Coalition and Threshold Hash-Based Signatures

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

[1] NIST
[2] COSIC/KU Leuven
[3] Bauhaus-Universität Weimar

Abstract. We show how to construct a threshold version of stateful hash-based signature schemes like those defined in XMSS (defined in RFC8391) and LMS (defined in RFC8554). Our techniques assume a trusted dealer and secure point-to-point communications; are efficient in terms of communications and computation; and require at least one party to have a large (but practical) amount of storage. We propose the addition of an untrusted Helper to manage the large storage required without being given access to any secret information. We prove the security of our schemes in a straightforward way, reducing their strength to that of the underlying hash-based signature scheme. Our schemes are quite practical, and substantially decrease the risk of accidental key reuse in hash-based signature schemes.

1 Introduction

Hash-based signatures[13, 15, 16] (HBS) are an old technology that is making a comeback due to uncertainty about which digital signature algorithms will ultimately resist attack by quantum computers. In recent years, several practical proposals for hash-based signatures have been published, includingSphincs[3], Gravity-Sphincs[1], and Sphincs+[11]. Additionally, NIST has published requirements for using stateful hash-based signatures safely[6].

Hash-based signatures are built from two different primitives:

1. A one-time signature scheme: a signature construction that allows only one signature per key; reusing a key leaks enough information to allow forgery of many other signatures with the key. (Some schemes, such as [18], allow a small number of signatures before leaking enough information to allow forgeries.)
2. A Merkle tree that contains a list of 2^d one-time keys, and allows an efficient proof that a given key is in the list. A Merkle tree has a single hash value (its root) that can be given to people to allow this proof.

Hash-based signatures consist of both a one-time signature on the actual message being signed and a Merkle tree path to prove that this one-time key is in the list of one-time keys owned by the user. In the rest of this article, we will refer to the collection of 2^k one-time keys, hashed by the Merkle tree, as a *composite key*. The composite public key is the root of the Merkle tree.

Practical hash-based signature schemes can be classified into *stateful* schemes, where the signer keeps track of which one-time keys have been used so far, and *stateless* schemes, where the signer chooses a one-time (or few-time) key at random for each signature. Both kinds of hash-based signatures have a maximum number of signatures allowed for a given composite public key.

A *threshold signature*[7, 8, 19] is a signature scheme in which the private key is split among n trustees, and some subset of k trustees (where $2 \leq k \leq n$) must work together to sign a message. We refer to such a subset of k trustees as a *quorum*. Some threshold signature schemes require a *trusted dealer* to generate and split up the signing key into shares; others can avoid this necessity. Our schemes require a trusted dealer.

Typical proposals for threshold signature schemes have been based on an underlying signature scheme, such as ElGamal or RSA. The threshold property is founded on exploiting homomorphic properties of the underlying signature scheme. Hash-based signature schemes lack such convenient homomorphic properties. To the best of our knowledge, hash-based signature schemes have not yet been considered as an underlying signature scheme for threshold signature schemes, though in [2], stateful hash-based group signatures were introduced. Their technique is similar to the "sharding" used in our schemes, as described below.

1.1 Our Results

Starting with an existing stateful hash-based signature scheme, such as LMS or XMSS, we provide practical techniques to split the private key material among n trustees, in order to construct the following mechanisms:

1. An n-of-n signature scheme, in which all n trustees must cooperate to create a signature.
2. A k-of-n signature scheme, in which any k of the n trustees may cooperate to create a signature.
3. A *coalition* signature scheme, in which any set of coalitions of trustees defined when the scheme was set up can cooperate to create a signature, but no signature can be formed by any subset of trustees that does not make up (or contain) a coalition.

In all three cases, the resulting signatures are verified in the same way as ordinary hash-based signatures– the exact same verification code for an existing HBS scheme can be used for verifying our signatures. In fact, there is a trusted instance, the "dealer", who generates the public key and the data for the trustees and could just as well serve as a single signer, without involving the trustees. Our scheme could thus also be viewed as a proxy signature scheme, where the holder of secret signing keys delegates[4] signing capabilities to a group of trustees, in which the delegator (aka dealer) D defines what subsets of the trustees are required to create valid signatures. The verifier will not observe any difference, since signatures generated by D itself are identical to signatures jointly generated by groups of trustees appointed by D.

We describe our techniques for Winternitz signatures (as in W-OTS+[10], XMSS[5, 9], and LMS[14]), but they can be easily applied to Lamport signatures as well.

Our techniques are efficient in computation and communications, but they require substantial memory. In order to make the scheme more practical, we introduce the notion of the *Helper*. The Helper can be implemented as a participant in the protocol, but can also be implemented via a cloud storage service, shared access to a large set of files, or even a physically distributed DVD or other large memory store. In its most general form, the Helper is just a large constant public bitstring.

All of our techniques require a trusted dealer to generate and distribute the initial shares of the scheme. (A natural case where this is reasonable is when the owner of a signing key sets up a threshold scheme to protect the key from loss or compromise in the future.) As a consequence of the need for a trusted dealer, our techniques will not work with tree-of-tree variants of these signature schemes[4], nor with stateless hash-based signatures.

For a given number of signatures that must be available to each coalition, the setup time, Helper storage requirements, and signature length in our schemes all scale linearly in the total number of coalitions that can sign a message. In an n-of-n scheme, there is only one coalition, no matter how many trustees there are. In a k-of-n scheme, there are $\binom{n}{k}$ coalitions. This imposes practical limits to the choices of k, n that our scheme can support, but still allows a wide range of such choices of practical use. For example, a 3-of-5 threshold or 2-of-45 threshold signature scheme are quite practical.

The security of our n-of-n, k-of-n, and coalition signatures directly directly relate to the security of the underlying hash-based signatures: An attacker who can forge a signature in our scheme without access to a full coalition of trustees can also forge signatures in the underlying hash-based signature scheme with only negligibly lower probability of success. (See Appendix B.).

1.2 Impact

Our techniques potentially solve the greatest risk associated with stateful hash-based signatures: the possibility of accidentally reusing a one-time signing key due to some attack, hardware failure, or other error. When a coalition of k trustees agree to sign a message, they must all sign with the same key[5]; no honest trustee will go along with reusing a key. Instead of a single device failure leading to disaster, multiple devices must fail in the same way at the same time to lead to disaster.

Additionally, many widely-discussed applications of stateful hash-based signatures, such as signing firmware updates for long-lived systems, will benefit from both the security and the error-tolerant

[4] Note that the delegator would have to take care to avoid reuse of any signing keys.

[5] Reusing a one-time key in a stateful hash-based signature scheme potentially allows an attacker to sign arbitrary messages with that key.

properties of threshold signatures. For example, using a 3-of-5 threshold signature scheme in a high-value firmware-update-signing application means that a damaged or compromised signing device need not lead to disaster.

1.3 Guide to the Paper

The rest of the paper is organized as follows: Section 2.5 introduces the notation we use, and defines some conventions. In Section 3, we show how to construct n-of-n stateful hash-based signatures. In Section 4, we extend this to k-of-n threshold signatures and more general coalition signatures. Coalition signatures are a generalization of threshold signatures that natively support a wide range of different signing coalitions rather than just k-of-n coalitions. In Section 6, we summarize our results and discuss possible future work. Appendix B provides a formal analysis of our security claims.

2 Notation and Conventions

2.1 Signatures, Coalition Signatures, and Threshold Signatures

A *signature scheme* $(\mathbf{generate}, \mathbf{sign}, \mathbf{verify})$ consists of three algorithms:

1. a randomized key generation algorithm **generate**, which chooses a random key pair $(X, Y) = \mathbf{generate}()$, where Y is the public key, and X the secret key.
2. a signing algorithm, **sign**, which, given the secret key X and a message M, computes a signature $\sigma = \mathbf{sign}(X, M)$,
3. and a verification algorithm **verify**, which takes the public key Y, the message M, and the signature σ and either accepts or rejects the triple (Y, M, σ).

The signature scheme is *sound*, if and only if

$$\mathbf{verify}(Y, M, \mathbf{sign}(X, M)) = \text{True}$$

for all messages M and all matching secret/public key pairs (X, Y).

Assume the adversary knows the public key Y, but not the secret key X. Also allow the adversary to choose messages M_1, M_2, ..., and receive signatures σ_1, σ_2, ..., with $\sigma_i = \mathbf{sign}(X, M_i)$. The signature scheme is *secure* (w.r.t. chosen messages), if the adversary cannot feasbibly find a pair $(M, \sigma) \notin \{(M_1, \sigma_1), (M_2, \sigma_2), \ldots\}$, such that $\mathbf{verify}(Y, M, \sigma)$ holds.

When running a signature scheme, at least two parties are involved, a *signer* and a *verifier*. The signer knows the secret key Y and, when willing to sign a message M, creates the signature $\sigma = \mathbf{sign}(X, M)$. Typically, the signer will also be the party that has run the key generation algorithm, though it may also delegate this task to some trusted party. The verifier is given the public key Y, a message M, and the signature σ. The verifier is thus able to run the verification algorithm.

For threshold and coalition signatures, the signer is split up into different parties: a *dealer* and n *trustees*. The dealer runs the key generation algorithm and splits the secret key X into *shares* $X_1, \ldots, X_n$, one share for each trustee. In this paper, we assume an honest dealer[6]. The signing algorithm **sign** is implemented as a joint protocol to be exectued by one or more trustees.

For $k \leq n$, a k-of-n *threshold signature scheme* allows any subset of at least k trustees to run the signing protocol and create a valid signature.[7] The security requirement is that for any group of less than k trustees, it is infeasible to create a valid signature.

A *coalition signature scheme* is a generalized threshold signature scheme. I.e., assuming a set $\mathcal{C}$ of *coalitions* $c \in \mathcal{C}, c \subseteq \{1, \ldots n\}$, any coalition $c \in \mathcal{C}$ of trustees (or a superset of c, cf, Footnote 7) can create a valid signature, while no group of trustees that is not a superset of a coalition can do so. For

[6] A natural situation is that the dealer is the owner of the key, setting up a coalition or threshold scheme to delegate its signing capability to subsets of trustees, and to protect the key from loss or compromise in the future. Being the owner of the key, the dealer has no incentive to cheat.

[7] In the current paper, all signing protocols will assume *exactly* k trustees. However, if there are more than k trustees, nothing stops any additional trustees from taking part in running the signing protocol by doing nothing at all.

example, Alice may be allowed to sign with either Bob, Carol, or Dave. Alternatively, Bob, Carol and Dave could jointly sign without Alice. But when only two of Bob, Carol and Dave join to sign, they cannot feasibly create a valid signature; neither can Alice on her own. If we identify Alice, Bob, Carol, and Dave by 1, 2, 3, and 4, respectively, then valid coalitions are $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, and $\{2,3,4\}$.

2.2 Notation

We introduce the following general notation:

- α is used throughout the paper to generically represent any variable that might be split into shares, randomly sampled, sent between trustees, etc.
- A bitstring α of length ℓ is an element $\alpha \in \{0,1\}^\ell$. We write ϵ for the bitstring of length 0.
- If $\alpha \in \{0,1\}^\ell$ and $\beta \in \{0,1\}^m$, then $\alpha \parallel \beta$ is the concatenation of α and β and thus in $\{0,1\}^{\ell+m}$.
- $\alpha \leftarrow\!\$\, \{0,1\}^\ell$ means that α gets assigned an ℓ-bit string chosen uniformly at random. When $\alpha = \alpha_0, \ldots, \alpha_{\ell-1}$ is a bitstring of length ℓ, we interchangeably interpret α as an integer $\alpha = \sum_{0 \le i < \ell-1} 2^{\ell-i}\alpha_i \in \{0, \ldots, 2^\ell - 1\}$, with α_0 the most significant (i.e., leftmost) bit, and $\alpha_{\ell-1}$ the least significant bit.
- $\hat{\alpha}$ = A transmitted value for α that may or may not be correct.
- α^t is trustee t's share of α. Superscripts in this paper usually refer to shares, rather than to exponentiation.
- $\alpha^{\neq t}$ = All trustees' shares of α except trustee t's share.
- α^H is the share of α held by the Helper.
- $\alpha^\oplus$ is the XOR of shares $1 \ldots n$ (that is, the XOR of all shares *except* that of the Helper).
- α^+ = The XOR of all shares of α held by the attacker.
- α_π = is the instance of α defined for one-time key π.
- α_{all} = are the instances of α defined for all one-time keys in the composite key.
- When A is an array, $A[i][j]$ is the i,jth component of the array.
- $\texttt{PRF}_K(x,n)$ is a pseudorandom function that takes key K and input x, and produces an n-bit pseudorandom bit string as output. See Appendix B for the formal definition of a PRF used in this paper.
- An input to PRF or hash is sometimes written as a tuple, such as (a,b,c), where the values included may be integers or bitstrings. We assume that the tuple is encoded in an unambiguous way into a bitstring for PRF or hash to process.

The following notation is used specifically in hash-based signatures.

- $h = \texttt{hash}(M)$ is the ℓ-bit hash value of a message $M \in \{0,1\}^*$.
- $h = \texttt{hash}(\rho, M)$ is the ℓ-bit hash value of prefix $\rho \in \{0,1\}^*$ and message $M \in \{0,1\}^*$
- ℓ is the output length of hash, in bits.
- A *one-time* signature scheme is a triple (**generate**, **sign**, and **verify**) of algorithms, see Section 2.3.
- A Merkle tree is a balanced binary tree with leaves V_π. See Section 2.4.
- $\pi \in \{0,1\}^d$ is the unique identifier of a one-time signing key. Equivalently, it is the index of a leaf in a depth-d Merkle tree whose root is the public key of the hash-based signature scheme.
- $\texttt{PATH}_\pi$ is the Merkle-tree path to one-time verification key π.
- $\sigma_\pi(h)$ is the one-time signature of hash h under one-time key π. Sometimes as a shortcut, we write $\sigma_\pi(M)$, where $h = \texttt{hash}(M)$.
- The full signature is usually written as $\texttt{PATH}_\pi, \sigma_\pi(h)$.
- For Winternitz signatures, we have:
  - w = the number of bits encoded in each hash chain
  - a = the number of w-bit components needed to encode the hash
  - c = the number of w-bit components needed to encode the checksum
  - C = the maximum value of the checksum
- X_π is a two-dimensional array containing the private one-time signing key π. For Winternitz signatures, note that $X_\pi[i][0]$ is a (pseudo)random starting value, while for $j > 0, X_\pi[i][j] = \texttt{hash}(X_\pi[i][j-1])$.

When discussing our threshold hash-based signatures and semi-blind signatures, we use some additional notation and terminology:

- For a threshold scheme, a *dealer* is the entity that sets up the scheme initially. A *trustee* is one of the entities allowed to hold a *share* of the signing key. A *quorum* of trustees is a set of trustees sufficient to construct a signature. See Section 2.1. The *Helper* is an additional trustee that is sometimes used, with high memory requirements and limited trust.
- n is the number of trustees. Trustees are numbered from 1, so that when there are three trustees, they are trustees 1, 2, and 3.
- t is often used to denote a particular trustee. In the security analysis, t is used to indicate the one uncompromised trustee.
- $\sigma_\pi^t(h)$ is the t-th share of the signature $\sigma_\pi(h)$. $\sigma^t(h)$ is produced by signing the hash h with trustee t's share of the one-time signing key. $\sigma_\pi(h) = \bigoplus_{\text{all } t} \sigma_\pi^t(h)$

2.3 One-time Signatures

A *one-time* signature scheme $(\mathbf{generate}, \mathbf{sign}, \mathbf{verify})$ consists of three algorithms:

1. a randomized key generation algorithm **generate**, that chooses a random key pair $(X_\pi, Y_\pi) = \mathbf{generate}(\pi)$, where the number π serves as an identifier for the key, Y_π denotes the public key, and X_π the secret key,
2. a signing algorithm, **sign**, which, given the secret key X_π and a value $h \in \{0,1\}^\ell$, computes a signature $\sigma_\pi = \mathbf{sign}(X_\pi, h)$,
3. and a verification algorithm **verify**, which takes the public key Y_π, the hash h, and the signature σ_π and either accepts or rejects the triple (Y_π, h, σ_π). The signature scheme is sound, if and only if

$$\mathbf{verify}(Y_\pi, h, \mathbf{sign}(X_\pi, h)) = \text{True}$$

for all hashes h and all matching secret/public key pairs (X_π, Y_π).

We require a one-time signature scheme $(\mathbf{generate}, \mathbf{sign}, \mathbf{verify})$ to be secure against a chosen-h existential forgery attacks: The adversary is given Y_π, then the adversary chooses *one single* value $h \in \{0,1\}^\ell$ and requests $\sigma = \mathbf{sign}(X_\pi, h)$ from the challenger. It is then required to be infeasible for the adversary, to find any $(h', \sigma') \neq (h, \sigma)$, with $\mathbf{verify}(h', \sigma') = Y_\pi$.

In the current paper, the secret key X_π is actually a two-dimensional array of components, and we will address a single component as $X_\pi[i][j]$.

To sign arbitrary messages $M \in \{0,1\}^*$, we will sign and verify the corresponding message hash $h = \texttt{hash}(M)$. If the hash function `hash` is collision resistant and the one-time signature scheme is secure as required above, than this new scheme is secure against chosen-message existential forgery attacks.

The one-time signature schemes we consider in the current paper have the following properties:

- The verification procedure can be tweaked by taking only two inputs: the hash h and the signature σ_π. The public key Y_π, which would be the other input for the general verification procedure, is actually computed as $\hat{Y}_\pi = \mathbf{verify}(h, \sigma_\pi)$. Verification succeeds if $\hat{Y}_\pi = Y_\pi$. That is, the signature scheme is sound if and only if
$$\mathbf{verify}(h, \mathbf{sign}(X_\pi, h)) = Y_\pi.$$
- Our one-time signatures are hash-based, and we will in general use the same hash function `hash` that we also employ to hash messages of arbitrary length to a fixed length of ℓ bits.[8]
- If we use one secret key X_π to sign two different hashes h and $h' \neq h$, it actually becomes easy for the adversary to generate an existential forgery (except, possibly, when the Hamming difference between h and h' is exactly 1).
- The **sign** algorithms we consider are deterministic. Thus, we are free to sign the same h more than once, without endangering the security of our one-time signature scheme.

In this paper, we will focus on Winternitz one-time signatures, since this is the most commonly used hash-based signature scheme. However, we note that our techniques are easily adapted to Lamport signatures as well.

For the hash-based one-time signature schemes that we consider in this paper, the *private key* consists of a set of secret components, and the *public key* consists of the result of hashing those secret components

[8] Note that the requirements for hashing arbitrary-length messages and for other applications of hashes in a hash-based signature scheme are somewhat different.

in some way. *Signing* a message involves hashing the message and using the bits of the hash to choose which secret components to reveal. *Verifying* a message involves hashing the message and using the bits of the hash to determine which secret components should have been revealed, then hashing them to verify that the signature was formed correctly.

Memory Requirements A naive implementation of Winternitz signatures would require a lot of storage for the one-time keys. For example, a typical parameter set for Winternitz signatures would require about 32 KiB; storing 2^{16} one-time keys would require around 2 GiB.
Instead of generating these values randomly, they are typically generated using a pseudorandom function such as HMAC[12]. This allows the signer to regenerate each one-time signing key on the fly as needed. For example, instead of setting

$$X[i][0] \leftarrow\$ \{0,1\}^{256}$$

we can set

$$X[i][0] \leftarrow \mathtt{PRF}_K((\pi, 1, i, j), 256)$$

This permits the signer to generate and store only the values it needs for a given signature. Any practical implementation of a hash-based signature scheme will use some variant of this technique to save memory.

Random Prefixes A signature is always computed on a hash, h, typically as the hash of a message M: $h \leftarrow \mathtt{hash}(M)$. This requires `hash` to be collision resistant. As an alternative option, with fewer requirements on the hash function, we consider a prefix ρ, which must be unpredictable for the party choosing M. The signature is computed on $h \leftarrow \mathtt{hash}(\rho, M)$, where `hash` somehow incorporates ρ into the hashing of M in a way that provides security even when `hash` is vulnerable to collision attacks. (While we refer to ρ as a prefix, many ways exist to incorporate the prefix into the hash–notably LMS and XMSS use quite different techniques. We will use the notation $\mathtt{hash}(\rho, M)$ for any such technique.)
Note that generating ρ in a threshold signature needs to be done with care to avoid losing the additional security guarantees it provides. A technique for doing so is described below.

2.4 Multi-Time Signatures from One-Time Signatures and Merkle Trees

A one-time signature scheme isn't very useful in practice. To create many signatures (up to 2^d), we combine one-time signatures with a balanced binary tree[9] of depth d. Every leaf $V_\pi = \mathtt{hash}(Y_\pi)$ with index $\pi \in \{0, \ldots, 2^d - 1\}$ is the hash of a public one-time key Y_π. The internal nodes are defined recursively; for $x \in \bigcup_{i \in \{0,\ldots,d-1\}} \{0,1\}^i$,

$$V_x = \mathtt{hash}(V_{x\|0} \parallel V_{x\|1})$$

with the root $V = V_\epsilon = \mathtt{hash}(V_0 \parallel V_1)$. The three algorithms of a signature scheme are the following:

- The composite key generation algorithm generates the 2^d one-time key pairs $(Y_0, X_0), \ldots, (Y_{2^d-1}, X_{2^d-1})$ and the corresponding Merkle tree. The composite public key $\mathtt{PK} = V$ is the root of the Merkle tree. The composite secret key consists of all the secret one-time keys $X_0, \ldots, X_{2^d-1}$.
- The composite signature generation for hash $h \in \{0,1\}^\ell$ consists of the following steps:
  1. pick a previously unused[10] index $\pi \in \{0, \ldots, 2^d - 1\}$,
  2. compute $\sigma_\pi(h) = \mathbf{sign}(X_\pi, h)$, and
  3. compute the Merkle-tree path

$$\mathtt{PATH}_\pi = \mathtt{PATH}_{b_0,\ldots,b_{d-1}} = \Big(\pi, V_{b_0,b_1,\ldots,b_{d-2},(1-b_{d-1})},\ \ldots,\ V_{b_0,(1-b_1)},\ V_{(1-b_0)}\Big).$$

[9] A more general definition would also consider a-ary and unbalanced trees – but we don't need this in our context.
[10] All composite schemes we consider are stateful, i.e., the signer knows which secret keys $X_{\pi'}$ have already been used, and which are currently unused.

Note that $\texttt{PATH}_\pi$ holds the information to compute the hashes on the path from the leaf V_π to the root V. Computing the hashes and then comparing the result to V verifyies that V_π is the leaf at position π in the Merkle-tree with root V.

The signature is the pair $S_\pi = (\sigma_\pi(h), \texttt{PATH}_\pi)$.[11]

- The composite verification procedure takes the public key V (the root of the Merkle tree), the hash h and the signature $S_\pi = (\sigma_\pi(h), \texttt{PATH}_\pi)$. It performs the following two operations:
  1. Call the one-time verification procedure to generate $Y_\pi = \textbf{verify}(h, \sigma_\pi)$,
  2. and then accept if $V_\pi = \texttt{hash}(Y_\pi)$ is the leaf at position π in the Merkle tree with root V.

Stateful hash-based signatures are constructed from a Merkle tree whose leaves are all one-time signature keys. We refer to the full set of these one-time keys as the *composite key*, and the root of this Merkle tree as the *composite public key*, written as PK.

The following lemma is almost trivial but useful:

Lemma 1. *When signing any two messages with hashes h and h' using different keys π, π', the corresponding signatures S_π and $S_{\pi'}$ will be different: $S_\pi \neq S_{\pi'}$. This holds even if $h = h'$.*

Proof. When signing a message, we pick a previously unused π, so that $\pi \neq \pi'$ holds. Consequently $\texttt{PATH}_\pi \neq \texttt{PATH}_{\pi'}$ and $S_\pi \neq S_{\pi'}$.

2.5 Diagrams

We use diagrams to explain protocols, security definitions, and security reductions. Our protocols often involve a single entity sending or receiving messages to/from several trustees in parallel. We use the following conventions to convey this in our diagrams:

1. Our n-of-n, k-of-n, and coalition signatures always start from a single *initiating trustee*. The *responding trustees* are shown in a single column. For readability, we always make trustee 1 the initiating trustee. However, any trustee can take that role.
2. Our signature protocols involve sending the same message from the initiating trustee to the responding trustees, and the initiating trustee receiving very similar messages (shares of some secret value) back from each. Messages are assumed to go over a private, authenticated, replay-protected point-to-point channel, using pre-shared symmetric keys between the trustees. However, we do *not* assume a broadcast channel; Trustee #2 has no way to know what message Trustee #3 has received.
3. Messages are shown as single arrows. When it isn't clear, the text under the arrow may specify when this message is going to/coming from many different trustees at once.

3 Stateful Hash-Based n-of-n Signatures

In this section, we explain the techniques to design hash-based threshold and coalition signatures in the following steps:

1. We show how to split a single one-time signing key into n shares, and how to do n-of-n Winternitz signatures using those shares.
2. We define a space-inefficient but simple way to construct an n-of-n version of a stateful hash-based signature scheme.
3. We refine the system into a more practical one, minimizing the memory requirements on trustees while adding a new role–the "Helper"–to handle memory requirements without needing to be trusted.

[11] Technically, there are two "signatures": The full signature S_π from the multi-time signature scheme and the one-time signature σ_π from the underlying one-time signature scheme. Also note that the **verify** procedure from Winternitz actually computes the one-time public-key $Y_\pi = \textbf{verify}(h, \sigma_\pi)$.

3.1 Splitting One-Time Signing Keys

As discussed above, a stateful HBS consists of two elements: A one-time signature scheme and a Merkle tree of one-time keys. Each one-time signing key may sign a single message securely; if a one-time key is ever used to sign two different messages, an attacker may be able to forge signatures from that one-time key.

For Winternitz signatures, the one-time private key consists of an array of hash chains. These can be represented as a two-dimensional array of secret values, $X[0 \ldots a-1][0 \ldots 2^w - 1]$. Each $X[i][0 \ldots 2^w - 1]$ consists of an array of 2^w values, constructed so that

$$X[i][j] = \begin{cases} \text{A random starting value} & j = 0 \\ \texttt{hash}(\text{metadata}, X[i][j-1]) & \text{otherwise} \end{cases}$$

In the rest of this paper, we ignore this structure, and simply treat each one-time signing key as a two-dimensional array.

In fully specified HBS standards, metadata is incorporated into each hash function call, to prevent multi-target attacks. Our techniques are compatible with a wide variety of ways this metadata might be specified and incorporated into the hash, as with the LMS and XMSS standards. We ignore these details here.

Suppose we have a one-time signing key, represented as a two-dimensional array, X. We can split a one-time private key into n shares in a very straightforward way: For every possible value of i, j, we choose the shares of trustee t, $X^t[i][j]$, so that

$$X[i][j] = \bigoplus_{\text{all } t} X^t[i][j]$$

For compactness, we can write X to mean the whole two-dimensional array of secret values (the whole one-time signing key) and X^t as the whole two-dimensional array of shares held by trustee t. If we want to specify *which* one-time private key we're discussing, we will subscript it: X^t_π. X^t_{all} represents trustee t's shares of *all* the one-time signing keys in the composite key.

3.2 `S_Sign`: Signing with Shares

Suppose we have constructed t shares of the one-time private signing keys so that they XOR together to produce the original one-time private keys. We can now define the `S_Sign` algorithm. This algorithm takes a *share* of the one-time signing key and the hash of the message to be signed and returns a *share* of the signature. The shares of the signature can be recombined into the full signature by XORing them all together.

In Algorithm 1, we show the `S_Sign` algorithm, alongside the native signing algorithm. Note that the algorithms are *identical* except for the inputs–`S_Sign` gets a share of the one-time key, while the native signing algorithm gets the full one-time key.

As described above, we represent Winternitz signatures as an array of $a + c$ values, where a is the number of hash chains used to encode the Winternitz hash, and c is the number used to encode the checksum. For example, when $\ell = 256$, typical values for $w = 4$ would be $a = 64, c = 3$; the resulting signature consists of 67 ℓ-bit strings. The one-time private key consists only of the two-dimensional array X. In practice, implementations of Winternitz signatures *compute* the entries in this two-dimensional array as needed rather than storing them in an array.

When the n shares of the signature are recombined,

$$\sigma_\pi(h) \leftarrow \bigoplus_{\text{all } t} \texttt{S\_Sign}(X^t_\pi, h)$$

we get a normal Winternitz signature, that can be verified by existing code. We can prove correctness in a straightforward way.

1: **function** `S_Sign`(X_π^t, h)
// *t = which trustee this is, 1..n*
// *$h = \texttt{hash}(message), a = \lceil \ell/w \rceil$*
// *$C = a \times (2^w - 1), c = \lceil \lg(C)/w \rceil$*
2: split$_w$ h into $b[0 \ldots a-1] \leftarrow h$
3: **for** $i \leftarrow 0 \ldots a-1$ **do**
4: $\sigma_\pi^t[i] \leftarrow X_\pi^t[i][b[i]]$
// *Compute Checksum*
5: $S \leftarrow C - \sum_{i=0}^{a-1} b[i]$
6: $d[0 \ldots c-1] \leftarrow S$ split into w-bit chunks
7: **for** $i \leftarrow 0 \ldots c-1$ **do**
8: $\sigma_\pi^t[i+a] \leftarrow X_\pi^t[i+a][d[i]]$
9: **return** $\sigma_\pi^t(h)$

1: **function** Sign(X_π, h)
// *Full Winternitz signature*
// *$h = \texttt{hash}(message), a = \lceil \ell/w \rceil$*
// *$C = a \times (2^w - 1), c = \lceil \lg(C)/w \rceil$*
2: split$_w$ h into $b[0 \ldots a-1] \leftarrow h$
3: **for** $i \leftarrow 0 \ldots a-1$ **do**
4: $\sigma_\pi \leftarrow \sigma_\pi \parallel X_\pi[i][b[i]]$
// *Compute Checksum*
5: $S \leftarrow C - \sum_{i=0}^{a-1} b[i]$
6: $d[0 \ldots c-1] \leftarrow S$ split into w-bit chunks
7: **for** $i \leftarrow 0 \ldots c-1$ **do**
8: $\sigma_\pi[i+a] \leftarrow X_\pi[i+a][d[i]]$
9: **return** $\sigma_\pi(h)$

Algorithm 1: Winternitz threshold signing algorithm vs original signing algorithm

Lemma 2 (`S_Sign` Lemma). *Let X be the one-time private signing key for a Winternitz signature and $X^{1\ldots n}$ be shares of the private key such that, for all i, j*

$$X[i][j] = \oplus_{k=1}^{n} X^k[i][j]$$

Then for any hash value h,

$$\sigma_\pi(h) = \oplus_{k=1}^{n} \texttt{S\_Sign}(X_\pi^k, h)$$

Proof. Given a one-time signing key X_π, a Winternitz signature consists of revealing a subset of components of X_π. By definition, the shares $X_\pi^{1\ldots n}$ satisfy

$$X[i][j] = \bigoplus_{\text{all } t} X^t[i][j]$$

For each component of X_π revealed by the Winternitz signing algorithm, the corresponding share of that component is revealed by `S_Sign`. Thus,

$$\sigma_\pi(h) = \oplus_{k=1}^{n} \texttt{S\_Sign}(X_\pi^k, h)$$

□

For compactness, in the rest of this paper, we will often write $\sigma_\pi^t(h)$ for $\texttt{S\_Sign}(X_\pi^t, h)$. We will also sometimes write $\sigma_\pi(M)$ or $\sigma_\pi^t(M)$ instead of the more correct but longer $\sigma_\pi(h)$ or $\sigma_\pi^t(h)$, where $h = \texttt{hash}(M)$ or $h = \texttt{hash}(\rho, M)$. When the hash and message being signed is clear from context, we will sometimes simply write σ for the one-time Winternitz signature, or σ^t for trustee t's share of the one-time Winternitz signature.

3.3 Multiple Keys

`S_Sign` allows us to do an n-of-n signature for a single one-time key. However, a stateful HBS scheme requires the ability to do this for *many* keys and to produce a Merkle tree path for the one-time public key whose corresponding private key is used to sign a message.

In order to get a secure n-of-n stateful hash-based signature scheme, we need each trustee to keep track of which keys it has used and to refuse to reuse any key[12]. In the rest of this paper, unless we say otherwise, we will assume that the keys are used in a completely rigid, sequential order.

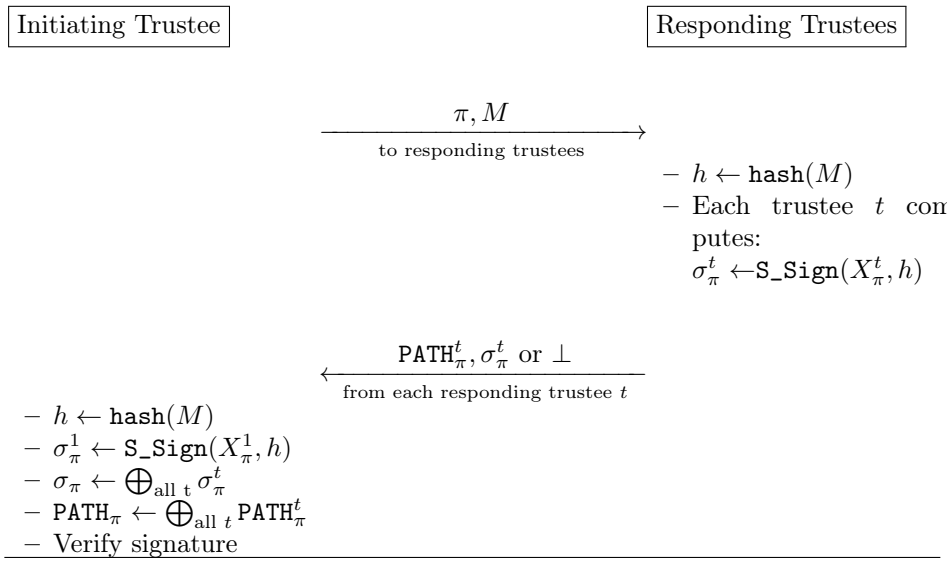
Producing the paths for each signing key in a standard HBS scheme is done by recomputing parts of the Merkle tree as needed, and a fair bit of cleverness has gone into efficient algorithms for doing so. (For example, see [20].) However, this can't be done by trustees holding shares of keys. A simple solution is simply to have each trustee store an XOR share of the Merkle tree path for each key in the tree[13].

Let π be the index used to indicate the current one-time key. Then, X_π is the private one-time signing key for π (and is a two-dimensional array of values), $\sigma_\pi(h)$ is the signature of hash value h under one-time key π, and $\texttt{PATH}_\pi$ is the Merkle-tree path of the corresponding one-time public key. We indicate shares of all these by a superscripted number or variable: $X^t_\pi, \texttt{PATH}^t_\pi$, and $\sigma^t_\pi(h)$ are trustee t's share of one-time private key π, the Merkle tree path of key π, and the signature under key π of h.

This lets us specify the full n-of-n signing protocol, given that the private keys and paths have somehow been split out into n shares that all XOR back together into their original values.

The process for producing an n-of-n signature appears in Protocol 1.

For ease of exposition, we assume that the initiating trustee is trustee 1, and that the responding trustees agree to sign the message. In reality, any trustee may initiate a signature, and the responding trustees may refuse a signature (returning $\perp$ to the initiating trustee) either because they are unwilling to sign the proposed message or because they believe the key-id has already been used. The initiating trustee must verify the resulting signature before relying on it being correct.



Protocol 1: n-of-n Signature Protocol

[12] In a real-world system, there also would need to be a mechanism for recovering after the trustees get out of synch. We will mostly ignore this engineering detail for the rest of this paper and assume that the trustees in the n-of-n scheme are always in agreement about which key they are going to use next. Note that in our protocols, the key-id to be used is included in the initial message of the protocol, so the trustees can quickly determine when they are out-of-synch.

[13] Having each trustee store a share of the Merkle tree path also simplifies the security proof for these schemes, as discussed below.

3.4 Generating the Shares: PRFs and the Helper

In our scheme, we assume a trusted dealer, who generates the initial set of one-time keys, constructs the Merkle tree for each one-time key, and distributes shares of the key material to the trustees.

Approach to Generating Shares The simplest way to generate the shares for a value α is to make the first $n-1$ shares random, and the final share the XOR of all other shares XORed with α. That is, we could do something like this for a value α that is to be split into n shares:

$$\alpha^t \leftarrow\$ \{0,1\}^\ell \text{ for } t \leftarrow 1 \ldots n-1$$
$$\alpha^n \leftarrow \alpha^1 \oplus \ldots \oplus \alpha^{n-1} \oplus \alpha$$

This would work, but would impose heavy memory requirements on the trustees. With commonly-used Winternitz parameters, a single one-time key would require about 17 KiB, and a collection of 2^{16} one-time keys would require about 2 GiB.

In existing HBS implementations, private keys are typically generated using a PRF call. We could use the same trick. Then, to split a value α into n shares, we could do:

$$\alpha^t \leftarrow \mathtt{PRF}_{K[t]}(\text{label for } \alpha) \text{ for } t \leftarrow 1 \ldots n-1$$
$$\alpha^n \leftarrow \alpha^1 \oplus \ldots \oplus \alpha^{n-1} \oplus \alpha$$

Again, this would work. And $n-1$ trustees would escape the memory requirement. However, the nth trustee would still require a large amount of storage. A practical system can be built this way, but we prefer a different approach–one that allows all trustees to be identical. If we do, then we get an additional output from splitting our values into shares, which we will call the *Helper shares*.

$$\alpha^t \leftarrow \mathtt{PRF}_{K[t]}(\text{label for } \alpha) \text{ for } t \leftarrow 1 \ldots n-1$$
$$\alpha^H \leftarrow \alpha^1 \oplus \ldots \oplus \alpha^n \oplus \alpha$$

We can think of the Helper shares as a kind of auxillary public bitstring that results from the process of splitting the signing key into shares. As long as the initiating trustee has access to that public bitstring, it can run the n-of-n signature protocol described above.

Generating Pseudorandom Shares In the rest of this paper, we use the convention of providing the PRF (and sometimes other functions) a *tuple* as input, such as $(1,\pi)$ or $(2,\pi,i,j)$. We assume that this tuple is encoded in an unambiguous way as a string, with each integer converted into a bitstring, so that every distinct tuple yields a different bitstring. The job of our protocol is to define these tuples so that each secret value is generated by exactly one PRF call. We will sometimes call these tuples *labels*.

The trustees' individual shares for each one-time key are generated as follows:

For one-time key π and Trustee t:

$$\mathtt{PATH}^t_\pi \leftarrow \mathtt{PRF}(K[t], (1,\pi), |\mathtt{PATH}_\pi|)$$
$$X^t_\pi[i,j] \leftarrow \mathtt{PRF}(K[t], (2,\pi,i,j), \ell)$$
$$\rho^t_\pi \leftarrow \mathtt{PRF}(K[t], (4,\pi), \ell)$$
$$\phi^t_\pi \leftarrow \mathtt{PRF}(K[t], (5,\pi), \ell \times n)$$

Since these shares are generated using a PRF, each trustee requires only a very small amount of storage. See Table 1 for a list of all the labels used in our schemes.

domain	variable	use	full label	reference
1	PATH	shares of PATH	$(1, \pi)$	this section
2	X	shares of one-time private key	$(2, \pi, i, j)$	this section
4	ρ	shares of prefix	$(4, \pi)$	3.5
5	ϕ	shares of prefix checksum	$(5, \pi)$	3.5
10	ϕ	computing check value for prefix	$(10, \pi, \rho)$	3.5

Table 1. Labels used with a PRF to derive trustee shares and check values in our scheme.

Deriving the Helper Shares We can now spell out the full process which the dealer uses to generate the shares. The individual trustees' shares are entirely determined by their PRF keys and the choice of key-id π; the following algorithm produces the Helper shares. We assume that the dealer starts by generating the entire set of one-time signing keys, their corresponding public keys, and the Merkle tree path for each one-time key. We also assume the attacker starts out knowing each trustee's PRF key. The dealer then runs the following function to produce the Helper shares.

1: **function** PseudorandomShares($K[1 \ldots n], X_{\text{all}}, \texttt{PATH}_{\text{all}}, n$)
// *Note: $K[i] \neq K[j]$ when $i \neq j$*
2: **for** each key, π **do**
3: $\texttt{PATH}^H_\pi \leftarrow$ PseudorandomSplit($K[1 \ldots n], \texttt{PATH}_\pi, (1, \pi), n$)
4: **for** each component (i, j) of X **do**
5: $X^H_\pi[i][j] \leftarrow$ PseudorandomSplit($K[1 \ldots n], X_\pi, (2, \pi, i, j), n$)
6: **return** ($\texttt{PATH}^H_{\text{all}}, X^H_{\text{all}}$)
7: **function** PseudorandomSplit($K[1 \ldots n], \alpha, L, n$)
8: $\alpha^H \leftarrow \alpha$
9: **for** $t \leftarrow 1 \ldots n$ **do**
10: $\alpha^H \leftarrow \alpha^H \oplus \texttt{PRF}_{K[t]}(L, |\alpha|, |\alpha|)$
11: **return** (α^H)

Algorithm 2: Splitting a Winternitz key into shares

At the end of this process, the trustees need only be given their keys and the set of values of π in order to be able to rederive their shares, imposing an extremely small storage requirement. The process also produces a set of Helper's shares, which will be fairly large–about 2 GiB for a composite key that supports 2^{16} signatures.

To see why the Helper shares can be safely made public, consider an attacker who knows the shares of trustees $2 \ldots n$. The attacker can determine

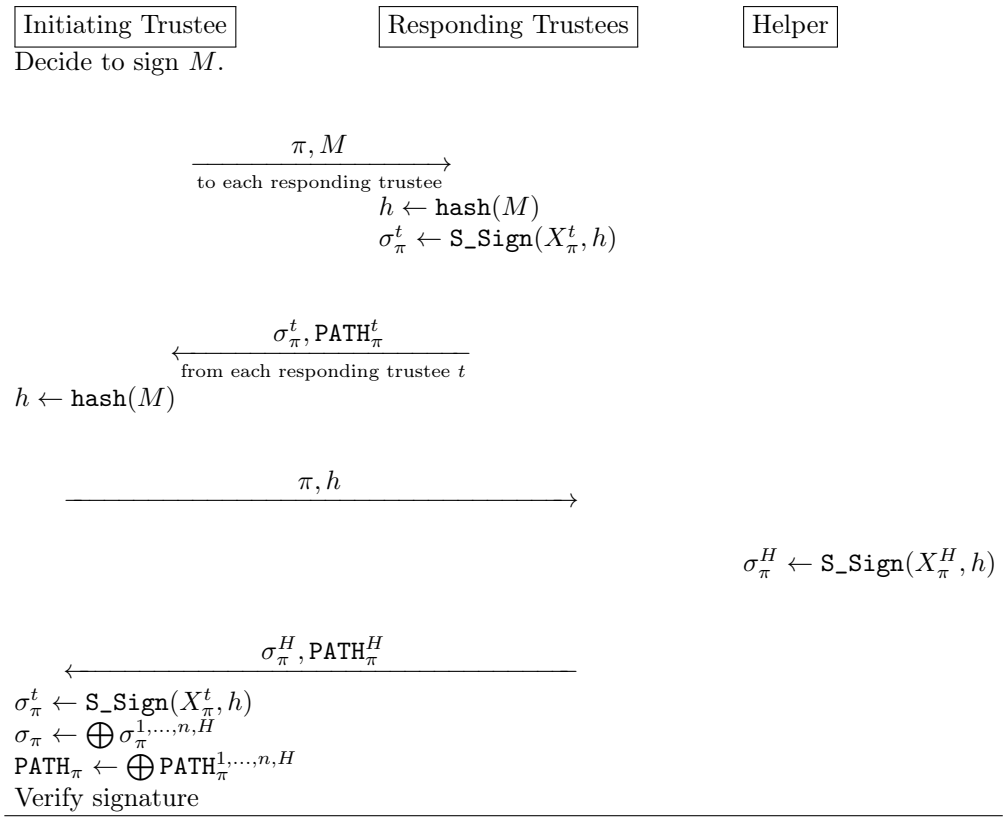
$$\alpha = (\alpha^H \oplus \alpha^2 \oplus \ldots \oplus \alpha^n) \oplus \alpha^1$$

However, since α^1 is indistinguishable from a random value, this leaks no information about α.

The Helper The Helper shares require a lot of storage. In order to prevent the need for any of the trustees to store these shares, we introduce the idea of the *Helper*–a minimally-trusted participant in the protocol that provides a large amount of memory, but holds no secret data, and is not trusted to decide whether or not to sign a message. The Helper's entire role in the protocol is to respond to queries by looking up the answers in the Helper shares, concatenating them, and sending the result back.

In the revised signing protocol (illustrated in Protocol 2), all messages go from or to the initiating trustee. Messages to the responding trustees are sent to each responding trustee individually–the responses are sent from each responding trustee back to the initiating trustee. The initiating trustee also communicates directly with the Helper. This allows us to secure the communications in this protocol with pre-shared keys, requiring only symmetric cryptography and quite limited storage for any reasonable number of

trustees. For ease of exposition, we assume in the diagram that the trustees agree to the signature. However, the responding trustees may reject a signature (returning $\perp$) if they refuse to sign M or believe the key-id to have already been used. By contrast, the Helper has no such option. (If corrupted, it could refuse to respond or send garbage, but there is no valid reason for it to refuse a message.)



Protocol 2: n-of-n Signatures with Pseudorandom Shares and Helper

3.5 Prefixes and Masking: Protecting Against Collision Attacks

Most modern hash-based signature schemes incorporate an unpredictable prefix or mask into the hash of the message. This ensures that even an attacker who can find collisions for the hash function cannot attack the HBS scheme–instead, these schemes hope to base their security on the second preimage resistance of the hash, or at least to retain security even against an attacker who can find collisions against the hash.

Adding an unpredictable randomization parameter (called a *prefix* and labeled as ρ in the rest of this paper) to an n-of-n signature scheme introduces a new problem: since we care about attacks by up to $n-1$ of the trustees against the signature scheme, if the security of the signature scheme relies on the unpredictability of the prefix before the message is chosen, then we need a secure way for the trustees to derive this value. The initiating trustee must specify the message to be signed *before* any subset of $n-1$ trustees can determine the prefix, and it must not be possible for the dishonest trustees to convince an honest trustee to accept an altered prefix.

Our solution is straightforward: When the shares are set up, the dealer generates the prefixes, using whatever mechanism is consistent with the HBS being used. Then, for each one-time key π, the dealer splits the prefix ρ_π across the trustees just as it does the other secret components of the key, ensuring

that no subset of $n-1$ trustees can learn ρ_π. The dealer also splits a check value, ϕ_π, which allows each trustee to efficiently check that ρ_π is the correct[14] prefix for one-time key π. ϕ_π, in turn, is composed of values that will allow each trustee to verify that the prefix is correct.

The signing protocol becomes somewhat more complicated–the requesting trustee must first send out M; all trustees respond with their shares of ρ_π, ϕ_π. The requesting trustee then sends out ρ_π, ϕ_π; each responding trustee verifies that ρ_π is the correct prefix for this one-time key and, if so, computes and sends back its shares of the one-time signature on $\texttt{hash}(\rho_\pi, M)$. Once again, verification is identical to that used for the underlying hash-based signature scheme with prefixes.

Note that we impose no restrictions on the choice of the prefixes, except that the dealer must know them all at the time the shares are constructed and distributed to the trustees. Our scheme works with both LMS[14] and XMSS[5, 9]. LMS leaves the generation of prefixes up to the implementation, while XMSS generates its prefix (called KEY) based on information known when the key is generated.

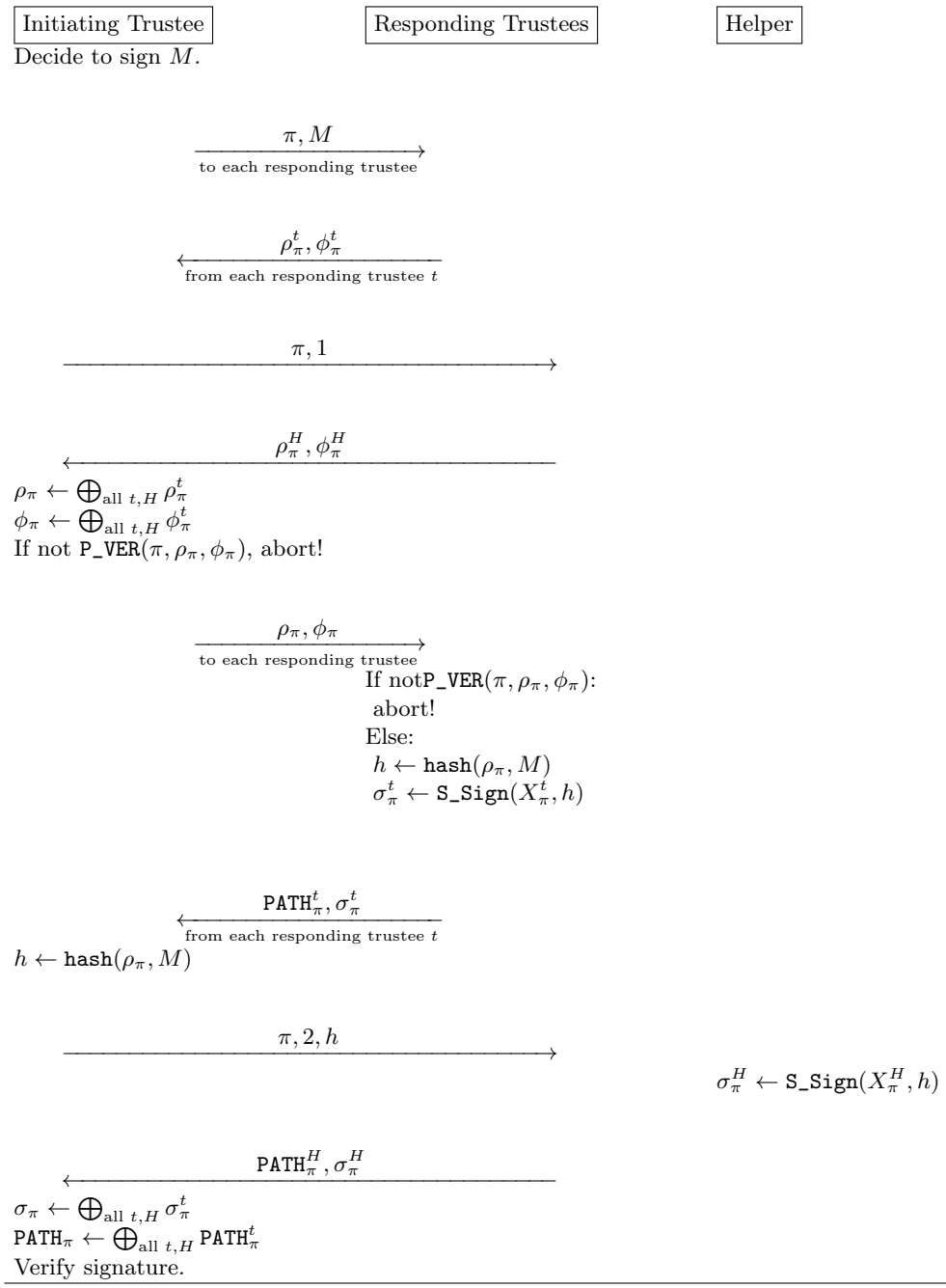
1: **function** PseudorandomSharesPrefix($, \rho_{\text{all}}$, X_{all} , $\texttt{PATH}_{\text{all}}$, n)
// *Requirement: $K[i] \neq K[j]$ when $i \neq j$*
2: **for** each key, π **do**
3: $\texttt{PATH}^H_\pi \leftarrow \text{PseudorandomSplit}(K[1 \ldots n], \texttt{PATH}_\pi, (1, \pi), n)$
4: $\rho^H_\pi \leftarrow \text{PseudorandomSplit}(K[1 \ldots n], \rho_\pi, (4, \pi), n)$
// *Create the check value ϕ. Note that $|\phi| = n \times \ell$.*
5: $\phi_\pi = \texttt{P\_CHK}(K[1 \ldots n], \pi, \rho_\pi)$
6: $\phi^H_\pi \leftarrow \text{PseudorandomSplit}(K[1 \ldots n], \phi_\pi, (5, \pi), n)$
7: **for** each component (i, j) of X **do**
8: $X^H_\pi[i, j] \leftarrow \text{PseudorandomSplit}(K[1 \ldots n], X_\pi, (2, \pi, i, j), n)$
9: **return** $(PATH^H_{\text{all}}, X^H_{\text{all}}, \rho^H_{\text{all}}, \phi^H_{\text{all}})$
// *Called by dealer to construct all n check values for this prefix.*
10: **function** $\texttt{P\_CHK}(K[1 \ldots n], \pi, \rho)$
// *ϕ is an array $1 \ldots n$ of ℓ-bit strings.*
11: **for** $i \leftarrow 1 \ldots n$ **do**
12: $\phi[i] \leftarrow \texttt{PRF}_{K[i]}((10, \pi, \rho), \ell)$
13: **return** ϕ
// *Called by trustee to verify one check value for this prefix.*
14: **function** $\texttt{P\_VER}(K, t, \pi, \rho, \phi)$
// *Use the* PRF *to verify that the prefix matches the check value.*
// *t is which trustee is doing the verification; K is its* PRF *key.*
15: **if** $\texttt{PRF}_K((10, \pi, \rho), \ell) = \phi[t]$ **then**
16: **return** 1
17: **else**
18: **return** 0
19: **function** PseudorandomSplit($K[1, \ldots, n], \alpha, L, n$)
20: $\alpha^H \leftarrow \alpha$
21: **for** $t \leftarrow 1 \ldots n$ **do**
22: $\alpha^H \leftarrow \alpha^H \oplus \texttt{PRF}_{K[t]}(L, |\alpha|, |\alpha|)$
23: **return** (α^H)

Algorithm 3: Splitting Winternitz shares and prefixes

The protocol for doing n-of-n signatures also becomes a bit more complex. The whole process is illustrated in Protocol 3. Once again, for ease of exposition we assume that trustee 1 is the initiating

[14] If the initiating trustee could learn or alter the prefix after sending the message to be signed, the security benefit of the prefix would be lost. Specifically, an initiating who could find collisions on the hash function could get a signature on some message not agreed on by the other $n-1$ trustees in the coalition.

trustee, and that the responding trustees accept the proposed message. In reality, any trustee may initiate the signature, and the trustees may reject the signature request by returning $\perp$. Also, the trustees will send $\perp$ and abort the protocol if they do not agree with the value of ρ. Just as before, the initiating trustee must verify the final signature before relying on it.



Protocol 3: Pseudorandom Share n-of-n Signatures With Prefix and Helper

3.6 Can the Helper Censor Signatures?

In the above protocol, it is easy to see that the trustees can decide whether or not to stop running the protocol based on not wanting to sign a given message. However, the Helper (by definition, an untrusted party) must not have the power to decide which messages are to be signed. This is ensured by the initiating trustee sending the Helper $\texttt{hash}(\rho, M)$, rather than M. Because the Helper has no way to know the value of ρ, it has no information about M, and so must respond without knowing the value of M. The Helper[15] can deny service, preventing any signatures from being constructed. However, the Helper never learns what message is being signed, so it cannot exert control over which messages are being signed.

3.7 Summary So Far

So far, we have developed a proof-of-concept n-of-n hash-based signature scheme, and refined it into a much more practical one that incorporates random prefixes to support randomized hashing of messages. The more practical scheme requires the assistance of an untrusted participant called the Helper (or access to a large public string called the *Helper shares*) to handle the storage required by the scheme. Next, we extend these schemes to k-of-n and coalition signatures.

4 Stateful Hash-Based *k*-of-*n* and Coalition Signatures

4.1 Overview

A major benefit of threshold signatures is the ability to combine split control (so that a single rogue employee or compromised device cannot sign arbitrary messages) with redundancy (so that a single device being lost, stolen, or destroyed doesn't prevent future signatures). With n-of-n signatures, we get split control; we still need to provide redundancy in the form of signatures that allow some incomplete subsets of trustees to sign messages while still limiting those subsets to ensure a good tradeoff between security and availability.

4.2 What Doesn't Work

The obvious way to turn our n-of-n signature scheme into a k-of-n threshold signature scheme (for $1 < k < n$) is to replace the XOR-based secret sharing we used above by a k-of-n secret sharing scheme. The requester would reconstruct the shares of signatures from the shares in much the same way as in our scheme. Unfortunately, this doesn't work, because of the requirement to never allow reuse of a key.

Recall that a stateful hash-based signature requires the signer to keep track of all one-time signing keys used so far–reusing a key allows anyone who sees both signatures to forge new signatures. Further, a k-of-n threshold signature scheme must be secure even against an attacker who controls a coalition of $k - 1$ trustees.

Unfortunately, a k-of-n stateful threshold signature scheme of this kind with $k < n$ would be insecure. For example, consider a 4-of-5 threshold signature scheme, with three corrupt trustees (A,B,C) and two honest trustees (D,E). A, B, C, and E work together to form a signature using key-id #1. Then, A, B, C, and D work together to form a signature also using key-id #1. A, B, and C have now seen two signatures with the same key, and may be able to forge many more signatures using key-id #1. This attack will work for any $k < n$.

Avoiding this attack appears to require some very heavyweight mechanisms: establishing a broadcast channel between all n trustees to keep track of which keys have been used, a trusted third party or timestamping service to keep track of which key-ids have been used, or some other much more complex protocol to prevent key reuse in the presence of $k - 1$ corrupt trustees.

[15] Recall that the Helper can be replaced with a shared string to which the inititiating trustee has access. In that case, nobody but the trustees can block signatures.

4.3 Sharding: Composing k-of-k Coalitions

There is a simple solution to this problem, which interacts remarkably well with the use of Merkle trees in stateful HBS schemes: We can always construct a k-of-n scheme as the composition of many k-of-k schemes. (We sometimes call each k-of-k coalition a *shard*.)

Suppose we want a 3-of-5 scheme. Since $\binom{5}{3} = 10$, we can get the functionality of a 3-of-5 scheme by establishing a 3-of-3 scheme for each of the ten possible coalitions of the trustees and distributing the shares for each coalition to the trustees who are members of that coalition. Each trustee must now keep track of which keys have been used in *each* of its six coalitions, as well as the range of key-ids that go with each of its coalitions.

The one-time keys for each of the ten coalitions are stored together in the same Merkle tree, and the Helper stores Helper shares for each of the one-time keys, but only k trustees have shares that can be used to sign with any given one-time key. We can think of this in two equivalent ways:

1. The total set of one-time keys is fixed to some value (say 2^{20}), and we divide the one-time keys among the coalitions.
2. We require that we always have some minimal number of signatures (say, 2^{16}) even if only one coalition remains, and we make a Merkle tree of 10×2^{16} one-time keys.

Stateful hash-based signature schemes have the property that the length of the signature depends partly on the number of possible signatures, so this scheme requires somewhat longer signatures. However, Merkle tree paths grow only logarithmically in the number of leaves. For example, suppose we want to ensure that each coalition in a 3-of-5 signature scheme can sign 2^{16} messages. There are $\binom{5}{3} = 10$ coalitions needed, so we need ten times as many total one-time keys in the scheme. A simple way to meet this requirement is to have 2^{20} instead of 2^{16} one-time keys in the whole Merkle tree. This makes the `PATH` four hashes longer. With commonly-used parameters for the underlying HBS scheme, this means that the full signatures become about 5 % longer[16] when we go from an n-of-n scheme with 2^{16} one-time keys to a 3-of-5 scheme with 2^{16} one-time keys per coalition, and 2^{20} total one-time keys.

4.4 Limitations for k-of-n Threshold Signatures

Sharding is in some ways an inelegant solution, and it limits the things we can practically do. Sharding works well for theshold signatures when n and k are small, but becomes impractical when the number $\binom{n}{k}$ of quorums is huge. Consider a 50-of-100 scheme. The number of different quorums is $\binom{100}{50} \approx 2^{96.35}$. Even providing every quorum with a single key is infeasible!

Fortunately, most practical uses of threshold signatures have a relatively small number of trustees, and a correspondingly small set of quorums. For such uses, our scheme is powerful and flexible. For example, while a 50-of-100 scheme spread across a huge company would not be practical, a scheme that allowed any 5/10 trustees from each of ten different sites to sign a message would be practical.

Suppose we start with a total of 2^{20} one-time signing keys. Table 2 shows the effect of different choices of n and k on theshold k-of-n signature schemes, listing the number of coalitions each trustee belongs to and the number of signatures each coalition can produce. The number of coalitions each trustee belongs to determines how much memory each trustee must dedicate to keeping track of the current key-id in each of its coalitions.

As can be seen from the table, relatively small choices of n, k yield practical schemes–for example, a 3-of-5 or 4-of-7 threshold hash-based signature scheme can work well in practice. On the other hand, a 10-of-20 scheme is not really practical–each coalition in such a scheme gets only about five signatures, and each trustee needs lots of storage to keep track of which keys can still be used in each coalition (at the very least $\approx$ 34 kilobytes for 92 378 three-bit counters, each taking values between zero and five).

[16] Note that these are still standard hash-based signatures–stateful hash-based signatures get slightly longer as the total number of one-time signatures available goes up, because the Merkle tree path for the one-time key used grows longer.

n	k	coalitions	coalitions per trustee	signatures per coalition
3	2	3	2	349525
5	2	10	4	104857
5	3	10	6	104857
7	2	21	6	49932
7	4	35	20	29959
9	2	36	8	29127
9	5	126	70	8322
20	10	184756	92378	5

Table 2. Effect of choice of n, k assuming 2^{20} one-time keys in composite key.

4.5 Generalized Permission Structures and Coalition Signatures

Sharding also offers a huge degree of flexibility in defining how signatures may be authorized, far more than is provided by a simple threshold signature scheme. With sharding, we can support *any* monotone permission structure, so long as the number of quorums isn't excessive.

We propose the term "coalition signature" to specify a scheme that allows arbitrarily-constructed coalitions of trustees to sign messages. We are unaware of any other signature scheme that fits this description. Our k-of-n threshold HBS schemes are simply instances of coalition signatures where the coalitions are formed out of all subsets of k trustees. However, coalition signatures allow for far finer control of how signatures may be carried out.

Below, we offer three simple examples, barely scratching the surface of what is possible. As above, our examples assume 2^{20} one-time keys per composite key. Table 3 shows how these coalition signature schemes relate to somewhat similar (but much less precisely controlled) threshold signature schemes.

Example # 1: Consider a company with five departments, each with four trustees. A signature requires approval from all five departments, but it doesn't matter which trustee from each department approves signing. There are $4^5 = 1024$ coalitions that can sign a message, and each trustee is a member of 256 coalitions. This can be handled by sharding in a straightforward way. The closest approximation to this access structure in a pure threshold signature would probably be a 5-of-20 scheme, but note that this scheme has much finer control of which trustees may sign a message, and also far fewer coalitions.

Example # 2: Consider an organization that has five highly-trusted senior trustees, and also five less-trusted junior trustees. We allow signatures by any three senior trustees, any two senior trustees with one junior trustee, or any one senior trustee with *three* junior trustees. The total number of coalitions is thus $\binom{5}{3} + \binom{5}{2} \times 5 + 5 \times \binom{5}{3} = 10 + 10 \times 5 + 5 \times 10 = 110$. Each senior trustee is a member of 36 coalitions; each junior trustee is a member of 40 coalitions. While this looks a little like a 3-of-10 scheme, note that it again has a great deal more precise control over who is allowed to sign, and also far fewer coalitions.

Example # 3: Similar to example # 3, assume ten senior and ten junior trustees. Signatures require ten trustees in total, including at least eight senior trustees. The total number of coalitions is $1 + \binom{10}{9} \times \binom{10}{1} + \binom{10}{8} \times \binom{10}{2} = 1+100+2025 = 2126$. A senior trustee is a member of $1+\binom{9}{8} \times \binom{10}{1} + \binom{9}{7} \times \binom{10}{2} = 1+90+1620$ coalitions. A junior trustee is a member of $\binom{10}{9} \times 1 + \binom{10}{8} \times \binom{9}{1} = 10 + 405 = 415$ coalitions. This scheme can still be considered practical, while the practicality of a proper 10-of-20 scheme is questionable, as pointed out in Section 4.4.

4.6 Enforcement

A natural question to ask is: what prevents a given trustee from providing shares for a shard it is not authorized to use? That is, if a given key is in a shard controlled by Alice and Bob, what prevents Carol from participating in the signature protocol for this key?

scenario	coalitions	coalitions per trustee	signatures per coalition
Example # 1	1024	256	1024
5-of-20 scheme	15504	3876	67
Example # 2	110	≤40	9532
3-of-10 scheme	120	36	8738
Example # 3	2126	≤1620	493
10-of-20 scheme	184756	92378	5

Table 3. Comparing examples #1 to #3 with proper threshold schemes, assuming 2^{20} on-time keys.

The answer lies in the Helper shares. When a key is in a shard controlled by Alice and Bob, its Helper shares for some shared value α are constructed as:

$$\alpha^H = \alpha \oplus \alpha^{\text{Alice}} \oplus \alpha^{\text{Bob}}$$

Nothing prevents Carol from using the `PRF` to compute her own shares for this key, but her shares are of no help in recovering α, because the Helper shares were computed using only the correct value of α and Alice and Bob's shares. Nor will Alice or Bob send or accept any shares to/from Carol for a shard where Carol is not a member.

4.7 Accountability

Most threshold schemes have no way to determine which quorum of trustees created the signature. In some applications, this is a desirable property; in others, the key owner may prefer to find out who is to blame for signing a particular message. Our scheme leaves the possibility of determining the responsible quorum, based on *which shard* was used to sign the message. This information may also be made available to outsiders, but this isn't required, and in fact, it is possible for the dealer to set the scheme up in a way that prevents the signatures leaking any information about which messages were signed by which coalition.

4.8 Coalition Signatures with a Helper

The protocol for doing k-of-n and coalition signatures with prefixes and the Helper is almost identical to the n-of-n signature with prefixes and the Helper; the only addition is a need for the initiating trustee to determine which coalition it should use. The details of how each trustee determines which other trustees are available are beyond the scope of this paper, though we note that a practical implementation will want to do some load-balancing across coalitions, to ensure that the one-time keys from each coalition get used at a consistent rate.

5 Practical Considerations

5.1 Threshold Signatures Make Stateful Hash-Based Signatures Work Better

We believe our approach makes stateful hash-based signatures significantly more useful in practice. Threshold signatures address many of the practical engineering and operational problems with using stateful hash-based signatures. Specifically:

– A great deal of engineering and operational complexity is involved in managing a device with a stateful hash-based signature key, in order to avoid accidentally reusing a one-time signing key. By splitting the key into n shares across n separate devices, the chances of accidental or malicious key reuse become enormously smaller–if k trustees are required to sign a message, then all k must agree on the same one-time key to use. Key reuse will only happen if all k trustees make the same error.

– Backing up a device that holds a stateful hash-based signature key is dangerous–restoring from backup can easily lead to reusing a one-time signing key. However, *not* backing up a device storing a high-value key risks having important functions stop working when a device fails or is lost. Splitting the stateful HBS key into the shares of a k-of-n signature scheme provides substantial fault tolerance without the need to back up any specific device. Further, the Helper can be backed up or replicated freely without raising any security issues.
– Many of the applications for which stateful hash-based signatures are proposed involve very high-value signatures that must remain secure for a long time. The split control of a threshold signature scheme is a very good fit for this kind of high-value application–a compromised device or rogue employee is limited in the damage they can do.

5.2 Adapting Our Scheme to Existing Standards

Fully-specified hash-based signature schemes such as LMS[14] or XMSS[9] incorporate many details we've omitted in this discussion–for example, each public key is given a globally-unique identifier, and hash calls incorporate metadata into each computation to avoid multitarget attacks. Different standards further define how the prefix is to be generated and incorporated into the message hash. Our schemes can easily be adapted to follow all of these requirements, so long as the prefixes can be known when the threshold scheme is set up.

5.3 Computing and Communications Overhead

Each signature requires two round-trip messages between the initiating trustee and each responding trustee to produce a signature, as well as two round-trip messages between the initiating trustee and the Helper. The messages are relatively short–the message to be signed, plus a few hundred bytes more than the length of the one-time signature.

The point-to-point secure channels between the trustees can be based on pre-shared symmetric keys, and require nothing more expensive than an authenticated encryption scheme.

Very little computing power and storage is needed for a trustee. During signing, each trustee does only fast symmetric operations–hash functions, `PRF` calls, and XORing together bitstrings. A trustee only needs to store: (1) its `PRF` key, (2) the current key-id to be used with each coalition of which it is a member, and (3) pre-shared symmetric keys for communicating with each of the other trustees.

Coordinating on which key to use is straightforward. In the first message of the protocol, the initiating trustee sends the key-id it expects to use to each responding member of the coalition. If any trustee in the coalition believes this key has already been used (it is possible for disagreements to arise honestly, due to communications failures), it will communicate this fact to the initiating trustee (shown as sending $\perp$ in our protocols). Each responding trustee can then send the key-id it believes should be next, and the initiating trustee can select the largest key-id, and use that key-id to initiate its request. This allows any member of a coalition to deny service by using up all the one-time keys available to the coalition at any time, but this was already possible, since any trustee can always refuse to sign with any keys in its coalition.

5.4 Memory Requirements and the Role of the Helper

The biggest downside of our techniques is the requirement for several gigabytes of memory, probably duplicated across multiple untrusted Helper devices, perhaps implemented as a distributed key:value database. (As noted above, with typical parameters, a composite key supporting 2^{16} signatures requires a Helper with about 2 GiB of storage.)

We use the Helper to avoid the need for trustees to have large amounts of memory–this is a quite important practical consideration if the trustees' keys are stored on secure cryptographic modules, which are unlikely to have gigabytes of storage. The Helper allows us to split out the role of providing bulk storage from the role of having access to secret data, the ability to cause a key to be reused, or the power to decide whether to sign a given message.

In the k-of-n and coalition signature case, however, the Helper offers another advantage: we can duplicate it many times (so there may be two or three Helpers) or back up all its information without weakening our system. Even an attacker who learns *all* of the Helper's data is unable to forge signatures, and backing up and restoring the Helper's state don't introduce a risk of key reuse or key compromise. Further, the Helper's shares can be safely backed up and stored in many locations without concern. This preserves the fault tolerance property we expect from a k-of-n threshold signature scheme. It's also possible to have each trustee store a few Helper shares for each of its coalitions, allowing continued functionality if the Helper or Helpers are temporarily unavailable. (Since backups of the Helper are safe to keep without added security, unavailability of the Helper will be temporary.)

The Helper can deny service to the trustees. However, when prefixes are used, it cannot do so selectively–it never sees the message being signed, only a hash. Because it doesn't know the prefix used, it cannot even precompute messages to block.

The Helper can be implemented on a cloud storage service[17], without requiring the dealer or user to trust the cloud storage service with their keys. While we show the Helper as taking part in the protocol in our diagrams, it is straightforward to implement the Helper as a key:value database and have the initiating trustee make queries to construct the Helper's share of the signature and `PATH`. (For typical Winternitz parameters, the initiating trustee would need to make 68 key:value database queries to produce the Helper's share of $\texttt{PATH}, \sigma$.) Alternatively, the initiating trustee could request the 17 KiB chunk of data associated with key π and compute the Helper's share locally, or a cloud provider could operate a Helper service for a fee–again, this does *not* require the dealer to trust the cloud service with any secrets, or to give the cloud service any power over which messages are to be signed.

5.5 Other Roles

The Dealer The dealer in our scheme is entirely trusted–they are assumed to be the owner of the signing key. Our protocols allow the dealer to delegate the signing process to some defined subsets of the trustees.

The Delegator in a proxy signature scheme Although we wouldn't expect the scheme to be used this way, the dealer could in principle start with signing messages in a single-party manner and, at any point of time, delegate the signing process to trustees (or rather, to specified subsets of a specified group of trustees).

The Verifier The verifier in our scheme is exactly the same as one used to verify signatures from the underlying HBS scheme. The signatures produced are identical in both cases.

6 Conclusions

In this article, we have shown how to turn stateful hash-based signatures into threshold signature schemes. Our techniques have low communications and computational overhead, require only fast symmetric cryptography, and can be applied to existing hash-based signature specifications such as LMS[14] or XMSS[9]. They do, however, require an untrusted participant in the signing protocol with abundant memory called the Helper, or alternatively, access to a large shared public string (the Helper shares). The Helper can be implemented using cloud storage or on a cloud service, without requiring the owner of the key to trust the cloud storage service with their signing keys.

Our techniques offer a practical solution to many practical engineering and operational difficulties with stateful hash-based signatures and provide security only negligibly lower than that provided by the underlying stateful hash-based signature scheme. On the other hand, our techniques have some limitations: along with the memory requirements, they require a trusted dealer and cannot support stateless hash-based signature schemes like SPHINCS+[11] or tree-of-tree approaches such as HSS[14]. It is not yet clear how the requirements in NIST's hash-based signature guidance [6] might apply to our techniques.

[17] For comparison, at the time of this writing, there are cloud storage services offering free accounts with as much storage as is needed for the Helper to support a threshold stateful HBS instance with 2^{16} keys.

6.1 Future Research

There are many questions left open by this research. Among them are:

1. In this paper, we have shown how to produce threshold Winternitz signatures. Lamport signatures are straightforward to adapt to our techniques. Can they be extended to other hash-based signature schemes such as HORS[18], FORS[11], and BiBa[17]?
2. Our techniques require that the whole set of prefixes for signatures be known at the time the shares are generated for the trustees. Can we adapt these techniques to allow more flexibility in generating these prefixes?
3. Our techniques require a trusted dealer. Can our techniques be improved to eliminate this requirement?
4. Relatedly, is there any way to adapt our techniques to tree-of-trees approaches or stateless hash-based signatures? This appears to require replacing the trusted dealer with an efficient protocol, in order to allow on-the-fly regeneration of subtrees of one-time keys.
5. In one sense, it is surprising that such primitives can be made from hash-based signatures at all. This work, alongside the group signatures developed in [2], raises the question of what other surprising bits of functionality remain to be found within these very old and well-studied techniques.

Acknowledgements

The authors wish to thank Bart Preneel, Vincent Rijmen, Frank Piessens, Nigel Smart, Nicky Mouha, Andreas Hülsing, Dan Bernstein, Luís Brandão, Carl Miller, Daniel Apon, Michael Davidson, Yi-Kai Liu, Nathalie Lang, and Eik List for useful conversations and feedback about this work.

References

[1] Aumasson, J.-P., and Endignoux, G. Gravity-SPHINCS. Tech. rep., National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[2] Bansarkhani, R. E., and Misoczki, R. G-merkle: A hash-based group signature scheme from standard assumptions. In *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings* (2018), T. Lange and R. Steinwandt, Eds., vol. 10786 of *Lecture Notes in Computer Science*, Springer, pp. 441–463.

[3] Bernstein, D. J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., and Wilcox-O'Hearn, Z. SPHINCS: Practical stateless hash-based signatures. In *EUROCRYPT 2015, Part I* (Sofia, Bulgaria, Apr. 26–30, 2015), E. Oswald and M. Fischlin, Eds., vol. 9056 of *LNCS*, Springer, Heidelberg, Germany, pp. 368–397.

[4] Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., and Vuillaume, C. Merkle signatures with virtually unlimited signature capacity. In *ACNS 07* (Zhuhai, China, June 5–8, 2007), J. Katz and M. Yung, Eds., vol. 4521 of *LNCS*, Springer, Heidelberg, Germany, pp. 31–45.

[5] Buchmann, J. A., Dahmen, E., and Hülsing, A. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011* (Tapei, Taiwan, Nov. 29 – Dec. 2 2011), B.-Y. Yang, Ed., Springer, Heidelberg, Germany, pp. 117–129.

[6] Cooper, D., Apon, D., Dang, Q., Davidson, M., Dworkin, M., and Miller, C. Recommendation for stateful hash-based signature schemes. Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2020.

[7] Desmedt, Y. Society and group oriented cryptography: A new concept. In *CRYPTO'87* (Santa Barbara, CA, USA, Aug. 16–20, 1988), C. Pomerance, Ed., vol. 293 of *LNCS*, Springer, Heidelberg, Germany, pp. 120–127.

[8] Desmedt, Y., and Frankel, Y. Threshold cryptosystems. In *CRYPTO'89* (Santa Barbara, CA, USA, Aug. 20–24, 1990), G. Brassard, Ed., vol. 435 of *LNCS*, Springer, Heidelberg, Germany, pp. 307–315.

[9] Hülsing, Butin, Gazdag, Rijneveld, and Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, RFC Editor, May 2018.

[10] Hülsing, A. W-OTS+ - shorter signatures for hash-based signature schemes. In *AFRICACRYPT 13* (Cairo, Egypt, June 22–24, 2013), A. Youssef, A. Nitaj, and A. E. Hassanien, Eds., vol. 7918 of *LNCS*, Springer, Heidelberg, Germany, pp. 173–188.

[11] Hulsing, A., Bernstein, D. J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M. M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., and Schwabe, P. SPHINCS+. Tech. rep., National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[12] Krawczyk, D. H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Feb. 1997.

[13] Lamport, L. Constructing Digital Signatures from a One Way Function. Tech. rep., SRI, Oct 1979.

[14] McGrew, Curcio, and Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, RFC Editor, April 2019.

[15] Merkle, R. C. A digital signature based on a conventional encryption function. In *CRYPTO'87* (Santa Barbara, CA, USA, Aug. 16–20, 1988), C. Pomerance, Ed., vol. 293 of *LNCS*, Springer, Heidelberg, Germany, pp. 369–378.

[16] Merkle, R. C. A certified digital signature. In *CRYPTO'89* (Santa Barbara, CA, USA, Aug. 20–24, 1990), G. Brassard, Ed., vol. 435 of *LNCS*, Springer, Heidelberg, Germany, pp. 218–238.

[17] Perrig, A. The BiBa one-time signature and broadcast authentication protocol. In *ACM CCS 2001* (Philadelphia, PA, USA, Nov. 5–8, 2001), M. K. Reiter and P. Samarati, Eds., ACM Press, pp. 28–37.

[18] Reyzin, L., and Reyzin, N. Better than BiBa: Short one-time signatures with fast signing and verifying. In *ACISP 02* (Melbourne, Victoria, Australia, July 3–5, 2002), L. M. Batten and J. Seberry, Eds., vol. 2384 of *LNCS*, Springer, Heidelberg, Germany, pp. 144–153.

[19] Shoup, V. Practical threshold signatures. In *EUROCRYPT 2000* (Bruges, Belgium, May 14–18, 2000), B. Preneel, Ed., vol. 1807 of *LNCS*, Springer, Heidelberg, Germany, pp. 207–220.

[20] Szydlo, M. Merkle tree traversal in log space and time. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings* (2004), C. Cachin and J. Camenisch, Eds., vol. 3027 of *Lecture Notes in Computer Science*, Springer, pp. 541–554.

A A Brief Review of Winternitz Signatures

Here, we give a very brief refresher on Winternitz signatures. For a more complete explanation of Winternitz signatures, see the descriptions in [9, 14].

As discussed above, for Winternitz signatures, the one-time private key consists of an array of hash chains. These can be represented as a two-dimensional array of secret values, $X[0 \dots a + c - 1][0 \dots 2^w - 1]$. Each $X[i][0 \dots 2^w - 1]$ consists of an array of 2^w values, constructed so that

$$X[i][j] = \begin{cases} \text{A random starting value} & j = 0 \\ \texttt{hash}(\text{metadata}, X[i][j-1]) & \text{otherwise} \end{cases}$$

Winternitz signatures sign an ℓ-bit hash value by first breaking the hash into $a = \lceil \frac{\ell}{w} \rceil$ chunks, each of w-bits, and then encoding each w-bit chunk as an entry in one of the hash chains. A verifier given the hash chain entry and the correct value for that w-bit chunk can then compute the final value in the hash chain, and the one-time public key is the hash of the concatenation of all the final values in the hash chain. This means that verifying a valid Winternitz (hash, signature) pair will produce the one-time public key; this is handy for stateful hash-based signatures, since the next part of verifying the signature is verifying a Merkle-tree path.

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

In fully specified HBS standards, metadata is incorporated into each hash function call, to prevent multi-target attacks. For this refresher, we mostly ignore such engineering details.

Suppose we have a one-time signing key, represented as a two-dimensional array, X. (Most implementations of Wintertnitz signatures will simply compute each desired value of $X[i][j]$, but conceptually it is easier to think in terms of the whole array being available.)

In this paper, we represent Winternitz signatures as an array of $a + c$ values, where a is the number of hash chains used to encode the Winternitz hash, and c is the number used to encode the checksum. For example, when $\ell = 256$, typical values for $w = 4$ would be $a = 64, c = 3$; the resulting signature consists of 67 ℓ-bit strings. The one-time private key consists only of the two-dimensional array X.

The following pseudocode shows the sign and verify functions for Winternitz signatures.

function SIGN(X_π, h)
  // *Full Winternitz signature*
  // $h = \texttt{hash}(message), a = \lceil \ell / w \rceil$
  // $C = a \times (2^w - 1), c = \lceil \lg(C)/w \rceil$
  $b[0 \ldots a-1] \leftarrow h$ split into w-bit chunks.
  for $i \leftarrow 0 \ldots a-1$ **do**
    $\sigma_\pi[i] \leftarrow X_\pi[i][b[i]]$
  // *Compute Checksum*
  $S \leftarrow C - \sum_{i=0}^{a-1} b[i]$
  $d[0 \ldots c-1] \leftarrow S$ broken into w-bit chunks
  for $i \leftarrow 0 \ldots c-1$ **do**
    $\sigma_\pi[i+a] \leftarrow X_\pi[i][b[i]]$
  return $(\sigma_\pi(h))$
function VERIFY(σ, h)
  // *Full Winternitz signature*
  // $h = \texttt{hash}(message), a = \lceil \ell / w \rceil$
  // $C = a \times (2^w - 1), c = \lceil \lg(C)/w \rceil$
  $b[0 \ldots a-1] \leftarrow h$ split into w-bit chunks.
  // *Compute Checksum*
  $S \leftarrow C - \sum_{i=0}^{a-1} b[i]$
  $b[a-1 \ldots a+c-1] \leftarrow S$ broken into w-bit chunks
  $T \leftarrow \epsilon$ // *Start with empty string*

15: **for** $i \leftarrow 0 \ldots a + c - 1$ **do**
16: $u \leftarrow \sigma[i]$
17: **for** $j \leftarrow 0 \ldots (2^w - 1 - b[i])$ **do**
18: $u \leftarrow \texttt{hash}(\text{metadata}, u)$
19: $T \leftarrow T \parallel u$
20: **return** $(\texttt{hash}(T))$

B Security Analysis

In this section, we provide a security proof for n-of-n, k-of-n, and coalition hash-based signatures, incorporating prefixes, pseudorandom shares, and the Helper.

B.1 Security Notion and Game Definitions

The standard security notion used for a signature scheme is EU-CMA. Informally, this means that no attacker should be able to sign a new message, even given many signatures on messages of its choice. More formally EU-CMA security is defined in terms of a particular forgery game–a signature scheme has EU-CMA security if there is no attacker that has more than negligible probability of winning the forgery game.

HBS Security Definition We start by defining the security of the hash-based signature scheme from which we construct our n-of-n signatures, in terms of a forgery game. Stateful hash-based signatures need a slightly different game definition than more conventional signatures, in order to account for the use of a large set of one-time signing keys.

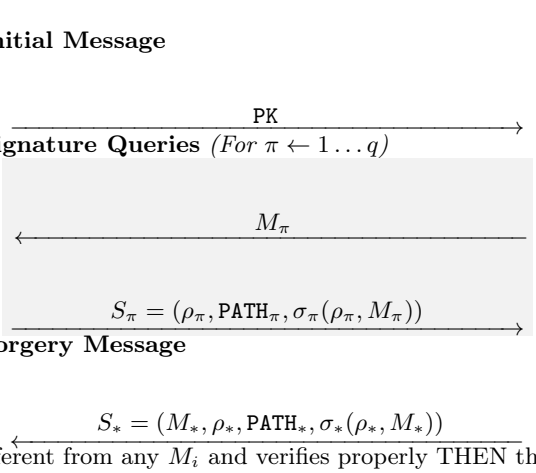
The only difference between the HBS forgery game and the standard EU-CMA forgery game is that the challenger must use a new one-time key for each message, and, thus, must include a Merkle tree path to the corresponding one-time public key. We assume there are exactly Q keys in the stateful hash-based signature, which also provides a bound on how many signatures may be requested by the attacker. Also, we assume the signer in the stateful hash-based signature scheme uses the one-time keys in a fixed order and never reuses a key. (We make the same assumption about the trustees in the threshold versions of HBS signatures.)

An HBS scheme has HBS-EU-CMA security if there is no attacker that wins the HBS-Forge game with greater than negligible probability.

Security Notion for Coalition Signatures Next, we need to define a security notion for n-of-n, k-of-n, and coalition hash-based signatures. Since k-of-n and coalition signatures are composed from n-of-n signatures, this translates to needing a security notion for n-of-n signatures. Once we have an n-of-n signature security notion, security claims about our k-of-n and coalition signatures are simply instances of n-of-n signature scheme security.

Game 2 illustrates the *incomplete-coalition forgery game*, in which the attacker is given $n - 1$ of the n trustees' states and then plays a forgery game, with the Challenger playing the role of the one non-compromised trustee.

The IC-Forge game is quite a bit more complicated than the normal Forgery game because a signature is a *protocol* between the attacker and the challenger (simulating a protocol between multiple trustees), rather than an operation done by the challenger. This means that the IC-Forge game must allow for all the different ways a run of the protocol might occur. Further, the attacker in this game controls $n - 1$ of the n trustees, as well as the Helper, and simply plays the role of all of these trustees in interacting with the challenger, who plays the role of the uncompromised trustee. This leads to three major differences with the Forgery game:



Game 1: The HBS With Prefixes Forgery Game

First, at the beginning of the game, the challenger must give the attacker not only the public key, but also the shares of $n-1$ trustees, as well as the Helper shares. (Thus, our security notion considers only static corruptions, not active ones.) The attacker is allowed to select which trustees it wishes to corrupt, but, since all trustees' keys are generated in exactly the same way (via a PRF call), this choice doesn't really affect anything.

Second, the attacker is allowed up to Q signature queries, where the challenger plays the role of the uncompromised trustee. But now, since a signature query is really a run of the protocol, the attacker has a lot more choices–it can make two entirely different kinds of queries:

A-queries The attacker plays the role of an initiating trustee to get a signature on message M_π. The challenger responds exactly as an uncompromised responding trustee would respond. This means first sending shares of ρ, ϕ and later sending shares of $\texttt{PATH}, \sigma$. The attacker may choose which of its compromised trustees is the initiating trustee.

C-queries The attacker gives the challenger M_π, and the challenger then plays the role of the uncompromised initiating trustee for M_π. The challenger then initiates a signature protocol exactly as the uncompromised trustee would and carries it through to completion. Finally, the challenger provides the attacker with either a valid signature on M_π, or $\perp$ if the attacker's responses caused it to detect an error in the protocol.

Finally, the attacker sends a final message. The attacker wins the game by producing a new message (not one previously queried), along with a valid signature on the new message.

We say that an n-of-n signature scheme has IC-EU-CMA security if there is no attacker who has more than a negligible chance of winning the incomplete coalition forgery game. This is another way of saying that an incomplete coalition (one not authorized to sign a message) has a negligible chance of producing a forged signature, and it is exactly the security property we need from an n-of-n, k-of-n, or coalition signature.

An n-of-n HBS scheme has IC-EU-CMA security if there is no attacker that wins the IC-Forge game with greater than negligible probability.

B.2 The Main Theorem and Intuition for the Proof

Theorem 1 (Main Theorem). *Write $P^{\text{HBS-Forge}}(Q, \mathcal{X})$ for the probability of an algorithm $\mathcal{X}$ to win the HBS forgery game, aka Game 1, requesting Q signatures, $P^{\text{IC-Forge}}(Q, \mathcal{Y})$ for the probability of an algorithm $\mathcal{Y}$ to win the IC-Forge game, aka Game 2, also requesting Q signatures, $\mathcal{A}^{\text{PRF}}(Q', \mathcal{Z})$ for the advantage of a distinguisher $\mathcal{Z}$ to distinguish a PRF from a*

random function, making Q' calls to the PRF, and $Q_{\mathtt{PRF}}(Q)$ for the number of PRF calls when running Protocol 3 to generate Q signatures.
Assume an algorithm $\mathcal{A}$ that wins the IC-Forge with probability $P^{IC\text{-}Forge}(Q, \mathcal{A})$ Then an algorithm $\mathcal{B}$ and a distinguisher $\mathcal{C}$ exist, such that

$$P^{IC\text{-}Forge}(Q, \mathcal{A}) \leq P^{HBS\text{-}Forge}(Q, \mathcal{B}) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}}(Q), \mathcal{C}) + (n-1) \cdot 2^{-\ell} + Q \cdot 2^{-\ell}.$$

Both $\mathcal{B}$ and $\mathcal{C}$ will be described in the proof, and both run $\mathcal{A}$ once, as a subprogram. Apart from the resources for running $\mathcal{A}$, the running time and storage for $\mathcal{B}$ and $\mathcal{C}$ is at most linear in the number of one-time signatures. Similarly, $Q_{\mathtt{PRF}}(Q)$ is linear in Q.

Intuition for the Proof We first (Figure 1) provide a simulator (Sim-RF) which uses an attacker that wins IC-Forge to win HBS-Forge (that is, to forge signatures in the underlying HBS scheme). However, the simulator uses calls to a random function where our scheme uses calls to a `PRF`.
We then (Figure 2) provide a second simulator (Sim-F). If there is an attacker that wins the IC-Forge game when the shares are generated by a `PRF` but not by a random function. Sim-F uses that attacker to distinguish the `PRF` from a random function.

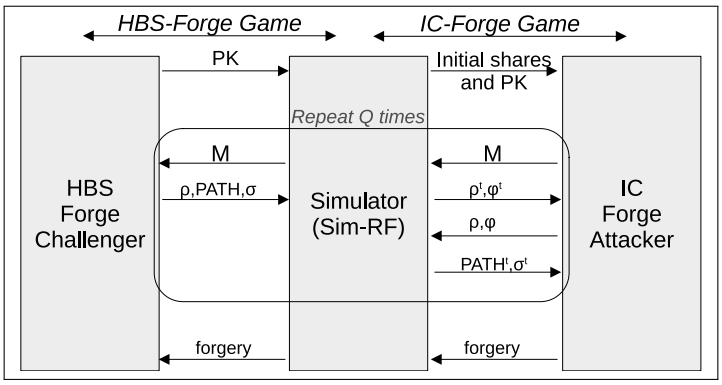


Fig. 1. Winning IC-Forge implies winning HBS-Forge

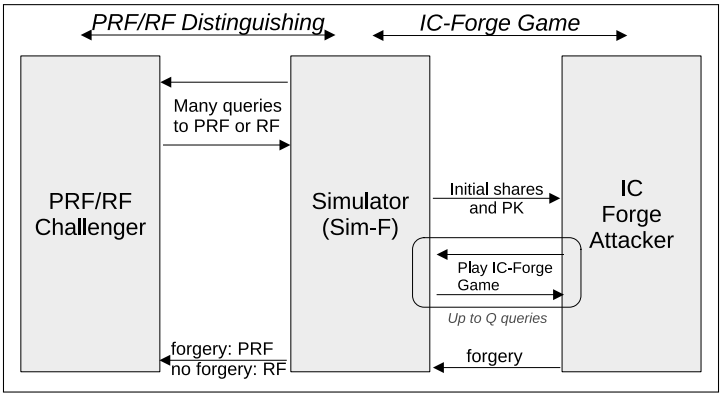


Fig. 2. Forging against PRF version of challenger implies distinguishing PRF/RF

B.3 Preliminary Lemmas and Definitions

We write T for the *transcript*, i.e., for the sequence of messages sent to and from A during one run of the IC-Forge game.

Definition 1 (Winning Transcript). *We call T a* winning transcript *when the final message in T is a valid forgery message–that is, it is a valid signature on some M_* which A never queried during the game. More formally, this happens when the final message in T is*

$$(m_*, PATH_*, \sigma_*(M_*))$$

such that

$$\textbf{\textit{verify}}(PK, \texttt{hash}(M_*), (PATH_*, \sigma_*(M_*))) = 1 \text{ and } M_* \notin \{M_1, M_2, \ldots, M_Q\}.$$

B.4 Proof of Main Theorem.

The proof appears in Section B.5 and is based on a sequence of lemmas, which we provide first. Algorithm B from the theorem is the algorithm Sim, as defined in Figure 4, and distinguisher C is D, as defined in Lemma 15.

The Simulator and Challenger for the IC-Forge Game with Prefixes Algorithm 4 demonstrates the approach to prove the main theorem. The left algorithm is the challenger in the IC-Forge game–it responds to attacker queries and ultimately receives the attacker's final message. If that message contains a forgery (that is, it includes a signature that wasn't produced during the query phase), then the attacker wins the IC-Forge game. The right algorithm is the simulator–it also plays the IC-Forge game with the attacker. The proof of our main theorem stems from two claims: Firstly, it is infeasible for the attacker to distinguish the simulator from the challenger. Secondly, if the attacker wins the IC-Forge game, i.e., if its final message contains a forgery, then the simulator uses that message to win the HBS forgery game.

Lemma 3 (Pseudorandom-Share C-Queries Not Needed). *A C-query against this challenger can be simulated exactly using only an A-query and knowledge of the public key. Thus, there is no need for the IC-Forge game to include C-queries.*

Proof. In the IC-Forge game, the challenger is asked to initiate a signature query on message M. It makes this query to the other trustees (all played by the attacker), who respond with $\rho^{\neq t}, \phi^{\neq t}$, the shares of the prefix and its check value. The challenger reconstructs and verifies ρ, ϕ and sends these values to the attacker, which responds with $\texttt{PATH}_\pi^{\neq t}, \sigma_\pi^{\neq t}(M)$ or $\perp$. The challenger finally sends back either $\texttt{PATH}_\pi, \sigma_\pi(M)$ or $\perp$, depending on the values sent by the attacker.

The attacker can compute the contents of the challenger's first response to the C-query, ρ, ϕ, by making an A-query on message M and getting back ρ^t, ϕ^t and using its own shares to reconstruct the values of ρ, ϕ. After sending these two values back to the challenger, the challenger will respond with $\texttt{PATH}_\pi^t, \sigma_\pi^t(M)$. Using its own shares, the attacker can reconstruct the challenger's second response to the C-query, $\texttt{PATH}_\pi, \sigma_\pi(\rho, M)$.

If $\texttt{PATH}, \sigma$ is a valid signature on the hash of ϕ and M using key π, then the attacker knows that the challenger would have sent it $\texttt{PATH}, \sigma$. If not, then the attacker knows that the challenger would have sent it $\perp$. Note that the challenger as well as the attacker can check this condition–it depends only on $\texttt{PATH}, \sigma, M$, and the composite public key of the underlying HBS scheme.

Since the attacker can exactly predict the result of making a C-query using the results of an A-query, there is no need to include C-queries in the game definitions for n-of-n hash-based signatures with prefixes. □

The IC-Forge Security Lemma The proof of security the n-of-n signatures follows from the following lemma, whose individual conditions must then be proven individually.

Lemma 4 (IC-Forge Security Lemma). *Let* Chal *be the challenger in an incomplete-coalition forgery game for signature scheme Σ. Let* Sim *be the simulator, which simulates* Chal *using calls to the challenger in the hbs-Forge game. Then, winning the incomplete-coalition forgery game for Σ implies winning the hbs-Forge game, if the following conditions hold:*

1. ***Equal Probability Shares**:*

$$Pr[\textit{initial shares sent to attacker} \mid \text{Chal}] = Pr[\textit{initial shares sent to attacker} \mid \text{Sim}]$$

2. ***Identical Responses**: Given any query from an attacker, and identical initial shares,* Chal *and* Sim *will always respond identically.*
3. ***Forgery Transfer**: Consider the* transcript *of running the signature scheme, either with* Chal*, or with* Sim*. Any such transcript is made up of the initial shares, the challenges from the attacker, the responses, and the final forgery message. A transcript τ could be either (1) created by running* Chal *and the attacker, or (2) by running* Sim *and the attacker.* Forgery Transfer *means that if τ implies the adversary winning in case (1), then the same τ implies that* Sim *wins its hbs-Forge game with HBS-challenger in case (2).*

Proof. For the proof, we consider the implications from the three conditions:

1. The first condition guarantees that the initial shares sent to the attacker have equal probability whether from *Chal* or *Sim*. The probability of the attacker making any given query must then be the same for *Chal* and *Sim*.
2. The second condition guarantees that the response to each query is likewise the same when coming from *Sim* or *Chal*. Thus, each additional query, and the final forgery message, must have the same probability of being seen by *Sim* and *Chal*.
   Thus, the probability of any given transcript is the same whether the transcript is with *Chal* or with *Sim*.
3. The third condition guarantees that a transcript T, which would be a winning transcript with *Chal*, is a transcript which will allow *Sim* to produce a winning transcript in the hbs-Forge game.

We conclude that for every transcript T, the probability for T to be a winning transcript against *Chal* is the same as the probability of T being a winning transcript for *Sim* against the HBS-challenger. Thus, an efficient attacker who wins the IC-Forge game can be used to construct a simulator which efficiently wins the hbs-Forge game with exactly the same probability. □

Forgery Transfer The forgery transfer condition holds for any challenger in the IC-Forge game. This is true because a winning transcript for the IC-Forge game ends with a valid signature on a message that wasn't queried during the game; for any winning transcript, it is easy to define a simulator that will turn this into a winning transcript for the hbs-Forge game.

Lemma 5 (Forgery Transfer). *Let T be the transcript of the IC-Forge game between the attacker and the challenger and let T' be the corresponding transcript between the simulator and the HBS-challenger that results when T occurs between the attacker and the simulator. If T is a winning transcript in the IC-Forge game then T' is a winning transcript in the hbs-Forge game.*

Proof. A winning transcript is one in which the final message sent by the attacker contains a valid signature for a message that was never queried during the game. The simulator forwards each query from the attacker to the HBS-challenger, and then forwards the final message to the HBS-challenger. If a winning transcript occurs between the attacker and the simulator, the final message in the transcript will be a valid signature on a message never queried during the game; the simulator will then send a final message to the HBS-challenger which is a valid signature on a message never queried during the game, and, thus, will win the hbs-Forge game. □

Proving Security of Threshold Hash-Based Signatures with Prefixes At last, we are ready to prove the security of our n-of-n hash-based signature scheme.

Approach to the Proof By the IC-Forge lemma above, we need three things to be true for the scheme to be secure:

1. Equal probability shares
2. Identical responses
3. Forgery transfer

Our scheme generates its shares using calls to a PRF. However, in order to prove security, we need to be able to make statements about the probability of those shares taking on a particular value. We start with defining variant of the challenger and the simulator, where the PRF is replaced by a random function (RF):
Following the approach above, we will prove these statements for Chal-RF (the challenger with all calls to the PRF that are used to derive shares for the uncompromised trustee replaced with calls to a random function), and then use the PRF/RF switching lemma to prove security for the challenger, which uses only PRF calls.

Definition 2 (Chal-RF, Sim-RF). *Let Chal-RF be a variant of the challenger in which each call to* $\texttt{PRF}_{K[t]}(\alpha, \ell)$ *(that is, each* PRF *call producing the uncompromised trustee's shares) is replaced with a call to a random function* $RF(\alpha, \ell)$.
Let Sim-RF be a variant of the simulator in which each call to $\texttt{PRF}_{K[t]}(\alpha, \ell)$ *(that is, each* PRF *call producing the uncompromised trustee's shares) is replaced with a call to a random function* $RF(\alpha, \ell)$.

We now prove that an attacker who can win the IC-Forge game against Chal-RF can be used to construct a new attacker who can win the HBS-Forge game.

Lemma 6 (Unique Labels). *The inputs to the* PRF *used to generate each component of each trustee's share are unique–that is, no two* PRF *calls with the same key* K *also get the same input.*

Proof. The components to the shares are generated as follows:

$$\texttt{PATH}^t_\pi = \texttt{PRF}_{K[t]}((1, \pi), |\texttt{PATH}|)$$
$$X^t_\pi[i][j] = \texttt{PRF}_{K[t]}((2, \pi, i, j), \ell)$$

The initial input (1 or 2) ensures that a PRF call used to generate a share of the PATH can never collide with that used to generate a share of the private key. The second input is the unique ID for the one-time signing key whose shares are being generated; this ensures that PRF calls used to generate shares for different one-time keys will never collide. For generating $X^t_\pi[i][j]$, the PRF call also includes i, j, ensuring that the PRF calls used to generate shares of different components of the one-time signing keys will also never collide. □

Lemma 7 (Chal-RF Equal Probability Initial Shares). *For each possible value of initial shares seen by the attacker, the probability of the challenger producing those shares is identical to the probability of the simulator producing those shares.*

Proof. Only PRF keys of the $n - 1$ compromised trustees are sent to the attacker, and these are generated identically (in line 5) by Chal-RF and Sim-RF and so have equal probability.
As shown in lines 13-21 of the Chal-RF and 16-21 of Sim-RF, the shares for the Helper are generated differently. For any internal value α, we have the Helper's share from the challenger:

$$\begin{aligned}\alpha^H_{\text{chal}} &= \alpha \oplus \bigoplus_{k=1}^{n-1} \alpha^k \\ &= \alpha^t \oplus \alpha \oplus \bigoplus_{k \neq t} \alpha^k\end{aligned}$$

Let C be a constant

$$C = \alpha \oplus \bigoplus_{k \neq t} \alpha^k$$

then we can say

$$\alpha^H = \alpha^t \oplus C$$

and so

$$\begin{aligned}Pr[\alpha^H = X] &= Pr[\alpha^t \oplus C = X] \\ &= Pr[\alpha^t = X \oplus C]\end{aligned}$$

By the Unique Labels lemma, we know that each input to RF is unique. Thus, α^t is uniformly random.

This also means that the Helper's share is a uniformly random bitstring. Likewise, from the simulator

$$\begin{aligned}\alpha^H_{\text{sim}} &= \bigoplus_{k=1}^{n-1} \alpha^k \\ &= \alpha^t \oplus \bigoplus_{k \neq t} \alpha^k\end{aligned}$$

Let C^* be a constant

$$C^* = \bigoplus_{k \neq t} \alpha^k$$

then we can say

$$\alpha^H = \alpha^t \oplus C^*$$

and so

$$\begin{aligned}Pr[\alpha^H = X] &= Pr[\alpha^t \oplus C^* = X] \\ &= Pr[\alpha^t = X \oplus C^*]\end{aligned}$$

Thus, each of the Helper's shares is a uniformly random bitstring from both Chal-RF and Sim-RF. Since they are drawn from the same distribution, their probabilities are equal. □

Next, we must prove identical responses.

First, we show correctness of (R, F, P, S) from the challenger and the simulator. Recall that

$$\sigma_\pi^i(\rho_\pi, M) = \texttt{S\_Sign}(\texttt{hash}(\rho_\pi, M), X_\pi^i)$$

For correctness, all the shares held by the attacker XORed with the shares provided by the uncompromised trustee must XOR to the correct values of $\rho_\pi, \phi_\pi, \texttt{PATH}_\pi, \sigma_\pi(\rho_\pi, M)$:

$$\begin{aligned}
\rho_\pi &= R \oplus \bigoplus_{\neq t} \rho_\pi^i \\
\phi_\pi &= F \oplus \bigoplus_{\neq t} \phi_\pi^i \\
\texttt{PATH}_\pi &= P \oplus \bigoplus_{\neq t} \texttt{PATH}_\pi^i \\
\sigma_\pi(\rho_\pi, M) &= S \oplus \bigoplus_{\neq t} \sigma_\pi^t(\rho_\pi, M)
\end{aligned}$$

Which can be rewritten to give the correctness conditions we need:

$$R = \rho_\pi \oplus \bigoplus_{\neq t} \rho_\pi^i \tag{1}$$

$$F = \phi_\pi \oplus \bigoplus_{\neq t} \phi_\pi^i \tag{2}$$

$$P = \texttt{PATH}_\pi \oplus \bigoplus_{\neq t} \texttt{PATH}_\pi^i \tag{3}$$

$$S = \sigma_\pi(\rho_\pi, M) \oplus \bigoplus_{\neq t} \sigma_\pi^t(\rho_\pi, M) \tag{4}$$

Lemma 8 (Challenger Correctness). *The challenger's (R, F, P, S) satisfy the correctness conditions equations 1-4, above.*

Proof. Note that initial shares are computed in lines 6-21 of the challenger, and (R, F, P, S) in lines 29-36.

In line 10, the challenger sets

$$\phi_\pi = \texttt{P\_CHK}(K[1 \ldots n], \pi, \rho_\pi)$$

In lines 30-36, the challenger sets

$$\begin{aligned}
R &= \rho^t \\
F &= \phi^t \\
P &= \texttt{PATH}^t \\
h &= \texttt{hash}(M_{\rho,\pi}) \\
S &= \texttt{S_Sign}(X_\pi^t, h)
\end{aligned}$$

In lines 13-21, the challenger sets the initial shares so that

$$\begin{aligned}
\texttt{PATH}_\pi^H &= \texttt{PATH}_\pi \oplus \bigoplus_{k=1}^{n-1} \texttt{PATH}_\pi^k \\
\phi_\pi^H &= \phi_\pi \oplus \bigoplus_{k=1}^{n-1} \phi_\pi^k \\
\rho_\pi^H &= \rho_\pi \oplus \bigoplus_{k=1}^{n-1} \rho_\pi^k \\
X_\pi^H[i][j] &= X_\pi[i][j] \oplus \bigoplus_{k=1}^{n-1} X_\pi^k[i][j] \text{ For all } i, j
\end{aligned}$$

By the `S_Sign` Lemma, we know that when X satisfies the above condition

$$\sigma_\pi(\rho_\pi, M_\pi) = \bigoplus_{\text{all } k} \sigma_\pi^k(\rho_\pi, M_\pi)$$

Rearranging the terms, we find that

$$\begin{aligned}
R &= \rho_\pi \oplus \bigoplus_{\neq t} \rho_\pi^i \\
F &= \phi_\pi \oplus \bigoplus_{\neq t} \phi_\pi^i \\
P &= \texttt{PATH}_\pi \oplus \bigoplus_{\neq t} \texttt{PATH}_\pi^i \\
S &= \sigma_\pi(\rho_\pi, M) \oplus \bigoplus_{\neq t} \sigma_\pi^t(\rho_\pi, M)
\end{aligned}$$

□

Lemma 9 (Simulator Correctness). *The simulator's P and S also satisfy the correctness conditions equations 1-4, above.*

Proof. Note that initial shares are computed in lines 6-21 of the simulator, and (R, F, P, S) in lines 29-36.

In lines 29-36, the simulator sets

$$\begin{aligned}
R &= \rho_\pi^t \oplus \rho_\pi \\
\phi_\pi &= \texttt{P_CHK}(K[1 \ldots n], \pi, \rho_\pi) \\
F &= \phi_\pi^t \oplus \phi_\pi \\
P &= \texttt{PATH}_\pi^t \oplus \texttt{PATH}_\pi \\
h &= \texttt{hash}(\rho_\pi, M_\pi) \\
S &= \sigma_\pi(M_\pi) \oplus \texttt{S_Sign}(X_\pi^t, h)
\end{aligned}$$

In lines 16-21, the simulator sets the initial shares so that

$$\begin{aligned}
\texttt{PATH}_\pi^H &= \bigoplus_{k=1}^{n-1} \texttt{PATH}_\pi^k \\
\rho_\pi^H &= \bigoplus_{k=1}^{n-1} \rho_\pi^k \\
\phi_\pi^H &= \bigoplus_{k=1}^{n-1} \phi_\pi^k \\
X_\pi^H[i][j] &= \bigoplus_{k=1}^{n-1} X_\pi^k[i][j] \text{ For all } i, j
\end{aligned}$$

By the `S_Sign` Lemma, we know that when X satisfies the above condition

$$\sigma_\pi(\rho_\pi, M_\pi) = \bigoplus_{\text{all } k} \sigma_\pi^k(\rho_\pi, M_\pi)$$

Rearranging the terms, we again find that

$$\begin{aligned}
R &= \rho_\pi \oplus \bigoplus_{\neq t} \rho_\pi^i \\
F &= \phi_\pi \oplus \bigoplus_{\neq t} \phi_\pi^i \\
P &= \texttt{PATH}_\pi \oplus \bigoplus_{\neq t} \texttt{PATH}_\pi^i \\
S &= \sigma_\pi(\rho_\pi, M) \oplus \bigoplus_{\neq t} \sigma_\pi^t(\rho_\pi, M)
\end{aligned}$$

□

We now know that (R, F, P, S) satisfy the same correctness conditions from the challenger and simulator. The values of R, F, P depend only on the initial shares. However, the value of S also depends on the value of ρ used in the calculation.

In the simulator, ρ comes from the HBS-challenger. In the challenger, it comes from the attacker, sent as $\hat{\rho}$. If it is possible for the attacker to alter $\hat{\rho}$ to be different from ρ, then the challenger and simulator will not give identical responses. (Indeed, this would represent an actual attack on the scheme!)

The challenger and simulator both verify the correctness of ρ via a call to $\texttt{P\_CHK}(\phi, \pi, \rho)$; if it fails, they send back $\perp$ to the attacker, otherwise, they send back P, S. Since S will be different if $\hat{\rho} \neq \rho$, we need to show that if $\hat{\rho} \neq \rho$, the challenger and simulator will send back $\perp$.

This leads us to the following definition:

Definition 3 (BAD#2). *BAD event #2 occurs when, for some π in the transcript, $\hat{\rho_\pi} \neq \rho_\pi$ and $RF((10, \pi, \hat{\rho_\pi}), \ell) = \hat{\phi_\pi}[t]$, where t is the uncompromised trustee.*

Lemma 10. $Pr[BAD2] \leq Q \times 2^{-\ell}$

Proof. When $\hat{\rho_\pi} \neq \rho_\pi$, $RF((10, \pi, \hat{\rho_\pi}), \ell)$ represents a new query to the random function and so its output is a random ℓ-bit string. The attacker cannot make queries to the random function, and so can only try to guess this output. The probability that the attacker correctly guesses

a random ℓ-bit string is $2^{-\ell}$. In the transcript, a maximum of Q queries may be made. Thus, the probability of seeing at least one instance of the attacker correctly guessing this value is no more than $Q \times 2^{-\ell}$. □

Lemma 11 (Identical Responses). *Given identical initial shares and composite keys, and assuming no BAD events, the responses to each query from the attacker will be identical from the Chal-RF and Sim-RF.*

Proof. The correctness conditions for (R, F, P, S) allow only one possible value of (R, F, P, S) for a given set of initial shares, prefixes, queries, and composite key. Since both the simulator and challenger generate correct values for (R, F, P, S) given their initial shares, they must also generate identical (R, F, P, S) when starting with identical initial shares. As long as no BAD events occur, the attacker will never successfully get a response from either the challenger or simulator with a different prefix than the one that was initially generated (in the case of the challenger) or provided by the HBS-challenger (in the case of the simulator). This guarantees that all four values must be identical from the challenger and simulator. □

Lemma 12 (Chal-RF secure). *If no BAD events occur, then the existence of any attacker who can win the IC-Forge game against Chal-RF implies an attacker who can win the IC-Forge game against HBS-Challenger with the same number of queries. Thus,*

$$P^{\textit{IC-Forge}}_{\textit{Chal-RF}}(Q) \leq P^{\textit{HBS-Forge}}(Q)$$

Proof. The proof is straightforward.

1. The initial shares have equal probability from Chal-RF and Sim-RF. [Prefix Equal-Probability Shares]
2. Given identical initial shares and composite keys, and no BAD events, the responses from Chal-RF are identical to those from Sim-RF. [Prefix Identical Responses]
3. A forgery against Chal-RF implies a forgery against Sim-RF which implies a forgery against the underlying HBS scheme. [Forgery Transfer]
4. Thus, winning the IC-Forge game against Chal-RF implies winning the forgery game against HBS-challenger, assuming no BAD events. [IC-Forge Lemma]

□

We now bound this inequality from the other side:

Lemma 13 (HBS-Forge bound).

$$P^{\textit{HBS-Forge}} = P^{\textit{IC-Forge}}_{\textit{Chal-RF}}$$

Proof. Suppose we have an algorithm HBS-att which wins the HBS-Forge game with probability P. We can construct an algorithm chal-att to win the IC-Forge game with the same probability and same number of queries. The algorithm works as follows:

1. After the initial shares are sent, we provide the public key to HBS-att.
2. We pass each query from HBS-att to Chal, reconstruct the resulting signature, and provide it to HBS-att.
3. At the end, if HBS-att produces a forgery, we pass it along to Chal. We win IC-Forge against Chal with the same probability as HBS-att wins the HBS-Forge game.

This demonstrates that

$$P^{\text{IC-Forge}}_{\text{Chal-RF}} \geq P\text{HBS-Forge}$$

Since we also know from the Chal-RF secure lemma that

$$P^{\text{IC-Forge}}_{\text{Chal-RF}} \leq P\text{HBS-Forge}$$

we can conclude that

$$P^{\text{IC-Forge}}_{\text{Chal-RF}} = P\text{HBS-Forge}$$

□

Finally, we need to show that the PRF version is also secure.

PRF/RF Switching Recall that Chal and Sim are defined with calls to a PRF to derive the trustees' shares, while Chal-RF and Sim-RF are defined so that the uncompromised trustee's shares are generated by calls to a random function.

Any forgery attacker Att may be connected to Chal, Sim, Chal-RF, or Sim-RF. The number of chosen message queries made by A will be restricted to some parameter Q. Accordingly, we write Att(Chal, Q), Att(Chal-RF, Q), etc. Below, we will consider probabilities such as $P^{\text{IC-Forge}}_{\text{Att(Chal)}}$ and $P^{\text{IC-Forge}}_{\text{Att(Chal-RF)}}$ to indicate the probability of Att to succeed in generating an IC-Forge with the connection specified.

Similarly, we consider a distinguisher Dist, which may be connected either to a PRF or a RF, which we write as Dist(PRF, q) and Dist(RF, q), where q is an upper bound on the number of queries Dist can make to the function at hand.

Definition 4 (Advantage of a distinguisher). *For a distinguisher Dist, we write $\mathcal{A}^{PRF}_{Dist}(q)$ for the advantage of Dist in distinguishing a PRF from a random function, making at most q queries to the function at hand:*

$$\mathcal{A}^{PRF}_{Dist}(q) = \left| P\left[Dist(RF,\ q){=}1\right] - P\left[Dist(PRF,\ q){=}1\right] \right|.$$

In order to reason about this advantage, we define two more algorithms, Chal-F and Sim-F:

Definition 5 (Chal-F, Sim-F). *Chal-F and Sim-F are defined only within the context of a PRF/RF distinguishing game, in which we have access to the PRF/RF oracle.*

Let Chal-F be a variant of the challenger in which each call to $\texttt{PRF}_{K[t]}(\alpha, \ell)$ (that is, each PRF *call producing the uncompromised trustee's shares) is replaced with a call to the PRF/RF oracle. Thus, when the oracle is responding from a PRF, this is an instance of Chal, whereas when the oracle is responding from a random function, this is an instance of Chal-RF.*

Let Sim-F be a variant of the simulator in which each call to $\texttt{PRF}_{K[t]}(\alpha, \ell)$ (that is, each PRF *call producing the uncompromised trustee's shares) is replaced with a call to the PRF/RF oracle. Thus, when the oracle is responding from a PRF, this is an instance of Sim, whereas when the oracle is responding from a random function, this is an instance of Sim-RF.*

We now must define an additional bad event, which can only happen when Chal-F or Sim-F is being called within a PRF/RF distinguishing game: the PRF oracle could happen to select a PRF key that is already in use for one of the other trustees. This would lead to an immediate and trivial distinguisher.

Definition 6 (BAD#1). *BAD event #1 occurs in the distinguisher D during PRF/RF distinguishing game, when the PRF/RF oracle selects the same PRF key as one of the PRF keys selected by Chal-F or Sim-F for some trustee.*

Lemma 14. $Pr[BAD\#1] \leq (n-1) \times 2^{-\ell}$

Proof. By the definition of Chal-F and Sim-F, the $(n-1)$ compromised trustees' keys are all ℓ-bit strings chosen uniformly at random. By the definition of the PRF/RF game, the PRF key chosen by the oracle is also an ℓ-bit string chosen uniformly at random. The probability of a match between the oracle's key and one of the $n-1$ compromised trustees' keys is thus

$$Pr[BAD\#1] \leq (n-1) \times 2^{-\ell}$$

□

Lemma 15 (PRF/RF Switching). *For any attacker A, let $P^{A}_{Chal}(Q)$ be the probability A produces a forgery against Chal, and $P^{A}_{Chal\text{-}RF}(Q)$ the probability that A produces a forgery against Chal-RF. Then, assuming no bad events, a distinguisher D exists, which just runs A(Chal-F), thus making $Q_{\mathtt{PRF}}$ queries to F, and gaining the advantage $\mathcal{A}^{\mathtt{PRF}}_{D}(Q_{\mathtt{PRF}})$, such that*

$$\left| P^{IC\text{-}Forge}_{A(Chal)}(Q) - P^{IC\text{-}Forge}_{A(Chal\text{-}RF)}(Q)) \right| \leq \mathcal{A}^{\mathtt{PRF}}_{D}(Q_{\mathtt{PRF}}).$$

Corollary 1. *If, assuming no bad events, for all feasible distinguishers D^*, we assume an upper bound $\mathcal{A}^{\mathtt{PRF}}_{D^*}(Q_{\mathtt{PRF}}) \leq \epsilon$ for the advantage, then*

$$P^{IC\text{-}Forge}_{A(Chal)}(Q) - P^{IC\text{-}Forge}_{A(Chal\text{-}RF)}(Q)) \leq \epsilon.$$

Furthermore, if we assume ϵ to be negligible – as required by standard security assumptions for PRFs –, then the security of Chal and Chal-RF is the same, up to a negligible difference.

Proof (PRF/RF Switching). Consider two distinguishers, D_0 and D_1:

1. Both D_0 and D_1 run Att(Chal-F).
2. D_0 outputs 0 if Att(Chal-F) succeeds in producing a forgery and 1 if Att(Chal-F) fails.
   D_1 outputs 1 if Att(Chal-F) succeeds in producing a forgery and 0 if Att(Chal-F) fails.

Observe that

$$\mathcal{A}^{\mathrm{PRF}}_{D_0}(Q) = P^{A}_{\mathrm{Chal}}(Q) - P^{A}_{\text{Chal-RF}}(Q).$$

I.e., $\mathcal{A}^{\mathrm{PRF}}_{D_0}(Q) \geq 0$ if

$$P^{\text{IC-Forge}}_{A(\mathrm{Chal})}(Q) \geq P^{\text{IC-Forge}}_{A(\text{Chal-RF})}(Q).$$

Similarly, $\mathcal{A}^{\mathrm{PRF}}_{D_1}(Q) > 0$ if $P^{\text{IC-Forge}}_{A(\text{Chal-RF})}(Q) > P^{\text{IC-Forge}}_{A(\mathrm{Chal})}(Q)$. □

Lemma 16 (Security With No Bad Events). *Assuming no BAD events:*

$$P^{IC\text{-}Forge}_{Chal}(Q) \leq \mathcal{A}^{HBS\text{-}Forge}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}})$$

Proof. By the PRF/RF Switching Lemma, we know that if no BAD events occur,

$$P^{\text{IC-Forge}}_{\text{Chal}}(Q) \leq P^{\text{IC-Forge}}_{\text{Chal-RF}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}})$$

By the Chal-RF Secure lemma, we know that if no BAD events occur,

$$P^{\text{IC-Forge}}_{\text{Chal-RF}}(Q) \leq P^{\text{HBS-Forge}}(Q)$$

Thus, we can say that if no BAD events occur,

$$P^{\text{IC-Forge}}_{\text{Chal}}(Q) \leq P^{\text{HBS-Forge}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}})$$

□

In words, this tells us that as long as no BAD events occur, there is no attacker that can win the IC-Forge game against our scheme with higher probability than the probability of the best attacker winning the HBS-Forge game, plus the advantage of the best possible attacker in distinguishing the PRF from a random function.

B.5 Main Theorem Proof

We can now prove our Main Theorem on the security of the scheme. Recall that this bounds the ability of any attacker to carry out incomplete-coalition forgeries against our n-of-n signature scheme in terms of the ability of any attacker to either forge signatures in the underlying HBS or distinguish a $\mathtt{PRF}$ from a random function. The bound is written:

$$P^{\text{IC-Forge}}(Q) \leq P^{\text{HBS-Forge}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}}(Q)) + (n-1) \cdot 2^{-\ell} + Q \cdot 2^{-\ell}.$$

Proof. By the Security With No Bad Events lemma, we know that if no bad events occur, our security bound is

$$P^{\text{IC-Forge}}_{\text{Chal}}(Q) \leq P^{\text{HBS-Forge}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}})$$

By Lemma 14, we know that

$$P^{\text{BAD\#1}} \leq (n-1) \cdot 2^{-\ell}$$

By Lemma 10, we know that

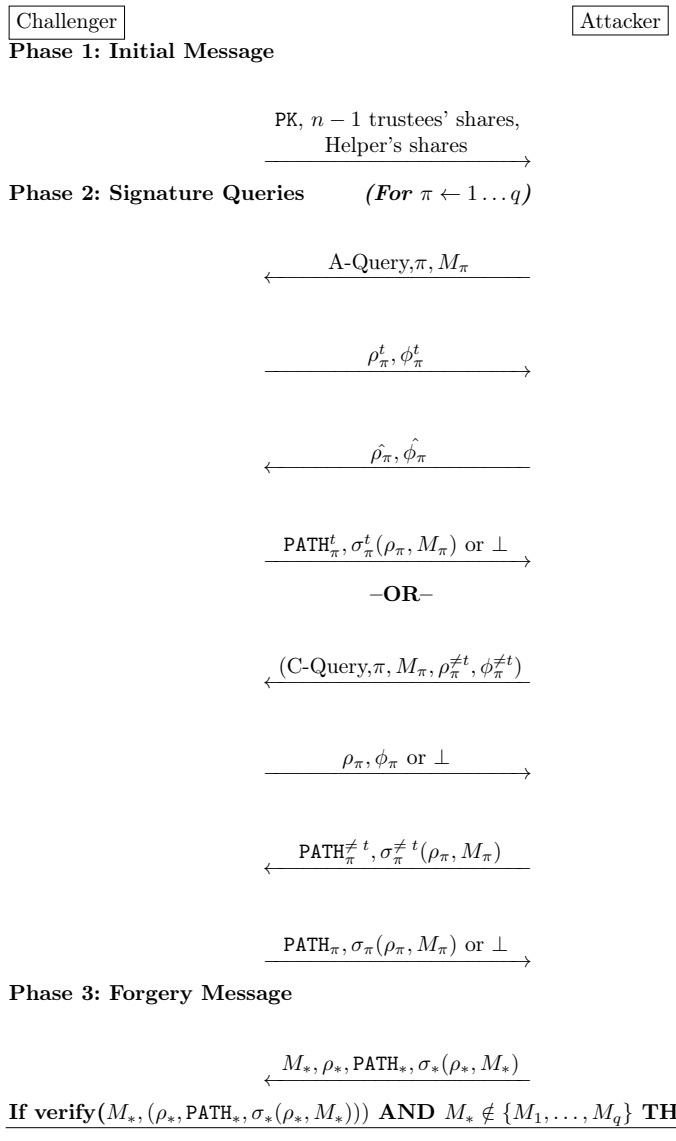
$$P^{\text{BAD\#2}} \leq Q \cdot 2^{-\ell}$$

Applying the union bound, we get:

$$\begin{aligned} P^{\text{IC-Forge}}_{\text{Chal}}(Q) &\leq P^{\text{HBS-Forge}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}}) + Pr[\text{BAD\#1}] + Pr[\text{BAD\#2}] \\ &\leq P^{\text{HBS-Forge}}(Q) + \mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}}) + (n-1)2^{-\ell} + Q2^{-\ell} \end{aligned}$$

□

For any plausible parameters of the scheme, $(n-1)2^{-\ell} + Q2^{-\ell}$ is negligibly small. If the $\mathtt{PRF}$ is secure, then by definition, $\mathcal{A}^{\mathtt{PRF}}(Q_{\mathtt{PRF}})$ is also negligibly small. Thus, our scheme's security is approximately the same a the security of the underlying hash-based signature scheme. Specifically, an attacker who can sign messages without a complete coalition in our scheme can also, with only negligibly smaller probability of success, forge messages for the underlying hash-based signature scheme.



Game 2: The Incomplete-Coalition Forgery Game with Prefixes

1: **function** CHAL(Q)
// *Generate composite key.*
2: $\mathtt{PK}, \mathtt{PATH}_{\text{all}}, X_{\text{all}} \leftarrow \text{KeyGen}()$
// *Find out which trustee to compromise*
3: Send $\mathtt{PK}$ to attacker.
4: Receive t from attacker.
// *Generate n pseudorandom shares*
5: $K[1 \ldots n]$ chosen randomly w/o dups.
6: **for** $k \leftarrow 1 \ldots n$ **do**
7: **for** each π **do**
8: $\mathtt{PATH}_\pi^k \leftarrow \mathtt{PRF}_{K[k]}((1, \pi), \ell)$
9: $\rho_\pi^k \leftarrow \mathtt{PRF}_{K[k]}(3, \pi), \ell)$
10: $\phi_\pi^k \leftarrow \mathtt{PRF}_{K[k]}(4, \pi), \ell)$
11: For each i, j:
12: $X_\pi^k[i][j] \leftarrow \mathtt{PRF}_{K[k]}((2, \pi, i, j), \ell)$
// *Generate Helper shares.*
13: **for** each π **do**
14: $\rho_\pi \leftarrow\$ \{0,1\}^\ell$
15: $\phi_\pi \leftarrow \mathtt{P\_CHK}(K[1 \ldots n], \pi, \rho_\pi)$
16: **for** each π **do**
17: $\mathtt{PATH}_\pi^H \leftarrow \mathtt{PATH}_\pi \oplus \bigoplus_{k=1}^n \mathtt{PATH}_\pi^k$
18: $\phi_\pi^H \leftarrow \phi_\pi \oplus \bigoplus_{k=1}^n \phi_\pi^k$
19: $\rho_\pi^H \leftarrow \rho_\pi \oplus \bigoplus_{k=1}^n \rho_\pi^k$
20: **for** each (i, j) **do**
21: $X_\pi^H[i][j] \leftarrow X_\pi[i][j] \oplus \bigoplus_{k=1}^n X_\pi^k[i][j]$
// *Give attacker requested shares*
22: Send $K[\neq t], \mathtt{PATH}_{\text{all}}^H, X_{\text{all}}^H, \phi_{\text{all}}^H, \rho_{\text{all}}^H$ to attacker.
// *Respond to A-queries*
23: **for** each signing query M_π **do**
// *Compute uncompromised trustee's response.*
24:
25:
26:
27: $R \leftarrow \rho^t$
28: $F \leftarrow \phi^t$
// *Use shares computed above to respond*
29: Send back R, F
30: Receive $\hat{\rho}, \hat{\phi}$ from attacker.
31: $P \leftarrow \mathtt{PATH}^t$
32: $h \leftarrow \mathtt{hash}(\hat{\rho}_\pi, M_\pi)$
33: $S \leftarrow \mathtt{S\_Sign}(X_\pi^t, h)$
// *Reject any altered prefix.*
34: **if** $\mathtt{P\_VER}(K[t], t, \hat{\phi}, \pi, \hat{\rho}) = 1$ **then**
35: Send back P, S
36: **else**
37: Send back $\perp$
// *Receive forgery message from attacker*
38: Receive $M_*, \rho_*, \mathtt{PATH}_*, \sigma_*$

1: **function** SIM(Q)
// *Get public key.*
2: Receive $\mathtt{PK}$ from HBS-Chall.
// ⟵ *See common code on left*
// ⟵ *See common code on left*
// *Generate fake Helper shares.*
13:
14:
15:
16: **for** each π **do**
17: $\mathtt{PATH}_\pi^H \leftarrow \bigoplus_{k=1}^n \mathtt{PATH}_\pi^k$
18: $\rho_\pi^H \leftarrow \bigoplus_{k=1}^n \rho_\pi^k$
19: $\phi_\pi^H \leftarrow \bigoplus_{k=1}^n \phi_\pi^k$
20: **for** each (i, j) **do**
21: $X_\pi^H[i][j] \leftarrow \bigoplus_{k=1}^n X_\pi^k[i][j]$
// ⟵ *See common code to left.*
// *Respond to A-queries*
23: **for** each signing query π, M_π **do**
// *Compute uncompromised trustee's shares.*
24: Send M_π to HBS-Challenger.
25: Receive $\rho_\pi, \mathtt{PATH}_\pi, \sigma_\pi(\rho_\pi, M_\pi)$
26: $R \leftarrow \rho_\pi^t \oplus \rho_\pi$
27: $\phi_\pi \leftarrow \mathtt{P\_CHK}(K[1 \ldots n], \pi, \rho_\pi)$
28: $F \leftarrow \phi_\pi^t \oplus \phi_\pi$
// ⟵ *See common code to left.*
31: $P \leftarrow \mathtt{PATH}_\pi^t \oplus \mathtt{PATH}_\pi$
32: $h \leftarrow \mathtt{hash}(\rho_\pi, M_\pi)$
33: $S \leftarrow \sigma_\pi(h) \oplus \mathtt{S\_Sign}(X_\pi^t, h)$
// ⟵ *See common code to left.*
// *Receive forgery message from attacker*
38: Receive $M_*, \rho_*, \mathtt{PATH}_*, \sigma_*$
39: Forward to HBS-Challenger

Algorithm 4: Challenger and Simulator Definitions