

One-Round Threshold XMSS and SPHINCS⁺ from Fully Homomorphic Encryption

Jung Hee Cheon^{1,2}, Kevin Choi¹, Hyeoncheol Joo¹, Taeseong Kim¹

¹ Seoul National University, Seoul, Republic of Korea

² CryptoLab Inc., Seoul, Republic of Korea

{jhcheon, kc2853, wnguscjf01, kts1023}@snu.ac.kr

Abstract. While hash-based signatures such as XMSS and SPHINCS⁺ are well-established and play a key role in post-quantum cryptography, thresholdizing them to enable use in distributed systems remains a challenge. In particular, their lack of exploitable structure prevents any algebraic attempts that are otherwise possible in the context of classical signatures such as ECDSA, Schnorr, RSA, or BLS, whereas general-purpose MPC techniques are costly in round complexity.

In this work, we construct and implement the first one-round threshold protocol for XMSS and SPHINCS⁺, with concrete parameters yielding full backward compatibility. At a high level, the secret key is shared among the signing parties while the signing algorithm is evaluated homomorphically using a threshold fully homomorphic encryption scheme. Any t -of- n parties, via threshold decryption, can collaboratively derive the signature, which is accepted by the unmodified RFC 8391 or FIPS 205 verifier.

At a technical level, we develop efficient homomorphic SHAKE evaluation over discrete CKKS for varying batch sizes. Analyzing XMSS and SPHINCS⁺ signing reveals parallel hash computations and data-dependent selections, which we handle through batching and homomorphic lookup to obtain end-to-end circuits. Our implementation amortizes SHAKE256/256 evaluation to 3.4 ms per hash and completes online threshold XMSS signing in 0.69 s. Threshold SPHINCS⁺ has roughly two orders of magnitude higher signing latency but over an order of magnitude lower setup latency than XMSS.

Keywords: Threshold signatures · Hash-based signatures · XMSS · SPHINCS⁺ · Threshold fully homomorphic encryption · CKKS

Table of Contents

1	Introduction	4
1.1	Our Contributions	4
1.2	Related Work	5
2	Preliminaries	6
2.1	System Model and Notation	6
2.2	Threshold Signature	6
2.3	Hash-Based Signatures	7
2.4	Threshold Fully Homomorphic Encryption	7
2.5	Keccak Permutation	8
3	Efficient Keccak Permutation in CKKS	8
3.1	Keccak State Packing	8
3.2	Homomorphic Evaluation Strategy	9
4	Round-Optimal Threshold Signatures	11
4.1	Threshold Signature Setup	12
4.2	Homomorphic Lookup for Hash-Based Signing	13
4.3	Threshold Signing Protocol	14
5	Security	14
6	Experimental Evaluation	15
6.1	Keccak Evaluation	16
6.2	Threshold Signing	16
7	Discussion	18
A	Threshold Signature Definitions and Blueprint	21
A.1	One-Round Threshold Signing Blueprint	22
A.2	Unforgeability	22
B	Comparison of Threshold Hash-Based Signatures	22
C	Additional Details on CKKS	23
D	Threshold Fully Homomorphic Encryption	25
E	Parameters	25
E.1	Signature Parameter Sets	25
E.2	Detailed CKKS Parameters	26
E.3	Threshold Parameters Analysis	26
F	Additional Details on Homomorphic Keccak	27
F.1	Counting Homomorphic Operations	29
F.2	Breakdown of Keccak Permutation	29
F.3	Message Magnitude and Error Analysis for Homomorphic Keccak Permutation	30
F.4	Key-Switching Comparison	30
F.5	Latency Comparison	31
G	XMSS	31
G.1	Hash Instantiation and Notation	32
G.2	WOTS ⁺	33

G.3	Prefixes and Bitmasks	34
G.4	Many-Time via Merkle Tree	35
G.5	Parameter Sets and Instantiation with SHAKE256/256	35
H	SPHINCS ⁺	36
H.1	WOTS ⁺ in SPHINCS ⁺	36
H.2	FORS	36
H.3	Hypertree	36
H.4	Tweakable Hash and the Simple/Robust Variants	38
H.5	Signing	38
I	Joint Signing Key Generation	38
J	Batched Public-Key Generation	38
K	Homomorphic WOTS ⁺ Signing	40
L	Authentication-Path Selection	43
M	Full algorithm for signing	43
N	Permutation Depth and Width	45
O	Threshold Signature Latency	47
P	Simulation Security of the Signing Protocol	47
Q	Discussion on Malicious Security	52

1 Introduction

A t -of- n threshold signature scheme distributes a signing key among n parties so that at least t parties are needed to collaboratively produce a valid signature. Such schemes are an essential building block in distributed systems, e.g. for facilitating consensus protocols [62, 64], safeguarding distributed validator protocols [27] in proof-of-stake blockchains, enabling applications requiring a secure financial custody [24], etc. It is widely known that the NIST Call for Multi-Party Threshold Schemes [14] is part of an ongoing standardization effort.

Various ecosystems [16, 28, 50, 63] are migrating towards cryptography based on post-quantum hardness assumptions. Unlike lattice-based schemes like ML-DSA [53], hash-based schemes such as XMSS (standardized in RFC 8391 [34] and approved by NIST SP 800-208 [20]) and SPHINCS⁺ [9, 54] derive security solely from the underlying hash function, which poses a challenge for thresholdization.

In fact, thresholdizing a hash-based signature remains an open problem [46, 65], which we tackle as follows. The concept of a universal thresholdizer, introduced by Boneh et al. [12], uses threshold fully homomorphic encryption (FHE) to add threshold functionality to *any* cryptographic scheme. Namely, the parties first share the threshold FHE key via distributed key generation (DKG) [4, 30, 51, 59], and then a separate secret key (e.g. for signing) is derived in a ciphertext form encrypted under that key.

As we opt to evaluate the signing algorithm homomorphically with the encrypted signing key and a plaintext message, the resulting ciphertext is one that encrypts a valid signature, and it is decrypted only through the cooperation of at least t parties, the process of which effectively subsumes the role of a threshold signature. This idea is general in theory. However, instantiating it efficiently for a concrete scheme requires designing a practical homomorphic evaluation framework of the scheme’s core operations.

In this work, we concretely apply the universal thresholdizer paradigm to hash-based signatures. Their evaluation costs alone are substantial: under our parameter sets, XMSS setup (with 2^{10} signing leaves) requires over one million hash evaluations, while SPHINCS⁺ signing requires more than 300,000. To realize threshold signing under this computational load, we develop efficient homomorphic SHAKE256 evaluation over a discrete variant of CKKS [7, 8, 18] and integrate it with homomorphic circuits for the remaining signing operations to obtain end-to-end threshold signing protocols.

We note that homomorphic SHAKE evaluation is not specific to hash-based signatures: it can also serve to build one-round threshold signatures from schemes such as Schnorr [60] and ML-DSA [53].

1.1 Our Contributions

- **High-throughput, low-latency Keccak evaluation.** We improve both the throughput and latency of homomorphic Keccak- f [1600], the permutation underlying SHAKE256, through the following optimizations.
 - *High-throughput.* We achieve $8\times$ the throughput of the state of the art [61]. Our bootstrapping schedule halves the required count by refreshing ciphertexts once every two Keccak rounds, with correctness established through message- and error-growth

Table 1. Comparison of homomorphic Keccak evaluation methods.

Scheme	Method	Latency (s)	Amor. time (s)	HW
FHEW / CGGI	Hong et al. [33]*	5 201	5 201	CPU
	Bon et al. [11]*	528	528	CPU
CKKS	Seuré and Suvanto [61]*	68	8.5	CPU
	Seuré and Suvanto [61] [†]	0.36	0.045	GPU
	Ours (Low-latency)	0.198	0.025	GPU
	Ours (High-throughput)	1.75	0.003	GPU

* Results reported in the original papers; hardware configurations differ from ours.

[†] Reproduced from the implementation of Seuré and Suvanto on the identical environment. Refer to the details Appendix F.5.

analysis. We further optimize bootstrapping to reduce its amortized cost across multiple ciphertexts.

- *Low-latency.* We achieve a 1.8× speedup over the state of the art [61]. Our new packing places an entire Keccak state in a single ciphertext and enables evaluation of slot permutations as homomorphic matrix–vector products. Applying BSGS and fusing the required slot alignment into adjacent Keccak steps reduces the key-switching count by a factor of 3.2.
- **Round-optimal threshold signatures.** We are the first to construct end-to-end homomorphic signing protocols for XMSS and SPHINCS⁺, with efficient homomorphic lookup circuits for selecting encrypted WOTS⁺ chain states and authentication paths. Only the final signature is produced through a single round of threshold decryption. The resulting round-optimal protocols output standard XMSS and SPHINCS⁺ signatures, requiring no modification to their verification procedures. Concretely, our threshold XMSS implementation produces a signature in 0.69 s, well within Ethereum’s 12-second block time.

1.2 Related Work

Homomorphic evaluation of SHA3. Hong et al. [33] introduced overlapped bootstrapping for TFHE-based Keccak, while Bon et al. [11] proposed p -encoding to evaluate multi-input Boolean functions in a single bootstrapping. Both use bit-wise representations requiring 1,600 ciphertexts per Keccak- f [1600] state, without SIMD amortization across states. Seuré and Suvanto [61] evaluated SHA3 over discrete CKKS [7] using XBOOT [55] and lane-wise packing, reporting 68 s for eight states on a single-threaded CPU. Our work targets both high-throughput and low-latency, extending homomorphic Keccak evaluation to end-to-end hash-based signing. Table 1 summarizes the reported performance.

Threshold hash-based signature. Cozzo and Smart [21] observe that MPC-based thresholdization of SPHINCS⁺ is impractical, estimating 85 minutes per signature for

the secure hash evaluations alone. Kelsey et al. [40] propose a framework for threshold XMSS with a trusted dealer, leveraging a common reference value (CRV) whose size grows super-linearly with the number of one-time keys and with the number of signing coalitions; the approach does not extend to SPHINCS⁺, for which the authors estimate a CRV of more than 2^{73} bytes. The following approaches target a non-standardized scheme: Adomnicăi [1] thresholdizes the stateful LeanSig [25] with Poseidon2 [31] on MP-SPDZ [39], and Khaburzaniya et al. [41] aggregate one-time signatures into a STARK proof. Table 5 in Appendix B compares these approaches with ours.

2 Preliminaries

2.1 System Model and Notation

We consider t -of- n threshold setting of a system comprising a set of n signing parties $\mathcal{S} = \{P_1, P_2, \dots, P_n\}$. By default, we assume an adversary $\mathcal{A}$ that corrupts up to $t - 1$ parties in a semi-honest manner, is computationally bounded to run in probabilistic polynomial time (PPT), and is *static* where it chooses parties to be corrupted before a protocol run. We assume broadcast rounds of communication.

Our computational model’s security parameter is denoted by λ . We call a function $\text{negl}(\lambda)$ *negligible* if for all $c > 0$ there exists λ_0 such that $\text{negl}(\lambda) < \frac{1}{\lambda^c}$ for all $\lambda > \lambda_0$.

For a non-empty set S , we let $x \xleftarrow{\$} S$ denote sampling an element of S uniformly at random and assigning it to x . We use $[n]$ to represent the set $\{1, \dots, n\}$. We may simplify the notation $\text{ThFHE}.\text{Enc}(\text{pk}, \text{msg})$ described in Section 2.4 and represent a ciphertext by ct_{msg} . Likewise, $\text{ThDec}(\cdot)$ denotes threshold decryption in a simplified manner, representing $\text{ThFHE}.\text{PartDec}$ (one round of broadcast) followed by $\text{ThFHE}.\text{FinDec}$ (local computation).

2.2 Threshold Signature

We defer the full definitions of (threshold) signature schemes to Appendix A. We convey our context here. For instance, the signing algorithm for a non-threshold signature includes an optional index input idx , to represent stateful signature schemes like XMSS. Also, the partial signing algorithm for a threshold signature includes auxiliary information as input, enabling the blueprint in Section 4.

Definition 1 (Signature). A signature scheme is represented by the following algorithms: $\text{Sig.Setup}(\lambda)$ for generating a secret-public keypair, $\text{Sig.Sign}(\text{sk}, m, \text{idx})$ for signing a message m (optionally on index idx), and $\text{Sig.Verify}(\text{pk}, m, \sigma)$ for verifying a signature.

Definition 2 (Threshold signature). A threshold signature scheme is represented by the following tuple of algorithms: $\text{ThSig.Setup}(\lambda, n, t)$ for key distribution, $\text{ThSig.PartSign}(\text{sk}_i, m, \text{aux})$ for outputting a partial signature (by P_i), $\text{ThSig.Combine}(\text{pk}, \{(i, \sigma_i)\}_{i \in \gamma})$ for combining a threshold number of partial signatures, and $\text{ThSig.Verify}(\text{pk}, m, \sigma)$ for verifying a threshold signature.

Next, we consider the following security property (see Appendix A for details).

Unforgeability¹. Let $\mathcal{A}$ be a PPT adversary that corrupts $t - 1$ parties. Given a polynomial (in λ) number of valid message-signature pairs where $\mathcal{A}$ chooses the messages, $\mathcal{A}$ cannot output a valid message-signature pair for a new message.

2.3 Hash-Based Signatures

We briefly recall constructions of hash-based signatures. Full specifications are given in Appendices G and H. See Figure 5 for a more intuitive representation.

WOTS⁺. WOTS⁺ [36] is a one-time signature. An n -byte digest is written in base- w as len_1 digits, a checksum over those digits contributes len_2 more, and the signature reveals $F^{B_i}(\text{sk}_i)$ on each of the $len = len_1 + len_2$ hash chains, where B_i is the i -th digit. The checksum prevents an adversary from extending selected chains without inducing compensating changes in other chain positions.

FORS [9, 54]. A few-time signature with parameters (k, a) : k Merkle trees of height a . A ka -bit digest selects one leaf per tree, and the signature reveals those k secret leaves together with their authentication paths.

XMSS [15, 34]. XMSS authenticates 2^h WOTS⁺ key pairs, all derived from sk , under a single Merkle tree whose root is pk ; each leaf compresses one WOTS⁺ public key through an L -tree. A signature consists of the leaf index, a per-signature randomizer, a WOTS⁺ signature, and the authentication path. XMSS is therefore stateful: the signer must track which leaf indices it has already used, since reusing one breaks the one-time nature of the underlying WOTS⁺ key.

SPHINCS⁺. SPHINCS⁺ [9, 54] is *stateless*: its signing indices are derived from a digest of the randomized message. A *hypertree* of d layers of Merkle trees, each of height $h' = h/d$, authenticates the FORS instances, with every layer's WOTS⁺ key signing the root of the layer below. Setup therefore processes only the topmost tree of height h' .

2.4 Threshold Fully Homomorphic Encryption

CKKS. The CKKS scheme [18] is a fully homomorphic encryption scheme (FHE) for approximate arithmetic over complex-valued slot messages. For ring dimension N , a ciphertext can encode up to $N_{\text{slot}} = N/2$ slots,

$$\mathbf{m} = (m_0, \dots, m_{N/2-1}) \in \mathbb{C}^{N_{\text{slot}}},$$

and supports slot-wise addition (FHE.Add), ciphertext-ciphertext multiplication (FHE.Mult), and plaintext-ciphertext multiplication (FHE.PMult) together with cyclic slot rotations (FHE.Rot). We also use coefficient encoding (CoeffEcd), in which the message is embedded directly into the polynomial coefficients.

¹ With a slight abuse of notation, we may take $t = 1$ in the security game for unforgeability of a standard (non-threshold) signature.

Recent extensions of CKKS enable homomorphic computation over discrete messages [7, 8, 26]. In particular, for encrypted bits $x, y \in \{0, 1\}$, binary gates can be evaluated as $x \oplus y = (x - y)^2$, $x \wedge y = xy$. More recently, the XBOOT framework [55] suggests replacing homomorphic XOR evaluation with addition and recovers the binary outputs through lazy modular reduction during bootstrapping. This optimization makes XBOOT applicable to a wide range of applications involving binary circuits [6, 61], as XOR gates can be evaluated using additions instead of multiplications. Further details on CKKS and its encoding procedures are provided in Appendix C.

Bootstrapping. After consuming the available modulus budget of a ciphertext, bootstrapping restores the modulus budget while approximately preserving the message. For discrete messages, bootstrapping simultaneously restores the modulus budget and cleans the approximation error in the encoded message [7, 8, 26]. When bootstrapping multiple ciphertexts, we use *shared-a bootstrapping* [5] to reduce the amortized cost across the ciphertexts. Further details on the CKKS scheme, encoding procedures, and bootstrapping are provided in Appendix C.

Threshold fully homomorphic encryption. A *threshold fully homomorphic encryption (ThFHE)* [12] scheme is represented by the following: ThFHE.Setup for key distribution, $\text{ThFHE.Enc}(\text{fpk}, \text{msg})$ for encrypting a message, and ThFHE.Eval for homomorphically evaluating a circuit C on ciphertexts $\text{ct}_1, \dots, \text{ct}_k$ with the public key fpk . The remaining algorithms are $\text{ThFHE.PartDec}(\text{ct}, \text{fsk}_i)$ for outputting a partial decryption by P_i , and $\text{ThFHE.FinDec}(\{(i, \text{pd}_i)\}_{i \in \gamma})$ for combining a threshold number of partial decryptions. We assume that our ThFHE scheme satisfies simulation security [12, Definition 5.5]. We defer the full definition of threshold FHE to Appendix D.

2.5 Keccak Permutation

The Keccak- f [1600] permutation operates on a 1600-bit state $A[x, y, z]$ with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{64}$, corresponding to 25 lanes of 64 bits each. Each of its 24 rounds applies the following five transformations: $\theta \rightarrow \rho \rightarrow \pi \rightarrow \chi \rightarrow \iota$. Here, θ mixes column parities, ρ rotates bits within each lane, π permutes lane positions, χ applies a nonlinear row-wise Boolean transformation, and ι injects the round constant RC_r . Detailed definitions of these transformations and their homomorphic realizations are provided in Appendix F.

3 Efficient Keccak Permutation in CKKS

In this section, we present an efficient homomorphic evaluation of the Keccak- f [1600] permutation over discrete CKKS. We propose two packing strategies tailored to low amortized cost and low latency, respectively. Then, we design specialized homomorphic circuits for different batch state sizes.

3.1 Keccak State Packing

Keccak evaluation exhibits different levels of parallelism across hash-based signature procedures. Components such as WOTS⁺ chains, FORS, and Merkle-tree construction

expose many independent Keccak states, whereas parts of the online XMSS signing procedure involve only a small number of states and are latency-sensitive. We therefore consider two packing strategies: lane-wise packing for high parallelism and compact single-ciphertext packing for low latency.

The Keccak- $f[1600]$ state is represented as $A[x, y, z]$ with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{64}$, consisting of 25 lanes of 64 bits.

Throughput-aware Packing. We encode each Keccak lane (x, y) into a separate ciphertext. Recall $N_{\text{slot}} = N/2$ and let $b = \frac{N_{\text{slot}}}{\gamma} = \frac{N_{\text{slot}}}{64}$. For $j \in \{0, \dots, b-1\}$, the bit $A_j[x, y, z]$ of the j -th Keccak state is placed at slot

$$\text{idx}(j, z) = j + bz, \quad 0 \leq z < \gamma.$$

Thus, 25 ciphertexts represent b Keccak states in parallel, with each ciphertext packing the same lane across all b states. This interleaving lets a $b \cdot \ell$ -slot rotation shift each state's lane by ℓ bits without cross-state mixing. The intra-lane rotations in θ and ρ can therefore be evaluated directly using FHE.Rot, whereas π requires only ciphertext reindexing.²

Latency-aware Packing. For small batches, we instead pack an entire Keccak states into a single ciphertext. We linearize the lane index as $x + 5y$ and place the 25 lane bits in the first 25 positions of a 32-slot block, padding the remaining seven positions with zeros. The bit $A[x, y, z]$ of the r -th state is placed at

$$\text{idx}(x, y, z, r) = x + 5y + 32z + 2^{11}r, \quad 0 \leq r < \frac{N_{\text{slot}}}{2^{11}}.$$

Thus, each state occupies 2^{11} slots, and one ciphertext accommodates up to $N_{\text{slot}}/2^{11}$ states. For batches below this capacity, we use fewer-slot encoding [18]. Unlike the lane-wise layout, this compact layout requires homomorphic slot permutations for inter-lane operations, including the π step. Figure 1 illustrates the two packing layouts.

3.2 Homomorphic Evaluation Strategy

Message and error growth analysis. Immediately after binary bootstrapping, let each input to χ be $m + e$, where $m \in \{0, 1\}$ and $|e| \leq 2^{-P}$. Additions accumulate absolute errors additively. For two inputs $m_1 + e_1$ and $m_2 + e_2$, the absolute error of their product satisfies

$$(m_1 + e_1)(m_2 + e_2) = m_1m_2 + m_1e_2 + m_2e_1 + e_1e_2,$$

Thus, the error introduced by a χ multiplication is amplified by the magnitudes of its input messages. Applying this recurrence through the round transformations gives the

² Seuré and Suvanto [61] use an equivalent lane-wise layout with a different interpretation of the slot indices. Their packing serves only as an intermediate bootstrapping transformation: the 25 lane ciphertexts are gathered into a ciphertext before bootstrapping and unpacked afterward. In their configuration, this layout supports eight Keccak states in parallel (for their implementation parameter). We instead use the lane-wise layout as the representation throughout the evaluation, populating each lane ciphertext's slots with batched Keccak states and avoiding zero-padding. For intermediate batches, we reuse gathering and fuse packing into the ρ -step.

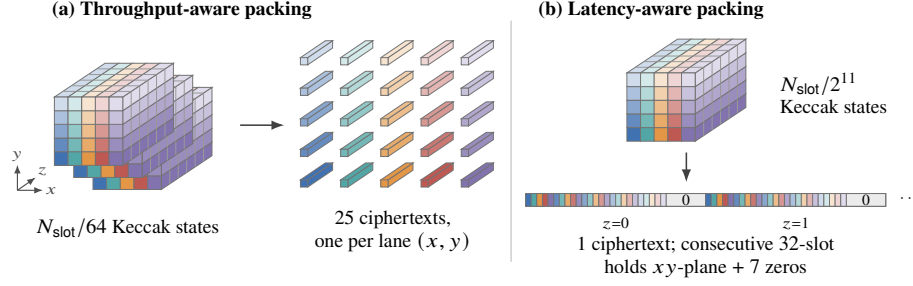


Fig. 1. Comparison of throughput- and latency-aware packing for Keccak evaluation.

step-by-step growth of the message magnitude and dominant absolute error shown in Table 11.

At the end of a two-round bootstrapping interval, the dominant absolute error reaches approximately 2^{14-P} ; extending the interval to three rounds exceeds the available precision margin. We therefore bootstrap every two rounds, halving the number of bootstrappings over the 24-round permutation. The resulting evaluation flow is illustrated in Figure 2. We additionally apply the cleaning polynomial $h(x) = 3x^2 - 2x^3$ to the bootstrapping output to suppress residual error before the next interval.

Batch-Dependent Evaluation Strategies. We use different circuit realizations depending on the number of Keccak states evaluated in parallel.

- **Large-batch evaluation.** Since bootstrapping accounts for most of the latency in large-batch (throughput-aware) evaluation (see Table 3), reducing the number of bootstrapping operations is crucial. Our message- and error-growth analysis shows that ciphertexts can be safely bootstrapped once every two Keccak rounds. The resulting homomorphic Keccak evaluation circuit is illustrated in Figure 2 (a). For implementation, we additionally use the conjugate-invariant (CI) ring optimization [42] for complex ciphertext bootstrapping [6] and shared- a bootstrapping [5] for the linear transformations S2C and C2S. A detailed ablation study for each optimization gain is provided in Table 14 of Appendix F.5.
- **Small-batch evaluation.** For batches that fit within a single ciphertext (latency-aware packing), the homomorphic Keccak itself becomes the dominant cost (see Table 3). We therefore reduce this by evaluating each permutation step using plaintext-matrix-ciphertext-vector multiplication (PCMV) with BSGS optimization [32].³ This design eliminates the Pack/Unpack overhead and lowers the total key-switching count (see Table 12).

Round-wise bootstrapping. Despite doubling the bootstrapping count, round-wise bootstrapping (Figure 2 (b)) is faster overall. The increased per-round depth means

³ The π step is free because it only re-indexes the xy -plane. The χ step additionally requires slot alignment, parts of which are fused into adjacent Keccak steps. The resulting permutation matrices for θ and χ vary across rounds.

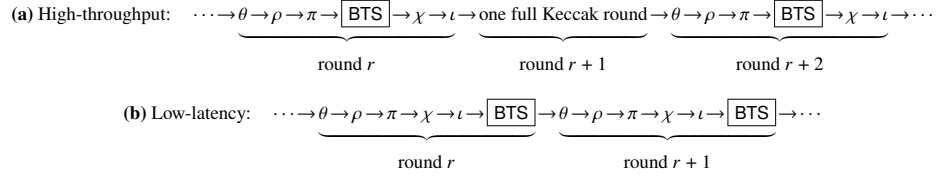


Fig. 2. Evaluation flows of the Keccak circuits. The high-throughput circuit bootstraps every two rounds, whereas the low-latency circuit bootstraps after each round.

Table 2. Batch-dependent Keccak evaluation strategies.

	Throughput-aware	Latency-aware
Permutation evaluation	Reindexing / Rotations	PCMV with BSGS
Bootstrapping	Shared- a + CI BTS	Normal BTS
Keccak round / BTS	2	1
BTS layout handling	Pack/Unpack [†]	—
Multiplicative depth / round	1	4

[†] Pack/Unpack is only required for intermediate batches; full-batch evaluation directly inputs 25 ciphertexts into the bootstrapping circuit.

that bootstrapping every two rounds requires a larger ring degree to satisfy the modulus budget, whereas bootstrapping after every round permits a smaller ring. The resulting reduction in the cost of all homomorphic operations outweighs the additional bootstrappings. This design reduces the homomorphic complexity by more than a factor of 2 compared with our throughput-aware approach and by a factor of 3.2 compared with [61] (see Table 12).

Remark 1 (Switching between ring dimensions). Our homomorphic evaluation uses both ring degree $\log N = 15$ and $\log N = 16$ ciphertexts. Intermediate ciphertexts are converted between these representations whenever consecutive operations require different ring dimensions. The $\log N = 15 \rightarrow 16$ conversion combines ring embedding with key switching, whereas the reverse conversion uses a homomorphic trace map followed by key switching. Any accompanying rearrangement of the packing or slot layout is performed separately using a permutation matrix evaluation with PCMV.

4 Round-Optimal Threshold Signatures

We construct one-round threshold XMSS and SPHINCS⁺ signatures using CKKS-based homomorphic evaluation. We first outline a generic construction from threshold FHE based on our syntax (as illustrated in Figure 3):

Blueprint: Threshold Signature from Threshold FHE.

1. $\text{ThSig.Setup}(\lambda, n, t) \rightarrow (\text{pp}, \text{pk}, \{\text{fsk}_i\}_{i=1}^n)$ runs $\text{ThFHE.Setup}(\lambda, n, t)$ in a distributed algorithm, via *distributed key generation (DKG)* [4, 30, 51, 59].

A DKG is a multi-party protocol realizing an ideal functionality that samples a (non-threshold) secret key fsk , samples its additive shares $\text{fsk}_1, \dots, \text{fsk}_n$, and distributes each fsk_i to P_i along with fpk . For CKKS, this is achieved by a recent protocol by Min et al. [51], which facilitates efficient bootstrapping via sparse secret encapsulation (SSE) [13].

Crucially, this allows the collaborative derivation of $\text{ThFHE.Enc}(\text{fpk}, \text{sk})$ (or simply ct_{sk}) representing a ciphertext encrypting the signing key sk where sk in plaintext is unknown to any party, e.g. by sourcing $\text{ct}_{\varepsilon_i}$ from P_i (where each P_i randomly samples ε_i) and letting $\text{ct}_{\text{sk}} = \sum_i \text{ct}_{\varepsilon_i}$.

Then encrypted pk is derived via ThFHE.Eval with a circuit C_{pk} that computes pk from sk , and it furthermore is threshold-decrypted so that pk becomes known in the clear. Note that threshold decryption is essentially ThFHE.PartDec (one round of broadcast) followed by ThFHE.FinDec (local computation).

2. $\text{ThSig.PartSign}(\text{fsk}_i, m, \text{aux}) \rightarrow \sigma_i$ runs $\text{ThFHE.PartDec}(\text{ct}, \text{fsk}_i)$ where ct is derived via ThFHE.Eval with a circuit C_σ that computes a signature σ on m from sk and m , given the provided $\text{aux} = \text{ct}_{\text{sk}}$.
3. $\text{ThSig.Combine}(\text{fpk}, \{(i, \sigma_i)\}_{i \in \gamma}) \rightarrow \sigma$ runs $\text{ThFHE.FinDec}(\{(i, \sigma_i)\}_{i \in \gamma})$ to recover the underlying plaintext, which is the signature σ .
4. $\text{ThSig.Verify}(\text{pk}, m, \sigma) \rightarrow \{0, 1\}$ verifies that σ is a valid signature on m .

We use this blueprint to organize the next subsections. Overall, the homomorphic signing occurs in two phases: an offline phase that derives encrypted signing material using batched hash evaluation and an online phase that constructs an encrypted signature, resolving message-dependent chain and tree selections through homomorphic lookup. The signers then jointly decrypt the encrypted signature to obtain the final signature.

4.1 Threshold Signature Setup

Joint Signing Key Generation. Both XMSS and SPHINCS⁺ require the jointly generated secret value sk (sk_{seed} and sk_R). Each semi-honest signer contributes an encryption of a random partial secret in CoeffEcd , and aggregate the encrypted contributions by homomorphic XOR so that the resulting secret values remain uniformly random. Then we apply halfBTS (see Appendix C) to obtain the slot encoding required by the homomorphic Keccak evaluation. The detailed packing and sampling procedure is given in Algorithm 1 of Appendix I. We note that sk remains encrypted throughout.

Homomorphic Public-Key Generation with Batching. Following the distributed setup, each signer independently evaluates $\text{PKGen}_{\text{sig}}$ (Algorithm 3) on the encrypted signing-key seeds. This computation consists of deriving WOTS⁺ secret keys by Keccak-based PRF and evaluating the WOTS⁺ chains for all leaves, compressing their chain heads into leaf nodes, and constructing the Merkle tree. Since these operations can be parallelized within each level, we batch the corresponding hash evaluations throughout $\text{PKGen}_{\text{sig}}$. See Appendix J for details.

4.2 Homomorphic Lookup for Hash-Based Signing

During online signing, each signer operates on the encrypted WOTS⁺ chain states and tree nodes (in case of threshold SPHINCS⁺) depending on the blinded signing key (sk_R), which is involved in derivation of message digest and hypertree leaf index. Considering round complexity, we must homomorphically generate mask from encrypted digest (resp. leaf index) of Keccak output form. We express both WOTS⁺ signature generation and authentication-path selection as homomorphic lookups over encrypted tables.

Multilinear Lookup. Let $\mathbf{T} = (T_0, \dots, T_{2^\ell-1})$ be an encrypted table, and let $\beta = (\beta_0, \dots, \beta_{\ell-1})$ be the encrypted binary representation of the lookup index. Writing i_r for the r -th bit of i , we evaluate the selected entry as

$$\text{Lookup}(\mathbf{T}, \beta) = \sum_{i=0}^{2^\ell-1} T_i \prod_{r=0}^{\ell-1} (\beta_r)^{i_r} (1 - \beta_r)^{1-i_r}.$$

A systematic treatment of lookup constructions based on this multilinear representation is provided in [44]. For our setting, we develop two evaluation strategies tailored to the respective table layouts. For WOTS⁺ signature selection, we apply the Möbius transform to the encrypted table and evaluate its multilinear polynomial at the encrypted index. For authentication-path selection, we construct the indicator basis from aligned index bits and apply it to encrypted tree nodes.

WOTS⁺ Block Selection from an Encrypted Digest. Given the encrypted digest M' , each signer first derives encrypted WOTS⁺ indices by extracting the base- w message digits and computing the checksum digits through IntMod [43]:

$$\text{ThFHE.Enc}(M') \xrightarrow{\text{base-}w \text{ and checksum}} (\text{ThFHE.Enc}(B_0), \dots, \text{ThFHE.Enc}(B_{\ell-1})).$$

The i -th index then selects one entry from the corresponding encrypted WOTS⁺ chain:

$$\text{ThFHE.Enc}(\sigma_i) = \text{Lookup}((C_{i,0}, \dots, C_{i,w-1}), \text{ThFHE.Enc}(B_i)).$$

We evaluate all ℓn lookups SIMD-wise using the Möbius-based strategy above, after aligning the chain-state tables and encrypted indices. Detailed packing and circuit descriptions are provided in Appendix K.

Signing Leaf Selection from an Encrypted Leaf Index. When the leaf index is encrypted, the chains to be signed are not known in the clear either. WOTS⁺ chains of the signing leaf must first be selected from those of all candidate leaves by a homomorphic lookup. This lookup is the same multilinear evaluation, with the chains of candidate leaves as the table.

Authentication-path Selection from an Encrypted Leaf Index. Unlike WOTS⁺ signature selection, the node index at each tree level is already available in binary form. We first replicate and align the encrypted index bits with the slots containing the candidate tree nodes using rotations and additions. Using plaintext masks, we then form the factors β_r and $1 - \beta_r$ corresponding to the two possible values of each index bit. Tree-structured

multiplications combine these factors into one-hot indicators for all candidate positions, as illustrated in Figure 7. Finally, we multiply the indicators by the encrypted tree nodes and sum the results to obtain the selected authentication-path node.

4.3 Threshold Signing Protocol

We now combine the preceding homomorphic components into complete XMSS and SPHINCS⁺ signing protocols. In both protocols, each signer independently performs the same homomorphic evaluation, and interaction is required only to threshold-decrypt the output ciphertexts.

XMSS. Each signer first computes $R = \text{PRF}_R(\text{sk}_R, \text{id}_{\text{sig}})$ and the message digest $M' = \text{H}_{\text{msg}}(R \parallel \text{pk} \parallel \text{id}_{\text{sig}}, M)$ homomorphically, where we use a small-batch Keccak evaluation. It then invokes the WOTS⁺ signature-selection procedure and the authentication-path lookup described above to obtain the encrypted XMSS signature components. Finally, the signers publish the partial decryption of encrypted R , WOTS⁺ signature, and authentication path. The full procedure is summarized in Algorithm 4.

SPHINCS⁺. For SPHINCS⁺, one first derives the randomizer, message digest, and the corresponding tree and leaf indices homomorphically. It then evaluates the FORS signature and authentication paths, followed by the d hypertree layers. Each layer reuses the batched WOTS⁺ evaluation, signature-selection, and authentication-path lookup procedures described above.

After all encrypted signature components are assembled, a single round of threshold decryption recovers the complete SPHINCS⁺ signature. The full procedure is summarized in Algorithm 4.

Threshold Decryption. After homomorphic signing, the signature components remain encrypted under the joint CKKS key fpk . Each signer computes a partial decryption of every output ciphertext and broadcasts it. The partial decryptions are then combined in a single communication round:

1. Each signer P_i computes, for every $j \in [\tilde{m}]$, a partial decryption $\text{pd}_{i,j} = \text{ThFHE.PartDec}(\text{ct}_j, \text{fsk}_i)$ with noise flooding [23].
2. Each signer broadcasts $\{\text{pd}_{i,j}\}_{j=1}^{\tilde{m}}$ to the other signers.
3. Any party that collects at least t partial decryptions for each ciphertext applies ThFHE.FinDec to recover the plaintext signature components.

Each signer broadcasts $\tilde{m}$ partially decrypted ciphertexts, where $\tilde{m} = 1$ for $\text{PKGen}_{\text{sig}}$ and XMSS signing, and $\tilde{m} = 13$ for SPHINCS⁺ signing to carry R , σ^{FORS} , $\sigma^{(\ell)}$, and the authentication paths. All partial decryptions are sent in a single communication round and combined locally via ThFHE.FinDec .

5 Security

Our schemes draw their security from two sources: the threshold FHE, and the signature scheme underneath. Threshold signing reveals neither the honest parties' key shares nor any signing material beyond what the signature itself contains, and depends on the

signing key only through the signatures themselves. For the simulation based security, see Appendix P. Unforgeability then reduces to that of the unmodified scheme, as follows.

Theorem 1 (Unforgeability of one-round threshold SPHINCS⁺ and XMSS). *Following the blueprint and the instantiation of ThFHE.Eval per Section 4, our concrete one-round threshold SPHINCS⁺ and XMSS are threshold signature schemes that satisfy unforgeability, assuming a secure threshold FHE based on the RLWE assumption.*

Proof. Recalling a theorem by Boneh et al. [12, Theorem 8.20], we make note of a couple observations. For instance, while we leverage the universal thresholdizer paradigm for our threshold signature constructions we operate in the semi-honest adversary model rather than in the malicious one (which is the default consideration for a universal thresholdizer, discussed below). Our threshold FHE is therefore sufficient to realize a universal thresholdizer, the security of which is based on the RLWE assumption [18, 48].

Also, we require the underlying signature scheme to be unforgeable. For XMSS and SPHINCS⁺, unforgeability stems from the standard properties of the hash function family: second-preimage resistance, undetectability, and multi-target multi-function resistance for XMSS [34] and interleaved target subset resilience for SPHINCS⁺ [9]. With these, we directly apply the theorem by Boneh et al. $\square$

Malicious security. Our protocols are described in the semi-honest setting. We sketch an extension to malicious security with abort: the output they produce is a signature and is therefore *self-verifying*, and malformed shares or withheld messages can be treated as an abort. Appendix Q sketches this direction stage by stage; we leave a formal treatment to future work.

Security assumption tradeoff for thresholdization. As we introduce an assumption (RLWE) on top of the underlying hash-based security, a sweet spot seems to be when one may require thresholdization of a hash-based signature (e.g. as in a post-quantum variant of Ethereum [28]) such that it has full backward compatibility. The MPC approach [21] trades off round efficiency, while the CRV approach [40] (which works for XMSS only) trades off storage; our “overhead” is the RLWE assumption (which is also plausibly post-quantum).

6 Experimental Evaluation

We implemented all evaluation circuits using the HEaasN2 [22] library, which provides both algorithmic and GPU-level optimizations for CKKS homomorphic operations. Experiments were conducted on a single-thread Intel Xeon Gold 6542Y processor equipped with eight NVIDIA RTX 4090 GPUs. All reported timings are averaged over 30 runs. Signing latency measures the local computation of one signer, assuming that all signers perform the same homomorphic evaluation in parallel; network latency and final ciphertext combination are excluded.

Parameters. We employed the *grafting* technique [17], which reduces the number of RNS primes by allowing the use of word-size moduli (approximately 60 bits), thereby accelerating homomorphic operations. We use ring degree $\log N = 15$ and

Table 3. Homomorphic Keccak- $f[1600]$ evaluation performance for various batch sizes, including total latency, bootstrapping ratio, and amortized latency per Keccak input.

Num. Batch	Latency-aware		Throughput-aware				
	1–8	16	32	64	128	256	512
Latency (s)	0.198	0.263	0.458	0.583	0.786	1.128	1.752
Bootstrap (%)	62.1	49.5	43.3	56.5	68.4	77.7	85.3
Amor. latency (ms)	24.8	16.4	14.3	9.1	6.1	4.4	3.4

$\log N = 16$ parameter sets, with a Hamming weight $h_{\text{sk}} = 256$, and sparse secret Hamming weight $\bar{h} = 32$ [13]. Across the evaluated workloads, the top-modulus size ranges from $\log PQ = 833$ to 1551 bits. Detailed parameters are given in Table 7 lists detailed parameters. These parameters achieve 128-bit security according to the lattice estimator [2].

6.1 Keccak Evaluation

Table 3 reports the total and amortized latency per Keccak state. For up to 16 states, latency-aware packing evaluates the entire batch in one ciphertext with a fixed latency of 0.198 s (0.263 s for 16 states). For larger batches, throughput-aware packing increasingly utilizes the available slots, reducing the per-state latency to 3.4 ms at batch size 512. At this scale, bootstrapping accounts for 85.3% of the runtime, motivating our optimized bootstrapping circuit.

Table 13 compares our implementations with the method of Seuré and Suvanto [61]. Our low-latency implementation achieves an overall latency that is 1.82× lower, while our high-throughput implementation reduces the amortized latency by 8.6×. The ablation in Table 14 attributes this gain to both full-batch packing (3.4×) and subsequent circuit and bootstrapping optimizations, which further reduce the amortized latency from 8.64 to 3.42 ms.

6.2 Threshold Signing

Our threshold implementation follows the RFC 8391 reference code [34, 35] for XMSS, instantiated with the XMSS-SHAKE256_10_256 parameter set (OID 0x10) [20], and the SPHINCS⁺ reference code [9, 10] with SLH-DSA-SHAKE-256f; every threshold signature produced ensures *full backward compatibility*. Table 6 fixes the standardized signature parameter sets used in our experiments. Throughout the homomorphic evaluation, we consider a 5-of-5 threshold setting as a proof of concept. We discuss overhead for general t -out-of- n threshold parameters in Appendix E.3.

Adapting Parallelism. We analyse the cost of XMSS and SPHINCS⁺ setup and signing as a function of the number of parallel processors. Write W for the width of a level, that is, the number of Keccak permutations it may evaluate in parallel, and g for the number

of processors. With the throughput-aware batch capacity $b = N_{\text{slot}}/\gamma$ of Section 3.1,

$$\text{batch size} = \min\left(b, \left\lceil \frac{W}{g} \right\rceil\right), \quad \text{number of batches} = \left\lceil \frac{W}{\text{batch size}} \right\rceil. \quad (1)$$

Appendix N gives the depth of each phase and the width of every level. We set $g = 8$ for XMSS setup and for SPHINCS⁺ setup and signing; XMSS signing needs a single processor, its permutations having width 1 and depth 3. Our implementation keeps the len chains of a leaf in one batch and assigns whole trees to GPUs, so some of its batches are smaller than Equation (1) gives.

Table 4 summarizes the setup and signing latency of XMSS and SPHINCS⁺ under their respective GPU configurations; Table 20 in Appendix O gives the full per-step breakdown.

Table 4. Setup and online-signing latency. XMSS setup uses $h = 10$ on 8 GPUs and signing 1 GPU; SPHINCS⁺ setup and signing use 8 GPUs.

	Setup (s)	Signing (s)
XMSS-SHAKE256_10_256	845.9	0.69
SLH-DSA-SHAKE-256f	26.8	238.5

XMSS. XMSS is dominated by its offline setup, which performs the WOTS⁺ chain evaluations and L-tree compression for all 2^{10} leaves. The leaves are processed in batches of seven ($\lfloor 512/\text{len} \rfloor = 7$) and distributed independently across eight GPUs. In contrast, the signing phase is subsecond on a single GPU.

The online phase is dominated by the generation of R and M' , both of which require single-state Keccak evaluations. We therefore use the low-latency packing described in Section 3.1. Moreover, under RFC 8391 [34], $R = \text{PRF}_R(\text{sk}_R, \text{id}_{\text{sig}})$ depends only on the secret key and leaf index. If all possible R values are precomputed and stored during setup, the online signing latency decreases from 0.69 s to approximately 0.49 s.

SPHINCS⁺. Unlike XMSS, SPHINCS⁺ has a lightweight setup phase: only the top subtree required for the public key is constructed. The remaining FORS and hypertree computations are deferred to online signing, which consequently dominates the total cost (see Table 4).

Online signing uses eight GPUs and is dominated by hypertree construction. FORS signing, WOTS⁺ signature selection, and authentication-path extraction contribute comparatively little. Appendix O gives a per-GPU breakdown.

Threshold Decryption. For the threshold decryption of discrete CKKS ciphertexts, we carefully adapt thrifty-cleaning technique of Choe et al. [19]. Each signer first performs a binary bootstrapping and then evaluates the cleaning function iteratively with a growing scale factor. We use the communication-efficient method of [58]: after noise flooding and rescaling, each output ciphertext is represented over a single word-sized prime. Details about noise flooding are summarized in Appendix E.3. Setup for both signatures and

XMSS signing requires one such ciphertext, whereas SPHINCS⁺ signing requires 13, resulting in communication costs of 0.524 MB and 6.812 MB, respectively.

7 Discussion

Tall trees and the multi-tree variant (XMSS^{MT}). Supporting more signatures in XMSS requires increasing the Merkle-tree height h . Then the computation cost and storage requirement of $\text{PKGen}_{\text{sig}}$ grow exponentially with h , since all $2^h \cdot \text{len}$ WOTS⁺ chains must be evaluated. The storage cost can be reduced by discarding each WOTS⁺ chain once it has been consumed by leaf compression. The computation cost can be addressed by the multi-tree variant. XMSS^{MT} [34] partitions the tree into d layers of subtrees of height $h' = h/d$, constructing only the top layers offline. These approaches require on-the-fly WOTS⁺ chain construction during the signing phase.

Outsourcing computation. As involving all signing parties as individuals to compute the same homomorphic signing operation may raise a practical concern, we note it is possible to delegate this computation to a semi honest server. Upon the completion of such outsourced computation, the server broadcasts the relevant ciphertexts, and each P_i can only perform $\text{ThFHE.PartDec}(\text{ct}_\sigma, \text{fsk}_i)$ without having to homomorphically compute ct_σ from ct_{sk} . In fact, the initial signing request on message m can be accompanied by the semi-trusted server's result ct_σ from the outset. All in all, high-throughput computing resources are no longer required of every signer in this setting.

Acknowledgments

The authors would like to thank Benedikt Wagner and Intak Hwang for feedback.

References

1. Adomnicăi, A.: Towards practical multi-party hash chains using arithmetization-oriented primitives: With applications to threshold hash-based signatures. IACR Communications in Cryptology (2026)
2. Albrecht, M., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology (2015)
3. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold fhe. In: Advances in Cryptology – EUROCRYPT (2012)
4. Bacho, R., Kavousi, A.: Sok: Dlog-based distributed key generation. In: 2025 IEEE S&P (2025)
5. Bae, Y., Cheon, J.H., Hanrot, G., Park, J.H., Stehlé, D.: Plaintext-ciphertext matrix multiplication and fhe bootstrapping: Fast and fused. In: Advances in Cryptology – CRYPTO (2024)
6. Bae, Y., Cheon, J.H., Kang, M., Kim, T.: High-throughput aes transciphering using ckks: Less than 1ms. In: WAHC '25' (2025)
7. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with ckks. In: Advances in Cryptology – EUROCRYPT (2024)

8. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with ckks. In: *Advances in Cryptology – ASIACRYPT* (2025)
9. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: *ACM CCS* (2019)
10. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: SPHINCS⁺ reference implementation. <https://github.com/sphincs/sphincsplus> (2019)
11. Bon, N., Pointcheval, D., Rivain, M.: Optimized homomorphic evaluation of boolean functions. *IACR TCHES* (2024)
12. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: *Advances in Cryptology – CRYPTO* (2018)
13. Bossuat, J.P., Troncoso-Pastoriza, J., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: *ACNS* (2022)
14. Brandão, L., Peralta, R.: Nist first call for multi-party threshold schemes (2026)
15. Buchmann, J., Dahmen, E., Hülsing, A.: Xmss - a practical forward secure signature scheme based on minimal security assumptions. In: *Post-Quantum Cryptography* (2011)
16. Buterin, V., lightclient, Lange, F., Weiss, Y., Forshtat, A., Tirosh, D., Nacson, S., Chiang, D., Wahrstätter, T.: EIP-8141: Frame Transaction. <https://eips.ethereum.org/EIPS/eip-8141> (2026), *ethereum Improvement Proposals*
17. Cheon, J.H., Choe, H., Kang, M., Kim, J., Kim, S., Mono, J., Noh, T.: Grafting: Decoupled scale factors and modulus in rns-ckks. In: *ACM CCS* (2025)
18. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: *Advances in Cryptology – ASIACRYPT* (2017)
19. Choe, H., Kim, J., Stehlé, D., Suvanto, E.: Leveraging discrete ckks to bootstrap in high precision. In: *ACM CCS* (2025)
20. Cooper, D.A., Apon, D.C., Dang, Q.H., Davidson, M.S., Dworkin, M.J., Miller, C.A.: Recommendation for Stateful Hash-Based Signature Schemes. NIST special publication (sp), National Institute of Standards and Technology (2020)
21. Cozzo, D., Smart, N.P.: Sharing the luov: Threshold post-quantum signatures. In: *Cryptography and Coding* (2019)
22. CryptoLab, H.: HEaaN2 library. <https://www.cryptolab.co.kr/en/products-en/heaan-he/> (2026)
23. Dahl, M., Demmler, D., El Kazdadi, S., Meyre, A., Orfila, J.B., Rotaru, D., Smart, N.P., Tap, S., Walter, M.: Noah’s ark: Efficient threshold-fhe using noise flooding. In: *WAHC* (2023)
24. Donoghue, S.: Custody in the age of digital assets: The path to building market infrastructure fit for a tokenised economy. *Journal of Securities Operations & Custody* (2024)
25. Drake, J., Khovratovich, D., Kudinov, M., Wagner, B.: Technical note: LeanSig for post-quantum ethereum. *Cryptology ePrint Archive*, Paper 2025/1332 (2025)
26. Drucker, N., Moshkovich, G., Pelleg, T., Shaul, H.: Bleach: Cleaning errors in discrete computations over ckks. *J. Cryptol.* (2023)
27. Ethereum: Ethereum distributed validator specification. <https://github.com/ethereum/distributed-validator-specs> (2022)
28. Ethereum Foundation: Lean consensus roadmap. <https://leanroadmap.org> (2026)
29. Garg, S., Goel, A., Jain, A., Roberts, B., Sekar, S.: Malicious security in collaborative zk-snarks: more than meets the eye. In: *Advances in Cryptology – CRYPTO* (2025)
30. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. In: *International Conference on the Theory and Applications of Cryptographic Techniques* (1999)
31. Grassi, L., Khovratovich, D., Schofnegger, M.: Poseidon2: A faster version of the poseidon hash function. In: *Progress in Cryptology – AFRICACRYPT* (2023)

32. Halevi, S., Shoup, V.: Faster homomorphic linear transformations in helib. In: *Advances in Cryptology – CRYPTO* (2018)
33. Hong, D., Choi, Y., Lee, Y., Kim, Y.S.: Overlapped bootstrapping for fhew/tfhe and its application to sha3. In: *Financial Cryptography and Data Security (FC)* (2025)
34. Huelsing, A., Butin, D., Gazdag, S.L., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC 8391 (2018)
35. Huelsing, A., Butin, D., Gazdag, S.L., Rijneveld, J., Mohaisen, A.: XMSS reference implementation. <https://github.com/XMSS/xmss-reference> (2018)
36. Hülsing, A.: W-ots+ – shorter signatures for hash-based signature schemes. In: *Progress in Cryptology – AFRICACRYPT* (2013)
37. Hwang, I., Hwang, Y., Kim, M., Lee, D., Song, Y.: On the (in)security of approximate computation protocols from CKKS. *Cryptology ePrint Archive*, Paper 2025/395 (2025)
38. Hwang, I., Lee, H., Seo, J., Song, Y.: Practical zero-knowledge piop for maliciously secure multiparty homomorphic encryption. In: *ACM CCS* (2025)
39. Keller, M.: Mp-spdz: A versatile framework for multi-party computation. In: *ACM CCS* (2020)
40. Kelsey, J., Lang, N., Lucks, S.: Turning hash-based signatures into distributed signatures and threshold signatures: Delegate your signing capability, and distribute it among trustees. *IACR Communications in Cryptology* (2025)
41. Khaburzaniya, I., Chalkias, K., Lewi, K., Malvai, H.: Aggregating and thresholdizing hash-based signatures using starks. In: *ACM Asia CCS* (2022)
42. Kim, D., Song, Y.: Approximate homomorphic encryption over the conjugate-invariant ring. In: *Information Security and Cryptology – ICISC* (2019)
43. Kim, J., Noh, T.: Modular reduction in CKKS. *IACR Communications in Cryptology* (2025)
44. Kim, T., Lee, J., Noh, T., Cheon, J.H., Hanrot, G.: Latency-aware, high-throughput homomorphic AES evaluation with CKKS. *Cryptology ePrint Archive*, Paper 2026/1209 (2026)
45. Kim, W., Lee, C., Lee, J., Passelègue, A., Stehlé, D.: Asynchronous lagrange-based threshold fhe with smaller modulus overhead. In: *Advances in Cryptology – CRYPTO* (2026)
46. Kudinov, M., Nick, J.: Hash-based signature schemes for bitcoin. *Cryptology ePrint Archive*, Paper 2025/2203 (2025)
47. Li, B., Micciancio, D., Schultz-Wu, M., Sorrell, J.: Securing approximate homomorphic encryption using differential privacy. In: Dodis, Y., Shrimpton, T. (eds.) *Advances in Cryptology – CRYPTO* (2022)
48. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: *Advances in Cryptology – EUROCRYPT* (2010)
49. Lyubashevsky, V., Seiler, G., Steuer, P.: The lazer library: Lattice-based zero knowledge and succinct proofs for quantum-safe privacy. In: *ACM CCS* (2024)
50. Mankowski, D., Wiggers, T., Moonsamy, V.: Tls→ post-quantum tls: Inspecting the tls landscape for pqc adoption on android. In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2023)
51. Min, S., Hanrot, G., Park, J.H., Passelègue, A., Stehlé, D.: Distributed key generation for efficient threshold-CKKS. In: *ACM CCS* (2026)
52. National Institute of Standards and Technology: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Information Processing Standards Publication (FIPS), U.S. Department of Commerce (2015)
53. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication (FIPS), U.S. Department of Commerce (2024)
54. National Institute of Standards and Technology: Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standards Publication (FIPS), U.S. Department of Commerce (2024)

55. Niu, C., Huang, Z., Yang, Z., Chen, Y., Kong, L., Hong, C., Wei, T.: Xboot: Free-xor gates for ckks with applications to transciphering. IACR TCHES (2025)
56. Ozdemir, A., Boneh, D.: Experimenting with collaborative {zk-SNARKs}:{Zero-Knowledge} proofs for distributed secrets. In: USENIX Security (2022)
57. Park, J.H., Passelègue, A., Stehlé, D.: THED: Threshold dilithium from FHE. Cryptology ePrint Archive, Paper 2026/638 (2026). <https://doi.org/10.1145/3830454.3846802>
58. Passelègue, A., Stehlé, D.: Low communication threshold fully homomorphic encryption. In: Advances in Cryptology – ASIACRYPT (2025)
59. Pedersen, T.P.: A threshold cryptosystem without a trusted party. In: Workshop on the Theory and Application of Cryptographic Techniques (1991)
60. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Advances in Cryptology – CRYPTO (1989)
61. Seuré, T., Suvanto, E.: Faster Homomorphic Evaluation of SHA3. Poster presentation at the FHE.org Conference (2026)
62. Shrestha, N., Abraham, I., Ren, L., Nayak, K.: On the optimality of optimistic responsiveness. In: ACM CCS (2020)
63. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in tls 1.3: A performance study. In: NDSS (2020)
64. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: ACM PODC (2019)
65. Zakharov, D., Kudinov, M., Balatska, V., Chopra, Y.: Lattice-based signature schemes for bitcoin. Cryptology ePrint Archive, Paper 2026/1628 (2026)

A Threshold Signature Definitions and Blueprint

We provide the definitions of signature schemes, both non-threshold and threshold, in a way that matches the context of our work. For instance, the signing algorithm for a non-threshold signature includes an optional index input idx , to represent stateful signature schemes like XMSS. Also, the partial signing algorithm for a (one-round) threshold signature includes auxiliary information as input.

Definition 3 (Signature). A signature scheme is represented by the following tuple of algorithms:

- $\text{Sig.Setup}(\lambda) \rightarrow (\text{pp}, \text{pk}, \text{sk})$ takes as input the security parameter λ and outputs public parameters pp (an implicit input to all other algorithms), a public key pk , and a secret key sk .
- $\text{Sig.Sign}(\text{sk}, m, \text{idx}) \rightarrow \sigma$ takes as input a secret key sk , a message m , and an index idx , and outputs a signature σ on m . A signature scheme is called *stateless* if $\text{idx} = \perp$, and *stateful* otherwise.
- $\text{Sig.Verify}(\text{pk}, m, \sigma) \rightarrow \{0, 1\}$ takes as input a public key pk , a message m , and a signature σ , and verifies that (m, σ) is a valid message-signature pair.

Definition 4 (Threshold signature). Let $\{P_1, \dots, P_n\}$ be a set of parties. A threshold signature scheme is represented by the following tuple of algorithms:

- $\text{ThSig.Setup}(\lambda, n, t) \rightarrow (\text{pp}, \text{pk}, \{\text{sk}_i\}_{i=1}^n)$ takes as input λ , the number of parties n , and the threshold parameter t , and outputs public parameters pp (an implicit input to all other algorithms), a public key pk , and each secret key share sk_i sent to P_i .

$\text{ThSig.PartSign}(\text{sk}_i, m, \text{aux}) \rightarrow \sigma_i$ takes as input a secret key share sk_i , a message m , and auxiliary information aux , and outputs a partial signature σ_i by P_i .
 $\text{ThSig.Combine}(\text{pk}, \{(i, \sigma_i)\}_{i \in \gamma}) \rightarrow \sigma$ takes as input a public key pk and a set $\{(i, \sigma_i)\}_{i \in \gamma}$ where $|\gamma| \geq t$, and outputs a signature σ , or $\perp$ if the combination fails.
 $\text{ThSig.Verify}(\text{pk}, m, \sigma) \rightarrow \{0, 1\}$ takes as input a public key pk , a message m , and a signature σ , and verifies that (m, σ) is a valid message-signature pair.

A.1 One-Round Threshold Signing Blueprint

Figure 3 illustrates the generic one-round threshold signing of Section 4.

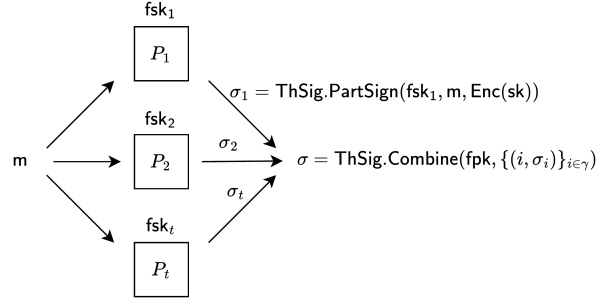


Fig. 3. One-round threshold signing from threshold FHE. Each signer P_i holds only a share fsk_i of the FHE secret key. Given the message m , it returns a partial signature $\sigma_i \leftarrow \text{ThSig.PartSign}(\text{fsk}_i, m, \text{Enc}(\text{sk}))$ in a single round. Any qualified set γ of partial signatures is then publicly combined into $\sigma \leftarrow \text{ThSig.Combine}(\text{fpk}, \{(i, \sigma_i)\}_{i \in \gamma})$.

Next, we consider the following security property:

A.2 Unforgeability

Let $\mathcal{A}$ be a PPT adversary that corrupts $t - 1$ parties. Given a polynomial (in λ) number of valid message-signature pairs where $\mathcal{A}$ chooses the messages, $\mathcal{A}$ cannot output a valid message-signature pair for a new message. Formally, unforgeability is satisfied if the probability of $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ winning in Figure 4 is negligible.

B Comparison of Threshold Hash-Based Signatures

Table 5 compares threshold hash-based signature approaches. A standard scheme is one of XMSS, LMS or SLH-DSA, and backward compatibility means that the produced signature is accepted by an unmodified verifier of the underlying scheme. Ours is the only approach that thresholdizes a standardized stateless scheme in a single online round; the price is high computation cost for homomorphic circuit, and that the thresholdization additionally assumes RLWE rather than symmetric primitives alone.

$\mathcal{G}_{\text{ThSig}, \mathcal{A}}^{\text{ufg}}(\lambda)$	
1 :	$(\text{pp}, \text{pk}, \{\text{sk}_i\}_{i=1}^n) \leftarrow \text{Setup}(\lambda, n, t)$
2 :	$\tilde{\gamma} \leftarrow \mathcal{A}_0(\text{pk})$
3 :	$(m, \sigma) \leftarrow \mathcal{A}_1^{\{\text{PartSign}(\text{sk}_i, \cdot, \cdot)\}_{i \in [n] \setminus \tilde{\gamma}}}(\text{pk}, \{\text{sk}_i\}_{i \in \tilde{\gamma}})$
4 :	return $\text{Verify}(\text{pk}, m, \sigma)$

Fig. 4. Security game for ThSig unforgeability. With a slight abuse of notation, $\{\text{PartSign}(\text{sk}_i, \cdot, \cdot)\}_{i \in [n] \setminus \tilde{\gamma}}$ denotes a set of oracles for honest partial signatures on messages other than m (to prevent trivial wins).

Table 5. Comparison of threshold hash-based signature approaches.

	Std. scheme	Backw. comp.	State- less	Online rounds	Comp. cost		Scalab. in (t, n)	Sec. for thresh.
					signer aggr.			
Cozzo and Smart [21]	✓	✓	✓	$\approx \text{depth}$	low	—	moderate	MPC
Kelsey et al. [40]	✓	✓	×	≥ 2	low	low	low	PRF
Adomniciăi [1]	×	✓	×	$\approx \text{depth}$	low	—	moderate	MPC
Khaburzaniya et al. [41]	×	×	×	1	low	high	high	STARK
Ours	✓	✓	✓	1	high	low	high	FHE

C Additional Details on CKKS

The Cheon-Kim-Kim-Song (CKKS) scheme [18] is a widely used fully homomorphic encryption (FHE) scheme for approximate arithmetic. Let $N > 1$ be a power-of-two integer and define a polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and its quotient $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ for $q \in \mathbb{Z}_{>1}$. A message vector $\mathbf{m} = (m_0, m_1, \dots, m_{N/2-1}) \in \mathbb{C}^{N/2}$ is slot encoded into a plaintext polynomial $\text{pt} \in \mathcal{R}$ through the inverse canonical embedding, implemented using an inverse discrete Fourier transform, with scaling factor Δ :

$$\text{Ecd}(\mathbf{m}, \Delta) = \text{pt} \approx \lfloor \Delta \cdot \text{DFT}^{-1}(\mathbf{m}) \rfloor,$$

while the decoding map $\text{Dcd} : \mathcal{R} \rightarrow \mathbb{C}^{N/2}$ is defined as $\text{Dcd}(\text{pt}, \Delta) = \Delta^{-1} \text{DFT}(\text{pt})$. We also consider an alternative encoding, called *coefficient encoding*. Given $\mathbf{z} = (z_i)_{0 \leq i < N}$, the coefficient encoding (CoeffEcd) is defined as $\text{CoeffEcd}(\mathbf{z}, \Delta) = \sum_{i=0}^{N-1} \lfloor \Delta \cdot z_i \rfloor X^i$. A ciphertext at modulus q_ℓ is an RLWE ciphertext $\text{ct} = \text{FHE.Enc}(\text{pt}, s) = (b, a) \in \mathcal{R}_{q_\ell}^2$ satisfying

$$\text{FHE.Dec}(\text{ct}, s) = b + a \cdot s \approx \text{pt} \pmod{q_\ell},$$

where $s \in \mathcal{R}$ is the secret key. We omit s in Dec if it is obvious. Applying the decoding map recovers the underlying message vector $\mathbf{m}$. For simplicity, we henceforth let Dec denote the composition of decryption and decoding.

Let ct_x and ct_y encrypt slot vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^{N_{\text{slot}}}$, respectively, and let pt_p encode a plaintext vector $\mathbf{p}$. CKKS supports the following homomorphic operations:

- $\text{FHE.Add}(\text{ct}_x, \text{ct}_y)$ returns a ciphertext encrypting $\mathbf{x} + \mathbf{y}$.
- $\text{FHE.Mult}(\text{ct}_x, \text{ct}_y)$ returns a ciphertext encrypting $\mathbf{x} \odot \mathbf{y}$.
- $\text{FHE.PMult}(\text{ct}_x, \text{pt}_p)$ returns a ciphertext encrypting $\mathbf{x} \odot \mathbf{p}$, where $\odot$ denotes component-wise multiplication.
- $\text{FHE.Conj}(\text{ct}_x)$ returns a ciphertext encrypting $\bar{\mathbf{x}} = (\overline{x_0}, \dots, \overline{x_{N_{\text{slot}}-1}})$.
- $\text{FHE.Rot}(\text{ct}_x, i)$ returns a ciphertext encrypting the left cyclic rotation

$$\text{Rot}(\mathbf{x}, i) = (x_i, \dots, x_{N_{\text{slot}}-1}, x_0, \dots, x_{i-1}).$$

If both inputs are encoded at scale Δ , multiplication increases the scale to approximately Δ^2 ; FHE.Rescale restores it to approximately Δ while dropping one modulus prime from the modulus chain.

Bootstrapping. When the modulus chain is exhausted, *bootstrapping* refreshes a ciphertext to restore its modulus budget. We use SlotsToCoeffs-first bootstrapping, consisting of

$$\text{S2C} \rightarrow \text{ModRaise} \rightarrow \text{C2S} \rightarrow \text{EvalMod},$$

where SlotsToCoeffs (S2C) and CoeffsToSlots (C2S) homomorphically evaluate the corresponding linear transforms, and EvalMod removes the modular ambiguity introduced by ModRaise. We employ sparse-secret encapsulation [13] to reduce the cost of EvalMod. For complex message bootstrapping, EvalMod can instead be evaluated over the conjugate-invariant ring [6, 42]. For coefficient-encoded ciphertexts, we use halfBTS, which starts from ModRaise.

CI-ring in Bootstrapping. Kim and Song [42] instantiate CKKS over the conjugate-invariant subring $R^+ = \{a \in R : a(X) = a(X^{-1})\}$ of the cyclotomic ring R . Its canonical embedding is real-valued, so a ciphertext over R^+ carries N real slots instead of the $N/2$ complex slots of standard CKKS, with security resting on RLWE over R^+ . Following Bae et al. [6], we use this ring for the nonlinear part of bootstrapping: after C2S, the real and imaginary parts of a complex-packed bit ciphertext are placed in the N real slots of a CI-ring ciphertext, and EvalMod is evaluated once over all N bits with real-valued arithmetic, rather than separately for the two parts. The result is mapped back to the complex-packed layout afterwards.

Shared-a Batch Bootstrapping. The shared- a RLWE format [5] represents multiple ciphertexts with a common RLWE component a . This structure is preserved under FHE.Add , FHE.PMult , FHE.Rot , and FHE.Conj . Since the linear transformations in CKKS bootstrapping, such as C2S and S2C, consist of these operations, computations associated with the shared component can be reused across multiple ciphertexts, reducing the amortized bootstrapping cost.

CKKS over Discrete Messages. Discrete CKKS [7] enables homomorphic computation over binary messages. Building on this representation, XBOOT [55] evaluates XOR gates using homomorphic addition and defers the required modulo-2 reduction to bootstrapping.

More generally, for an integer modulus $t > 2$, the primitive IntMod_t [43] homomorphically computes

$$\text{IntMod}_t(x) = x \bmod t$$

for an encrypted small integer x . We use this primitive for encrypted digit extraction in the signing procedure.

D Threshold Fully Homomorphic Encryption

Definition 5 (Threshold fully homomorphic encryption [12]). A threshold fully homomorphic encryption (ThFHE) scheme is represented by the following:

ThFHE.Setup(λ, n, t) $\rightarrow$ ($\text{pp}, \text{fpk}, \{\text{fsk}_i\}_{i=1}^n$) takes as input λ , the number of parties n , and the threshold parameter t , and outputs public parameters pp (an implicit input to all other algorithms), a public key fpk , and each secret key share fsk_i sent to P_i .
 ThFHE.Enc(fpk, msg) $\rightarrow \text{ct}$ takes as input a public key fpk and a message msg , and outputs a ciphertext ct .
 ThFHE.Eval($\text{fpk}, C, \text{ct}_1, \dots, \text{ct}_k$) $\rightarrow \text{ct}$ takes as input a public key fpk , a circuit $C : \{0, 1\}^{\lambda \cdot k} \rightarrow \{0, 1\}^\lambda$, and a set of ciphertexts $\text{ct}_1, \dots, \text{ct}_k$, and outputs a ciphertext ct . Wherever applicable and obvious, we may simplify notation and use $C(\text{ct}_1, \dots, \text{ct}_k)$ instead.
 ThFHE.PartDec(ct, fsk_i) $\rightarrow \text{pd}_i$ takes as input a ciphertext ct , and a secret key share fsk_i , and outputs a partial decryption pd_i by P_i .
 ThFHE.FinDec($\{(i, \text{pd}_i)\}_{i \in \gamma}$) $\rightarrow \text{msg}$ takes as input a set $\{(i, \text{pd}_i)\}_{i \in \gamma}$ where $|\gamma| \geq t$, and outputs a message msg .

We assume our ThFHE satisfies simulation security [12, Definition 5.5].

E Parameters

E.1 Signature Parameter Sets

We instantiate XMSS with XMSS-SHAKE256_10_256 and SPHINCS⁺ with SLH-DSA-SHAKE-256f. Table 6 summarizes the corresponding signature parameters used throughout our construction and experiments.

Table 6. Signature parameter sets. $\ell = \text{len} = \text{len}_1 + \text{len}_2 = 64 + 3$ is the WOTS⁺ chain count under the checksum encoding of RFC 8391 [34] and FIPS 205 [54]. For SPHINCS⁺, h : total hypertree height, d : layers, $h' = h/d$. For XMSS, h : tree height. Signature sizes $|\sigma|$ are in bytes.

Scheme	n	w	ℓ	h	d	h'	(a, k)	$ \sigma $
XMSS_10_256	32	16	67	10	–	–	–	2 500
SLH-256f	32	16	67	68	17	4	(9, 35)	49 856

E.2 Detailed CKKS Parameters

We use separate modulus-chain configurations for homomorphic signing and threshold decryption while keeping the ring degree and secret-key distributions fixed, since during threshold decryption, we should do noise-flooding, and this needs more modulus than the evaluation circuit. Table 7 summarizes the resulting CKKS parameters and the modulus allocation across the bootstrapping stages.

Table 7. CKKS parameters used in the experiments. An entry $b \times r$ denotes r RNS primes of b bits. We use secret-key Hamming weight $h_{\text{sk}} = 256$ and sparse-secret Hamming weight $\tilde{h} = 32$ [13].

Chain	N	$(h_{\text{sk}}, \tilde{h})$	base	StC	EvalMod	CtS
I	2^{16}	(256, 32)	41	29×3	45×8	36×3
II	2^{15}	(256, 32)	41	29×3	45×8	36×3
III	2^{15}	(256, 32)	41	27×3	42×8	29×3

Parameter set	Chain	Eval	FromCI	ToCI	$\log PQ$
Keccak, Thr.-aware (full batch)	I	36×4	45	36	1303
Keccak, Thr.-aware (inter. batch)	I	36×5	45	36	1350
Keccak, Lat.-aware	I	36×10	–	–	1551
Keccak, Thr.-aware	II	36×4	–	–	833
Keccak, Lat.-aware	III	41×4	–	–	833
WOTS ⁺ signing, ThDec	I	36×11	–	–	1542

E.3 Threshold Parameters Analysis

Baseline. Our proof of concept implementation is n -of- n : the key is additively shared, all reconstruction coefficients are 1, and partial decryption costs no modulus beyond the flooding noise. By the smudging lemma [3, 47], a flooding bound B_{fl} hides a residual error B_e up to statistical distance NtB_e/B_{fl} over the N coefficients and t shares. Requiring this distance to be at most $2^{-\lambda_{\text{sm}}}$ for a statistical parameter λ_{sm} , in our experiment, the decryption modulus becomes

$$\log q \approx \log B_e + \lambda_{\text{sm}} + \log N + 2 \log t + O(1) \approx 99 \text{ bits}$$

at $\lambda_{\text{sm}} = 64$, $N = 2^{16}$, $t = 5$ and $B_e \leq 2^7$, the residual error of our released ciphertexts measured in the coefficient domain.

Lagrange overhead. For t -out-of- n , each partial decryption is scaled by a Lagrange coefficient, which amplifies both the decryption error and the flooding noise. Kim et al. [45] bound this amplification for Shamir sharing in the asynchronous setting, which matches our one-round broadcast: a signer does not know the participating set when it computes its share. Their partial decryption returns $C_{\text{dec}} \cdot a \cdot \text{fsk}_i + C_{\text{sim}} \cdot e_{\text{fl},i} \bmod q$

for two ring constants that clear the denominators of the Lagrange coefficients, and the modulus must satisfy

$$\frac{q}{2p} > \left(B_{\text{eval}} + \frac{p}{4}\right) \cdot (B_1 + 1) + t \cdot B_{\text{eval}} \cdot 2^{\lambda_{\text{sm}}} \cdot N \cdot B_2 \cdot B_3,$$

$$B_1 := \|C_{\text{dec}}\|_1, \quad B_2 := \max_{i, \mathcal{S}} \|C_{\text{sim}} \lambda_i^{\mathcal{S} \rightarrow 0}\|_1, \quad B_3 := \max_{i, \mathcal{S}^*} \|(C_{\text{dec}}/C_{\text{sim}}) \lambda_0^{\mathcal{S}^* \rightarrow \alpha_i}\|_1,$$

all reduced modulo q and maximized over authorized sets $\mathcal{S}$ and maximal corrupted sets $\mathcal{S}^*$. The same bound applies to CKKS with p replaced by the scale Δ . Our n -of- n instantiation has $B_1 = B_2 = B_3 = 1$ and recovers the baseline, so the entire overhead is $B_2 \cdot B_3$, which is independent of the flooding magnitude.

Following [45], we place the parties evenly among the $2N$ monomials, $\alpha_i = x^{(i-1) \cdot 2N/n'}$ with $n' = 2^{\lceil \log n \rceil}$, and add c public *virtual* parties so that the interpolation size becomes $t' = t + c$ without changing the threshold structure. With $C_{\text{dec}} = C_{\text{sim}} = 2^{\lceil \log t' \rceil}$,

$$B_2 \cdot B_3 \leq 2^{\lceil \log t' \rceil} \cdot \frac{n'}{2} \cdot \Psi_{\lfloor t'/2 \rfloor}(n'), \quad \Psi_m(n') := \prod_{k \in [m]} \frac{1}{\sin^2\left(\pi \frac{k}{n'}\right)} \cdot \prod_{k \in [m-1]} \cos^2\left(\pi \frac{k}{n'}\right).$$

Cost. Table 8 evaluates the bound: the overhead peaks near $t \approx n'/2$ and vanishes as $t \rightarrow n$. Share size is proportional to $\log q$, so these ratios are also the communication overhead; on average it stays below one third of the baseline up to $n = 20$, and reaches 90% at $n = 100$. Computation overhead is minor, as a signer is able to evaluate thrifty cleaning with adjustment.

Table 8. Additional decryption-modulus bits for a t -out-of- n threshold and their ratio to the 99-bit baseline of our n -of- n instantiation, using the bound of [45] at $N = 2^{16}$ and $\lambda_{\text{sm}} = 64$. Averages are taken over $t \in \{2, \dots, n\}$.

n	Avg. (bits)	Avg. (%)	Worst t	Worst (bits)	Worst (%)
5	10.3	10	4	12.8	13
10	17.0	17	7	21.2	21
20	27.7	28	13	35.3	36
50	50.5	51	27	67.4	68
100	88.9	90	53	121.9	123

F Additional Details on Homomorphic Keccak

This section details the Keccak round operations and its homomorphic evaluation under the low-amortized-cost lane-wise packing (Section 3). Under XBOOT, XOR operations are evaluated as ciphertext additions, with the deferred modulo-2 reduction performed during bootstrapping.

Each of the 24 rounds of Keccak- $f[1600]$ applies five steps in sequence: $\theta, \rho, \pi, \chi, \iota$. We now describe how each step is realized on the encrypted state.

θ — **column parity XOR**. For given $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_\gamma$, the θ step transforms the state $\mathbf{A}$ into $\mathbf{A}'$ as follows:

$$\begin{aligned} C[x, z] &= \bigoplus_{y=0}^4 \mathbf{A}[x, y, z], \\ D[x, z] &= C[(x-1) \bmod 5, z] \\ &\quad \oplus C[(x+1) \bmod 5, (z-1) \bmod \gamma], \\ \mathbf{A}'[x, y, z] &= \mathbf{A}[x, y, z] \oplus D[x, z]. \end{aligned}$$

In the homomorphic setting, θ is evaluated using only FHE.Add and FHE.Rot operations (no depth consumption) :

$$\begin{aligned} C_x &= \bigoplus_{y=0}^4 A_{x,y}, \\ D_x &= C_{x-1 \bmod 5} \oplus \text{FHE.Rot}_{-b}(C_{x+1 \bmod 5}), \\ A'_{x,y} &= A_{x,y} \oplus D_x. \end{aligned}$$

For each column, the parity value C_x is computed using four FHE.Add operations. The term $C[(x+1) \bmod 5, (z-1) \bmod \gamma]$ corresponds to a one-bit cyclic rotation within a lane. Under the interleaved indexing $\text{idx}(j, z) = j + bz$, this one-bit logical lane rotation corresponds to a physical rotation by $-b$ slots. Computing D_x then requires one additional FHE.Add, and the final update $A'_{x,y}$ requires another FHE.Add.

ρ — **intra-lane bit rotation**. Let $r[x, y]$ denote the rotation-offset table specified by Keccak. The ρ step transforms the state as follows:

$$\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, (z - r[x, y]) \bmod \gamma].$$

In the homomorphic setting, ρ is evaluated using slot rotations:

$$A'_{x,y} = \text{FHE.Rot}_{-r[x,y]b}(A_{x,y}), \quad x, y \in \mathbb{Z}_5.$$

Since adjacent bits within a lane are separated by the bit stride b , a cyclic rotation by $r[x, y]$ bits corresponds to a slot rotation by $r[x, y]b$ positions. Among the 25 lanes, only lane $(0, 0)$ has rotation offset 0; therefore, the ρ step requires 24 FHE.Rots in total.

π — **lane permutation**. The π step permutes the lane positions according to

$$\mathbf{A}'[x, y, z] = \mathbf{A}[(x+3y) \bmod 5, x, z].$$

In the homomorphic setting, this operation is implemented by reindexing the ciphertexts corresponding to the 25 lanes. Since each lane is stored in a separate ciphertext, no homomorphic operation is required.

χ — **nonlinear substitution**. The χ step applies a nonlinear transformation to each row:

$$\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus \left(\neg \mathbf{A}[(x+1) \bmod 5, y, z] \wedge \mathbf{A}[(x+2) \bmod 5, y, z] \right).$$

For each lane (x, y) , let $A_{x,y}$ denote the ciphertext packing the bits $\{A_j[x, y, z]\}_{j,z}$ according to the lane-wise layout of Section 3.1. χ is evaluated as

$$A'_{x,y} = A_{x,y} \oplus \text{FHE.Mult}(1 - A_{(x+1) \bmod 5, y}, A_{(x+2) \bmod 5, y}).$$

Consequently, χ is the only nonlinear step that consumes a multiplicative depth. The final XOR with $A_{x,y}$ is evaluated by one FHE.Add .

ι — **round constant addition**. The ι step injects a round-dependent constant into lane $(0, 0)$:

$$\mathbf{A}'[0, 0, z] = \mathbf{A}[0, 0, z] \oplus RC_r[z],$$

where RC_r denotes the 64-bit round constant for round r .

In the homomorphic setting, RC_r is encoded as a plaintext and added to the ciphertext corresponding to lane $(0, 0)$. Thus, ι requires a single FHE.Add .

Table 9 summarizes the homomorphic operation counts for one Keccak round. Among the five steps, only χ consumes multiplicative depth due to its nonlinear multiplication. FHE.Rots are required only in θ and ρ , which account for all 29 rotations per round.

F.1 Counting Homomorphic Operations

Table 9 counts operations for one round of Keccak- $f[1600]$ under throughput-aware packing.

Table 9. Homomorphic operation counts for one round of Keccak- $f[1600]$ under throughput-aware packing.

Step	#XOR (FHE.Add)	#AND (FHE.Mult)	FHE.Rot
θ	50	0	5
ρ	0	0	24
π	0	0	0
χ	25	25	0
ι	1	0	0
Total per round	76	25	29

F.2 Breakdown of Keccak Permutation

Table 10 provides latency breakdown of each Keccak permutation step and bootstrapping under throughput-aware packing, with full batches.

Table 10. Latency breakdown of one homomorphic Keccak- $f[1600]$ evaluation for $b = 512$.

Step	Latency (s)	%
θ	0.025	1.4
ρ	0.056	3.2
π	-	-
χ	0.175	10.0
ι	-	-
Bootstrapping	1.495	85.3
Total	1.752	

F.3 Message Magnitude and Error Analysis for Homomorphic Keccak Permutation

We track the message magnitude and ciphertext error throughout one bootstrapping interval on the Keccak permutation. Let $\eta := 2^{-P}$ denote the error of a freshly bootstrapped bit ciphertext, and let ε denote the additive error introduced by key switching or rescaling. Before the deferred modulo-2 reduction, XOR gates are evaluated as integer additions, while the only nonlinear term in χ is $(1 - a_1) \cdot a_2$. For inputs $m_1 + e_1$ and $m_2 + e_2$, the dominant multiplication error bound satisfies

$$\|m_1 e_2 + m_2 e_1 + e_1 e_2\|_\infty \leq \|m_1\|_\infty \|e_2\|_\infty + \|m_2\|_\infty \|e_1\|_\infty + \|e_1\|_\infty \|e_2\|_\infty.$$

Applying this recurrence across the Keccak transformations gives the step-by-step growth in Table 11. The table reports the dominant terms in η ; higher-order terms and the smaller additive contributions in ε are omitted for readability. The interval begins at the χ step of round r , immediately after bootstrapping, and ends after the ρ and π steps of round $r + 2$, immediately before the next bootstrapping.

F.4 Key-Switching Comparison

For small batches, we consider two circuit realizations. The first switches to the latency-aware layout, which evaluates lane permutations directly within a single ciphertext and avoids Pack/Unpack altogether. The second retains the throughput-aware lane-wise layout and packs the sparsely populated lane ciphertexts into fewer ciphertexts before bootstrapping, followed by unpacking to recover the original layout; this reduces the number of ciphertexts processed by bootstrapping but introduces additional key switches for layout conversion.

Table 12 compares these two realizations with the lane-wise evaluation of Seuré and Suvanto [61]. The latency-aware realization requires less rotations for the linear steps, reduces number of multiplication in χ , and eliminates Pack/Unpack. Comparing [61] and our latency-aware packing, the total key-switching count 2 448 decreases to 764.

Table 11. Step-by-step growth of the message magnitude and dominant approximation error over one bootstrapping interval ($\eta = 2^{-p}$). Higher-order and key-switching errors are omitted. The three blocks cover $\chi \rightarrow \iota$ of round r , the full round $r + 1$, and $\theta \rightarrow \pi$ of round $r + 2$, respectively; a_0, a_1, a_2 denote the lanes $A[x, y]$, $A[x+1, y]$, and $A[x+2, y]$.

Round	Step	Op.	$\ m\ _\infty$	$\ e\ _\infty$
–	Input	A	1	η
r	χ	$\bar{a}_1 = 1 - a_1$	1	η
	χ	$p = \bar{a}_1 \cdot a_2$	1	$2^1 \eta$
	χ	$B = p \oplus a_0$	$1^\dagger$	$2^{1.6} \eta$
	ι	$A = B \oplus RC_r$	2^1	$2^{1.6} \eta$
$r + 1$	θ	$C = \bigoplus_{y \in \mathbb{Z}_5} A[x, y]$	$2^{3.3}$	$2^{3.9} \eta$
	θ	$D = C \oplus \text{rot}(C)$	$2^{4.3}$	$2^{4.9} \eta$
	θ	$A = A \oplus D$	$2^{4.5}$	$2^{5.0} \eta$
	ρ, π	$A = \text{rot}(A)$	$2^{4.5}$	$2^{5.0} \eta$
	χ	$\bar{a}_1 = 1 - a_1$	$2^{4.5}$	$2^{5.0} \eta$
	χ	$p = \bar{a}_1 \cdot a_2$	$2^{9.0}$	$2^{10.5} \eta$
	χ	$B = p \oplus a_0$	$2^{9.0}$	$2^{10.6} \eta$
	ι	$A = B \oplus RC_{r+1}$	$2^{9.0}$	$2^{10.6} \eta$
$r + 2$	θ	$C = \bigoplus_{y \in \mathbb{Z}_5} A[x, y]$	$2^{11.4}$	$2^{12.9} \eta$
	θ	$D = C \oplus \text{rot}(C)$	$2^{12.4}$	$2^{13.9} \eta$
	θ	$A = A \oplus D$	$2^{12.5}$	$2^{14.0} \eta$
	ρ, π	$A = \text{pack}(\text{rot}(A)) (\rightarrow \text{BTS})$	$2^{12.5}$	$2^{14.0} \eta$

[†] XOR is performed by subtraction when the input messages lie in $\{0, 1\}$, since the two operations agree after modular reduction during bootstrapping, thereby keeping the message magnitude smaller.

F.5 Latency Comparison

Table 13 compares our two packings with the lane-wise evaluation of Seuré and Suvanto [61] at the same ring degree and modulus chain. The full-state packing, which places one Keccak state in a single ciphertext, gives the lowest single-evaluation latency, whereas the full-batch packing fills all slots and therefore gives the lowest amortized cost per state. Table 14 breaks the amortized improvement at $\log N = 16$ down by optimization.

G XMSS

XMSS [15, 34] is a stateful hash-based signature scheme standardized in RFC 8391 [34]. It combines WOTS⁺ with a Merkle tree to enable 2^h signatures under a single public key.

Table 12. Circuit-level key-switching operations, excluding those internal to bootstrapping, over one 24-round Keccak- $f[1600]$ evaluation. The throughput-aware column represent the full batch.

	[61]	Ours (Latency-aware)	Ours (Throughput-aware)
θ	120	300	120
ρ	576	312	576
Pack/Unpack	1152	0	0 [†]
χ -alignment	0	128	0
χ	600	24	600
Cleaning	0	0	600 [†]
Total	2 448	764	1 896[†]

[†] For intermediate batches, the lanes are packed into fewer ciphertexts before bootstrapping: Unpack then adds key switches (Pack is fused into the ρ -step) while cleaning, which acts on the packed ciphertexts, needs fewer; both depend on the batch size.

Table 13. Latency of one Keccak- $f[1600]$ evaluation and the amortized latency per Keccak state, for $\log N \in \{15, 16\}$ under the same modulus chain. The number of states evaluated in parallel is 8 (resp. 16) for the full-state packing and 256 (resp. 512) for the full-batch packing at $\log N = 15$ (resp. 16).

	$\log N = 15$		$\log N = 16$	
	Lat. (s)	Amor. (ms)	Lat. (s)	Amor. (ms)
Seuré and Suvanto [61]	0.360	45.0	0.470	29.4
Ours (Latency-aware)	0.198	24.8	0.263	16.4
Ours (Thr.-aware, full batch)	1.362	5.32	1.751	3.42
Improvement	1.82×	8.46×	1.79×	8.60×

Both figures of [61] are our reproduction of their design on the same GPU; the $\log N = 16$ one uses the same modulus chain with $dnum = 1$. Our latencies count the final bootstrapping only up to its S2C, as in Table 3. The improvement row compares latency-aware, throughput-aware packings and [61] in each metric.

G.1 Hash Instantiation and Notation

Both schemes are instantiated with SHAKE256/256 [52], which we write as

$$\text{S256}(M) = \text{SHAKE256}(M, 8n),$$

so that every hash invocation reduces to a single SHAKE256/256 call. We use the XMSS-SHAKE256_10_256 parameter set and the SHAKE-based fast profile SPHINCS⁺-256f, both summarized in Table 6.

Key structure. The secret key is $(\text{sk}_{\text{seed}}, \text{sk}_R, \text{pp}_{\text{seed}}, \text{pk})$ and the public key is $(\text{pp}_{\text{seed}}, \text{pk})$, all n -byte values. Only sk_{seed} and sk_R must remain encrypted in the threshold protocol.

Table 15 summarizes the key components and the hash functions of the two schemes.

Table 14. Incremental evaluation of the throughput-aware optimizations at $\log N = 16$. S2C, C2S and EvalMod count the bootstrapping-internal linear transforms and modular reductions over one Keccak- f [1600] evaluation, where the final bootstrapping is counted only up to its S2C (as in Table 3). Amor. is the latency per state.

	Configuration	Batch	S2C	C2S	EvalMod	Latency (s)	Amor. (ms)	Speedup
L1	Lane-wise [61]	16	24	24	24	0.470	29.4	1.0×
L2	L1 + full batch	512	625	600	600	4.422	8.64	3.4×
L3	L2 + complex packing	512	325	312	624	3.555	6.94	4.2×
L4	L3 + 1 BTS / 2 rounds	512	169	156	312	2.026	3.96	7.4×
L5a	L4 + CI-ring EvalMod	512	169	156	156	1.832	3.58	8.2×
L5b	L4 + shared- a S2C,C2S	512	26*	24*	312	1.993	3.89	7.6×
L6	Ours, Thr.-aware	512	26*	24*	156	1.751	3.42	8.6×

* on MSRLWE ciphertexts of 7 members. L1–L3 bootstrap after every round on a 3-level evaluation chain; L4–L6 bootstrap every two rounds on the 4-level chain of Table 7. All rows were measured back to back on one GPU.

G.2 WOTS⁺

WOTS⁺ [36] signs a message by interpreting it as base- w digits and applying a hash chain to each digit.

Parameters. Let n be the hash output length in bytes, w the Winternitz parameter, and define, following RFC 8391 [34]:

$$len_1 = \left\lceil \frac{8n}{\log_2 w} \right\rceil, \quad len_2 = \left\lceil \frac{\log_2(len_1(w-1))}{\log_2 w} \right\rceil + 1,$$

and $len = len_1 + len_2$. The first len_1 chains carry the base- w digits of the message digest, and the remaining len_2 chains carry the base- w digits of the checksum. With $n = 32$ and $w = 16$ this gives $len = 64 + 3 = 67$ chains.

Key generation. The secret key $sk_0, \dots, sk_{len-1}$ is derived from a seed via PRF_{seed} . The public key is:

$$pk_i = F^{w-1}(sk_i), \quad i = 0, 1, \dots, len - 1.$$

Signing. To sign a message digest M of n bytes:

1. Compute the base- w digits $(B_0, \dots, B_{len_1-1}) = \text{base}_w(M)$ of the digest.
2. Compute the checksum $S = \sum_{i=0}^{len_1-1} (w-1-B_i)$ and its base- w digits $(B_{len_1}, \dots, B_{len-1}) = \text{base}_w(S)$.
3. Output $\sigma = (\sigma_0, \sigma_1, \dots, \sigma_{len-1})$ where $\sigma_i = F^{B_i}(sk_i)$.

Signing is deterministic given M : no auxiliary randomness is sampled and no rejection loop is required, which is what allows the whole encoding to be evaluated as a fixed-depth homomorphic circuit (Section 4.2).

Verification. Given a signature $(\sigma_0, \dots, \sigma_{len-1})$ and message digest M , recompute $(B_0, \dots, B_{len-1})$ exactly as in signing (including the checksum digits) and check:

$$F^{w-1-B_i}(\sigma_i) \stackrel{?}{=} pk_i, \quad \forall i = 0, 1, \dots, len - 1.$$

Table 15. Notation for XMSS and SPHINCS⁺: key components (top) and hash functions (bottom). All hash functions are instantiated with SHAKE256/256, written S256.

Component	Description	
sk_{seed}	Secret seed (OTS/FORS keys)	
sk_R	PRF key (per-signature randomizer)	
pp_{seed}	Public seed (domain separation)	
pk_{root}	Tree root	
pk	Public key (pp_{seed}, pk_{root})	
idx	Leaf index (state for XMSS)	
idx_{tree}	Hypertree subtree index (for SPHINCS ⁺)	
idx_{leaf}	Hypertree subtree leaf index (for SPHINCS ⁺)	
Purpose	XMSS	SPHINCS ⁺
WOTS ⁺ chain hash	F ($d=0$)	$F = T_1$
L-tree / Merkle node hash	H ($d=1$)	T_2 (Merkle)
WOTS ⁺ pk compression	(via L-tree)	T_{len}
FORS root compression	—	T_k
Message digest	H_{msg} ($d=2$)	H_{msg} (49 B)
Per-signature randomizer	PRF_R ($d=3$)	PRF_R
OTS/FORS secret key generation	PRF_{seed} ($d=4$)	PRF_{seed}

All key components are n -byte values. In XMSS, all hash functions share a common SHAKE256/256 template $S256(tB(d, 32) || KEY || M)$ distinguished by a domain-separation byte $d \in \{0, \dots, 4\}$ and use a prefix-and-bitmask construction (Appendices G.3 and G.5). In SPHINCS⁺, $F = T_1, T_2, T_{len}$, and T_k are members of a tweakable hash family $T_\ell(pp_{seed}, ADRS, M) = S256(pp_{seed} || ADRS || M)$ (Appendix H.4); $PRF_{seed} = S256(pp_{seed} || ADRS || sk_{seed})$; $PRF_R = S256(sk_R || oprand || M)$; $H_{msg} = SHAKE256(R || pk || M, 8m)$ where $pk = (pp_{seed}, pk_{root})$. All outputs are n -byte values except H_{msg} in SPHINCS⁺, which returns $m = 49$ bytes for SLH-DSA-SHAKE-256f and is split into md , idx_{tree} and idx_{leaf} .

Role of the checksum. An attacker holding $\sigma_i = F^{B_i}(sk_i)$ can advance any chain forward, i.e. forge any digest whose digits satisfy $B'_i \geq B_i$ for all i . The checksum rules this out: increasing a message digit strictly decreases S and hence decreases at least one checksum digit, so every distinct digit vector requires inverting F on at least one chain. The guarantee costs $len_2 = 3$ additional chains for $w = 16$.

G.3 Prefixes and Bitmasks

Every hash invocation in XMSS is domain-separated by a 32-byte *address* (ADRS) that encodes the position within the scheme. A *prefix* (key) and a *bitmask* are derived from ADRS and the public seed pp_{seed} . The chaining function F computes:

$$F(x) = S256(tB(0, 32) || KEY || (x \oplus \text{bitmask})),$$

where $KEY = PRF_R(pp_{seed}, ADRS)$ and $\text{bitmask} = PRF_R(pp_{seed}, ADRS')$, with $ADRS'$ differing only in the key-and-mask field. The tree hash H applies the same construction with separate bitmasks for left and right children. This reduces the security assumption to second-preimage resistance and PRF security [15, 36].

G.4 Many-Time via Merkle Tree

XMSS uses a Merkle tree of height h to authenticate 2^h one-time public keys under a single root.

Tree construction. Each leaf compresses the len WOTS⁺ public key components into a single n -byte value via an L -tree (a binary hash tree over the components):

$$\text{leaf}_i = \text{Ltree}(\text{pk}_i^{\text{OTS}}).$$

Internal nodes are computed as:

$$\text{node}_{j,k} = H(\text{node}_{j-1,2k} \parallel \text{node}_{j-1,2k+1}),$$

where j is the tree level and k the node index. The public key is the root $\text{node}_{h,0}$.

Signing. To produce the i -th XMSS signature:

1. Generate a per-signature randomizer $R = \text{PRF}_R(\text{sk}_R, i)$ and compute the message digest $M' = H_{\text{msg}}(R \parallel \text{pk} \parallel i, M)$.
2. Generate a WOTS⁺ signature σ_i^{OTS} on M' using OTS_i .
3. Compute the *authentication path* $\text{AP}_i = (a_0, a_1, \dots, a_{h-1})$, where a_j is the sibling of the j -th ancestor of leaf_i .
4. Increment the leaf index: $\text{idx} \leftarrow i + 1$.
5. Output $\sigma = (i, R, \sigma_i^{\text{OTS}}, \text{AP}_i)$.

Verification. The verifier recomputes M' , recovers pk_i^{OTS} from σ_i^{OTS} , reconstructs the root via the authentication path, and accepts iff it equals pk .

Stateful nature. XMSS is *stateful*: the signer must track the leaf index i and never reuse a key pair. This is the central challenge for thresholdizing XMSS, as multiple signers must coordinate to avoid index reuse.

Multi-tree variant (XMSS^{MT}). XMSS^{MT} extends XMSS to d layers of Merkle trees, each of height h/d , allowing 2^h signatures at the cost of d OTS signatures and d authentication paths per signature.

G.5 Parameter Sets and Instantiation with SHAKE256/256

We follow RFC 8391 [34] and instantiate it with SHAKE256/256 [52] ($n = 32$ bytes), i.e. the XMSS-SHAKE256_10_256 parameter set of NIST SP 800-208 [20]. RFC 8391 leaves the derivation of the WOTS⁺ secret keys from a seed to the implementation; we use PRF_{seed} (domain byte $d = 4$) as specified in [20], a choice that is internal to the signer and does not affect verification. All hash functions $F, H, H_{\text{msg}}, \text{PRF}_R, \text{PRF}_{\text{seed}}$ share a common template with a domain-separation byte d :

$$f(\text{KEY}, M) = \text{S256}(\text{tB}(d, 32) \parallel \text{KEY} \parallel M, 256)$$

where $\text{tB}(x, y)$ is the y -byte big-endian encoding of x , and the domain byte $d \in \{0, \dots, 4\}$ selects the function as listed in Table 15.

Key structure. The secret key consists of $\text{sk}_{\text{seed}}, \text{sk}_R$, and the leaf index idx ; the public key is $(\text{pk} = \text{node}_{h,0}, \text{pp}_{\text{seed}})$. All seeds are n -byte values.

H SPHINCS⁺

SPHINCS⁺ [9], standardized as FIPS 205 (SLH-DSA) [54], is a *stateless* hash-based signature scheme. It combines a hypertree of Merkle trees, a few-time signature scheme (FORS), and pseudorandom leaf selection to support an unbounded number of signatures without per-signature state.

Merkle Tree Helper Algorithms. Throughout, $\text{TreeRoot}_\ell(\{\ell_i\})$ denotes the root obtained from a sequence of leaves, $\text{TreeRoot}_p(\ell, \text{AP})$ the root recomputed from a leaf and its authentication path, and $\text{TreePath}(\text{idx}, \text{aux})$ the authentication path of leaf idx given a description aux of the tree.

H.1 WOTS⁺ in SPHINCS⁺

SPHINCS⁺ reuses WOTS⁺ (Appendix G.2) with two differences: (i) secret keys are derived via $\text{PRF}_{\text{seed}}(\text{sk}_{\text{seed}}, \text{ADRS})$ where ADRS identifies the leaf within the hypertree, and (ii) the chaining function uses a tweakable hash with *simple* or *robust* variants (Appendix H.4).

H.2 FORS

FORS (Forest Of Random Subsets) is a few-time signature scheme parameterized by (k, a) : k Merkle trees of height a , each with 2^a pseudorandom leaves.

Key generation. For each tree $j \in \{0, \dots, k-1\}$, the 2^a secret values

$$\text{sk}_{j,u} = \text{PRF}_{\text{seed}}(\text{sk}_{\text{seed}}, \text{ADRS}_{j,u}), \quad u = 0, \dots, 2^a - 1,$$

are derived from sk_{seed} and hashed into leaves $\text{leaf}_{j,u} = F(\text{pp}_{\text{seed}}, \text{ADRS}_{j,u}, \text{sk}_{j,u})$. The FORS public key is:

$$\text{pk}^{\text{FORS}} = T_k(\text{pp}_{\text{seed}}, \text{ADRS}^{\text{roots}}, \text{root}_0 \parallel \dots \parallel \text{root}_{k-1}).$$

Signing. To sign a ka -bit message digest D , partition D into k chunks $(d_0, d_1, \dots, d_{k-1})$ of a bits each, interpret each d_j as an integer in $[0, 2^a)$, and output

$$\sigma^{\text{FORS}} = ((\text{sk}_{0,d_0}, \text{AP}_{0,d_0}), \dots, (\text{sk}_{k-1,d_{k-1}}, \text{AP}_{k-1,d_{k-1}})),$$

i.e., the secret leaf at position d_j in tree j together with its authentication path.

H.3 Hypertree

SPHINCS⁺ arranges d layers of Merkle trees, each of height $h' = h/d$, into a *hypertree* of total height h . Bottom-layer WOTS⁺ keys sign FORS public keys; upper-layer keys sign the root of the layer below. The public key is the root of the topmost tree. Leaves are computed via T_{len} (compressing len chain heads) and internal nodes via T_2 .

XMSS.Sign(sk, m, idx) <pre> 1 : // idx ≠ ⊥ as a stateful signature 2 : // aux describes the Merkle tree whose root is pk 3 : // R is a randomizer derived from sk 4 : σ₀ ← WOTS⁺.Sign(sk_{idx}, $\tilde{m}$, ⊥) 5 : σ₁ ← TreePath(idx, aux) 6 : return (R, σ₀, σ₁) </pre>
XMSS.Verify(pk, m, σ) <pre> 1 : (R, σ₀, σ₁) ← σ 2 : // The following specific tree is called the L-tree 3 : $\tilde{\ell}$ ← TreeRoot_ℓ(WOTS⁺.VerifyComp($\tilde{m}$, σ₀)) 4 : return pk = TreeRoot_p($\tilde{\ell}$, σ₁) </pre>
SPHINCS⁺.Sign(sk, m, idx) <pre> 1 : // idx = ⊥ as a stateless signature 2 : // Instead, we derive idx₁, ..., idx_d 3 : // from the message for d subtrees 4 : // R is a randomizer derived from sk, m 5 : σ₀ ← FORS.Sign(sk₀, $\tilde{m}$, ⊥) 6 : r_d ← H(FORS.VerifyComp($\tilde{m}$, σ₀)) 7 : σ_{d,1} ← WOTS⁺.Sign(sk_d, r_d, ⊥) 8 : σ_{d,2} ← TreePath(idx_d, aux_d) 9 : for i ∈ {d − 1, ..., 1}, do 10 : r_i ← TreeRoot_ℓ(aux_{i+1}) 11 : σ_{i,1} ← WOTS⁺.Sign(sk_i, r_i, ⊥) 12 : σ_{i,2} ← TreePath(idx_i, aux_i) 13 : return (R, σ₀, {σ_{i,1}, σ_{i,2}}_{i=1}^d) </pre>
SPHINCS⁺.Verify(pk, m, σ) <pre> 1 : (R, σ₀, {σ_{i,1}, σ_{i,2}}_{i=1}^d) ← σ 2 : r_d ← H(FORS.VerifyComp($\tilde{m}$, σ₀)) 3 : $\tilde{\ell}_d$ ← H(WOTS⁺.VerifyComp(r_d, σ_{d,1})) 4 : for i ∈ {d − 1, ..., 1}, do 5 : r_i ← TreeRoot_p($\tilde{\ell}_{i+1}$, σ_{i+1,2}) 6 : $\tilde{\ell}_i$ ← H(WOTS⁺.VerifyComp(r_i, σ_{i,1})) 7 : return pk = TreeRoot_p($\tilde{\ell}_1$, σ_{1,2}) </pre>

Fig. 5. XMSS and SPHINCS⁺ per the syntax of our work. While the key sk_{idx} is derived from sk and idx in XMSS, the keys $sk_0, \dots, sk_d$ are derived from sk and m in SPHINCS⁺. Also, aux_i describes the i -th subtree for $i \in [d]$, and $\tilde{m}$ is derived from m and randomizer R . The corresponding Setup algorithms are described in Section 2.3.

H.4 Tweakable Hash and the Simple/Robust Variants

SPHINCS⁺ uses a *tweakable hash* family $T_\ell : \{0, 1\}^n \times \text{ADRS} \times \{0, 1\}^{\ell n} \rightarrow \{0, 1\}^n$ with two instantiations:

- *Simple*: $T_\ell^{\text{simple}}(\text{pp}_{\text{seed}}, \text{ADRS}, M) = \text{S256}(\text{pp}_{\text{seed}} \parallel \text{ADRS} \parallel M)$.
- *Robust*: $T_\ell^{\text{robust}}(\text{pp}_{\text{seed}}, \text{ADRS}, M) = \text{S256}(\text{pp}_{\text{seed}} \parallel \text{ADRS} \parallel (M \oplus \text{bitmask}))$,
where $\text{bitmask} = \text{SHAKE256/256}(\text{pp}_{\text{seed}} \parallel \text{ADRS}, 8\ell n)$.

Simple instantiation is standardized in SLH-DSA [54].

H.5 Signing

A SPHINCS⁺ signature is produced as follows:

1. Generate a per-signature randomizer $R = \text{PRF}_R(\text{sk}_R, \text{oprand}, M)$.
2. Compute $(\text{md} \parallel \text{idx}_{\text{tree}} \parallel \text{idx}_{\text{leaf}}) = \text{H}_{\text{msg}}(R, \text{pk}, M)$.
3. Sign md with the FORS instance at $(\text{idx}_{\text{tree}}, \text{idx}_{\text{leaf}})$, producing σ^{FORS} and pk^{FORS} .
4. Sign pk^{FORS} with the corresponding WOTS⁺ key and propagate authentication paths up through all d hypertree layers.
5. Output $\sigma = (R, \sigma^{\text{FORS}}, (\sigma^{(0)}, \text{AP}_0, \dots, \sigma^{(d-1)}, \text{AP}_{d-1}))$.

Stateless property. The leaf indices are derived deterministically from the message via H_{msg} , eliminating per-signature state. However, every signature recomputes a FORS instance and d WOTS⁺ signatures, so the homomorphic cost is dominated by hash evaluations across thousands of chains.

I Joint Signing Key Generation

In the following, we represent each 256-bit value by $\text{lane} = 4$ ciphertexts. The 256 bits are partitioned into four consecutive 64-bit blocks, and each block is packed using the same lane-wise representation as in the homomorphic Keccak evaluator. We write

$$\mathbf{ct} := (\text{ct}^{(0)}, \text{ct}^{(1)}, \text{ct}^{(2)}, \text{ct}^{(3)}) \quad (2)$$

for the resulting tuple of lane ciphertexts and treat it as a single logical object.

Each (semi-honest) signer broadcasts an encryption of a partial-secret polynomial whose coefficients lie in $\{0, \Delta\}$ with $\Delta = q_0/2$; summing ciphertexts modulo q_0 acts as coefficient-wise XOR of underlying bits, so the aggregate is uniformly random.

Algorithm 1 gives the protocol in detail; each sampled bit is replicated across the b slots of its lane position (cf. Section 3.1) so that all batched states share the same seed.

J Batched Public-Key Generation

This appendix details $\text{PKGen}_{\text{sig}}$ (Algorithm 3), which homomorphically derives the WOTS⁺ chain states, compresses their heads into leaf nodes, and constructs the public-key tree. Independent hash computations within each stage are evaluated in parallel using the low-amortized-cost packing.

Algorithm 1 SKGen_{sig}: signature secret key generation.

Input: Signers $\mathcal{S}$ having CKKS public key fpk
Output: Lane bundles $\mathbf{ct}_{\text{sk}_{\text{seed}}}, \mathbf{ct}_{\text{sk}_R}$ encrypting sk_{seed} and sk_R

```

1: for each signer  $\mathcal{P}_j \in \mathcal{S}$  do
2:   for  $k = 1, \dots, 4$  do
3:     Sample random bit vector  $\beta_j^{(k)}$ 
4:      $m_j^{(k)} \leftarrow \text{CoeffEcd}(\beta_j^{(k)}) \in R_{q_0}$ 
5:      $\mathbf{c}_j^{(k)} \leftarrow \text{FHE.Enc}_{\text{pk}}(m_j^{(k)})$ 
6:     Broadcast  $\mathbf{c}_j^{(k)}$ 
7:   end for
8: end for
9: for  $k = 1, \dots, 4$  do
10:   $\mathbf{c}^{(k)} \leftarrow \sum_j \mathbf{c}_j^{(k)}$ 
11:   $(\mathbf{u}^{(2k-1)}, \mathbf{u}^{(2k)}) \leftarrow \text{halfBTS}(\mathbf{c}^{(k)})$ 
12: end for
13:  $\mathbf{ct}_{\text{sk}_{\text{seed}}} \leftarrow (\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(4)}); \mathbf{ct}_{\text{sk}_R} \leftarrow (\mathbf{u}^{(5)}, \dots, \mathbf{u}^{(8)})$ 
14: Return  $\mathbf{ct}_{\text{sk}_{\text{seed}}}, \mathbf{ct}_{\text{sk}_R}$ 

```

Batched WOTS⁺ Chain Evaluation. The dominant computation in PKGen_{sig} is the evaluation of WOTS⁺ chains for all leaves. Each chain starts from a secret-key element derived homomorphically from sk_{seed} using a SHAKE256/256-based pseudorandom function. Since computations are independent across leaves and chain indices, we evaluate them using the low-amortized-cost packing. A batch of capacity b packs the len chains from each of $L = \lfloor b/\text{len} \rfloor$ leaves, evaluating $L\text{len}$ chains in parallel and padding unused slots with zeros.

Using this layout, WOTS⁺.BatchSetup (Algorithm 2 and Figure 6) advances all packed chains by one F -step through a single homomorphic SHAKE256/256 evaluation. Repeating this process for $w - 1$ steps produces the chain heads, while the intermediate states are retained for subsequent WOTS⁺ signing.

Algorithm 2 WOTS⁺.BatchSetup: Homomorphic chain evaluation.

Input: Encrypted chain starts $\{\mathbf{ct}_{\text{sk}_b}\}_{b \in [B]}$ in the leaf-major layout (slot $\text{len} \cdot k + i$ of $\mathbf{ct}_{\text{sk}_b}$ encrypts the chain- i start of the k -th leaf in batch b); Winternitz parameter w
Output: Chain heads $\{\mathbf{ct}_{\text{pk}_b}\}_{b \in [B]}$ in the same layout, together with all intermediate chain blocks

```

1:  $\{\mathbf{ct}_{b,s}\}_{b \in [B], 0 \leq s < w}$ 
2:  $\mathbf{ct}_{b,0} \leftarrow \mathbf{ct}_{\text{sk}_b}$  for all  $b \in [B]$ 
3: for  $s = 1, \dots, w-1$  do                                      $\triangleright B$  batched Keccak perms per step
4:    $\mathbf{ct}_{b,s} \leftarrow F(\mathbf{ct}_{b,s-1})$  for all  $b \in [B]$               $\triangleright \text{SIMD across all } K_{\text{max}} \cdot \text{len chain slots}$ 
5: end for
6: return  $(\{\mathbf{ct}_{\text{pk}_b} := \mathbf{ct}_{b,w-1}\}_{b \in [B]}, \{\mathbf{ct}_{b,s}\}_{b \in [B], 0 \leq s < w})$ 

```

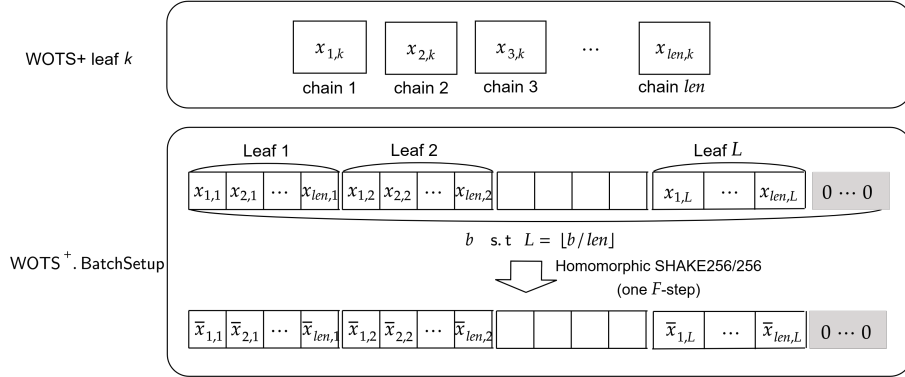


Fig. 6. Batched homomorphic evaluation of WOTS⁺ chains. Up to $L = \lfloor b/len \rfloor$ leaves are packed into one batch, and one homomorphic SHAKE256/256 evaluation advances all packed chain states by one F -step in parallel.

Batched Leaf and Tree Construction. After the batched WOTS⁺ chain evaluation, each signer completes the public-key generation as follows:

- **Leaf compression (LeafCompress).** For each leaf, the len chain heads forming its WOTS⁺ public key are compressed into a single n -byte leaf node. XMSS uses an L-tree that recursively hashes adjacent chain heads with H , whereas SPHINCS⁺ applies the tweakable hash T_{len} . The L packed leaf computations are evaluated in parallel.
- **Merkle-tree construction (TreeRoot).** At each tree level, adjacent encrypted nodes are hashed using H for XMSS and T_2 for SPHINCS⁺, with independent node computations evaluated in parallel. This process is repeated until the root is obtained. Intermediate encrypted nodes are retained for authentication-path selection during signing, and the signers jointly decrypt only the final root to obtain the public-key root. For SPHINCS⁺, only the top hypertree layer is constructed during $\text{PKGen}_{\text{sig}}$.

K Homomorphic WOTS⁺ Signing

This appendix gives the packing and the concrete procedure behind the encrypted WOTS⁺ signing step of Section 4.2.

Base- w Encoding. The signers first decompose the encrypted 256-bit message digest M' (which is stored in the lane-wise representation of Equation (2)) into 64 base- w digits $B_0, \dots, B_{63}$. To obtain each digit, signers apply plaintext masks encoding the weights $(1, 2, 4, 8)$ to consecutive groups of four bits and then combine the weighted bits through two rotation-and-add steps. The resulting ciphertext slots contain the encrypted base- w digits $B_0, \dots, B_{63}$.

Algorithm 3 PKGen_{sig}: Homomorphic threshold public key generation.**Input:** $\mathbf{ct}_{\mathbf{sk}_{\text{seed}}}$ (encrypted $\mathbf{sk}_{\text{seed}}$); height h ; $\mathbf{pp}_{\text{seed}}$; $B = \lceil 2^h / K_{\max} \rceil$ **Output:** $\mathbf{pk}$; for XMSS, the encrypted hash chains $\mathbf{C}$ and tree nodes $\mathcal{T}$ *Step 1: Batched WOTS⁺ chain evaluation*

- 1: **for** leaf batch $b \in [B]$ **do**
- 2: $\mathbf{ct}_{\mathbf{sk}_b} \leftarrow \text{PRF}_{\text{seed}}(\mathbf{ct}_{\mathbf{sk}_{\text{seed}}}, \{\text{ADRS}(bK_{\max} + k, j)\}_{k \in [K_{\max}], j \in [\text{len}]})$
- 3: **end for**
- 4: $(\{\mathbf{ct}_{\mathbf{pk}_b}\}_{b \in [B]}, \mathbf{C}) \leftarrow \text{WOTS}^+. \text{BatchSetup}(\{\mathbf{ct}_{\mathbf{sk}_b}\})$ ▷ Algorithm 2; $\mathbf{C} = \{\mathbf{ct}_{b,s}\}$

Step 2: Leaf compression

- 5: **for** leaf batch $b \in [B]$ **do**
- 6: $\mathbf{ct}_{\text{leaf}_b} \leftarrow \text{LeafCompress}(\mathbf{ct}_{\mathbf{pk}_b})$
- 7: **end for**

Step 3: Merkle tree

- 8: $(\mathbf{ct}_{\text{root}}, \mathcal{T}) \leftarrow \text{TreeRoot}_\ell(\{\mathbf{ct}_{\text{leaf}_b}\}_{b \in [B]})$ ▷ $\mathcal{T}$: encrypted merkle tree nodes

Step 4: Root extraction

- 9: $\mathbf{pk}_{\text{root}} \leftarrow \text{ThDec}(\mathbf{ct}_{\text{root}})$, $\mathbf{pk} = (\mathbf{pp}_{\text{seed}}, \mathbf{pk}_{\text{root}})$
- 10: **return** $\begin{cases} (\mathbf{pk}, \mathbf{C}, \mathcal{T}) & \text{if XMSS} \\ (\mathbf{pk}, \perp) & \text{if SPHINCS}^+ \end{cases}$

Next, the signers compute the checksum

$$S = \sum_{i=0}^{63} (w - 1 - B_i) = \sum_{i=0}^{63} (15 - B_i)$$

homomorphically. Since each digit satisfies $0 \leq B_i < 16$, the checksum is bounded by $S \leq 64 \cdot 15 < 2^{10}$ and therefore fits within three base- w digits. The checksum digits are extracted homomorphically through successive IntMod evaluations (see Appendix C), yielding B_{64}, B_{65} , and B_{66} . The resulting encrypted digits $B_0, \dots, B_{66}$ determine the chain positions used in the subsequent WOTS⁺ signature generation step.

Chain Block Selection. For the signing leaf $\text{id}_{\mathbf{x}_{\text{sig}}}$, we identify the corresponding WOTS⁺ chain block bundle. In particular, for $\delta = \lfloor \text{id}_{\mathbf{x}_{\text{sig}}} / L \rfloor$, we select the associated bundle

$$\mathbf{c}_{\delta,0}, \dots, \mathbf{c}_{\delta,w-1},$$

which corresponds to the 16 chain states of a WOTS⁺ instance under $w = 16$. Within each $\mathbf{c}_{\delta,s}$, the position of the signing leaf inside the batch is $k^* = \text{id}_{\mathbf{x}_{\text{sig}}} \bmod L$, where $L = \lfloor b / \text{len} \rfloor$ is the number of leaves per batch (Appendix J), and the block of chain i is directly accessed as $\mathbf{b}_{k^*,i}$, which occupies slot position $k^* \cdot \text{len} + i$ under the throughput-aware packing. This direct access applies to XMSS, where $\text{id}_{\mathbf{x}_{\text{sig}}}$ is public; the SPHINCS⁺ case is treated at the end of this appendix.

The signing task consists of, for each chain i , selecting the chain block at depth B_i , obtained from the base- w encoding (including the checksum) of the message digest.

The target chain block is obtained by evaluating a multilinear selection polynomial. Let $(\beta_i^3, \beta_i^2, \beta_i^1, \beta_i^0)$ denote the binary representation of B_i , and let (s_3, s_2, s_1, s_0) denote

the binary expansion of the depth $s \in \mathbb{Z}_{16}$. Then, the selection can be evaluated as follows:

$$P_i(\beta_i^3, \beta_i^2, \beta_i^1, \beta_i^0) = \sum_{s=0}^{15} \mathbf{c}_{\delta,s} \prod_{r=0}^3 ((\beta_i^r)^{s_r} \cdot (1 - \beta_i^r)^{1-s_r}). \quad (3)$$

This can be divided into two steps: generate multivariate monomials and inner product with the associated bundle $\mathbf{c}_{\delta,s}$. To evaluate P_i homomorphically, we apply the Möbius transform to the chain blocks $\{\mathbf{ct}_{\delta,s}\}_{s=0}^{15}$ to precompute the coefficients $\{\mathbf{coeff}_s\}_{s=0}^{15}$, and then compute the inner product with the multivariate monomials $\{\mathbf{basis}_s\}$ constructed from the bit ciphertexts $\{\beta_i^r\}_{r=0}^3$.

Since each bundle consists of four ciphertexts, the selection (Equation (3)) is repeated for each of them. The four evaluations are independent, so they can be merged into a single ciphertext and carried out SIMD-wise, which saves latency.

SIMD Parallelism. A bundle still carries all L leaves of its batch, so we first rotate it by $k^* \text{len}$, which brings the len chains of the signing leaf to slots $0, \dots, \text{len} - 1$ and leaves the other leaves to be dropped by the masks below. Each of the lane lane ciphertexts of the bundle then occupies only $\text{len} \times \gamma \ll N_{\text{slot}}$ slots, so we aggregate them into a single ciphertext by masking, rotation, and addition, placing lane l in the 128-slot block at offset $128l$. The lane lanes of every chain thus sit side by side inside one 512-slot block,

$$(\mathbf{L}_0, \dots, \mathbf{L}_{\text{lane}-1}),$$

where each lane holds the len chains of the signing leaf,

$$\mathbf{L}_l = (\underbrace{\mathbf{b}_{k^*,0}^{(l)}, \dots, \mathbf{b}_{k^*,\text{len}-1}^{(l)}}_{\text{len}}, \underbrace{0, \dots, 0}_{128-\text{len}}),$$

where $\mathbf{b}_{k^*,i}^{(l)}$ is the l -th 64-bit lane of the chain block $\mathbf{b}_{k^*,i}$, in the sense of Equation (2). Bit z of that lane occupies slot $512z + 128l + i$.

Based on this packing, the encrypted chain depths B_i are first aligned with the slots they act on: each B_i is replicated to position i of all lane lane blocks, which requires several rotations and additions. We then extract the binary representation of $\{B_i\}$ via a homomorphic bit extraction procedure (**BitExtract**) [8], which returns the bit representation and so enables the evaluation of the chain selection function (Equation (3)).

Signing Leaf Selection. The direct access above needs the leaf position in the clear. XMSS has it: the leaf index is public state, so δ and k^* are known before signing, and so are the positions of the authentication-path nodes, which are taken from the encrypted Merkle nodes by a plaintext mask (Algorithm 4, Step 3). In SPHINCS⁺ the leaf index comes from H_{msg} and stays encrypted, so both selections become homomorphic lookups.

Selecting the leaf is the same multilinear evaluation as Equation (3), with the $2^{h'}$ candidate leaves of the subtree as the table and the encrypted leaf index in place of the digit: the Möbius transform of their bundles gives the coefficients, and the inner product with the monomials in the h' index bits gives C_{leaf} , the len chain blocks of the signing leaf. At $h' = 4$ this is Equation (3) with 16 leaves in place of 16 depths; it is the **LeafSelect** of Appendix M.

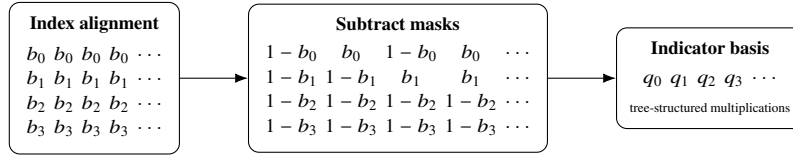


Fig. 7. Indicator-basis construction for authentication-path selection. The encrypted leaf-index bits are aligned with the candidate-node slots, converted into the corresponding β_r or $1 - \beta_r$ factors using plaintext masks, and combined through tree-structured multiplications.

L Authentication-Path Selection

During public-key generation, we retain the encrypted intermediate nodes of the Merkle tree. During signing, the encrypted leaf index determines one authentication-path node at each tree level. This is required for SPHINCS⁺ only: an XMSS signer knows its leaf index in the clear and reads the path off the retained nodes with a plaintext mask (Appendix K). As described in Section 4.2, we align the encrypted index bits with the candidate-node slots and construct their indicator basis using plaintext masks and tree-structured multiplications. Figure 7 illustrates this construction.

M Full algorithm for signing

Algorithm 4 shows the full signing algorithm for XMSS and SPHINCS⁺. It uses the following subroutines, all of which operate entirely on ciphertexts.

- $\text{WOTS}^+. \text{Sign}(\text{ct}_{M'}, C_{\text{leaf}}, \perp) \rightarrow \text{ct}_\sigma$: selects, for every chain of the signing leaf, the chain block at the depth given by the encrypted base- w digit of the message digest, so that the digits never have to be decrypted (Appendix K).
- $\text{SubtreeSetup}(\text{ct}_{\text{sk}_{\text{seed}}}, \text{ct}_{\text{id}_{\text{xtree}}}, \ell) \rightarrow (C^{(\ell)}, \mathcal{T}^{(\ell)})$: builds the hypertree subtree of layer ℓ addressed by the encrypted index, returning its encrypted WOTS⁺ chain blocks and Merkle nodes (Appendix J).
- $\text{LeafSelect}(C^{(\ell)}, \text{ct}_{\text{id}_{\text{xleaf}}}) \rightarrow C_{\text{leaf}}$: extracts the chain blocks of the signing leaf from the subtree, using the encrypted leaf index as a one-hot selector (Appendix K).
- $\text{AuthPathSelect}(\mathcal{T}^{(\ell)}, \text{ct}_{\text{id}_{\text{xleaf}}}) \rightarrow \text{ct}_{\text{AP}^{(\ell)}}$: selects, at each tree level, the sibling node on the path of the encrypted leaf index (Appendix L).
- $\text{Root}(\mathcal{T}^{(\ell)}) \rightarrow \text{ct}_{M^{(\ell+1)}}$ and $\text{NextIndex}(\text{ct}_{\text{id}_{\text{xtree}}}) \rightarrow (\text{ct}_{\text{id}_{\text{xtree}}}, \text{ct}_{\text{id}_{\text{xleaf}}})$: return the encrypted subtree root, which becomes the message of the next layer, and split the encrypted tree index into the index pair of that layer.
- $\text{ThDec}(\text{ct}_1, \dots, \text{ct}_m) \rightarrow (\mu_1, \dots, \mu_m)$: the one-round threshold decryption of Section 4.3; every signer broadcasts one partial decryption per ciphertext and the plaintext signature components are recovered locally.

Algorithm 4 Homomorphic signing: XMSS (top) and SPHINCS⁺ (bottom).**XMSS**

Input: Message M (or up to L messages for batching); leaf index idx_{sig} (or indices); encrypted sk_R ; chain-block bundles $\mathcal{C} = \{\mathbf{ct}_{b,s}\}$ (Section 4.1); encrypted Merkle nodes $\mathcal{T} = \{\mathbf{ct}_{\text{node}_{\ell,k}}\}$; public key $\text{pk} = (\text{pk}_{\text{root}}, \text{pp}_{\text{seed}})$

Output: XMSS signature σ

Step 1: Message digest generation

- 1: $\mathbf{ct}_R \leftarrow \text{PRF}_R(\mathbf{ct}_{\text{prf}}, \text{idx}_{\text{sig}})$
- 2: $\mathbf{ct}_{M'} \leftarrow \text{H}_{\text{msg}}(\mathbf{ct}_R \parallel \text{pk} \parallel \text{idx}_{\text{sig}}, M)$

Step 2: WOTS⁺ signing

- 3: $\mathbf{ct}_{\sigma} \leftarrow \text{WOTS}^+.\text{Sign}(\mathbf{ct}_{M'}, \{\mathbf{ct}_{b^*,s}\}_s, \perp)$ for $b^* = \lfloor \text{idx}_{\text{sig}} / L \rfloor$

Step 3: AP masking

- 4: $\{\mathbf{ct}_{\text{AP}_i}\}_{i=0}^{h-1} \leftarrow \text{mask encrypted auth path for } \text{idx}_{\text{sig}} \text{ from } \{\mathbf{ct}_{\text{node}_{\ell,k}}\}$

Step 4: Signature extraction

- 5: $(R, \{\sigma_i\}, \{\text{AP}_i\}) \leftarrow \text{ThDec}(\mathbf{ct}_R, \mathbf{ct}_{\sigma}, \{\mathbf{ct}_{\text{AP}_i}\})$
- 6: **return** $\sigma = (\text{idx}_{\text{sig}}, R, (\sigma_0, \dots, \sigma_{\text{len}-1}), \text{AP})$

SPHINCS⁺

Input: Message M ; encrypted $\mathbf{ct}_{\text{sk}_R}$, $\mathbf{ct}_{\text{sk}_{\text{seed}}}$; public key $(\text{pp}_{\text{seed}}, \text{pk})$

Output: SPHINCS⁺ signature σ

Step 1: Message digest generation

- 1: $\mathbf{ct}_R \leftarrow \text{PRF}_R(\mathbf{ct}_{\text{sk}_R}, \text{optrand}, M)$
- 2: $(\mathbf{ct}_{\text{md}}, \mathbf{ct}_{\text{idx}_{\text{tree}}}, \mathbf{ct}_{\text{idx}_{\text{leaf}}}) \leftarrow \text{H}_{\text{msg}}(\mathbf{ct}_R, \text{pk}, M)$

Step 2: FORS construction and signing

- 3: $\mathcal{F}, \mathbf{ct}_{\text{root}^{\text{FORS}}} \leftarrow \text{FORS.Setup}(\mathbf{ct}_{\text{sk}_{\text{seed}}}, \mathbf{ct}_{\text{md}}, \mathbf{ct}_{\text{idx}_{\text{tree}}}, \mathbf{ct}_{\text{idx}_{\text{leaf}}})$
- 4: $\mathbf{ct}_{\sigma^{\text{FORS}}} \leftarrow \text{FORS.Sign}(\mathcal{F}, \mathbf{ct}_{\text{md}}, \perp)$

Step 3: Hypertree construction and signing

- 5: $\mathbf{ct}_{M^{(0)}} \leftarrow \mathbf{ct}_{\text{root}^{\text{FORS}}}$
- 6: **for** $\ell = 0, \dots, d-1$ **do**
- 7: $\mathcal{C}^{(\ell)}, \mathcal{T}^{(\ell)} \leftarrow \text{SubtreeSetup}(\mathbf{ct}_{\text{sk}_{\text{seed}}}, \mathbf{ct}_{\text{idx}_{\text{tree}}}, \ell)$
- 8: $C_{\text{leaf}} \leftarrow \text{LeafSelect}(\mathcal{C}^{(\ell)}, \mathbf{ct}_{\text{idx}_{\text{leaf}}})$
- 9: $\mathbf{ct}_{\sigma^{(\ell)}} \leftarrow \text{WOTS}^+.\text{Sign}(\mathbf{ct}_{M^{(\ell)}}, C_{\text{leaf}}, \perp)$
- 10: $\mathbf{ct}_{\text{AP}^{(\ell)}} \leftarrow \text{AuthPathSelect}(\mathcal{T}^{(\ell)}, \mathbf{ct}_{\text{idx}_{\text{leaf}}})$
- 11: $\mathbf{ct}_{M^{(\ell+1)}} \leftarrow \text{Root}(\mathcal{T}^{(\ell)})$
- 12: $(\mathbf{ct}_{\text{idx}_{\text{tree}}}, \mathbf{ct}_{\text{idx}_{\text{leaf}}}) \leftarrow \text{NextIndex}(\mathbf{ct}_{\text{idx}_{\text{tree}}})$
- 13: **end for**

Step 4: Signature extraction

- 14: $\sigma \leftarrow \text{ThDec}(\mathbf{ct}_R, \mathbf{ct}_{\sigma^{\text{FORS}}}, \{\mathbf{ct}_{\sigma^{(\ell)}}, \mathbf{ct}_{\text{AP}^{(\ell)}}\}_{\ell=0}^{d-1})$
- 15: **return** σ

N Permutation Depth and Width

Tables 16 to 19 give, for each phase, the depth of the dependency chain of Keccak permutations and the width W of every level, that is, the number of permutations that may be evaluated in parallel at that level. The last two columns apply Equation (1): the batch size $\min(b, \lceil W/g \rceil)$ with $b = 512$, and the number of batches, with $g = 8$ processors except for XMSS signing, which uses one. The implementation keeps the len chains of a leaf in one batch (at most $\lfloor 512/len \rfloor = 7$ leaves per batch), assigns whole trees to GPUs rather than repartitioning each level, and does not merge the permutations of different hash kinds into one batch, so some of its batches are smaller than listed.

Table 16. XMSS setup, $h = 10$: depth 33, total 1,166,335 permutations.

Level	Content	Width	Batch size	#Batches
1–16	WOTS ⁺ chain ($\text{PRF}_{\text{seed}} 1 + F 15$)	68,608	512	134
17	L-tree 1	33,792	512	66
18	L-tree 2	17,408	512	34
19	L-tree 3	8,192	512	16
20	L-tree 4	4,096	512	8
21	L-tree 5	2,048	256	8
22	L-tree 6	1,024	128	8
23	L-tree 7	1,024	128	8
24–33	Merkle 1–10	512, 256, ..., 2, 1	64, 32, ..., 1, 1	8, ..., 8, 4, 2, 1

Table 17. XMSS signing: depth 3, total 3 permutations.

Level	Content	Width	Batch size	#Batches
1	$R = \text{PRF}_R(\text{sk}_R, \text{id}_{\text{sig}})$	1	1	1
2–3	H_{msg} (2 blocks, serial)	1	1	1

Table 18. SPHINCS⁺ setup: depth 37, total 17,439 permutations.

Level	Content	Width	Batch size	#Batches
1–16	WOTS ⁺ chain ($\text{PRF}_{\text{seed}} 1 + F 15$)	1,072	134	8
17–33	T_{len} (17 blocks, serial)	16	2	8
34–37	Merkle 1–4	8, 4, 2, 1	1, 1, 1, 1	8, 4, 2, 1

Table 19. SPHINCS⁺ signing: depth 39, total 350,199 permutations. Levels 3–22 run the hypertree and FORS concurrently; Width is their combined width, and batching is over the combined pool.

Level	Hypertree (17 layers)	FORS (35 trees)	Width (Hypertree + FORS)	Batch size	#Batches
1	R	—		1	1
2	H_{msg}	—		1	1
3	chain PRF_{seed}	sk PRF_{seed}	18,224 + 17,920	512	71
4	F 1	leaf F	18,224 + 17,920	512	71
5	F 2	Merkle 1	18,224 + 8,960	512	54
6	F 3	Merkle 2	18,224 + 4,480	512	45
7	F 4	Merkle 3	18,224 + 2,240	512	40
8	F 5	Merkle 4	18,224 + 1,120	512	38
9	F 6	Merkle 5	18,224 + 560	512	37
10	F 7	Merkle 6	18,224 + 280	512	37
11	F 8	Merkle 7	18,224 + 140	512	36
12	F 9	Merkle 8	18,224 + 70	512	36
13	F 10	Merkle 9	18,224 + 35	512	36
14–18	F 11–15	T_k blocks 1–5	18,224 + 1	512	36
19–22	T_{len} blocks 1–4	T_k blocks 6–9	272 + 1	35	8
23–35	T_{len} blocks 5–17	—	272	34	8
36–39	Merkle 1–4	—	136, 68, 34, 17	17, 9, 5, 3	8, 8, 7, 6

Table 20. Latency breakdown of XMSS (XMSS-SHAKE256_10_256) and SPHINCS⁺ (SLH-DNA-SHAKE-256f). For XMSS, setup uses $h = 10$ on 8 GPUs, and signing uses 1 GPU. For SPHINCS⁺, setup and signing use 8 GPUs. Per-stage times are those measured on rank 0, whose total is the end-to-end latency; Table 21 gives the per-GPU breakdown. Etc. denotes extra costs such as GPU I/O.

Phase	XMSS		SPHINCS ⁺	
	Step	Latency (s)	Step	Latency (s)
Setup	SKGen _{sig}	0.07	SKGen _{sig}	0.07
	WOTS ⁺ chains	574.8	WOTS ⁺ chains	19.7
	L-tree	248.5	T _{len}	5.0
	Merkle tree	6.4	Merkle tree	1.0
	Thrifty cleaning	0.02	Thrifty cleaning	0.02
	Partial decryption	0.01	Partial decryption	0.02
	Etc.	16.1	Etc.	1.0
	Total	845.9	Total	26.8
Signing			$R, M', \mathbf{ct}_{\text{idx}}$	7.9
			Hypertree construction	204.4
	R, M'	0.61	FORS signing	2.6
	WOTS ⁺ .Sign	0.06	WOTS ⁺ .Sign	1.9
	Thrifty cleaning	0.01	AP	0.8
	Partial decryption	0.01	Thrifty cleaning	0.3
	Etc.	0.00	Partial decryption	0.08
			Etc.	20.5
	Total	0.69	Total	238.5

O Threshold Signature Latency

Table 20 summarizes the step-by-step evaluation time in setup and signing for XMSS and SPHINCS⁺.

Using 8 GPUs in SPHINCS⁺ signing, we distribute equal amount of workload to each core. Each GPU core evaluates different portion of each signing steps. Table 21 splits the SPHINCS⁺ signing time of Table 20 by GPU rank. Rank 0 alone performs the threshold decryption.

P Simulation Security of the Signing Protocol

We instantiate the simulation security assumed in Appendix D for our protocol, following the proof approach of Hwang et al. [37]. We write everything for XMSS and for the implemented n -of- n case for simplicity, so $\mathcal{A} = [n] \setminus \{h\}$, where $\{h\}$ denotes an honest party. For approximate functionalities, Hwang et al. [37] need collaborative sampling

Table 21. Per-GPU latency (s) of SPHINCS⁺ signing on 8 GPUs (ranks r0–r7).

Step (s)	r0	r1	r2	r3	r4	r5	r6	r7
R, M' generation + Index extraction	7.9	3.3	3.3	3.3	3.2	3.8	3.4	3.4
Hypertree construction	204.4	173.3	166.7	167.6	167.2	164.1	164.8	216.9
FORS signing (construction + AP)	2.6	47.7	54.2	53.4	53.8	54.0	53.8	0.0
WOTS ⁺ .Sign	1.9	1.3	1.3	1.3	1.3	1.3	1.3	1.7
Hypertree AP	0.8	2.2	2.6	2.7	2.7	2.7	2.7	0.8
Thrifty cleaning	0.3	-	-	-	-	-	-	-
Partial decryption	0.1	-	-	-	-	-	-	-
Etc (GPU I/O, MPI wait)	20.5	9.4	9.0	8.9	9.0	11.3	11.2	14.4
Total	238.5				237.2			

of the smudging error to reach standard MPC security, since a perturbed output alone does not determine the exact one. Our decrypted values are bits that rounding recovers exactly (Lemma 1), so the standard definition already hands the simulator the exact output. Figure 8 states the ideal functionality $\mathcal{F}_{\text{ThSign}}$ to be realized, together with the key-generation functionality $\mathcal{F}_{\text{ThKeyGen}}$ assumed as a hybrid; Figure 9 states the protocol Π_{ThSign} ; and Figure 10 states the simulator for the view of the corrupted parties, which Theorem 2 then shows to be indistinguishable from a real execution.

Definition 6 (Smudging). For $B > 0$, a distribution D over R is B -smudging if D and $e + D$ are statistically indistinguishable for every $e \in R$ with $\|e\|_\infty \leq B$.

Definition 7 (IND-CPA). ThFHE is IND-CPA secure if every PPT adversary $\mathcal{A}$ that chooses m_0, m_1 on seeing fpk , evaluation keys included, has negligible advantage $\left| 2 \Pr[b = b' \mid b \xleftarrow{\$} \{0, 1\}, b' \leftarrow \mathcal{A}(\text{fpk}, \text{ThFHE}.\text{Enc}(\text{fpk}, m_b))] - 1 \right|$. For CKKS this rests on RLWE [18, 48]; with the evaluation keys given out, it is the circular-security form usual for bootstrappable FHE.

$\mathcal{F}_{\text{ThKeyGen}}$
1 : $(\text{fsk}, \text{fpk}) \leftarrow \text{ThFHE.KeyGen}(\lambda)$
2 : $\text{fsk}_i \xleftarrow{\$} R_q$ for $i \in [n-1]$; $\text{fsk}_n \leftarrow \text{fsk} - \sum_{i \in [n-1]} \text{fsk}_i \bmod q$
3 : output $(\text{fsk}_i, \text{fpk})$ to P_i for $i \in [n]$

$\mathcal{F}_{\text{ThSign}}$
1 : Key generation: on ρ_i from every P_i
2 : $\text{sk} \leftarrow \bigoplus_i \rho_i$; output $\text{pk} \leftarrow \text{Sig.Setup}(\lambda; \text{sk})$
3 : Signing: on (M, idx) from every P_i
4 : output $\sigma \leftarrow \text{Sig.Sign}(\text{sk}, M, \text{idx})$

Fig. 8. The hybrid key-generation functionality (top) and the one realized by the protocol (bottom). Sig is XMSS or SPHINCS⁺ and idx is the public leaf index, absent for SPHINCS⁺. Both outputs are exact; the protocol recovers them from perturbed values by rounding (Lemma 1).

Π_{ThSign}
1 : All parties call $\mathcal{F}_{\text{ThKeyGen}}$; each P_i obtains $(\text{fsk}_i, \text{fpk})$
2 : Each P_i broadcasts $\text{ct}_i \leftarrow \text{ThFHE.Enc}(\text{fpk}, \rho_i; r_i)$; $\text{ct}_{\text{sk}} \leftarrow \sum_i \text{ct}_i$
3 : $(\text{ct}_{\text{pk}}, C, \mathcal{T}) \leftarrow \text{ThFHE.Eval}(\text{fpk}, \text{PKGen}_{\text{sig}}, \text{ct}_{\text{sk}})$; $\text{ct}_{\text{pk}} = (b', a')$
4 : Each P_i samples $e'_i \xleftarrow{\$} D$, broadcasts $v_i \leftarrow a' \text{fsk}_i + e'_i$; $\text{pk} \leftarrow \lfloor b' + \sum_i v_i \rfloor$
5 : On (M, idx) : $\text{ct}_{\sigma} = (b, a) \leftarrow \text{ThFHE.Eval}(\text{fpk}, C_{M, \text{idx}}, \text{ct}_{\text{sk}}, C, \mathcal{T})$
6 : Each P_i samples $e_i \xleftarrow{\$} D$, broadcasts $u_i \leftarrow a \text{fsk}_i + e_i$; $\sigma \leftarrow \lfloor b + \sum_i u_i \rfloor$

Fig. 9. The protocol; all arithmetic is modulo q . Steps 1–4 are the key-generation phase of Section 4.1 and Steps 5–6 the signing of Algorithm 4. The v_i and u_i are the partial decryptions pd_i of Appendix D, written out. In SPHINCS⁺, $\text{PKGen}_{\text{sig}}$ returns ct_{pk} alone and C_M rebuilds the chain blocks and tree nodes it needs at each signing.

Lemma 1 (Rounding). *Let Δ be the scale at which the released bits are encoded, let every smudging error satisfy $\|e_i\|_{\infty}, \|e'_i\|_{\infty} \leq B_{\text{fl}}$, and let the residual errors of ct_{pk} and ct_{σ} be bounded by B_e . If $nB_{\text{fl}} + B_e < \Delta/2$, then Steps 4 and 6 of Figure 9 output exactly pk and σ .*

Proof. By linearity of decryption, $b + \sum_i u_i = \sigma + e_{\text{out}} + \sum_i e_i$ with $\|e_{\text{out}}\|_\infty \leq B_e$, so the perturbation is bounded by $B_e + nB_{\text{fl}} < \Delta/2$ and rounding to the nearest encoded bit returns σ ; likewise for pk . $\square$

The view of $\mathcal{A}$ is

$$\text{view}_{\mathcal{A}}^\Pi = ((\rho_i, r_i, e'_i, e_i)_{i \in \mathcal{A}}, (\text{fsk}_i)_{i \in \mathcal{A}}, \text{fpk}, \text{ct}_h, v_h, u_h),$$

its own inputs and randomness, what $\mathcal{F}_{\text{ThKeyGen}}$ gave it, and the three messages P_h broadcasts. Every ciphertext of the protocol — $\text{ct}_{\text{sk}}, \text{ct}_{\text{pk}}, \text{ct}_\sigma$, the retained $\mathcal{C}, \mathcal{T}$, and all intermediates — is a deterministic function of these, and so are the perturbed sums $\widehat{\text{pk}} = b' + \sum_i v_i$ and $\widehat{\sigma} = b + \sum_i u_i$; the simulator has only ct_h, v_h, u_h to produce.

Theorem 2. *Let ThFHE be IND-CPA secure (Definition 7), let D be B_e -smudging (Definition 6), and let the conditions of Lemma 1 hold. Then, in the $\mathcal{F}_{\text{ThKeyGen}}$ -hybrid model, Π_{ThSign} securely computes $\mathcal{F}_{\text{ThSign}}$ in the standard sense of [37, Definition 3] against a semi-honest adversary corrupting $\mathcal{A} = [n] \setminus \{h\}$: with Sim as in Figure 10,*

$$\{\text{Sim}(\mathcal{A}, (\rho_i)_{i \in \mathcal{A}}, \text{pk}, \sigma), (\text{pk}, \sigma)\} \approx_c \{\text{view}_{\mathcal{A}}^\Pi, \text{output}^\Pi\},$$

the two sides differing statistically by at most $2NB_e/B_{\text{fl}}$ and computationally by the IND-CPA advantage.

$\text{Sim}_{\text{ThSign}}^{\mathcal{A}}((\rho_i)_{i \in \mathcal{A}}, \text{pk}, \sigma)$	
1 :	Emulating $\mathcal{F}_{\text{ThKeyGen}}$, generate fpk and $(\text{fsk}_i)_{i \in \mathcal{A}}$
2 :	$\text{ct}_i \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, \rho_i; r_i)$ for $i \in \mathcal{A}$; $\text{ct}_h \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, 0)$
3 :	$\text{ct}_{\text{sk}} \leftarrow \sum_i \text{ct}_i$
4 :	$(b', a', \mathcal{C}, \mathcal{T}) \leftarrow \text{ThFHE}.\text{Eval}(\text{fpk}, \text{PKGen}_{\text{sig}}, \text{ct}_{\text{sk}})$
5 :	$(b, a) \leftarrow \text{ThFHE}.\text{Eval}(\text{fpk}, C_{M, \text{idx}}, \text{ct}_{\text{sk}}, \mathcal{C}, \mathcal{T})$
6 :	$e'_i \xleftarrow{\$} D$ for $i \in [n]$; $v_i \leftarrow a' \text{fsk}_i + e'_i$ for $i \in \mathcal{A}$
7 :	$v_h \leftarrow \text{pk} + \sum_i e'_i - b' - \sum_{i \in \mathcal{A}} v_i$
8 :	$e_i \xleftarrow{\$} D$ for $i \in [n]$; $u_i \leftarrow a \text{fsk}_i + e_i$ for $i \in \mathcal{A}$
9 :	$u_h \leftarrow \sigma + \sum_i e_i - b - \sum_{i \in \mathcal{A}} u_i$
10 :	return $((\rho_i, r_i, e'_i, e_i, \text{fsk}_i)_{i \in \mathcal{A}}, \text{fpk}, \text{ct}_h, v_h, u_h)$

Fig. 10. Simulator for $\text{view}_{\mathcal{A}}^\Pi$. It receives the exact outputs, samples every smudging error itself, and uses neither fsk_h nor ρ_h .

Proof. Fix the inputs $\rho = (\rho_i)_{i \in [n]}$, and with them sk, pk and σ ; σ and pk stand for their encodings at scale Δ . Writing $\text{aux} := ((\rho_i, r_i, e'_i, e_i)_{i \in \mathcal{A}}, (\text{fsk}_i)_{i \in \mathcal{A}}, \text{fpk})$ for the part

of the view that no step touches,

$$\begin{aligned}
& \{(\text{view}_{\mathcal{A}}^{\Pi}, \text{output}^{\Pi})\} \\
& \equiv \{(\text{aux}, \text{ct}_h, v_h, u_h), (\text{pk}, \sigma) \mid \text{ct}_h \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, \rho_h; r_h), \\
& \quad v_h := a' \text{fsk}_h + e'_h, u_h := a \text{fsk}_h + e_h\} \\
& \equiv \{(\text{aux}, \text{ct}_h, v_h, u_h), (\text{pk}, \sigma) \mid \text{ct}_h \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, \rho_h; r_h), \\
& \quad v_h := \text{pk} + e'_{\text{out}} + \sum_i e'_i - b' - \sum_{i \in \mathcal{A}} v_i, \\
& \quad u_h := \sigma + e_{\text{out}} + \sum_i e_i - b - \sum_{i \in \mathcal{A}} u_i\} \\
& \approx_s \{(\text{aux}, \text{ct}_h, v_h, u_h), (\text{pk}, \sigma) \mid \text{ct}_h \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, \rho_h; r_h), \quad (\text{Definition 6}) \\
& \quad v_h := \text{pk} + \sum_i e'_i - b' - \sum_{i \in \mathcal{A}} v_i, \\
& \quad u_h := \sigma + \sum_i e_i - b - \sum_{i \in \mathcal{A}} u_i\} \\
& \approx_c \{(\text{aux}, \text{ct}_h, v_h, u_h), (\text{pk}, \sigma) \mid \text{ct}_h \leftarrow \text{ThFHE}.\text{Enc}(\text{fpk}, 0; r_h), \quad (\text{Definition 7}) \\
& \quad v_h := \text{pk} + \sum_i e'_i - b' - \sum_{i \in \mathcal{A}} v_i, \\
& \quad u_h := \sigma + \sum_i e_i - b - \sum_{i \in \mathcal{A}} u_i\} \\
& \equiv \{(\text{Sim}_{\text{ThSign}}^{\mathcal{A}}((\rho_i)_{i \in \mathcal{A}}, \text{pk}, \sigma), (\text{pk}, \sigma))\}.
\end{aligned}$$

□

Corollary 1. *Under the assumptions of Theorem 2, the following hold:*

- (i) **Key privacy.** *Anything $\mathcal{A}$ computes from an execution it computes from $((\rho_i)_{i \in \mathcal{A}}, \text{pk}, \sigma)$ alone, since Sim is a function of those and reads neither fsk_h nor ρ_h . The execution therefore leaks nothing about the honest share fsk_h , hence nothing about fsk .*
- (ii) **Security of the ciphertexts computed before the decrypted blocks.** *Only ct_{pk} and ct_{σ} are ever decrypted, and ct_{σ} is a homomorphic selection from $\mathcal{C}$ and $\mathcal{T}$ of exactly the blocks a signature reveals. The intermediate ciphertexts from which the decrypted ones were computed stay encrypted. In the simulated view all of these are computed from $\bigoplus_{i \in \mathcal{A}} \rho_i$ alone, without sk , and so reveal nothing about it. Being a public function of ct_{sk} , they can be told from the real ones only by telling ct_h from an encryption of zero, which is the computational step in the proof of Theorem 2, where ct_h is the only input that changes.*
- (iii) **Unforgeability.** *By (i) and (ii) the view depends on sk only through (pk, σ) , so an $\mathcal{A}$ forging in Π_{ThSign} yields a forger against Sig whose advantage is smaller by at most the distinguishing advantage of Theorem 2, taken over the Q signing invocations by a hybrid.*

The security of our threshold signature thus rests on exactly two assumptions: that of the threshold FHE, through (i) and (ii), and unforgeability of the unmodified XMSS or SPHINCS⁺, through (iii). This is [12, Theorem 8.20] instantiated for our protocol.

Remark 2 (Extension to t -of- n). We discuss how the argument above carries over to a t -of- n access structure for two common ways of sharing fsk .

The first replaces the additive sharing of fsk by a linear secret sharing whose reconstruction coefficients are in $\{0, 1\}$ [12], as in the threshold signature of [57]. Every

reconstruction is then still a sum of shares, so the partial decryptions and their simulation are exactly those above, at the cost of share size.

The second, with one ring element per party, is Shamir sharing: let $\text{fsk}_i := f(\alpha_i)$ for a uniform f of degree below t with $f(0) = \text{fsk}$, $\text{fsk}_i := f(\alpha_i)$ for a uniform f of degree below t with $f(0) = \text{fsk}$, so that $\text{fsk} = \sum_{i \in S} \lambda_i^{S \rightarrow 0} \text{fsk}_i$ for every set S of t parties. Only Steps 4 and 6 of Figure 9 change; key generation is unchanged, and t parties suffice to sign since the signing circuit reads only the public ct_{sk} . We distinguish two settings, according to whether S is known when the partial decryptions are computed.

Synchronous: the decrypting set S is fixed, and known to its members, before the partial decryptions are computed. Each $P_i \in S$ broadcasts $u_i \leftarrow \lambda_i^{S \rightarrow 0} a \text{fsk}_i + e_i$, and $\sum_{i \in S} u_i = a \text{fsk} + \sum_{i \in S} e_i$ as before. Theorem 2 holds for $\mathcal{A} = S \setminus \{h\}$ with the same simulator and n replaced by t in Lemma 1 [37, Appendix D].

Asynchronous: S is unknown when the shares are computed, as in our one-round broadcast. The Lagrange coefficients are then applied at combination and amplify the errors. Kim et al. [45] clear the denominators with constants $C_{\text{dec}}, C_{\text{sim}}$, broadcasting $u_i \leftarrow C_{\text{dec}} a \text{fsk}_i + C_{\text{sim}} e_i$, and prove simulation security for any $t - 1$ corrupted parties. Theorem 2 holds with their simulator, with D required to be $B_3 B_e$ -smudging and the bounds B_1, B_2, B_3 of Appendix E.3 entering Lemma 1 and the statistical term.

Q Discussion on Malicious Security

Malicious security with abort in light of MPC. Although described in the semi-honest setting, our protocols admit a qualitative interpretation achieving malicious security with abort in light of MPC, the general idea of which we convey next. The insight, as identified by Garg et al. [29] in the case of collaborative zk-SNARKs [56], is that our threshold computation’s output (signature) is *self-verifying*, i.e. there exists a separate routine (Sig.Verify) to verify each protocol output, meaning correctness does not need to be enforced by the MPC itself.

As with the approach taken by Garg et al., we (informally speaking) may relax the notion of malicious security to allow the adversary to inject an additive bias δ into the final output, effectively forcing the output $\sigma + \delta$ as opposed to σ . We model this through a relaxed ideal functionality that permits the adversary to choose such additive bias; the linear nature of threshold FHE’s threshold decryption process allows this [12]. Notably, such relaxation still captures the notion of privacy of inputs given by honest parties (e.g. FHE secret keys of parties). Coupled with the self-verifying nature of signatures, this lifts the otherwise relaxed notion of malicious security to the original notion of malicious security [29].

Additionally, the adversary may submit malformed ciphertexts or withhold messages, causing a decryption failure event. We model this scenario as abort, i.e. malicious security with abort as a whole, given the self-verifying property.

We leave a formal treatment of full malicious security (without abort) that is also practical as future work. In theory, harnessing a proof (of correctness) system as a black-box should suffice; the hurdles include formally defining and showing malicious security for the offline subprotocols (DKG, joint signing key generation, and threshold-decrypting ct_{pk}) in addition to the online signing (threshold-decrypting each ct_σ) for an

end-to-end notion. More importantly, practical implementations are an active area of research [38, 49].