Aker: Efficient Two-Party SLH-DSA

Yaron Salman
Fordefi
yaron@fordefi.com

Ben Riva
Fordefi
Mysten Labs
ben@fordefi.com

Dmitry Kogan
Fordefi
dima@fordefi.com

***Abstract*—Threshold signing is widely deployed to protect cryptographic keys, but extending these deployments to post-quantum signatures remains challenging. While NIST's lattice-based signatures now have efficient threshold protocols, distributed signing for SLH-DSA (FIPS 205), NIST's stateless hash-based signature standard, has been considered prohibitively expensive because it lacks the algebraic structure those protocols typically exploit.**

We present Aker, a maliciously secure two-party signing protocol compatible with an unmodified FIPS 205 verifier. Our approach combines generic secret-sharing MPC with four optimizations that leverage the structure of the SLH-DSA hypertree. Evaluating active layers concurrently collapses sequential hashing depth to that of a single chain. Caching public keys allows authentication paths to be computed locally, so that only the signing leaf is evaluated in MPC. Fixing path prefixes lets signers reuse entire upper-layer signatures across messages. Finally, one-time subtrees sample lower-layer signature components locally rather than deriving them from the secret key, further reducing MPC work. Some of these gains come at the cost of stateful signers and a lower limit on the total number of signatures per key.

We implement and evaluate our protocols using SHAKE-based SLH-DSA in two deployment settings. For server–server deployments, we combine these techniques with a new GPU engine for maliciously secure Boolean GMW. Between two single-GPU servers in separate cloud providers over the public Internet, online signing for SLH-DSA-128f takes 2.09 **seconds, or** 0.76 **seconds with a lower signature limit, while input-independent Beaver triple generation runs offline and scales horizontally across keyless CPU workers. For client–server deployments, a periodic offline session prepares a reserve of over a hundred signatures, after which online signing requires only four low-bandwidth communication rounds and local hashing.**

These results show that maliciously secure distributed signing for SLH-DSA is within reach today in the two-party setting, while our optimizations extend directly to more parties.

1. Introduction

Rapid advances in quantum hardware and fault-tolerant error correction [1], [2] have accelerated the transition to post-quantum cryptography, as large-scale quantum computers can break the factoring and discrete-logarithm assumptions underlying RSA, ECDSA, and Schnorr [3]. To standardize quantum-resistant replacements, the U.S. National Institute of Standards and Technology (NIST) initiated its Post-Quantum Cryptography (PQC) standardization process in 2016 [4], which concluded by selecting three digital signature schemes that balance assumption conservatism against performance: the stateless hash-based standard SLH-DSA (FIPS 205, derived from SPHINCS$^+$) [5], [6], which relies solely on standard cryptographic hash functions, alongside two lattice-based schemes—ML-DSA (FIPS 204) [7] and FN-DSA (FALCON) [8]—that achieve smaller signatures and faster signing. Relying only on hash functions makes SLH-DSA the preferred choice for high-assurance, long-lived keys, such as PKI roots [9] and conservative blockchains [10], and some national cybersecurity agencies approve it standalone while requiring lattice-based schemes to be hybridized [11].

Adopting any new signature scheme requires supporting it in the mechanisms that protect signing keys. Hardware security modules (HSMs), the traditional choice for high-value keys, can support a new scheme by implementing its standard signing algorithm. Over the past decade, secure multiparty computation (MPC) and threshold cryptography have since become widely adopted [12] as an alternative that avoids specialized hardware by splitting a secret key across independent parties so that no single party ever holds it. In threshold signing, a quorum of shareholders jointly produces a standard signature, serving as a drop-in replacement for a single signer while leaving the verification algorithm untouched. Threshold signing is now foundational in institutional digital-asset custody and MPC wallets [13], [14], distributed blockchain validators [15], and distributed certificate authorities [16]. These deployments rely on mature protocols for ECDSA [17], [18], [19], Schnorr and EdDSA [20], and BLS [21]. Unlike HSMs, however, these deployments cannot simply implement the standard algorithm: each scheme first requires a dedicated threshold protocol.

NIST's Multi-Party Threshold Cryptography initiative [22] and industry roadmaps for post-quantum blockchain migration [23] both call for post-quantum threshold protocols, and for lattice-based signatures they now exist: recent work gives practical threshold protocols for ML-DSA [24], [25] and even FN-DSA [26].

For SLH-DSA, however, threshold signing has remained impractical because the FIPS-approved hash functions it relies on are expensive to evaluate in MPC. Unlike algebraic signatures, these hash functions offer no homomorphic structure to split across signers, and a single SLH-DSA signature requires up to millions of secret-dependent hash calls. Moreover, recent work shows that threshold signing for standard hash-based schemes cannot treat the hash function as a black box [27], [28], forcing signers to evaluate these calls directly inside MPC. Prior estimates found doing so prohibitively expensive: Cozzo and Smart [29] estimated that evaluating the secret-dependent SHA-3 permutations of a single SPHINCS$^+$ signature using state-of-the-art garbled circuits would take at least 85 minutes, and recent garbled-circuit benchmarks by Kondi, Kumar, and Vanegas [30] project more than five hours per SLH-DSA signature. A recent approach using threshold FHE [31] showed more practical results, bringing signing down to minutes of computation, though it still requires eight GPUs per signer, relies on lattice assumptions, achieving only semi-honest security.

In this paper, we present Aker, a maliciously secure distributed signing protocol for unmodified SLH-DSA that completes online signing in seconds, combining new structural optimizations of the SLH-DSA hypertree with secret-sharing Boolean MPC.

Our structural optimizations apply generally to distributed SLH-DSA signing with any number of parties, as we discuss later in the paper. We focus on the two-party setting with malicious security. This is a common setting in practice, since many deployments naturally involve two parties: a client and a service provider, as in password breach checking [32] and digital-asset wallets [13], [14], [33], [34], [35], [36], or two independent operators, as in private telemetry [37]. Two parties is also the most efficient configuration for malicious security, since communication and authentication costs grow with the number of parties.

In SLH-DSA, a randomized message digest selects a path through a multi-layer hypertree in which each layer uses hash chains to sign the root of the tree below it. Evaluating this structure directly in MPC is bottlenecked by both the sequential depth across layers and the sheer number of hash calls needed to build every tree along the path. We overcome both bottlenecks through four structural optimizations (Section 3):

- *Parallel tree evaluation*: Although each layer signs the root of the layer below it, the hash chains themselves depend only on the secret seed and their tree addresses—the child root only determines which chain positions to reveal. Signers can therefore evaluate all active layers concurrently in a single pass and select the required positions afterward, collapsing sequential hashing depth to that of a single chain.
- *Public-key caching*: Most of the hashing in each tree goes toward computing the public keys of non-signing leaves just to build the Merkle authentication path. Because these leaf public keys depend only on the secret seed, precomputing and caching them in the frequently visited upper layers lets signers assemble authentication paths locally, reducing online MPC in each cached layer from a full tree of leaves to the single signing leaf.
- *Fixed path prefixes*: Every hypertree leaf always signs the same child root regardless of the message, so its signature is identical whenever that path is visited. By computing the signatures along a fixed upper-path prefix once and locally searching for a message randomizer whose digest lands on this prefix, signers reuse those upper signatures across messages and eliminate online MPC for the prefixed layers.
- *One-time subtrees*: The verifier never checks that signatures in the bottom layers of the hypertree were derived from the secret key—only that they hash up to the root signed by the layer above. Instead of computing those layers from the secret key in MPC, the signers locally sample random values for their signatures, reconstruct the resulting root, and sign it from the layer above, trading total signing capacity for far less MPC work.

Our optimizations remain fully compatible with an unmodified FIPS 205 verifier. Parallel tree evaluation and public-key caching follow the standard signing algorithm directly, while fixed path prefixes and one-time subtrees trade lightweight signer state and a bounded signing budget for much greater efficiency—a natural fit for MPC, where signers already coordinate on every signature and only need to persist a small allocation record with their key shares.

To evaluate the resulting circuit, we use maliciously secure Boolean GMW [38], [39] with Beaver triples generated offline via post-quantum malicious silent OT [40], [41], [42], [43], departing from prior work that relied on garbled circuits [29], [30]. While garbled circuits execute in a constant number of rounds, conventional symmetric-key authenticated garbling such as Katz et al. [44] transmits 258 bits per AND gate in its circuit phase compared to 2 bits per party online in Boolean GMW—requiring over 50 GB of circuit communication per SLH-DSA-128f signature even with our structural optimizations, which is both slow to transfer and expensive under public-Internet cloud egress pricing, as we compare in Appendix D.

Boolean GMW instead trades bandwidth for communication rounds proportional to the circuit's AND depth. To keep this depth small, we target the SHAKE-based parameter sets of FIPS 205, whose Keccak permutation has only 24 sequential AND layers per hash call, far fewer than SHA-256. Combined with our structural optimizations, a full signing sweep requires only a few hundred communication rounds, which adds modest delay over modern inter-cloud links where round-trip latency is a few milliseconds.

Even with a shallow circuit, online signing still evaluates thousands of independent Keccak permutations in lockstep.

TABLE 1. SERVER–SERVER GPU ONLINE SIGNING, PER-SIGNATURE BEAVER PREPROCESSING, TOTAL PER-SIGNATURE COST, LIFETIME SIGNATURE BUDGET, AND PER-PREFIX CACHE PREPARATION.

	Prefix signing			Hybrid OTS				
Mode	Online	Offline	Cost	Online	Offline	Cost	Budget	Prefix prep.
128f	2.09 s	8.69 s	$0.86	0.76 s	0.53 s	$0.06	2^{38}	9.42 min
256f	7.84 s	28.01 s	$2.96	0.80 s	0.75 s	$0.07	2^{39}	16.5 min
128s	44.99 s	167.58 s	$17.93	0.65 s	0.21 s	$0.01	2^{35}	1.08 min
256s	96.46 s	698.74 s	$41.88	0.65 s	0.27 s	$0.02	2^{31}	1.98 min

To execute this workload efficiently, we build a custom multi-core CPU backend and the first GPU engine for maliciously secure Boolean GMW in C++ and CUDA (Section 5), engineered for wide parallel circuits where heavy computation must run in tandem with high-volume network communication at every round.

Experimental results. We evaluate two deployment settings. In the *server–server* setting, we evaluate *prefix signing*, which combines fixed prefixes with public-key caching, and *hybrid OTS*, which extends it with one-time subtrees. Between servers in separate cloud providers over the public Internet, prefix signing for SLH-DSA-128f already takes 3.39 s on 128-core CPUs, and drops to 2.09 s using a single GPU per server—or 0.76 s with hybrid OTS. Table 1 summarizes the measured online signing and offline preprocessing times and monetary costs across all four parameter sets. Per-signature Beaver triple generation is input-independent and scales horizontally across keyless CPU workers, while per-prefix cache preparation amortizes over millions of signatures, starting online signing from the top layers while caching the rest in the background.

In the *client–server* setting, where a network-constrained client such as a phone co-signs with a server, the parties combine fixed prefixes with one-time subtrees to move all MPC work offline. Online signing then requires only four low-bandwidth communication rounds and local hashing, completing in 0.24–0.53 s across our Wi-Fi and 5G profiles. Offline, a periodic preprocessing session of 66.6–80.2 min on four client cores prepares 128–256 signatures, with roughly 85% of this time spent on Beaver triple generation that can be delegated to keyless worker servers. Repeating this session across fresh prefixes supports billions of signatures over the key's lifetime, trading the 2^{64} budget of standard SLH-DSA for MPC-free online signing.

Beyond two parties. Our structural optimizations operate directly on the signing circuit, so they apply unchanged with any MPC protocol and any number of signers; one-time subtrees are especially attractive here, as they drastically reduce or eliminate online MPC work regardless of N. Because Boolean GMW natively supports N parties, our protocols extend directly to N-out-of-N signing against up to $N-1$ malicious parties with the same round complexity and per-party cost growing linearly in $N-1$, as we detail in Appendix C. Supporting t-out-of-N thresholds for $t < N$ is an interesting direction for future work.

Contributions. In summary, this paper contributes:

- Four structural optimizations for SLH-DSA (parallel tree evaluation, public-key caching, fixed path prefixes, and one-time subtrees) that collapse sequential hashing depth and heavily reduces secret-dependent hashing (Section 3).
- Maliciously secure two-party signing protocols for an unmodified FIPS 205 verifier that combine our optimizations with generic secret-sharing MPC, yielding a single-sweep protocol that massively reduces online work for server–server deployments and a client–server protocol whose online phase needs only four communication rounds and local hashing (Section 4).
- The first GPU engine for maliciously secure Boolean GMW, combining authentication-aware bitslicing, on-device MAC verification, and sharded communication overlap (Section 5.2).
- A comprehensive evaluation across a cross-cloud testbed over the public Internet, achieving server–server online signing in 2.09 s for SLH-DSA-128f, or 0.76 s with one-time subtrees, and 0.24–0.53 s client–server signing across the FIPS 205 parameter sets (Section 6).

1.1. Related Work

Recent work on threshold post-quantum signatures has largely focused on lattice based [24], [25], [26], [45] and other leading post-quantum candidates such as multivariate [46] schemes, where each construction poses distinct thresholdization challenges—such as performing rejection checks before revealing signature shares in ML-DSA [24], [25] and handling floating-point arithmetic during Gaussian sampling in FALCON [26]. Standard hash-based schemes, however, face a more fundamental barrier: Threshold signing cannot treat the hash function as a black box [27], [28] and naive circuit evaluation in MPC is prohibitively slow [29], [30], prompting recent works to circumvent this bottleneck by relaxing the scheme or its security model. For stateful and few-time hash-based schemes such as XMSS, LMS, and FORS, where the number of signing leaves is small, Kelsey, Lang, and Lucks [47] and Boneh et al. [48] avoid online MPC hash evaluations by having a trusted dealer materialize and mask the tree's intermediate hash values during key generation, which the signers later unmask jointly during signing. As both works note, however, materializing every leaf during key generation cannot scale to the 2^{60} leaves of the stateless SLH-DSA hypertree without pruning the hypertree down to a small subset of leaves [49]. Other proposals depart from standard FIPS 205 verification—either by replacing SHA-2/SHAKE with arithmetization-oriented hashes such as Poseidon2 [50], or by forgoing native threshold signatures in favor of custom verification layers or trusted HSMs [51]. Closest to our work, Cheon et al. [31] evaluate unmodified SLH-DSA in one round using threshold FHE, but reintroduces lattice assumptions, achieves only semi-honest security, and takes nearly four minutes of local computation across eight GPUs for SLH-DSA-256f.

GPU-accelerated MPC. Prior GPU-accelerated MPC frameworks target either garbled circuits [52] or machine-

learning workloads in the semi-honest or honest-majority settings, such as CrypTen [53] and Piranha [54], where the heavy lifting is local arithmetic computation. In contrast, evaluating hash-based signatures via maliciously secure Boolean GMW [39] constantly alternates between local bitwise computation and hundreds of sequential network rounds across billions of authenticated gates—a regime not supported by existing GPU frameworks.

2. Preliminaries

2.1. SLH-DSA

SLH-DSA, standardized in FIPS 205 [5], is a stateless hash-based signature scheme composed of WOTS$^+$, FORS, and an XMSS hypertree. We review these components and establish the terminology used throughout the paper.

WOTS$^+$. WOTS$^+$ is a one-time signature scheme that signs a message by revealing selected values from hash chains. The message determines which positions are revealed, and the chain endpoints determine the public key against which the signature is verified.

The signer encodes the message and its checksum in base w ($w = 16$ in SLH-DSA) as ℓ digits, with one hash chain per digit. Each digit specifies a *signature position* in its chain. To provide all w possible positions, each chain starts from a secret value at position 0 and applies an address-dependent hash function through $w - 1$ steps. The public key is obtained by compressing all chain endpoints.

The signature contains the chain values at the signature positions determined by the message and its checksum. The verifier completes each chain from its signature position to its endpoint to reconstruct the public key. Verification succeeds if this result equals the expected public key.

XMSS. To sign multiple messages under one public key, XMSS combines $2^{h'}$ one-time public keys as the leaves of a binary Merkle tree of height h'. We consider the XMSS construction used in SLH-DSA, whose leaves are WOTS$^+$ public keys. The XMSS public key is the root of this tree, computed by successively hashing pairs of child nodes starting from the leaf public keys.

The leaf used to sign a message is called the *signing leaf*. The XMSS signature contains the leaf's signature and an *authentication path*. The authentication path consists of the h' sibling nodes needed to reconstruct the root from the signing leaf's public key.

The verifier first reconstructs the signing leaf's public key from its signature. It then successively hashes this value with the authentication-path nodes to recover the root. Verification succeeds if this equals the XMSS public key.

FORS. FORS is a few-time signature scheme that supports a limited number of signatures under the same key. Each signature reveals selected secret values, authenticated under a common public key. A *FORS instance* organizes these values into k binary hash trees of height a, with each leaf containing the hash of a secret value. The public key is obtained by compressing the k tree roots.

The message digest selects one leaf in each tree. The signature contains the corresponding secret values and an authentication path for each selected leaf. The verifier hashes each revealed secret to recover its leaf, then uses the accompanying authentication path to reconstruct the tree root. Compressing the recovered roots yields the FORS public key. Verification succeeds if this equals the expected public key.

Hypertree. The hypertree links a FORS public key to the SLH-DSA public root through a chain of XMSS signatures. It consists of d layers of XMSS trees, each of height h', giving a total height $h = dh'$. Layers are numbered from 0 at the bottom to $d - 1$ at the top. Each bottom-layer leaf is associated with a FORS instance and signs its public key. Each leaf at a higher layer signs the root of a child tree in the layer below. The top layer contains a single tree, whose root is the SLH-DSA public root.

For each signature, a randomized message digest selects a FORS instance and its route to the top root. The selected tree at each layer is called the *active tree*. The sequence of active trees and their signing leaves forms the *signing path*.

The resulting signature includes the FORS signature and one XMSS signature from each active tree. Together, these components allow the verifier to reconstruct the top root and check that it equals the expected SLH-DSA public root.

Key material and addresses. To keep its key material compact, SLH-DSA generates the secret values of its component schemes from a seed. The public key consists of a public seed `PK.seed` and the top root `PK.root`. The secret key additionally contains `SK.seed` for secret derivation and `SK.prf` for message randomization.

FORS leaf secrets and WOTS$^+$ chain starting values are derived deterministically from `SK.seed` using PRF, with `PK.seed` and an address as public inputs. Each hash invocation takes a public address specifying its location in the hypertree (layer, tree index, and leaf index) along with the operation type and step within a chain or tree. FIPS 205 instantiates these hash functions and PRFs using either SHA-2 [55] or the SHA-3 extendable-output function SHAKE256 [56]; in this work, we focus on the SHAKE-based parameter sets.

Signing and verification. To select the components used for signing, the signer hashes the message and public keys together with a randomizer R derived from `SK.prf`. The digest determines the FORS instance, its selected leaves, and the signing path. Within each signing leaf, the value signed at that layer determines the chain signature positions. The signature includes R, allowing the verifier to recompute the same digest and recover these choices. Verification reconstructs the top root and checks it against `PK.root`. Figure 1 presents the signing algorithm.

2.2. Maliciously Secure Boolean MPC

System and threat model. We consider distributed signing among N parties $P_0, \ldots, P_{N-1}$ that hold additive secret shares of the signing key SK and jointly compute a standard SLH-DSA signature on an agreed message M. We work

Input: Message M, signing key SK, randomness opt_rand.
Output: Signature SIG.
$R \leftarrow \mathrm{PRF}_{\mathrm{msg}}(\mathtt{SK.prf}, opt_rand, M)$
$D \leftarrow \mathrm{H}_{\mathrm{msg}}(R, \mathtt{PK.seed}, \mathtt{PK.root}, M)$
$(md, t, j) \leftarrow \mathrm{Parse}(D)$; $(t_i, j_i)_{i=0}^{d-1} \leftarrow \mathrm{SigningPath}(t, j)$
$(\mathrm{SIG}_{\mathrm{FORS}}, P) \leftarrow \mathrm{FORS.Sign}(md, \mathrm{SK}, t_0, j_0)$
for $i = 0, \ldots, d-1$ **do**
  $(\sigma_i, P) \leftarrow \mathrm{XMSS.Sign}(P, \mathrm{SK}, i, t_i, j_i)$
end for
return $R \parallel \mathrm{SIG}_{\mathrm{FORS}} \parallel \sigma_0 \parallel \cdots \parallel \sigma_{d-1}$

Figure 1. SLH-DSA signing overview.

in the dishonest-majority setting against a static, malicious quantum adversary that corrupts up to $N-1$ parties. We focus on the two-party setting ($N = 2$) between P_0 and P_1 in the main body and describe the N-party extension in Appendix C. Our protocols achieve security with abort: the adversary may cause an honest party to abort, but cannot learn the signing key or forge a valid signature on any message the honest party did not agree to sign.

Authenticated Boolean secret sharing. To evaluate Boolean circuits with active security, we use Tinier, the maliciously secure Boolean MPC protocol of Frederiksen et al. [39], as implemented in MP-SPDZ [57].[1] Each party P_i holds a global MAC key $\Delta_i \in \mathbb{F}_{2^\kappa}$ for statistical security parameter κ. A secret bit $x \in \{0,1\}$ is shared as $[x]$, where each party holds an additive bit share $x_i \in \{0,1\}$ such that $x = x_0 \oplus x_1$ together with $\mathbb{F}_{2^\kappa}$ MAC shares authenticating x under the other party's key. To reveal a shared value, $\mathsf{OpenAuth}([x])$ verifies the linear MAC relations (batched across prior openings) and exchanges the bit shares to reconstruct x, aborting if verification fails.

Preprocessing and gate evaluation. In the input-independent offline phase, the parties generate random authenticated bits and authenticated Beaver multiplication triples $([a], [b], [c])$ satisfying $c = a \wedge b$ via malicious silent OT extension [40], [41], which expands a small set of initial base OTs under the LPN assumption.

In the online phase, XOR gates and NOT gates are linear and are evaluated locally without communication. Evaluating an AND gate $[z] = [x \wedge y]$ consumes one fresh Beaver triple and requires opening two masked bits, $d = x \oplus a$ and $e = y \oplus b$. Because independent AND gates at the same circuit layer open their masked inputs in a single message, two circuit metrics govern performance: the *total AND gate count* determines the preprocessing volume and online communication, while the *multiplicative depth* (the longest path of sequential AND gates) determines the number of online communication rounds.

SHAKE256 in Boolean MPC. SHAKE-based SLH-DSA instantiates all hash functions using SHAKE256, which relies on the 24-round Keccak-$f[1600]$ permutation [56]. Within each round, the linear steps $(\theta, \rho, \pi, \iota)$ consist solely of XOR gates and wire permutations and are evaluated locally, whereas the nonlinear χ step evaluates 1,600 parallel AND gates across the state. Evaluating a secret-shared Keccak-$f[1600]$ permutation therefore has multiplicative depth 24 and requires 38,400 AND gates. In SHAKE-based SLH-DSA, each secret-derivation call and chain hash step (F) invokes one Keccak-$f[1600]$ permutation [5], requiring 24 sequential AND-evaluation rounds per call.

1. We follow MP-SPDZ in naming the FKOS15 binary protocol *Tinier*.

3. Structural Optimizations

We next present four structural optimizations for SLH-DSA that form the basis of our MPC protocols, targeting both multiplicative depth, which sets the number of communication rounds, and total AND gate count, which sets computation and bandwidth. To isolate these algorithmic changes from the MPC layer, we describe them from the perspective of a single signer before instantiating them as two-party protocols in Section 4.

In SLH-DSA, secret-dependent work is confined to generating tree leaves, after which internal nodes are computed locally. The first two optimizations follow the standard signing algorithm, while the last two modify signing without changing verification.

3.1. Parallel Tree Evaluation

A naive implementation of SLH-DSA signing processes the signing path sequentially, so its multiplicative depth grows with the number of hypertree layers: The signer first generates the selected FORS instance's signature and public key. The signing leaf in the bottom active tree signs this public key, and each higher signing leaf signs the root of the active tree below it. In this schedule, the signer starts evaluating an active tree's hash chains only after obtaining the value its signing leaf must sign, waiting for all w steps of one layer to finish before starting the next and incurring $O(dw)$ sequential hash evaluations across the d layers.

Observation 1: independent public-key generation. Public-key generation can proceed independently across the active trees. The chain values depend only on the signing key and their addresses, not on the values signed at each layer. For example, the signer does not need the root of the bottom active tree to compute the leaf public keys of any of the active trees above it. Once the digest identifies the active trees, their leaf chains can therefore be evaluated in parallel, making the secret-dependent hashing depth independent of the number of hypertree layers.

Observation 2: public-key chains contain all signature values. Evaluating a signing leaf's chains to their endpoints computes every position along each chain. Retaining these intermediate values avoids a second evaluation: once the public roots are available, their encodings and checksums identify the signature positions to select.

Our approach: evaluate first, select later. Once the digest fixes the signing path, the signer generates the public keys in one concurrent sweep while retaining the intermediate chain values for the signing leaves (Figure 2), by deriving all active-tree and FORS leaf secrets in parallel and advancing their chains together through the $w-1$ hash steps; for each signing leaf the signer retains every chain value, whereas other leaves require only the endpoints. The leaf public

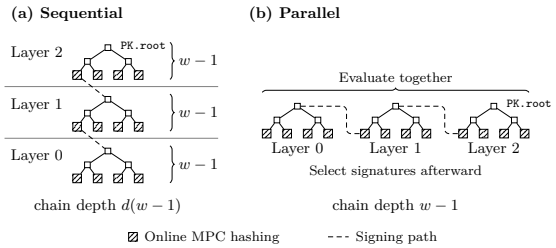


Figure 2. Sequential versus parallel tree evaluation across hypertree layers, reducing sequential chain depth from $d(w-1)$ to $w-1$.

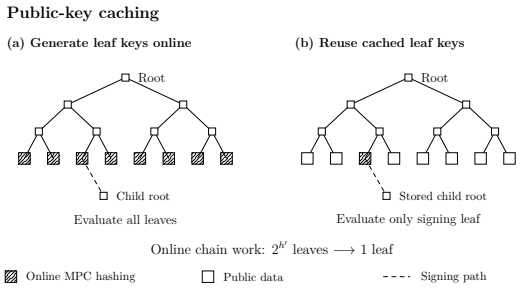


Figure 3. Public-key caching in an active XMSS tree reduces online chain evaluation from all $2^{h'}$ leaves to only the selected signing leaf.

keys are then obtained by compressing the chain endpoints, allowing local computation of the roots needed for signing, after which the signer constructs each leaf signature by selecting the required positions from the retained chain values without further chain evaluation.

Analysis. Recall that d is the number of hypertree layers. Parallel evaluation reduces the sequential chain steps from $d(w-1)$ to $w-1$, reducing the number of sequential hash rounds by a factor of d. Including the initial derivation, the secret-dependent hashing requires w sequential hash evaluations after the signing path is known, matching the minimum depth for evaluating a full chain directly from the master seed. The schedule targets servers with sufficient compute capacity to evaluate many chains concurrently.

3.2. Public-Key Caching

Generating an XMSS signature from scratch requires more than evaluating the signing leaf's chains. The signer also computes the public keys of the other leaves in the signing tree, to construct the authentication path. Most of the secret-dependent chain work therefore produces these public keys, which are regenerated for each signature.

Our approach: cache the leaf public keys. Observe that the leaf public keys in an XMSS tree are fixed by the signing key and their addresses. They can therefore be reused to construct authentication paths regardless of the message or signing leaf, storing only the n-byte compressed keys rather than full chains.

The signer precomputes and caches all leaf public keys in selected upper hypertree layers, where relatively few trees serve many signatures (Figure 3). Because these leaf public keys are non-secret, the signer stores them in the clear during preprocessing—along with the child-tree roots one layer below—and reuses them across future signatures.

During online signing, for each active tree in a cached layer, the signer constructs the authentication path and root locally from these stored public keys, and only the selected signing leaf requires online chain evaluation, stopping early at the signature positions fixed by the cached child-tree root.

Analysis. For an active tree in a cached layer, online chain evaluation falls from $2^{h'}$ leaves to one. Stopping at the signature positions saves further online hashing while preserving the parallel schedule of Section 3.1.

The cache requires preprocessing and storage, with the preprocessing cost amortized over subsequent signatures. Caching lower layers saves more online computation and communication, but each additional layer contains $2^{h'}$ times as many public keys as the layer above, increasing the required storage and the offline pre-processing.

3.3. Fixed Path Prefixes

Computing an SLH-DSA signature involves generating an XMSS signature in every active tree. The signer evaluates the leaf chains at each layer to produce these components. This secret-dependent work is repeated for every message, even when the upper part of the signing path is unchanged.

Our approach: reuse the upper signatures. The signer fixes the upper portion of the signing path, which we call a *prefix*, and computes its XMSS signatures once (Figure 4). The signer prepares a prefix by computing one XMSS signature at each upper hypertree layer it covers. It stores these signatures, including their authentication paths, as public data for reuse. Then, to sign a message, the signer deterministically searches for a randomizer R whose digest selects the prefix, allowing the signer to reuse stored signatures. Only the remaining lower part of the signature needs to be generated, and simply appended to the stored upper signatures. After sufficient usage, the signer retires the active prefix and switches to a fresh one.

Randomizer search. A prefix p covering the top L hypertree layers fixes $b = Lh'$ address bits. To sign a message M under p, the signer tests candidate n-byte randomizers R until finding one whose digest $D = \mathrm{H}_{\mathrm{msg}}(R, \mathtt{PK.seed}, \mathtt{PK.root}, M)$ selects the prefix p. Since each candidate matches p with probability 2^{-b}, the search requires 2^b trials in expectation. This search replaces $\mathrm{PRF}_{\mathrm{msg}}$, so $\mathtt{SK.prf}$ is not used, which does not affect signature verification.

Analysis. A prefix covering L of the d hypertree layers fixes $b = Lh'$ address bits and eliminates L/d of the hypertree chain work on each subsequent signature, at the cost of storing the L upper XMSS signatures, tracking prefix usage, and testing an expected $2^b = 2^{Lh'}$ candidate randomizers locally per message. Increasing L removes more upper layers, but grows this local search cost exponentially, trading online chain evaluation against local search overhead. For the remaining unprefixed layers, the signer can bound FORS

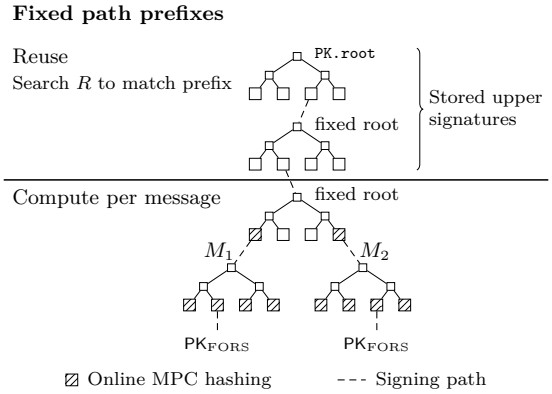


Figure 4. Fixed path prefixes reuse upper-layer XMSS signatures across messages by searching for a randomizer R whose matches the prefix.

usage beneath the active prefix either by tracking used FORS instances to prevent or cap reuse before rotating to a fresh prefix, or by bounding the total number of signing attempts per prefix; our analysis focuses on the simpler, conservative option of never reusing a FORS instance, and we detail these approaches in Appendix A.

Security. Fixed path prefixes modify SLH-DSA signing by replacing $\mathrm{PRF}_{\mathrm{msg}}$ with randomizer search and reusing the top L XMSS signatures across messages. Because each upper signing leaf always signs the same fixed child-tree root, reusing the top L signatures reveals no additional secret material, and never reusing a FORS instance ensures that each FORS key is exposed at most once.

3.4. One-Time Subtrees

In standard SLH-DSA, the signer derives and evaluates the active trees at every layer of the signing path from the secret seed, even though the lower layers exist only to expand each upper leaf from a one-time signature into a many-time subtree.

Observation. The verifier obtains the lower public keys and tree nodes from the signature and authenticates them through the upper-layer signatures, checking only that the final reconstructed root matches `PK.root`. An unused signing leaf at an intermediate layer can therefore authenticate any root reconstructed from locally chosen lower signature values and authentication nodes, without requiring those lower layers to be derived from the secret key.

Example. Consider a two-layer hypertree whose upper tree (with root `PK.root`) has two leaves, each authenticating a lower tree. To sign M_1, instead of deriving the lower tree from the secret key, the signer randomly samples a lower-layer signature, reconstructs the resulting root r_1 just as the verifier would, and signs r_1 with the first upper leaf's secret key. Since this upper signature links r_1 to `PK.root`, standard verification passes.

Our approach: sample the lower components. Rather than evaluating secret-dependent signing paths of full length d, the signer uses unique upper signing paths of length

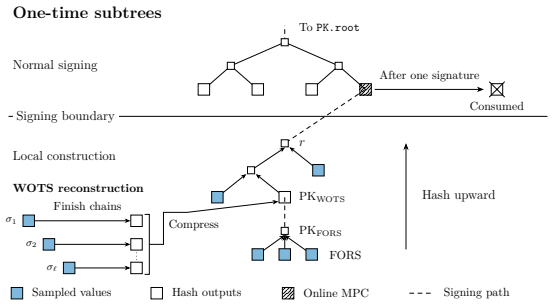


Figure 5. One-time subtrees locally sample lower signature components and reconstruct the root r upward to be signed once by an unused boundary leaf.

$d' < d$ and emulates the remaining $d - d'$ lower layers and FORS locally by sampling their signature components (Figure 5): the revealed FORS values, the WOTS$^+$ chain values at the selected signing leaves, and the authentication nodes. We call the lowest layer of the d'-layer upper path the *signing boundary*, and its selected signing leaf the *boundary leaf*. Working upward from the sampled values, the signer reconstructs the lower root and signs normally from the boundary upward to link it to `PK.root`.

Subtree sampling. Below the signing boundary, the signer directly samples the values that form the lower part of the signature from a fresh random n-byte seed ρ:

$$X_{\mathsf{ADRS}} = H_{\mathrm{gen}}(\rho, \texttt{PK.seed}, M, R, \mathsf{ADRS}), \qquad (1)$$

where H_{gen} is a domain-separated hash function returning n bytes, and ADRS is the component's FIPS 205 address. These values are sampled directly, without computing earlier chain steps or the subtrees beneath the authentication nodes, and determine the lower root signed at the boundary leaf.

One-time use. The sampled components cover only the selected lower signing path. Recovering the material needed to sign another message through any other leaf or path in the subtree beneath the boundary leaf would require finding hash preimages. Each new signature therefore constructs a fresh lower root, and because the boundary leaf's key is one-time, it can authenticate only one such root. The entire subtree beneath a boundary leaf thus becomes a *one-time subtree*: once used for a single signature, neither the boundary leaf nor the subtree below it can be used again.

Analysis. Sampling replaces secret-dependent key generation and signing across the bottom $d - d'$ layers and FORS with local hashing, while the upper d' layers from the boundary upward can be evaluated online or precomputed in advance. Because each boundary leaf and its subtree can be used only once, the number of available boundary leaves limits how many signatures can be produced. Increasing d' by one layer expands the pool of boundary leaves—and thus the signing quota—by a factor of $2^{h'}$, at the cost of evaluating an additional layer of trees. The signer must also track used boundary leaves, and finding an unused leaf requires more search effort as leaves are consumed.

Signature indistinguishability. Sampling one-time subtrees also preserves the distribution of produced signatures. Intuitively, because each subtree is used at most once, the lower values revealed in standard SLH-DSA are hash outputs on unrevealed inputs and thus indistinguishable from the pseudorandom values sampled via H_{gen} from a fresh secret seed ρ; we omit the proof of Lemma 1 for brevity:

Lemma 1 (One-Time Subtree Indistinguishability)**.** *Consider SLH-DSA signing with one-time subtrees where each boundary leaf is used at most once and each subtree is sampled via H_{gen} from a fresh secret seed ρ. The resulting signatures are computationally indistinguishable from standard SLH-DSA signatures along the same signing paths under the standard SLH-DSA assumptions and the pseudorandomness of H_{gen}.*

3.5. Deployment Modes

Having introduced our four structural optimizations individually, we now show how to combine them into complete signing configurations. Because these techniques target different parts of the hypertree, relying on any single optimization alone leads to poor trade-offs, requiring either an expensive randomizer search, a prohibitive cache across the full hypertree, or a small signing quota. For server–server deployments, we combine a fixed prefix with public-key caching—and optionally with one-time subtrees in a *hybrid OTS* mode that trades signing capacity for lower online latency—while for resource-constrained clients such as mobile devices, we combine fixed prefix with one-time subtrees to eliminate online chain evaluation altogether (Figure 6).

Prefix with public-key caching. Combining a fixed prefix with public-key caching (Figure 6(a)) avoids pushing either technique to an expensive extreme: the prefix supplies reusable XMSS signatures for the topmost layers, while restricting paths to that prefix means the cache only needs to be prepared and stored for the trees beneath it. In the cached layers directly below the prefix, the stored leaf public keys and child-tree roots provide the authentication paths and signing inputs locally, so online evaluation computes only each signing leaf's chains up to their signature positions. In the remaining lower XMSS layers and the selected FORS instance, the signer evaluates all leaf chains in full. All online chain work—both the cached signing leaves and full lower trees—runs in a single parallel sweep, suiting server–server deployments with high compute and bandwidth.

Hybrid OTS. To further reduce server–server online work, *hybrid OTS* extends the public-key caching configuration above by placing a signing boundary at or below the cached layers and locally sampling the lower XMSS layers and FORS components as a one-time subtree. When the boundary is placed at the bottom cached layer, online evaluation computes only the signing leaves in the cached layers, while placing it deeper evaluates additional lower trees online to exponentially increase the signing budget.

Prefix with one-time subtrees. Precomputing signing material at and above the signing boundary removes online chain evaluation altogether, and combining one-time subtrees with a fixed prefix (Figure 6(b)) restricts this precomputation to a manageable, replenishable reserve beneath the active prefix rather than the entire hypertree. The prefix supplies the public, reusable XMSS signatures for the top layers. In the retained layers between the prefix and the signing boundary, each leaf signs a fixed child-tree root, so the signer precomputes and stores only those protected signature values alongside the public leaf keys needed for authentication paths; at the boundary, where the lower root is determined online, the signer stores all protected WOTS$^+$ chain positions. Online signing locally samples the lower XMSS and FORS components, reconstructs the lower root, selects the corresponding boundary and retained signature values, and appends the prefix signature, making this mode well suited to mobile clients as well as low-latency servers.

Switching prefixes. In all configurations, the prepared cache or signing reserve is tied to the active prefix. When that prefix reaches its usage limit—either through the prefix search budget or by exhausting its available one-time boundary leaves—the signer retires it, activates a fresh prefix under the same public key, and prepares the corresponding cache or reserve. Appendix A details the prefix retirement limits and state handling.

Security. We establish the unforgeability of our structural modifications by analyzing a single signer that combines all four optimizations: evaluating active trees in parallel, reusing a fixed prefix and cached public keys, and sampling one-time subtrees below the signing boundary.

Theorem 1 (Unforgeability of Optimized SLH-DSA)**.** *Consider SLH-DSA signing with parallel tree evaluation, public-key caching, fixed path prefixes, and one-time subtrees where each boundary leaf is used at most once. The resulting signing scheme achieves existential unforgeability (EUF-CMA) under the standard SLH-DSA assumptions.*

Proof sketch. Parallel tree evaluation and public-key caching compute the unmodified FIPS 205 signing function, and reusing a fixed prefix reveals no additional secret material because each upper signing leaf always signs the same child-tree root.

For one-time subtrees, an adversary that forges a signature after interacting with the optimized signer can be used to break the one-wayness or collision resistance of the hash function. Because each subtree is used at most once and sampled via H_{gen} from a fresh secret seed ρ over the hash output distribution (which is trivial in the random oracle model, where outputs are uniform), the sampled lower components (FORS values, WOTS$^+$ chain values, and authentication siblings) are indistinguishable from fresh hash outputs. During signing, the reduction can therefore replace these sampled values with challenge hash outputs without changing the adversary's view.

Since each boundary leaf signs at most one lower root r_i, a forgery on a new message must either: (1) authenticate a new root at or above the signing boundary, breaking WOTS$^+$ or XMSS unforgeability in the upper layers; or (2) reconstruct a previously signed boundary root r_i from a different lower signature (unless it collides in H_{msg}). In the second

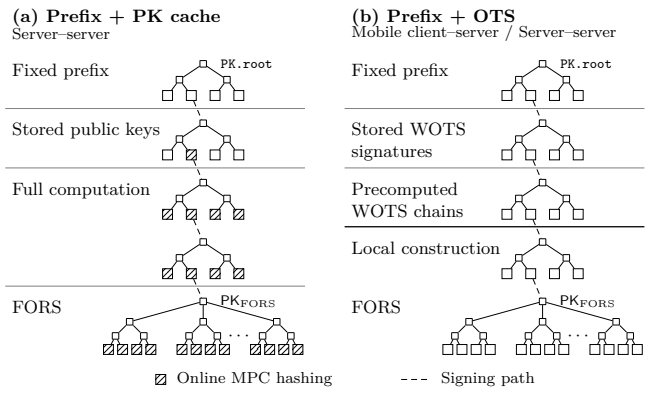


Figure 6. Hypertree structure and signature construction in our two primary deployment modes.

case, without a hash collision along the tree, reconstructing r_i from different lower values requires either reconstructing a sampled authentication sibling from its children or, by WOTS$^+$ checksum incomparability, starting a chain at an earlier step. In both cases, a preimage of an embedded challenge output appears in the forged signature. □

4. Two-Party Signing

We now instantiate the deployment modes of Section 3.5 as maliciously secure two-party signing protocols over authenticated Boolean shares.

4.1. Secure Evaluation and Key Generation

Every secret-dependent SHAKE256 invocation is evaluated directly as a Boolean circuit over authenticated shares by executing the AND gates of its Keccak-$f[1600]$ rounds. Following the parallel schedule of Section 3.1, the parties derive leaf secrets from $[\texttt{SK.seed}]$ and compute the chain endpoints and FORS leaves on authenticated shares. They open these endpoints via OpenAuth to compute the public keys, roots, and authentication paths locally. Once the values to be signed are known, each party selects the required signature shares locally from the retained chain values for a final authenticated opening.

Distributed key generation. Since both deployment modes replace $\mathrm{PRF}_{\mathrm{msg}}$ with randomizer search, the secret key consists only of the n-byte seed $\texttt{SK.seed}$. Each party samples a uniform share of $\texttt{SK.seed}$ and authenticates it to form $[\texttt{SK.seed}]$, while the parties open a jointly generated seed to fix $\texttt{PK.seed}$. They then run the parallel evaluation on the top-layer XMSS tree, open its chain endpoints via OpenAuth, and locally compute the leaf public keys and Merkle root $\texttt{PK.root}$. Once the key pair is instantiated, the parties use the same evaluation procedure to prepare the initial prefix and its public-key cache or OTS reserve.

Security. Parallel evaluation over authenticated secret shares preserves the standard signature distribution while keeping the secret seed and all unselected chain intermediates hidden. Opening the WOTS$^+$ endpoints and FORS leaves reveals only the public keys of those instances, which are already public in the standard SLH-DSA security reduction [5], [6]. Since the underlying authenticated Boolean MPC acts as a black box that exposes only these public values and the final signature components, the security of our two-party protocols reduces directly to the single-signer unforgeability of Theorem 1.

Input: Common: M, p, PK, prefix usage state.
Output: Randomizer R, or $\perp$.
Party P_0**:**
Find R s.t. $D = \mathrm{H}_{\mathrm{msg}}(R, \mathrm{PK}, M)$ selects p and passes the usage check
Update prefix usage state; $P_0 \longrightarrow P_1 : R$
Party P_1**:**
$D \leftarrow \mathrm{H}_{\mathrm{msg}}(R, \mathrm{PK}, M)$
if $\mathsf{Prefix}(D) \neq p$ or D fails usage check **then return** $\perp$
Update prefix usage state; **return** R

Figure 7. Prefix randomizer search and verification.

4.2. Fixed Prefix with Public-Key Caching

In the public-key caching mode, the parties precompute the public keys of the cached layers beneath the prefix p and use randomizer search to select a signing path matching p.

Public-key cache preprocessing. Using the secure evaluation of Section 4.1, the parties evaluate the configured trees beneath the active prefix, open their chain endpoints via OpenAuth, and store the resulting compressed leaf public keys and child-tree roots. During online signing, authentication paths and signing inputs in these cached layers are available locally, so secure evaluation computes only the signing leaves' chains up to their signature positions.

Randomizer search. During signing, one party searches locally for a randomizer R whose digest matches p and satisfies the prefix usage check of Appendix A, sends R to the other party, who verifies both conditions and updates the prefix usage state. The protocol in Figure 7 specifies the search and verification.

Online signing and round complexity. To sign a message M under the active prefix p, the parties first run the protocol in Figure 7 to obtain a valid randomizer R, whose digest D fixes the signing path beneath p. In a single parallel sweep (Section 4.1), the parties evaluate the signing leaves' chains in the cached layers up to their known signature positions, together with all leaf chains in the uncached lower layers and the selected FORS instance. They then open the uncached endpoints and FORS leaves via OpenAuth to reconstruct the lower roots locally, select the corresponding lower signature shares, and open all signature shares in a final batched OpenAuth before appending the stored prefix signatures. Because randomizer search replaces $\mathrm{PRF}_{\mathrm{msg}}$, the online phase avoids any MPC evaluation for $\texttt{SK.prf}$: its communication rounds consist only of the brief randomizer agreement in the protocol in Figure 7, the w sequential Keccak calls of the chain sweep, and two batched openings.

Security. This mode builds on public-key caching and fixed path prefixes. For public-key caching, the cached leaf public keys and child-tree roots are already public in SLH-DSA and are evaluated as in Section 4.1. For fixed path prefixes, the protocol in Figure 7 verifies that R matches p and passes the prefix usage check (with state handling and crash recovery detailed in Appendix A). Together with the authenticated Boolean MPC evaluation, this ensures that the two-party protocol preserves unforgeability.

4.3. Fixed Prefix with One-Time Subtree

In one-time subtree (OTS) mode, the parties precompute the reserve's signing material and derive the lower signature components locally from a joint seed opened online.

Precomputed signing material. Using the secure evaluation of Section 4.1, the parties prepare the reserve for the active prefix. They open only the chain endpoints to store the compressed leaf public keys. The signing material remains in authenticated secret-shared form: all chain values at the signing boundary, and the signature values for the fixed child-tree roots in the retained layers.

Online signing and round complexity. To sign a message M under the active prefix p using the protocol in Figure 8, the server first searches for a randomizer R whose digest matches p and selects an unused boundary leaf $j \notin U$; the client verifies both conditions and marks j as reserved in U. Next, the parties open a fresh precomputed shared seed $[\rho]$ via OpenAuth (yielding a post-quantum simulatable uniform seed in one round, discarded if the session aborts), locally sample the lower signature components X_{ADRS} via Equation (1), and reconstruct the lower root P at the signing boundary. Finally, they select the required shares from the boundary and retained layers and open them together in a single batched OpenAuth step before appending the stored prefix signatures. Since all secret-dependent hashing is precomputed, online signing requires no MPC gate evaluations and completes in four communication rounds: randomizer proposal, acceptance and leaf reservation, joint-seed opening, and batched signature opening.

Security. To ensure each one-time subtree is used at most once, the parties maintain the set U of reserved boundary leaves, marking each selected leaf before opening $[\rho]$; we discuss state handling and crash recovery in Appendix A. Because ρ is opened only between the signers, external verifiers never learn it, so signature indistinguishability in honest executions follows directly from Lemma 1. Meanwhile, even though a malicious co-signer learns ρ during signing, combining the leaf-reservation checks in the protocol in Figure 8 with the authenticated Boolean MPC evaluation ensures that the two-party protocol preserves unforgeability.

4.4. Hybrid OTS

Hybrid OTS combines the two protocols above for server–server deployments: the parties precompute only the public-key cache (Section 4.2), reserve an unused boundary leaf and open $[\rho]$ online to reconstruct the boundary root P locally (Figure 8), and evaluate the layers from the boundary up to the prefix in one parallel sweep.

Input: Common: M, p, PK, public tree data, reserved leaves U. Secret shared: stored shares $[C]$, fresh $[\rho]$.
Output: Signature Σ, or $\perp$.
Round 1 (Server P_0):
$R \leftarrow \mathsf{SearchUnused}(M, p, U)$
$j \leftarrow \mathsf{BoundaryLeaf}(\mathrm{H}_{\mathrm{msg}}(R, \mathrm{PK}, M))$; $U \leftarrow U \cup \{j\}$
$P_0 \longrightarrow P_1 : R$
Round 2 (Client P_1):
$D \leftarrow \mathrm{H}_{\mathrm{msg}}(R, \mathrm{PK}, M)$; $j \leftarrow \mathsf{BoundaryLeaf}(D)$
if $\mathsf{Prefix}(D) \neq p \vee j \in U$ **then return** $\perp$
$U \leftarrow U \cup \{j\}$; $P_1 \longrightarrow P_0 : \mathsf{accept}(R, j)$
Round 3 (Both parties):
$\rho \leftarrow \mathsf{OpenAuth}([\rho])$
Local computation (Both parties):
Sample Σ_{lower} via Equation (1) and reconstruct lower root P
$[W_{\mathrm{bnd}}] \leftarrow \mathsf{SelectBoundaryShares}([C], j, P)$
$[W_{\mathrm{ret}}] \leftarrow \mathsf{RetainedShares}([C], D)$
Round 4 (Both parties):
$(W_{\mathrm{bnd}}, W_{\mathrm{ret}}) \leftarrow \mathsf{OpenAuth}([W_{\mathrm{bnd}}], [W_{\mathrm{ret}}])$
Assemble Σ from R, Σ_{lower}, W_{bnd}, W_{ret}, Σ_{prefix}; **return** Σ

Figure 8. Online signing with one-time subtrees.

5. Implementation

Evaluating thousands of concurrent Keccak permutations in MPC requires an execution engine tailored to wide, parallel Boolean circuits. We describe our system architecture (Section 5.1) and the design of our multi-core CPU and GPU Boolean GMW engines (Section 5.2).

5.1. System Architecture

Core engines and runtime. We implement Aker in C++20 and CUDA as a modular stack that separates generic MPC primitives from SLH-DSA-specific logic. At the base, we implement the Tinier maliciously secure Boolean GMW protocol [39] from scratch in two reusable execution backends—one for multi-core CPUs and one for GPUs—that manage authenticated shares, Beaver triple consumption, and batched MAC verification, providing a general foundation on which other Boolean circuits can be readily implemented. Atop these backends, we implement a SHAKE256-specific evaluator for executing thousands of concurrent Keccak-$f[1600]$ permutations, driven by an SLH-DSA runtime that orchestrates the structural optimizations of Section 3 to produce signatures compatible with an unmodified FIPS 205 verifier.

Off-the-shelf components and OT preprocessing. For networking and local cryptography, we use Boost.Asio for TCP communication, OpenSSL (`libcrypto`) for hashing and key derivation, and a bitsliced Boyar–Peralta AES circuit [58] for expanding MAC-check challenges. For prefix randomizer grinding, we use Intel's `intel-ipsec-mb` multi-buffer AVX-512 library on CPUs and NVIDIA's `cuPQC` library on GPUs as our SHAKE256 primitives. For offline Beaver triple generation, we use `libOTe`'s malicious silent-OT binary-triple generator [43] on CPUs

for both server–server and client–server preprocessing. The construction builds on malicious silent OT and LPN-based correlation generators [40], [41]; we instantiate its base OTs with the Kyber-based Masny–Rindal protocol [42], making the preprocessing post-quantum secure.

5.2. MPC Engine Design and Optimizations

Flattening the SLH-DSA hypertree into a single parallel sweep makes signing a natural fit for multi-core CPUs and GPUs, as the workload evaluates thousands of identical, independent hash circuits in lockstep. At a high level, this execution alternates between wide batches of local linear operations and interactive nonlinear steps that interleave local computation with network rounds. Accelerating this pattern differs fundamentally from prior GPU-accelerated MPC frameworks such as CrypTen [53] and Piranha [54], which target machine-learning workloads dominated by compute-heavy local arithmetic. In maliciously secure Boolean GMW, while the billions of local bitwise operations still require careful optimization, the primary bottleneck is tightly interleaving them with hundreds of sequential communication rounds and MAC updates, so performance hinges on keeping parallel execution units, memory movement, and network I/O simultaneously saturated without stalling.

Both our CPU and GPU backends share two core design principles tailored to this workload, followed by hardware-specific optimizations for each architecture:

Authentication-aware bitslicing. Evaluating Boolean gates one bit at a time wastes the width of 64-bit hardware registers. We instead use *bitslicing*, a technique that packs corresponding state bits from 64 parallel hash evaluations into a single 64-bit word, so one bitwise instruction evaluates 64 identical gates at once across those evaluations. In maliciously secure GMW, however, each bit share is paired with a multi-bit MAC tag, and storing them together would break this packing. We therefore store the 64-bit packed shares separately from their MAC tags, preserving 64-way bitwise operations while updating both arrays in place during nonlinear steps.

Sharded communication overlap. Since every nonlinear layer requires a network round-trip and SLH-DSA exchanges large volumes of data, executing all instances as a single batch would alternate between computation and bursty network I/O: compute units would sit idle waiting on massive transfers, and the network would sit idle during computation. We instead partition the active instances into independent *shards*, each with its own state, buffers, and dedicated TCP connection. Whenever a shard reaches a communication step, it exchanges data immediately while other shards continue computing, streaming traffic steadily and keeping both compute units and the network saturated without per-round barriers.

Multi-core CPU engine. To keep MAC arithmetic from becoming a CPU bottleneck, we use *SIMD vectorization*, a technique that processes multiple 64-bit words in parallel within wide vector registers, alongside hardware carry-less multiplication instructions (`PCLMULQDQ` on x86 and `PMULL` on ARM) for $\mathbb{F}_{2^{40}}$ field multiplication. We also fuse the nonlinear step in place to avoid redundant memory copies. On large multi-socket servers, storing data in a single CPU socket's memory creates a bandwidth bottleneck when many cores access it at once; we therefore use *NUMA interleaving*, a technique that spreads memory allocations evenly across sockets to utilize their combined memory bandwidth. Finally, we balance shards dynamically across cores and offload checkpoint MAC reductions to background threads so verification runs alongside ongoing signing rounds.

GPU engine. On a GPU, one of the primary bottlenecks is the PCIe link between CPU and GPU memory, which is an order of magnitude slower than on-device memory. Following the device-resident design of Piranha [54], we allocate GPU memory once at startup and keep all secret states, public constants, and MAC records on the GPU throughout signing, streaming Beaver triples from host memory in batches as they are consumed. We then minimize data movement between the CPU and GPU: the CPU uploads only the small inputs that change between steps, and the GPU returns only the compact masked bits needed for network rounds, while MAC verification runs entirely on device and returns only the final check value. To overlap these PCIe transfers and network exchanges with GPU computation across shards, we use *CUDA streams*, a technique that executes independent queues of GPU kernels and memory copies concurrently on the device without global barriers. Finally, because GPUs execute threads in lockstep groups where branches and integer divisions stall execution, we precompute lookup tables on the CPU and unroll loops so the GPU kernels remain branchless and bitwise.

6. Evaluation

6.1. Experimental Setup

Methodology. We implement the full online signing protocol, verified against a standard FIPS 205 verifier, and benchmark MPC evaluation and `libOTe` Beaver preprocessing as separate components. During online signing experiments, key shares and caches are preloaded on device while synthetic Beaver triples stream from host memory as consumed, reproducing transfer and consumption overhead while isolating online signing from preprocessing.

Testbeds and pricing. For server–server signing, we place an AWS host in `us-east-1` and a Runpod host in `us-md1` over the public Internet (2.3 ms RTT, 2.5 Gbit/s bandwidth). We benchmark one NVIDIA RTX PRO 6000 GPU per party ($\$5.46/\text{h}$ nominal combined) and CPU hosts with 16–128 physical cores per party ($\$0.094$ nominal per combined core-hour).

For Beaver preprocessing, we deploy a 64-pair keyless CPU cluster of AWS `c8a.xlarge` and GCP `n4-standard-8` workers (256 physical cores per party), paired with an AWS `c8a.48xlarge` and GCP `n4-standard-64` main host that emulate the signing nodes aggregating and consuming incoming triples

($37.01/h nominal combined worker rate).[2] Workers stream generated triple batches over each provider's internal network to their main host at over 20 Gbit/s per party.

For client–server signing, we pair a GPU server with a four-core CPU client over emulated Wi-Fi (30 ms RTT, 100/40 Mbit/s down/up) and moderate 5G (50 ms RTT, 100/10 Mbit/s down/up). All monetary costs assume 50% compute utilization and include public-Internet egress ($0.09/GB on AWS and $0.08/GiB on GCP at scale; Runpod egress and all ingress are free).

6.2. Server–Server Online Signing

We select L and C to balance online signing latency against prefix-preparation time and storage, as detailed in Appendix B. Figure 9 compares our multi-core CPU and GPU engines across all four parameter sets. Scaling from 16 to 64 CPU cores cuts 128f signing time from 12.50 s to 3.94 s, though doubling to 128 cores yields only 3.39 s. This pattern of diminishing returns holds across all four modes.

Since the parallel signing sweep evaluates thousands of identical hash instances in lockstep, it maps even more effectively onto a GPU: a single RTX PRO 6000 per party achieves both the lowest latency cost across every parameter set, signing 128f in 2.09 s and 256f in 7.84 s, while completing 128s and 256s in 44.99 s and 96.46 s.[3] Because CPU and GPU executions transmit the same protocol messages, they incur the same public-network egress cost ($0.0388 for 128f), which dominates the total per-signature cost; on compute alone, the GPU is also cheaper than every CPU configuration (0.63¢ vs. 1.04–2.27¢ for 128f).

Table 2 reports prefix signing alongside hybrid OTS (Section 3.5), which samples the lower-tree and FORS components locally and evaluates only the signing leaves in the cached layers. This reduces online signing to 0.76 s, 0.80 s, 0.65 s, and 0.65 s across the four modes and cuts total per-signature compute cost to $0.0024–$0.0052. While this baseline yields a 2^{31}–2^{39} signing budget, the cap is tunable: each additional layer in L, C, or evaluated below the cache expands capacity by $2^{h'}$ for a linear increase in online or offline work (Appendix A.3).

6.3. Preprocessing

Beaver triple supply. The online measurements above isolate the signing engine by beginning with synthetic Beaver triples in host memory; a complete deployment must also generate and deliver this material. Because triple generation is input-independent and requires no SLH-DSA key shares, we run it on separate CPU workers and scales horizontally without expanding the trusted signing infrastructure.

2. We use GCP rather than Runpod for the second party's preprocessing cluster to provide a high-bandwidth internal VPC network between workers and the main host, which Runpod does not offer.

3. Public-network bandwidth is the main bottleneck here, especially for GPUs; co-locating both parties on AWS (128f/256f/128s/256s) reduces GPU signing to 0.63 s/1.95 s/8.76 s/21.57 s and 128-core CPU signing to 1.94 s/6.98 s/47.40 s/80.48 s.

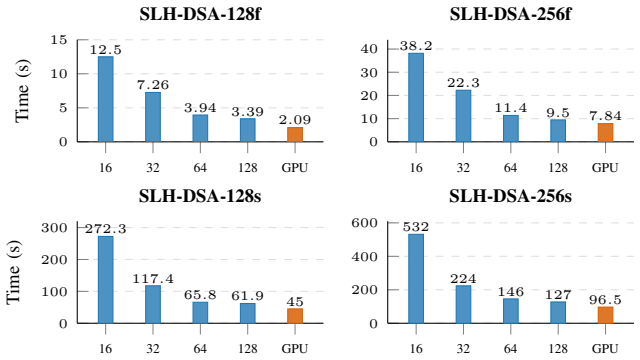


Figure 9. Cross-cloud online prefix-signing latency across 16–128 CPU cores (blue) and a single GPU (orange).

To quantify this cost, we benchmark our complete CPU triple-generation pipeline: `libOTe`'s malicious silent-OT binary-triple generator followed by batched conversion into our $\mathbb{F}_{2^{40}}$-authenticated representation and streaming to the main host.[4] Across our 64-pair worker cluster (256 physical cores per party), this pipeline generates up to 220 million authenticated triples per second and streams them to each main host at over 20 Gbit/s, giving the per-signature times, communication, and costs in Table 3 where public-Internet egress accounts for 64–80% of the cost. Additional worker pairs scale throughput almost linearly, trading hourly capacity for faster generation at nearly the same per-signature cost.[5] Compared to authenticated garbling on the same optimized circuits, our combined offline and online phases are roughly an order of magnitude faster and several times cheaper than our analytical estimates for garbling, which requires over 55 GB of circuit communication (177 s wire time, $4.98 egress) for 128f alone, as we detail in Appendix D.

The triple footprint governs how this material is supplied to the GPU. While the 128f set (26.4 GB) fits in GPU memory, the larger modes exceed device capacity. Because our engine already streams triples from host memory to the GPU in bounded batches, these modes can directly feed those host buffers from local NVMe SSDs or the worker cluster during signing; moreover, to avoid storing expanded triples on disk, each party can instead store only its local randomness seed and received OT messages and locally replay triple expansion when needed, cutting storage by over 6×. This also highlights the benefit of hybrid OTS, whose triples occupy under 2 GB, require at most 0.28 GB of cross-cloud communication, and take 0.21–0.75 s (1.25–6.15¢) to generate across all four parameter sets.

Key generation and per-prefix preparation. Before online signing begins, the parties evaluate all L prefix trees (whose top layer produces `PK.root`) and the C-layer public-key cache; sequential prefix rotation reuses the upper prefix

4. GPU OT libraries such as `cuOT` [59] currently support only semi-honest RCOT; extending them to malicious security would significantly accelerate triple generation.

5. Cloud-provider quota limits capped our testbed at 64 worker pairs.

TABLE 2. SERVER–SERVER GPU ONLINE PERFORMANCE FOR PREFIX SIGNING WITH PUBLIC-KEY CACHING AND HYBRID OTS SIGNING, AVERAGED OVER 100 RUNS. COSTS INCLUDE 50% COMPUTE UTILIZATION AND AWS PUBLIC-INTERNET EGRESS.

			Prefix signing			Hybrid OTS signing			
Mode	L	C	Time (s)	Comm. (GB)	Cost ($)	Time (s)	Comm. (GB)	Cost ($)	Lifetime budget
128f	9	4	2.090	0.432	0.0452	0.764	0.026	0.0047	2^{38}
256f	7	3	7.840	1.514	0.1600	0.798	0.031	0.0052	2^{39}
128s	3	1	44.988	9.210	0.9654	0.647	0.005	0.0024	2^{35}
256s	3	1	96.464	17.483	1.8661	0.647	0.010	0.0029	2^{31}

TABLE 3. PER-SIGNATURE CPU BEAVER TRIPLE GENERATION FOR PREFIX AND HYBRID OTS SIGNING ON A 64-PAIR WORKER CLUSTER (256 PHYSICAL CORES PER PARTY). COSTS INCLUDE 50% COMPUTE UTILIZATION AND BIDIRECTIONAL AWS–GCP EGRESS.

	Prefix signing					Hybrid OTS signing				
Mode	Triples (10^9)	Time (s)	Comm. (GB)	Storage (GB)	Cost ($)	Triples (10^9)	Time (s)	Comm. (GB)	Storage (GB)	Cost ($)
128f	1.716	8.69	3.84	26.4	0.8104	0.1057	0.53	0.24	1.63	0.0504
256f	6.052	28.01	13.50	93.0	2.7967	0.1235	0.75	0.28	1.90	0.0615
128s	36.831	167.58	82.17	566.3	16.9631	0.0211	0.21	0.05	0.33	0.0125
256s	69.876	698.74	155.89	1,074.3	40.0117	0.0412	0.27	0.09	0.63	0.0204

TABLE 4. SERVER–SERVER PREFIX AND CACHE PREPARATION.

Mode	Initial prep.	Prefix rotation	Storage
128f	9.57 min	9.42 min	0.134 MiB
256f	17.0 min	16.5 min	0.258 MiB
128s	4.37 min	1.08 min	16.0 KiB
256s	7.70 min	1.98 min	16.0 KiB

TABLE 5. CLIENT–SERVER OTS ONLINE LATENCY (MS), AMORTIZED RESERVE PREPARATION (MIN), AND SIGNATURE CAPACITY AT 50% LEAF OCCUPANCY.

			Online (ms)		Amortized prep. (min)			Signatures	
Mode	L	C	Wi-Fi	5G	Triples	WOTS	Total	Prefix	Life
128f	10	3	482	525	66.9	13.3	80.2	2^8	2^{38}
256f	7	2	431	533	60.6	11.4	72.0	2^7	2^{35}
128s	3	1	272	379	59.1	9.2	68.3	2^8	2^{35}
256s	3	1	239	342	57.6	8.9	66.6	2^7	2^{31}

trees and updates only the bottom prefix leaf and its cache. Table 4 reports the total preparation time (Beaver generation plus GPU WOTS$^+$ evaluation) and cache size. Amortized over tens of millions of signatures per prefix, this cost is negligible. Moreover, because upper cached layers contain far fewer trees, the signers can prepare only the top cached layers first and immediately begin online signing—evaluating uncached layers online at slightly higher latency

6.4. Client–Server Signing with One-Time Subtrees

For client–server signing, we evaluate the *prefix with one-time subtrees* mode of Section 3.5 with a four-core CPU client. Unlike hybrid OTS, this mode precomputes all secret-dependent work and evaluates no MPC gates online; signing requires only server-side grinding, local hashing, and four lightweight communication rounds, completing in 239–533 ms across Wi-Fi and 5G (Table 5). We retire prefixes at 50% boundary-leaf occupancy to bound expected grinding overhead by $2\times$, though higher occupancy can trade additional grinding for a larger per-prefix budget. Beyond occupancy, the signing budget scales exponentially along two axes: increasing C expands the per-prefix reserve at the cost of longer preprocessing, while increasing L expands the lifetime budget across prefixes at the cost of more server-side grinding, and subtly longer pre-procesisng —which is especially attractive here since online signing otherwise takes only a few hundred milliseconds.

The main cost is preparing the periodic signing reserve, which we benchmark under the emulated Wi-Fi profile above with `libOTe` generating approximately 3.1 million authenticated triples per second. Replenishing a reserve takes 66.6–80.2 min and prepares 128–256 usable signatures, supporting billions of signatures over the key's lifetime; the initial session additionally computes `PK.root` and the full prefix path (90.9/101.4/272.0/266.2 min for 128f/256f/128s/256s), which subsequent sequential prefixes largely reuse. Notably, Beaver triple generation accounts for $\approx 85\%$ of this time and is independent of the SLH-DSA key shares, so it can be delegated to keyless worker servers without exposing either party's key share, leaving only 8.9–13.3 min of WOTS$^+$ evaluation.

7. Conclusion

We have shown that maliciously secure distributed signing for SLH-DSA is practical without relaxing FIPS 205 verification. By exploiting the hypertree structure and accelerating Boolean GMW on GPUs, Aker signs in seconds across clouds and in four lightweight rounds for mobile clients. Future work includes extending our N-out-of-N protocols to t-out-of-N thresholds, supporting the SHA-2 parameter sets, and accelerating malicious silent OT on server and mobile GPUs.

References

[1] D. Bluvstein, S. J. Evered, A. A. Geim, S. H. Li, H. Zhou, T. Manovitz, S. Ebadi, M. Cain, M. Kalinowski, D. Hangleiter, J. P. B. Ataides, N. Maskara, I. Cong, X. Gao, P. S. Rodriguez, T. Karolyshyn, G. Semeghini, M. J. Gullans, M. Greiner, V. Vuletić, and M. D. Lukin, "Logical quantum processor based on reconfigurable atom arrays," *Nature*, vol. 626, pp. 58–65, 2024.

[2] Google Quantum AI and Collaborators, "Quantum error correction below the surface code threshold," *Nature*, vol. 638, pp. 920–926, 2025.

[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1994, pp. 124–134.

[4] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the third round of the NIST post-quantum cryptography standardization process," National Institute of Standards and Technology, Tech. Rep. NIST IR 8413-upd1, 2022.

[5] National Institute of Standards and Technology, "Stateless hash-based digital signature standard," National Institute of Standards and Technology, Tech. Rep. FIPS 205, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf

[6] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2129–2146.

[7] National Institute of Standards and Technology, "Module-lattice-based digital signature standard," National Institute of Standards and Technology, Tech. Rep. FIPS 204, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf

[8] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON: Fast-fourier lattice-based compact signatures over NTRU," Submission to the NIST Post-Quantum Cryptography Standardization Process, 2020. [Online]. Available: https://falcon-sign.info/

[9] K. Bashiri, S. Fluhrer, M. Gazdag, D. V. Geest, and S. Kousidis, "Internet X.509 public key infrastructure: Algorithm identifiers for SLH-DSA," RFC 9909, IETF, 2025.

[10] M. Kudinov and J. Nick, "Hash-based signature schemes for bitcoin," Cryptology ePrint Archive, Paper 2025/2203, 2025. [Online]. Available: https://eprint.iacr.org/2025/2203

[11] Federal Office for Information Security (BSI), "Cryptographic mechanisms: Recommendations and key lengths," BSI, Tech. Rep. BSI TR-02102-1, 2025.

[12] Y. Lindell, "Secure multiparty computation (MPC)," *Communications of the ACM*, vol. 64, no. 1, pp. 86–96, 2021.

[13] Coinbase, "Digital asset management with MPC," Coinbase Blog, https://www.coinbase.com/blog/digital-asset-management-with-mpc-whitepaper, 2023.

[14] Fordefi, "Why MPC wallets are becoming the standard for institutions moving into Web3," https://medium.com/fordefi/why-mpc-wallets-are-becoming-the-standard-for-institutions-moving-into-web3-54b91ae2978e, 2022.

[15] Ethereum Foundation, "Distributed validator technology (DVT)," https://ethereum.org/staking/dvt/, 2024.

[16] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and abhi shelat, "Blind certificate authorities," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1015–1032.

[17] Y. Lindell, "Fast secure two-party ECDSA signing," in *Advances in Cryptology – CRYPTO 2017*, ser. Lecture Notes in Computer Science, vol. 10402. Springer, 2017, pp. 613–644.

[18] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled, "UC non-interactive, proactive, threshold ECDSA with identifiable aborts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2020, pp. 1769–1787.

[19] J. Doerner, Y. Kondi, E. Lee, and abhi shelat, "Threshold ECDSA in three rounds," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3047–3065.

[20] C. Komlo and I. Goldberg, "FROST: Flexible round-optimized Schnorr threshold signatures," in *Selected Areas in Cryptography – SAC 2020*, ser. Lecture Notes in Computer Science, vol. 12804. Springer, 2020, pp. 34–65.

[21] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-Group signature scheme," in *Public Key Cryptography – PKC 2003*, ser. Lecture Notes in Computer Science, vol. 2567. Springer, 2003, pp. 31–46.

[22] L. T. A. N. Brandão and R. Peralta, "NIST first call for multi-party threshold schemes," National Institute of Standards and Technology, Tech. Rep. NIST IR 8214C, 2026.

[23] S. Agrawal, "Working together on Bitcoin's post-quantum readiness," Coinbase Blog, https://www.coinbase.com/blog/working-together-on-btc-post-quantum-readiness, Oct. 2026.

[24] S. Celi, R. del Pino, T. Espitau, G. Niot, and T. Prest, "Efficient threshold ML-DSA," in *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, 2026. [Online]. Available: https://eprint.iacr.org/2026/013

[25] A. Bienstock, L. de Castro, D. Escudero, A. Polychroniadou, and A. Takahashi, "Quorus: Efficient, scalable threshold ML-DSA signatures from MPC," in *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, 2026. [Online]. Available: https://eprint.iacr.org/2025/1163

[26] R. Garg, D. Escudero, A. Polychroniadou, A. Takahashi, and X. Wang, "Thresholdizing standardized FALCON signatures," Cryptology ePrint Archive, Paper 2026/1300, 2026. [Online]. Available: https://eprint.iacr.org/2026/1300

[27] J. Doerner, Y. Kondi, and L. N. Rosenbloom, "Sometimes you can't distribute random-oracle-based proofs," in *Advances in Cryptology – CRYPTO 2024*. Springer, 2024. [Online]. Available: https://eprint.iacr.org/2023/1381

[28] Y. Kondi and N. Kumar, "A black-box impossibility result for threshold signing standard hash-based signatures," Cryptology ePrint Archive, Paper 2026/2194, 2026. [Online]. Available: https://eprint.iacr.org/2026/2194

[29] D. Cozzo and N. P. Smart, "Sharing the LUOV: Threshold post-quantum signatures," in *Cryptography and Coding*, ser. Lecture Notes in Computer Science, vol. 11929. Springer, 2019, pp. 128–153. [Online]. Available: https://eprint.iacr.org/2019/1060

[30] Y. Kondi, N. Kumar, and H. Vanegas, "Threshold signing for standard hash-based signatures is expensive," Presentation slides, NIST Workshop on Multi-Party Threshold Schemes (MPTS 2026), Jan. 2026, presented by Naman Kumar on January 29, 2026; program title: Black-Box Threshold Signing of Hash-Based Signatures is Impossible. [Online]. Available: https://csrc.nist.gov/presentations/2026/mpts2026-4b4

[31] J. H. Cheon, K. Choi, H. Joo, and T. Kim, "One-round threshold XMSS and SPHINCS+ from fully homomorphic encryption," Cryptology ePrint Archive, Paper 2026/2360, 2026. [Online]. Available: https://eprint.iacr.org/2026/2360

[32] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, "Protecting accounts from credential stuffing with password breach alerting," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 1556–1571. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/thomas

[33] I. Malinger, "Introducing Curv's newest MPC breakthrough: Curv Air Gap," Medium, https://medium.com/@itaymalinger/introducing-curvs-newest-mpc-breakthrough-curv-air-gap-bdf847a502a1, 2020.

[34] Zengo, "Crypto & Bitcoin wallet: Signature security standards," https://zengo.com/security-in-depth/, 2023.

[35] A. Urban, "Deploying MPC in the wild," DFNS Blog, https://dfns.co/article/deploying-mpc-in-the-wild, 2026.

[36] Fireblocks, "What is MPC (multi-party computation) crypto custody?" https://www.fireblocks.com/what-is-mpc/, 2024.

[37] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017, pp. 259–282. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[38] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1987, pp. 218–229.

[39] T. K. Frederiksen, M. Keller, E. Orsini, and P. Scholl, "A unified approach to MPC with preprocessing using OT," Cryptology ePrint Archive, Paper 2015/901, 2015. [Online]. Available: https://eprint.iacr.org/2015/901

[40] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," Cryptology ePrint Archive, Paper 2019/1159, 2019. [Online]. Available: https://eprint.iacr.org/2019/1159

[41] S. Raghuraman, P. Rindal, and T. Tanguy, "Expand-convolute codes for pseudorandom correlation generators from LPN," Cryptology ePrint Archive, Paper 2023/882, 2023. [Online]. Available: https://eprint.iacr.org/2023/882

[42] D. Masny and P. Rindal, "Endemic oblivious transfer," Cryptology ePrint Archive, Paper 2019/706, 2019. [Online]. Available: https://eprint.iacr.org/2019/706

[43] P. Rindal, L. Roy, and L. K. L. Ng, "libOTe: an efficient, portable, and easy to use Oblivious Transfer Library," https://github.com/osu-crypto/libOTe.

[44] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, "Optimizing authenticated garbling for faster secure two-party computation," in *Advances in Cryptology – CRYPTO 2018*, ser. LNCS, vol. 10993. Springer, 2018, pp. 365–391.

[45] R. del Pino, S. Katsumata, M. Maller, F. Mouhartem, T. Prest, and M.-J. O. Saarinen, "Threshold Raccoon: Practical threshold signatures from standard lattice assumptions," in *Advances in Cryptology – EUROCRYPT 2024*, ser. Lecture Notes in Computer Science, vol. 14652. Springer, 2024, pp. 219–248.

[46] D. F. Aranha, G. Borin, S. Celi, and G. Niot, "Optimizing and implementing threshold MAYO," Cryptology ePrint Archive, Paper 2026/710, 2026. [Online]. Available: https://eprint.iacr.org/2026/710

[47] J. Kelsey, N. Lang, and S. Lucks, "Turning hash-based signatures into distributed signatures and threshold signatures: Delegate your signing capability, and distribute it among trustees," *IACR Communications in Cryptology*, vol. 2, no. 2, Jul. 2025. [Online]. Available: https://cic.iacr.org/p/2/2/24

[48] D. Boneh, B. Bünz, Justin D., Y. Litchev, and K. Nazirkhanova, "PRAWNS: Threshold hash-based signatures from threshold PRFs," Cryptology ePrint Archive, Paper 2026/2142, 2026. [Online]. Available: https://eprint.iacr.org/2026/2142

[49] Conduition, "Pruning hypertrees for the lax and lazy," Online article, Apr. 2026, published April 20, 2026. [Online]. Available: https://conduition.io/cryptography/hypertree-pruning/

[50] A. Adomnicăi, "Towards practical multi-party hash chains using arithmetization-oriented primitives: With applications to threshold hash-based signatures," *IACR Communications in Cryptology*, vol. 2, no. 4, 2026. [Online]. Available: https://cic.iacr.org/p/2/4/23

[51] D. Porechna, "Threshold authorization without threshold signatures: Signature-agnostic MPC custody," arXiv:2607.08226, 2026. [Online]. Available: https://arxiv.org/abs/2607.08226

[52] T. K. Frederiksen and J. B. Nielsen, "Fast and maliciously secure two-party computation using the GPU," Cryptology ePrint Archive, Paper 2013/046, 2013. [Online]. Available: https://eprint.iacr.org/2013/046

[53] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," arXiv preprint arXiv:2109.00984, 2021. [Online]. Available: https://arxiv.org/abs/2109.00984

[54] J.-L. Watson, S. Wagh, and R. A. Popa, "Piranha: A GPU platform for secure computation," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022, pp. 827–844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/watson

[55] National Institute of Standards and Technology, "Secure hash standard (SHS)," National Institute of Standards and Technology, Tech. Rep. FIPS 180-4, 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[56] ——, "SHA-3 standard: Permutation-based hash and extendable-output functions," National Institute of Standards and Technology, Tech. Rep. FIPS 202, 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[57] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[58] J. Boyar and R. Peralta, "A new combinational logic minimization technique with applications to cryptology," in *Experimental Algorithms (SEA 2010)*, ser. LNCS, vol. 6049. Springer, 2010, pp. 178–189.

[59] A. Gan, S. Yuki, T. Rogers, and Z. Ghodsi, "cuOT: Accelerating oblivious transfer on GPUs for privacy-preserving computation," in *2025 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2025, pp. 300–311. [Online]. Available: https://ieeexplore.ieee.org/document/11050054

[60] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Automata, Languages and Programming (ICALP 2008)*, ser. LNCS, vol. 5126. Springer, 2008, pp. 486–498.

Appendix A.
Prefix State and Usage Limits

Restricting signing to an active prefix requires tracking prefix usage and retiring the prefix once it reaches its usage limit. We first describe how both deployment modes manage prefix state and recover from state loss, and then detail the FORS usage limits that govern prefix retirement in the public-key caching mode.

A.1. State and Recovery

Both deployment modes separate state into two levels: a durable *prefix-allocation record* and the active prefix's *usage state*. Each party stores the allocation record alongside its secret-key shares with integrity and rollback protection. A prefix is durably marked as allocated before it is first used, and this mark is never removed. We allocate prefixes sequentially in base-$2^{h'}$ order, so the durable allocation record reduces to a monotonic prefix index.

During normal operation, the parties track usage within the active prefix according to the deployment mode:

- **Public-key caching:** The parties maintain the active prefix's FORS usage state, updating it before each signing sweep and retiring the prefix once it reaches a predefined usage limit.
- **One-time subtrees:** The parties maintain the set U of reserved boundary leaves, marking each leaf as reserved before opening $[\rho]$, and retiring the prefix once U reaches the occupancy limit.

Crucially, the active prefix's usage state does not need a durable write on every signature. If this operational state is lost or cannot be trusted after a crash, the parties simply abandon the active prefix and activate the next unallocated prefix in the durable allocation record—requiring a durable update only when switching prefixes. This addresses the same operational concern as SHRINCS [10], which pairs a stateful scheme with a stateless fallback after state loss; our construction instead recovers by switching to a fresh prefix under the same public key, provided the prefix-allocation record survives alongside the key shares.

A.2. FORS Usage Limits

Tracking used FORS instances. In our baseline configuration, the signers prevent FORS reuse by recording the index of each used FORS instance beneath the active prefix and rejecting any randomizer whose digest selects a previously used instance. A prefix covering L layers fixes $b = Lh'$ address bits and leaves 2^{h-b} FORS instances beneath it, where each index requires 5 bytes to store. Choosing a modest storage budget of 80 MiB records $Q_{\text{prefix}} = 2^{24}$ FORS instances per prefix ($\approx$ 16.8 million signatures), yielding roughly 2^{50} total signatures across the 2^b prefixes as shown in Table 6; allocating more storage scales capacity linearly (e.g., 640 MiB supports 2^{27} signatures per prefix and up to 2^{55} total). Because the prefix is retired while almost all of its 2^{h-b} FORS instances remain unused, collisions with previously used instances during randomizer search are negligible. Since each FORS instance is used only once and each upper signing leaf always signs the same fixed child-tree root, this eliminates multi-signature FORS exposure altogether. Moreover, because FORS is a few-time signature scheme, deployments can optionally allow each recorded FORS instance to be used a small predefined number of times before rejecting it.

TABLE 6. SERVER–SERVER SIGNING CAPACITY FOR PREFIX SIGNING ($Q_{\text{prefix}} = 2^{24}$ TRACKED FORS INSTANCES)

Mode	L	Lifetime ($2^b \cdot Q_{\text{prefix}}$)
128f	9	2^{51}
256f	7	2^{52}
128s	3	2^{51}
256s	3	2^{48}

TABLE 7. PER-PREFIX RETIREMENT LIMITS $A_{\max}$ AND LIFETIME CAPACITY FOR $c = 4$, $\approx 98.2\%$ SEARCH SUCCESS PROBABILITY PER ATTEMPT.

Mode	L	Per prefix ($A_{\max}$)	Lifetime ($2^b \cdot A_{\max}$)
128f	9	2^{35}	2^{62}
256f	7	2^{34}	2^{62}
128s	3	2^{35}	2^{62}
256s	3	2^{38}	2^{62}

Bounding total signing attempts. Alternatively, instead of recording every used FORS index, the signers can allow random FORS reuse as in standard SLH-DSA and track only a single counter A of signing attempts under the current prefix, supporting orders of magnitude more signatures per prefix. To prevent a malicious searcher from grinding arbitrarily many candidates to bias FORS selection, the parties derive at most $B = c\,2^b$ candidate randomizers per attempt from a fresh offline seed $[\rho]$ (succeeding with probability $1 - (1 - 2^{-b})^B \approx 1 - e^{-c}$). After A attempts, the number of exposures at any fixed FORS instance beneath the prefix is bounded by $\text{Binomial}(AB, 2^{-h})$. Retiring the prefix after $A_{\max} = \lfloor Q_{\text{ref}}/B \rfloor$ attempts, where $Q_{\text{ref}} = 2^{64}$ is the FIPS 205 signing budget [5], [6], ensures $AB \le Q_{\text{ref}}$ and keeps the expected FORS forgery probability at every instance within the standard SLH-DSA bound. Table 7 gives the resulting per-prefix limits $A_{\max}$ for $c = 4$, reaching 2^{34}–2^{38} attempts per prefix and matching the full 2^{62} lifetime capacity across all prefixes.

A.3. Hybrid OTS Signing Budget

In hybrid OTS mode (Section 3.5), the signing boundary can be placed at any layer at or below the C cached layers, sampling all lower layers and FORS components locally. In our benchmarked configuration, we place the boundary directly at the lowest cached layer so no full lower trees are

TABLE 8. SERVER–SERVER USABLE SIGNING CAPACITY FOR HYBRID OTS SIGNING AT 50% BOUNDARY-LEAF OCCUPANCY.

Mode	L	C	Per prefix ($2^{Ch'-1}$)	Lifetime ($2^{(L+C)h'-1}$)
128f	9	4	2^{11}	2^{38}
256f	7	3	2^{11}	2^{39}
128s	3	1	2^{8}	2^{35}
256s	3	1	2^{7}	2^{31}

evaluated online, leaving $2^{Ch'}$ boundary leaves beneath the active prefix. Because randomizer search must skip candidate digests that land on an already-used boundary leaf, retiring each prefix at 50% occupancy bounds expected grinding overhead to at most $2\times$ and yields $Q_{\text{prefix}} = 2^{Ch'-1}$ usable signatures per prefix, or $2^{(L+C)h'-1}$ across all $2^{Lh'}$ prefixes (Table 8). More generally, the budget is independently configurable: placing the boundary k uncached layers below the cache—or increasing L or C—multiplies both the per-prefix and lifetime budgets by $2^{h'}$ per layer, trading online evaluation of k lower trees, online grinding, or cache preparation for higher signing capacity.

Appendix B. Parameter Selection

Both deployment modes are configured by the number of prefixed layers L and the number of cached or retained layers C beneath the prefix. Because each hypertree layer has branching factor $2^{h'}$, linear changes in L and C induce exponential changes in randomizer search, preparation time, storage, and signing capacity. We select (L, C) for each deployment mode to reduce online latency while keeping per-prefix preparation and storage within practical limits. Since all four parameter sets follow the same linear-versus-exponential trade-offs, Figure 10 illustrates them on SLH-DSA-128f ($h' = 3$).

Prefix depth L. Increasing L removes one active XMSS tree per layer from online MPC evaluation, reducing online signing work linearly while multiplying the expected number of randomizer trials by $2^{h'}$ (Figure 10(a)). While randomizer search time is initially negligible, it grows by $2^{h'}$ per layer and eventually outweighs the linear MPC savings. We select the largest L before search latency rises sharply, keeping randomizer grinding a small fraction of online signing time. In OTS mode, increasing L also multiplies the number of disjoint prefixes—and thus the key's total lifetime signature capacity—by $2^{h'}$. Sequential prefix allocation makes the additional upper-path preparation negligible when amortized across prefixes, although the first prefix must initialize the complete upper path. The primary recurring cost of increasing L is therefore the additional online randomizer grinding; for 128f, we select $L = 9$ in server–server mode and $L = 10$ in client–server OTS mode to reach a 2^{38} lifetime budget with $C = 3$.

Server–server public-key cache depth C. With L fixed, each cached layer C removes another active tree's non-

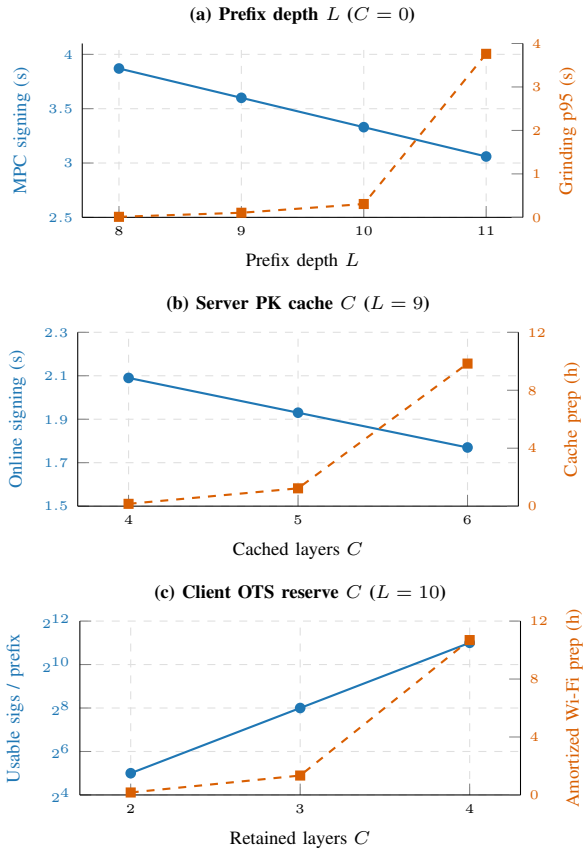


Figure 10. Parameter selection trade-offs for SLH-DSA-128f (solid blue, left axis vs. dashed orange, right axis): (a) online MPC signing time at $C = 0$ vs. p95 R-grinding latency as a function of L; (b) server–server online signing time vs. PK-cache preparation time as a function of C; and (c) client–server usable signatures per prefix Q_{prefix} vs. Wi-Fi reserve preparation time as a function of C.

signing leaves from online evaluation, reducing online signing time linearly while multiplying the number of cached public keys, and thus per-prefix preparation time and storage, by $2^{h'}$ (Figure 10(b)). We select the largest C before this exponential growth makes preprocessing impractical ($C = 4$ for 128f, taking 9.42 min to prepare); because a single prefix supports tens of millions of signatures, this one-time preparation cost is negligible when amortized over the prefix's lifetime. Moreover, because the top cached layers contain very few trees and can be prepared quickly, the signers can begin signing as soon as those upper layers are ready while generating the remaining lower cached layers in the background.

Client–server OTS parameters. In OTS mode, online latency depends only on prefix grinding governed by L and four communication rounds, making a larger L a practical option when a larger lifetime signature budget is desired. Meanwhile, C governs the trade-off between offline preparation time and the per-prefix signing capacity: each additional layer in C multiplies both the per-prefix quota and total lifetime budget by $2^{h'}$, at the cost of $2^{h'}\times$ more

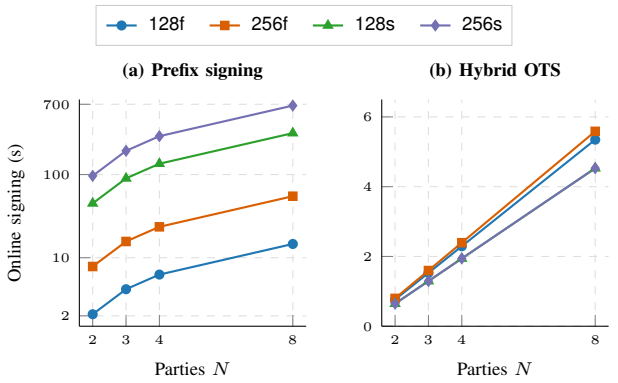


Figure 11. Measured two-party and estimated N-party GPU online signing latency for (a) prefix signing with public-key caching and (b) hybrid OTS signing.

preprocessing work (Figure 10(c)). Retiring each prefix at 50% leaf occupancy bounds grinding overhead to $2\times$, yielding $Q_{\text{prefix}} = 2^{Ch'-1}$ usable signatures per prefix and $Q_{\text{total}} = 2^{(L+C)h'-1}$ over the key's lifetime, though higher occupancy can be used at the cost of additional grinding. We select $C = 3$ for 128f so an amortized session prepares 256 usable signatures in 80.2 min, while a larger C can be chosen for a much larger per-prefix and lifetime budget. The first reserve also initializes the full prefix path, taking 90.9 min for 128f and up to $4\times$ the amortized time for the s modes.

Appendix C. Multi-Party Extension

The structural optimizations detailed in Section 3 operate on the SLH-DSA hypertree and evaluate the hash function directly as a Boolean circuit, making them agnostic to the number of signers. Because our protocols and implementation build on Tinier [39], which natively supports dishonest-majority computation over additive Boolean shares, they extend directly from two parties to N-out-of-N signing. In an N-party execution, the circuit evaluation and round count remain unchanged, while each party interacts with $N-1$ peers so per-party communication and authentication work scale linearly with $N-1$.

To illustrate how performance scales with N, we estimate N-party online signing runtimes under the configurations of Table 2 by scaling our measured two-party protocol work by $N-1$, capturing the increased per-party computation and communication volume without modeling multi-party network topologies. Figure 11 shows the resulting GPU online signing times for prefix signing with public-key caching (a) and hybrid OTS signing (b). Because hybrid OTS evaluates only the cached signing leaves online, signing remains interactive across all four parameter sets even at $N = 8$.

TABLE 9. ANALYTICAL CIRCUIT-PHASE COMMUNICATION, WIRE SERIALIZATION TIME AT 2.5 GBIT/S, AND AWS EGRESS COST ($0.09/GB) FOR THE KRRW [44] BASELINE (258 BITS/AND AT $\lambda = 128$) ON OUR OPTIMIZED CIRCUITS.

			Prefix signing (KRRW)			Hybrid OTS (KRRW)		
Mode	L	C	Comm.	Wire time	Egress	Comm.	Wire time	Egress
128f	9	4	55.3 GB	177.0 s	$4.98	3.41 GB	10.9 s	$0.31
256f	7	3	195.2 GB	624.6 s	$17.57	3.98 GB	12.8 s	$0.36
128s	3	1	1,187.8 GB	3,801.0 s	$106.90	0.68 GB	2.2 s	$0.06
256s	3	1	2,253.5 GB	7,211.2 s	$202.82	1.33 GB	4.3 s	$0.12

Appendix D. Comparison with Conventional Authenticated Garbling

Prior garbled-circuit evaluations for unoptimized SLH-DSA project end-to-end signing times of 85 minutes to over five hours per signature [29], [30]. To show that conventional symmetric-key garbling remains bottlenecked by communication even after applying all of our structural optimizations, we compute an analytical communication estimate for our reduced signing circuits using the authenticated garbling protocol of Katz et al. (KRRW) [44]. These figures model only the network serialization of the KRRW circuit phase on our cross-cloud link, excluding the prohibtive local computation and TinyOT preprocessing.

Communication model. Under free-XOR [60], linear operations incur no communication, so bandwidth scales linearly with the number of AND gates. Using a conservative $\lambda = 128$-bit computational label length across all modes, KRRW transmits $2\lambda + 2 = 258$ bits per AND gate in its function-dependent phase ($2\lambda + 1$ bits from garbler to evaluator and 1 bit in reverse), compared to 2 bits per party (4 bits aggregate, before batched MAC checks) online in Boolean GMW. While fixed-topology garbled circuits can be streamed before the message is known if addresses are passed on wires, the total per-signature communication volume and egress cost remain unchanged.

Table 9 reports the resulting KRRW circuit communication, serialization time at 2.5 Gbit/s, and AWS egress cost (assuming the garbler is hosted on AWS at $0.09/GB) under the same (L, C) configurations as Table 2. Even excluding KRRW's preprocessing and local compute, transmitting the 55.3–2,253.5 GB of circuit data for prefix signing requires 177.0–7,211.2 s of wire serialization time and $4.98–$202.82 in egress per signature. For 128f, the wire time alone is $85\times$ our 2.09 s online signing latency and $16\times$ our combined online and offline time, while the egress cost alone is approximately $5.8\times$ our total online-plus-offline cost ($0.86 in Table 1).