, and Xiao Wang. Thresholdizing standardized FALCON signatures. In Véronique Cortier and Zhiqiang Lin, editors, *ACM CCS 2026*. ACM Press, November 2026.
- [27] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008. doi:10.1145/1374376.1374407.
- [28] Kamil Doruk Gur, Patrick Hough, Jonathan Katz, Caroline Sandsbråten, and Tjerdand Silde. Olingo: Threshold lattice signatures with DKG and identifiable abort. Cryptology ePrint Archive, Report 2025/1789, 2025. URL: <https://eprint.iacr.org/2025/1789>.
- [29] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. doi:10.1145/3372297.3417872.
- [30] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016. doi:10.1145/2976749.2978357.
- [31] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Cham, April / May 2018. doi:10.1007/978-3-319-78372-7_6.

- [32] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O’Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, Cham, October 2020. doi:10.1007/978-3-030-81652-0_2.
- [33] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017. doi:10.1145/3133956.3133999.
- [34] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Berlin, Heidelberg, December 2009. doi:10.1007/978-3-642-10366-7_35.
- [35] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Berlin, Heidelberg, April 2012. doi:10.1007/978-3-642-29011-4_43.
- [36] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [37] Rafaël Del Pino, Shuichi Katsumata, Guilhem Niot, Michael Reichle, and Kaoru Takemure. Unmasking TRaccoon: A lattice-based threshold signature with an efficient identifiable abort protocol. In Yael Tsauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part VI*, volume 16005 of *LNCS*, pages 423–456. Springer, Cham, August 2025. doi:10.1007/978-3-032-01887-8_14.
- [38] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Cham, December 2019. doi:10.1007/978-3-030-35423-7_12.
- [39] Zygimantas Straznickas and Stephen A. Weis. HAWK- n key recovery reduces to SVP in dimension $n/2 + 1$. Technical report, Anthropic, July 2026. Available at <https://www-cdn.anthropic.com/e8d50c167ad>

47beeb03d6109a4a484be95cb38ea/hawk_key_recovery.pdf.

Functionalities

Functionality 1: $\mathcal{F}_{\text{KG}}$	17
Functionality 2: $\mathcal{F}_{\text{eABB}}$	18
Functionality 3: $\mathcal{F}_{\text{PRep}}$	18
Functionality 4: $\mathcal{F}_{\text{Sign}}$	18

Protocols

Protocol 1: Π_{PRep}	8
Protocol 5: Π_{Sign}	10
Protocol 6: $\Pi_{\text{CO-Sign}}$	10

Procedures

Procedure 2: $\pi_{\text{HalfGS}}(T)$	8
Procedure 3: $\pi_{\text{RndFlip}}(\llbracket x \rrbracket)$	9
Procedure 4: π_{Gaussian}	9

Ethics Considerations

This work does not involve human subjects, personal or otherwise sensitive data, or the measurement of systems operated by third parties. All experiments were run on cloud instances rented and controlled by the authors, executing only code written by the authors. The contribution is defensive: threshold signatures remove the single point of failure represented by a monolithic signing key.

Open Science

We are committed to the open science policy. Our implementation and the script computing the communication costs reported in this paper are available in anonymized form at <https://github.com/hdvanegasm/kettle>. They will be made publicly available upon acceptance.

A Application Domains

Threshold signatures are useful whenever signing authority should not be concentrated in a single device, administrator, or organization. A prominent example is the custody of blockchain assets, where control of a wallet can be distributed among several users, devices, or service providers. This reduces the risk that a single stolen key, compromised endpoint, or malicious insider can authorize a transaction, while also allowing policies such as requiring approval from several departments or geographically separated custodians. Similar mechanisms arise in exchanges, institutional custody platforms,

decentralized autonomous organizations, and smart-contract administration, where threshold authorization can protect privileged operations such as contract upgrades, treasury transfers, or changes to governance parameters. Compared with conventional multisignature constructions, threshold signatures can retain a compact, ordinary signature format and expose only a single public verification key, which is often desirable for compatibility, privacy, and reduced on-chain overhead.

Beyond blockchain systems, threshold signing is relevant to a wide range of high-value key-management services. HSM-backed infrastructures can distribute a signing key across several hardware modules, data centers, or administrative domains, thereby reducing dependence on any single HSM and improving resilience against both compromise and failure. This model is applicable to cloud key-management and signing services, certification-authority keys and other PKI operations, code-signing systems, secure software-update mechanisms, document-signing services, and the authentication of logs or audit records. It can also support organizational separation of duties: for example, a sensitive release may require joint approval from engineering, security, and operations, without reconstructing the signing key at any one location. More generally, threshold signatures combine compromise resistance, fault tolerance, and distributed control while preserving the standard verification interface of the underlying signature scheme, making them attractive as a drop-in strengthening of existing signing infrastructures.

B The HAWK Signature Scheme

Algorithm 1 recalls the HAWK algorithms used throughout the paper, in the notation of Section 2.4.

C Deferred Functionalities

We give here the full descriptions of the key-generation functionality $\mathcal{F}_{\text{KG}}$ of Section 2.6.1, of its extension $\mathcal{F}_{\text{eABB}}$ of Section 2.6.2, of the preprocessing functionality $\mathcal{F}_{\text{Prep}}$ of Section 3.1, and of the threshold signing functionality $\mathcal{F}_{\text{Sign}}$ of Section 4.1.

Functionality 1: $\mathcal{F}_{\text{KG}}$

The functionality maintains an internal dictionary mapping handles id to values, each tagged with a domain in $\{\text{arith}, \text{bin}\}$, and a one-shot flag initialized $\in \{\text{true}, \text{false}\}$, initially false. Parameterized by a domain $\text{dom} \in \{\text{arith}, \text{bin}\}$, the functionality supports the ABB-style commands:

- $(\text{input}, \text{id}, P_i, \text{dom})$ from all parties and $(\text{input}, \text{id}, P_i, x, \text{dom})$ from P_i : store (x, id) .
- $(\text{add}, \text{id}_1, \text{id}_2, \text{id}_3)$: fetch (x, id_1) and (y, id_2) from the same domain and store $(x + y, \text{id}_3)$.

Algorithm 1 HAWK algorithms

HAWK.KeyGen()

- 1: Sample coefficients of $f, g \in \mathcal{R}$ i.i.d. from $\text{Bin}(\eta)$.
- 2: **if** f, g are not invertible modulo $X^n + 1$ and modulo 2 or $\|(f, g)\|_2^2 \leq 2n \cdot \sigma_{\text{sec}}^2$ **then**
- 3: **restart**
- 4: $r \leftarrow \text{NTRUSolve}(f, g, 1)$
- 5: **if** $r = \perp$ **then**
- 6: **restart**
- 7: $(F, G) \leftarrow r$
- 8: **if** $\|(F, G)\|_\infty > 127$ **then**
- 9: **restart**
- 10: $B \leftarrow \begin{pmatrix} f & F \\ g & G \end{pmatrix}, Q \leftarrow B^* B$
- 11: $\text{hpub} \leftarrow H(Q)$
- 12: **return** $(\text{vk}, \text{sk}) = (Q, (B, \text{hpub}))$

HAWK.Sign(sk, m)

- 13: $M \leftarrow H(m \parallel \text{hpub})$
- 14: $\text{salt} \xleftarrow{\$} \{0, 1\}^\ell$
- 15: $\mathbf{h} = (h_0, h_1) \leftarrow H(M, \text{salt})$
- 16: $\mathbf{t} \leftarrow B \cdot \mathbf{h} \bmod 2$
- 17: $\mathbf{x} = (x_0, x_1) \xleftarrow{\$} \mathcal{D}^{(t)}$
- 18: **if** $\|\mathbf{x}\|^2 > 8n \cdot \sigma_{\text{ver}}^2$ **then**
- 19: **restart**
- 20: $w_1 \leftarrow -g \cdot x_0 + f \cdot x_1$
- 21: **if** $\text{sym-break}(w_1)$ **then**
- 22: $w_1 \leftarrow -w_1$
- 23: $s_1 \leftarrow (h_1 - w_1)/2$
- 24: **return** $\sigma = (s_1, \text{salt})$

HAWK.SignDet(sk, m, salt)

- 25: $M \leftarrow H(m \parallel \text{hpub})$
- 26: $\mathbf{h} = (h_0, h_1) \leftarrow H(M, \text{salt})$
- 27: $\mathbf{t} \leftarrow B \cdot \mathbf{h} \bmod 2$
- 28: $\mathbf{x} = (x_0, x_1) \xleftarrow{\$} \mathcal{D}^{(t)}$
- 29: $w_1 \leftarrow -g \cdot x_0 + f \cdot x_1$
- 30: **if** $\text{sym-break}(w_1)$ **then**
- 31: $w'_1 \leftarrow +w_1$
- 32: **else**
- 33: $w'_1 \leftarrow -w_1$
- 34: $s_1 \leftarrow (h_1 - w'_1)/2$
- 35: **return** $\sigma = (s_1, \text{salt})$ and w_1

HAWK.Verify(vk, σ, m)

- 36: $\text{hpub} \leftarrow H(Q)$
- 37: $M \leftarrow H(m \parallel \text{hpub})$
- 38: $\mathbf{h} \leftarrow H(M, \text{salt})$
- 39: $s_0 \leftarrow \text{Decompress}(s_1, \mathbf{h}, Q)$
- 40: $\mathbf{s} \leftarrow (s_0, s_1); \mathbf{w} = (w_0, w_1) \leftarrow \mathbf{h} - 2\mathbf{s}$
- 41: **if** $\text{salt} \in \{0, 1\}^\ell \wedge \mathbf{s} \in \mathcal{R}^2 \wedge \text{sym-break}(w_1) \wedge \|\mathbf{w}\|_Q^2 \leq 8n \cdot \sigma_{\text{ver}}^2$ **then**
- 42: **return** 1
- 43: **return** 0

- (addmult-cons, $\text{id}_1, \text{id}_2, (a, b)$): fetch (x, id_1) and store $(a \cdot x + b, \text{id}_2)$ in the same domain.
- (mult, $\text{id}_1, \text{id}_2, \text{id}_3$): fetch (x, id_1) and (y, id_2) from the same domain and store $(x \cdot y, \text{id}_3)$.
- (open, id): fetch (x, id) and send x to all parties.
- (rand, id, dom): sample r uniformly from the corresponding ring and store (r, id) .
- (coin, dom): sample r uniformly from the corresponding ring and send r to all parties.
- (mod2, id_1, id_2): fetch (x, id_1) from arith and store $(x \bmod 2, \text{id}_2)$ in bin.

In addition:

- On input (key-gen, $\text{id}_f, \text{id}_g, \text{id}_F, \text{id}_G$) from all parties:
 1. If initialized = true, abort.
 2. Run HAWK.KeyGen() (Algorithm 1) to obtain $(\text{vk}, \text{sk}) = (\mathbf{Q}, (\mathbf{B}, \text{hpub}))$ with $\mathbf{B} = \begin{pmatrix} f & F \\ g & G \end{pmatrix}$ and $f, g, F, G \in \mathcal{R}$.
 3. Store $(f, \text{id}_f), (g, \text{id}_g), (F, \text{id}_F), (G, \text{id}_G)$ in the arithmetic domain (each polynomial coordinate-wise).
 4. Set initialized $\leftarrow$ true and send $\text{vk} = (\mathbf{Q}, \text{hpub})^a$ to all parties.

^aWhile hpub is a public component of sk in the standard HAWK, we include it in vk for simplicity.

Functionality 2: $\mathcal{F}_{\text{eABB}}$

Supports all commands of $\mathcal{F}_{\text{KG}}$. In addition:

- On input (daBit, id_a, id_b) from all parties: sample $r \xleftarrow{\$} \{0, 1\}$, store (r, id_a) in the arithmetic domain, and store (r, id_b) in the binary domain.
- On input (ltc, $\llbracket u \rrbracket_2, c, \text{id}_b$) from all parties, where $\llbracket u \rrbracket_2$ refers to a stored binary-domain string $u \in \mathbb{F}_2^\ell$ and $c \in \mathbb{F}_2^\ell$ is a public constant (both interpreted as unsigned ℓ -bit integers): set $b \leftarrow 1$ if $u < c$ and $b \leftarrow 0$ otherwise, and store (b, id_b) in the binary domain.

Functionality 3: $\mathcal{F}_{\text{Prep}}$

Supports all commands of $\mathcal{F}_{\text{eABB}}$. In what follows, let $\mathcal{D}^{(0)}, \mathcal{D}^{(1)}$ denote the two coordinate-wise RCDT samplers used in HAWK (see Section 2.4), and assume the secret-key polynomials $f, g \in \mathcal{R}$ have been previously stored in the arithmetic domain (e.g., by key-gen). Set $\mathbf{M} := [-\text{rot}(g) \mid \text{rot}(f)] \in \mathbb{Z}_q^{n \times 2n}$.

- On input (prep, $\text{id}_{w_0}, \text{id}_V, \text{id}_r$) from all parties:
 1. Fetch f, g from the arithmetic domain.

2. Sample $\text{salt} \xleftarrow{\$} \{0, 1\}^\ell$, $\mathbf{x}^{(0)} \xleftarrow{\$} (\mathcal{D}^{(0)})^{2n}$, $\mathbf{x}^{(1)} \xleftarrow{\$} (\mathcal{D}^{(1)})^{2n}$, and $\mathbf{r} \xleftarrow{\$} \{0, 1\}^{2n}$.
3. Set $\Delta \mathbf{x} \leftarrow \mathbf{x}^{(1)} - \mathbf{x}^{(0)}$, $\mathbf{s}_0 \leftarrow \mathbf{x}^{(0)} + \mathbf{r} \odot \Delta \mathbf{x}$, and $\mathbf{s}_1 \leftarrow (\mathbf{1} - 2\mathbf{r}) \odot \Delta \mathbf{x}$.
4. Compute $\mathbf{w}_0 \leftarrow \mathbf{M} \cdot \mathbf{s}_0 \in \mathbb{Z}_q^n$ and the matrix $\mathbf{V} \in \mathbb{Z}_q^{n \times 2n}$ whose (j, i) -entry is $\mathbf{V}_{j,i} \leftarrow \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i$.
5. Store $(\mathbf{w}_0, \text{id}_{w_0})$ and $(\mathbf{V}, \text{id}_V)$ in the arithmetic domain, and $(\mathbf{r}, \text{id}_r)$ in the binary domain.
6. Send salt to all parties.

Each call samples fresh randomness and writes to the handles supplied by the caller, so distinct calls produce independent preprocessing and the functionality keeps no state of its own.

Functionality 4: $\mathcal{F}_{\text{Sign}}$

The functionality maintains a one-shot flag initialized $\in \{\text{true}, \text{false}\}$ (initially false), the HAWK signing key $\text{sk} = (\mathbf{B}, \text{hpub})$ and verification key $\text{vk} = (\mathbf{Q}, \text{hpub})$ (populated by key-gen), and a queue Q of unused salts (initially empty).

On input (key-gen) from all parties:

1. If initialized = true, abort.
2. Run HAWK.KeyGen() (Algorithm 1) to obtain $(\text{vk}, \text{sk}) = (\mathbf{Q}, (\mathbf{B}, \text{hpub}))$ with $\mathbf{B} = \begin{pmatrix} f & F \\ g & G \end{pmatrix}$ and $f, g, F, G \in \mathcal{R}$.
3. Store sk and vk internally.
4. Set initialized $\leftarrow$ true and send vk to all parties.

On input (sample-salt) from all parties:

1. Sample $\text{salt} \xleftarrow{\$} \{0, 1\}^\ell$ and append salt to Q .
2. Send salt to all parties.

On input (sign, m) from all parties:

1. If Q is empty, abort.
2. Remove the first element salt from Q .
3. Run HAWK.SignDet(sk, m , salt) (Algorithm 1), obtaining a signature σ together with the pre-sym-break polynomial w_1 used internally.
4. Send (σ, w_1) to all parties.

D Generating daBits

A daBit is a pair $(\llbracket r \rrbracket, \llbracket r \rrbracket_2)$ holding the same bit r in the arithmetic and in the binary domain. We recall here how the (daBit) command of $\mathcal{F}_{\text{eABB}}$ is instantiated, which depends on the arithmetic domain; the cost of this generation is excluded from the counts of Section 5.5, as discussed there.

Power-of-two modulus. When $q = 2^k$, reduction modulo 2 is a ring homomorphism and hence local on shares. Over $\mathbb{Z}_{2^k}$ with replicated or additive sharing, a daBit is obtained by having the parties sample a shared random bit and reduce it locally, the reduction mapping the arithmetic sharing onto a binary sharing of the same bit. For Shamir sharing the arithmetic domain becomes the Galois ring $\text{GR}(2^k, \kappa)$ with $2^\kappa > N$ [1], whose reduction modulo 2 lands in $\mathbb{F}_{2^\kappa}$ — exactly the extension the binary domain already uses — so the same local argument applies. See also Remark 5.

Prime modulus. When q is prime the two domains are unrelated and a dedicated cross-domain protocol is required. daBits were introduced by Rotaru and Wood [38]: each party inputs a random bit in both domains, proves consistency, and the bits are combined by XOR so that a single honest party suffices to randomize the result; a cut-and-choose step removes inconsistent contributions in the malicious setting. Escudero et al. [23] give the more efficient edaBit variant, which produces a whole shared k -bit value together with the sharings of its bits at amortized cost well below k daBits, and is what a concrete instantiation should use in practice. We refer to these works for the protocols and their security analysis, and to [18] for instantiations whose arithmetic domain is $\mathbb{Z}_{2^k}$.

E Deferred Proofs

Theorem 1. *Protocol Π_{Prep} UC-realizes the functionality $\mathcal{F}_{\text{Prep}}$ in the $\mathcal{F}_{\text{eABB}}$ -hybrid model against any static adversary corrupting an arbitrary strict subset of the parties.*

Proof. Let $\mathcal{A}$ be a static adversary corrupting a strict subset $\mathcal{C}$ of the parties before the protocol begins, and let $\mathcal{Z}$ be a probabilistic polynomial-time environment. We describe a simulator Sim that, by interacting with the ideal functionality $\mathcal{F}_{\text{Prep}}$ and emulating $\mathcal{F}_{\text{eABB}}$ towards $\mathcal{A}$, produces a view that is identically distributed to the view of $\mathcal{A}$ in the real-world execution of Π_{Prep} in the $\mathcal{F}_{\text{eABB}}$ -hybrid model. This implies that no environment can distinguish the two executions, which is the UC-realization requirement.

Where the adversary's information comes from. In the $\mathcal{F}_{\text{eABB}}$ -hybrid model, parties communicate only with $\mathcal{F}_{\text{eABB}}$; there is no peer-to-peer communication. Consequently, the entire view of $\mathcal{A}$ during the execution of Π_{Prep} consists of the messages that $\mathcal{F}_{\text{eABB}}$ delivers to the parties in $\mathcal{C}$. Inspecting the command interface of $\mathcal{F}_{\text{eABB}}$ (Section 2.6.2 and the ABB-style commands inherited from $\mathcal{F}_{\text{KG}}$ in Section 2.6.1, with the full descriptions in Section C), only two commands produce party-visible messages: `open`, which broadcasts the stored value, and `coin`, which broadcasts a freshly sampled value. Every other command — `input`, `add`, `addmult-cons`, `mult`, `rand`, `mod2`, `key-gen`, `daBit`, and `lrc` — only updates $\mathcal{F}_{\text{eABB}}$'s internal store and produces no message visible to any party. To define Sim it therefore suffices to identify every open and

coin call issued by Π_{Prep} , and to show that Sim can deliver matching distributions for each.

Enumerating the open and coin calls of Π_{Prep} . We trace the protocol step by step.

Step 1: $\text{salt} \leftarrow \text{coin}(\{0, 1\}^\ell)$. One coin call; the output is a uniformly random ℓ -bit string broadcast to all parties.

Step 2: $(\llbracket \mathbf{x}^{(0)} \rrbracket, \llbracket \mathbf{x}^{(1)} \rrbracket) \leftarrow \pi_{\text{Gaussian}}$. Procedure π_{Gaussian} invokes, for each of the $2n$ coordinates, two instances of π_{HalfGS} (with tables T_0 and T_1) and two instances of π_{RndFlip} , for a total of $4n$ invocations of π_{HalfGS} and $4n$ invocations of π_{RndFlip} . π_{RndFlip} uses one daBit and one mult, neither of which is party-visible. π_{HalfGS} , on the other hand, contains the only opening of the protocol: in step 3.b of π_{HalfGS} , each of the 13 bits $d_i \leftarrow \text{open}(\llbracket b_i \rrbracket_2 \oplus \llbracket r_i \rrbracket_2)$, $i \in \{0, \dots, 12\}$, is opened to all parties. Across the $4n$ invocations of π_{HalfGS} this produces $52n$ opened bits. All other commands within π_{Gaussian} are non-visible.

Steps 3–9 of Π_{Prep} . Step 3 is a local subtraction. Step 4 issues $2n$ daBit calls (silent). Step 5 is a Hadamard product realized as $2n$ mult calls (silent). Step 6 is a local addition and subtraction. Step 7 issues two polynomial multiplications, each consisting of n^2 mult calls (silent), followed by a local subtraction. Step 8 derives the entries of $\llbracket \mathbf{M} \rrbracket$ by local signed permutations of the stored $\llbracket f \rrbracket, \llbracket g \rrbracket$ (silent), then issues $2n^2$ mult calls to populate $\llbracket \mathbf{V} \rrbracket$ (silent). Step 9 outputs handles — no values are revealed.

In total, Π_{Prep} issues exactly one coin call (step 1) and exactly $52n$ open calls (inside π_{Gaussian}); every other operation leaves results in $\mathcal{F}_{\text{eABB}}$'s store and produces no party-visible message.

The simulator. Sim proceeds as follows.

1. Whenever $\mathcal{A}$ (controlling the corrupt parties in $\mathcal{C}$) triggers the start of Π_{Prep} by joining the protocol's first command, Sim forwards the corresponding prep call to $\mathcal{F}_{\text{Prep}}$ on behalf of the corrupt parties (using the public output handles $\text{id}_{\mathbf{w}_0}, \text{id}_{\mathbf{V}}, \text{id}_{\mathbf{r}}$ of the protocol). $\mathcal{F}_{\text{Prep}}$ samples its own salt, $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{r}$, computes $\mathbf{w}_0, \mathbf{V}$, stores the latter, and returns salt to all parties; Sim records salt.
2. Sim emulates the call to `coin` in step 1 of Π_{Prep} by delivering exactly the salt it just received from $\mathcal{F}_{\text{Prep}}$ to $\mathcal{A}$ as the result of the coin broadcast.
3. Sim emulates every other $\mathcal{F}_{\text{eABB}}$ command of Π_{Prep} in its internally simulated copy of $\mathcal{F}_{\text{eABB}}$. Since none of these commands generate party-visible messages, the only thing Sim must produce is internal consistency of its emulated state, which is straightforward: Sim maintains its own handle dictionary and updates it according to the (deterministic, non-revealing) semantics of each $\mathcal{F}_{\text{eABB}}$ command.
4. For each of the $52n$ open calls inside the π_{HalfGS} invocations, Sim samples a fresh uniform bit $\tilde{d}_i \xleftarrow{\$} \{0, 1\}$,

independently across the $52n$ openings, and delivers $\tilde{d}_i$ to $\mathcal{A}$ as the broadcast result of the corresponding open.

Indistinguishability. We compare the view of $\mathcal{A}$ in the two worlds.

Distribution of salt. In the real world, $\text{salt} \in \{0, 1\}^\ell$ is sampled uniformly by $\mathcal{F}_{\text{eABB}}$ inside the coin call. In the ideal world, $\mathcal{F}_{\text{Prep}}$ samples salt uniformly in step 2 of its prep command (Section 3.1) and Sim forwards exactly this value. Both distributions are uniform on $\{0, 1\}^\ell$; they coincide exactly.

Distribution of the $52n$ opened bits. Consider a single invocation of π_{HalfGS} . In the real world, π_{HalfGS} samples $\llbracket u \rrbracket_2 \in \mathbb{F}_2^{78}$ via 78 independent rand calls, and obtains 13 binary-domain stored bits $\llbracket b_i \rrbracket_2$ via $\text{lrc}(\llbracket u \rrbracket_2, T[i])$. Each b_i is a deterministic function of the (secret) u and the public threshold $T[i]$. For each $i \in \{0, \dots, 12\}$, an independent daBit call produces a uniformly random bit r_i , and d_i is opened as $d_i = b_i \oplus r_i$.

Conditioned on any fixed value of u (equivalently, any fixed values of $b_0, \dots, b_{12}$), the bits $r_0, \dots, r_{12}$ are mutually independent and uniform on $\{0, 1\}$, so the bits $d_0 = b_0 \oplus r_0, \dots, d_{12} = b_{12} \oplus r_{12}$ are also mutually independent and uniform. Marginalizing over u leaves the same distribution: the 13 opened bits of one π_{HalfGS} invocation are i.i.d. uniform on $\{0, 1\}$, independently of u . Across the $4n$ invocations of π_{HalfGS} , each invocation uses fresh independent r -bits and fresh independent u -strings, so all $52n$ opened bits are mutually independent and uniform on $\{0, 1\}$.

In the ideal world, Sim samples $\tilde{d}_i$ independently and uniformly on $\{0, 1\}$ for each of the $52n$ openings. The joint distribution of the $52n$ simulated bits is therefore exactly the same as the joint distribution of the $52n$ real-world opened bits.

Joint distribution. In the real world, salt is sampled by an $\mathcal{F}_{\text{eABB}}$ call (step 1 of Π_{Prep}) that is independent of the calls that produce u and r_i (steps inside π_{Gaussian}). In the ideal world, $\mathcal{F}_{\text{Prep}}$ samples salt independently of the $\tilde{d}_i$ that Sim samples afterwards. Hence in both worlds the salt is independent of the $52n$ opened bits, and the joint view of $\mathcal{A}$ is the uniform distribution on

$$\{0, 1\}^\ell \times \{0, 1\}^{52n}.$$

Stored values are never revealed. No $\mathcal{F}_{\text{eABB}}$ command is issued by Π_{Prep} after step 9 reveals any of the stored values $\mathbf{w}_0, \mathbf{V}, \mathbf{r}$. In the real world these values satisfy

$$\mathbf{w}_0 = \mathbf{M} \cdot \mathbf{s}_0, \quad \mathbf{V}_{j,i} = \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i,$$

with $\mathbf{s}_0 = \mathbf{x}^{(0)} + \mathbf{r} \odot \Delta \mathbf{x}$, $\mathbf{s}_1 = (\mathbf{1} - 2\mathbf{r}) \odot \Delta \mathbf{x}$, $\Delta \mathbf{x} = \mathbf{x}^{(1)} - \mathbf{x}^{(0)}$, and $\mathbf{x}^{(0)} \sim (\mathcal{D}^{(0)})^{2n}$, $\mathbf{x}^{(1)} \sim (\mathcal{D}^{(1)})^{2n}$, $\mathbf{r} \sim \{0, 1\}^{2n}$ as sampled inside π_{Gaussian} and $\mathcal{F}_{\text{eABB}}$.daBit. In the ideal world, $\mathcal{F}_{\text{Prep}}$ samples its own $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{r}$ from the same distributions and stores values defined by the same formulas (steps 3 and 4 of $\mathcal{F}_{\text{Prep}}$). Since these stored values are never opened during Π_{Prep} (in either world), they contribute nothing to $\mathcal{A}$'s view.

Conclusion. $\mathcal{A}$'s view in the real-world execution of Π_{Prep} in the $\mathcal{F}_{\text{eABB}}$ -hybrid model and $\mathcal{A}$'s view in the ideal-world execution mediated by Sim are both distributed as the uniform distribution on $\{0, 1\}^\ell \times \{0, 1\}^{52n}$. These two views are therefore identically distributed, and no environment $\mathcal{Z}$ can distinguish them. This concludes the proof. $\square$

Theorem 2. *Protocol Π_{Sign} UC-realizes the functionality $\mathcal{F}_{\text{Sign}}$ in the $\mathcal{F}_{\text{Prep}}$ -hybrid model against any static adversary corrupting an arbitrary strict subset of the parties.*

Proof. In the $\mathcal{F}_{\text{Prep}}$ -hybrid model the parties interact only with $\mathcal{F}_{\text{Prep}}$, and the only commands issued by Π_{Sign} that produce output visible to the parties are the two openings of steps 3 and 5; every other step is local. It therefore suffices to exhibit a simulator for those two values.

For correctness, recall from Section 3.1 that $\mathbf{w}_0 = \mathbf{M} \cdot \mathbf{s}_0$ and $\mathbf{V}_{j,i} = \mathbf{M}_{j,i} \cdot (\mathbf{s}_1)_i$, where $\mathbf{s}_0 = \mathbf{x}^{(0)} + \mathbf{r} \odot \Delta \mathbf{x}$ and $\mathbf{s}_1 = (\mathbf{1} - 2\mathbf{r}) \odot \Delta \mathbf{x}$. Since $\mathbf{c} = \mathbf{t} \oplus \mathbf{r}$ gives $\mathbf{t} = \mathbf{c} + (\mathbf{1} - 2\mathbf{c}) \odot \mathbf{r}$ over the integers, we obtain $\mathbf{s}_0 + \mathbf{c} \odot \mathbf{s}_1 = \mathbf{x}^{(0)} + \mathbf{t} \odot \Delta \mathbf{x}$, so the value opened in step 5 is $\mathbf{w} = \mathbf{M} \cdot (\mathbf{x}^{(0)} + \mathbf{t} \odot \Delta \mathbf{x})$, which is precisely the polynomial w_1 computed by HAWK.SignDet on input (m, salt) . Moreover $\mathbf{x}^{(0)}$ and $\mathbf{x}^{(1)}$ are sampled by $\mathcal{F}_{\text{Prep}}$ independently of $\mathbf{t}$, so the i -th coordinate of $\mathbf{x}^{(0)} + \mathbf{t} \odot \Delta \mathbf{x}$ is distributed according to $\mathcal{D}^{(i)}$. The Gaussian vector implicitly used by Π_{Sign} is therefore distributed exactly as the one sampled inside SignDet, even though it was prepared before m was known.

The simulator Sim forwards the key-gen command to $\mathcal{F}_{\text{Sign}}$ and delivers the resulting vk to $\mathcal{A}$, and forwards the j -th (prep) call as the j -th (sample-salt) command, delivering the salt it receives. On a signing call with message m it sends (sign, m) to $\mathcal{F}_{\text{Sign}}$, obtains (σ, w_1) , samples $\tilde{\mathbf{c}} \xleftarrow{\$} \mathbb{F}_2^{2n}$, and delivers $\tilde{\mathbf{c}}$ and w_1 as the outputs of the two openings. In the real execution $\mathbf{c} = \mathbf{t} \oplus \mathbf{r}$ with $\mathbf{r}$ uniform and independent of everything else, so $\mathbf{c}$ is uniformly distributed, and by the argument above the second opening equals w_1 with the same distribution in both worlds. Finally, step 1 takes the first unused preprocessing tuple, so each tuple is used in at most one execution and the salts are consumed in the order in which they were produced, matching the queue discipline of $\mathcal{F}_{\text{Sign}}$. The two views are therefore identically distributed, and no environment can distinguish the real execution from the ideal one. $\square$

F Security of Explicit-salt HAWK

We provide an analysis of the security of the explicit-salt variant of HAWK, where the signing algorithm is replaced with SignDet. Toward this, we first need to argue the distance between the distribution of $\mathbf{x}$ sampled from the discrete Gaussian distribution and that from RCDT sampling. This can be evaluated by following the same analysis using Rényi divergence in [9, Sec. 6.5]. Consequently, we can focus on a

variant of the explicit-salt scheme that uses the discrete Gaussian distribution instead of RCDT sampling. We denote the signing algorithm of this variant by $\text{SignDet}'$. To conclude this analysis, we prove that the explicit-salt variant instantiated with $\text{SignDet}'$ is strongly unforgeable under chosen message attack (SUF-CMA). Following the proof strategy of [3] and leveraging the well-known proof technique of the Full-Domain Hash by Coron [15], we show that the explicit-salt variant is SUF-CMA if the HAWK signature scheme, that uses discrete Gaussian sampling, is SUF-CMA.

Theorem 3. *The HAWK scheme instantiated with $\text{SignDet}'$ is SUF-CMA in the random oracle model if the standard HAWK is SUF-CMA. Formally, if there exists an adversary $\mathcal{A}$ against SUF-CMA of the explicit-salt HAWK scheme, making at most q_h and q_s queries to the random oracle H and the signing oracle, respectively, there exists an adversary $\mathcal{B}$ against SUF-CMA of HAWK such that*

$$\epsilon_{\mathcal{A}} \leq 4(q_s + 1) \cdot \epsilon_{\mathcal{B}} + q_s \cdot \exp(-\Theta(n)) \quad (1)$$

where $\epsilon_{\mathcal{A}}$ and $\epsilon_{\mathcal{B}}$ are the winning probabilities of $\mathcal{A}$ and $\mathcal{B}$ in the SUF-CMA game.

Proof. We construct $\mathcal{B}$ that takes as input $\text{vk} = \mathcal{Q}$ and internally runs $\mathcal{A}$ by giving vk and simulating the random oracle and the signing oracle. Let H and O_S be the random oracle and the signing oracle for the standard HAWK, and H' and O'_S be the random oracle and the signing oracle simulated by $\mathcal{B}$ for the explicit-salt HAWK scheme. $\mathcal{B}$ has a biased coin b which becomes $b = 1$ with probability $p = q_s / (q_s + 1)$ and manages a list L and a set Q_m . $\mathcal{B}$ simulates the random oracle and the signing oracle as follows:

Random Oracle H' . If the query is not of the form $(m \parallel \text{salt})$, it forwards it to H and returns the response. If $L[m, \text{salt}] \neq \perp$, it returns $\mathbf{h}$ where $L[m, \text{salt}] = (\mathbf{h}, \sigma', b)$. Otherwise, it tosses the coin b . If $b = 1$, it first queries $m \parallel \text{salt}$ to O_S and obtains $\sigma' = (s'_1, \text{salt}')$. It obtains $\mathbf{h} \leftarrow H(m \parallel \text{salt} \parallel \text{salt}')$ by accessing H , stores $L[m, \text{salt}] = (\mathbf{h}, \sigma', 1)$. If $b = 0$, it obtains $\mathbf{h} \leftarrow H(m \parallel \text{salt} \parallel \mathbf{0})$ by accessing H , and stores $L[m, \text{salt}] = (\mathbf{h}, \perp, 0)$. Finally, it returns $\mathbf{h}$.

Signing oracle O'_S . If the query is not of the form (m, salt) or $(m, (\cdot, \text{salt})) \in Q_m$, it returns $\perp$. If $L[m, \text{salt}] = \perp$, then it makes the hash query $(m \parallel \text{salt})$ by itself. If $L[m, \text{salt}] = (\cdot, \cdot, 0)$, then $\mathcal{B}$ aborts. Otherwise, it parses $(\mathbf{h}, \sigma' = (s'_1, \text{salt}'), 1) \leftarrow L[m, \text{salt}]$, computes $w_1 \leftarrow h_1 - 2s_1$, and flips the sign of w_1 with probability $1/2$, where $\mathbf{h} = (h_0, h_1)$. Then it updates $Q_m \leftarrow Q_m \cup \{(m, (s_1, \text{salt}))\}$ and returns $\sigma = (s'_1, \text{salt})$ and w_1 .

After $\mathcal{A}$ terminates with an output $(m^*, \sigma^* = (s_1^*, \text{salt}^*))$, $\mathcal{B}$ first checks its validity and whether $(m^*, (s_1^*, \text{salt}^*)) \notin Q_m$. If either check fails, $\mathcal{B}$ aborts. Without loss of generality, we suppose that $L[m^*, \text{salt}^*] \neq \perp$ when $\mathcal{A}$'s output passes the check. It parses $(\mathbf{h}, \sigma' = (s'_1, \text{salt}'), b) \leftarrow L[m^*, \text{salt}^*]$. In the case of $(m^*, (\cdot, \text{salt}^*)) \notin Q_m$, if $b = 1$, then $\mathcal{B}$ aborts. If not,

it sets $\widetilde{\text{salt}} = \mathbf{0}$. Otherwise (namely, $(m^*, (\cdot, \text{salt}^*)) \in Q_m$), it sets $\widetilde{\text{salt}} = \text{salt}'$. It eventually outputs $((m^* \parallel \text{salt}^*), (s_1^*, \widetilde{\text{salt}}))$ as a forgery.

Analysis of Advantage We first analyze the distribution of $\mathcal{A}$'s view. Since $\mathcal{B}$ returns the output of H in the random oracle simulation, the distribution of responses of H' is correct. For the signing oracle, there is a subtle gap between the distribution of signatures. Concretely, while the signing algorithm of the standard HAWK repeats until a valid signature is generated, the signing algorithm of the explicit-salt HAWK terminates with $\perp$ if $\mathbf{x}$ is too large. By following the same argument as in [22, Proof of Lemma 10] leveraging the tail bound of the Gaussian distribution, the probability that the signing algorithm returns $\perp$ is upper bounded by $\exp(-\Theta(n))$. Due to the union bound, the probability that the signing algorithm aborts at least once is upper bounded by $q_s \cdot \exp(-\Theta(n))$. O_S also returns w_1 which is a value before applying sym-break. Since $\mathbf{w}$ follows the Gaussian distribution under $\mathcal{Q}$ -norm (see [9, 22] for details), the sign of the first non-zero coefficient of w_1 is positive with probability $1/2$. Thus the distribution of w_1 remains identical. Also, since $\mathcal{A}$ is not allowed to make the same signing query (m, salt) twice, O'_S always returns a signature which is not opened yet. Let $\epsilon'_{\mathcal{A}}$ be the winning probability conditioned on the signing algorithm never aborting. Due to the above argument, we have $\epsilon_{\mathcal{A}} \leq \epsilon'_{\mathcal{A}} + q_s \cdot \exp(-\Theta(n))$.

Next, we show that $\mathcal{B}$'s output satisfies the winning condition in SUF-CMA of the standard HAWK. Since we have $H'(m^* \parallel \text{salt}^*) = H(m^* \parallel \text{salt}^* \parallel \widetilde{\text{salt}})$ due to the construction of $\mathcal{B}$, if $(m^*, \sigma^* = (s_1^*, \text{salt}^*))$ passes the verification, $((m^* \parallel \text{salt}^*), (s_1^*, \widetilde{\text{salt}}))$ also passes the verification. Moreover, there are two cases: (i) a pair (m^*, salt^*) does not appear in the signing query, i.e., $(m^*, (\cdot, \text{salt}^*)) \notin Q_m$, and (ii) a pair (m^*, salt^*) appears in the signing query, i.e., $(m^*, (\cdot, \text{salt}^*)) \in Q_m$. In case (i), due to the abort condition, we have $b = 0$. Then since $(m^* \parallel \text{salt}^*)$ has never been queried to O_S , $\mathcal{B}$'s output is a non-trivial forgery. In case (ii), since $(m^*, (\cdot, \text{salt}^*)) \in Q_m$, $(m^* \parallel \text{salt}^*)$ has been queried to O_S which means $L[m^*, \text{salt}^*] = (\mathbf{h}, \sigma' = (s'_1, \text{salt}'), 1)$. Due to $(m^*, (\cdot, \text{salt}^*)) \in Q_m$ and $(m^*, (s_1^*, \text{salt}^*)) \notin Q_m$, we obtain $s'_1 \neq s_1^*$. Then we also have $(s_1^*, \widetilde{\text{salt}}) \neq (s'_1, \text{salt}')$. Therefore, $\mathcal{B}$'s output is a non-trivial forgery.

Finally, we analyze $\epsilon'_{\mathcal{A}}$ by following the same analysis in [3]. $\mathcal{B}$ aborts unless (I) $b = 1$ holds in all signing queries, (II) $b = 0$ holds for $\mathcal{A}$'s output in the case of $(m^*, (\cdot, \text{salt}^*)) \notin Q_m$, and (III) $\mathcal{A}$ outputs a valid forgery. Since the coins are independently sampled and $\mathcal{A}$ cannot see coins unless $\mathcal{B}$ terminates, we have $\epsilon_{\mathcal{B}} \geq \epsilon'_{\mathcal{A}} \cdot p^{q_s} \cdot (1 - p)$. By substituting $p = q_s / (q_s + 1)$, using the fact that $(1 - 1/a)^a \geq 1/4$ for $a \geq 2$, and combining all above arguments, we have

$$\epsilon_{\mathcal{A}} \leq 4(q_s + 1) \cdot \epsilon_{\mathcal{B}} + q_s \cdot \exp(-\Theta(n)).$$

□

Remark 6 (Reduction Loss). The security bound above has two reduction losses: a multiplicative loss of $4(q_s + 1)$ and an additive loss of $q_s \cdot \exp(-\Theta(n))$. For the first one, we could replace this loss with another additive loss by adopting another proof strategy. Specifically, if there are sufficiently many valid w_1 for the target $\mathbf{h}$, we can bound the probability that $\mathcal{A}$ outputs the same signature s_1 as the one received from O_S based on its entropy. Then the reduction no longer needs to abort the game in the signing oracle since it can always assign $\mathbf{h}$ as in the case of $b = 1$. Regarding the second loss, for the HAWK-1024 parameters the abort probability of a single signing attempt is bounded by 2^{-121} [9], so with $q_s \leq 2^{64}$ the additive term is at most 2^{-57} , and the bound above is already meaningful. For parameter sets where this term is non-negligible, it can be removed via an alternative proof strategy that proves security directly under an average-case one-more (approximate) shortest vector problem omSVP [9, 22]. In the game of omSVP, the sample oracle returns $\mathbf{w}$ sampled from the Gaussian distribution under Q -norm without performing a norm check. Consequently, we no longer have to account for the gap between distributions arising from the norm check. Instead, this approach requires additional analysis of the distributions as in [9].