Merkle Tree Ladder Mode: Reducing the Size Impact of NIST PQC Signature Algorithms in Practice

Andrew Fregly[1,2][0000-0002-5760-9197], Joseph Harvey[1][0000-0002-9047-9320], Burton S. Kaliski Jr.[1][0000-0002-1233-5380] and Swapneel Sheth[1][0000-0002-0075-7914]

[1] Verisign Labs, Reston, VA 20190, USA
[2] afregly@verisign.com

Abstract. We introduce the *Merkle Tree Ladder (MTL) mode of operation* for signature schemes. MTL mode signs messages using an underlying signature scheme in such a way that the resulting signatures are *condensable*: a set of MTL mode signatures can be conveyed from a signer to a verifier in fewer bits than if the MTL mode signatures were sent individually. In MTL mode, the signer sends a shorter *condensed signature* for each message of interest and occasionally provides a longer *reference value* that helps the verifier process the condensed signatures. We show that in a practical scenario involving random access to an initial series of 10,000 signatures that expands gradually over time, MTL mode can reduce the size impact of the NIST PQC signature algorithms, which have signature sizes of 666 to 7856 bytes with example parameter sets, to a condensed signature size of 472 bytes per message. Even adding the overhead of the reference values, MTL mode signatures still reduce the overall signature size impact under a range of operational assumptions. Because MTL mode itself is quantum-safe, the mode can support long-term cryptographic resiliency in applications where signature size impact is a concern without limiting cryptographic diversity only to algorithms whose signatures are naturally short.

Keywords: Post-Quantum Cryptography, Digital Signatures, Merkle Trees, Modes of Operation.

1 Introduction

The transition to post-quantum cryptography under NIST's leadership [1] has resulted in a remarkable variety of new, fully specified cryptographic techniques [2] that have been assessed, through a public evaluation process, to resist cryptanalysis by both classical and quantum computers. NIST has also issued recommendations for two additional post-quantum signature schemes [3], which are also endorsed (along with one of the other techniques) in the latest U.S. National Security Systems suite [4]. The next step in the transition, as the various algorithms are standardized and incorporated into cryptographic libraries, is to upgrade applications to support them [5].

Applications of cryptography in the "pre-quantum" era have often been designed based on the characteristics of the cryptographic techniques available, one of which has been relatively small signature sizes (by post-quantum standards). Classical signature

sizes range from 64 to 256 bytes in typical examples [6]. The leading post-quantum signature algorithms in the NIST PQC project, in contrast, have minimum sizes that range from 666 to 7856 bytes with example parameter sets (see Tables 8 and 9 in [1]) — an order of magnitude (or two) increase.

Given the increasing sizes of all kinds of data, the relatively large size of the new signature algorithms won't necessarily present an obstacle to their adoption. But size concerns could still present a challenge in some environments, and for the greatest benefit, it will be helpful to have techniques that reduce the size impact. In addition, it would be desirable from the perspective of cryptographic diversity if these techniques could be applied to multiple families of signature algorithms.

Our focus in this paper is on reducing signature size impact in a practical scenario that we call *message series signing*. In this scenario, a signer continuously signs new messages and publishes the messages and their signatures. A verifier then continuously requests *selected* messages and verifies their signatures. As examples, the messages could be web Public-Key Infrastructure certificates [7], Domain Name System Security Extensions (DNSSEC) records [8] or signed certificate timestamps [9].

We are interested in a way for the signer to convey a set of signatures on messages of interest to the verifier in fewer bits than if the signatures were sent individually. We propose to do so through a process we call *condensation and reconstitution*. We show how to make a signature scheme *condensable* through a technique we call *Merkle Tree Ladder (MTL) mode*, named for both its relationship with Merkle trees [10] and with *modes of operation* of cryptographic techniques pioneered by NIST for encryption algorithms [11].

In brief, MTL mode constructs an evolving sequence of Merkle tree nodes, which we call *ladders*, from the sequence of messages being signed, then signs each ladder using the underlying signature scheme. An MTL mode signature has three parts: an authentication path from a message to a Merkle tree ladder node or "rung"; the ladder; and the signature on the ladder. A *condensed signature* conveys the authentication path; a *reference value* conveys a ladder and its signature. The signer sends the verifier a condensed signature and a handle pointing to a reference value; the verifier computes a *reconstituted signature* from the condensed signature and a suitable reference value, requesting a new reference value if needed, and then verifies the reconstituted signature. The condensation process evolves the authentication paths to maximize reuse of ladders and therefore minimize their size impact.

MTL mode improves upon the basic idea of forming a Merkle tree from a fixed set of messages and then signing the Merkle tree root in two important ways. First, the *message series can expand* as the signer continuously signs new messages without explicitly constructing new trees. Second, both the initial (uncondensed) signature and the reconstituted signature produced by MTL mode are *actual signatures* that can be verified by the MTL mode verification operation. Condensation and reconstitution are therefore *optional upgrades* that can be deployed incrementally.

Two other points are worth noting.

- MTL mode, like other Merkle tree techniques, is *based only on hash functions*. It's therefore *quantum-safe* under the same assumptions as hash-based signatures.

- Condensation and reconstitution are *public processes:* They involve only the signer's public key, not the private key. The processes therefore *don't impact the security of the underlying signature scheme* and they can be *performed by anyone.* Another party can perform the operations independently of the signer and verifier, which adds to deployment flexibility. (We note that the operations are different than compression / decompression in that the reconstituted signature may be different than the initial signature on the message.)

Summary of Our Contributions. (1) We provide a formal model for condensing and reconstituting signatures given a suitably constructed signature scheme; (2) We show how to use Merkle tree ladders to transform an arbitrary underlying signature scheme into a stateful tagged signature scheme suitable for condensation and reconstitution; and (3) We demonstrate that the transformation can reduce the size impact of NIST PQC signature algorithms in practice.

Organization. Section 2 gives the conventions for our Merkle tree constructions and defines Merkle tree ladders. Section 3 defines a tagged signature scheme. Section 4 then defines a condensation scheme. Sections 5 and 6 show how to transform an arbitrary signature scheme into a tagged signature scheme using MTL mode and how to condense the resulting signatures. Section 7 discusses the practical impact of our techniques on NIST PQC signature algorithms, Section 8 reviews related work, and Section 9 concludes the main body of the paper. Appendices provide additional details helpful for implementers as well as proof of one of the technical claims.

2 Merkle Tree Ladders

2.1 Conventions

For our constructions, we will use a Merkle graph — a variant of the classic Merkle tree where nodes may have multiple parents, so long as no node is its own ancestor — with the following additional restrictions:

1. Leaf nodes are indexed consecutively from 1 to N, where N is the number of data values being authenticated;
2. Every internal node has two child nodes, i.e., "left" and "right" subtrees;
3. The leaf node descendants of every internal node are consecutively indexed;
4. Every internal node has a different set of leaf node descendants; and
5. The left and right subtrees of an internal node have no leaf nodes in common.

Based on these properties, we can associate each node with a unique index pair L and R such that $1 \leq L \leq R \leq N$ where L is the lowest and R is the highest index of any leaf node descendants of the node. (For a leaf node, we let $L = R = i$ be the index of the leaf node itself.) The node with this index pair is denoted $[L{:}\,R]$; we say that $[L{:}\,R]$ *spans* leaf nodes L through R. We denote the hash value associated with this node as $V[L{:}\,R]$. We refer to the full set of leaf nodes associated with a set of data values, and their internal node ancestors in the graph, a *node set*.

A conventional binary Merkle tree where N is a power of 2 follows these restrictions with the further convention that the descendants of the internal node $[L{:}\,R]$ are adjacently and equally apportioned between its two child nodes, i.e., the children are $[L{:}\,M]$ and $[M+1{:}\,R]$ where $M = (L+R-1)/2$; the root node is $[1{:}\,N]$.

For our more general case, we take an approach similar to Certificate Transparency [9] in allowing N to be different than a power of 2, and we apportion descendants adjacently, but not necessarily equally, by choosing M as the (unique) integer between L and $R-1$ that is divisible by the largest power of 2. The successive *left* children of the node $[1{:}\,N]$, if present, and its descendants are thus always complete binary trees, but the node set and its ladders need not include $[1{:}\,N]$, depending on the rung strategy.

2.2 Ladders and Rungs

A subset of nodes that collectively spans every leaf node in a node set is a *Merkle tree ladder*; the individual ladder nodes are called *rungs*. Our motivation for the terminology is that the ladder provides a way to "climb the tree" and authenticate new leaf nodes.

The *authentication path* from a leaf node to a ladder is the set of sibling node hash values (in the usual Merkle tree sense) from the leaf node to an *associated rung* in the ladder that spans the leaf node.

A *rung strategy*, denoted $\mathcal{RS}$, specifies the process for updating a ladder as new leaf nodes are added and for choosing which rung in the ladder to use in an authentication path if more than one rung spans a given leaf node. We define one such strategy below, but others are possible (with different efficiency tradeoffs), including the traditional *single-rung strategy* where the ladder includes just the evolving root node $[1{:}\,N]$.

2.3 Hash Functions

In the following, H_{leaf} and H_{int} are cryptographically separated hash functions. $H_{\text{leaf}}(SID, i, d) \longrightarrow V$ maps a series identifier SID, an index i and a data value d to a 2ℓ-bit hash value V. $H_{\text{int}}\big(SID, L, R, V_{\text{left}}, V_{\text{right}}\big) \longrightarrow V$ maps a series identifier SID, two indexes L and R and two 2ℓ-bit hash values V_{left} and V_{right} to a 2ℓ-bit hash value V. (Here and elsewhere in this document, the integer ℓ is a variable security parameter indicating the desired cryptographic bit strength, e.g., $\ell = 128$.) The series identifier should be generated at random for each node set or derived pseudorandomly from another random value. Appendix A proposes instantiations of these functions and recommends the length of the series identifier.

2.4 Node Set Operations

We define the following operations for interacting with a node set according to a rung strategy $\mathcal{RS}$:

- *Node set initialization.* $\text{INITNODESET}_{\mathcal{RS}}(SID) \longrightarrow T_0$ returns a new node set T_0 associated with the series identifier SID.

- *Leaf node count.* $\text{GETLEAFNODECOUNT}_{\mathcal{RS}}(T) \longrightarrow N$ returns the number of leaf nodes in the node set T.
- *Leaf node addition.* $\text{ADDLEAFNODE}_{\mathcal{RS}}(T, d) \longrightarrow \langle T', \Lambda \rangle$ adds a leaf node corresponding to the data value d to the node set T, assigning the next leaf node index to it. The operation may also compute and add other nodes as needed to evolve the next ladder per the rung strategy $\mathcal{RS}$. ADDLEAFNODE returns the updated node set T' and the ladder Λ spanning the leaf nodes in the node set.
- *Authentication path construction.* $\text{GETAUTHPATH}_{\mathcal{RS}}(T, i) \longrightarrow \Pi$ returns the authentication path Π from the i^{th} leaf node in the node set T to its associated rung in the current ladder per the rung strategy $\mathcal{RS}$. The operation requires that $1 \leq i \leq N$ where N is the current leaf node count.
- *Authentication path verification.* $\text{CHECKAUTHPATH}_{\mathcal{RS}}(SID, i, N, N', d, \Pi, \Lambda) \longrightarrow b$ verifies that the i^{th} leaf node of a node set corresponds to a data value d using an authentication path Π and a ladder Λ, with the assumptions that Π is the authentication path from the i^{th} leaf node to its associated rung in the N^{th} ladder and Λ is the N'^{th} ladder. The operation requires that $1 \leq i \leq N' \leq N$. CHECKAUTHPATH returns $b = \text{TRUE}$ if the authentication path is valid and $b = \text{FALSE}$ otherwise.

ADDLEAFNODE is defined here so that the updated node set is returned as an output, consistent with the tagged signature scheme operations and the condensation operations returning the updated state. The operation could be redefined so that the node set is updated in place as discussed in Appendix B. As also shown in Appendix B, a node set with N leaf nodes can be represented with at most $2N - 1$ hash values (and in the case that the node set is used only for generating signatures, not for condensing them, with at most $\lfloor \log_2 N \rfloor + 1$ hash values).

2.5 Correctness

It is easy to see by the definition of a rung strategy that CHECKAUTHPATH will correctly verify the authentication path for a data value when the ladders for CHECKAUTHPATH and GETAUTHPATH are the same, i.e., $N' = N$, and the same SID is input to CHECKAUTHPATH as to INITNODESET. To make our schemes more efficient, however, we'd like verification also to be correct when $N' \neq N$. For this purpose, we will use the rung strategy defined next.

2.6 Binary Rung Strategy

In the binary rung strategy, denoted $\mathcal{BRS}$, the rungs in the N^{th} ladder Λ_N are defined based on the binary representation of N. Write

$$N = \sum_{j=1}^{B} 2^{\nu_j},$$

where the ν_j are the indexes of the ones bits in the binary representation of N from highest to lowest, so that $\lfloor \log_2 N \rfloor = \nu_1 > \nu_2 > \cdots > \nu_B \geq 0$. The rungs of the N^{th} ladder Λ_N are then

$$\langle [L_N(1): R_N(1)], \ldots, [L_N(B): R_N(B)] \rangle$$

where we define $R_N(0) = 0$ and for $j = 1$ to B, we set $L_N(j) = R_N(j-1) + 1$ and $R_N(j) = R_N(j-1) + 2^{v_j}$. It follows that the rung spans are adjacent and that $L_1(1) = 1$ and $R_N(B) = N$, so the ladder rungs collectively span every leaf node. Moreover, the descendants of each rung form a complete binary tree. The rung associated with the i^{th} leaf node is the unique rung $[L_N(B): R_N(B)]$ for which $L_N(B) \leq i \leq R_N(B)$. The number of rungs is $O(\log N)$. Fig. 1 gives an example.

Because the rung node index pairs can all be computed deterministically from N, as long as the value N is available, we don't need to include the index pairs in a representation of the ladder, just the rung node hash values.

Appendix B gives pseudocode for the node set operations for this strategy.

2.7 General Path Verification

Claim. For all positive integers i, N, N' where $i \leq N' \leq N$, if d_i is the data value corresponding to the i^{th} leaf node in a node set assembled using the binary rung strategy, $\Pi_{i,N}$ is the authentication path from the i^{th} leaf node to its associated rung in the N^{th} ladder and $\Lambda_{N'}$ is the N'^{th} ladder, then

$$\text{CHECKAUTHPATH}_{\mathcal{BRS}}\left(SID, i, N, N', d_i, \Pi_{i,N}, \Lambda_{N'}\right) = \text{TRUE}.$$

The proof is given in Appendix C.

3 Tagged Signature Schemes

We adopt the generalization of the framework proposed by Abe et al. [12] for one-time signatures where messages are associated with tags. Tagging provides a convenient way to divide messages signed under the same public/private key pair into multiple series, while providing cryptographic separation between the series. We allow a tagged signature scheme to be stateful. Following Yuan, Tibouchi and Abe [13], our scheme operations take the current state as an input and return the updated state as an output. (The scheme operations could be redefined so that the state is updated in place.)

An implementation may place constraints on the lengths of the tags, the number of different tags allowed, and the number of messages that may be associated with a given tag. In MTL mode, we allow the tag to be an arbitrary string.

3.1 Scheme Definition

A *tagged signature scheme* $\mathcal{TSS}$ is a stateful signature scheme where signatures are associated with tags. A tagged signature scheme has three operations:

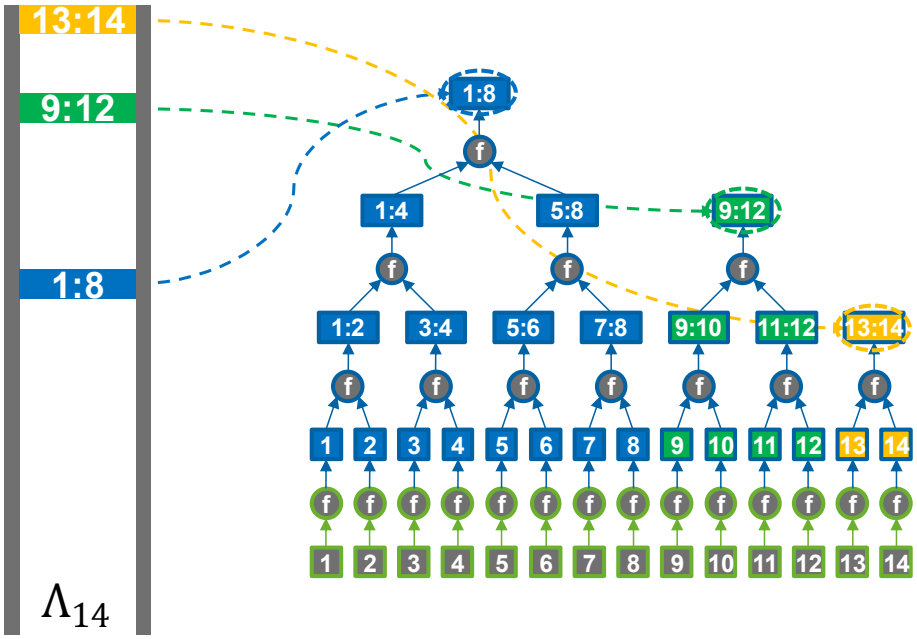

Fig. 1. Example of a Merkle tree ladder and node set following a binary rung strategy. Rungs [1:8], [9:12] and [13:14] collectively span all 14 leaf nodes.

- *Key pair generation.* $\text{KEYGEN}(1^{\ell}) \longrightarrow \langle pk, sk, st_0 \rangle$ generates a public/private key pair $\langle pk, sk \rangle$ with security parameter ℓ and returns the newly generated key pair and an initial state st_0.
- *Signature generation.* $\text{SIGN}_{sk}(m, \tau, st) \longrightarrow \langle \sigma, st' \rangle$ signs a message m and a tag τ with a private key sk under the state st and returns the resulting signature σ and an updated state st'. SIGN returns an error if the scheme does not support the tag τ.
- *Signature verification.* $\text{VERIFY}_{pk}(m, \tau, \sigma) \longrightarrow b$ verifies a signature σ on a message m and a tag τ with a public key pk. VERIFY returns $b = \text{TRUE}$ if the verification check is successful and $b = \text{FALSE}$ otherwise. We say that the tuple $\langle m, \tau, \sigma \rangle$ is *valid* if $\text{VERIFY}(m, \tau, \sigma) = \text{TRUE}$.

We define the *message series* for a tag τ as the sequence of messages $M_N^{\tau} = \langle m_1, \ldots, m_N \rangle$ input to successive calls to SIGN in association with τ, and the *signature series* for τ as the corresponding sequence of signatures $\Sigma_N^{\tau} = \langle \sigma_1, \ldots, \sigma_N \rangle$ returned.

3.2 Correctness and Unforgeability

The security objectives for a tagged signature scheme are similar to those for an ordinary signature scheme. We want the objectives to hold for a randomly generated

key pair, every supported tag τ and every polynomially bounded (in ℓ) message series M_N^τ (and corresponding signature series Σ_N^τ):

- *Correctness.* For all $m_i \in M_N^\tau$, if $\sigma \in \Sigma_N^\tau$ is the corresponding signature, then $\langle m_i, \tau, \sigma \rangle$ is valid.
- *Unforgeability.* For all messages $m \notin M_N^\tau$, it is computationally infeasible (in ℓ) for an adversary to produce a valid $\langle m, \tau, \sigma \rangle$ with more than a negligible probability of success, even if the adversary otherwise has full control of the messages and the tags input to the scheme operations.

We assume that the adversary respects the scheme's state, i.e., when calling an operation, the adversary inputs the state output by the previous operation (intervening calls may be made with different tags). State management is an important consideration and would benefit from a more complete treatment than given herein, including the impact on the underlying signature scheme and on MTL mode itself.

3.3 Signature Malleability

We allow a tagged signature scheme to be *malleable* [14] in the sense that given a valid tuple $\langle m, \tau, \sigma \rangle$, an adversary (and anyone else) may be able to produce a valid tuple $\langle m, \tau, \sigma' \rangle$ where $\sigma' \neq \sigma$. (Our definition of unforgeability doesn't require that $\sigma \in \Sigma_N^\tau$.)

While the condensation / reconstitution process (as well as MTL mode) demonstrates a beneficial use of signature malleability, the property also introduces potential vulnerabilities [15]. In particular, a signer using a malleable signature scheme cannot assume that a transaction has *not* been processed simply because the initial signature on the transaction does not appear in a transaction log. A malleated version of the signature could have been posted instead, i.e., a reconstituted signature in our case.

4 Signature Series Condensation Schemes

Given a signature series, we are looking for a way for the signer to convey a subset of the signatures to the verifier in fewer bits than if the signatures were sent individually. We envision the following arrangement (see Fig. 2):

- A signer signs a series of messages, producing *initial signatures* on the messages.
- *(Condensation.)* The signer or an intermediary produces a *condensed signature* and a *reference value handle* from an initial signature. These values are sent to the signer instead of the initial signature with messages of interest to the verifier. The values may depend on previous signatures produced by the signer.
- *(Reconstitution.)* The verifier or an intermediary produces a *reconstituted signature* from a *condensed signature* and a *reference value*. If the verifier or intermediary doesn't have a suitable reference value, it requests one based on the handle.
- The verifier then verifies the reconstituted signature.

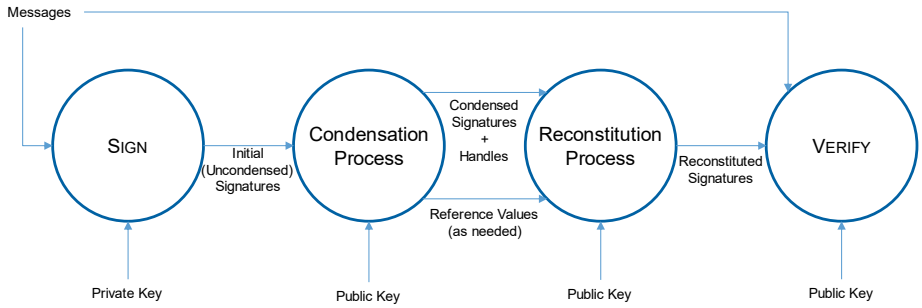

Fig. 2. Condensation and reconstitution processes applied to a signature scheme.

These operations involve access *only to the public key*, not the private key, so they can be performed by anyone, not just the signer and the verifier. For instance, a server that publishes messages and signatures on behalf of the signer could incorporate signatures into the condensation state, produce condensed signatures and reference value handles, and resolve reference value handles into reference values. Similarly, an agent that requests messages and signatures on behalf of the verifier could request condensed signatures and reference values. The mapping from handles to reference values could also be implemented by a database.

The handle provides a layer of *indirection*. Different condensed signatures may be associated with different reference values, and verifiers and their intermediaries may request reference values at different times. The handle indicates the specific reference value of interest.

Because we allow $\mathcal{TSS}$ to be malleable, the reconstituted signature on a message may be different than the initial signature.

Although we define condensation and reconstitution relative to a tagged signature scheme, the tagging is a convenience, making it possible for the signature scheme to support multiple signature series that can be condensed and reconstituted independently of one another under the same key pair. The processes could also be applied to a non-tagged signature scheme, as illustrated in Appendix D.

4.1 Scheme Definition

A *condensation scheme* $\mathcal{CS}$ has five operations defined relative to an associated tagged signature scheme $\mathcal{TSS}$. As in Section 3, the operations below could also be redefined so that the state is updated in place.

- *Initialization.* $\text{CONDENSEINIT}_{pk}() \longrightarrow st_0$ returns a new scheme state st_0.
- *Signature incorporation.* $\text{ADDINITSIG}_{pk}(\tau, \sigma, st) \longrightarrow \langle i, st' \rangle$ incorporates an initial signature σ associated with a tag τ into the state st and returns the signature index i for this signature (relative to τ) and the updated state st'.
- *Condensed signature production.* $\text{GETCONDENSEDSIG}_{pk}(\tau, i, st) \longrightarrow \langle \varsigma, \chi, st' \rangle$ produces a condensed version ς of the i^{th} signature associated with the tag τ under

the state st and returns the condensed signature ς, the associated reference value handle χ, and the updated state st'.

- *Reference value production.* $\text{GETREFVAL}_{pk}(\tau, \chi, st) \longrightarrow \langle \upsilon, st' \rangle$ produces the reference value υ associated with the tag τ and the handle χ under the state st and returns υ and the updated state st'.
- *Signature reconstitution.* $\text{RECONSTSIG}_{pk}(\tau, \varsigma, \upsilon) \longrightarrow \sigma'$ reconstitutes a signature σ′ from a condensed signature ς and a reference value υ associated with a tag τ and returns σ′. (Appendix E proposes an alternative stateful set of reconstitution operations that includes a check for reference value compatibility.)

4.2 Correctness

A condensation scheme has one security objective. As above, we want it to hold for a randomly generated key pair, every supported tag τ and every polynomially bounded (in ℓ) message series M_N^{τ}:

- *Correctness.* For all $m_i \in M_N^{\tau}$, if the corresponding signature series Σ_N^{τ} is incorporated into the condensation state with calls to CONDENSEINIT and ADDINITSIG, then a condensed signature ς and a reference handle χ are obtained from τ and i with a call to GETCONDENSEDSIG, then a reference value υ is obtained from τ and χ with a call to GETREFVAL, and finally a reconstituted signature σ′ is obtained τ, ς and χ with a call to RECONSTSIG, then $\langle m_i, \tau, \sigma' \rangle$ is valid under $\mathcal{TSS}$.

We didn't include unforgeability as a security objective because a condensation scheme doesn't introduce any new access to the signer's private key. We again assume for simplicity that the calls respect the schemes state (with the possibility of intervening calls for other tags, or to get other condensed signatures and/or reference values).

4.3 Reference Value Compatibility

We'd like the verifier to be able to reconstitute a signature from a condensed signature and a previously produced reference value, i.e., one obtained with a call to GETREFVAL from τ and a reference value handle χ' older than the one returned by GETCONDENSEDSIG. A scheme with this property is *reference value compatible*.

5 Merkle Tree Ladder Mode

We now describe a general technique that can be applied to any signature scheme $\mathcal{S}$ to transform it into a tagged signature scheme that can then be condensed, asymptotically, to the size of a Merkle tree authentication path. Our basic approach is to construct an *evolving* sequence of Merkle tree ladders from the message series and sign each ladder using the underlying signature scheme. We call the transformation *Merkle Tree Ladder (MTL) mode*.

MTL mode operations follow the binary rung strategy $\mathcal{BRS}$ described in Section 2.6. Because of the general path verification property of this rung strategy, the verification operation will accept both initial MTL mode signatures and reconstituted MTL mode signatures. The reference value υ is identified by the *MTL mode reference value handle* $\chi = N'$.

We chose to transform the underlying signature scheme into a tagged signature scheme so that an application can sign multiple message series with the same public/private key pair and have cryptographic separation between the series. We could have instead transformed $\mathcal{S}$ into a non-tagged, stateful signature scheme, but the mode would have directly supported only a single message series.

MTL mode applied to an underlying signature scheme $\mathcal{S}$ has the following profile:

- *Public key.* $pk = \langle pk^{\mathcal{S}}, S \rangle$ where $pk^{\mathcal{S}}$ is a public key for $\mathcal{S}$ and S is a 2ℓ-bit seed;
- *Private key.* $sk = sk^{\mathcal{S}}$ where $sk^{\mathcal{S}}$ is the corresponding private key for $\mathcal{S}$;
- *Signature:* $\sigma = \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'}, d_i^* \rangle$ where SID is a series identifier, c_i is a randomizer, i, N and N' are indexes, $\Pi_{i,N}$ is an authentication path, $\Lambda_{N'}$ is a ladder, $\sigma^{\mathcal{S}}_{N'}$ is a signature under $\mathcal{S}$ and d_i^* is an optional data value. For an initial signature, $i = N = N'$ and d_i^* is included in σ. For a reconstituted signature, as discussed in Section 6, $i \leq N \leq N'$ and d_i^* is omitted.

Note that SID could be omitted from the signature because it can be computed from the seed and the tag. Including SID makes it possible to start the node set operations before the public key is obtained, provided that other scheme parameters are already known.

5.1 Hash Functions

H_{msg} and H_{SID} are additional cryptographically separated hash functions. $H_{\text{SID}}(S, \tau) \longrightarrow SID$ maps a 2ℓ-bit seed S and a tag τ to a series identifier SID. $H_{\text{msg}}(SID, i, m, c) \longrightarrow d$ maps a series identifier SID, an index i, a message m and a randomizer c to a 2ℓ-bit data value. We use a message-specific randomizer to base the security of MTL mode on second-preimage-resistance than collision-resistance, similar to all of the example signature schemes except CRYSTALS–Dilithium. (Having randomizers in both parts also protects against adversarial access either to MTL mode or to the underlying scheme.) The randomizer should be generated randomly or derived pseudorandomly from another random value. Appendix A proposes instantiations of these functions and also recommends lengths for the tag, series identifier and randomizer.

5.2 Tag Constraints

The tag τ input to MTL mode operations may be a bit string of any length. The mode doesn't specify any constraints on the number of different tags allowed or the number of messages that may be associated with a given tag. (Implementations may have practical limits.)

5.3 Scheme State

The scheme state includes a security parameter ℓ, a seed S and a set of *series tuples* $\langle SID, T\rangle$ where SID is a series identifier and T is the node set for the series. (We maintain the formality of taking the current state as input to the scheme operation and returning the updated state as output, with the caveat that the large size of the state for MTL motivates the alternative suggested in Section 3 of updating the state in place.)

Although we've defined MTL mode in terms of a stateless underlying signature scheme, the mode can also readily be applied to a stateful underlying signature scheme by maintaining the state of the underlying scheme in the mode's scheme state.

5.4 Key Pair Generation

$\text{KEYGEN}(1^{\ell}) \longrightarrow \langle pk, sk, st_0\rangle$ generates a key pair $\langle pk^{\mathcal{S}}, sk^{\mathcal{S}}\rangle$ with security parameter ℓ using $\mathcal{S}$, generates a random 2ℓ-bit seed S, creates a new state st_0 including ℓ, S, and an empty set of series tuples, formats the public key pk as

$$pk \Leftarrow \langle pk^{\mathcal{S}}, S\rangle,$$

sets the private key $sk = sk^{\mathcal{S}}$, and returns pk, sk and st_0.

5.5 Tagged Signature Generation

$\text{SIGN}_{sk}(m, \tau, st) \longrightarrow \langle \sigma, st'\rangle$ computes a series identifier SID as

$$SID \coloneqq H_{\text{SID}}(S, \tau),$$

where S is the seed in the state st. Let $\langle SID, T\rangle$ be the series tuple for SID in st. If st doesn't have a series tuple for SID, then SIGN adds a new tuple $\langle SID, T\rangle$ to st where the node set T is initialized as

$$T := \text{INITNODESET}_{\mathcal{BRS}}(SID).$$

SIGN then obtains the leaf node count as

$$N := \text{GETLEAFNODECOUNT}_{\mathcal{BRS}}(T)$$

and sets $i = N + 1$. SIGN next generates a randomizer c_i, and computes a data value d_i via randomized hashing and incorporates it into the node set, obtaining an updated node set T' and a ladder Λ_i:

$$d_i := H_{\text{msg}}(SID, i, m, c_i);$$
$$\langle T', \Lambda_i\rangle := \text{ADDLEAFNODE}_{\mathcal{BRS}}(T, d_i).$$

SIGN then obtains an authentication path $\Pi_{i,i}$ from the leaf node to the ladder as

$$\Pi_{i,i} := \text{GETAUTHPATH}_{\mathcal{BRS}}(T', i).$$

SIGN then computes an underlying signature $\sigma_i^{\mathcal{S}}$ on Λ_i as

$$\sigma_i^{\mathcal{S}} := \mathcal{S}.\text{SIGN}_{sk}(\langle SID, 1, i, \Lambda_i \rangle)$$

and formats the signature σ as

$$\sigma \Leftarrow \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^{\mathcal{S}}, d_i \rangle.$$

SIGN finally updates the state with the updated series tuple $\langle SID, T' \rangle$ and returns σ and the updated state st'.

5.6 Tagged Signature Verification

$\text{VERIFY}_{pk}(m, \tau, \sigma) \longrightarrow b$ parses the public key pk and the signature σ as

$$pk \Longrightarrow \langle pk^{\mathcal{S}}, S \rangle;$$
$$\sigma \Longrightarrow \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}}, d_i^* \rangle,$$

then checks that $SID = H_{\text{SID}}(S, \tau)$ and verifies the underlying signature $\sigma_{N'}^{\mathcal{S}}$ on $\Lambda_{N'}$ as

$$b^{\mathcal{S}} := \mathcal{S}.\text{VERIFY}_{pk^{\mathcal{S}}}\left(\langle SID, 1, N', \Lambda_{N'} \rangle, \sigma_{N'}^{\mathcal{S}}\right).$$

If the parsing or the check fails, or if $b^{\mathcal{S}} = \text{FALSE}$, then VERIFY returns FALSE. VERIFY then computes a data value d_i via randomized hashing:

$$d_i = H_{\text{msg}}(SID, i, m, c_i),$$

verifies the presumed authentication path $\Pi_{i,N}$ from the leaf node corresponding to d_i to the ladder $\Lambda_{N'}$ as

$$b \coloneqq \text{CHECKAUTHPATH}_{\mathcal{BRS}}\left(SID, i, N, N', d_i, \Pi_{i,N}, \Lambda_{N'}\right)$$

and returns b. For completeness, VERIFY may also check that the final signature component d_i^*, if present, matches d_i.

5.7 Correctness and Unforgeability

Correctness follows from the general path verification property of the binary rung strategy (where we set $i = N = N'$) and from the correctness of $\mathcal{S}$.

Unforgeability can be shown with the usual arguments for a Merkle tree signature scheme. In particular, for an adversary to produce a valid tuple $\langle m, \tau, \sigma \rangle$ where the message m has not been input to SIGN with tag τ, the adversary must do one of the following: (i) forge an underlying signature on a ladder under $\mathcal{S}$; (ii) produce a second authentication path to a previously signed ladder; (iii) produce a second data value that maps to a leaf hash value in a previous node set; (iv) produce a second randomizer / message pair that maps to the data value in a previous signature; or (v) produce a second tag that maps to the same series identifier as a previously signed tag. If $\mathcal{S}$ is unforgeable

then the first is infeasible, and if H_{int}, H_{leaf}, H_{msg} and H_{SID} are second-preimage-resistant (as appropriately defined) then the rest are infeasible.

Because we include SID and the node indexes in the inputs to $\mathcal{S}.\text{SIGN}$, H_{int} and H_{leaf}, and because a node set is "append only" (node hash values don't change), an adversary can target only one node in one node set per hash function invocation, making MTL mode secure in a multi-user (and multi-tag) setting. A full security proof would need to consider the interactions of the MTL mode hash function instantiations with the underlying signature scheme (see Appendix A).

6 Condensing and Reconstituting MTL Mode Signatures

We now show how to condense and reconstitute the signatures produced in the previous section, following the framework in Section 4.

Observe that the signature series for a given tag has enough information to reconstruct the full node set for the series *without access to the messages*. In particular, each signature includes the relevant leaf node, either as part of a ladder or as an explicit component. The leaf nodes can then be incorporated into the node set in the order in which they were computed to reconstruct the node set.

The condensation scheme produces condensed signatures relative to the *current* node set and thus to the current ladder. The condensed signature includes the authentication path; the reference values include the ladder. Multiple condensed signatures can thus reference the same ladder, making the reference values reusable.

6.1 Scheme State

The condensation scheme state includes a seed S and a set of *signed series tuples* $\langle SID, T, \bar{c}, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ where SID is a series identifier, T is the node set for the series, $\bar{c}$ is the sequence of randomizers for the series, $\overline{\Lambda}$ is the sequence of ladders , and $\Sigma^{\mathcal{S}}$ is the sequence of underlying signatures. (Similar to Section 5, the operations below could also be defined so that the state is updated in place.)

6.2 Initialization

$\text{CONDENSEINIT}_{pk}() \longrightarrow st_0$ parses the public key pk as

$$pk \Longrightarrow \langle pk^{\mathcal{S}}, S \rangle,$$

then creates a new state st_0 including the seed S and an empty set of signed series tuples.

6.3 Signature Incorporation

$\text{ADDINITSIG}_{pk}(\tau, \sigma, st) \longrightarrow \langle i, st' \rangle$ parses the signature σ as

$$\sigma \Longrightarrow \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^{\mathcal{S}}, d_i \rangle.$$

Let $\langle SID, T, \bar{c}, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ be the current signed series tuple for SID. If the state st doesn't include a signed series tuple for SID, then ADDINITSIG adds a new tuple $\langle SID, T, \bar{c}, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ to st where the node set T is initialized as

$$T := \text{INITNODESET}_{\mathcal{BRS}}(SID)$$

and where $\bar{c} = \overline{\Lambda} = \Sigma^{\mathcal{S}} = \emptyset$, where $\emptyset$ denotes an empty sequence ADDINITSIG then obtains the leaf node count as

$$N := \text{GETLEAFNODECOUNT}_{\mathcal{BRS}}(T)$$

and sets $i = N + 1$ (which should be the same as the i in the signature). ADDINITSIG next incorporates the data value d_i into the node set, obtaining an updated node set T' and a ladder Λ_i^* (which should be the same as the Λ_i in the signature) as

$$\langle T', \Lambda_i^* \rangle := \text{ADDLEAFNODE}_{\mathcal{BRS}}(T, d_i).$$

ADDINITSIG then appends the randomizer c_i to $\bar{c}$, producing $\bar{c}'$; appends the ladder Λ_i to $\overline{\Lambda}$, producing $\overline{\Lambda}'$; and appends the underlying signature $\sigma_i^{\mathcal{S}}$ to $\Sigma^{\mathcal{S}}$, producing $\Sigma^{\mathcal{S}'}$. ADDINITSIG finally updates the state with the updated signed series tuple $\langle SID, T', \bar{c}', \overline{\Lambda}', \Sigma^{\mathcal{S}'} \rangle$ and returns the index i and the updated state st'.

6.4 Condensed Signature Production

$\text{GETCONDENSEDSIG}_{pk}(\tau, i, st) \longrightarrow \langle \varsigma, \chi, st' \rangle$ computes a series identifier SID as

$$SID := H_{\text{SID}}(S, \tau),$$

where S is the seed in the state st. Let $\langle SID, T, \bar{c}, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ be the current signed series tuple for SID. If st doesn't include a signed series tuple for SID, then GETCONDENSEDSIG returns an error. GETCONDENSEDSIG then obtains the current leaf node count N and computes an authentication path $\Pi_{i,N}$ from the i^{th} leaf node to the current ladder as

$$N := \text{GETLEAFNODECOUNT}_{\mathcal{BRS}}(T);$$
$$\Pi_{i,N} := \text{GETAUTHPATH}_{\mathcal{BRS}}(T, i).$$

GETCONDENSEDSIG next formats the condensed signature ς as

$$\varsigma \Longleftarrow \langle SID, c_i, i, N, \Pi_{i,N} \rangle$$

where c_i is the i^{th} randomizer in the randomizer sequence $\bar{c}$. GETCONDENSEDSIG sets the reference value handle $\chi = N$ and returns ς, χ and the (unchanged) state $st' = st$.

6.5 Reference Value Production

$\text{GETREFVAL}_{pk}(\tau, \chi, st) \longrightarrow \langle \upsilon, st' \rangle$ computes a series identifier SID as

$$SID \coloneqq H_{\text{SID}}(S, \tau),$$

where S is the seed in the state st. Let $\langle SID, T, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ be the current signed series tuple for SID. If st doesn't include a signed series tuple for SID, then GETREFVAL returns an error. GETREFVAL then sets $N' = \chi$ and formats the reference value υ as

$$\upsilon \Leftarrow \langle N', \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'} \rangle$$

where $\Lambda_{N'}$ is the N'^{th} ladder in the ladder sequence $\overline{\Lambda}$ and $\sigma^{\mathcal{S}}_{N'}$ is the N'^{th}underlying signature in the underlying signature sequence $\Sigma^{\mathcal{S}}$. The operation returns υ and the (unchanged) state $st' = st$.

6.6 Signature Reconstitution

$\text{RECONSTSIG}_{pk}(\tau, \varsigma, \upsilon) \longrightarrow \sigma'$ parses the condensed signature ς and the reference value υ as

$$\varsigma \Longrightarrow \langle SID, c_i, i, N, \Pi_{i,N} \rangle;$$

$$\upsilon \Longrightarrow \langle N', \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'} \rangle$$

RECONSTSIG then formats the reconstituted signature σ' as

$$\sigma' \Leftarrow \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'}, \emptyset \rangle.$$

and returns σ'. (The data value is empty as it's not needed for signature verification.)

6.7 Correctness and Reference Value Compatibility

We want to show that if a reconstituted signature $\sigma' = \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'}, \emptyset \rangle$ for a message m_i and a tag τ is obtained from RECONSTSIG following the condensation / reconstitution process, and $i \leq N' \leq N$, then $\langle m_i, \tau, \sigma' \rangle$ is valid. This follows from the general path verification property of the binary rung strategy in Section 2.6 and from the correctness of $\mathcal{S}$. We rely on the fact that CHECKAUTHPATH can verify the authentication path $\Pi_{i,N}$ using any ladder $\Lambda_{N'}$ where $i \leq N' \leq N$. Any handle returned by GETCONDENSEDSIG after the i^{th} signature is incorporated into the state leads to a ladder $\Lambda_{N'}$ that is compatible with the authentication path $\Pi_{i,N}$.

6.8 Error Checks

We haven't included many error checks in these operations because they're not necessary for the formal definition of correctness, which assumes that inputs are provided correctly (the only question is whether the outputs are computed correctly). In practice, an implementation should check that inputs are in valid ranges, that series

identifiers are consistent with the tags, that authentication paths and ladders are consistent with data values, and possibly even that signatures can be verified.

We've also left out optimizations such as caching the results of previous calls to GETCONDENSEDSIG and GETREFVAL in the condensation state, recomputing the ladders from the node set rather than storing them separately and removing older signatures and ladders from the state when they're no longer likely to be requested because the condensation process has moved on to newer reference values. An implementation should take these optimizations into account as well.

7 Practical Impact of MTL Mode

We now show that MTL mode can reduce the size impact of the NIST PQC signature algorithms and other signature algorithms with large signature sizes in practice.

For simplicity, we divide our operations into *iterations*, and we assume that prior to the first iteration, the signer has signed an initial message series with N_0 messages all in association with the same tag τ and the verifier has received the reference value υ_{N_0}. We further assume that during each iteration, the signer signs α additional messages and the verifier requests condensed signatures on ρ messages, where the signatures of interest are randomly and independently chosen among the signatures generated up to and including that iteration.

If the verifier is interested in a signature on message $m_{i'}$ and $i' \leq N_0$, then because of MTL mode's reference value compatibility, the verifier can produce a valid reconstituted signature from a newly received condensed signature corresponding to $m_{i'}$ and the reference value υ_{N_0}. (We assume that the condensed signature was produced relative to the *current* message series, which has $N \geq N_0$ messages.) If $i' > N_0$, however, then the verifier will need to request a new reference value.

7.1 Condensed Signatures Per Reference Value

Under our operational assumptions, the probability that a verifier *doesn't* need to request a new reference value during any of the first κ iterations is the product

$$\prod_{t=1}^{\kappa} \left(\frac{N_0}{N_0 + t\alpha}\right)^{\rho} = \prod_{t=1}^{\kappa} \left(\frac{1}{1 + t\alpha/N_0}\right)^{\rho}.$$

Assuming N_0 is much larger than ρ and α, we can approximate this probability as:

$$\prod_{t=1}^{\kappa} \exp(-t\alpha\rho/N_0) \approx \exp(-\kappa^2\alpha\rho/2N_0).$$

(The analysis is similar to the Birthday Paradox.) Accordingly, we can estimate the number of iterations until the probability reaches $1/2$ as $\kappa \approx \sqrt{2\ln 2}\,\sqrt{N_0/\alpha\rho}$. It

Table 1. NIST PQC signature algorithms with example parameters selected for analysis.

Signature Algorithm / Parameters	Claimed Security	Signature Size (bytes)	Reference
CRYSTALS–Dilithium	Level 2	2420	[16]
FALCON-512	Level 1	666	[17]
SPHINCS$^+$-128s	Level 1	7856	[18]
HSS/LMS ($L = 2$, LMS_SHA256_M32_H10, LMOTS_SHA256_N32_W8)	n/a	2964	[19]
XMSS^MT (XMSSMT-SHA2_20/2_256)	n/a	4963	[20]

follows that we can estimate the number of condensed signatures until the verifier will need to request a new reference value as $K = \kappa\rho \approx \sqrt{2 \ln 2}\sqrt{N_0\rho/\alpha}$.

7.2 Impact on Example PQC Signature Algorithms

We now consider the reduction in signature overhead for five NIST PQC signature algorithms with example parameters given in Table 1. We selected the variants of CRYSTALS–Dilithium, FALCON and SPHINCS$^+$ with the shortest signatures (see Tables 8 and 9 in [1]). We selected the recommended parameterization of XMSS^MT with the shortest signatures (see Table 5 in [20]) and opted for a comparable parameterization of HSS/LMS.

For our analysis, we set $N_0 = 10{,}000$, so our ladders include up to 14 hash values and our authentication paths include up to 13. We selected a security parameter of 128 and hash size of 256 bits to match the underlying signature algorithms, and used the hash function instantiations in Appendix A. The sizes of the various MTL mode components in bytes can be computed as follows:

- *Initial signature* $\sigma = \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^S, d_i \rangle$ is $16 + 32 + 4 + 4 + 4 + 13 \cdot 32 + 14 \cdot 32 + 32 = 956$ plus the size of the underlying signature σ_i^S.
- *Condensed signature* $\varsigma = \langle SID, c_i, i, N, \Pi_{i,N} \rangle$ is $16 + 32 + 4 + 4 + 13 \cdot 32 = 472$.
- *Reference value* $\upsilon = \langle N', \Lambda_{N'}, \sigma_{N'}^S \rangle$ is $4 + 14 \cdot 32 = 452$ plus the size of $\sigma_{N'}^S$.

Here, we've ignored the overhead of the tag τ and the reference value handle χ, which may be relatively small, as well as protocol overheads such as identifying algorithms and public keys that would also be needed in the underlying signature scheme.

We define the effective signature size as

$$\phi(K, K') = |\varsigma| + \frac{K'}{K}|\upsilon|;$$

where K is the number of condensed signatures received, K' is the number of reference values received, $|\varsigma|$ is the size in bits of a condensed signature and $|\upsilon|$ is the size in bits

of a reference value. The effective signature size thus reflects the average number of bits that the signer sends per signature of interest. (We've assumed that $|\varsigma|$ and $|\upsilon|$ are constant and have ignored the presumably short handle χ and other protocol overheads.)

Fig. 3 shows the effective signature size $\phi(K, 1)$ as a function of K for the five examples. We've set $K' = 1$, given that only the initial reference value has been received up until this point. The effective signature size becomes smaller than the underlying signature size when $K = 6$ for FALCON and $K = 2$ for the other examples.

Fig. 4 shows the expected value of K as a function of ρ for three values of α (10, 100, 1000). Under nearly all of this range of operational assumptions, except when ρ is near its low end and α is at its high end, it is reasonable to expect that K will be large enough that the effective signature size will be less than the underlying signature size for all five examples. We expect the *ongoing* effective signature size to be even less than our estimate because the signature series is expanding, thus increasing K.

7.3 Further Considerations

The example just given illustrates one practical scenario and one mode of operation. Other modes may also be helpful in this and other scenarios. A few suggestions follow:

- *Multiple message series.* We can reduce the condensed signature size (and/or accommodate more messages) by arranging the messages into multiple series, each with a different tag; each individual series would potentially have a shorter maximum authentication path size than a combined series. Our analysis would need to account for the larger number of reference values that would have to be maintained.
- *Condensing underlying signatures.* It may also be helpful to consider constructions where part of the authentication path is included in the reference value or where the underlying signature within the reference value is itself conveyed via a condensable signature scheme. As discussed in Appendix D, the three hash-based signature schemes all support a modest amount of condensation with certain parameter sets.
- *Storage optimizations.* With MTL mode, a server that performs condensation operations potentially only needs to store for each message series, the current ladder, its underlying signature, the Merkle tree nodes and the per-message randomizers — an average of less than two nodes and one randomizer per message, plus the overhead of the ladder and the underlying signature. This can be a significant savings compared to storing an underlying signature for each message.
- *Batch signing and verification.* When multiple messages are signed during an iteration, it is possible to "batch" the signing and reduce the number of underlying signatures by signing just a single updated ladder that spans all the newly signed messages. The initial signatures produced for these messages would then be relative to this single ladder rather than per-message ladders. The verifier can also effectively batch verification if the underlying signatures are verified as reference values are received. MTL mode may therefore improve effective signing and verification performance compared to the underlying signature algorithm.

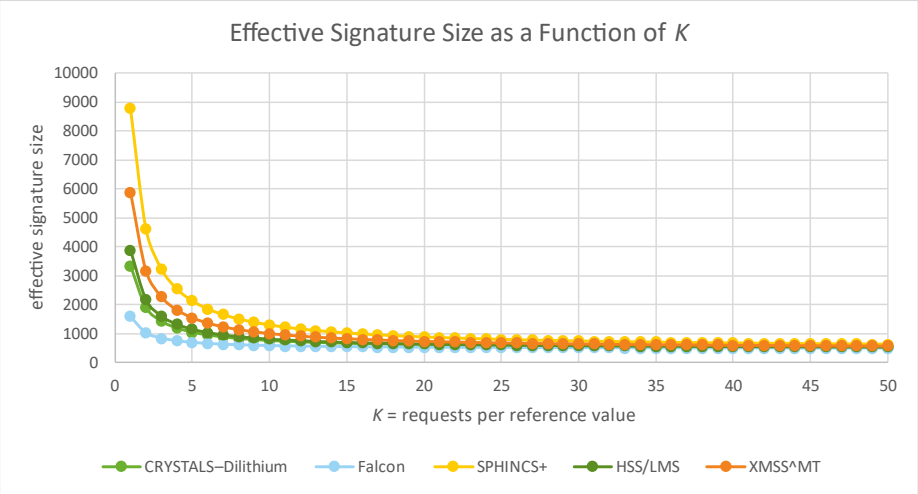

Fig. 3. Effective signature size in bytes for five post-quantum algorithms with example parameters as a function of number of condensed signatures per reference value. signature

- *Hybrid signature schemes.* MTL mode can help make hybrid signature schemes [21,22] more practical. In these schemes, the signer employs two or more signature schemes in parallel. If the underlying signature scheme itself is a hybrid scheme, then MTL mode can be applied to it directly. Alternatively, a variant MTL mode of operation could be defined in terms of multiple underlying signature schemes, where the evolving Merkle tree ladder is signed using each of the schemes. Either way, the additional signatures involved would only increase the size of the reference values, not the condensed signatures.
- *Caching condensed signatures* may require special processing because the reference values held in a cache may be *older* than the one held by the verifier and the reference value compatibility property may not necessarily apply. Appendix F provides guidance on how to handle caching of condensed signatures in practice.

8 Related Work

The binary rung strategy appears under different names in other cryptographic constructions based on Merkle trees. Champine defines a *binary numeral tree* [23] with similar structure (the successive complete binary subtrees are called *eigentrees*) and also specifies additional operations on the tree such as a proof that leaf nodes are consecutively ordered. Champine also references related constructions including Certificate Transparency [9]. The earlier constructions also include Crosby and Wallach's *history trees* [24] and Todd's *Merkle mountain ranges* [25]. Bünz et al. [26] provide a formal definition and analysis of the latter.

Cryptographic accumulators [27] have a similar structure to condensation and reconstitution in that a common *accumulator value* (viz, reference value) helps a

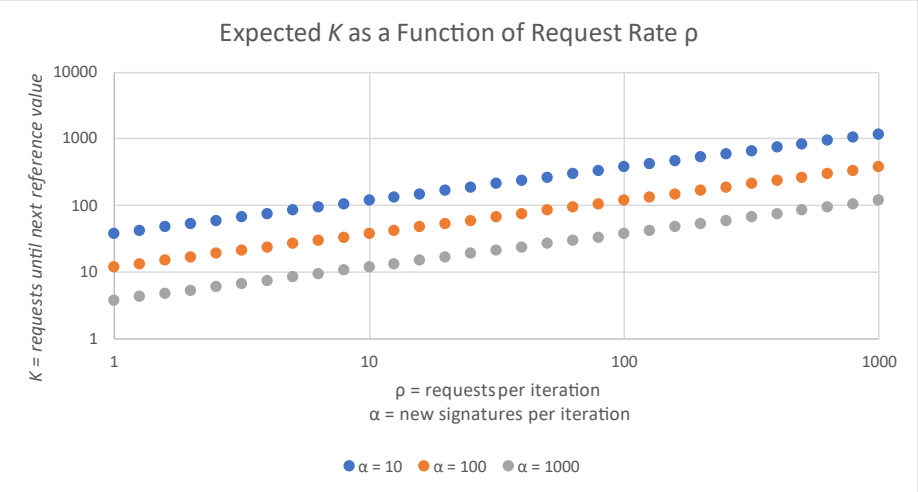

Fig. 4. Expected number of condensed signatures per reference value as a function of request rate and new signature rate.

verifier authenticate multiple elements, each of which has a *witness* relative to the accumulator value (viz, condensed signature). Reyzin and Yakoubov's accumulator [28], applying a binary rung strategy-like construction, also achieves an "old-accumulator compatibility" property comparable to general path verification for Merkle tree ladders.

Verkle trees, proposed by Kuszmaul [29] and further elaborated by Buterin [30] replace the hash function that authenticates pairs of subtrees in a conventional Merkle tree construction with a vector commitment scheme [31] that authenticates a large number of subtrees. With the proposed construction, the size of the authentication path can be significantly reduced. However, the construction is based on pre-quantum techniques. Peikert, Pepin and Sharp [32] propose a post-quantum vector commitment scheme, but the size of its authentication path is on the same order as a conventional Merkle tree. Buterin [30] suggests Scalable Transparent ARguments of Knowledge (STARKs) [33] as a future post-quantum alternative for Verkle trees.

Aggregate signatures convert multiple signatures into a shorter common value. In Boneh et al.'s original construction [34], a verifier can authenticate each signed message based only on the aggregate signature, provided that the verifier also has access to the other messages that were signed. Aggregate signatures can thus reduce the size impact of the signature scheme to which they're applied when the verifier has a large number of messages to verify. Khaburzaniya et al. show how to aggregate hash-based signatures using hash-based constructions [35]. Goyal and Vaikuntanathan [36] propose an improved scheme where the signatures can be made "locally verifiable" such that the verifier only needs access to specific messages of interest. However, their constructions are based on pre-quantum techniques (bilinear maps, RSA).

Merkle tree constructions are applied to the problem in of authenticating an evolving or "streaming" data series by Li et al. [37]. Papamanthou et al. propose an authenticated

data structure for a streaming data series [38] that uses lattice-based cryptography rather than traditional hash functions. The construction provides additional flexibility and efficiency, as well as another potential path toward post-quantum cryptography.

Stern et al. [14] define signature malleability in the limited sense we have adopted here. Chase et al. [39], building on work by Ahn et al. [40] and Attrapadung, Libert and Peters [41] broaden the definition to include the ability to produce a new signature on a message *related* in a specified way to a message that has already been signed. MTL mode only requires the narrower property. Decker and Wattenhofer [15] analyze claims that the bankruptcy of the MtGox exchange was a result of an attack involving signature malleability. They concluded that while signature malleability is a concern for the Bitcoin network, there is little evidence of such attacks prior to MtGox's bankruptcy.

Focusing on the Transport Layer Security protocol, Sikeridis, Kampanakis and Devetsikiotis anticipate that the TLS certificate chain and the server's signature in the TLS handshake would become the "bottleneck of [post-quantum] authentication" from a size and processing perspective [42]. Their observations further motivate TLS protocol extensions where the server omits any certificates that the client already has. Sikeridis et al. [43] propose an efficient signaling technique for determining which intermediate certificates to omit or "suppress." Suppression is complementary to condensation in that it reduces communication cost when the client already has a given certificate, whereas condensation helps when the client has a *different* certificate signed by the same certification authority.

Kudinov et al. [44] propose several techniques for reducing the size of SPHINCS+ signatures, including an example with 20% savings. Baldimtsi et al. [45] describe a general framework for reducing the size of cryptographic outputs using brute-force "mining" techniques, estimating 5– 12% savings. Such techniques are complementary to condensation as they reduce the size of the underlying signature whereas condensation reduces the need to send full underlying signatures at all.

9 Conclusion

We have shown that MTL mode can help reduce signature size impact in practical application scenarios. We suggest this mode, or another mode with similar properties, can be a standard way to use NIST PQC signature algorithms in message series-signing applications where signature size impact is a concern.

We plan to develop a more detailed, interoperable specification for MTL mode and its implementation choices (parameter values, functions H_{msg}, H_{leaf}, H_{int}, H_{SID}, signature and reference value formats, algorithm identifiers, etc.). We also intend to model the operational characteristics of MTL mode for various underlying signature schemes and operational assumptions.

In addition, we plan to consider how MTL mode can be integrated into applications such as those involving web PKI, DNSSEC and certificate transparency. As an initial approach, we imagine a "semi-indirect" format where a signer conveys a condensed signature ς together with information on how the verifier may resolve the associated handle χ into a reference value, such as a uniform resource identifier (URI) or a domain

name where the reference value is stored, or from which it may be obtained. (Some information about how to resolve a handle or access condensation scheme operations may also be conveyed in the representation of the public key and/or in the format for an uncondensed signature.)

NIST recently announced a call for additional signature candidates with shorter signature sizes and more cryptographic diversity than the current NIST PQC signature algorithms [46]. The call complements our suggestion of modes of operation. Indeed, even if a new algorithm with a much shorter signature size were introduced, MTL mode may still be helpful because it can be applied to any of the current algorithms, thereby maintaining diversity.

Modes of operation have historically provided a way to realize additional capabilities from an underlying cryptographic technique, such as a block cipher in the case of NIST's classic modes. We hope that modes of operation such as MTL mode can offer a way to achieve additional capabilities from post-quantum signature schemes as well.

References

1. Post-Quantum Cryptography Standardization, NIST, https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization, last accessed 2022/12/02.
2. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., et al.: NIST IR 8413-upd1: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. NIST (2022); includes updates as of 2022/09/26. https://doi.org/10.6028/NIST.IR.8413-upd1, last accessed 2022/12/02.
3. Cooper, D.A., D. Apon, Q.H. Dang, Davidson, M.S., Dworkin, M.J., Miller, C.A.: NIST Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes. NIST (2020). https://doi.org/10.6028/NIST.SP.800-208.
4. Announcing the Commercial National Security Algorithm Suite 2.0, National Security Agency, https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF, last accessed 2022/12/02.
5. Migration to Post-Quantum Cryptography. NIST National Cybersecurity Center of Excellence, https://www.nccoe.nist.gov/crypto-agility-considerations-migrating-post-quantum-cryptographic-algorithms, last accessed 2022/12/02.
6. Wouters, P., Sury, O: RFC 8624, Algorithm Implementation Requirements and Usage Guidance for DNSSEC. IETF (2019). https://doi.org/10.17487/RFC8624.
7. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: RFC 5280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. IETF (2008). https://doi.org/10.17487/RFC5280.
8. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. IETF (2005). https://doi.org/10.17487/RFC4033.
9. Laurie, B., Messeri, E., Stradling, R.: RFC 9162: Certificate Transparency Version 2.0. IETF (2021). https://doi.org/10.17487/RFC9162.
10. Merkle, R.: Secrecy, Authentication, and Public Key Systems. Ph.D. thesis, Stanford University (1979). http://www.ralphmerkle.com/papers/Thesis1979.pdf, last accessed 2002/12/02.
11. FIPS PUB 81: DES Modes of Operation. National Bureau of Standards, U.S. Department of Commerce (1980). https://doi.org/10.6028/NBS.FIPS.81.

12. Abe, M., Chase, M., David, B., Kohlweiss, M., Nishimaki, R., Ohkubo, M.: Constant-size structure-preserving signatures: Generic constructions and simple assumptions. Journal of Cryptology 29(4), 833–878 (2016). https://doi.org/10.1007/s00145-015-9211-7.
13. Yuan, Q., Tibouchi, M., Abe, M.: Security notions for stateful signature schemes. IET Information Security 16(1), 1–17 (2022). https://doi.org/10.1049/ise2.12040.
14. Stern, J., Pointcheval, D., Malone-Lee, J., Smart, N.P.: Flaws in applying proof methodologies to signature schemes. In: Yung, M. (ed.) Advances in Cryptology — CRYPTO 2002, LNCS, vol. 2442, pp. 93–110. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45708-9_7.
15. Decker, C., Wattenhofer, R.: Bitcoin transaction malleability and MtGox. In: Kutyłowski, M., Vaidya, J. (eds.) Computer Security — ESORICS 2014, LNCS, vol. 8173, pp. 313–326. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_18.
16. Bai, S., L. Ducas, E. Kiltz, Lepoint, T., Lyubashevsky, V., Schwabe, P. et al.: CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1), dated 2021/02/08, https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf, last accessed 2022/12/20.
17. Fouque, P.-A., J. Hoffstein, P. Kirchner, Lyubashevsky, V., Pornin, T., Prest, T., et al.: Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU Specification v1.2, dated 2020/01/10, https://falcon-sign.info/falcon.pdf, last accessed 2022/12/02.
18. Aumasson, J.-P., D.J. Bernstein, W. Beullens, Dobraunig, C., Eichlseder, M., Fluhrer, S., et al.: SPHINCS$^+$ Submission to the NIST post-quantum project, v.3.1, dated 2022/06/10, https://sphincs.org/data/sphincs+-r3.1-specification.pdf, last accessed 2022/12/02.
19. McGrew, D., Curcio, M., Fluhrer, S.: RFC 8554, Leighton-Micali Hash-Based Signatures (2019). https://doi.org/10.17487/RFC8554.
20. Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: RFC8391, XMSS: eXtended Merkle Signature Scheme. IETF (2018). https://doi.org/10.17487/RFC8391.
21. Barker, W., Polk, W., Souppaya, M.: Getting Ready for Post-Quantum Cryptography: Exploring Challenges Associated with Adopting and Using Post-Quantum Cryptographic Algorithms, NIST Cybersecurity White Paper, 2021/04/28. https://doi.org/10.6028/NIST.CSWP.04282021.
22. Driscoll, F.: Terminology for Post-Quantum Traditional Hybrid Schemes,. https://datatracker.ietf.org/doc/draft-driscoll-pqt-hybrid-terminology, last accessed 2022/12/02. Work in progress.
23. Champine, L.: Streaming Merkle Proofs within Binary Numeral Trees. In: Cryptology ePrint Archive, Paper 2021/038. https://eprint.iacr.org/2021/038, last accessed 2022/12/02.
24. Crosby, S., Wallach, D.: Efficient data structures for tamper-evident logging. In Proceedings of the 18th USENIX Security Symposium, pp. 317–334. USENIX Association (2009). https://dl.acm.org/doi/abs/10.5555/1855768.1855788.
25. Todd, P.: Merkle Mountain Ranges, https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, last accessed 2022/12/02.
26. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: FlyClient: Super-light clients for cryptocurrencies. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 928–946. IEEE (2020), https://doi.org/10.1109/SP40000.2020.00049.
27. Benaloh, J., de Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Helleseth, T. (ed.) Advances in Cryptology — EUROCRYPT '93, LNCS, vol. 765, pp. 274–285. Springer, Berlin, Heidelberg (1993). https://doi.org/10.1007/3-540-48285-7_24.
28. Reyzin, L., Yakoubov, S.: Efficient asynchronous accumulators for distributed PKI. In: Zikas, V., De Prisco, R. (eds) Security and Cryptography for Networks, SCN 2016, LNCS,

vol. 9841, pp. 292–309. Springer, Cham, 2016. https://doi.org/10.1007/978-3-319-44618-9_16.
29. Kuszmaul, J.: *Verkle Trees*, https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf, last accessed 2022/12/02.
30. Buterik, V.: Verkle Trees, https://vitalik.ca/general/2021/06/18/verkle.html, last accessed 2022/12/02.
31. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) Public-Key Cryptography — PKC 2013, LNCS, vol. 7778, pp. 55–72. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_5.
32. Peikert, C., Pepin, Z., Sharp, C.: Vector and functional commitments from lattices. In: Nissim, K., Waters, B. (eds.) Theory of Cryptography, TCC 2021, LNCS, vol. 13044, pp. 480–511. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90456-2_16.
33. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, Transparent, and Post-Quantum Secure Computational Integrity. In: Cryptology ePrint Archive, Paper 2018/046, https://eprint.iacr.org/2018/046, last accessed 2022/12/02.
34. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) Advances in Cryptology — EUROCRYPT 2003, LNCS, vol. 2656, pp. 416–432. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_26.
35. Khaburzaniya, I., Konstantinos, C., Lewi, K., Malvai, H.: Aggregating and thresholdizing hash-based signatures using STARKs. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, pp. 393–407. ACM, New York (2022). https://doi.org/10.1145/3488932.3524128.
36. Goyal, R., Vaikuntanathan, V.: Locally Verifiable Signature and Key Aggregation, In: Cryptology ePrint Archive, Paper 2022/179, https://eprint.iacr.org/2022/179, last accessed 2022/12/02. To appear, Advances in Cryptology — CRYPTO 2022.
37. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G.: Proof-infused streams: Enabling authentication of sliding window queries on streams. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 147–158. VLDB Endowment (2007). https://dl.acm.org/doi/10.5555/1325851.1325871.
38. Papamanthou, C., Shi, E., Tamassia, R., Yi, K.: Streaming authenticated data structures. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013, LNCS, vol. 7881, pp. 353–370. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_22.
39. Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Malleable signatures: New definitions and delegatable anonymous credentials. In 2014 IEEE 27th Computer Security Foundations Symposium, pp. 199–213. IEEE (2014). https://doi.org/10.1109/CSF.2014.22.
40. Ahn, J.H., Boneh, D., Camenisch, J., Hohenberger, S., Shelat, A., Waters, B.: Computing on authenticated data. *Journal of Cryptology* 28(2), 351–395. https://doi.org/10.1007/s00145-014-9182-0.
41. Attrapadung, N., Libert, B., Peters, T.: Computing on authenticated data: New privacy definitions and constructions. In: Wang, X., Sako, K. (eds) Advances in Cryptology — ASIACRYPT 2012, LNCS, vol. 7658, pp. 367–385. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_23.
42. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: a performance study. In: Network and Distributed Systems Security (NDSS) Symposium 2020, The Internet Society (2020). https://dx.doi.org/10.14722/ndss.2020.24203.

43. Sikeridis, D., Huntley, S., Ott, D., Devetsikiotis, M.: Intermediate Certificate Suppression in Post-Quantum TLS: An Approximate Membership Querying Approach, In: Cryptology ePrint Archive, Paper 2022/1556, https://eprint.iacr.org/2022/1556, last accessed 2022/12/02. To appear, 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22).
44. Kudinov, M., Hülsing, A., Ronen, E. Yogev, E., SPHINCS+C: Compressing SPHINCS+ With (Almost) No Cost, In: Cryptology ePrint Archive, Paper 2022/778, https://eprint.iacr.org/2022/778, last accessed 2022/12/02.
45. Baldimtsi, F., Chalkias, K., Chatzigiannis, P., Kelkar, M.: Truncator: Time-space Tradeoff of Cryptographic Primitives, In: Cryptology ePrint Archive, Paper 2022/1581, https://eprint.iacr.org/2022/1581, last accessed 2022/12/02.
46. Draft Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process, NIST, https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf, last accessed 2022/12/02.
47. FIPS PUB 180-4: Secure Hash Standard. NIST (2015). https://doi.org/10.6028/NIST.FIPS.180-4.
48. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, CM., Chung, KM., Persiano, G., Yang, BY. (eds) Public-Key Cryptography — PKC 2016, LNCS, vol. 9614, pp. 387–416. Springer, Berlin, Heidelberg, 2016. https://doi.org/10.1007/978-3-662-49384-7_15.
49. Sloane, N.J.A.: The ruler function: 2^a(n) divides 2n. Or, a(n) = 2-adic valuation of 2n. In: The On-Line Encyclopedia of Integer Sequences, Entry A001511, https://oeis.org/A001511, last accessed 2022/12/02.
50. Sloane, N.J.A., Wilks, A.: a(n) = a(floor(n/2)) + n; also denominators in expansion of 1/sqrt(1-x) are 2^a(n); also 2n - number of 1's in binary expansion of 2n. In: The On-Line Encyclopedia of Integer Sequences, Entry A005187, https://oeis.org/A005187, last accessed 2022/12/02.

Appendix A Hash Function Instantiations

MTL mode uses four hash functions as noted in Sections 2 and 5:

- $H_{\text{msg}}(SID, i, m, c) \longrightarrow d$ maps a series identifier SID, an index i, a message m and a randomizer c to a data value d;
- $H_{\text{leaf}}(SID, i, d) \longrightarrow V$ maps a series identifier SID, an index i and a data value d to a 2ℓ-bit leaf hash value V;
- $H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) \longrightarrow V$ maps a series identifier SID, two indexes L and R, and two 2ℓ-bit hash values V_{left} and V_{right} to a 2ℓ-bit hash value V; and
- $H_{\text{SID}}(S, \tau) \longrightarrow SID$ maps a 2ℓ-bit seed and a tag to a series identifier SID.

We want the functions to be cryptographically separate from one another and also from any hash functions involved in the underlying signature scheme $\mathcal{S}$. Because the example underlying signature schemes instantiate their own hash functions in different ways, we find it more practical to propose custom instantiations of the four hash functions for each scheme than to construct a generic set for use across all schemes. While we've adopted a concatenate-then-hash style for our instantiations, the flexibility gives the option to move to a different style, e.g., mask-then-hash, to align better with the security proofs for the underlying schemes. An implementation of MTL can use the same underlying hash function as the underlying signature scheme or a different hash function.

In the following, let H be a second-preimage-resistant hash function with a 2ℓ-bit output (e.g., SHA-256 [47] for the case $\ell = 128$). We assume the 2ℓ-bit hash function output is represented as an octet string as per the hash function's specification, e.g., Section 3 of [47]; $hLen = \ell/4$ denotes the length of the octet string. We also adopt the following notation: $[x]_w$ converts a non-negative integer x to its w-octet unsigned representation, most significant octet first; $x\langle 1{:}\, w\rangle$ returns the first w octets of an octet string x (we start our numbering with octet 1); and 0x denotes a hexadecimal representation.

The next sections propose instantiations for the five underlying post-quantum signature schemes mentioned in the paper. While Section 7 focuses on specific parameter sets for analysis, the instantiations are more general and could be applied to other parameter sets as well.

A.1 HSS/LMS Instantiations

HSS/LMS defines its hash functions by formatting their inputs into input strings to the underlying hash function H; the values of the 21^{st} and 22^{nd} octets provide separation between the different uses (see Section 9.1 of [19]). We take a similar approach for the MTL mode's uses and propose

$$H_{\text{msg}}(SID, i, m, c) := H(SID \parallel [i]_4 \parallel D_{\text{MTLM}} \parallel c \parallel m);$$
$$H_{\text{leaf}}(SID, i, d) := H(SID \parallel [i]_4 \parallel D_{\text{MTLL}} \parallel d);$$

$$H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) := H(SID \parallel [L]_4 \parallel D_{\text{MTLI}} \parallel [R]_4 \parallel V_{\text{left}} \parallel V_{\text{right}}); \text{ and}$$
$$H_{\text{SID}}(S, \tau) := H(\,[0]_{16} \parallel \tau \parallel D_{\text{MTLS}} \parallel S)\langle 1{:}\,16\rangle.$$

To align with HSS/LMS's formats, we place three constraints on our MTL mode implementation: SID must be a 16-octet string; i, L and R must be at most $2^{32} - 1$; and τ must be a four-octet string. We suggest $D_{\text{MTLM}} = 0\text{x}9090$, $D_{\text{MTLL}} = 0\text{x}9191$, $D_{\text{MTLI}} = 0\text{x}9292$ and $D_{\text{MTLS}} = 0\text{x}9393$, contrasting with HSS/LMS's identifiers which either start with $0x8$ or have most significant bit 0. In addition, so that the boundary between c and m is unambiguous, we require that the randomizer has a fixed length. We suggest 2ℓ bits, the same length as HSS/LMS's own message randomizer (see Section 7.1 of [19]). We put the seed S at the end of the input format for H_{SID} so that an implementation can choose a length larger than 16 octets, which would be the limit if we put it at the beginning. If S is 32 octets or shorter, then the input to the hash function in H_{SID} is at most 54 octets, which, after padding, fits within a single SHA-256 compression function call. H_{leaf} similarly takes a single call and H_{int} takes two, matching their counterparts in HSS/LMS.

With these instantiations, up to $2^{32} - 1$ messages can be associated with a given tag and up to $2^{64} - 2^{32}$ messages can be signed in MTL mode with a given HSS/LMS key pair. The latter limit is greater than the total number of messages supported by any of the recommended HSS/LMS parameter sets (i.e., 2^{40}; see Section 6.4 of [19]).

A.2 Instantiations for Other Underlying Signature Schemes

We now provide some suggestions on how one might instantiate the four hash functions when MTL mode is applied to the other underlying schemes.

XMSS^MT. Like HSS/LMS, XMSS^MT separates its hash functions by distinguishing certain octets in the inputs to H; here, the first *hLen* octets vary (see Section 5.1 of [20]). Following this approach, we propose

$$H_{\text{msg}}(SID, i, m, c) := H(D_{\text{MTLM}} \parallel SID \parallel [i]_4 \parallel c \parallel m);$$
$$H_{\text{leaf}}(SID, i, d) := H(D_{\text{MTLL}} \parallel SID \parallel [i]_4 \parallel d);$$
$$H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) := H(\,D_{\text{MTLI}} \parallel SID \parallel [L]_4 \parallel [R]_4 \parallel V_{\text{left}} \parallel V_{\text{right}}); \text{ and}$$
$$H_{\text{SID}}(S, \tau) := H(D_{\text{MTLS}} \parallel S \parallel \tau \parallel)\langle 1{:}\,16\rangle.$$

We suggest $D_{\text{MTLM}} = [256]_{hLen}$, $D_{\text{MTLL}} = [257]_{hLen}$, $D_{\text{MTLI}} = [258]_{hLen}$ and $D_{\text{MTLS}} = [259]_{hLen}$, contrasting with XMSS^MT's identifiers which involve integers in the range 0–3. For consistency with our HSS/LMS instantiations and make to the formats unambiguous, we again constrain SID to be a 16-octet string; i, L and R to be at most $2^{32} - 1$; and τ to be a four-octet string. However, the instantiations could be redefined with different lengths. As above, with these constraints up to $2^{64} - 2^{32}$ messages can be signed in MTL mode with a given XMSS^MT key pair, a limit that again is greater than the total number of messages supported by any of the recommended parameter sets (i.e., 2^{60}; see Section 5.4.1 of [20]).

Note that we've maintained the concatenate-then-hash style of our HSS/LMS instantiations. We could instead follow XMSS^MT's mask-then-hash style where bitmasks are derived from "address" components such as SID, L and R and exclusive-

ored with other inputs. We could also adjust the formatting to align with the boundaries of the hash function's internal compression function, as XMSS^MT does.

(Notational comment: the H_{msg} we define here is the MTL mode function, not XMSS^MT's `H_msg`; and the H we use here is the underlying hash function, e.g., SHA-256, not XMSS^MT's `H`.)

SPHINCS$^+$. A 32-octet address field separates different uses of the underlying hash function for this scheme (see Sections 7.2 and 2.7.3 of [18]). The first four octets are the *layer address*. SPHINCS$^+$'s own layer addresses are in the range 0–6, so we again suggest the range 256–259 for the MTL mode functions. The other inputs would be formatted to align with SPHINCS$^+$'s formats (e.g., padding the first field to the length of the compression function). We could also adopt a mask-then-hash style in addition to the concatenate-then-hash style as SPHINCS$^+$ does in its "robust" variant. The instantiations could impose the same constraints as the other instantiations above, or they could move to larger sizes (e.g., eight-octet indexes and tag), given that SPHINCS$^+$ is stateless and therefore doesn't have a built-in limit on the number of messages that can be signed.

FALCON's only internal use of a hash function is for mapping a 320-bit salt and a message to a polynomial; the scheme uses SHAKE-256, where the input is the concatenation of the salt and the message (see Section 3.9.1 of [17]). FALCON's own instantiation thus doesn't directly provide a way to separate other uses of the underlying hash function, and it doesn't support SHA-256 (it requires an extendable-output function (XOF)). Given that we don't have an opportunity to separate from MTL mode's uses from FALCON's, any of the instantiations for the other schemes seems an equally reasonable choice.

CRYSTALS–Dilithium uses a hash function (SHAKE-256 or SHAKE-128) for several purposes (see Section 5.3 of [16]). Like FALCON, it doesn't directly provide a way to separate other uses from its own, nor does it support SHA-256. Again, any of the previous instantiations would seem to be equally reasonable.

A.3 Hash Function Usage Outside Signature Schemes

As evidenced above, two of the five example post-quantum signature schemes considered don't provide a direct way to separate their uses of an underlying hash function from other uses outside the signature scheme.

Given that a signature scheme will often be combined with other uses of the same hash function in an application, it would be worthwhile to have a common convention for using a hash function within a signature scheme that does provide for such separation. The convention would be another aspect of the ongoing improvements in multi-user / multi-target security [48], where a design goal is to limit each of the adversary's hash function queries to a specific context.

Appendix B Binary Rung Strategy Operations

We now give example pseudocode for the Merkle tree ladder operations in the binary rung strategy $\mathcal{BRS}$ described in Section 2.6, which is the basis for the MTL mode operations in Section 5. The pseudocode takes an iterative approach where the authentication paths and ladders are constructed with "for" loops, following the tree structure from leaf to ladder. An alternative would be a recursive approach where the components are constructed with recursive calls that proceed from ladder to leaf. Arrays are indexed starting with 1.

Signature-generation-only optimizations. MTL mode's signature generation operation calls ADDLEAFNODE to add a leaf node corresponding to a message being signed and to obtain a ladder spanning the leaf nodes added so far. The operation then calls GETAUTHPATH to obtain an authentication path from the newly added leaf node to the newly produced ladder. The mode's condensation operations (Section 6) likewise call ADDLEAFNODE to add a leaf node corresponding to the signature being incorporated, but in contrast, call GETAUTHPATH to obtain an authentication path from an *arbitrary* leaf node to the current ladder. An implementation such as a hardware security module that is intended only to support signature generation, not condensation, therefore only needs to maintain enough hash values to produce the next ladder and the authentication path to it from the newly added leaf node. The pseudocode below covers the general case and requires storage for $O(N)$ hash values, where N is the number of leaf nodes in the node set. The notes suggest optimizations for the signature-generation-only case and require storage for only $O(\log N)$ hash values.

B.1 Node Set Representation

A node set T includes three parts: the series identifier, denoted $T.SID$; the number of leaf nodes, denoted $T.N$; and zero or more node hash values, each denoted $T.V[L{:}R]$ where L and R is the pair of indexes that uniquely identifies the node.

We assume a suitable data structure for mapping the index pair to the hash value. For instance, an implementation could maintain a lookup table $(L,R) \rightarrow V$, which would effectively serve as a sparse representation for an expanding $T.N \times T.N$ array. Alternatively, an implementation could map the index pair to a single index Z corresponding to the order in which the value $T.V[L{:}R]$ is computed by ADDLEAFNODE, and keep the value at this index in a one-dimensional array.

For the binary rung strategy, the $(L,R) \rightarrow Z$ mapping could take the form

$$Z := 2R - B(R) - \nu_B(R) + k(L,R).$$

where $B(R)$ is the number of ones bits in the binary representation of R, $\nu_B(R)$ is the index of the lowest ones bit in the representation (where bits are indexed starting at 0), and $k(L,R)$ is the unique integer such that $L = R - 2^k + 1$ (if such an integer exists; otherwise (L,R) is not a valid index pair for the binary rung strategy).

To see this, consider that the hash values added during the R^{th} call to ADDLEAFNODE, i.e., when $i = R$ are those with right index R and left index $L = R - 2^k + 1$ for each

value of k between 0 and $\nu_B(R)$; they're added in increasing order of k. To determine the one-dimensional index Z, then, we only need to know how many hash values are added up to and including the R^{th} call.

The index of the lowest ones bit in an the binary representation of an integer x, plus one, i.e., $\nu_B(x) + 1$, is the *ruler function* of the integer x [49]. The total number of hash values added up to and including the R^{th} call, i.e., $\sum_{x=1}^{R}(\nu_B(x) + 1)$, thus equals the sequential sum of the ruler function up to R, which is $2R - B(R)$ [50]. It follows that the overall order in which the hash value $T.V[L{:}R]$ is added is $2R - B(R) - \nu_B(R) + k(L, R)$. This order can then be used as an index to a one-dimensional array for storing and retrieving $T.V[L{:}R]$.

Another way to see the result is to consider that the ladder after the R^{th} call will include $B(R)$ complete, adjacent binary trees spanning the R leaf nodes. A complete binary tree has one fewer internal nodes than leaf nodes, so collectively, the $B(R)$ trees have $R - B(R)$ internal nodes and $2R - B(R)$ total nodes. These are the only nodes whose hash values will have been added up to and including this call. A node set with N leaf nodes can therefore be represented with $2N - B(N) \leq 2N - 1$ hash values.

Signature-generation-only case. If GETAUTHPATH and ADDLEAFNODE will be used only for signature generation, then the node set representation only needs to maintain enough to produce the next ladder and authentication path. In this case, the node set has four parts: $T.SID$; $T.N$; the current ladder, denoted $T.\Lambda$; and the current authentication path, denoted $T.\Pi$. In this case, the node set representation would include $B(N) + \nu_B(N)$ hash values (corresponding to the number of hash values in the ladder and in the authentication path); the sum is at most the number of bits in the binary representation of N. It follows that the storage requirement in the signature-only case is at most $\lfloor log_2 N \rfloor + 1$ hash values.

B.2 Node Set Initialization

$\text{INITNODESET}_{\mathcal{BRS}}(SID) \longrightarrow T_0$ returns a new node set T_0 associated with the series identifier SID.

1. Create a new, empty node set T_0.
2. Set $T_0.SID := SID$ and $T_0.N := 0$. The initial node set will include no node hash values.
3. Return T_0.

Signature-generation-only case. Step 2 also sets $T.\Lambda$ and $T.\Pi$ to empty arrays.

B.3 Leaf Node Count

$\text{GETLEAFNODECOUNT}_{\mathcal{BRS}}(T) \longrightarrow N$ returns the number of leaf nodes in the node set T.

1. Set $N := T.N$.
2. Return N.

B.4 Leaf Node Addition

ADDLEAFNODE$_{\mathcal{BRS}}(T, d) \longrightarrow \langle T', \Lambda \rangle$ adds a leaf node corresponding to the data value d to the node set T, assigning the next leaf node index to it, and returns the updated node set T' and the next ladder Λ spanning the leaf nodes in the node set, following the binary rung strategy.

1. Set $SID := T.SID$.
2. Set $i := T.N + 1$.
3. Set $T' := T$.
4. Set $T'.N := i$.
5. Compute $V := H_{\text{leaf}}(SID, i, d)$.
6. Set $T'.V[i{:}\,i] := V$, adding the new leaf node to the node set.
7. Write $i = \sum_{j=1}^{B} 2^{v_j}$ where the v_j are the indexes of the ones bits in the binary representation of i from highest to lowest.
8. For k from 1 to v_B do:
   a. Compute $V := H_{\text{int}}(SID, i - 2^k + 1, i, T'.V[i - 2^k + 1{:}\,i - 2^{k-1}], V)$.
   b. Set $T'.V[i - 2^k + 1{:}\,i] := V$, adding the new internal node to the node set.
9. Create a new empty array Λ.
10. Set $R := 0$.
11. For j from 1 to B do:
    a. Set $L := R + 1$ and $R := R + 2^{v_j}$.
    b. Set $\Lambda[j] := T'.V[L{:}\,R]$, adding this rung hash value to the ladder.
12. Return T' and Λ, which will be an array of B hash values.

Step 8 computes the new ladder rung $[i - 2^{v_B} + 1{:}\,i]$ from leaf to ladder. As also noted in Appendix C, Step 8 computes this rung from the last v_B rungs of Λ_{N-1}, so the rung is their ancestor. Step 11 then assembles the rungs into the ladder.

Signature-generation-only case. Instead of retrieving $T'.V[i - 2^k + 1{:}\,i - 2^{k-1}]$ from the set of node hash values during the call to H_{int}, Step 8a selects the rung $T.\Lambda[B + v_B - k]$ from the input node set. Instead of storing the new hash value, Step 8b copies the selected rung to the authentication path in the new node set by setting $T'.\Pi[k] := T.\Lambda[B + v_B - k]$. Similarly, instead of retrieving $T'.V[L{:}\,R]$ from the set of node hash values, Step 11b sets $\Lambda[j] := T.\Lambda[j]$ (if $j < B$) or $\Lambda[j] := V$ (if $j = B$). The step then copies the selected rung to the ladder in the new node set by setting $T'.\Lambda[j] := \Lambda[j]$. Step 11a is omitted.

Updating in place. ADDLEAFNODE can be changed to update the node set in place simply by removing the $T' := T$ step from the pseudocode and operating on the input T directly thereafter (and not returning T').

B.5 Authentication Path Construction

GETAUTHPATH$_{\mathcal{BRS}}(T, i) \longrightarrow \Pi$ returns the authentication path Π from the i^{th} leaf node in the node set T to its associated rung in the current ladder following the binary rung strategy.

1. Set $SID := T.SID$.
2. Set $N := T.N$.
3. If $i < 1$ or $i > N$ then return "index out of range."
4. Write $N = \sum_{j=1}^{B} 2^{\nu_j}$ where the ν_j are the indexes of the ones bits in the binary representation of N from highest to lowest.
5. Set $R := 0$.
6. For j from 1 to B do:
   a. Set $L := R + 1$ and $R := R + 2^{\nu_j}$.
   b. If $i \leq R$ then break.
7. Set $\Delta := i - L$.
8. Write $\Delta = \sum_{k=1}^{\nu_j} \delta_k 2^{k-1}$ where the δ_k are the bits of the binary representation of Δ from lowest to highest.
9. Create a new empty array Π.
10. For k from 1 to ν_j do:
    a. If $\delta_k = 0$ then set $\Pi[k] := T.V[L + \Delta + 2^{k-1} : L + \Delta + 2^k - 1]$.
    b. Else ($\delta_k = 1$) set $\Pi[k] := T.V[L + \Delta - 2^{k-1} : L + \Delta - 1]$ and $\Delta := \Delta - 2^{k-1}$.
11. Return Π, which will be an array of ν_j hash values.

Step 6 determines which rung of the ladder spans the leaf node, and Step 10 then constructs the authentication path from leaf to ladder, based on the binary representation of Δ, the relative position of the leaf node within the ladder rung span. (Recall that each rung spans a complete binary tree.)

Signature-generation-only case: Step 3 instead checks if $i \neq N$, given that this case assumes that the authentication path is from the newly added leaf node to the newly produced ladder only. Steps 5–10 are replaced by a loop that copies $T.\Pi[k]$ to $\Pi[k]$ for k from 1 to ν_B. (In Step 11, we have $\nu_j = \nu_B$.)

B.6 Authentication Path Verification

$\text{CHECKAUTHPATH}_{\mathcal{BRS}}(SID, i, N, N', d, \Pi, \Lambda) \longrightarrow b$ verifies that the i^{th} leaf node of a node set corresponds to the data value d using an authentication path Π and a ladder Λ following the binary rung strategy.

1. If $i < 1$ or $i > N'$ or $N' > N$ then return "index out of range."
2. Write $N' = \sum_{j=1}^{B} 2^{\nu_j}$ where $\nu_1, \ldots, \nu_B$ are the indexes of the ones bits in the binary representation of N' from highest to lowest.
3. If Λ is an array of fewer than B hash values then return "ladder too short."
4. Set $R := 0$.
5. For j from 1 to B do:
   a. Set $L := R + 1$ and $R := R + 2^{\nu_j}$.
   b. If $i \leq R$ then break.
6. If Π is an array of fewer than ν_j hash values then return "authentication path too short."
7. Set $\Delta := i - L$.

8. Write $\Delta = \sum_{k=1}^{\nu_j} \delta_k 2^{k-1}$ where the δ_k are the bits of the binary representation of Δ from lowest to highest.
9. Compute $V := H_{\text{leaf}}(SID, i, d)$.
10. For k from 1 to ν_j do:
    a. If $\delta_k = 0$ then compute $V := H_{\text{int}}(SID, L + \Delta, L + \Delta + 2^k - 1, V, \Pi[k])$.
    b. Else $(\delta_k = 1)$ compute $V := H_{\text{int}}(SID, L + \Delta - 2^{k-1}, L + \Delta + 2^{k-1} - 1, \Pi[k], V)$ and set $\Delta := \Delta - 2^{k-1}$.
11. If $V == \Lambda[j]$ then return TRUE else return FALSE.

Step 5, similar to the previous operation, selects the rung of the ladder to match. The rung may be reached by just a portion of the authentication path, given that the operation allows N' and N to be different. Step 10 then evaluates the authentication path from leaf to ladder based on the binary representation of Δ, similar to the previous operation.

Appendix C Proof of General Path Verification for Binary Rung Strategy

Claim. For all positive integers i, N, N' where $i \leq N' \leq N$, if d_i is the data value corresponding to the i^{th} leaf node in a node set assembled using the binary rung strategy, $\Pi_{i,N}$ is the authentication path from the i^{th} leaf node to its associated rung in the N^{th} ladder and $\Lambda_{N'}$ is the N'^{th} ladder, then

$$\text{CHECKAUTHPATH}_{\mathcal{BRS}}\left(SID, i, N, N', d_i, \Pi_{i,N}, \Lambda_{N'}\right) = \text{TRUE}.$$

Proof. If $N = 1$ then the result is trivial. Suppose $N > 1$ and consider the binary representation of $N - 1$. We write

$$N - 1 = \sum_{j=1}^{B} 2^{\nu_j} - 1 = \sum_{j=1}^{B-1} 2^{\nu_j} + (2^{\nu_B} - 1) = \sum_{j=1}^{B-1} 2^{\nu_j} + \sum_{k=1}^{\nu_B} 2^{k-1}.$$

The first $B - 1$ ones bits of N are the same as the first $B - 1$ ones bits of $N - 1$, while the last ones bit of N is replaced by ν_B consecutive lower-order ones bits of $N - 1$. The first $B - 1$ rungs in Λ_N are thus the same as the corresponding rungs in Λ_{N-1} and the last rung in Λ_N is an ancestor of each of the last ν_B rungs in Λ_{N-1} (compare Step 10 in Appendix B. 5 where $\nu_j = \nu_B$). Each of the rungs in Λ_{N-1} is therefore either the same as, or a descendant of, one of the rungs in Λ_N. By induction, the same holds for each of the rungs in $\Lambda_{N'}$ where $1 \leq N' \leq N$.

The evaluation of the authentication path from the i^{th} leaf node to its associated rung in ladder Λ_N recomputes the rung as well as every descendant of the rung whose span includes i. Because the rungs in each ladder have non-overlapping sets of descendants, it follows that the rung in $\Lambda_{N'}$ that spans the i^{th} leaf node is either the same as or a descendant of the rung in $\Lambda_N$that spans the i^{th} leaf node. $\text{CHECKAUTHPATH}_{\mathcal{BRS}}$can therefore verify $\Pi_{i,N}$ using $\Lambda_{N'}$. ■

Appendix D Condensing and Reconstituting Hash-Based Signatures

The three hash-based signature schemes among the NIST PQC signature algorithms, with certain parameter sets, all support a modest amount of condensation. (To fit our framework, we re-cast each scheme as a tagged signature scheme that allows only a null tag.)

We consider again the three example parameterizations given in Section 7:

- SPHINCS$^+$-128s
- HSS/LMS with parameters $L = 2$, LMS_SHA256_M32_H10 and LMOTS_SHA256_N32_W8
- XMSS^MT with parameter XMSSMT-SHA2_20/2_256

All three schemes involve multiple layers of Merkle trees; their signatures include multiple sets of one-time signatures and authentication paths. Condensation can be achieved by treating the one-time signature and authentication path for the top-layer tree as a reference value and the rest of each signature as a condensed signature. The SPHINCS$^+$ example has seven layers of trees, so its condensed signature size will be roughly 86% of its initial (i.e., uncondensed) signature size. The HSS/LMS and XMSS^MT examples have two layers; their condensation ratio will be roughly 50%.

The handle returned by GETCONDENSEDSIG resolves to the top part of the signature, which is common to all signatures involving a particular leaf of the top-layer tree. A verifier only needs to obtain a new reference value when a new top-level leaf is encountered.

In the HSS/LMS and XMSS^MT examples, the top-layer tree has $2^{10} = 1024$ leaf nodes. As a result, the number of reference values needed is at most 1024 regardless of K, leading to an upper bound on the effective signature size of

$$\phi(K, K') \leq |\varsigma| + \frac{1024}{K}|\upsilon|.$$

If K is more than about $1024 \times 2 = 2048$ for two-layer HSS/LMS or XMSS^MT, the effective signature size will be lower than the initial signature size and thereafter will continue to decrease, converging to the 50% ratio above. In the SPHINCS$^+$ example, the top-layer tree has $2^9 = 512$ leaf nodes and the transition point is around $512 \times 7 = 3584$.

The actual transition points for all three examples will be lower in practice because not every top-level leaf will necessarily be involved in the first K signatures, especially for HSS/LMS and XMSS^MT which exhaust each top-level leaf node before moving to the next one. If we want to reduce further, faster, and for non-hash-based signatures schemes, however, we need a different approach such as MTL mode.

Appendix E Stateful Reconstitution Operations

Just as the condensation operations are stateful, we could similarly restructure RECONSTSIG so that it maintains state between operations, e.g., with operations such as the following:

- $\text{RECONSTINIT}_{pk}() \longrightarrow st_0$ returns a new scheme state st_0.
- $\text{ADDCONDENSEDSIG}_{pk}(\tau, \varsigma, \chi, st) \longrightarrow \langle i, b, st' \rangle$ incorporates a condensed signature ς associated with the tag τ and the handle χ into the state st and returns the signature index i for this signature (relative to τ), a flag b indicating whether a new reference value is needed and the updated state st'.
- $\text{ADDREFVAL}_{pk}(\tau, \chi, \upsilon, st) \longrightarrow st'$ incorporates a reference value υ associated with the tag τ and a handle χ into the state st and returns the updated state st'.
- $\text{GETCONDENSEDSIG}_{pk}(\tau, i, st) \longrightarrow \langle \sigma', st' \rangle$ produces a reconstituted version σ' of the i^{th} signature associated with the tag τ under the state st and returns σ' and the updated state st'.

By returning the flag, ADDCONDENSEDSIG automates the reference value compatibility checks mentioned above. If the flag is TRUE, then the verifier requests a new reference value and incorporates it with ADDREFVAL. Otherwise, the verifier proceeds directly to GETRECONSTSIG. With the stateless version of RECONSTSIG, the application would need to do the reference value compatibility check itself. While this is straightforward in a mode based on the binary rung strategy (just compare $i \leq N' \leq N$), the check may be more complex in general (e.g., when directly condensing and reconstituting hash-based signatures as proposed in Appendix D).

Appendix F Caching Condensed Signatures

In Section 7, we assumed that each condensed signature ς received by the verifier was produced relative to a reference value υ_N that was newer than the reference value υ_{N_0} held by the verifier, i.e., $N_0 \leq N$. The assumption was the basis for our use of the reference value compatibility property of the binary rung strategy (Section 4.3). It enabled the verifier to reconstitute a signature provided that the message index i satisfied $i \leq N_0$. As a result, the verifier only needed to request a new reference value when $i > N_0$.

Our assumption may be realistic when the verifier interacts directly with a signer or intermediary that performs condensation operations. However, it may not be realistic when the verifier interacts with a responder that merely holds condensed signatures obtained from other parties. Indeed, a condensed signature held by such a responder will be associated with a reference value υ_N that was available when the responder itself obtained the condensed signature. That reference value may be *older* than the one held by the verifier, i.e., we may have $N < N_0$. If so, the reference value compatibility property won't necessarily apply and special processing may be required, potentially increasing the effective signature size and diminishing the benefit of MTL mode.

We refer to a responder that holds but does not produce condensed signatures as a *condensed signature caching server*. Two examples of such a responder include:

- A *recursive DNS server* that requests and holds signed resource record sets (RRsets) on behalf of its clients. The condensed signatures on the resource record sets would previously have been produced by an authoritative DNS server or its provisioning system (or by an intermediary that performs condensation operations). A DNS RRset has a *time to live (TTL)* value indicating how long the RRset should be held before requesting a new version from the authoritative name server. The reference value associated with a condensed signature returned by the recursive DNS server will thus generally be at most as old as the authoritative name server's maximum TTL, e.g., on the order of a day. The validity period for the signature can be much longer, on the order of weeks or months. The new signed version of the RRset can thus include a new condensed version of the same initial signature of the RRset (if the RRset hasn't changed).
- A *web server* that holds certificates for the websites it serves and provides these certificates to its clients. Here, the condensed signatures on the certificates would previously have been produced by a certification authority (or, again, by an intermediary). A web PKI certificate doesn't have an independent TTL, however; the certificate is simply held until the end of its validity period. Thus, the reference value associated with a condensed signature returned by a web server could be as old as the certificate itself, e.g., on the order of a year.

Given the importance of caching for application performance, it's worth considering how to mitigate the effect of caching on effective signature size for clients of these servers, e.g., for an browser or other application that validates a condensed signature on a resource record set or certificate. For this purpose we need to look more closely at how a verifier processes condensed signatures.

F.1 Processing Condensed Signatures with Caching

As a starting point, let's review a typical approach by which a verifier may process a condensed signature in MTL mode, taking caching into account.

Expanding on Section 7, we assume that the verifier already holds a set of reference values $\upsilon_{N_0^{(1)}}, \upsilon_{N_0^{(2)}}, \ldots$, where the reference value $\upsilon_{N_0^{(b)}}$ includes the $N_0^{(b)}$th Merkle tree ladder and an underlying signature on the ladder. The processing may involve the following steps. (We omit the public key pk and the tag τ for simplicity.)

1. The verifier obtains a condensed signature ς on a message m_i with index i, and a reference value handle χ. The condensed signature ς includes an authentication path $\Pi_{i,N}$ relative to the N^{th} ladder where $1 \leq i \leq N$; the reference value handle $\chi = N$.
2. If there exists a b such that $i \leq N_0^{(b)} \leq N$, then the verifier reconstitutes a signature from ς and $\upsilon_{N_0^{(b)}}$. Note that there can be more than b for which this condition holds.

3. If there doesn't exist a b such that $i \leq N_0^{(b)}$, then the verifier requests the reference value υ_N and reconstitutes a signature from ς and υ_N. The verifier then adds the reference value υ_N to its set of reference values.
4. If there exists a b such that (a) $N < N_0^{(b)}$ and (b) i, N and $N_0^{(b)}$ are compatible (in the sense defined below), then the verifier reconstitutes a signature from ς and $\upsilon_{N_0^{(b)}}$.
5. If there doesn't exist a b such that (a) and (b) in Step 4 hold (the only remaining possibility), then the verifier performs special processing as discussed below.

Note that if $N_0^{(b)} \leq N$ for all b (as we effectively assumed in Section 7), then only Steps 1–3 are needed. Steps 4 and 5 occur as a result of the $N < N_0$ case associated with caching condensed signatures.

We say that i, N and N' are *compatible* in a rung strategy if the authentication path from the i^{th} leaf node to its associated rung in the N^{th} ladder can be verified using the N'^{th} ladder. Appendix C shows that i, N and N' are compatible if $i \leq N' \leq N$; this is the basis for reconstitution in Step 2 above. We can also show that i, N and N' are compatible if $i \leq N < N'$ and $N' < R + 2^v$ where $R = 2^v$ is the (unique) integer between i and N that is divisible by the largest power of 2; this is the basis for Step 4. (To see, this consider that R is then also the unique integer with this property between i and N' and that $[R - 2^v + 1 : R]$ is then the rung associated with both $\Pi_{i,N}$ and $\Pi_{i,N'}$ — so the authentication paths are the same.) Special processing is therefore required when $N' \geq R + 2^v$.

The effective signature size for conveying a signature following Steps 1–5 includes the condensed signature size from Step 1, the overhead of the occasional reference value in Step 3, and the overhead of the occasional special processing in Step 5. We can mitigate the effect of caching on effective signature size by reducing the impact of special processing and/or the likelihood that special processing is performed. We may also be able to mitigate the effect of caching by changing to a different rung strategy. The next three subsections go into further detail on each of these mitigations.

F.2 Reducing Impact of Special Processing

A straightforward way to implement the special processing in Step 5 is for the verifier to request the reference value υ_N and then reconstitute a signature from ς and υ_N. However, this approach would involve the overhead of sending a full underlying signature (and a ladder) every time special processing is performed.

A more efficient approach is for the verifier instead to request the *current* version ς' of the condensed signature on m_i, and then reconstitute a signature from ς' and $\upsilon_{N_0^{(b)}}$, where $\upsilon_{N_0^{(b)}}$ is any one of its reference values. The signer or another intermediary could fulfill requests for the current version of the condensed signature by providing an externally accessible interface to GETCONDENSEDSIG in the same way that it fulfills requests for reference values via an interface to GETREFVAL. This approach would involve only the overhead of sending a condensed signature.

An even more efficient approach is for the verifier to request the *difference* between ς and the current version of the condensed signature. We suggest the following additional condensation scheme operations for this purpose:

- $\text{GETEXTVAL}_{pk}(\tau, i, \chi, st) \longrightarrow \langle \beta, st' \rangle$ produces an extension value β that can be used to transform a condensed signature associated with the tag τ and the handle χ to a condensed signature relative to the current reference value. It returns β and the updated state st'.
- $\text{EXTENDCONDENSEDSIG}_{pk}(\tau, \chi, \varsigma, \beta) \longrightarrow \varsigma'$ transforms a condensed signature ς associated with the tag τ and the handle χ into a condensed signature ς′ relative to τ and the reference value associated with the extension value β, and returns ς′.

In this approach, the verifier would request the extension value β for τ, i and χ, then call EXTENDCONDENSEDSIG to obtain a condensed signature ς′ relative to the current reference value. The verifier could then reconstitute a signature from ς′ and any of its reference values. As above, the signer or another intermediary would provide an external interface to GETEXTVAL. This approach would involve only the overhead of the extension value, which in MTL mode would include the missing sibling nodes in the authentication path. The combined overhead of ς and β would thus be comparable to the current condensed signature ς′.

F.3 Reducing Likelihood of Special Processing

Intuitively, the reason that special processing may be required is that a condensed signature received from a caching server is "too short" relative to the verifier's reference values — it's missing one or more sibling nodes. Therefore, a natural way to reduce the need for special processing is to refresh each condensed signature periodically to add the missing sibling hash nodes. Following the discussion above, assume that a condensed signature for index i is first added to the cache when N is the current number of leaf nodes, and that the condensed signature's authentication path is associated with the rung $[R - 2^v + 1 : R]$ in the current reference value's ladder. A new sibling node will then need to be added whenever the number of leaf nodes reaches a multiple of a larger power of 2, i.e., at $R + \gamma_1$, $R + \gamma_2$, $R + \gamma_3$, etc., where

$$\gamma_j = \begin{cases} \gamma_0 + 2^{\nu_{B-j+1}} \text{ if } 1 \le j \le B; \\ 2^{\lfloor \log_2 R \rfloor + j - B} \text{ if } j > B, \end{cases}$$

with $\gamma_0 = 0$ and where $\nu_1, \ldots, \nu_B$ are the indexes of the ones bits in the binary representation of $2^{\lfloor \log_2 R \rfloor + 1} - R$ from highest to lowest:

$$2^{\lfloor \log_2 R \rfloor + 1} - R = \sum_{j=1}^{B} 2^{\nu_j} .$$

Because 2^v is the largest power of 2 dividing R, we have $\gamma_1 = 2^{\nu_B} = 2^v$.

The TTL on a cache entry will automatically lead to a refresh. However, a sibling node may already need to be added before a typical TTL is reached. Consequently, it may be helpful to set the TTL on the condensed signature in proportion to the time expected until the next sibling node would be added. Because the rate at which leaf

nodes are added may be hard to predict, a time-based approach for refreshing condensed signatures may provide inconsistent results as a mitigation for the likelihood of special processing. An approach based on the number of leaf nodes may be more effective.

We suggest the following tactic: When a responder receives a condensed signature relative to reference value newer than any others it has encountered, say the N'^{th} reference value, or otherwise learns that there are N' (or more) leaf nodes, it invalidates any condensed signature in the cache that is based on an authentication path $\Pi_{i,N}$ where i, N and N' are incompatible. The responder then either proactively refreshes the condensed signature or waits until the associated record is requested by a client, and then refreshes. By updating condensed signatures based on newly encountered reference values, the responder then stays ahead of any verifier that relies on the same source of reference values. It may not be necessary to stay this far ahead, e.g., the verifier may be able to verify $\Pi_{i,N}$ with the reference values it holds, but it's sufficient. (A full treatment would require modeling of the evolution of the set of reference values held by a verifier.)

F.4 Changing Rung Strategy

The *extended binary rung strategy* makes the following enhancement to the binary rung strategy: In addition to the B rungs in the ladder corresponding to the ones bits of the binary representation of N, the ladder also includes $\lfloor \log_2 N \rfloor + 1 - B$ rungs corresponding to the zero bits. The span of each such rung is the same as it was the previous time the binary representation had a one bit in the corresponding position (say, the 2^v position). The rung is thus "extended" for an additional 2^v leaf nodes compared to the binary rung strategy (the number of leaf nodes until the position next has a one bit). (Only rungs corresponding to ones bits are used for constructing authentication paths, which are the same as in the binary rung strategy.)

The extended binary rung strategy shares the binary rung strategy's $O(\log N)$ authentication path and ladder sizes as well as its general path verification property. Due to the extension of the rungs, the extended binary rung strategy also has a lower likelihood of incompatibility in the $i \leq N < N'$ case. In particular, a new sibling node will not need to be added until the number of leaf nodes reaches $R + 2\gamma_1$, $R + 2\gamma_2$, $R + 2\gamma_3$, etc. — a doubling of the distance from R.

The extension can offer a significant advantage in refresh *timing* over the binary rung strategy for the following reason. In the binary rung strategy, rungs are removed from the ladder immediately after they've been in use as selected rungs for producing new authentication paths. Moreover, multiple rungs may be removed at the same time Consider the example in Fig. 1: When the number of leaf nodes in the tree reaches 16, all three rungs shown, [1: 8], [9: 12] and [13: 14], will no longer be used for producing new authentication paths and all three will be removed from the ladder. The addition of a leaf node may therefore trigger many condensed signature refreshes at the same time. Indeed, although the average number of condensed signatures that need to be refreshed for each leaf node added is $O(\log N)$, some leaf node additions may trigger as many as $N - 1$ refreshes. For instance, all 14 authentication paths leading to the three rungs

shown in the example (as well as the one leading to $[15:15]$) will need to be refreshed when the number of leaf nodes in the tree reaches 16.

In the extended binary rung strategy, in contrast, rungs remain in the ladder for an extended period during which they are still available for verifying previous authentication paths. Condensed signature refreshes for authentication paths relative to a rung can therefore be staggered. Returning to Fig. 1, $[1:8]$ will no longer be used for producing new authentication paths when the number of leaf nodes reaches 16, but it won't be removed until the number reaches 24. For the eight authentication paths associated with $[1:8]$, then, we would have eight leaf node additions in which to make the refresh. Put another way, although a sibling node doesn't need to be added until $R + 2\gamma_j$, it can be added as early as $R + \gamma_j$. As a result, we can schedule the refreshes so that there are $O(\log N)$ refreshes for each and every new leaf node added, not just on average, thus distributing the workload more evenly than in the binary rung strategy.