Turning Hash-Based Signatures into Distributed Signatures and Threshold Signatures

Delegate Your Signing Capability, and Distribute it Among Trustees

John Kelsey[1,2], Nathalie Lang[3] and Stefan Lucks[3]

[1] National Institute of Standards and Technology, Gaithersburg, USA
[2] COSIC, KU Leuven, Computer Security and Industrial Cryptography, Leuven, Belgium
[3] Bauhaus-Universität Weimar, Weimar, Germany

Abstract. We introduce techniques to transform existing stateful hash based signature (HBS) schemes, such as LMS [MCF19] or XMSS [HBG+18], into efficient threshold and distributed signature schemes. Our approach requires a trusted dealer for setup, and uses a large (up to a few GiB, typically) common reference value (`CRV`) for each new public key. The dealer generates the keypair and distributes shares of the signing key to the trustees, while creating the `CRV`. Signing involves an untrusted aggregator communicating point-to-point with a set of trustees. Only the aggregator needs access to the `CRV`; the trustees need only a PRF key and enough space to remember which one-time keys they have helped to sign with so far. Signing requires two round trips between the aggregator and each participating trustee, and only a little more computation from the trustees and aggregator than is done when signing with the underlying HBS scheme. We reduce the security of our scheme to that of the underlying HBS scheme, assuming the availability of a secure PRF. A dishonest aggregator or tampered `CRV` can prevent valid signatures from being constructed, but does not allow forgeries. Our techniques offer a powerful practical defense against accidental reuse of a one-time key in stateful HBS schemes by requiring multiple trustees to fail in the same way in order for key reuse to occur.

Keywords: hash-based signatures · distributed signatures · provable security

1 Introduction

1.1 Background

Hash-Based Signatures. Hash-based signature (HBS) schemes [Lam79, Mer88, Mer90, Hül13] have a long history, but recent years have witnessed a resurgence in interest in these schemes, because they are resistant to attack by a quantum computer, and because they can be comparatively efficient and small. Recent proposals include stateless schemes such as SPHINCS and its counterparts [BHH+15, AE17, HBD+17], as well as stateful ones like XMSS [HBG+18] and LMS [MCF19]. Additionally, NIST has published requirements for using stateful hash-based signatures safely [CAD+20].

In this paper, we focus solely on stateful HBS. These schemes have a limited number (D) of allowed signatures, and must keep some working state to remain secure. Stateful HBS schemes combine two primitives:

1. A *one-time* signature scheme: a signature construction that allows the generation of only one signature per key. When the user signs two different messages with the

E-mail: john.kelsey@nist.gov (John Kelsey), nathalie.lang@uni-weimar.de (Nathalie Lang), stefan.lucks@uni-weimar.de (Stefan Lucks)

This work is licensed under a "CC BY 4.0" license.
Date of this document: 2025-07-01.

same key, the scheme is compromised and forgeries (valid but fake signatures for other messages) become feasible.

2. A *Merkle tree*: a cryptographic structure that commits to a sequence of D elements, and later allows an efficient proof (a Merkle tree path, always written as `PATH` in this work) that a given element is in a particular position in the sequence. The commitment is called the *root* of the Merkle tree.

Generating a key for these schemes (called a *composite public key* (`CPK`) and *composite private key* (`CSK`) in this paper to distinguish them from the one-time keys) involves generating D one-time keypairs, putting the one-time public keys into the Merkle tree, and publishing the root of the Merkle tree as the composite public key.

A signature in these schemes consists of a Merkle tree path to an unused one-time signing key, and a one-time signature on the message.

Stateful HBS are quantum-resistant, and have security based only on the security of the underlying hash function. However, they require the signer to keep track of which one-time keys have been used. If the signer somehow signs two messages with the same one-time key, anyone who observes both signatures can sign arbitrary messages with the key.

While existing HBS proposals are practical and secure as long as the state is managed correctly, they have, until now, lacked the capability to efficiently create threshold signatures.

Threshold and Distributed Signatures. A threshold signature scheme [Des88, DF90, Sho00, MSY99, BMV19, GKMN21] is a digital signature scheme in which the signing key is split into *shares* with each share assigned to a *trustee*. In a k-of-ℓ threshold signature scheme, there are ℓ trustees, and any combination of k trustees can jointly sign a message. A distributed signature scheme [HPS03, WWW+14, WWW+15] is a generalization of a threshold signature scheme, enabling more complex subsets of trustees to participate in the signing process. For example, consider $\ell = 5$ and $k = 3$. Instead of allowing any three of the five trustees to sign, we might have a scheme where the top trustee (this can be for example the CEO of a company) can sign with any one other trustee, but nobody can sign without the top trustee.

Previously-published distributed or threshold signature schemes (those defined for schemes like RSA or ElGamal signatures) exploit homomorphic properties of the underlying signature schemes. Hash-based signatures lack such convenient properties, so the very existence of an efficient threshold HBS may be surprising.

Formally, we have a set $\mathcal{T}$ of $|\mathcal{T}| = \ell$ "trustees" and different subsets $\mathcal{C}_i \subseteq \mathcal{T}$, which we will refer to as coalitions. The members of any coalition $\mathcal{C}_i$ can jointly generate a valid signature σ for a given message M. For $\ell \geq k$, a k-of-ℓ threshold signature scheme [Des88, DF90, Sho00, MSY99] is a distributed signature scheme with $|\mathcal{C}| = \ell$ in which all subsets $\mathcal{C}_i \subseteq \mathcal{T}$ form a coalition if and only if $|\mathcal{C}_i| = k$. Below, we will often refer to two special cases with only one coalition able to sign: k-of-k signatures and 1-of-1-signatures. The latter ones are just normal single-signer signatures.

1.2 Related and Prior Work

In [BM18], a group signature scheme is constructed for a stateful HBS. Our scheme uses a technique (*sharding*) that is related to their scheme, and in some sense, a group signature scheme can be thought of as a 1-of-ℓ threshold signature scheme.

A natural approach to obtain threshold functionality from a stateful HBS would employ general-purpose multiparty computation techniques. However, this would be quite inefficient. [CS19] describes an attempt to develop a threshold version of SPHINCS+, a

stateless HBS which was selected for standardization in the NIST post-quantum competition. Applying standard multiparty computation (MPC) techniques, they concluded that just computing the hash functions necessary to sign a single message would require 85 minutes. Our techniques, while limited to *stateful* HBS, are much more efficient.

More recently, [KCLM22, KCLM21] proposed using STARKs along with stateful hash-based signatures to get threshold and aggregated signature functionality. However, their techniques produce large signatures that are not standard LMS or XMSS signatures, and indeed use a specialized ZKP-friendly hash function to get acceptable performance.

1.3 Our Usage Model

Unlike most other types of signatures—and, specifically, the types of signatures previous work on distributed and threshold signatures is based on—the stateful hash-based signatures we employ are restricted to generating a *fixed number (D) of signatures*, for some number D. Each signature is associated with a unique $\texttt{KeyID} \in \{0, \ldots, D-1\}$, which is visible to the verifier. For security reasons, no KeyID must ever be used more than once.

This impacts threshold and distributed signatures. As usual, the dealer defines coalitions $C_{i..v}$ of trustees. For our stateful hash-based scheme, each coalition is assigned a set $\texttt{KL}_i \subseteq \{0, \ldots, D-1\}$ of KeyIDs. The trustees from any coalition C_i can jointly sign a message using any KeyID from $\texttt{KL}_i$, but they must make sure no KeyID is ever used twice. Since trustees from coalition C_i cannot know which KeyIDs have been used by coalition C_j, we require $\texttt{KL}_i \cap \texttt{KL}_j = \{\}$ for $i \neq j$. See also the "disjoint honest trustees problem" in Section 4.1. The implications are as follows.

1. A coalition C_i must stop signing when all KeyIDs from $\texttt{KL}_i$ have been exhausted—even if other coalitions C_j still have KeyIDs left.

2. Assume v coalitions $C_{i..v}$ with coalition C_i being able to sign at most ζ_i messages. Then the number of KeyIDs the dealer needs to prepare is $D \geq \sum_i \zeta_i$. Specifically, if a given public key must enable signature generation for ζ messages even in the situation where only one coalition remains intact, then $D \geq v\zeta$. This is relevant for performance: key generation time and CRV size approximately grow proportionally to D, signature length and verification time grow proportionally to $\log D$.

3. For most other threshold and digital signature schemes, every superset of a coalition is also a coalition by itself. In our case, if coalition C_i is picked to sign a message, all other trustees stay passive—even if they would also be willing to sign. If trustees from several coalitions are willing to sign a message, they must decide on one coalition which then generates the signature.

4. For our scheme, it can make sense to define coalitions as supersets of other coalitions. E.g., consider a $(k-2)$-of-k scheme and optimistically assume that most of the time all k trustees will jointly sign, and sometimes a coalition of $k-1$ trustees. Coalitions of $k-2$ trustees are only for emergencies. This optimistic assumption allows us to improve the efficiency. We assign most of the KeyIDs to the single full coalition of k trustees, a decent number KeyIDs to each of the k coalitions of $k-1$ trustees, and only a few KeyIDs to each of the $k(k-1)/2$ emergency coalitions of $k-2$ trustees. Section 6.2 includes a concrete example for an *optimistic 4-of-6 threshold scheme.*

1.4 Our Results

As discussed above, we demonstrate techniques for turning a stateful HBS like those defined in XMSS [HBG+18] and LMS [MCF19] into a distributed signature scheme. The resulting distributed signature scheme is as secure as the original stateful HBS on which it

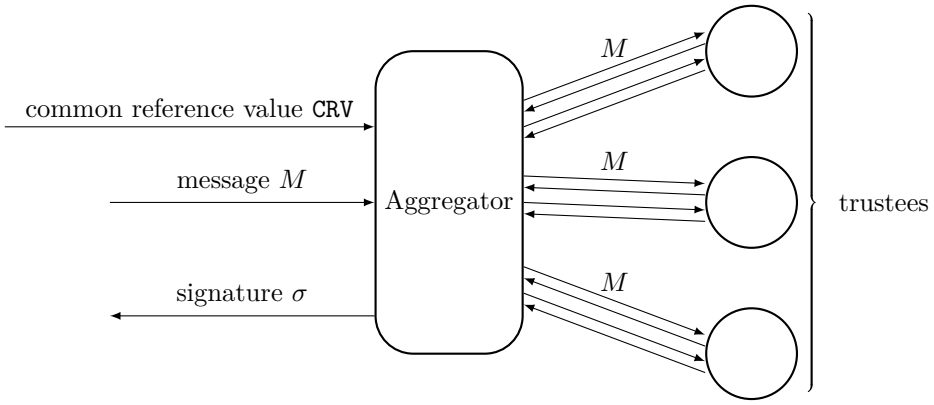

Figure 1: An abstract view at our signing protocols: The message is sent to the aggregator, who also has access to the string of common reference values. The aggregator engages in a protocol with each trustee involved in this signature, with two round-trip messages. At the end of the protocol, the aggregator outputs the signature.

is based, and in fact, the verification algorithm is unchanged between the original stateful HBS and our threshold version.

Setup: Key Generation and Delegation by the Dealer. The trusted dealer runs the key generation function for the stateful HBS, splits the one-time private signing keys into shares, and publishes the public key and a common reference string[1] associated with this particular public key. It then distributes the shares and some other administrative data to the trustees and deletes all private data associated with the key.

Signing Protocol. The signing protocol involves an *aggregator* who is authorized to request signatures[2], but has no power to forge or alter signatures, and k *trustees* who respond to the aggregator's messages to construct a signature. Only the aggregator needs access to the common reference string; the trustees need only their shares and the messages they receive from the aggregator. Signing requires two round trips of communications between the aggregator and each trustee. At the end of a successful run of the signing protocol, the aggregator can construct a valid signature on the message.

The aggregator communicates with the trustees over a point-to-point network; there is no need for a broadcast channel. The role of the aggregator could be taken over by one of the trustees with no loss of either security or performance, but for simplicity, in the rest of this paper, we describe the protocol with the aggregator as a separate entity.

Verification. The threshold or coalition signatures are identical to signatures from the underlying stateful HBS, and are verified in exactly the same way. The verifier has no way to distinguish between signatures created by a single signer and distributed signatures produced by our protocol.

1.5 Security

Our scheme for making distributed stateful hash based signatures relies on the security of the underlying stateful hash based signature scheme, and on a secure PRF. An adversary who can efficiently forge a signature in our scheme without control over a full coalition of trustees can either:

- forge signatures in the underlying stateful hash-based signature scheme,
- or break the underlying PRF,

Notably, neither a corrupt aggregator nor a corrupted version of the `CRV` weakens the security of the scheme[3], and in fact, our security proofs assume that both are fully under the adversary's control. See Section 5 for our security assumptions and proofs.

1.6 Practicality

Our scheme is easy to implement and works well in real-world situations. Each trustee's work in the signing protocol is only marginally more intensive than that required for generating a single hash-based signature. The aggregator does a similar amount of work. During setup, the dealer does about twice the work needed for an ordinary user to set up a stateful hash based signature key with the same number of signatures.

The signing protocol requires two round trips between the aggregator and each participating trustee. These messages can be sent through standard point-to-point connections such that no broadcast channel is needed. Further, the total size of the messages back and forth between the aggregator and each trustee is only a few hundred bytes more than the size of the message being signed. The aggregator also has to retrieve specific values from the `CRV`, but since the `CRV` need not be trustworthy, it can be stored locally, on a remote machine that can be queried online, or even on a cloud storage service.

Depending on the parameters of the stateful HBS, the `CRV` will typically need to be between a few hundred MiB and a few GiB to support 2^{15} signatures. For example, for an LMS instance with 256-bit hashes, 16-element hash chains, and 2^{15} one-time signatures, the CRV requires about 1.1 GiB, see Table 1 below. This is large, but practical. Consider the aggregator being a desktop computer, it is likely to have a hard drive with 1000 GiB or more of storage. The size of the `CRV` and the work done by the dealer during setup scale linearly in the total number of one-time signatures available.

Our scheme allows to exclude trustees that are no longer trusted from further signing, by simply informing the honest trustees to cease signing with any coalition that includes the trustee whose access is to be revoked. No changes to the `CRV` or `CPK` are required.

Our scheme does not work for every imaginable choice of threshold parameters or coalitions in the distributed signature scheme. For example, a 5-of-10 threshold signature scheme with only $\binom{5}{10}$ coalitions is entirely practical, but a 50-of-100 scheme would imply a total number of signing coalitions of $\binom{100}{50}$ and thus be far beyond the pale of practicality. See Section 6 for further discussions on practicality.

1.7 Why it Matters

Post-Quantum Security. The advent of quantum computers as a potential tool for the adversary will be a game-changer in cryptography. Most currently-used digital signature

[1] If the common reference string is altered, this will cause the signatures produced to be invalid, but it will not affect the security of our scheme.

[2] Because any stateful HBS public key will have a limited number of signatures it can support, the ability to request a signature is potentially important.

[3] Note this refers to security against forgery attacks. Deleting the `CRV`, or tampering with the `CRV`, would be very effective as a denial-of-service attack. Fortunately, the `CRV` is not secret. Thus, one can store multiple copies of the `CRV` at different locations. A hash of of the `CRV` can be used to discover tampering.

schemes [RSA78, JMV01, JL17, Nat13] are based on the difficulty of factoring or finding discrete logs, and so will be lost (i.e. become insecure) in a post-quantum world. Similarily, most known threshold/distributed signature schemes, such as [MSY99, AHS20, CKM23], will be lost.

A surprising number of post-quantum signature schemes have been proposed (see for example, the many proposed schemes in [Nat16, Nat23], as well as the recently-standardized [Nat24a]). So far, however, only a few post-quantum *threshold* signatures based on lattices or isogenies have been proposed (see for example [BGG+18, DOTT22, ASY22]). Our scheme adds one more to this list. We claim two significant advantages over these schemes:

1. Our scheme ultimately relies only on a second preimage resistant cryptographic hash function and a good pseudorandom function (PRF). Arguably, this provides a better security assurance, compared to assuming the hardness of newer and apparently less well-studied problems, such as those based on lattices or isogenies.

2. Our scheme provides distributed signatures. That is, we support a richer set of signing coalitions than a threshold signature scheme. To the best of our knowledge, our scheme is the first post-quantum secure *distributed* signature scheme.

Preventing Key Reuse Our techniques can help solve the critical vulnerability inherent in stateful hash-based signatures: the accidental reuse of one-time signing keys. There are many ways for an HBS implementation to end up reusing a key, including attacks, hardware failures, operator mistakes, or even otherwise normal behaviors like restoring a computer to a backed-up state.

When a coalition of k trustees agree to sign a message, they must all sign with the same key; no honest trustee will go along with reusing a one-time key. Consequently, to use a one-time key more than once, all devices in a coalition must simultaneously fail in the same way. By implementing our techniques even with 2-of-2 signatures, a stateful HBS application can get a huge improvement in practical security.

Additionally, many widely-discussed applications of stateful hash-based signatures, such as signing firmware updates for long-lived systems, can benefit from threshold signatures. For example, when using a 4-of-7 threshold signature scheme for a high-value firmware-update-signing application, three or fewer damaged or compromised signing devices do not lead to disaster.

1.8 Guide to the Paper

Section 2 recalls Winternitz one-time signatures and Merkle trees and describes how their combination can be used to derive stateful D-time hash-based signature schemes. Section 3 provides a stepping stone towards proper distributed signature schemes, namely k-of-k signatures. The sharding technique allows to go from k-of-k signatures to proper distributed signatures, as demonstrated in Section 4. Section 5 provides the security analysis of our scheme, Section 6 discusses practical issues and applications of our scheme, and Section 7 concludes the paper with some comments on post-quantum security and other remarks, including ideas for future work.

As a service to the reader, and since this paper provides a lot of functions in pseudocode, Appendix A provides an overview over those functions and briefly describes the algorithmic purpose of each such function.

2 Winternitz-Signatures and Merkle-Trees

In this section, we give an overview of how one-time hash-based signatures and stateful hash based signatures work. There are different ways of constructing stateful HBS, which

differ in some details; our description below attempts to be generic enough to apply to all of them.

2.1 Hash Functions

The most basic building block for HBS is a hash function. Modern HBS schemes incorporate additional information into each hash function, so that each hash function input is distinct, in order to avoid multi-target preimage attacks. In this paper, we call the additional information input into the hash a *tweak*, and write the resulting hash computation as $\texttt{hash}_{\texttt{tweak}}(\text{input})$. Conceptually, $\texttt{hash}_{(X)}$ and $\texttt{hash}_{(Y,Z)}$ are independent hash functions; in practice, they are constructed using some standard hash function such as SHA256, with the tweak and other inputs being domain separated to avoid trivial collisions. Different HBS schemes use different methods of adding these tweaks to the hash function, and our scheme will work with any of them, so we do not specify *how* the tweak is incorporated into the hash, only that it may be incorporated somehow, and that each tweak essentially gives an independent hash function. Further, both the tweak and the main input to the hash will sometimes be a tuple of inputs separated by commas, e.g., $\texttt{hash}_{(1,\texttt{KeyID})}(R, M)$. We assume that these inputs are somehow domain-separated, encoded, and turned into an input to the underlying hash function in an unambiguous way. We leave the specifics to be defined in the specific stateful HBS scheme is being turned into a distributed signature scheme.

2.2 Metadata and Multi-Target Attacks

A typical stateful HBS has some additional data, the "metadata". E.g., instead of

$$\texttt{hash}_{(1,\texttt{KeyID})}(R, M),$$

the hash function call would actually look more like

$$\texttt{hash}_{(1,\texttt{KeyID},\text{metadata})}(R, M).$$

For example, in LMS, this metadata is a unique 128-bit identifier for each composite public key.

The purpose of the metadata is to resist multi-target preimage attacks on the hash function even across different composite keys. Consider a set of composite private/public keys, $(\texttt{CSK}_1, \texttt{CPK}_1)$, ..., $(\texttt{CSK}_\mu, \texttt{CPK}_\mu)$. Without the metadata, an attacker could try to find a new one-time public key with the same hash as (for example) the first one-time public key in *any* of these composite public keys. This would allow the attacker a μ-times speedup. Including the metadata prevents this speedup.

If the metadata are unique, i.e., for $i \neq j$ the metadata attached to the i-th key pair are different from the metadata for the j-th key pair, then the metadata-input to hash function calls depends on the key pair, which is tantamount to using a unique hash function for each key pair. Under appropriate assumptions on the hash function, the benefit from multi-target attacks on stateful HBS, compared to single-target attacks, is negligible. We conjecture that the same reasoning applies to the threshold and distributed signature schemes proposed in the current paper, though in the remainder of this paper, we focus on single-target security.

2.3 Randomized Winternitz One-Time Signatures

Randomized Hashing. Existing HBS standards like LMS and XMSS incorporate randomized hashing. Producing this kind of signature involves generating a randomizer R,

Algorithm 1 Utility functions called by WINTER (Algorithm 2). $\texttt{Split}_w$ splits S into w-bit chunks.

function $\texttt{ToString}(S)$	**function** $\texttt{Split}_w(S)$
return String representation of S	$m \leftarrow$ length of S
	return $s[0 \dots m/w - 1]$

somehow incorporating R into the hash of the message to be signed, and then deterministically signing that hash. The final signature then includes the randomizer. Randomized hashing reduces the security requirements on the hash function used,[4] and, as discussed in [Hül13], can be combined with some other techniques to allow the signatures to be made considerably smaller with no loss in security. Our techniques must support randomized hashing of the message to be signed to maintain compatibility with existing HBS standards. Nevertheless, our techniques can also be applied to a non-randomized version of these signatures in a straightforward way.

Winternitz Signatures. We first need to define some parameters and auxiliary values.

- Let n be the output size of the hash function at hand; n is a critical security parameter, typically 128 or 256.
- Fix a "bit-size" w. For the sake of simplicity, we require n to be a multiple of w. As we continue, it will become clear that the signature size is approximately linear in $1/w$, and the time for key generation, signing and verification is approximately exponential in w. In nearly all cases, $w \in \{1, 2, 4, 8\}$.
- From n and w derive values

$$a = n/w \quad \text{and} \quad c = \lceil \log_2(a * (2^w - 1))/w \rceil. \tag{1}$$

The core idea for Winternitz signatures, summarized in Algorithm 2, is as follows.

- Key Generation
  - Represent the secret key as a two-dimensional array of secret n-bit values, $\mathsf{SK}[0 \dots a + c - 1][0 \dots 2^w - 1]$. Each $\mathsf{SK}[i][0 \dots 2^w - 1]$ consists of an array of 2^w values, constructed by

$$\mathsf{SK}[i][j] = \begin{cases} \text{A random starting value} & j = 0 \\ \texttt{hash}_{\texttt{tweak}}(\mathsf{SK}[i][j-1]) & \text{otherwise} \end{cases}$$

    where the specific function $\texttt{hash}_{\texttt{tweak}}$ and the concrete choice of the tweak depend on different variants of Winternitz signatures.
  - Compute the public key as $\mathsf{PK}[i] = \texttt{hash}_{\texttt{tweak}}(\mathsf{SK}[i][2^w - 1])$ for $i \in \{0, \dots, a + c - 1\}$.
- Signature Generation
  - To sign an n-bit hash value, one first breaks the hash into a chunks, each containing w-bits, and then encodes each w-bit chunk as an entry in one of the hash chains.

[4] In short, the hash function needs to provide some form of second preimage resistance, while a non-randomized Winternitz signature scheme would require collision-resistance.

Algorithm 2 Randomized Winternitz one-time signatures. `Gen` generates a private signing key SK and a corresponding public key PK. $\texttt{Sign}_{\mathsf{SK}}$ first computes a hash h of the randomizer R and the message M and then computes the deterministic Winterniz signature $Z = \texttt{WINTER}(h, \mathsf{SK})$ for h. The pair (R, Z) jointly constitutes a randomized Winternitz signature for M. `Verify` returns a candidate PK' for the public key. Verification passes if and only if $\mathsf{PK}' = \mathsf{PK}$. Note: w is the Winternitz parameter, a is the number of hash chains used to encode the hash, and c is the number used to encode the checksum. `ToString` and $\texttt{Split}_w$ are defined in Algorithm 1.

function `Gen`(`KeyID`)
  for $i \leftarrow 0 \ldots a + c - 1$ **do**
    $\mathsf{SK}[i][0] \xleftarrow{\$} \{0,1\}^n$ // May be produced using a PRF.
    for $j \leftarrow 1 \ldots (2^w - 1)$ **do**
      $\mathsf{SK}[i][j] \leftarrow \texttt{hash}_{(2,i,j,\texttt{KeyID})}(\mathsf{SK}[i][j-1])$
    $Y[i] \leftarrow \mathsf{SK}[i][2^w - 1]$
  $R \xleftarrow{\$} \{0,1\}^n$
  $\mathsf{PK} \leftarrow \texttt{hash}_{(0,\texttt{KeyID})}(Y[0] \| \ldots \| Y[a + c - 1])$
  return $(\mathsf{PK}, \mathsf{SK})$

function $\texttt{Sign}_{\mathsf{SK}}(M, R, \texttt{KeyID})$
  $h \leftarrow \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
  $Z \leftarrow \texttt{WINTER}(h, \mathsf{SK})$
  return (R, Z)

function `WINTER`(h, SK)
  $C \leftarrow a(2^w - 1)$
  $b[0 \ldots a-1] \leftarrow \texttt{Split}_w(h)$
  $\mathrm{CSum} \leftarrow C - \sum_{i=0}^{a-1} b[i]$
  $b[a \ldots a + c - 1] \leftarrow \texttt{Split}_w(\texttt{ToString}(\mathrm{CSum}))$
  for $i \leftarrow 0 \ldots a + c - 1$ **do**
    $Z[i] \leftarrow \mathsf{SK}[i][b[i]]$
  return (Z)

function `Verify`$(R, Z, M, \texttt{KeyID})$
  $h = \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
  $b[0 \ldots a-1] \leftarrow \texttt{Split}_w(h)$
  $\mathrm{CSum} \leftarrow C - \sum_{i=0}^{a-1} b[i]$
  $b[a \ldots a + c - 1]$
    $\leftarrow \texttt{Split}_w(\texttt{ToString}(\mathrm{CSum}))$
  $Y' \leftarrow$ empty string
  for $i \leftarrow 0 \ldots a + c - 1$ **do**
    $u \leftarrow Z[i]$
    for $j \leftarrow b[i] + 1 \ldots (2^w - 1)$ **do**
      $u \leftarrow \texttt{hash}_{(2,i,j,\texttt{KeyID})}(u)$
    $Y' \leftarrow Y' \| u$
  $\mathsf{PK}' \leftarrow \texttt{hash}_{(0,\texttt{KeyID})}(Y')$
  return (PK')

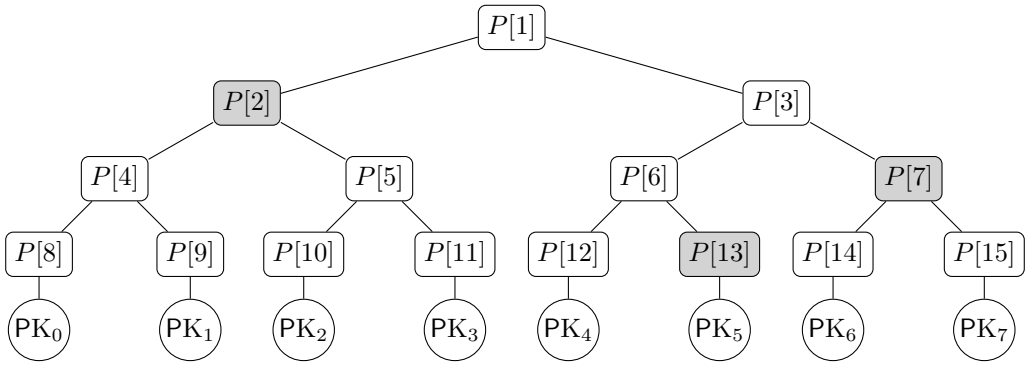

Figure 2: A Merkle tree with $D = 8$ leaves, where each leaf is a one-time public key PK_π with $0 \leq \pi < D - 1$. To verify, say, PK_4, one needs the index 4 and the hash values $P[13]$, $P[7]$, and $P[2]$ on the path from PK_4 to $P[1]$. The verifier needs to compute $P[12] = \mathtt{hash}_{(3,12)}(\mathsf{PK}_4)$, $P[6] = \mathtt{hash}_{(3,6)}(P[12], P[13])$, $P_3 = \mathtt{hash}_{(3,3)}(P[6], P[7])$, and the root $P[1] = \mathtt{hash}_{(3,1)}(P[2], P[3])$. Verification succeeds if $P[1]$ is the root in the composite public key. Specific details can vary for different stateful HBS schemes, such as, e.g., the tweaks for the hash function.

  - Because an adversary given $\mathsf{SK}[i][j]$ can always compute $\mathsf{SK}[i][j+1 \ldots 2^w - 1]$ by simply applying the formula above, we must also incorporate a *checksum* (CSum) on the w-bit chunks of the hash being signed, and encode that checksum in the hash chain. The effect of the checksum is to guarantee that the sum of all the w-bit chunks (hash and checksum) is constant. This guarantees that in order to alter the signature to make it sign a different hash by walking forward in one or more hash chains, the adversary must go backwards in at least one hash chain, which is prevented by the preimage-resistance property of the hash function.

- Verification

  - A verifier, given the revealed hash chain entries in the signature Z and the correct value for the message hash h, can compute the final values in each hash chain. The one-time public key is the hash of the concatenation of all these final values. This means that verifying a Winternitz signature produces a candidate one-time public key; if the signature is valid, the key will be correct.

Note that in our description, we represent the one-time private key as a two-dimensional array. The "rows" of this array consist of hash chains of length $2^w - 1$. In most implementations of hash-based signatures, the values in this array are generated "on the fly" rather than being precomputed, but this is an implementation choice that has no impact on the signature.

2.4 Adding a Merkle Tree: Stateful Hash-Based Signatures

One-time signatures are seldom useful by themselves. Instead, D one-time signing keys are combined in a single big public key (called a *composite public key* and written as `CPK` in the rest of this paper, to distinguish it from one-time public keys), to allow a signer to produce a practically useful number of signatures. The resulting schemes (as specified in standards like XMSS and LMS) are called *stateful hash-based signature* schemes, because the signer must keep track of which one-time keys have been used so far—the "state."

Stateful HBS are secure as long as the signer manages the state properly, but lose all security if the signer uses the same one-time key to sign two distinct messages. Managing the state correctly is the big practical challenge in implementing these schemes securely. As discussed above, one-time keys are identified individually with a `KeyID`. Without loss of generality, in this paper, we assume `KeyID` is an integer such that $\texttt{KeyID} \in \{0, \ldots, D-1\}$, where D is the number of one-time keys included in the public key `CPK`.

Merkle trees. A Merkle tree is constructed from a set of $D = 2^d$ leaves, which in our case, are the public one-time keys $\mathsf{PK}[0 \ldots D-1]$. When a particular leaf value in the array is considered, we indicate its index $\texttt{KeyID} \in \{0, \ldots, D-1\}$. A path in a Merkle tree depends on an index i and consists of those d internal values needed to verify $\mathsf{PK}[i] = X$. In the following, we briefly explain the functions needed in order to construct and use Merkle trees (See Algorithm 3).

`MT_Construct` takes a set of D leaf values ($\mathsf{PK}[0 \ldots D-1]$) and constructs a Merkle tree from them, returning said tree.

`MT_MakePath` takes($\mathsf{PK}[0 \ldots D-1]$) and an index `KeyID`, and then constructs a `PATH` from $\mathsf{PK}[\texttt{KeyID}]$ to the root, i.e., it computes those $d = \log_2(D)$ values, which are needed to verify that path.

`MT_Verify` follows the `PATH` from the claimed leaf, makes (some of) the hash function calls made by `MT_Construct` to compute the root and eventually computes that root. Verification succeeds if the root from `MT_Verify` is the same as the root from `MT_Construct`.

`MT_Extract` recovers the `KeyID` from `PATH`.

See Figure 2 for a Merkle tree with $D = 2^3$ leaves $\texttt{MT_Construct}(\mathsf{PK}[0 \ldots 7])$ returns the Merkle tree P with root $P[1]$. $\texttt{MT_MakePath}(\mathsf{PK}[0 \ldots 7], 4)$ returns `PATH`. `PATH` holds the values $P[13]$, $P[7]$, $P[2]$ and the index 4. $\texttt{MT_Extract}(\texttt{PATH})$ returns the index 4. If $\texttt{PATH} = \texttt{MT_MakePath}(\mathsf{PK}[0 \ldots 7], 4)$, then $\texttt{MT_Verify}(\texttt{PATH}, \text{leaf})$ computes $P[12]$ as the hash of leaf, hashes $P[12]$ and $P[13]$ into $P[6]$, $P[6]$ and $P[7]$ into $P[3]$, and $P[2]$ and $P[3]$ into the root $P[1]$, which it returns. Verification succeeds if the value returned by $\texttt{MT_Verify}(\texttt{PATH}, \text{leaf})$ is actually $P[1]$. If $\text{leaf} = P[4]$ then verification succeeds, and, under appropriate assumptions on the hash function, it is infeasible to find a value $\text{leaf} \neq P[4]$ for which verification succeeds.

Combining Winternitz signatures with Merkle trees. In modern stateful hash-based signature schemes, the signature $\sigma = (R, \texttt{PATH}, Z)$ is a triple composed of the randomized Winternitz signature (R, Z) and `PATH`. The public Winternitz key $\mathsf{PK}_{\texttt{KeyID}}$ needed to verify (R, Z) is the leaf with index `KeyID` in the Merkle-Tree. The `PATH` provides the information needed to move from that public key to the root of the Merkle tree. The composite public key `CPK` holds the root of the Merkle tree.

In order for the full stateful HBS scheme to be secure, the underlying Winternitz signature scheme must be secure, and there must be no way for an adversary to create a Merkle tree path that falsely associates a new one-time public key with the composite public key.

Formal Definition. A stateful HBS consists of three algorithms.

1. **Generate.** The dealer generates a new stateful HBS key consisting of D one-time signing keys.

$$\texttt{CPK}, \texttt{CSK} \leftarrow \texttt{StatefulGen}(D)$$

Algorithm 3 Merkle tree operations. Specific details can vary for different stateful HBS schemes, such as, e.g., the way how the index is embedded in the `PATH`, cf. Line 19. `MT_Construct` builds a Merkle tree from D public key values, while `MT_MakePath` creates a verification path for a specific `KeyID`. `MT_Verify` checks this path by recomputing the root, and `MT_Extract` retrieves the `KeyID` index from the path.

```
1:  function MT_Construct(PK[0 ... D − 1])
2:      for i = 0 to D − 1 do
3:          P[i + D] ← hash_tweak(PK[i])
4:      for i from D down to 1 do
5:          P[i] ← hash_tweak(P[2i], P[2i + 1])
6:      return P
7:  function MT_MakePath(PK[0 ... D − 1], KeyID)
8:      P ← MT_Construct(PK[0 ... D − 1])
9:      PATH ← [ ]
10:     i ← KeyID + D
11:     j ← 1
12:     while i > D do
13:         if i is even then
14:             PATH[j] ← P[i + 1]
15:         else
16:             PATH[j] ← P[i − 1]
17:         i ← ⌊i/2⌋
18:         j ← j + 1
19:     PATH[j] ← KeyID
20:     return PATH
21: function MT_Verify(PATH, leaf-value)
22:     KeyID ← MT_Extract(PATH)
23:     h ← hash_tweak(leaf − value)
24:     i ← KeyID + D
25:     j ← 1
26:     while i > 1 do
27:         i ← ⌊i/2⌋
28:         if i is even then
29:             h ← hash_tweak(h, PATH[j])
30:         else
31:             h ← hash_tweak(PATH[j], h)
32:         j ← j + 1
33:     ROOT ← h
34:     return ROOT
35: function MT_Extract(PATH)
    // Extract the the KeyID from the Merkle tree path.
36:     KeyID ← PATH[last_index]
37:     return KeyID
```

where CPK is the composite public key, and CSK is the corresponding private key. In principle, CSK could be an array of all D one-time signing keys, and we will represent it this way below, but in practice, the one-time keys are always generated on the fly using a PRF. For single-layer schemes like LMS and XMSS, StatefulGen constructs the full Merkle tree, and so must generate all one-time signing keys that will ever be used in the scheme.

2. **Sign a message** The signer produces a randomized signature on message M:

$$R, \texttt{PATH}, Z = \texttt{StatefulSign}_{\texttt{CSK}}(\texttt{KeyID}, M)$$

Note that StatefulSign is a *stateful* algorithm. Thus, the signer must keep track of which one-time keys have been used and never allow the use of the same one-time key (and thus the same KeyID) twice. The signature contains a Merkle tree path that relates CPK and the one-time signing key used for the signature. Here, we assume that StatefulSign will return an error if asked to sign twice with the same KeyID. (In most real-world implementations of stateful hash-based signatures, the signer keeps track of the most recently used KeyID and always increments it before signing.)

3. **Verify a signature.** The verifier checks the validity of the signature, using the composite public key, message, and signature.

$$\texttt{StatefulVerify}(\texttt{CPK}, R, \texttt{PATH}, Z) = \text{True or False}$$

Verification consists of both verifying the randomized one-time Wintetnitz signature (yielding a one-time public key), and verifying that this one-time public key is associated with the composite public key by checking PATH.

2.5 The Way Forward

In the rest of this paper, we can ignore most of the complexity of stateful hash-based signatures. Once a one-time signing key is represented as a two-dimensional array holding a set of hash chains, our techniques work without reference to how the values in the array were generated. Once the dealer knows how to construct a Merkle tree path for each one-time key, again, our techniques can operate on those PATHs without reference to the specific way they were constructed.

3 k-of-k One-Time Signatures

A k-of-k signature scheme involves splitting the signing key into k shares, and giving one share to each of k trustees; signing a message requires all k trustees to agree and participate in creating the signature. In this section, we define a k-of-k one-time randomized signature scheme, based on the one-time hash-based signature $W(\texttt{tweak})$, described above.

3.1 Overview

In modern stateful hash-based signatures, the signature is always carried out on a randomized hash of the message. Each individual one-time key in the stateful HBS has a unique identifier, which we will call the KeyID. The final signature consists of a triple $\sigma = (R, \texttt{PATH}, Z)$, where R is the randomizer, PATH is the Merkle tree path for the one-time key used, and Z is the Winternitz one-time signature, computed over the hash $h = \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$. Our scheme constructs one-time signatures that follow this same format: they incorporate a randomizer R, and carry a Merkle tree path, PATH, necessary

Algorithm 4 Generic description of a stateful hash-based signature scheme, operating on the Merkle tree, see Algorithm 3. `StatefulGen` generates key pair, $(\texttt{CPK}, \texttt{CSK})$ to sign D messages under D unique `KeyID`s. The public key `CPK` holds, in addition to some optional metadata, the root `CPK.ROOT` of the Merkle tree. `StatefulSign` depends on the secret key `CSK` and signs a message M under a given `KeyID`. `StatefulVerify` performs the matching verification process. It returns 1 if and only if verification succeeds.
Note that maintaining the state is only neccessary to perform the check in Line 8. I.e., the functions `StatefulGen` and `StatefulVerify` are stateless, even though they are components of a stateful signature scheme.

1: **function** `StatefulGen`(D)
2: **for** `KeyID` $\leftarrow 0 \dots D-1$ **do**
3: $\mathsf{PK}[i], \mathsf{SK}[i] \leftarrow \mathrm{Gen}(\texttt{KeyID})$
4: $\texttt{CPK.ROOT} \leftarrow \texttt{MT_Construct}(PK[0 \dots D-1])$
5: $\texttt{CSK} \leftarrow \mathsf{SK}[0 \dots D-1]$
6: **return** $(\texttt{CPK}, \texttt{CSK})$

7: **function** $\texttt{StatefulSign}_{\texttt{CSK}}(\texttt{KeyID}, M)$
8: **if** `KeyID` has been used before **then**
9: **return** a failure code
10: **else**
11: Mark this `KeyID` as used.
12: $R \leftarrow\!\!\$\ \{0,1\}^n$
13: $h \leftarrow \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
14: $Z \leftarrow \texttt{WINTER}(h, \texttt{CSK}[\texttt{KeyID}])$
// $PK[0 \dots D-1]$ *can be cached or recomputed from* `CSK`.
15: $\texttt{PATH} \leftarrow \texttt{MT_MakePath}(PK[0 \dots D-1], \texttt{KeyID})$
16: **return** $(R, \texttt{PATH}, Z)$

17: **function** $\texttt{StatefulVerify}_{\texttt{CPK}}(M, R, \texttt{PATH}, Z)$
18: $h = \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
19: $PK' \leftarrow \texttt{Verify}(R, Z, M, \texttt{KeyID})$
20: **if** $\texttt{MT_Verify}(\texttt{PATH}, PK') = \texttt{CPK.ROOT}$ **then**
21: **return** (1)
22: **else**
23: **return** (0)

when using the one-time signature as part of a full stateful hash-based signature scheme. The unique `KeyID` of the one-time key is likewise incorporated into the message hash.

Our scheme requires a trusted *dealer* for setup. The dealer can be thought of as the owner of the one-time signing key, delegating its ability to sign to a set of trustees. Our scheme also requires a common reference string with a particular format: For each composite public key `CPK`, each one-time signing key has a public bitstring associated with it. In this section, we write `CRV` for the common reference values associated with a given one-time signing key. The `CRV` consists of several fields: $\texttt{CRV}.R, \texttt{CRV}.\textsc{chk}, CRV.\texttt{PATH}$, and $\texttt{CRV}.\mathsf{SK}$. $\texttt{CRV}.\textsc{chk}$ and $\texttt{CRV}.\mathsf{SK}$ are themselves arrays of bitstrings. Later, when more than one `KeyID` is in use, we will write `CRV`[`KeyID`] for the `CRV` associated with one-time key `KeyID`.

The dealer creates the `CRV` during setup. For typical choices of Winternitz parameters and security strength, the `CRV` for a single one-time signing key is around 9 KiB.

Signing in our scheme assumes an untrusted party called an *aggregator*, which interacts with each trustee and reads the `CRV`, but is not entrusted with any secrets and has no power except the ability to ask trustees to participate in signing messages. Only the aggregator ever accesses the `CRV`. Neither the trustees nor the verifier ever need to look at the `CRV`. There is nothing preventing an implementation of our scheme from having one of the trustees also act as the aggregator, and doing so will not affect the security of the scheme. All communications go between the aggregator and individual trustees; there is no broadcast channel assumed.

3.2 Pseudorandom Function (PRFs)

Our scheme for making k-of-k one-time hash based signatures requires the ability for trustees to generate a number of values on the fly, using a PRF. In the rest of the paper, we assume a PRF algorithm which incorporates both explicit inputs and also a label that domain-separates different uses of the PRF, and which provides a requested number of bits of output. For readability, the label is put as a superscript to the PRF call, while the key is a subscript.

$$\mathrm{PRF}_{\mathrm{key}}^{\mathrm{label}}(\text{input, length})$$

represents a call to the PRF for a purpose described by *label*, the PRF key *key*, called over a particular *input* and generating *length* bits of output. For convenience, we define the following five PRF labels:

- $\mathrm{PRF}^{\mathrm{R}}$ for generating a trustee's share of the randomizer.
- $\mathrm{PRF}^{\textsc{Chain}}$ for generating each component of a trustee's share of the one-time signing key.
- $\mathrm{PRF}^{\textsc{Chk}}$ for generating a trustee's share of the CHK value (used to ensure the randomizer is correct).
- $\mathrm{PRF}^{\texttt{PATH}}$ for generating a trustee's share of the Merkle tree path for this one-time key.
- $\mathrm{PRF}^{\textsc{Auth}}$ for generating the value that a trustee uses to verify the correctness of the randomizer.

However, we emphasize that these are just calls to the same PRF incorporating different label values into the PRF input. Our scheme is agnostic as to how this PRF is constructed, though a natural choice might be KMAC [KCP16], using the customization string to incorporate the different labels. We only require that the label and input be

encoded in an unambiguous way before being fed into the PRF. As before, we assume that an *input* consisting of a sequence of comma separated values is encoded in an unambiguous way. The security assumptions on the PRF are discussed in Section 5.2, below.

3.3 Formal Definition of k-of-k One-Time Signatures

Definition 1 (k-of-k one-time signatures)**.** Formally, our scheme for k-of-k one-time signatures consists of the following algorithms:

1. **Setup.** The trusted dealer is given k PRF keys, $K[1 \dots k]$, a one-time signing key SK, a randomizer R, a `PATH`, a unique `KeyID` and computes

$$\texttt{CRV} \leftarrow \texttt{KK_Setup}(K[1 \dots k], k, \texttt{KeyID}, \mathsf{SK}, R, \texttt{PATH})$$

   where `CRV` is the common reference string for this one-time signature, consisting of several fields. Note that the trustees' shares will all be computed on-the-fly using K and `KeyID`, so the only output of the setup algorithm is the `CRV`.

2. **Sign a message** The untrusted aggregator runs a protocol with all k trustees to sign a message M. This is shown below in the algorithms used by the aggregator (`KK_Aggregator_Sign`) and the trustees (`KK_Sign1` and `KK_Sign2`).

3. **Verify a signature.** The verifier checks the validity of the signature:

$$\texttt{StatefulVerify}_{\texttt{CPK}}(M, R, \texttt{PATH}, Z) \in \{0, 1\}$$

   This is the application of the verification function from Algorithm 4.[5]

At a high level, our technique for k-of-k one-time signatures is quite straightforward: We split each secret value needed to sign with the one-time key into $k + 1$ XOR shares, so that k shares are held by the trustees, and one is stored in the common reference string. The trustees' shares are derived from a PRF to decrease the resource requirements for trustees; the `CRV` values cannot be computed on the fly, but instead must be stored.

Randomized signatures pose a challenge for a k-of-k signature scheme. The randomizer R must not be known by any subset of $k - 1$ of the k trustees until all k trustees have seen the message to be signed. We generate the randomizer for the key at setup time, and likewise compute an authentication value CHK, which allows each trustee to verify that the randomizer is correct.

To sign a message M, the aggregator sends M to all k trustees, and each trustee sends back shares of the randomizer and check value. Using the `CRV`, the aggregator can reconstruct the correct randomizer and check value, and send those values to all k trustees. Then, each trustee computes the randomized hash of the message, and signs the hash with its share of the one-time key. Each trustee sends back its share of the Merkle-tree path and the one-time signature. The aggregator, using the trustees' shares and the `CRV`, is able to construct the `PATH` and one-time signature Z. It can thus output the final signature: $(R, \texttt{PATH}, Z)$.

3.4 Setup

In the following, we will denote Y^t as trustee t's share of Y. The trusted dealer is given k PRF keys, $K[1 \dots k]$, a one-time signing key SK, a randomizer R, a path `PATH`, a unique

[5] In fact, consider the `CPK`, the `CRV`, the message M, and a matching signature, either generated by the dealer himself/herself running the signing function from Algorithm 4, or by k trustees and an aggregator running the k-of-k signing protocol. Then the distribution of the triple $(R, \texttt{PATH}, Z)$ is exactly the same, regardless of the signature generated by the dealer or by the trustees and the aggregator.

Algorithm 5 `KK_Setup`: The trusted dealer splits a one-time signing key into k trustee shares, plus a `CRV` share.

function $\texttt{KK_Setup}(K[1\ldots k], k, \texttt{KeyID}, \mathsf{SK}, R, \texttt{PATH})$
  // Generate check value for the randomizer.
  for $t \leftarrow 1\ldots k$ **do**
    $\textsc{chk}[t] \leftarrow \mathrm{PRF}^{\textsc{Auth}}_{K[t]}(\texttt{KeyID}, R, n)$ // Key-dependent check value for R
  // We can now construct the trustees' shares using the PRF.
  for $t \leftarrow 1\ldots k$ **do**
    $R^t = \mathrm{PRF}^{\mathrm{R}}_{K[t]}(\texttt{KeyID}, n)$
    $\textsc{chk}^t \leftarrow \mathrm{PRF}^{\textsc{Chk}}_{K[t]}(\texttt{KeyID}, |\textsc{chk}|)$
    $\texttt{PATH}^t \leftarrow \mathrm{PRF}^{\texttt{PATH}}_{K[t]}(\texttt{KeyID}, |\texttt{PATH}|)$
    for all i, j in SK **do**
      $\mathsf{SK}^t[i][j] \leftarrow \mathrm{PRF}^{\textsc{Chain}}_{K[t]}(\texttt{KeyID}, i, j, n)$
  // Construct the `CRV` *shares*
  $\texttt{CRV}.R \leftarrow R \oplus \bigoplus_{s=1}^{k} R^s$
  $\texttt{CRV}.\textsc{chk} \leftarrow \textsc{chk} \oplus \bigoplus_{s=1}^{k} \textsc{chk}^s$
  $\texttt{CRV}.\texttt{PATH} \leftarrow \texttt{PATH} \oplus \bigoplus_{s=1}^{k} \texttt{PATH}^s$
  for all i, j in SK **do**
    $\texttt{CRV}.\mathsf{SK}[i][j] \leftarrow X[i][j] \oplus \bigoplus_{s=1}^{k} \mathsf{SK}^t[i][j]$
  return `CRV`

Algorithm 6 The k-of-k signing process as carried out by the aggregator. The aggregator calls functions `KK_Sign1` and `KK_Sign2` from Algorithm 7, which are evaluated by the trustees. `WINTER` computes a deterministic Winternitz signature for h, cf. Algorithm 2.

function $\texttt{KK_Aggregator_Sign}(M, \texttt{CRV}, \texttt{KeyID})$
  for $t \in \texttt{CL}[\texttt{KeyID}]$ **do**
    $R^t, \textsc{chk}^t \leftarrow \texttt{KK_Sign1}(\texttt{KeyID}, M)$ *// First Round*
  $R \leftarrow \texttt{CRV}.R \oplus \bigoplus_{s=1}^{k} R^s$
  $\textsc{chk} \leftarrow \texttt{CRV}.\mathrm{Auth} \oplus \bigoplus_{s=1}^{k} \mathrm{Auth}^s$
  $i \leftarrow 0$
  for $t \in \texttt{CL}[\texttt{KeyID}]$ **do**
    $\texttt{PATH}^t, Z^t \leftarrow \texttt{KK_Sign2}(t, R, \textsc{chk}[i])$ *// Second Round*
    $i \leftarrow i + 1$
  $h = \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
  $Z^{\texttt{CRV}} \leftarrow \texttt{WINTER}(h, \texttt{CRV}.\mathsf{SK})$
  $Z \leftarrow Z^{\texttt{CRV}} \oplus \bigoplus_{s=1}^{k} Z^s$
  $\texttt{PATH} \leftarrow \texttt{CRV}.\texttt{PATH} \oplus \bigoplus_{s=1}^{k} \texttt{PATH}^s$
  return $R, \texttt{PATH}, Z$

Algorithm 7 The k-of-k signing process as carried out by the aggregator and the trustees. Every trustee t knows $\texttt{KeyID}, K[t]$, and its state is stored in $\texttt{current}[t]$, which starts as `None`. Functions `KK_Sign1` and `KK_Sign2` handle the first and second round of the signing protocol, `KK_Auth` checks if the input to the second round is OK (implying that all k trustees did perform their first-round operation), and `KK_GenSig1` and `KK_GenSig2` actually perform the first and second steps, resp., of the signing process. `WINTER` computes a deterministic Winternitz signature for h, cf. Algorithm 2.

function `KK_Sign1`$(t, \texttt{KeyID}, M)$
  if `KeyID` not used before **then**
    $\texttt{current}[t] \leftarrow M$
    return `KK_GenSig1`$(K[t], \texttt{KeyID})$
  else
    return $\perp$

function `KK_GenSig1`$(K[t], \texttt{KeyID})$
  $R^t = \mathrm{PRF}^{\mathrm{R}}_{K[t]}(\texttt{KeyID}, n)$
  $\mathrm{CHK}^t \leftarrow \mathrm{PRF}^{\mathrm{CHK}}_{K[t]}(\texttt{KeyID}, |\mathrm{CHK}|)$
  return (R^t, CHK^t)

function `KK_Sign2`$(t, R', \mathrm{CHK}'[t])$
  if $\texttt{current}[t] = \texttt{None}$ **then**
    return $\perp$
  else
    $M \leftarrow \texttt{current}[t]$
    $\texttt{current}[t] \leftarrow$None
    if `KK_Auth`$(K[t], \texttt{KeyID}, R', \mathrm{CHK}')$ **then**
      $h \leftarrow \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
      return `KK_GenSig2`$(K[t], \texttt{KeyID}, h)$
    else
      return $\perp$

function `KK_Auth`$(K[t], \texttt{KeyID}, R', \mathrm{CHK}'[t])$
  if $\mathrm{PRF}^{\mathrm{AUTH}}_{K[t]}(\texttt{KeyID}, R') = \mathrm{CHK}'[t]$ **then**
    return 1
  else
    return 0

function `KK_GenSig2`$(K[t], \texttt{KeyID}, h)$
  for $i \leftarrow 0 \ldots a-1$ **do**
    for $j \leftarrow 0 \ldots 2^w - 1$ **do**
      $\mathsf{SK}^t[i][j] \leftarrow \mathrm{PRF}^{\mathrm{CHAIN}}_{K[t]}(\texttt{KeyID}, i, j, n)$
  $Z^t \leftarrow \texttt{WINTER}(h, \mathsf{SK}^t)$
  $\texttt{PATH}^t \leftarrow \mathrm{PRF}^{\texttt{PATH}}_{K[t]}(\texttt{KeyID}, |\texttt{PATH}|)$
  return $(\texttt{PATH}^t, Z^t)$

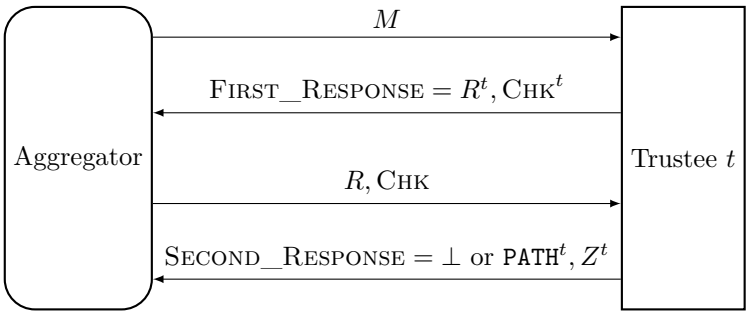

Figure 3: The protocol between the aggregator and each trustee. The aggregator combines the trustees' responses along with the contents of the CRV to construct (R, CHK), and later to construct PATH, Z, allowing it to learn the full signature $\sigma = R, \texttt{PATH}, Z$.

KeyID and computes

$$\texttt{CRV} \leftarrow \texttt{KK_Setup}(K[1 \ldots k], SK, R, \texttt{PATH}, \texttt{KeyID})$$

where CRV is the common reference string for this one-time signature.

Randomized signatures require a randomizer that is unpredictable to anyone before signing, and that cannot be controlled by anyone. We achieve these goals by having the dealer choose a randomizer and pass it into the KK_Setup algorithm, and having KK_Setup to produce a value (CHK$[1 \ldots k]$) allowing the trustees to verify R.

The algorithm uses $K[1 \ldots k]$ and KeyID to split the values $SK, R,$ PATH into shares, so that the k trustees' shares plus the CRV shares together can reconstruct them.

At the end of this process, the dealer gives each trustee its key $K[t]$, outputs the CRV, and deletes all the internal values from the calculation, as well as the one-time signing key SK. The algorithm is shown in Algorithm 5.

3.5 Signing Protocol

The k-of-k signing protocol takes place between the aggregator and each of the k trustees individually. The interaction between the aggregator and trustee t is shown in Figure 3. Pseudocode showing how the aggregator and trustees operate during the protocol appears in Algorithm 6 and Algorithm 7. In this protocol, the aggregator has no secret knowledge or power, but does have access to the CRV; each trustee t has access to its PRF key $K[t]$. The signing protocol has two round-trips–the first establishes the randomizer, the second constructs the signature. This is analogous to some threshold versions of Schnorr (for example, [GKMN21]), where there is a pre-signing interaction and a signing interaction. However, because of the way we rely on the randomizer for security, in our protocol, the message to be signed must be seen by all trustees before the randomizer is established.

The ability to turn a one-time hash-based signature into a k-of-k one-time signature provides a building block for constructing distributed signatures.

4 Sharding: from k-of-k to Distributed Signatures

A k-of-k signature scheme provides a security benefit, since now an adversary must compromise all k trustees to learn the private signing key. However, it makes availability worse: a single unavailable trustee prevents signing. The benefit of threshold and distributed signatures is that they provide improvements in both security and availability. Here, we consider how to achieve these benefits for stateful hash-based signatures. Achieving these

benefits is made more difficult by the fact that in a stateful HBS scheme, the signer(s) must ensure that no one-time signing key is ever reused.

4.1 The Disjoint Honest Trustees Problem

When all k trustees must work together to sign a message, all trustees must agree to use a one-time key. One single honest trustee is sufficient to avoid reusing a key. However, when k of ℓ trustees suffice to sign a message with a one-time key, $\ell - k$ trustees will not even know that the key has been used.

Consider a 3-of-5 threshold signature with trustees A, B, C, D, and E. If C is corrupt, it can sign a message jointly with A and B, and another message jointly with D and E, *employing the same secret one-time key* for both signatures. After seeing both signatures, C can learn enough of the one-time signing key to sign many additional messages with that key. This general attack works for *all* k-of-ℓ schemes, whenever $k < \ell$: just $2k - \ell$ dishonest trustees can force the reuse of a one-time key. Indeed, when $k \leq \ell/2$, even a set of entirely *honest* trustees can accidentally reuse a one-time key.

4.2 Sharding

The above attack works because of the absence of some kind of broadcast channel between the trustees. If the signature scheme is not of the k-of-k type, a trustee cannot know what one-time keys have been used by other trustees. Requiring a broadcast channel between trustees would impose an unworkable level of overhead for many applications.

A practical solution is *sharding*: The set of all one-time keys belonging to the user's composite public key is divided into *shards*, with each shard being a set of one-time keys owned by a single signing coalition. The one-time keys in the shard owned by the (A, B, C) signing coalition can only be signed by A, B, and C working together.

For example, consider our 3-of-5 threshold signature scheme. There are $\binom{5}{3} = 10$ signing coalitions. In order to get a 3-of-5 threshold signature scheme, we simply assign $\frac{1}{10}$ of the keys to each coalition. Keys owned by the coalition for trustees A, B, and C are the keys of our 3-of-3 signature scheme, with trustees A, B, and C each able to derive shares with their PRF keys. In this way, we use the k-of-k one-time signature scheme above to construct threshold signatures. The fact that for each one-time key there is only one signing coalition solves the Disjoint Honest Trustees problem.

Consider the full set of one-time keys associated with the composite public key. Each one-time key is owned by a single coalition. In our 3-of-5 case, if the total number of one-time keys is D, each of the ten coalitions (coalitition ABC, ABD, etc.) controls about $\frac{D}{10}$ of the one-time keys, and, whenever a coalition is about to sign a message, they have to pick one of those $\frac{D}{10}$ keys (an unused one, of course). For example,, trustees A and B do not participate in any way in a signature with a one-time key that belongs to the CDE coalition, and have no information about that one-time signing key.

Mechanics of Sharding. During setup, the dealer starts with a composite public key and knowledge of the corresponding D one-time signing keys. Each one-time key is associated with a unique integer called a `KeyID`. We will assume in the rest of this paper that `KeyID` $\in \{0 \ldots D-1\}$. The dealer starts with the set of allowed signing coalitions and the number of keys each coalition will own. This information can be encoded in the *coalition list*, $CL[0 \ldots D-1]$. CL[`KeyID`] is a list of the trustees in the signing coalition for this `KeyID`.

Along with distributing PRF keys and the composite public key to the trustees, the dealer gives each trustee the description of its shards—for each coalition to which it belongs, a trustee must know the range of `KeyID`s associated with that coalition. Each trustee is responsible for knowing the next available `KeyID` in each of its shards, as well as how many

KeyIDs are in each shard. All this is determined by the contents of CL, and can be sent to each trustee[6].

The signing protocol now includes one more step—the aggregator must determine which coalition of trustees will do the signing. Once a coalition has been selected, the aggregator uses the k-of-k one-time signing protocol described above, and no longer involves the rest of the trustees. That is, if the aggregator decides to ask the coalition of trustees A, B, and C to sign the message, it simply runs the 3-of-3 one-time signing protocol with trustees A, B, and C; it sends no messages to D or E.

Determining which trustees will be included in the signature is outside the scope of the protocol. Many practical applications of a threshold signature scheme will already have done this step before signing (for example, when several human trustees must show up in person and use their smartcards to authorize some signature), but even in the online setting, this simply adds one more interaction between the aggregator and the trustees, in which the aggregator polls the trustees to find out which ones are available to sign a message.

Just as with the k-of-k one-time signatures described above, verification is exactly the same as the verification process of the underlying stateful HBS. The verifier has no need to know what coalitions are allowed to sign, and there is no difference in the signature produced when the signature is produced by a coalition of trustees or when it is produced by a single signer holding the signing key. *In fact, the verifier cannot even distinguish a distributed signature, jointly generated by a coalition of trustees, from a conventional stateful HBS signature, generated by a single signer.*

Sharding vs Multisigs. A trivial way to get threshold or distributed signatures is by aggregation. To get a k-of-ℓ threshold scheme, we simply have ℓ trustees each with its own public key, and form signatures by getting any k of them to sign a message and concatenating the signatures together. However, this has a lot of downsides: the verifier must use custom code to verify, the verifier must know what signing coalitions are allowed, and the signatures (at least if done in a naive way) and public keys are $O(k)$ times as large as in the normal version of the signature.

Sharding is quite different: the verifier uses the same code for our signatures as for normal stateful HBS, and does not need to know anything about what signing coalitions are allowed. The signatures and public keys look the same as in non-threshold versions of the signature. However, sharding does impose some limitations on our choice of coalitions, because we have a fixed number of total one-time signatures and each one must be allocated to a single signing coalition. Further, the aggregator must know the allowable signing coalitions and which KeyID belongs to which coalition.

Limitations. As pointed out above, our hash-based signatures provdide a fixed number of D KeyIDs, and sharding means we have to split these KeyIDs between different coalitions. Compared to other distributed and threshold signature schemes, this limits the number of coalitions. If each coalition must be able to sign at least m messages, there can be no more than $\lfloor D/m \rfloor$ coalitions. See Section 6.2 for a deeper discussion.

4.3 Constructing a Distributed Signature Scheme

Requirements on The Underlying Stateful HBS. In order for our algorithm to work, we require several conditions on the stateful hash-based signature scheme on which it is based. (We note that all these are met for both LMS and XMSS, and are consistent

[6] At first glance, CL might seem like a lot of information to share with all the trustees. However, by making each signing coalition's shard a contiguous block of KeyIDs, CL can be encoded very compactly, and each trustee can store the information it needs from CL even more compactly, needing only two KeyIDs for each of the trustee's signing coalition.

with the requirements in NIST SP 800-208.) Let $\texttt{CPK}, \texttt{CSK}$ be the public and private keys for the full stateful HBS, generated with the call

$$\texttt{CPK}, \texttt{CSK} \leftarrow \texttt{StatefulGen}(D)$$

where D is the number of one-time signatures to be included in the composite public and private keys. Typically, $D = 2^d$ will be a power of two. The conditions are the following:

1. The full signature can be parsed as a triple of randomizer, Merkle tree path, and one-time signature, written in the rest of this paper as

$$(R, \texttt{PATH}, Z).$$

2. The randomizer R can be chosen at setup time, rather than at signing time. (Although the randomizer R is usually chosen at signing time, choosing R at setup time is possible for both LMS and XMSS.)

3. Each one-time key has a unique identifier, called `KeyID` in the rest of this paper. For simplicity, we will assume $\texttt{KeyID} \in \{0 \dots D-1\}$.

4. We require the underlying HBS to be secure against forgery attacks. We will formalize this in form of a security game below. Both LMS and XMSS have been proven secure against such attacks, based on standard assumptions on the underlying hash function, or in the random oracle model.

5. An additional requirement is on the underlying Winternitz one-time signature scheme. Both LMS and XMSS generate D one-time key pairs $(\mathsf{PK}_i, \mathsf{SK}_i)$. A key pair $(\mathsf{PK}_j, \mathsf{SK}_j)$ must preserve security against forgery attack, even if all other key pairs $(\mathsf{PK}_i, \mathsf{SK}_i)$ with $i \neq j$ have been compromised. Under the same assumptions, both LMS and XMSS satisfy this criterion.

The security of our scheme also seems to depend on Merkle trees as vector commitment schemes:

- Initially, the signer/dealer commits to D public one-time keys $\mathsf{PK}_0, \dots, \mathsf{PK}_{D-1}$, computing the composed public key `CPK` as a commitment string.
- Subsequently, the signer proves its commitment to PK_i by providing $\texttt{PATH}_i$.
- Merkle-trees must be binding, i.e., it must be infeasible to find j, $\texttt{PATH}_j$ and $\mathsf{PK}'_j \neq \mathsf{PK}_j$ such that one can prove a commitment for PK'_j.

It is possible (and even easy, at least in the random oracle model) to prove that the Merkle-trees in LMS and XMSS are binding in the above sense. In the current paper, we do not need such proof or assumption. For the security of our scheme, Item 4 and Item 5 suffice.

Our distributed signature scheme uses the one-time hash-based signatures described above. We thus make use of the function

$$\texttt{CRV} \leftarrow \texttt{KK_Setup}(K[1 \dots k], k, \texttt{KeyID}, X, \texttt{PATH})$$

This function produces the `CRV` shares for a given one-time key, given the PRF keys, `KeyID`, the one time private key X (in the form of a two-dimensional array, as discussed above), the randomizer R, and the Merkle tree path `PATH` of this one-time key. Note that each entry in `CRV` has the fields $X, \texttt{PATH}, R, \textsc{chk}$.

Algorithm 8 `ShardSetup`: The trusted dealer sets up a threshold or distributed signature scheme from a stateful HBS. It calls `StatefulGen` from Algorithm 4 and then splits the secret key in a way that one needs the trustees' secret keys $K[1], \ldots K[k]$ and the public `CRV` to restore the secret key.

```
1:  function ShardSetup(D, k, CL)
2:      Let CRV[0 ... D − 1] be an array of D values
3:      for i ← 1 ... k do
4:          K[i] ←$ {0,1}^n
5:      CPK, CSK ← StatefulGen(D)
6:      for KeyID ← 0 ... D − 1 do
7:          R ←$ {0,1}^n
        // PK[0 ... D − 1] is computed in StatefulGen
        // and can be recomputed from CSK.
8:          PATH = MT_MakePath(PK[0 ... D − 1], KeyID)
9:          SK ← CSK[KeyID]
10:         k' ← |CL[KeyID]|                    // Number of keys in the coalition
11:         keys ← MakeKeyList(K, CL[KeyID])
12:         CRV[KeyID] ← KK_Setup(keys, k', KeyID, SK, R, PATH)
13:     return (CPK, CRV, K[1 ... k])
```

Signing Coalitions and the Coalition List. Our distributed signature scheme allocates each one-time signing key to a particular *signing coalition*. A signing coalition C is a nonempty subset $C \subseteq \mathcal{T} = \{1, \ldots, k\}$ of the set $\mathcal{T}$ of all trustees.

In order to describe the set of signing coalitions, we use a *coalition list*, written as `CL` in the rest of this paper. `CL` is an array, indexed by `KeyID`, where each entry in the array is a signing coalition. For example, if `KeyID` $= 1$ has a a signing coalition consisting of trustees 1, 2, and 3, we would write `CL`$[1] = \{1, 2, 3\}$. The coalition list gives a straightforward way to define everything important about our scheme—the number of trustees, the number of one-time keys, the set of signing coalitions, and how many signatures each coalition is permitted.

Setting up the Distributed Signature Scheme. Our scheme requires a trusted dealer. The setup of our scheme is done as follows:

$$K[1 \ldots k], \texttt{CRV} \leftarrow \texttt{ShardSetup}(\texttt{CL})$$

As discussed above, the coalition list contains all the information needed for the setup—the number of trustees, number of one-time keys, and signing coalitions are all defined. Once the dealer has run `ShardSetup`, distributed the PRF keys $K[1..k]$ and coalition list `CL` to the trustees, and made the `CRV` and `CL` available to anyone who might play the role of the aggregator in the protocol, it deletes all memory of the secret values used. The dealer can in some sense be seen as the owner of the stateful HBS key, who is delegating the power to sign to a set of trustees. The setup is shown in Algorithm 8. When each trustee receives its PRF key, it must initialize its internal state. The algorithm for this is the `TrusteeSetup` function, and appears in Algorithm 11.

Signing Signing a message with one-time key `KeyID` involves a protocol running between the *aggregator* and each of the trustees in `CL`[`KeyID`].

For the aggregator, the signing protocol involves selecting a coalition of trustees and

Algorithm 9 `MakeKeyList`: Construct an array containing the PRF keys of each trustee in the coalition C.

function `MakeKeyList`(K, C)
  Let keylist be an array of $|C|$ elements.
  for $i \leftarrow 0 \ldots |C| - 1$ **do**
    keylist[i] $\leftarrow$ K[C[i]]
  return keylist

Algorithm 10 `AggregatorSign`: The aggregator takes a message M, the public `CRV` and the `KeyID` and then interacts with trustees to generate a signature for M. The behaviour of the trustees is defined in Algorithm 11. If any response from any trustee is $\perp$, the aggregator stops the signing process and returns $\perp$ itself.

function `AggregatorSign`$(M, \texttt{CRV}, \texttt{KeyID})$
  Determine coalition C of trustees and `KeyID` with $\texttt{CL}[\texttt{KeyID}] = C$
  for each $t \in C$ **do**
    $R^t, \text{CHK}^t \leftarrow \texttt{ShardSign1}(t, \texttt{KeyID}, M)$
  $R \leftarrow \texttt{CRV}[\texttt{KeyID}].R \oplus \bigoplus_{t \in CL[\texttt{KeyID}]} R^t$
  $\text{CHK} \leftarrow \texttt{CRV}[\texttt{KeyID}].\text{CHK} \oplus \bigoplus_{t \in CL[\texttt{KeyID}]} \text{CHK}^t$
  $i \leftarrow 0$
  for each $t \in \mathcal{C}$ in sequential order **do**
    $\texttt{PATH}^t, Z^t \leftarrow \texttt{ShardSign2}(t, R, \text{CHK}[i])$
    $i \leftarrow i + 1$
  $h = \texttt{hash}_{(1,\texttt{KeyID})}(R, M)$
  $Z^{\texttt{CRV}} \leftarrow \texttt{WINTER}(h, \texttt{CRV}[\texttt{KeyID}].\text{SK})$
  $\texttt{PATH} \leftarrow \texttt{CRV.PATH} \oplus \bigoplus_{t \in CL[\texttt{KeyID}]} \texttt{PATH}^t$
  $Z \leftarrow Z^{\texttt{CRV}} \oplus \bigoplus_{t \in CL[\texttt{KeyID}]} Z^t$
  Return $\texttt{PATH}, R, Z$

a `KeyID`[7], interacting with the trustees, and ending up with information that can be combined with the `CRV` to get a valid signature. The aggregator's view of the signing protocol appears in Algorithm 10; the trustees' view appears in Algorithm 11

Each trustee communicates only with the aggregator, and must decide how to respond to requests without interacting with the other trustees. Trustee t knows its PRF key $K[t]$ and also keeps track of two pieces of information: the list of one-time keys it is permitted to use, and the current state of the signing protocol. Both of these evolve as the trustee receives messages from the aggregator.

Verification. Verification works exactly as with the stateful HBS algorithm.

5 Security

5.1 Notation and Conventions

In the rest of this paper, we will represent our security games in terms of function calls: the adversary in the game is able to make a set of function calls, which the challenger must

[7] A full implementation will need to include a mechanism for re-establishing synchronization between trustees on which `KeyID` a given signing coalition is ready to use, and determining which trustees are available. It is straightforward to poll each trustee on its availability and which `KeyID` it will sign with next for each of its coalitions, but we omit these details here to concentrate on the cryptographic aspects of our scheme.

Algorithm 11 Signing with the distributed HBS: trustee view. Each trustee t maintains an internal state, which holds trustee's PRF key $K[t]$, the list `keylist` of `KeyID`s the trustee may still use, and the trustee's state variable `current`. When the trustee commits to participate in signing a message M under a key indexed by `KeyID`, the variable `current` holds the pair $(\texttt{KeyID}, M)$. Otherwise, `current` holds None.

```
    // Setting up trustee t for the distributed signing protocol.
 1: function TrusteeSetup(CL, K, k)
 2:     Let keylist[1 ... k] be an array of sets of KeyIDs.
 3:     Let current[1 ... k] be an array of tuples.
 4:     for t ← 1 ... k do
 5:         Ensure trustee t knows K[t].
 6:         keylist[t] ← {}
 7:         for KeyID ← 0 ... D − 1 do
 8:             if t ∈ CL[KeyID] then
 9:                 Add KeyID to keylist[t]
10:         current[t] ← None
```

```
    // Distributed signature protocol on trustee side: first message from trustee t
11: function ShardSign1(t, KeyID, M)
12:     if KeyID is not in keylist[t] then
13:         return ⊥
14:     else
15:         Remove KeyID from keylist[t]
16:         current[t] ← (KeyID, M)
17:         return KK_GenSig1(K[t], KeyID)
```

```
    // Distributed signature protocol on trustee side: second message from trustee t
18: function ShardSign2(t, R′, CHK′[t])
19:     if current =None then
20:         return ⊥
21:     else
22:         (KeyID, M) ← current[t]
23:         current[t] ←None
24:         if KK_Auth(K[t], KeyID, R′, CHK′[t]) then
25:             h = hash_(1,KeyID)(R, M)
26:             return KK_GenSig2(K[t], KeyID, h)
27:         else
28:             return ⊥
```

answer. For example, in the IC-Forge game (defined below), the adversary is able to make calls to the functions `Init`, `Sign1`, `Sign2`, and `Final`. We could write $\mathcal{A}^{\texttt{Init},\texttt{Sign1},\texttt{Sign2},\texttt{Final}}$ to indicate that the adversary A has access to these four oracles. However, as a shortcut, we write $\mathcal{A}^{\text{CHAL}}$ to represent $\mathcal{A}$ making these function calls, and CHAL answering them.

As usual, A begins the game by calling `Init`, and ends the game by calling `Final`. When A wins the game, `Final` returns a 1, and by convention, A then also returns a 1. Thus, we can write $A^{\text{CHAL}} = 1$ to indicate that A plays the IC-Forge game against CHAL and wins, and $\Pr[A^{\text{CHAL}} = 1]$ to indicate the probability that adversary A wins the IC-Forge game when CHAL is answering its queries. To indicate the number of signing queries made, we write $A^{\text{CHAL}}(q)$

We write $\mathsf{Adv}^{\text{game}}$ for the maximum advantage of any adversary in the game. For forging games, this is

$$\mathsf{Adv}^{\text{forge}} = \max_{A} \Pr[A^{\text{forge}}(q) = 1]$$

For distinguishing games, it is

$$\mathsf{Adv}^{\text{dist}} = \max_{A} \left| \Pr[A^{\text{dist}}(q) = 1] - \frac{1}{2} \right|$$

Definition 2 (Transcript)**.** Consider the sequence of function calls made by the adversary in playing the IC-Forge game, starting with one call to `Init`, proceeding with calls to `Sign1` and `Sign2` in any order, and ending with one call to `Final`. We call the sequence of these calls and the values returned from them the *transcript* of the game, and write this as `TR`.

5.2 Security Notions: One-More Unforgeabilty, HBS-Forge, and IC-Forge

We will use several security notions in our proof; here, we define them.

PRF security. The security of a PRF is defined by the PRF/RF distinguishing game. In the game, the adversary calls the `Init` function once, then the `Query` function up to q times, followed by a call to `Final`. The adversary wins iff `Final` returns 1, and when this happens, the adversary also returns 1. The challenger for the PRF/RF game is defined in Algorithm 12.

Unlike other games we consider, which are information-theoretically secure, (their security depends only on the number q of queries), PRF-security is *computational*—an adversary with sufficient running time can exhaustively search for the secret key and break any PRF's security. In other words, PRF security depends on the number q of queries, but also on the adversary's running time τ. For some unit of measurement, a τ-time adversary is an adversary who's running time never exceeds τ units. The actual unit of time does not matter. E.g., searching through τ secret keys on the same machine the adversary is running on could take τ units of time.

The distinguishing advantage of an adversary $\mathcal{A}$ with respect to the PRF, given q queries, is

$$\left| \Pr[\mathcal{A}^{\text{PRF}} = 1] - \frac{1}{2} \right|.$$

We write $\mathsf{Adv}^{\text{PRF}}_{\tau}(q)$ for the maximum distinguishing advantage over all possible τ-time adversaries $\mathcal{A}$:

$$\mathsf{Adv}^{\text{PRF}}_{\tau}(q) = \max_{A\tau\text{-time}} \left| \Pr[A^{\text{PRF}} = 1] - \frac{1}{2} \right|$$

Algorithm 12 The PRF distinguishing game. The adversary has oracle access to either the function F with a random key or a random function RF, and must distingush between these cases. When the adversary's advantage is negligible, we say the PRF is secure.

```
    // PRF/RF Distinguishing Game
 1: function Init
 2:     b ←$ {0,1}
 3:     if b then
 4:         K ←$ {0,1}^n
 5:     else
    // Set of all functions from n to n bits
 6:         RF ←$ {{0,1}^n → {0,1}^n}
 7: function Query(x)
 8:     if b then
 9:         return F(K, x)
10:     else
11:         return RF(x)
12: function Final(b')
13:     return (b' = b)
```

One-More Unforgeability. A common security definition used for signatures is existential unforgeability (EU-CMA). Informally, this says that there is no adversary who, allowed to query a signing oracle on messages $M_{1\ldots q}$, can produce a signature on a message $M_* \notin \{M_{1\ldots q}\}$. However, this definition is somewhat weaker than what is provided by existing stateful HBS, because there are many surprising things the definition does not forbid. For example, it might be possible to take an existing signature on message M and create multiple valid signatures on M—this would not contradict the EU-CMA definition, but could cause practical security problems in some applications.

A stronger definition is one-more unforgeability (OMUF). Informally, this says that there is no adversary who, allowed to make q queries to a signing oracle, can produce a list of $q+1$ valid (message, signature) pairs. As discussed in [NT24], this security notion is especially useful for multisignatures and threshold signatures, as there are examples of signature schemes for which the single-signer version has OMUF security, but the threshold or multisignature version does not. OMUF is a very strong notion of security for a signature scheme, which implies both EU-CMA and strong unforgeability (SUF-CMA). Existing HBS schemes satisfy OMUF security in the single-signer case, and so it is worthwhile to prove that our threshold and distributed version of these signatures achieve the same security properties.

For a much more in-depth discussion of these definitions and their applications to threshold signatures, we refer the reader to [NT24].

RA-HBS-Forge: Security for a Stateful HBS. All stateful HBS are constructed from a set of one-time keys stored in a Merkle tree, and all modern designs also sign a randomized hash of the input message. Here, as in our definitions above, we assume that signing queries to the HBS include a `KeyID`—an index to the one-time key to be used for this signature. The HBS must never sign two messages with the same `KeyID` (that would mean reusing a one-time signing key). We assume the challenger will answer `KeyID` signing queries in any order. The `KeyID` can always be extracted from the full signature, and in fact is necessary to verify the signature in modern stateful HBS schemes.

A novel part of this definition is the existence of *compromise queries*, which allow an adversary to learn any desired one-time key, but then will not count signatures with a compromised key as being valid forgeries. This definition means that leaking any given one-time signing key does not undermine the security of other one-time keys in the composite key. This is necessary in order to model the security of distributed signatures against as adversary who has compromised all but one trustee. The adversary must not be able to forge signatures from a coalition that includes the honest trustee, but can trivially sign from coalitions for which it controls all trustees.

Definition 3 (RA-HBS Game)**.** The adversary interacts with the HBS challenger by

making function calls, which the challenger answers. The adversary must start by calling `Init`, followed by calls to `Compromise` and `Sign`. To end the game, the adversary calls `Final`. The adversary wins the game when `Final` returns 1, and the adversary likewise returns the value returned by its `Final` call. The functions are defined in Algorithm 13.

Remark: We will follow these conventions for the rest of this paper. We write our security games in terms of function calls made by the adversary and answered by a challenger or simulator. All our games will start with a call to `Init` and end with a call to `Final`. Where it is ambiguous which entity is answering the function calls, we will use dot notation, as with HBS.`Init` vs CHAL.`Init`.

Algorithm 13 The functions used by the RA-HBS forgery game. The function Init is called once with the adversary choosing D. Then the adversary can make any number of `Compromise` and `Sign` requests until calling `Final` with a forgery attempt. The internal state consists of a set of `KeyID`s "used" by the challenger to generate a signature, and another set of "compromised" `KeyID`s. If a `KeyID` is already used, `Sign` (`KeyID`,M) for some message M returns $\perp$. The adversary wins if it finally presents $s+1$ distinct, valid signatures from uncompromised one-time keys, where $s = |\text{used}|$ is the number signing queries under previously unused keys. We write σ_i for the triple $(R_i, \texttt{PATH}_i, Z_i)$.

```
1:  function Init(D)
2:      used ← {}
3:      compromised ← {}
4:      CPK, CSK ← StatefulGen(D)
5:      return CPK
6:  function Compromise(KeyID)
    // PK[0...D−1] can be computed from CSK
7:      PATH ← MT_MakePath(PK[0...D−1], KeyID)
8:      compromised ← ∪{KeyID}
9:      return CSK[KeyID], PATH
10: function Sign(KeyID, M)
11:     if KeyID ∈ used then
12:         respond ⊥                  // Don't sign with a key that has been used before.
13:     else
14:         σ_i ← StatefulSign_CSK(KeyID_i, M_i)
15:         used ← used ∪ {KeyID};
16:         return σ_i
17: function Final(L)
18:     if |L| ≠ |used| + 1 then
19:         return 0 // Present one signature more than messages signed by challenger.
20:     else
21:         if ∃(M_i, σ_i), (M_j, σ_j) ∈ L : (M_i, σ_i) = (M_j, σ_j), i ≠ j then
22:             return 0
23:         for each (M, σ) ∈ L do
24:             if MT_Extract(PATH) ∈ compromised then
25:                 return 0           // Signatures with compromised keys do not count.
26:             if StatefulVerify_CPK(M, σ) ≠ 1 then
27:                 return 0                       // Invalid signatures do not count.
28:         return 1  // There were |used| + 1 valid sigs using non-compromised keys
```

The goal of our proof, below, is to show that *any efficient algorithm that can defeat the*

security of our distributed hash-based signature scheme can be used to construct an efficient algorithm that defeats the security of the underlying stateful HBS scheme in this model.

IC-Forge. We need to define what it means for a distributed signature scheme like ours to be secure. Our scheme is different from other distributed signature schemes because it makes use of stateful hash-based signatures and sharding. This means that each possible signing coalition has a limited number of signatures, and that each one-time signing key may have a different signing coalition. However, the basic security requirement remains the same—an adversary cannot create a signature without the cooperation of all the trustees in one of the signing coalitions.

In order to capture this, we define the IC-Forge game as follows:

`Init:` The adversary decides on a set of coalitions, $C_{1..v}$. For each coalition, the adversary decides on a number of signatures from that coalition and a number of trustees k. Based on this, the adversary constructs the coalition list `CL` determining which `KeyID`s will be associated with each allowed signing coalition. The adversary also chooses one trustee to be honest, f. Finally, the adversary sends `CL`, k, f to the challenger, and the challenger sends back the public key `CPK`, the common reference string `CRV` and the secrets for all trustees except the honest one.

`Sign1` and `Sign2`: The adversary makes signature queries which the challenger (playing the role of the one uncompromised trustee) answers.

`Final:` The adversary provides list L of (message, signature) pairs to the challenger. There are two requirements for the forgery message to be valid (and thus, for `Final` to return 1):

1. The list L contains $q + 1$ distinct and valid (message, signature) pairs.
2. None of the signatures come from one-time keys for which the adversary has compromised all members of the signing coalition.

That last requirement is unique to our setting, and follows from the fact that in our game, there can be some one-time keys whose signing coalition does not include the honest trustee. In that case, the adversary knows the private key, and so can trivially sign with those keys offline, so our security game excludes signatures from those keys—they aren't actually forgeries!

Definition 4 (IC-Forge challenger CHAL)**.** The IC-Forge game is formally defined with respect to our sharding scheme and stateful hash-based signatures. The challenger's behavior is depicted in Algorithm 14. Recall that the OMU security definition (c.f. Section 5.2) requires that we can count (or at least bound) the number of signing queries. In the IC-Forge game, CHAL counts a signing query as happening when the honest trustee receives the first signing call, `Sign1`. This is an upper bound on the total number of valid signatures: the adversary could follow this call with an invalid `Sign2` call and thus not learn a valid signature, but will never learn a valid signature without making the initial `Sign1` call.

For the proof, it will be useful to consider a variation of the games in Algorithm 14:

Definition 5 (Random Function IC-Forge Game)**.** The random function IC-Forge game (also refered to as "CHAL*") is identical to CHAL, except for the following changes:

1. CHAL* begins by sampling a random function RF.

Algorithm 14 The challenger in the IC-Forge game (also referred to as "CHAL"): The adversary calls `Init` to start the game, then may call `Sign1` and `Sign2` in any order up to q times each. The game ends when the adversary calls `Final`. The game keeps track of the number of calls to `Sign1` made (count). A call to `Final` (L) wins the game iff it contains count+1 distinct, valid signatures, where each signature comes from a one-time signing key that was not known to the adversary. Note that `ShardSign1`$(f, \texttt{KeyID}, M)$ in Line 10 returns $\perp$ if `KeyID` is not in the list of f's unused `KeyID`s. Here k is the number of trustees, D the number of signatures, the "coalition list" `CL`$[0, \ldots, D-1]$ is an array of D coalitions, i.e., nonempty subsets from $\{1, \ldots, k\}$. The internal state consists of the composite public key `CPK`, the honest trustee's index f, its secret key $K[f]$, and the variable "current". After making a first and before making a second request, current holds the pair $(\texttt{KeyID}, M)$. Otherwise, it holds $\perp$.

```
1: function Init(k, D, CL, f)
2:     count ← 0
3:     CPK, CRV, K[1..k] = ShardSetup(D, k, CL)
4:     current ← ⊥
5:     TrusteeSetup(CL, K, k)
6:     return CPK, CRV, K[t ≠ f]

7: function Sign1(KeyID, M)
8:     count ← count +1
9:     current ← (KeyID, M)
10:    return ShardSign1(f, KeyID, M)

11: function Sign2(R′, CHK′)
12:    if current = ⊥ then
13:        return ⊥
14:    else
15:        (KeyID, M) ← current
16:        current ← ⊥
17:        return ShardSign2(f, R′, CHK′[f])

18: function Final(L)
19:    if |L| ≠ count + 1 then
20:        return 0 // Present one signature more than messages signed by challenger.
21:    else
22:        if any two (message, signature) pairs in L are identical then
23:            return 0                          // Duplicate signatures don't count.
24:        for each M, (R, PATH, Z) ∈ L do
25:            KeyID ← MT_Extract(PATH)
26:            if f ∉ CL[KeyID] then
27:                return 0              // Discount signatures without honest trustee.
28:            if StatefulVerify_CPK(M, R, PATH, Z) ≠ 1 then
29:                Return 0                      // Invalid signatures do not count.
30:        return 1   // There were count + 1 valid sigs using non-compromised keys
```

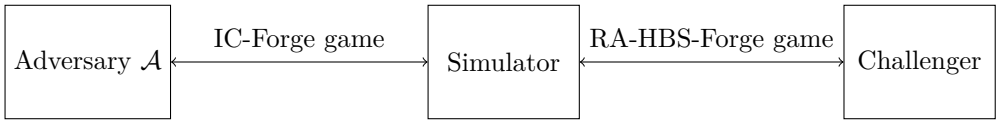

Figure 4: The simulator plays the IC-Forge game with the adversary, and at the same time, plays the RA-HBS-Forge game with the RA-HBS challenger. The IC-Forge game is defined in Definition 4 and the RA-HBS-Forge game in Definition 5.

2. Let f be the honest trustee. Then CHAL* replaces each $\mathrm{PRF}(K[f], x)$ call with a call to $\mathrm{RF}(x)$.[8]

5.3 The Simulator

The simulator is the core of our security proof: it plays the IC-Forge game with the adversary, and at the same time, plays the RA-HBS-Forge game with the RA-HBS challenger. Below, we will demonstrate that an adversary that wins the IC-Forge game against the challenger with probability P_1 will win the game against the simulator with almost the same probability: $P_1 - \epsilon$ for some very small ϵ. We will also demonstrate that when the adversary wins the IC-Forge game against the simulator, the simulator always wins the HBS-Forge game against the RA-HBS challenger.

Following our conventions above, we represent the simulator's interactions with the adversary in terms of a set of function calls made by the adversary, which the simulator answers. We represent the simulator's interactions with the RA-HBS challenger in terms of function calls made by the simulator to the functions defined in Algorithm 13. Following the convention described above, we write these as HBS.`Init`, HBS.`Compromise`, HBS.`Sign` and HBS.`Final`.

The simulator's behavior is defined in Algorithm 15. In addition, an overview is given in Figure 4.

5.4 The Proof

Overview of the Proof: Informally, our proof aims to show that if there exists an adversary $\mathcal{A}$ that wins the IC-Forge game with probability P, we can use it to construct a new adversary, SIM, which will win the HBS-Forge game with probability at least $P - \epsilon$, where ϵ is a very small probability that we bound below.

We do this in a few steps:

1. We start from the following definitions:

$$\mathsf{Adv}^{\text{IC-Forge}}(q) = \max_{\mathcal{A}} \Pr[\mathcal{A}^{\text{CHAL}}(q) = 1]$$
$$\mathsf{Adv}^{\text{RA-HBS}}(q) = \max_{\mathcal{A}} \Pr[\mathcal{A}^{RA-HBS}(q) = 1]$$
$$\mathsf{Adv}^{\text{PRF}}_{\tau}(q) = \max_{\mathcal{A}\tau\text{-time}} \Pr[\mathcal{A}^{PRF}(q) = 1]$$

2. Forgery Transfer Lemma (Lemma 1): For any $\mathcal{A}$,

$$\mathcal{A}^{\text{SIM}}(q) = 1 \implies SIM^{\text{RA-HBS}}(q) = 1\,,$$

And thus,

$$\Pr[\mathcal{A}^{\text{SIM}}(q) = 1] = \Pr[SIM^{\text{RA-HBS}}(q) = 1]\,.$$

[8] Note that none of the functions in CHAL (Algorithm 14) directly make such calls, but there are functions on the chain calling $\mathrm{PRF}(K[f], x)$.

Algorithm 15 The Simulator SIM. It plays the IC-Forge game against the adversary, supporting the same interface as CHAL: the adversary calls `Init` once, makes any number of calls to `Sign1` and `Sign2`, and eventually calls `Final` with a forgery attempt. However, it simultaneously plays the HBS-Forge game (calling HBS.`Init`, HBS.`Compromise`, HBS.`Sign`, and HBS.`Final`) against the RA-HBS challenger. As proven below, (1) an adversary that wins the IC-Forge game against the challenger wins it against the simulator with almost the same probability, and (2) when the adversary wins the IC-Forge game against the simulator, the simulator wins the HBS-Forge game against the RA-HBS challenger.

function `Init`$(k, D, \texttt{CL}, f)$
  `CPK` $\leftarrow$ HBS.`Init`(D)
  for $t = 1 \ldots k$ **do**:
    $K[t] \leftarrow\!\!\$\ \{0,1\}^n$
  current $\leftarrow \perp$
  for `KeyID` $\leftarrow 0 \ldots D-1$ **do**:
    if $f \in$ `CL`[`KeyID`] **then** // Fill the `CRV` entries with random bits.
      `CRV`[`KeyID`].$SK[i,j] \leftarrow\!\!\$\ \{0,1\}^n$ for all i, j
      `CRV`[`KeyID`].$R \leftarrow\!\!\$\ \{0,1\}^n$
      `CRV`[`KeyID`].CHK $\leftarrow\!\!\$\ \{0,1\}^{n \times |\texttt{CL[KeyID]}|}$
      `CRV`[`KeyID`].`PATH` $\leftarrow\!\!\$\ \{0,1\}^{|\texttt{PATH}|}$
    else
      SK, `PATH` $\leftarrow$ HBS.`Compromise`(`KeyID`)
      $R \leftarrow\!\!\$\ \{0,1\}^n$
      $k' \leftarrow |$`CL`[`KeyID`]$|$
      keys $\leftarrow$ MakeKeyList$(K,$ `CL`[`KeyID`]$)$
      `CRV`[`KeyID`] $\leftarrow$ `KK_Setup`$(keys, k',$ `KeyID`, SK, $R,$ `PATH`$)$
  return `CPK`, `CRV`, `CL`, $K[t \neq f]$

---

function `Sign1`(`KeyID`, M)
  R, `PATH`, $Z \leftarrow$ HBS.`Sign`(`KeyID`, M)
  // Compute the shares of the honest trustee to make the protocol work out.
  Let CHK be an array of $|$`CL`[`KeyID`]$|$ values
  for $i \leftarrow 0 \ldots |$`CL`[`KeyID`]$| - 1$ **do**
    CHK$[i] \leftarrow \mathrm{PRF}^{\mathrm{AUTH}}_{K[\texttt{CL}[i]]}(\texttt{KeyID}, R)$ // Need to work out how this is indexed //
  Compute $h = \texttt{hash}_{\texttt{tweak}(1)}(R, M,$ `KeyID`$)$
  for $t \in$ `CL`[`KeyID`]$, t \neq f$ **do**
    R^t, CHK$^t \leftarrow$ `KK_GenSig1`$(K[t],$ `KeyID`$)$
    `PATH`^{t}, $Z^t \leftarrow$ `KK_GenSig2`$(K[t],$ `KeyID`$, h)$
  $R^f \leftarrow (\bigoplus_{t \in \texttt{CL[KeyID]}, t \neq f} R^t) \oplus \texttt{CRV[KeyID]}.R \oplus R$
  CHK$^f \leftarrow (\bigoplus_{t \in \texttt{CL[KeyID]}, t \neq f}$ CHK$^t) \oplus$ `CRV`[`KeyID`].CHK $\oplus$ CHK
  `PATH`$^f \leftarrow (\bigoplus_{t \in \texttt{CL[KeyID]}, t \neq f}$ `PATH`$^t) \oplus$ `CRV`[`KeyID`].`PATH` $\oplus$ `PATH`
  $Z^{\texttt{CRV}} \leftarrow$ `WINTER`$(h,$ `CRV`[`KeyID`].SK$)$
  $Z^f \leftarrow (\bigoplus_{t \in \texttt{CL[KeyID]}, t \neq f} Z^t) \oplus Z^{\texttt{CRV}} \oplus Z$
  current $\leftarrow$ `PATH`$^f, Z^f$
  return R^f, CHK^f

---

function `Sign2`$(R',$ CHK$')$
  if $R' \neq R$ OR CHK$'[f] \neq$ CHK$[f]$ OR current $= \perp$ **then**
    return $\perp$
  else
    current $\leftarrow \perp$
    return `PATH`$^f, Z^f$

---

function `Final`(L)
  return RA-HBS.`Final`(L)

3. CHAL-CHAL* Lemma (Lemma 2): For any τ-time $\mathcal{A}$,

$$\left|\Pr[\mathcal{A}^{\mathrm{CHAL}}(q)=1]-\Pr[\mathcal{A}^{\mathrm{CHAL}^*}(q)=1]\right| \le \mathsf{Adv}^{PRF}_{\tau'}(q\times u)\,,$$

   where $\tau' \in O(\tau+q)$ is τ plus the time to evaluate either of CHAL or CHAL*.

4. CHAL*-SIM Lemma (Lemma 4): For any $\mathcal{A}$,

$$\left|\Pr[A^{\mathrm{CHAL}^*}(q)=1]-\Pr[A^{\mathrm{SIM}}(q)=1]\right| \le q\times 2^{-n}\,.$$

5. CHAL-SIM Lemma (Lemma 5): For any τ-time $\mathcal{A}$ and τ' as above,

$$\left|\Pr[A^{\mathrm{CHAL}}(q)=1]-\Pr[A^{\mathrm{SIM}}(q)=1]\right| \le \mathsf{Adv}^{\mathrm{PRF}}_{\tau'}(q\times u)+q\times 2^{-n}\,.$$

6. Combining all these, we can see that for any τ-time $\mathcal{A}$ and τ' as above,

$$\Pr[\mathcal{A}^{\mathrm{CHAL}}(q)=1] \le \Pr[\mathcal{A}^{\mathrm{SIM}}(q)=1]+\mathsf{Adv}^{\mathrm{PRF}}_{\tau'}(q\times u)+q\times 2^{-n}\,.$$

7. Main Security Theorem (Theorem 1):

$$\mathsf{Adv}^{\text{IC-Forge}}(q) \le \mathsf{Adv}^{\text{HBS-Forge}}(q)+\mathsf{Adv}^{\mathrm{PRF}}_{\tau'}(q\times u)+q\times 2^{-n}\,.$$

Lemma 1 (Forgery Transfer)**.** *If $A^{SIM^{HBS}}=1$, then SIM also wins the HBS-Forge game.*

Proof. SIM.`Final`(`L`) returns the result of HBS.`Final`(`L`). HBS.`Final`(`L`) returns a 1 only when its adversary has won the HBS-Forge game by producing more valid signatures that were requested by queries to HBS.`Sign`, where each valid signature uses a `KeyID` for which there was never a HBS.`Compromise` (`KeyID`) query. Thus, if SIM.`Final` returns 1, SIM has forged a signature against RA-HBS. □

The Forgery Transfer lemma simply says that for A to win the IC-Forge game against SIM, it must also allow SIM to win the RA-HBS forgery game and thus violate the security of the underlying HBS scheme.

Indistinguishability

If we receive the same forgery message when the adversary plays against the challenger and the simulator, then winning the IC-Forge game against our scheme implies winning the RA-HBS-Forge game against the hash-based signature scheme. Thus, we need to bound the advantage of any adversary in distinguishing the challenger from the simulator when playing the IC-Forge game against it. The bound on distinguishing the challenger from the simulator leads directly to bound on the difference in probability between an adversary winning IC-Forge and the simulator winning RA-HBS-Forge.

Let CHAL, CHAL*, and SIM be the algorithms defined above which play the IC-Forge game against the adversary. We now bound the advantage of any adversary distinguishing CHAL from SIM.

Lemma 2 (CHAL-CHAL* Indistinguishability)**.** *There is no τ-time algorithm A_1 that can distinguish playing the IC-Forge game against CHAL vs CHAL* with advantage greater than $\mathsf{Adv}^{\mathrm{PRF}}_{\tau'}(q)$. Here τ' is τ plus the maximum time to evaluate either of CHAL and CHAL*.*

Proof. Suppose algorithm $\mathcal{A}_1$ distinguishes playing IC-Forge against CHAL vs CHAL* with advantage α. Then we can use $\mathcal{A}_1$ to build an RF/PRF distinguishing algorithm: When playing the PRF/RF distinguishing game, we implement CHAL, but

with the calls to $\mathrm{PRF}(K[f], x)$ replaced with responses to queries to the function oracle on input x. We then play IC-Forge against $\mathcal{A}_1$. Suppose (wlog) that $Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}] > Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}^*]$. Our new PRF-RF distinguishing algorithm guesses PRF when $\mathcal{A}_1(q) = 1$, and RF otherwise. This algorithm will have an advantage of $|Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}] - Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}^*]|$. $\mathcal{A}_1$ makes $q \times u$ queries to the PRF/RF oracle, where u is a relatively small integer (at most a few hundred) whose exact value is determined by the parameters of the HBS scheme used.

By definition, $\mathsf{Adv}_{\tau'}^{\mathrm{PRF}}(q \times u)$ is the maximum distinguishing advantage between the PRF and random function, given $q \times u$ queries. Thus, for any $\mathcal{A}$,

$$|Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}] - Pr[\mathcal{A}_1(q) = 1|\mathrm{CHAL}^*]| \leq \mathsf{Adv}_{\tau}^{\mathrm{PRF}}(q \times u)$$

□

Next, we need to define and bound a BAD event.

Definition 6 (BAD)**.** The event BAD occurs when the adversary sends R', CHK' such that $R' \neq R$ but $\mathrm{CHK}'[f] = \mathrm{PRF}_{K[f]}^{\mathrm{AUTH}}(\mathtt{KeyID}, R, n)$

Lemma 3. $\Pr[\mathtt{BAD}] \leq q \times 2^{-n}$.

Proof. By the definition of CHAL*, each call to $\mathrm{PRF}(K[f], X)$ is replaced by a call to random function $RF(X)$. In ShardSetup, $\mathrm{CHK}[f] = \mathrm{PRF}(K[f], \mathtt{KeyID}, R, n)$; in CHAL* this is replaced by $RF(\mathtt{KeyID}, R, n)$. By the definition of a random function, $\Pr[RF(\mathtt{KeyID}, R, n) = \mathrm{RF}(\mathtt{KeyID}, R', n)] = 2^{-n}$. Since the adversary makes at most q signature queries, $\Pr[\mathtt{BAD}] \leq q \times 2^{-n}$. □

Lemma 4 (CHAL*-SIM Indistinguishability)**.** *There is no algorithm that distinguishes CHAL* from SIM with advantage greater than* $\Pr[\mathtt{BAD}]$.

Proof. Consider the transcript TR of all messages passed between the adversary and either CHAL* or SIM.

The CRV from SIM is filled with random bits. The CRV from CHAL* is filled with known values XORed with the output of a random function with distinct inputs for each random function call. These yield identical distributions for the CRV, so $\Pr[\mathtt{CRV}|\mathrm{CHAL}^*] = \Pr[\mathtt{CRV}|\mathrm{SIM}]$. Thus, the response to the Init query has equal probability when it is answered by CHAL* or by SIM.

The value of $R, \mathtt{PATH}, Z$ are determined by the stateful HBS scheme, and are identically distributed in CHAL* and SIM.

The response to a Sign1 query is R^f, CHK^f. Given $R, \mathrm{CHK}, \mathtt{CRV}$, SIM and CHAL* send identical values of R^f, CHK^f. The value of CHK in SIM and CHAL* differ only in $CHK[f]$. In CHAL*, $\mathrm{CHK}[f]$ is the output of RF applied to a sequence of inputs including R; in SIM, it is a random n-bit string. The output of a random function has the same distribution as a random string, so this adds no distinguishing advantage for the adversary.

The response to a Sign2 query is either $\perp$ or $\mathtt{PATH}^f, Z^f$. If the response is $\mathtt{PATH}^f, Z^f$, then once again, given $\mathtt{PATH}, Z, \mathtt{CRV}$ these values are identical from SIM and from CHAL*.

In CHAL*, the second response is $\perp$ if $\mathrm{CHK}[f]$ does not agree with R'. In SIM, the second response is $\perp$ if either $\mathrm{CHK}[f]$ does not agree with R', or if $R' \neq R$. However, $\mathrm{CHK}[f]$ agrees with R' when $R' \neq R$ is the definition of BAD, and so happens with probability $\Pr[\mathtt{BAD}]$.

The response to a Final(L) query is 1 or 0. Playing against SIM, this is the result of HBS.Final(L). (See Algorithm 15). From Algorithm 13, we see that this returns 1 if the following holds:

1. L consists of used + 1 distinct entries, where used is the number of valid Sign queries made by the adversary.

2. Every signature is valid.
3. No signature came from a `KeyID` for which a compromise query was made.

Further, examining Algorithm 15, SIM makes compromise queries for exactly the `KeyID`s for which the adversary controls all trustees.

Playing against CHAL* (see Algorithm 14), the result is 1 if the following holds:

1. L consists of count + 1 distinct entries, where count is the number of valid `Sign1` queries made by the adversary.
2. Every signature is valid.
3. No signature came from a `KeyID` for which the adversary controlled all trustees.

In either case, the three conditions are the same. Thus, conditioned on the rest of the transcript, SIM.`Final`(`L`) = CHAL*.`Final`(`L`). □

Lemma 5 (CHAL-SIM Indistinguishability)**.** *Let τ and τ' be as in Lemma 2. No τ-time algorithm distinguishes CHAL from SIM with advantage greater than $ADV_{PRF}(\tau') + \Pr[\texttt{BAD}]$.*

Proof. This follows directly from combining the CHAL-CHAL* and the CHAL*-SIM Indistinguishability lemmas. □

Theorem 1 (Main Security Theorem)**.** *Let τ and τ' be as in Lemma 2. For every τ-time adversary, the probability to win the IC-Forge game making q queries is at most*

$$Adv^{IC\text{-}Forge}(q) \leq Adv^{HBS\text{-}Forge}(q) + Adv^{PRF}_{\tau'}(q \times u) + q \times 2^{-n}.$$

Proof. By the Forgery Transfer Lemma, we know that a successful forgery message at the end of playing IC-Forge against SIM becomes a successful forgery message playing RA-HBS-Forge against RA-HBS. Thus, assuming the adversary sends the same forgery message to SIM as to IC-Forge, an incomplete-coalition forgery reduces to a forgery on a random-access version of the stateful HBS scheme.

By the CHAL-SIM indistinguishability lemma, the adversary has advantage in distinguishing playing IC-Forge against CHAL vs playing it against SIM of no more than $\mathrm{ADV}^{\mathrm{PRF}}_{\tau'} + \Pr[\texttt{BAD}]$. Thus, with probability $1 - (\mathrm{ADV}^{\mathrm{PRF}}_{\tau'} + \Pr[\texttt{BAD}])$, a valid forgery message against CHAL leads to a valid forgery message against SIM, and thus against RA-HBS.

By definition,

$$\mathsf{Adv}^{\text{IC-Forge}}(q) = \max_{\mathcal{A}\tau\text{-time}} \Pr[\mathcal{A}^{\text{CHAL}}(q) = 1],$$

and

$$\mathsf{Adv}^{\text{RA-HBS}}(q) = \max_{\mathcal{A}} \Pr[A^{HBS}(q) = 1].$$

Combining these together and using the union bound, we get

$$\mathsf{Adv}^{\text{IC-Forge}}(q) \leq \mathsf{Adv}^{\text{HBS-Forge}}(q) + \mathsf{Adv}^{\text{PRF}}_{\tau'}(q \times u) + q \times 2^{-n},$$

for every τ-time adversary, as claimed. □

6 Practical Considerations

A major goal for our scheme is to construct a *practical* mechanism for doing threshold stateful HBS. Namely, one that can find use in the real world. This shaped many of our design decisions. For example, our use of a large common reference string is a consequence of wanting trustees that are possible to be implemented using hardware security modules with limited secure storage, and our use of sharding is a consequence of avoiding the simplifying (but not very practical) assumption of a broadcast channel for our protocol.

6.1 Computing, Memory, and Communications

Key Generation and Setup. The dealer generates a stateful HBS composite key in the same way as a normal user would generate it. Splitting the stateful HBS composite key into ℓ shares requires approximately ℓ times the work needed to generate the keys.

While splitting the composite key into ℓ shares, the `CRV` is created and stored by the dealer. This requires sufficient storage to keep the `CRV`, see below.

Each trustee needs only to be given its PRF key, its coalitions, and an indication of which one-time keys go with each coalition. This means that each trustee's secrets and state can be kept on a smartcard or hardware security module.

Signing. A signature requires two round-trip messages between the aggregator and each responding trustee, as well as the aggregator looking up values in the `CRV`. Communications can be point-to-point and secured using only symmetric cryptography[9]; there is no need for a broadcast channel. The data exchanged between the aggregator and each trustee consists of the message being signed plus a few hundred additional bytes.

Trustees can be very lightweight. During the signing protocol, each trustee does a little more work than is needed to compute the underlying one-time signature, plus some additional PRF calls. Unlike the full stateful HBS, the trustee never needs to construct a Merkle tree path, so its total work is typically less than is needed to compute a signature with the underlying stateful HBS. A trustee only needs to store: (1) its PRF key, (2) the current `KeyID` to be used with each coalition of which it is a member, and (3) any pre-shared symmetric keys needed for communications. These requirements make it practical to use a hardware security module or a smartcard to hold each trustee's shares and carry out its computations.

In practice, the aggregator is likely to be a desktop or laptop computer. To construct a signature, the aggregator does about the same work as is needed to sign with the underlying one-time signature scheme, plus some very inexpensive computations (XORing together bitstrings), but must have access to the `CRV`. During a signing protocol, it communicates with the trustees, computes a message hash, looks up a few values in the `CRV`, XORs some bitstrings together, and finally verifies the signature.

Memory Requirements for the `CRV`. We have written a proof-of-concept implementation which requires approximately the following amount of storage for each of the D `KeyID`s, assuming at most k trustees for each coalition:

- $(a + c)2^w$ values for the Winternitz signature,
- $\log_2(D)$ values for the path, and
- $k + 1$ values to generate and check the randomizer R.

Given D keys and an n-bit hash function, the `CRV` of our reference implementation needs about

$$|\texttt{CRV}| = n \cdot \big(D \cdot (a + c) \cdot 2^w + D \cdot \log_2(D) + D \cdot (k + 1)\big) \text{ bits.}$$

Table 1 shows the `CRV` sizes for LMS keys with constant $k = 3, D = 2^{15}$, and various choices of the other LMS parameters.

The size of the `CRV` scales linearly with the number of one-time keys. Thus, in order to approximate the `CRV` sizes for different choices of D, simply scale the given `CRV` size up or down by the correct value of D. E.g., a 3-of-5 LMS instance with $n = 256, w = 4$, and $D = 2^{25}$ would require a `CRV` of 2^{10} times the storage needed for the $n = 256, w = 4$

[9] Unsecured communication would not introduce any weaknesses to the security of the signatures or keys, but the trustees must be able to decide whether a signature request came from anyone they trust.

Table 1: CRV sizes required by our proof-of-concept implementation for $D = 2^{15}$ keys. All signing coalitions contain $k = 3$ trustees. The hash length is given in bits, w is the Winternitz parameter. Parameter sets with hash length 128 are actually not defined in LMS, but extrapolated in the table. Hash length 128 is, e.g., used in SLH-DSA [Nat24b].

hash length	defined in	**w**	$\log_2(\mathbf{D})$	**size (GiB)**
128	—	4	15	0.3
128	—	8	15	2.3
192	[MCF19, Int19]	4	15	0.6
192	[MCF19, Int19]	8	15	4.9
256	[MCF19, Int19]	4	15	1.1
256	[MCF19, Int19]	8	15	8.5

entry in the table. This would be about 1.1 TiB. (For comparison: at the time of writing, we could buy an external 2 TiB storage device for about $80 or 70€.)

The trustees (presumably most of them hardware security modules with limited memory and connectivity) do not need access to the `CRV`. The aggregator, which is untrusted, must access to the `CRV` and can run on a general-purpose computer. Even a few GiB will fit easily on the hard drive of even a low-budget computer. The aggregator can also employ a cloud storage service to maintain the `CRV`.[10] Keeping the `CRV` on an untrusted device or service is acceptable, cf. Footnotes 1 and 3.

6.2 Coalitions, Scaling, and Trustee Management

As pointed out in Section 1.3, our usage model is quite different from the usage model implied in most of the related work. This puts some constraints on the usage, but also enables some novel scenarios.

A major constraint, quite in contrast to other types of threshold cryptosystems, is to keep the number of coalitions manageable. Recall that no two different coalitions can share even a single `KeyID`, and that every trustee must keep track of which `KeyID`s have been used (e.g., a 10-of-20 scheme, with $\binom{20}{10} = 184756 \approx 2^{17.5}$ different coalitions). Assuming, say, $D = 2^{20}$ `KeyID`s, there would be just 5–6 `KeyID`s for each coalition.

On the other hand, distributed signature schemes, such as ours, allow for a much richer set of coalitions than threshold schemes. For example, assume Alice owns the secret key, but worries about her computer being taken over. She thus sets up u trustees $T_1, \ldots T_u$, such that she needs any T_i, regardless of which one, *and* her own computer, to sign. For reliability, Alice also allows all coalitions of $u - 1$ trustees without her to jointly sign. See Example 5 below.

Feasibility. The main performance parameter is the number $D = 2^d$ of `KeyID`s: Key generation time and storage requirements for the `CRV` grow linearly with D, signature size linearly with d. The number D is determined by two factors: the number of different coalitions we support, and the number of signatures for each coalition. Also relevant, especially for lightweight trustees with small storage, is the number of coalitions each trustee is in, as each trustee must perform bookkeeping for the `KeyID`s used so far.[11]

Examples. Assume a given `CPK` must support $20\,000 \approx 2^{14.3}$ signatures. Depending on the specific requirements, the depth d of the Merkle-tree and the number of counters each

[10] At the time of writing, there are cloud storage services offering free accounts with 2 GiB of storage.

[11] Assume $\texttt{KeyID}_1, \ldots, \texttt{KeyID}_L$ are assigned to coalition C and used in sequential order. Beyond knowing the bounds $\texttt{KeyID}_1$ and $\texttt{KeyID}_L$, each trustee must at least maintain a counter $c \in \{0, \ldots, L\}$, to indicate the index $\texttt{KeyID}_{c+1}$ of the coalition's next secret key to be used (or none, if $c = L$).

trustee must store can vary. We consider four threshold/distributed scenarios. For the sake of a proper comparison, we start with the scenario of single-signer signatures.

1. *Single-signer:* $d = 15$ suffices, and the signer must maintain one counter.
2. *Full k-of-k Threshold:* There is one single coalition. As in the single-signer case, $d = 15$ suffices. Since there is only one coalition, and all trustees are in that coalition, each trustee just has to maintain one counter.
3. *Pessimistic 4-of-6 Threshold:* There are $\binom{6}{4} = 15$ coalitions, and each coalition must be able to perform the required 20 000 signatures. This implies $D \geq 15 * 20\,000 = 300\,000 \approx 2^{18.2}$, i.e., $d = 19$. Each trustee is in $\binom{5}{3} = 10$ coalitions.
4. *Optimistic 4-of-6 Threshold:* Assume that usually all six trustees are available. Once in a while (for, at most about 10% of the signatures), only five trustees will be available. Coalition of four trustees are reserved for a small number of emergencies. This is consistent with assigning 20 000 `KeyID`s to the coalition with all six trustees, 2048 `KeyID`s to each of the six coalitions with five trustees, and 32 `KeyID`s to each of the 15 emergency coalitions. The total number of `KeyID`s is $20\,000 * 1 + 2048 * 6 + 32 * 15 = 32768 = 2^{15}$, i.e., $d = 15$ suffices, as in the single-signer case. For this 4-of-6 threshold scheme, each trustee is in $1 + 5 + 10 = 16$ coalitions.
5. *Distributed, non-Threshold:* Assume six trustees A, B, C, D, E, F and two rules: (1) Trustee A can jointly form a coalition with any other trustee. (2) Without A, any quadruple from {B, C, D, E, F} is a coalition. Thus, there are five coalitions from rule (1) and also five coalitions from rule (2), i.e., ten coalitions in total. Each coalition must be able to sign up to 20 000 messages. Since $20\,000 * 10 < 2^{18}$, $d = 18$ suffices. Each trustee is in exactly five coalitions.

Dynamic Coalition and Trustee Management. A major benefit of our scheme is the ability to selectively revoke `KeyID`s. If, e.g., trustee t^* is no longer trusted, the dealer can revoke all `KeyID`s assigned to all coalitions C_i with $t^* \in C_i$, without touching any other coalitions. E.g., a 4-of-6 threshold scheme would then turn into a 4-of-5 scheme, without the immediate need to generate and distribute a new public key. The dealer could either maintain a public list of revoked `KeyID`s, or simply inform each still-honest trustee to stop signing with coalitions including t^*. Suppose a trustee t^* becomes permanently unavailable (for example, the HSM storing its key fails). If the dealer still knows the secret key of t^*, it can update the `CRV` by XORing the shares of t^* into `CRV` entries of `KeyID`s for which t^* was part of the signing coalition. From that point on, all one-time keys that previously required t^* to sign no longer need t^*—the remaining trustees in that key's coalition suffice to sign a message. Coalitions that never included t^* will not be affected. This salvages the one-time keys associated with t^*. E.g., a 4-of-6 threshold scheme becomes a mixture of a 4-of-5 and a 3-of-5 scheme (all previous 4-coalitions with t^* turn into 3-coalitions, while previous 4-coalitions without t^* do not change). [12]

If the dealer initially keeps some `KeyID`s unallocated, it can also create new coalitions, add trustees, or assign new `KeyID`s to coalitions which run out of `KeyID`s. The dealer just runs the dealing process for (some of) the previously unallocated `KeyID`s. Arguably, this is tantamount to stopping and continuing the dealing process. While the `CRV` is updated (or extended), the public key is not affected. Furthermore, given at least one still unallocated `KeyID`, the dealer can also make a new public key, rerun the setup process, and use the unallocated `KeyID` to sign that public key. However, the signatures would get longer, and they would become hypertree signatures (like XMSS^{MT} or HSS).

[12] The dealer might consider to XOR the shares of a new t^{**} to the `CRV`, e.g., to turn those 3-coalitions back into 4-coalitions. But this would be unsafe: the shares of a trusted t^{**} are secret, and the XOR of old and new `CRV` would reveal those secrets. Also, t^{**} would not know which `KeyID`s have already been used.

7 Conclusions

7.1 Post-Quantum Security

This paper contributes to the small number of post-quantum secure threshold signature schemes, see, e.g., [BGG+18, ASY22, Ata23, Ata24]. But our approach has two major benefits over that research:

1. [ASY22, BGG+18] are based on the assumed hardness of *lattice* problems. [Ata23, Ata24] is based on *isogenies*. In contrast to those schemes, hash-based signature schemes have been known since the early days of public-key cryptography, and have been deeply studied over time. Assuming a well-studied standard hash function is used, their security is widely trusted. Thus, we argue that hash-based signature schemes *provide a better security assurance*, compared to lattice-based and isogeny-based schemes.

2. We propose a proper *distributed* post-quantum-secure signature scheme, thus going beyond pure threshold signature schemes. To the best of our knowledge, there has not yet been any proposal for a post-quantum secure distributed (beyond-threshold) signature scheme.

7.2 Final Remarks

In this article, we have shown how to turn stateful hash-based signatures (HBS) into threshold and distributed signature schemes. This may come as a surprise, since hash-based signatures lack the convenient homomorphic properties of RSA, ElGamal, and other signature schemes for which practical threshold variants are known.

Practicality. Our techniques are practical: they have low communications and computational overhead, require only fast symmetric cryptography, and can be applied to existing HBS standards such as LMS[MCF19] or XMSS[HBG+18]. Our techniques require a common reference value (`CRV`) for each stateful HBS public key, typically between a few hundred MiB and a few GiB, but this `CRV` can be stored on an untrusted device, and will fit easily on the hard drive of a desktop or laptop computer.

Implementation. We have written an implementation of our scheme for LMS signatures in Python, including a dealer method to generate `CPK` and `CRV` and an aggregator method running the signing protocol. The public key and signatures are LMS compatible. We stress that this is only a proof of concept; trustees are implemented as local objects called by the aggregator.

Trust and Distributed Trust. Our techniques assume a trusted party to generate the key and construct the shares and `CRV` during the setup phase. This is both a strength and a weakness. It is a strength, because it dynamically allows the owner of the key to distribute its trust by delegating its signing capability to trustees. This can be done at any point of time, as long as the key owner does not distribute any already-used one-time keys. The key owner could even reserve some one-time keys for signing by itself, and split the remaining one-time keys into shares to be distributed to a set of trustees. The downside is that our techniques require a trusted dealer for setup.

Limitations. Our techniques have a few limitations: As pointed out above, the need for a dealer is one. Also, while our techniques work well for relatively small numbers of signing coalitions, as with, say, a 5-of-10 threshold signature scheme, they do not scale

well for bigger numbers of coalitions. Even a 10-of-20 threshold signature scheme would hardly be practical, using our techniques.

Stateless Signatures. Our techniques work with FORS signatures in a straightforward way. This might suggest the possibility of using our techniques with stateless HBS like SLH-DSA. Unfortunately, the CRV must be at least as large as the sum of the size of the one-time signing keys for *all* of the Winternitz and FORS signatures. For the parameter sets listed in SLH-DSA [Nat24b] (the NIST standard based on SPHINCS+), the setup time and CRV size would both be completely impractical. For example, the SLH-DSA-SHA2-128s parameter set would require more than 2^{73} bytes of CRV and at least 2^{80} work for the dealer to set up the scheme.

[FD24] study not-yet-standardized variants of stateless SPHINCS+ with smaller maximum numbers of signatures. For some of the smallest parameter sets, the size of the CRV could be feasible. As one example, we estimate that the first set of signature parameters for 2^{20} signatures per key from [FD24] would require a CRV around 7 GiB and a dealer setup time of around 2^{37} hashes, which would be expensive but practical for some applications.

Impact. Threshold and distributed signatures offer a solution to the most vexing practical problem with using stateful HBS: the risk of reusing a one-time signing key. When the key is held and used by a single device, a single glitch or operator error can lead to a one-time key being reused, with disastrous results. When the key is held by *many* devices and signing involves running a signing protocol with (for example) three different devices, a key will be reused only if all three of these devices fail at the same time.

Additionally, many proposed applications of stateful HBS, such as firmware signing, can benefit from the ability to split control of the signing key among multiple trustees. Theft or compromise of the device holding the keys and leakage from the signing device immediately become much less likely, once the key is split into multiple shares on separate devices, each under a different person's control.

Future Work. In this paper, we have considered only Winternitz signatures. We believe it is straightforward to extend our techniques to Lamport signatures and FORS few-times signatures. Some novel techniques for shortening Winternitz signatures have been proposed in [KHRY22, HKRY23]; our techniques are compatible with their techniques.

We have considered security in the single-target setting. A multi-target setting would involve many composite key pairs $(\mathtt{CSK}_1, \mathtt{CPK}_1) \ldots (\mathtt{CSK}_\mu, \mathtt{CPK}_\mu)$. The adversary would make encryption queries with respect to all of the μ key pairs, and would win the game by presenting a forgery with respect to any $(\mathtt{CSK}_i, \mathtt{CPK}_i)$. Modern HBS schemes like LMS and XMSS incorporate a unique identifier for the composite public key into every hash computation in order to resist these attacks. Since our scheme produces public keys and signatures identical to the underlying stateful HBS scheme, we inherit the same multi-target resistance. We further conjecture that our full threshold and distributed signature scheme is also secure against multi-target attacks. For now, we leave proving (or disproving) this conjecture as an open problem.

Considering the work of Bansarkhani and Misoczki demonstrating group signatures for stateful HBS [BM18] combined with our work, it is clear that this venerable cryptographic primitive, which dates back to the very dawn of academic cryptography as a discipline, is capable of providing some surprising advanced functionality beyond simple signatures. We look forward to discovering what else may be possible to do with these schemes.

Acknowledgements

The authors wish to thank Marcus Almert, Daniel Apon, Dan Bernstein, Luís Brandão, Michael Davidson, Andreas Hülsing, Eik List, Carl Miller, Frank Piessens, Luigi Portwich, Bart Preneel, Vincent Rijmen, Nigel Smart, Vladimir Spassov, and the anonymous CiC reviewers for useful conversations and feedback about this work.

References

[AE17] Jean-Phillippe Aumasson and Guillaume Endignoux. Gravity-SPHINCS. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[AHS20] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. *IACR Cryptol. ePrint Arch.*, page 1390, 2020. URL: https://eprint.iacr.org/2020/1390.

[ASY22] Shweta Agrawal, Damien Stehlé, and Anshu Yadav. Round-optimal lattice-based threshold signatures, revisited. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.8.

[Ata23] Shahla Atapoor. Identity-based threshold signatures from isogenies. *IACR Cryptol. ePrint Arch.*, page 1459, 2023. URL: https://eprint.iacr.org/2023/1459.

[Ata24] Shahla Atapoor. Identity-based threshold signatures from isogenies. In Elizabeth A. Quaglia, editor, *Cryptography and Coding*, pages 220–240, Cham, 2024. Springer Nature Switzerland.

[BGG+18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 565–596. Springer, 2018. doi:10.1007/978-3-319-96884-1_19.

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 368–397, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-46800-5_15.

[BM18] Rachid El Bansarkhani and Rafael Misoczki. G-merkle: A hash-based group signature scheme from standard assumptions. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 441–463. Springer, 2018. doi:10.1007/978-3-319-79063-3_21.

[BMV19] Luís T. A. N. Brandão, Nicky Mouha, and Apostol Vassilev. Threshold schemes for cryptographic primitives: Challenges and opportunities in standardization and validation of threshold cryptography. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2019. URL: https://doi.org/10.6028/NIST.IR.8214.

[CAD+20] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. Recommendation for stateful hash-based signature schemes. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2020.

[CKM23] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully adaptive schnorr threshold signatures. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part I*, volume 14081 of *Lecture Notes in Computer Science*, pages 678–709. Springer, 2023. doi:10.1007/978-3-031-38557-5_22.

[CS19] Daniele Cozzo and Nigel P. Smart. Sharing the LUOV: threshold post-quantum signatures. In Martin Albrecht, editor, *Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings*, volume 11929 of *Lecture Notes in Computer Science*, pages 128–153. Springer, 2019. doi:10.1007/978-3-030-35199-1_7.

[Des88] Yvo Desmedt. Society and group oriented cryptography: A new concept. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 120–127, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany. doi:10.1007/3-540-48184-2_8.

[DF90] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. doi:10.1007/0-387-34805-0_28.

[DOTT22] Ivan Damgård, Claudio Orlandi, Akira Takahashi, and Mehdi Tibouchi. Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. *J. Cryptol.*, 35(2):14, 2022. doi:10.1007/s00145-022-09425-3.

[FD24] Scott R. Fluhrer and Quynh Dang. Smaller sphincs+. *IACR Cryptol. ePrint Arch.*, page 18, 2024. URL: https://eprint.iacr.org/2024/018.

[GKMN21] François Garillot, Yashvanth Kondi, Payman Mohassel, and Valeria Nikolaenko. Threshold schnorr with stateless deterministic signing from standard assumptions. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 127–156. Springer, 2021. doi:10.1007/978-3-030-84242-0_6.

[HBD+17] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS+. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[HBG+18] Hülsing, Butin, Gazdag, Rijneveld, and Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, RFC Editor, May 2018. URL: https://tools.ietf.org/html/rfc8391.

[HKRY23] Andreas Hülsing, Mikhail Kudinov, Eyal Ronen, and Eylon Yogev. Sphincs+c: Compressing sphincs+ with (almost) no cost. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1435–1453, 2023. doi:10.1109/SP46215.2023.10179381.

[HPS03] Javier Herranz, Carles Padró, and Germán Sáez. Distributed RSA signature schemes for general access structures. In Colin Boyd and Wenbo Mao, editors, *Information Security, 6th International Conference, ISC 2003, Bristol, UK, October 1-3, 2003, Proceedings*, volume 2851 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2003. doi:10.1007/10958513_10.

[Hül13] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013. doi:10.1007/978-3-642-38553-7_10.

[Int19] Internet Assigned Numbers Authority. Leighton-Micali Signatures (LMS) parameters. https://www.iana.org/assignments/leighton-micali-signatures/leighton-micali-signatures.xhtml, 2019. IANA Registry.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. URL: https://www.rfc-editor.org/info/rfc8032, doi:10.17487/RFC8032.

[JMV01] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001. doi:10.1007/s102070100002.

[KCLM21] Irakliy Khaburzaniya, Konstantinos Chalkias, Kevin Lewi, and Harjasleen Malvai. Aggregating and thresholdizing hash-based signatures using STARKs. Cryptology ePrint Archive, Paper 2021/1048, 2021. URL: https://eprint.iacr.org/2021/1048.

[KCLM22] Irakliy Khaburzaniya, Konstantinos Chalkias, Kevin Lewi, and Harjasleen Malvai. Aggregating and thresholdizing hash-based signatures using starks. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, pages 393–407. ACM, 2022. doi:10.1145/3488932.3524128.

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. Sha-3 derived functions: cshake, kmac, tuplehash and parallelhash. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2016.

[KHRY22] Mikhail Kudinov, Andreas Hülsing, Eyal Ronen, and Eylon Yogev. Sphincs+c: Compressing sphincs+ with (almost) no cost. Cryptology ePrint Archive, Paper 2022/778, 2022. https://eprint.iacr.org/2022/778. URL: https://eprint.iacr.org/2022/778.

[Lam79] Leslie Lamport. Constructing Digital Signatures from a One Way Function. Technical report, SRI, Oct 1979. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Constructing-Digital-Signatures-from-a-One-Way-Function.pdf.

[MCF19] McGrew, Curcio, and Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, RFC Editor, April 2019. URL: https://tools.ietf.org/html/rfc8554.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany. doi:10.1007/3-540-48184-2_32.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. doi:10.1007/0-387-34805-0_21.

[MSY99] Shingo Miyazaki, Kouichi Sakurai, and Moti Yung. On threshold rsa-signing with no dealer. In JooSeok Song, editor, *Information Security and Cryptology - ICISC'99, Second International Conference, Seoul, Korea, December 9-10, 1999, Proceedings*, volume 1787 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 1999. doi:10.1007/10719994_16.

[Nat13] National Institute of Standards and Technology. Digital signature standard (DSS). Technical Report Federal Information Processing Standards (FIPS) Publication 186-4, U.S. Department of Commerce, Washington, D.C., July 2013. doi:10.6028/NIST.FIPS.186-4.

[Nat16] National Institute of Standards and Technology. NIST post-quantum cryptography standardization, 2016. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[Nat23] National Institute of Standards and Technology. NIST round 1 additional signatures, 2023. URL: https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards (FIPS) Publication 204, U.S. Department of Commerce, Washington, D.C., August 2024. doi:10.6028/NIST.FIPS.204.

[Nat24b] National Institute of Standards and Technology. Stateless hash-based digital signature standard. Federal Information Processing Standards Publication 205, US Department of Commerce, Washington D.C., 2024. https://csrc.nist.gov/pubs/fips/205/final.

[NT24] Sela Navot and Stefano Tessaro. One-more unforgeability for multi - and threshold signatures. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 429–462. Springer, 2024. doi:10.1007/978-981-96-0875-1_14.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. URL: http://doi.acm.org/10.1145/359340.359342, doi:10.1145/359340.359342.

[Sho00] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 207–220, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany. doi:10.1007/3-540-45539-6_15.

[WWW+14] Yujue Wang, Duncan S. Wong, Qianhong Wu, Sherman S. M. Chow, Bo Qin, and Jianwei Liu. Practical distributed signatures in the standard model. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 307–326. Springer, 2014. doi:10.1007/978-3-319-04852-9_16.

[WWW+15] Yujue Wang, Duncan S. Wong, Qianhong Wu, Sherman S. M. Chow, Bo Qin, Jianwei Liu, and Yong Ding. Practical (fully) distributed signatures provably secure in the standard model. *Theor. Comput. Sci.*, 595:143–158, 2015. doi:10.1016/j.tcs.2015.06.029.

A Pseudocode Overview

Table 2: Overview of the functions used throughout this paper

Name	Purpose
Winternitz utility functions, Algorithm 1, Page 8	
$\texttt{ToString}(S)$	Returns the string representation of S
$\texttt{Split}_w(S)$	Splits the string S into w-bit chunks
Randomized Winternitz, Algorithm 2, Page 9	
$\texttt{Gen}(\texttt{KeyID})$	Generates public and secret key for Winternitz
$\texttt{Sign}_{\mathsf{SK}}(M, R, \texttt{KeyID})$	Creates a randomized Winternitz signature of M under the secret SK, randomizer R and $\texttt{KeyID}$
$\texttt{WINTER}(h, \mathsf{SK})$	Auxiliary function to return deterministic Winternitz signature of h
$\texttt{Verify}(R, Z, M, \texttt{KeyID})$	Verifies Winternitz signature (R, Z) for message M with $\texttt{KeyID}$; the result is the public key neccessary to pass verification
Merkle Tree Algorithm 3, Page 12	
$\texttt{MT_Construct}(X_{0\ldots D-1})$	Constructs a Merkle tree with D leafs $X_0, \ldots, X_{D-1}$ and returns $\texttt{ROOT}$
$\texttt{MT_MakePath}(X_{0\ldots D-1}, i)$	Creates $\texttt{PATH}$ from X_i to $\texttt{ROOT}$
$\texttt{MT_Verify}(\texttt{PATH}, X'_i)$	follows path from X'_i and returns $\texttt{ROOT}'$; if $X'_i = X_i$ then $\texttt{ROOT}' = \texttt{ROOT}$
$\texttt{MT_Extract}(\texttt{PATH})$	Gets index i from $\texttt{PATH}$
Generic Stateful Hash-based Signature Scheme, Algorithm 4, Page 14	
$\texttt{StatefulGen}(D)$	Generates key pair, $(\texttt{CPK}, \texttt{CSK})$ to sign D messages under D $\texttt{KeyID}$s
$\texttt{StatefulSign}_{\texttt{CSK}}(\texttt{KeyID}, M)$	Depends on the secret key $\texttt{CSK}$ and signs a message M under a given $\texttt{KeyID}$
$\texttt{StatefulVerify}_{\texttt{CPK}}(M, R, \texttt{PATH}, Z)$	Performs the matching verification process; returns 1 if verification succeeds
Setting up k-of-k one-time Signatures, Algorithm 5, Page 17	
$\texttt{KK_Setup}(K[1\ldots k], k, \texttt{KeyID}, \mathsf{SK}, R, \texttt{PATH})$	The trusted dealer sets up the $\texttt{CRV}$ using the given parameters
Setting up k-of-k one-time Signatures, Algorithm 6, Page 17	

KK_Aggregator_Sign(M, CRV, KeyID)	The k-of-k signing process as carried out by the aggregator
k-of-k Signing Process by Aggregator and Trustees, Algorithm 7, Page 18	
KK_Sign1(t, KeyID, M)	First round of the signing protocol
KK_GenSig1($K[t]$, KeyID)	First round steps of the signing process
KK_Sign2(t, R', CHK$'[t]$)	Second round of the signing protocol
KK_GenSig2($K[t]$, KeyID, h)	Second round steps of the signing process
KK_Auth($K[t]$, KeyID, R', CHK$'[t]$)	Checks if the input to the second round is OK
Setting up of threshold or distributed signature scheme, Algorithm 8, Page 23	
ShardSetup(D, k, CL)	The trusted dealer sets up a threshold or distributed signature scheme from a stateful HBS
Constructing an array of PRF keys, Algorithm 9, Page 24	
MakeKeyList(K, C)	Construct an array containing the PRF keys of each trustee in the coalition C
Aggregator Distributed Signature Generation, Algorithm 10, Page 24	
AggregatorSign(M, CRV, KeyID)	The aggregator takes a message M, the public CRV and the KeyID and then interacts with trustees to generate a signature for M.
Trustee Signing with the distributed HBS, Algorithm 11, Page 25	
TrusteeSetup(CL, K, k)	Setting up trustee t for the distributed signing protocol.
ShardSign1(t, KeyID, M)	Distributed signature protocol on trustee side: first message from trustee t
ShardSign2(t, R', CHK$'[t]$)	Distributed signature protocol on trustee side: second message from trustee t
PRF/RF Distinguishing Game, Algorithm 12, Page 27	
Init	Initializes key K and bit b and chooses RF from a set of all functions from n to n bits.
Query(x)	Return $F(K, x)$ or RF(x) depending on b
Final(b$'$)	Finalization
RA-HBS Forgery Game, Algorithm 13, Page 28	
Init(D)	Initializes used, compromised and CPK and CSK.
Compromise(KeyID)	Compromise KeyID
Sign(KeyID, M)	Sign message M by using KeyID
Final(L)	Present $s + 1$ signatures and win if they are distinct, valid signatures from uncompromised one-time keys.
IC-Forge Game, Algorithm 14, Page 30	
Init(k, D, CL, f)	Returns CPK, CSK and all PRF keys from compromised trustees.
Sign1(KeyID, M)	First part of signing protocol with honest trustee
Sign2(R$'$, CHK$'$)	Second part of signing protocol with honest trustee
Final(L)	Present count + 1 signatures and win if they are distinct, valid signatures from uncompromised one-time keys.
RA-HBS Simulator, Algorithm 15, Page 32	
Init(k, D, CL, f)	Returns CL, CPK, CSK and all PRF keys from compromised trustees.
Sign1(KeyID, M)	First part of signing protocol with honest trustee
Sign2(R$'$, CHK$'$)	Second part of signing protocol with honest trustee
Final(L)	Returns RA-HBS.Final(L)