Merkle Tree Ladder Mode: Reducing the Size Impact of NIST PQC Signature Algorithms in Practice
(expanded version)*

Andrew Fregly[1] [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
{afregly,jsharvey,bkaliski,ssheth}@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 49,856 bytes with example parameters at various security levels, to a condensed signature size of 248 to 472 bytes depending on the selected security level. 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

* This article is an expanded version of a contribution to CT-RSA 2023. The changes from the contribution include (a) incorporating the appendices from the prior version of this ePrint, updated for consistency; (b) referencing additional related work; and (c) minor editorial changes. The Version of Record of this contribution was first published in Topics in Cryptology – CT-RSA 2023, Lecture Notes in Computer Science, vol 13871, pp 415-441, 2023 by Springer Nature, and is available online at https://doi.org/10.1007/978-3-031-30872-7_16

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 algorithms [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]. (NIST has recently published its initial selections as draft standards [6,7,8].)

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 [9]. The leading post-quantum signature algorithms in the NIST PQC project, in contrast, have minimum sizes from 666 to 7856 bytes and maximums from 1280 to 49,586 with example parameter sets (see Tables 8 and 9 in [2]) — an order of magnitude (or more) 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 [10], Domain Name System Security Extensions (DNSSEC) records [11] or signed certificate timestamps [12].

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 [13] and with *modes of operation* of cryptographic techniques pioneered by NIST for encryption algorithms [14].

In brief, MTL mode constructs an evolving sequence of Merkle tree nodes, which we call *ladders*, from the series 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 underlying 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 reuse ladders and 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 constructing an entirely new tree. 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.

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. In addition, condensation and reconstitution are *public processes:* They involve only the signer's public key, not its private key. The processes therefore *don't impact the security of the underlying signature scheme* and they can be *performed by anyone*, which adds to deployment flexibility.

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 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 provides preliminary notation and Section 3 introduces Merkle tree ladders. Section 4 defines MTL mode and Section 5 provides a detailed security analysis. Section 6 shows how to condense and reconstitute MTL mode signatures, and Section 7 discusses the practical impact of our techniques on NIST PQC signature algorithms with DNSSEC as an example use case. Section 8 proposes some extensions, Section 9 reviews related work, and Section 10 concludes the main body of the paper. Appendices provide additional details helpful for implementers as well as a proof of one of the technical claims.

2 Preliminaries

Our specifications use several symbols frequently that we define here for reference:

- ℓ is our security parameter, the length in bits of hash values; a typical minimum value for security against quantum adversaries is $\ell = 128$;
- ℓ_c is the length of the randomizer in our message hashing operation; and
- SID is a *series identifier*, a value associated with an instance of MTL mode that provides cryptographic separation from other instances.

We use three families of hash functions: two with fixed input lengths in our Merkle tree operations and one with a variable input length for message hashing:

- $H_{\text{leaf}}(SID, i, d) \longrightarrow V$ maps a series identifier SID, an index i, and a ℓ-bit data value d to an ℓ-bit hash value V;
- $H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) \longrightarrow V$ maps a series identifier SID, a node index pair L and R and two ℓ-bit hash values V_{left} and V_{right} to a ℓ-bit hash value V; and

- $H_{\text{msg}}(SID, i, m, c) \longrightarrow d$ maps a series identifier SID, an index i, a variable-length message m and a ℓ_c-bit randomizer c to a ℓ-bit data value d.

Appendix A proposes instantiations of these functions. The operation RANDOM(ℓ) returns a random ℓ-bit string.

3 Merkle Tree Ladders

For authenticating an evolving series of messages, instead of a Merkle tree with a single root, we maintain an evolving set of perfect binary trees according to the binary representation of *N*, the number of messages. Consider Fig. 1, which shows how we would authenticate 14 messages. The binary representation of 14 is $8 + 4 + 2$. We put the first eight messages (bottom row) in a tree with eight leaf nodes (the row above it). The root of this tree is denoted $[1{:}\,8]$, indicating that it authenticates or *spans* leaf nodes 1 through 8. The next four messages go in a tree with four leaf nodes with root $[9{:}\,12]$. The last two go into a tree with root $[13{:}\,14]$.

We call the set of root nodes spanning the tree leaves a *Merkle tree ladder*, which we envision as a way of "climbing the trees" and reaching the evolving set of roots. We refer to the tree roots as *rungs* and the full set of leaf and internal nodes as a *Merkle node set* (since it is not necessarily a single tree); we call this particular arrangement of rungs the *binary rung strategy*. As new leaf nodes are added to the right, new trees are formed; rungs are added to the ladder and removed. For instance, when the 15^{th} leaf node is added, the rung $[15{:}\,15]$ would be added to the ladder. When the 16^{th} is added, the four previous rungs would be replaced by $[1{:}\,16]$.

As usual in Merkle tree authentication, each node has a hash value that is computed from the hash values of its descendants. We also include a series identifier SID that cryptographically separates this node set from other node sets. We denote the hash value at the node spanning leaf nodes L through R as $V[L{:}\,R]$. When $L = R$, we have a leaf node with index $i = L = R$ and we compute

$$V[i{:}\,i] := V_i := H_{\text{leaf}}(SID, i, d)$$

where d is a ℓ-bit data value corresponding to the i^{th} message. (We will show later how the data value is computed from the message.) When $L < R$, we have an internal node and compute

$$V[L{:}\,R] := H_{\text{int}}(SID, L, R, V[L{:}\,M], V[M+1{:}\,R])$$

where $V[L{:}\,M]$ and $V[M+1{:}\,R]$ are the hash values of the child nodes of $V[L{:}\,R]$ and $M = (L + R - 1)/2$.

We compute the N^{th} ladder, denoted Λ_N, as follows. Write $N = \sum_{j=1}^{B} 2^{\nu_j}$, where the ν_j are the indexes of the 1-bits in the binary representation of N from highest to lowest, so that $\lfloor \log_2 N \rfloor = \nu_1 > \nu_2 > \cdots > \nu_B \geq 0$. Λ_N then consists of the hash values $V[L_N(1){:}\,R_N(1)], \ldots, V[L_N(B){:}\,R_N(B)]$ where we define $R_N(0) = 0$ and for $j = 1$ to B, we set $L_N(j) = R_N(j-1) + 1$ and $R_N(j) = R_N(j-1) + 2^{\nu_j}$. In the example, the 14^{th} ladder Λ_{14} consists of the hash values $V[1{:}\,8]$, $V[9{:}\,12]$ and $V[13{:}\,14]$.

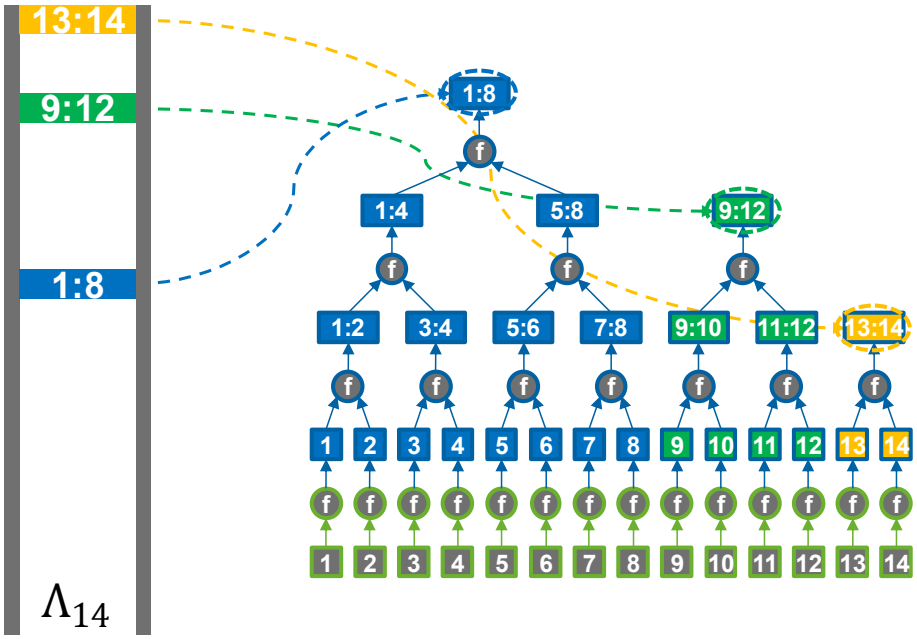

Fig. 1. Example of a Merkle tree ladder following a binary rung strategy. Rungs $[1:8]$, $[9:12]$ and $[13:14]$ collectively authenticate all 14 leaf nodes.

We can compute the authentication path from the i^{th} leaf node to the N^{th} ladder, denoted $\Pi_{i,N}$, in the usual way by including the sibling nodes from the i^{th} leaf node to the root of its tree. In the example, the authentication path $\Pi_{10,14}$ for the 10^{th} leaf node $V[10:10]$ consists of the sibling hash values $V[9:9]$ and $V[11:12]$ leading to the rung hash value $V[9:12]$. The position of the rung among the hash values in the ladder is determined uniquely by i and N.

What's convenient about the binary rung strategy (and what has made it attractive in other contexts — see the related work in Section 9) is that it has a *backward compatibility* property: An authentication path relative to a new ladder can be verified using an old ladder. For example, consider the authentication path $\Pi_{10,16}$ for the 10^{th} leaf node relative to the 16^{th} ladder $\Lambda_{16} = [1:16]$. It consists of sibling hash values $V[9:9]$, $V[11:12]$, $V[13:16]$ and $V[1:8]$. $\Pi_{10,16}$ can naturally be authenticated relative to Λ_{16}. But it can also be authenticated relative to Λ_{14} (and any other ladder between Λ_{10} and Λ_{16}), because the authentication recomputes the old rung hash value $V[9:12]$ as an intermediate step on the way to $V[1:16]$.

Node set operations. We define four operations for interacting with a node set:

- *Node set initialization.* INITNODESET$(SID) \longrightarrow T$ returns a new node set T with the series identifier SID.

- *Leaf node addition.* $\text{ADDLEAFNODE}(T, d) \longrightarrow \langle \Lambda_N \rangle$ adds a leaf node corresponding to a data value d to the node set T and returns the current ladder Λ_N where N is the current leaf node count.
- *Authentication path construction.* $\text{GETAUTHPATH}(T, i) \longrightarrow \Pi_{i,N}$ returns the authentication path $\Pi_{i,N}$ from the i^{th} leaf node in the node set T relative to the current ladder. The operation requires that $1 \leq i \leq N$.
- *Authentication path verification.* $\text{CHECKAUTHPATH}(SID, i, N, N', d, \Pi_{i,N}, \Lambda_{N'}) \longrightarrow b$ verifies that the i^{th} leaf node corresponds to a data value d using an authentication path $\Pi_{i,N}$ from the i^{th} leaf node relative to the N^{th} ladder Λ_N, and the N'^{th} ladder $\Lambda_{N'}$. It returns $b = \text{TRUE}$ if the authentication path is valid and $b = \text{FALSE}$ otherwise. The operation requires that $1 \leq i \leq N' \leq N$.

Appendix B gives pseudocode for the node set operations.

We can formalize the backward compatibility property as follows:

Backward compatibility. 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, $\Pi_{i,N}$ 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

$$\text{CHECKAUTHPATH}(SID, i, N, N', d_i, \Pi_{i,N}, \Lambda_{N'}) = \text{TRUE}.$$

The proof is given in Appendix C.

4 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 stateful 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 constructed from the messages that are signed, and sign each ladder using $\mathcal{S}$. We call the transformation *Merkle Tree Ladder (MTL) mode* and designate a signature scheme $\mathcal{S}$ in MTL mode as $\mathcal{S}$-MTL.

MTL mode has the following profile:

- *Public key* $pk = \langle pk^{\mathcal{S}}, SID \rangle$ where $pk^{\mathcal{S}}$ is a public key for the underlying scheme $\mathcal{S}$ and SID is a ℓ-bit series identifier.
- *Private key* $sk = \langle sk^{\mathcal{S}}, SID, N, T \rangle$ where $sk^{\mathcal{S}}$ is the corresponding private key for $\mathcal{S}$, SID is the matching series identifier, N is number of signatures produced and T is the evolving node set constructed from the messages that have been signed so far. sk includes state; the signature operation updates sk in place.
- *Signature* $\sigma = \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma^{\mathcal{S}}_{N'}, d_i^* \rangle$ where SID is the series identifier, c_i is a randomizer, i, N and N' are indexes, $\Pi_{i,N}$ is an authentication path to the ladder, $\Lambda_{N'}$ is the ladder, $\sigma^{\mathcal{S}}_{N'}$ is a signature on the ladder under $\mathcal{S}$ and d_i^* is an optional data value.

KeyGen$(1^\ell) \longrightarrow \langle pk, sk\rangle$: $\langle pk.pk^{\mathcal{S}}, sk.sk^{\mathcal{S}}\rangle := \mathcal{S}.\text{KeyGen}(1^\ell)$. $SID := \text{Random}(\ell)$. $pk.SID := sk.SID := SID$. $sk.N := 0$. $sk.T := \text{InitNodeSet}(SID)$. Return $\langle pk, sk\rangle$. Sign$(sk, m) \longrightarrow \sigma$: $i := sk.N + 1$. $SID := sk.SID$. $c_i := \text{Random}(\ell_c)$. $d_i := H_{\text{msg}}(SID, i, m, c_i)$. $\Lambda_i := \text{AddLeafNode}(sk.T, d_i)$. $\Pi_{i,i} := \text{GetAuthPath}(sk.T, i)$. $\sigma_i^{\mathcal{S}} := \mathcal{S}.\text{Sign}(sk.sk^{\mathcal{S}}, \langle SID, 1, i, \Lambda_i\rangle)$. $\sigma \Longleftarrow \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^{\mathcal{S}}, d_i\rangle$. $sk.N := i$. Return σ.	Verify$(pk, m, \sigma) \longrightarrow b$: $\sigma \Longrightarrow$ $\langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}}, d_i^*\rangle$. Check $SID = pk.SID$. $b^{\mathcal{S}} := \mathcal{S}.\text{Verify}$ $\left(pk.pk^{\mathcal{S}}, \langle SID, 1, N', \Lambda_{N'}\rangle, \sigma_{N'}^{\mathcal{S}}\right)$. If $b^{\mathcal{S}} = \text{False}$ then return False. $d_i := H_{\text{msg}}(SID, i, m, c_i)$. $b := \text{CheckAuthPath}$ $\left(SID, i, N, N', d_i, \Pi_{i,N}, \Lambda_{N'}\right)$. Return b.

Fig. 2. MTL mode's key pair generation, signature generation and signature verification operations (see text for discussion).

We reference the components of the keys as $sk.SID$, $sk.N$, etc. Note that other than the underlying private key $sk^{\mathcal{S}}$, none of the values in the private key sk needs to be kept secret; they are just included as part of the state. Indeed, all of them including the node set can be reconstructed from the signatures that are generated.

Scheme operations. The mode's operations are detailed in Fig. 2. In brief:

- *Key pair generation.* KeyGen generates a key pair for $\mathcal{S}$, initializes a Merkle tree node set, and forms an MTL mode key pair from the foregoing.
- *Signature generation.* Sign hashes the message with a randomizer, adds a leaf node corresponding to the resulting data value to the private key's node set, signs the current Merkle tree ladder using $\mathcal{S}$, and forms a signature from the authentication path to the ladder, the ladder and the underlying signature.
- *Signature verification.* Verify verifies the underlying signature on the Merkle tree ladder using $\mathcal{S}$, re-hashes the message with the randomizer, and verifies the leaf node corresponding to the resulting data value using the authentication path and the ladder.

Sign only produces signatures with $i = N = N'$ (hence the i, i, i triple in the initial signature format). However, Verify can verify signatures with $i \leq N' \leq N$ due to the backward compatibility property (see Section 3). The difference is the basis for the condensation and reconstitution operations we describe in Section 6. (We do not

consider these signatures forgeries; rather, they are alternative representations of the same signature. MTL mode is a malleable signature scheme in this sense, with the caveats that come from this property — see the related work in Section 9.)

5 Security Analysis

We now give two detailed security proofs of MTL mode. Our terminology and notation generally follows XMSS-T [15]. We adopt the common security goals of existential unforgeability against chosen message attacks (EU-CMA) and random message attacks (EU-RMA); and multi-target, multi-function second preimage resistance (MM-SPR). InSec denotes the maximum success probability that an adversary breaks a specific security goal within a certain number of queries (and running time).

Our analysis assumes the series identifier SID is different for every instance of MTL mode. The analysis thus scales to the multi-user setting, as every invocation of the scheme's hash functions will have different inputs. MTL mode's key pair generation operation generates SID as a random string, but it could also include a unique identifier. Fluhrer's proof for LMS [16] models the possibility of SID collisions and could be adapted here.

5.1 Random Oracle Model Proof

We start with a basic proof in the random oracle model against classical adversaries. Motivated by Fluhrer's and Katz's [17] proofs for LMS, we model all three hash functions as random oracles. (Fluhrer also observes the importance of ensuring appropriate interaction with hash function's compression function; we defer such details to specific instantiations.)

Theorem 1. $\mathcal{S}$-MTL is EU-CMA in the random oracle model if:

- $\mathcal{S}$ is EU-CMA in the random oracle model;
- H_{msg}, H_{leaf} and H_{int} are modeled as independent random oracles; and
- the random oracles are independent of one another and any assumed in the security analysis of $\mathcal{S}$.

In particular, we have for classical adversaries,

$$\text{InSec}^{\text{EU-CMA}}(\mathcal{S}\text{-MTL}; \xi) \leq \text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi) + \frac{(q+1)}{2^{\ell}} + \frac{q}{2^{\ell_c}},$$

where q is the total number of oracle queries to H_{msg}, H_{leaf} and H_{int} made by the adversary and ξ is the adversary's running time.

Proof. We engage the adversary $\mathcal{A}$ in the following EU-CMA experiment:

1. Generate a key pair $\langle pk, sk \rangle$ by calling $\mathcal{S}$-MTL.KeyGen.
2. Generate q_{s} ℓ-bit data values $d_1, \ldots, d_{q_{\text{s}}}$ at random, where q_{s} is a bound on the number of signatures requested by the adversary.

3. Give $\mathcal{A}$ the public key pk, access to $\mathcal{S}$-MTL.SIGN and oracle access to H_{msg}, H_{leaf} and H_{int} (and any oracles in $\mathcal{S}$).
   - We modify the call to H_{msg} from within SIGN as follows: When SIGN calls $H_{\text{msg}}(SID, i, m_i, c_i)$, where m_i is the message provided by $\mathcal{A}$ in the i^{th} SIGN query, we program H_{msg} so that $H_{\text{msg}}(SID, i, m_i, c_i) = d_i$.
4. Await a forgery from $\mathcal{A}$.

Now suppose that $\mathcal{A}$ succeeds in producing a forgery $\langle \hat{m}, \hat{\sigma} \rangle$ with $\hat{m} \neq m_i$. Parse $\hat{\sigma} \Rightarrow \langle SID, \hat{c}, i, N, N', \hat{\Pi}, \hat{\Lambda}, \widehat{\sigma^{\mathcal{S}}}, \widehat{d^*} \rangle$ and set $\hat{d} = H_{\text{msg}}(SID, i, \hat{m}, \hat{c})$ and $\hat{V} = H_{\text{leaf}}(SID, i, \hat{d})$. Further assume for the moment that $\mathcal{A}$ hasn't queried H_{msg} for any $(SID, j, *, c_j)$ prior to the j^{th} SIGN query. $\mathcal{A}$ then can't detect the reprogramming; the values produced in the signing operations will all be random to $\mathcal{A}$. Let $\bar{V} = \{\langle L, R, V[L{:}R] \rangle\}$ and $\bar{\Lambda}$ be the nodes and ladders produced during the signing operations.

The forgery will fall into one of the following cases:

- ***S* forgery.** If $\hat{\Lambda} \notin \bar{\Lambda}$, then, with $\mathcal{A}$'s assistance, we've produced a signature forgery $\langle \hat{\Lambda}, \widehat{\sigma^{\mathcal{S}}} \rangle$ against $\mathcal{S}$. Because our experiment interacts with $\mathcal{S}$ only through its KEYGEN and SIGN operations, it's achieved EU-CMA success against $\mathcal{S}$.
- ***H*int second preimage.** If $\hat{\Lambda} \in \bar{\Lambda}$ but $\langle i, i, \hat{V} \rangle \notin \bar{V}$, then $\mathcal{A}$ has found a H_{int} second preimage in the evaluation of the authentication path $\hat{\Pi}$ from the i^{th} leaf node to its associated rung in $\hat{\Lambda}$: $H_{\text{int}}(SID, L, R, \widehat{V_{\text{left}}}, \widehat{V_{\text{right}}}) = H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}})$ at some node position $[L{:}R]$ but $(\widehat{V_{\text{left}}}, \widehat{V_{\text{right}}}) \neq (V_{\text{left}}, V_{\text{right}})$.
- ***H*leaf second preimage.** If $\langle i, i, \hat{V} \rangle \in \bar{V}$ but $\hat{d} \neq d_i$**,** then $\mathcal{A}$ has found a H_{leaf} second preimage: $H_{\text{leaf}}(SID, i, \hat{d}) = H_{\text{leaf}}(SID, i, d_i)$.
- ***H*msg second preimage.** If $\hat{d} = d_i$, then $\mathcal{A}$ has found a H_{msg} second preimage: $H_{\text{msg}}(SID, i, \hat{m}, \hat{c}) = H_{\text{msg}}(SID, i, m_i, c_i)$.

The probability that any adversary produces a $\mathcal{S}$ forgery is bounded by $\text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi)$. The probability that $\mathcal{A}$ finds a H_{int} second preimage with any single query to the oracle is at most $1/2^{\ell}$. (The logic is as follows: only queries of the form $H_{\text{int}}(SID, L, R, *, *)$ can target the node at position $[L{:}R]$; $\bar{V}$ includes only one node value at this position; so each H_{int} query has probability at most $1/2^{\ell}$ of yielding a second preimage.) The probabilities for H_{leaf} and H_{msg} are each $1/2^{\ell}$ by similar logic. As the oracles are independent of any in the security analysis of $\mathcal{S}$, we can add the bounds to $\text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi)$. We then add the probability that $\mathcal{A}$ has queried H_{msg} for some $(SID, j, *, c_j)$ prior to the j^{th} SIGN query, which is at most $q/2^{\ell_c}$, and the probability $1/2^{\ell}$ that the adversary has simply guessed a message that happens match a hash value target without making an oracle query, and the result follows. ■

5.2 (Mostly) Standard Model Proof for a "Robust" Variant

We now offer an alternative proof that is in the standard model (without random oracles) for two underlying hash functions. To do so, we define a variant of MTL mode called *MTLr mode*, where the H_{int} and H_{leaf} computations embed challenge preimages, following XMSS-T's design (and SPHINCS^{+}'s terminology; the "r" is for "robust"). Adapting the design to our notation and framework, we use two additional families of fixed-input-length hash functions and two families of pseudorandom functions which we model as random oracles:

- $H'_{\text{leaf}}(k_{\text{leaf}}, m_{\text{leaf}}) \longrightarrow V$ maps a ℓ-bit key k_{leaf} and a ℓ-bit message m_{leaf} to a ℓ-bit hash value;
- $H'_{\text{int}}(k_{\text{int}}, m_{\text{int}}) \longrightarrow V$ maps a ℓ-bit key k_{int} and a 2ℓ-bit message m to a ℓ-bit hash value;
- $F'_{\text{leaf}}(SID, i) \longrightarrow \langle k_{\text{leaf}}, r_{\text{leaf}} \rangle$ maps a ℓ-bit series identifier SID and a leaf index i to a ℓ-bit key k_{leaf} and a ℓ-bit mask r_{leaf}; and
- $F'_{\text{int}}(SID, \langle L, R \rangle) \longrightarrow \langle k_{\text{int}}, r_{\text{int}} \rangle$ maps a ℓ-bit series identifier SID and a node index pair $\langle L, R \rangle$ to a ℓ-bit key k_{int} and a 2ℓ-bit mask r_{int}.

In MTLr mode, H_{int} and H_{leaf} are defined as

- $H_{\text{leaf}}(SID, i, d_i) = H'_{\text{leaf}}(k_{\text{leaf}}, d_i \oplus r_{\text{leaf}})$; and
- $H_{\text{int}}\big(SID, L, R, V_{\text{left}}, V_{\text{right}}\big) = H'_{\text{int}}\big(k_{\text{int}}, \big(V_{\text{left}} \parallel V_{\text{right}}\big) \oplus r_{\text{int}}\big)$,

where $\langle k_{\text{leaf}}, r_{\text{leaf}} \rangle = F'_{\text{leaf}}(SID, i)$ and $\langle k_{\text{int}}, r_{\text{int}} \rangle = F'_{\text{int}}(SID, \langle L, R \rangle)$. We denote the application of MTLr mode to $\mathcal{S}$ as $\mathcal{S}$-MTLr. We are now ready for our second theorem.

Theorem 2. $\mathcal{S}$-MTLr is EU-CMA in the random oracle model if:

- $\mathcal{S}$ is EU-CMA in the random oracle model;
- H_{msg} is a modeled as a random oracle;
- H'_{leaf} and H'_{int} are multi-target multi-function second-preimage-resistant hash function families;
- F'_{leaf} and F'_{int} are modeled as random oracles; and
- the random oracles are independent of one another and any assumed in the security analysis of $\mathcal{S}$.

In particular, we have

$$\text{InSec}^{\text{EU-CMA}}(\mathcal{S}\text{-MTLr}; \xi) \leq \text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi) + \text{InSec}^{\text{MM-SPR}}(H'_{\text{leaf}}; \xi) + \text{InSec}^{\text{MM-SPR}}(H'_{\text{int}}; \xi) + \frac{(q+1)}{2^{\ell}} + \frac{q}{2^{\ell_c}}$$

for classical adversaries and

$$\text{InSec}^{\text{EU-CMA}}(\mathcal{S}\text{-MTLr}; \xi) \leq \text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi) + \text{InSec}^{\text{MM-SPR}}(H'_{\text{leaf}}; \xi)$$

$$+\mathrm{InSec}^{\text{MM-SPR}}(H'_{\text{int}};\xi) + \frac{8(q+q_{\text{s}}+2)^2}{2^{\ell}} + 3q_{\text{s}}\sqrt{\frac{q+q_{\text{s}}+1}{2^{\ell_c}}}.$$

for quantum adversaries, where ξ is the adversary's running time, q is the number of queries to the MTLr mode oracles and q_{s} is a bound on the number of signatures requested by the adversary. (For simplicity, we ignore queries to $\mathcal{S}$'s oracles, if any.)

Proof. Observe that $\mathcal{S}$-MTLr (as well as $\mathcal{S}$-MTL) employs a hash-and-sign construction; denote its internal fixed-message-length signature scheme (the processing of the data value d) as $\mathcal{S}$-MTLr$^{\#}$. Grilo et al. [18] recently gave a general bound for this construction in the quantum random oracle model for the case that the fixed-message-length scheme is EU-RMA. Dropping the present schemes (and notation) into their bound, we get

$$\mathrm{InSec}^{\text{EU-CMA}}(\mathcal{S}\text{-MTLr};\xi) \le \mathrm{InSec}^{\text{EU-RMA}}\left(\mathcal{S}\text{-MTLr}^{\#};\xi\right) + \frac{8q_{\text{s}}(q+q_{\text{s}}+2)^2}{2^{\ell}} + 3q_{\text{s}}\sqrt{\frac{q+q_{\text{s}}+1}{2^{\ell_c}}}.$$

Improving a proof by Bos et al. [19], the authors of [18] also gave a tighter bound for the case that the fixed-message-length scheme is EU-CMA, each signature is associated with a separate nonce, and the nonce is also input to H_{msg}. The tighter bound, which they detailed for XMSS-T, removes the q_{s} factor in the first term (due to the nonce) and halves the factor of 3 (due to the move to EU-CMA). Like XMSS-T's internal fixed-message-length scheme, $\mathcal{S}$-MTLr$^{\#}$ also associates each signature with separate nonce (the index i), and the nonce is also input to H_{msg} in $\mathcal{S}$-MTLr. We argue that the factor of q_{s} can be removed from the first term for $\mathcal{S}$-MTLr for the same reason (but the factor of 3 in the second term remains).

Our remaining task is to analyze the security of $\mathcal{S}$-MTLr$^{\#}$ against a random message attack. Motivated by the proofs for XMSS-T and SPHINCS^{+}, we engage the $\mathcal{S}$-MTLr$^{\#}$ adversary $\mathcal{A}^{\#}$ in the following experiment which also interacts with MM-SPR challengers for H'_{leaf} and H'_{int}:

1. Generate a key pair $\langle pk, sk\rangle$ by calling $\mathcal{S}$-MTLr$^{\#}$'s KeyGen operation.
2. Generate q_{s} ℓ-bit data values $d_1, \ldots, d_{q_{\text{s}}}$ at random.
3. Call $\mathcal{S}$-MTLr$^{\#}$'s Sign operation on each data value $d_1, \ldots, d_{q_{\text{s}}}$ in succession, producing signatures $\sigma_1^{\#}, \ldots, \sigma_{q_{\text{s}}}^{\#}$.

— We modify the calls to H_{leaf} and H_{int} from within Sign as follows:
  - When Sign calls $H_{\text{leaf}}(SID, i, d_i)$ and H_{leaf} calls $F'_{\text{leaf}}(SID, i)$, we call the MM-SPR challenger for H'_{leaf} to get a new challenge $(k_{\text{leaf}}, m_{\text{leaf}})$ then program F'_{leaf} so that $F'_{\text{leaf}}(SID, i) = \langle k_{\text{leaf}}, r_{\text{leaf}}\rangle$ where $r_{\text{leaf}} = d_i \oplus m_{\text{leaf}}$; H_{leaf} will then compute $H_{\text{leaf}}(SID, i, d_i) = H'_{\text{leaf}}(k_{\text{leaf}}, d_i \oplus r_{\text{leaf}}) = H'_{\text{leaf}}(k_{\text{leaf}}, m_{\text{leaf}})$, thus embedding the challenge preimage.
  - When Sign calls $H_{\text{int}}\left(SID, L, R, V_{\text{left}}, V_{\text{right}}\right)$ and H_{int} calls $F'_{\text{int}}(SID, \langle L, R\rangle)$, we call the MM-SPR challenger for H'_{int} to get a new challenge $(k_{\text{int}}, m_{\text{int}})$ then

program F'_{int} so that $F'_{\text{int}}(SID, \langle L, R\rangle) = \langle k_{\text{int}}, r_{\text{int}}\rangle$ where $r_{\text{int}} = \left(V_{\text{left}} \parallel V_{\text{right}}\right) \oplus m_{\text{int}}$.

4. Give $\mathcal{A}^{\#}$ the public key pk, the data values $d_1, \ldots d_{q_S}$, the signatures $\sigma_1^{\#}, \ldots, \sigma_{q_S}^{\#}$, and oracle access to F'_{int} and F'_{leaf} (and any oracles in $\mathcal{S}$).
5. Await a forgery from $\mathcal{A}^{\#}$.

As in Theorem 1, let $\bar{V} = \{\langle L, R, V[L{:}R]\rangle\}$ and $\bar{\Lambda}$ be the nodes and ladders produced during the signing operations. Now suppose that $\mathcal{A}^{\#}$ succeeds in producing a forgery $\langle \hat{d}, \widehat{\sigma^{\#}}\rangle$ with $\hat{d} \neq d_i$. Parse $\widehat{\sigma^{\#}} \Rightarrow \langle SID, i, N, N', \hat{\Pi}, \hat{\Lambda}, \widehat{\sigma^{\mathcal{S}}}, \widehat{d^*}\rangle$ and set $\hat{V} = H_{\text{leaf}}\left(SID, i, \hat{d}\right)$. The forgery will fall into one of the following cases, which are comparable to those in Theorem 1 but in the standard model (and without H_{msg}):

- ***S* forgery.** If $\hat{\Lambda} \notin \bar{\Lambda}$, then with $\mathcal{A}^{\#}$'s assistance, we've produced a signature forgery $\langle \hat{\Lambda}, \widehat{\sigma^{\mathcal{S}}}\rangle$ against $\mathcal{S}$.
- ***H*′int second preimage.** If $\hat{\Lambda} \in \bar{\Lambda}$ but $\langle i, i, \hat{V}\rangle \notin \bar{V}$, then $\mathcal{A}^{\#}$ has found a H_{int} second preimage: $H_{\text{int}}\left(SID, L, R, \widehat{V_{\text{left}}}, \widehat{V_{\text{right}}}\right) = H_{\text{int}}\left(SID, L, R, V_{\text{left}}, V_{\text{right}}\right)$. Expanding H_{int}, we get $H'_{\text{int}}\left(k_{\text{int}}, \left(\widehat{V_{\text{left}}} \parallel \widehat{V_{\text{right}}}\right) \oplus r_{\text{int}}\right) = H'_{\text{int}}\left(k_{\text{int}}, \left(V_{\text{left}} \parallel V_{\text{right}}\right) \oplus r_{\text{int}}\right)$ where $\langle k_{\text{int}}, r_{\text{int}}\rangle = F'_{\text{int}}(SID, \langle L, R\rangle)$. The right-hand inputs to H'_{int} are one of the MM-SPR challenges, so the left-hand inputs are a H'_{int} second preimage.
- ***H*′ileaf second preimage.** If $\langle i, i, \hat{V}\rangle \in \bar{V}$ but $\hat{d} \neq d_i$, then $\mathcal{A}^{\#}$ has found a H_{leaf} second preimage: $H_{\text{leaf}}\left(SID, i, \hat{d}\right) = H_{\text{leaf}}(SID, i, d_i)$. Expanding, we get $H'_{\text{leaf}}\left(k_{\text{leaf}}, \hat{d} \oplus r_{\text{leaf}}\right) = H'_{\text{leaf}}(k_{\text{leaf}}, d_i \oplus r_{\text{leaf}})$ where $\langle k_{\text{leaf}}, r_{\text{leaf}}\rangle = F'_{\text{leaf}}(SID, i)$; the left-hand inputs are a H'_{leaf} second preimage.

The probability of a $\mathcal{S}$ forgery is bounded by $\text{InSec}^{\text{EU-CMA}}(\mathcal{S}; \xi)$. The probability that $\mathcal{A}^{\#}$ finds a H'_{int} second preimage is at most $\text{InSec}^{\text{MM-SPR}}(H'_{\text{int}}; \xi)$ and the probability of a H'_{leaf} second preimage is similarly $\text{InSec}^{\text{MM-SPR}}(H'_{\text{leaf}}; \xi)$. As the oracles are independent of any in the security analysis of $\mathcal{S}$, we can again add the bounds to get $\text{InSec}^{\text{EU-CMA}}(\mathcal{S}\text{-MTLr}; \xi)$.

Because the programming of F'_{leaf} and F'_{int} occurs before the signatures are given to $\mathcal{A}^{\#}$ and $\mathcal{A}^{\#}$ does not have access to the SIGN operation, the programming does not affect our bounds (except that F'_{leaf} and F'_{int} must be modeled as random oracles). Adding the terms from Grilo et al.'s reduction, the result for quantum adversaries follows.

For classical adversaries, we simply add the two final terms from Theorem 1 instead of their quantum random oracle counterparts.■

5.3 Bit Security of MTL Mode

We can now estimate the bit security of MTL (and MTLr) mode. As usual, we are interested in determining the log of the number of hash function queries for which the adversary's success probability equals 1. Because we don't necessarily know the bit security of the underlying scheme $\mathcal{S}$, however, we focus instead on the *incremental* success probability due to MTL mode's components and estimate the number of queries for which this probability reaches $1/2$ (leaving the other $1/2$ for the underlying

scheme). For brevity, we focus our analysis on the security parameter $\ell = 256$ and initially assume $\ell = \ell_c$.

For the fully random model proof in Theorem 1 against classical adversaries, the incremental success probability is $q/2^{\ell} + q/2^{\ell_c}$. This gives us a classical security level of 254 bits: $2^{254}/2^{256} + 2^{254}/2^{256} = 1/2$.

For the mostly standard model proof in Theorem 2, we assume the bounds on generic attacks for MM-SPR given in [15]. This gives us a classical security level of 253 bits. For quantum adversaries, we need to set the bound on the number of signatures q_s. (The number doesn't directly affect the classical bit security bounds.) Following [15], we initially consider two cases, $q_s = 2^{20}$ and $q_s = 2^{60}$; for both, we get 125 bits quantum security. The quantum security level begins to decline around $q_s \approx 2^{\ell/4} = 2^{64}$ if $\ell = \ell_c = 256$; at that point, the second term in the reduction begins to dominate the first. We can maintain the quantum security level by then increasing the size of the randomizer as in [15]. For example, for $q_s = 2^{64}$ and $\ell_c = 259$, we get 125 bits quantum security. Adding in the adversary's cost of evaluating the hash functions, MTL mode with these parameters arguably reaches NIST's security level V [20] where the attack difficulty is comparable to 256-bit exhaustive key search.

We can also target NIST's security level I, comparable to 128-bit exhaustive key search. With $\ell = \ell_c = 128$ and $q_s = 2^{20}$ or $q_s = 2^{60}$, we get 125 bits classical and 61 bits quantum security. In summary, MTL mode does not significantly reduce the bit security of the underlying scheme $\mathcal{S}$.

6 Condensing and Reconstituting MTL Mode Signatures

We now show how a signer can convey multiple MTL mode signatures to a verifier in fewer bits than if the signatures were sent individually. Our approach is based on the backward compatibility property: once the signer has provided a ladder $\Lambda_{N'}$ to the verifier (signed with the underlying signature scheme), the verifier can verify any message m_i where $i \leq N'$ given an authentication path $\Pi_{i,N}$ constructed relative to any ladder Λ_N where $i \leq N' \leq N$. As a result, the amount of information required to convey multiple MTL mode signatures to a verify is essentially one authentication path per message, plus a signed ladder when needed.

We formalize our approach as follows (see Fig. 3):

- *(Condensation.)* For each signature of interest to the verifier, the signer computes, from the initial (uncondensed) signatures produced by the MTL mode signature operation, a *condensed signature* ς and a *reference value handle* χ. The signer sends these values instead of the initial signature σ. The handle refers to a *reference value* υ that the signer provides to the verifier separately.
- *(Reconstitution.)* The verifier computes a *reconstituted signature* σ' from the condensed signature and a reference value. If the verifier doesn't have a suitable reference value, it requests one based on the handle. The reference value may or may not be the same one referred to by the handle. The verifier can then verify the reconstituted signature using the MTL mode verification operation.

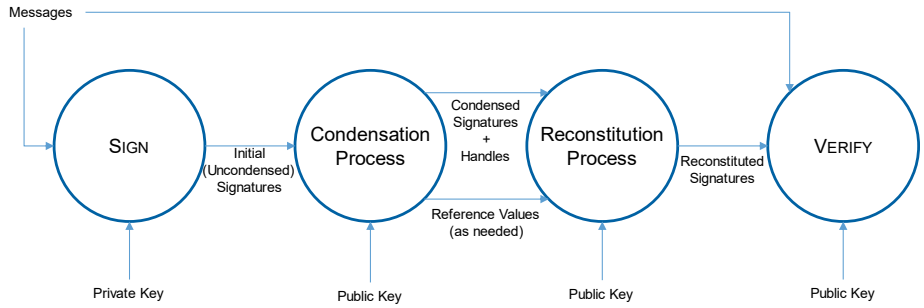

Fig. 3. Condensation and reconstitution processes applied to a signature scheme.

For MTL mode, the relevant values are:

- *Initial signature* $\sigma = \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^{\mathcal{S}}, d_i \rangle$.
- *Condensed signature* $\varsigma = \langle SID, c_i, N, \Pi_{i,N} \rangle$ where c_i is a randomizer and $\Pi_{i,N}$ is the authentication path from the i^{th} leaf node to the N^{th} ladder.
- *Reference value handle* $\chi = N$ where N is the index of the ladder.
- *Reference value* $\upsilon = \langle N', \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}} \rangle$ where $\Lambda_{N'}$ is a ladder and $\sigma_{N'}^{\mathcal{S}}$ is the underlying signature on the ladder.
- *Reconstituted signature* $\sigma' = \langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}}, \emptyset \rangle$, combining elements of the i^{th} condensed signature relative to the N^{th} ladder with the N'^{th} ladder. (The final element, the data value, is not needed for signature verification.)

A set of condensation and reconstitution operations for MTL mode are presented in Fig. 4. They follow a generalized *condensation scheme* $\mathcal{CS}$ with five operations:

- *Initialization.* $\text{CONDENSEINIT}(pk) \longrightarrow \langle st \rangle$ returns a new condensation state st relative to the public key pk.
- *Signature incorporation.* $\text{ADDINITSIG}(st, \sigma)$ incorporates the next previously generated initial signature σ into the state st.
- *Condensed signature production.* $\text{GETCONDENSEDSIG}(st, i) \longrightarrow \langle \varsigma, \chi \rangle$ condenses the i^{th} signature in the state and returns the condensed signature ς and an associated reference value handle χ.
- *Reference value production.* $\text{GETREFVAL}(st, \chi) \longrightarrow \langle \upsilon \rangle$ returns the reference value υ associated with the handle χ.
- *Signature reconstitution.* $\text{RECONSTSIG}(\varsigma, \upsilon) \longrightarrow \langle \sigma' \rangle$ reconstitutes a signature σ' from a condensed signature ς and a reference value υ. (Appendix E proposes an alternative stateful set of reconstitution operations that includes a check for reference value compatibility.)

The operations involve access only to the signer's public key so they don't affect the security analysis in the previous section. Moreover, the operations can be performed by anyone who has access to the signatures / reference values, not just the signer or verifier.

CondenseInit$(pk) \longrightarrow st$: $st.SID := pk.SID$. $st.N := 0$. $st.T :=$ InitNodeSet(SID). $st.\bar{c} := \emptyset$. $st.\overline{\Lambda} := \emptyset$. $st.\Sigma^{\mathcal{S}} = \emptyset$. Return st. AddInitSig(st, σ): $st.N := st.N + 1$. $\sigma \Longrightarrow \langle SID, c_i, i, i, i, \Pi_{i,i}, \Lambda_i, \sigma_i^{\mathcal{S}}, d_i \rangle$. $\Lambda_i^* :=$ AddLeafNode$(st.T, d_i)$. $st.\bar{c} := st.\bar{c} \parallel c_i$. $st.\overline{\Lambda} := st.\overline{\Lambda} \parallel \Lambda_i$. $st.\Sigma^{\mathcal{S}} := st.\Sigma^{\mathcal{S}} \parallel \sigma_i^{\mathcal{S}}$. *[optional checks]* [Check $SID = st.SID$.] [Check $i = st.N$.] [Check $\Lambda_i = \Lambda_i^*$.]	GetCondensedSig$(st, i) \longrightarrow \langle \varsigma, \chi \rangle$: Check $1 \leq i \leq st.N$. $\Pi_{i,N} :=$ GetAuthPath$(st.T, i)$. $\varsigma \Longleftarrow \langle st.SID, st.c[i], i, st.N, \Pi_{i,N} \rangle$. $\chi := st.N$. Return $\langle \varsigma, \chi \rangle$. GetRefVal$(st, \chi) \longrightarrow \upsilon$: $N' := \chi$. Check $1 \leq N' \leq st.N$. $\upsilon \Longleftarrow \langle N', st.\overline{\Lambda}[N'], st.\Sigma^{\mathcal{S}}[N'] \rangle$. Return υ. ReconstSig$(\varsigma, \upsilon) \longrightarrow \sigma'$: $\varsigma \Longrightarrow \langle SID, c_i, i, N, \Pi_{i,N} \rangle$. $\upsilon \Longrightarrow \langle N', \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}} \rangle$. $\sigma' \Longleftarrow$ $\langle SID, c_i, i, N, N', \Pi_{i,N}, \Lambda_{N'}, \sigma_{N'}^{\mathcal{S}}, \emptyset \rangle$. Return σ'.

Fig. 4. MTL mode's condensation scheme operations (see text for discussion).

The scheme state st is a tuple $\langle SID, N, T, \bar{c}, \overline{\Lambda}, \Sigma^{\mathcal{S}} \rangle$ where SID is a series identifier, N is the number of initial signatures incorporated into the state, T is the node set, and $\bar{c}$, $\overline{\Lambda}$ and $\Sigma^{\mathcal{S}}$ are respectively the series of randomizers, ladders and underlying signatures in the initial signatures. We reference the components of the state as $st.SID$, $st.N$, etc. We denote the i^{th} randomizer in $\bar{c}$ as $\bar{c}[i]$ and similarly define $\overline{\Lambda}[i]$ and $\Sigma^{\mathcal{S}}[i]$.

For correctness, we need to show that if a signature is reconstituted from a new condensed signature $\Pi_{i,N}$ and a previous reference value $\Lambda_{N'}$, the reconstituted signature can still be verified. This follows from the backward compatibility property of the binary rung strategy. Because CheckAuthPath can verify the authentication path $\Pi_{i,N}$ using any ladder $\Lambda_{N'}$ where $i \leq N' \leq N$, it follows that for any reconstituted signature σ' on a message m_i produced through the condensation / reconstitution process defined here, Verify$(pk, m_i, \sigma') =$ True.

Condensation and reconstitution can also be applied directly to hash-based signature schemes, as illustrated in Appendix D.

7 Practical Impact

We now show that MTL mode can reduce the size impact of the NIST PQC signature algorithms and other signature schemes 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 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}. 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 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. The table shows the shortest and largest example signature sizes in the published specifications of the algorithms; other sizes may also be supported. Note that the maximum number of signatures can vary for the fourth and fifth algorithms, which can give them an advantage particularly over the others that are designed to meet a NIST requirement of a 2^{64} maximum.

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 targeted level V (which is supported by all five algorithms) and selected $\ell = \ell_c = 256$ for MTL mode's parameters. The sizes of the various MTL mode components in bytes can be computed as follows:

Table 1. NIST PQC signature algorithms with shortest and largest example signature sizes in published specifications. Security Level indicates the level specified in NIST's selection criteria [20] as stated by the submitters of the first three algorithms. Level I is comparable to 128-bit exhaustive key search, Level II to 256-bit hash function collision search and Level V to 256-bit exhaustive key search. The fourth and fifth algorithms have been classified based on their security proofs, hence the *. Max. signatures is a security analysis parameter for the first three algorithms. The fourth and fifth use one-time signatures so their maximum is a functional (and security) limit. (XMSS^MT also has optional examples at a higher security level and NIST SP 800-208 includes parameterizations at a lower security level that are not listed here.) CRYSTALS-Dilithium and SPHINCS$^+$ are being standardized in the draft FIPS 204 [7] and FIPS 205 [8] respectively.

Signature Algorithm / Parameters	Security Level	Signature Size (bytes)	Max. Signatures	Ref.
CRYSTALS–Dilithium	II	2420	2^{64}	[21]
CRYSTALS–Dilithium	V	4595	2^{64}	
FALCON-512	I	666	2^{64}	[22]
FALCON-1024	V	1280	2^{64}	
SPHINCS$^+$-128s	I	7856	2^{64}	[23]
SPHINCS$^+$-256f	V	49,856	2^{64}	
HSS/LMS (ParmSet 15)	V*	1616	2^{15}	[24]
HSS/LMS (ParmSet 25/15)	V*	3652	2^{40}	
XMSS^MT (SHA2_20/2_256)	V*	4963	2^{40}	[25]
XMSS^MT (SHA2_60/12_256)	V*	27,688	2^{60}	

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

Here, we've ignored the overhead of the reference value handle χ as well as protocol overheads such as algorithm and public key identifiers 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 the sizes $|\varsigma|$ and $|\upsilon|$ are the same for all signatures for simplicity.)

Fig. 5 shows the effective signature size $\phi(K, 1)$ as a function of K for the five level V 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 $K = 3$ for FALCON and $K = 2$ for the other examples.

Fig. 6 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.

We've focused on security level V. MTL mode could also be parameterized at the lower security levels with further reduction in condensed signature size. For applications where level I is acceptable, we could reduce our parameters to $\ell = \ell_c = 128$. Maintaining $N_0 = 10{,}000$, MTL mode would then have a condensed signature size of 248 bytes, comparable to RSA-2048 today.

7.3 Example Use Case: DNSSEC

The Domain Name System (DNS) [26] is the core protocol for translating human-readable names to internet protocol (IP) addresses. DNSSEC adds digital signatures to DNS records. This use case has been the core motivator for our research because of the size constraints of DNS responses.

In brief, DNS involves a hierarchy of name servers that provide authoritative responses to requests for information about domain names, e.g., for the Internet Protocol (IP) address of a server such as `www.example.com`. Because the DNS records returned in response to a given request are generally predetermined, the accompanying DNSSEC signatures, conveyed in RRSIG records, can typically be generated in advance of the request, and independent of the requester. This arrangement works well for MTL mode: the name server can provide a condensed signature (and a reference value handle) in an RRSIG record, in place of an initial signature, and requester can look up the corresponding reference value, if needed, perhaps from the same name server. (Indeed, the fact that DNS is by nature an online lookup service makes the DNSSEC use case particularly amenable to MTL mode.)

Data analysis. To estimate the potential benefits of MTL mode on effective signature size for DNSSEC, we analyzed published sample files of DNS requests to and responses from authoritative name servers. A conventional DNSSEC signature scheme was used by these servers. We considered how the same request / response flow might be processed if MTL mode condensed signatures were used instead.

The DNS Operations, Analysis, and Research Center (DNS-OARC) provides a platform for researchers to share and analyze DNS data, including the annual Day In The Life of the Internet (DITL) collection [27]. We focused our analysis on the 2015 DITL raw data provided to DNS-OARC by NZRS, the registry operator for the .NZ top-level domain (TLD). We selected this data set because it includes both DNS requests and responses. (All analysis was performed on DNS-OARC's servers except for the final formatting of the results graph, which involved only summary statistics.)

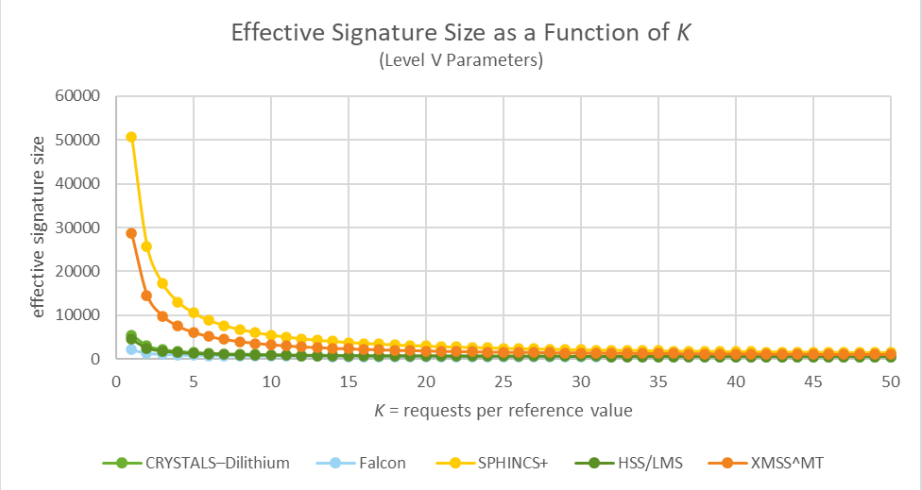

Fig. 5. Effective signature size in bytes for five post-quantum signature algorithms with NIST level V parameters as a function of number of signatures received per reference value.

Each sample file included a series of DNS request / response pairs processed during a specified time period. We considered just the middle 24-hour "day" in the data set (April 14, 2015, 00:00–23:59 UTC) and filtered the request / response pairs to include only those where the response contained an RRSIG record from the .NZ TLD. For simplicity, we also limited our analysis to the common DNS scenario where the traffic was exchanged via the User Datagram Protocol over Internet Protocol version 4. We then organized the RRSIG-containing pairs according to the requester's IP address and the key identifier of the private key that generated the signature (the signer name and key tag in DNSSEC terminology). For each combination of requester IP address and key identifier, we then produced the following time series in chronological order:

$$(t_1, \text{start}_1), (t_2, \text{start}_2), \cdots, (t_K, \text{start}_K)$$

where t_i is the time at which the request / response pair was processed by the name server, start_i is the time at which the signature in the RRSIG record in the response became valid, and K is the number of requests / response pairs. Let RRSIG_i denote the RRSIG record associated with the i^{th} response in the series.

We then measured how often the start value reached a new maximum. We use this "high-water mark" as a proxy for how frequently a requester may need to request a new reference value in MTL mode if the server had used MTL mode condensed signatures.

Our rationale is as follows. Assume for simplicity that every RRSIG record published by an authoritative name server has a different start value and that RRSIG records are signed in increasing order of start value. (For the case $\text{start}_i = \text{start}_j$, we assume that an MTL mode signer would wait until all of the RRSIG records that share a start value have been signed then publish them all at once. The reference value for the

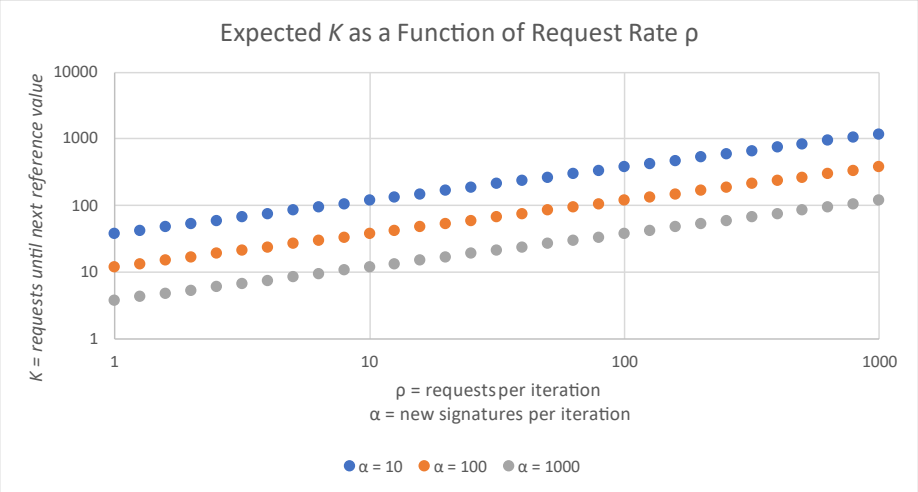

Fig. 6. Expected number of condensed signatures per reference value as a function of request rate and new signature rate.

last one would then cover them all, so a new reference value wouldn't be needed if the start values match.)

Now suppose that start_i is not a new maximum. Then there is an index $j < i$ such that $\text{start}_i < \text{start}_j$. In MTL mode, to have reconstituted and verified RRSIG_j at time t_j, the requester would have at some point obtained a reference value that covers RRSIG_j. Such a reference value would also cover any RRSIG record generated before RRSIG_j, including RRSIG_i. If $\text{start}_i < \text{start}_j$, then the requester would have enough information at time t_i to verify RRSIG_j without requesting a new reference value. The number of high-water marks thus provides an upper bound on K', the number of new reference values the requester would have needed to obtain.

Results. We estimate the average *endurance* of a reference value that would be sent to a requester in MTL mode as the ratio (number of high-water marks) / (number of RRSIG responses) for the requester's IP address. Fig. 7 plots estimated endurance vs. number of RRSIG responses received for each of the requester IP addresses we analyzed. The up-and-to-the-right trend confirms the hypothesis in our model that higher request rates result in higher endurance. Because we didn't have access to the rate α at which new signatures were generated, we couldn't confirm the parameters of the relationship between endurance, the number of records signed, α and the request rate ρ. Nevertheless, what stands out in the plotted data is that reference value endurance is consistently over 10 except for the most slowly querying requesters (fewer than 1000 requests in the 24-hour measurement period; they represent just over a third of all signatures received), and over 100 for the fastest querying requesters. Regardless of query rate, no requester would have looked up more than 29 new reference values in the 24 hours analyzed. MTL mode would therefore significantly reduce the size impact of PQC signatures on these exchanges.

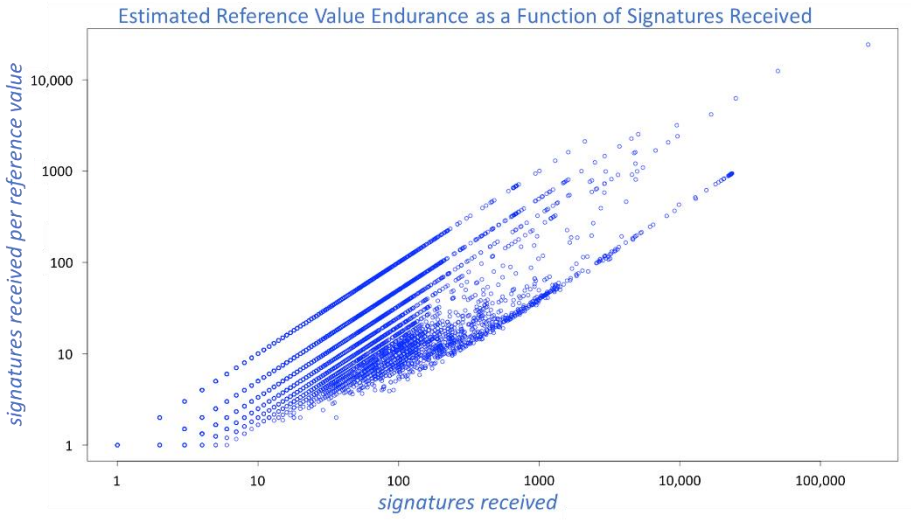

Fig. 7. Estimated endurance of reference values vs. number of signatures received if MTL mode were applied to one of the sample DNS data sets from DNS-OARC's DITL collection.

7.4 Compute, Storage and State Requirements

MTL mode is very efficient. If a node set has N nodes, then the authentication path and the ladder will include at most $\sim \log_2 N$ hash values. Verifying a signature thus takes at most $\sim \log_2 N$ hash operations to verify the authentication path, plus the hash on the message and the underlying signature verification. Generating a signature likewise takes at most $\sim \log_2 N$ hash operations to construct a new authentication path, plus the hash on the message and the underlying signature generation.

A node set with N leaf nodes has at most $N - 1$ internal nodes, so the storage cost for a party performing signature condensation is at most two hash values per message, plus any underlying signatures it maintains as part of reference values. The signer, meanwhile, only needs to keep the nodes in the current ladder and authentication path, as these are sufficient to compute the next ladder and authentication path.

Only the signer in MTL mode needs to maintain state as part of generating initial signatures. The verifier doesn't need to do so to verify signatures, unless the verifier is performing reconstitution operations itself. In the DNSSEC use case, a *resolver* that interacts with name servers on behalf of a collection of clients could perform reconstitution operations on their behalf and provide reconstituted signatures to its clients. (The size constraints on DNS exchanges between resolvers and clients may not be as significant operationally as those between resolvers and authoritative name servers.) A verifier could also perform its own reconstitution operations, in which case a reference value would just be another item for the verifier to keep in its cache, along with local copies of DNS records and DNSSEC public keys.

8 Extensions

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 node sets.* We can reduce the condensed signature size (and/or accommodate more messages) by arranging messages into multiple node sets. So that we don't need additional key pairs, rather than initializing a single node set during MTL mode key pair generation, we could extend MTL mode so that a new node set can be added to an existing key pair. Each such node set would be associated with a separate series identifier, which could be derived from a common seed and a per-series tag. Such an arrangement may be convenient for a signer that has a high message volume and wants to perform signature generation in parallel. (We'd want the sizes of each node set to stay large enough that the ladders maintain a high endurance.)
- *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 also improve signing and verification performance compared to the underlying signature scheme.
- *Hybrid signature schemes.* MTL mode can help make hybrid signature schemes [28,29,30] 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.

9 Related Work

The binary rung strategy appears under different names in other cryptographic constructions based on Merkle trees. Champine defines a *binary numeral tree* [31] with similar structure (the successive perfect 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 [12]. The earlier constructions also include Crosby and Wallach's *history trees* [32] and Todd's *Merkle mountain ranges* [33]. Bünz et al. [34] provide a formal definition and analysis of the latter.

Cryptographic accumulators [35] have a similar structure to condensation and reconstitution in that a common *accumulator value* (viz, reference value) helps a

verifier authenticate multiple elements, each of which has a *witness* relative to the accumulator value (viz, condensed signature). Reyzin and Yakoubov's accumulator [36], applying a binary rung strategy-like construction, also achieves an "old-accumulator compatibility" property comparable to backward compatibility property of the binary rung strategy.

Verkle trees, proposed by Kuszmaul [37] and further elaborated by Buterin [38] replace the hash function that authenticates pairs of subtrees in a conventional Merkle tree construction with a vector commitment scheme [39] 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 [40] propose a post-quantum vector commitment scheme, but the size of its authentication path is on the same order as for a conventional Merkle tree. Buterin [38] suggests Scalable Transparent ARguments of Knowledge (STARKs) [41] 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 [42], 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 [43]. Goyal and Vaikuntanathan [44] 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 authenticating an evolving or "streaming" data series by Li et al. [45]. Papamanthou et al. propose an authenticated data structure for a streaming data series [46] 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. [47] define signature malleability in the limited sense we have adopted here. Chase et al. [48], building on work by Ahn et al. [49] and Attrapadung, Libert and Peters [50] 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 [51] 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 [52]. Their observations further motivate TLS protocol extensions where the server omits any certificates that the client already has. Sikeridis et al. [53] 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.

Benjamin [54] proposes the use of Merkle trees to batch the signing operations for multiple server signatures in a TLS handshake; such an optimization would decrease the server's computational overhead but would not necessarily reduce the communications cost, as both the Merkle tree authentication path and the underlying signature are sent to each verifier. Benjamin, O'Brien and Westerbaan [55] combine Merkle trees and Certificate Transparency concepts [12] into a "Merkle tree certificate" type, where signing operations for multiple certificates are batched using Merkle trees similarly to [54]. In [55], the authentication path is sent in the TLS handshake while the underlying signature is provided out of band via a transparency service. Aguilar-Melchor et al. [56] analyze and provide formal security proofs for these and other batch signature techniques.

Kudinov et al. [57] propose several techniques for reducing the size of SPHINCS$^+$ signatures, including an example with 20% savings. Baldimtsi et al. [58] describe a general framework for reducing the size of cryptographic outputs using brute-force "mining" techniques, estimating 5% to 12% savings. Such techniques are also complementary to condensation as they reduce the size of the underlying signature whereas condensation reduces the need to send full underlying signatures at all.

Internet-Drafts specifying MTL mode [59] and its use with DNSSEC [60] have recently been published by the authors of this paper.

10 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}, 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 [61]. 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.

Acknowledgments. We thank our Verisign colleagues for reviewing drafts of this paper and discussing its concepts, with particular appreciation to Duane Wessels for guidance on the selection of data sources for Section 7.3 and assistance with the data analysis. Thanks also to DNS-OARC for providing access to their data sets and servers. Finally, the paper would not have reached its final form without the improvements encouraged by the anonymous CT-RSA reviewers. We thank them for their generous commitment to the peer review process.

References

1. Post-Quantum Cryptography Standardization, NIST, https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization, last accessed 2024/01/02.
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.
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 2024/01/02.
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 2024/01/02.
6. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. National Institute of Standards and Technology, US Department of Commerce, initial public draft, dated 2023/08/24. https://doi.org/10.6028/NIST.FIPS.203.ipd.
7. FIPS 204: Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology, US Department of Commerce, initial public draft, dated 2023/08/24. https://doi.org/10.6028/NIST.FIPS.204.ipd.
8. FIPS 205: Stateless Hash-Based Digital Signature Standard. National Institute of Standards and Technology, US Department of Commerce, initial public draft, dated 2023/08/24. https://doi.org/10.6028/NIST.FIPS.205.ipd.
9. Wouters, P., Sury, O: RFC 8624, Algorithm Implementation Requirements and Usage Guidance for DNSSEC. IETF (2019). https://doi.org/10.17487/RFC8624.

10. 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.
11. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. IETF (2005). https://doi.org/10.17487/RFC4033.
12. Laurie, B., Messeri, E., Stradling, R.: RFC 9162: Certificate Transparency Version 2.0. IETF (2021). https://doi.org/10.17487/RFC9162.
13. Merkle, R.: Secrecy, Authentication, and Public Key Systems. Ph.D. thesis, Stanford University (1979). http://www.ralphmerkle.com/papers/Thesis1979.pdf, last accessed 2024/01/02.
14. 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.
15. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, C.M.., Chung, K.M., Persiano, G., Yang, B.Y. (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.
16. Fluhrer, S.: Further Analysis of a Proposed Hash-Based Signature Standard. In: Cryptology ePrint Archive, Paper 2017/553, https://eprint.iacr.org/2017/553, last accessed 2024/01/02.
17. Katz, J.: Analysis of a Proposed Hash-Based Signature Standard. In: Chen, L., McGrew, D., Mitchell, C. (eds) Security Standardisation Research. SSR 2016. LNCS, vol. 10074. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49100-4_12.
18. Grilo, A.B., Hövelmanns, K., Hülsing, A., Majenz, C.: Tight adaptive reprogramming in the QROM. In: Tibouchi, M., Wang, H. (eds) Advances in Cryptology — ASIACRYPT 2021, LNCS, vol. 13090, pp. 637–667. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92062-3_22.
19. Bos, J.W., Hülsing, A., Renes, J., van Vredendaal, C.: Rapidly Verifiable XMSS Signatures, Cryptology ePrint Archive, Paper 2020/898, https://eprint.iacr.org/2020/898, last accessed 2024/01/02.
20. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, NIST, https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, last accessed 2024/01/02.
21. Bai, S., Ducas, L., Kiltz, E., 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 2024/01/02.
22. 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 2024/01/02.
23. 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 2024/01/02.
24. McGrew, D., Curcio, M., Fluhrer, S.: RFC 8554, Leighton-Micali Hash-Based Signatures. IETF (2019). https://doi.org/10.17487/RFC8554.
25. Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: RFC8391, XMSS: eXtended Merkle Signature Scheme. IETF (2018). https://doi.org/10.17487/RFC8391.
26. Mockapetris, P.: RFC 1034, Domain Names - Concepts and Facilities. IETF (1987). https://doi.org/10.17487/RFC1034.
27. Day In The Life of the Internet Traces, DNS-OARC, https://www.dns-oarc.net/oarc/data/catalog, last accessed 2024/01/02.

28. 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.
29. Driscoll, F.: Terminology for Post-Quantum Traditional Hybrid Schemes, https://datatracker.ietf.org/doc/draft-ietf-pquip-pqt-hybrid-terminology/, last accessed 2024/01/02. Work in progress.
30. Bindel, N., Hale, B.: A Note on Hybrid Signature Schemes. In: Cryptology ePrint Archive, Paper 2023/423. https://eprint.iacr.org/2023/423, last accessed 2024/01/02.
31. Champine, L.: Streaming Merkle Proofs within Binary Numeral Trees. In: Cryptology ePrint Archive, Paper 2021/038. https://eprint.iacr.org/2021/038, last accessed 2024/01/02.
32. 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.
33. Todd, P.: Merkle Mountain Ranges, https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, last accessed 2024/01/02.
34. 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.
35. 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.
36. 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.
37. Kuszmaul, J.: *Verkle Trees*, https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf, last accessed 2024/01/02.
38. Buterik, V.: Verkle Trees, https://vitalik.ca/general/2021/06/18/verkle.html, dated 2022/06/18, last accessed 2023/02/13.
39. 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.
40. 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.
41. 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 2024/01/02.
42. 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.
43. Khaburzaniya, I., Chalkias, K., Lewi, K., Malvai, H.: Aggregating and thresholdizing hash-based signatures using STARKs. In: Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security, pp. 393–407. ACM, New York (2022). https://doi.org/10.1145/3488932.3524128.

44. Goyal, R., Vaikuntanathan, V.: Locally Verifiable Signature and Key Aggregation, In: , Dodis, Y., Shrimpton, T. (eds), Advances in Cryptology — CRYPTO 2022, LNCS, vol. 13508, pp. 761–791. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15979-4_26.
45. 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.
46. 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.
47. 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.
48. 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.
49. Ahn, J.H., Boneh, D., Camenisch, J., Hohenberger, S., Shelat, A., Waters, B.: Computing on authenticated data. *Journal of Cryptology* 28(2), 351–395 (2015). https://doi.org/10.1007/s00145-014-9182-0.
50. 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.
51. Decker, C., Wattenhofer, R.: Bitcoin transaction malleability and MtGox. In: Kutyłowski, M., Vaidya, J. (eds.) Computer Security — ESORICS 2014, LNCS, vol. 8713, pp. 313–326. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_18.
52. 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.
53. Sikeridis, D., Huntley, S., Ott, D., Devetsikiotis, M.: Intermediate certificate suppression in post-quantum TLS: An approximate membership querying approach, In: CoNEXT '22: Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies, pp. 35–42. ACM (2022). https://dl.acm.org/doi/abs/10.1145/3555050.3569127.
54. Benjamin, D.: Batch Signing for TLS, https://datatracker.ietf.org/doc/draft-davidben-tls-batch-signing/, last accessed 2024/01/02. Work in progress.
55. Benjamin, D., O'Brien, D., Westerbaan, B.: Merkle Tree Certificates for TLS, https://datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs, last accessed 2024/01/02. Work in progress.
56. Aguilar-Melchor, C., Albrecht, M.R., Bailleux, T., Bindel, N., Howe, J., Hülsing, A., Joseph, D., Manzano, M.: Batch Signatures, Revisited. In: Cryptology ePrint Archive, Paper 2023/492, https://eprint.iacr.org/2023/492, last accessed 2024/01/02.
57. 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 2024/01/02.

58. 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 2024/01/02.
59. Harvey, J., Kaliski, B., Fregly, A., Sheth, S.: Merkle Tree Ladder Mode (MTL) Signatures, https://datatracker.ietf.org/doc/draft-harvey-cfrg-mtl-mode/, last accessed 2024/01/02. Work in progress.
60. Fregly, A.M., Harvey, J., Kaliski, B., Wessels, D.: Stateless Hash-Based Signatures in Merkle Tree Ladder Mode (SLH-DSA-MTL) for DNSSEC, https://datatracker.ietf.org/doc/draft-fregly-dnsop-slh-dsa-mtl-dnssec, last accessed 2024/01/02. Work in progress.
61. 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, updated October 2022, last accessed 2023/12/11.
62. FIPS PUB 180-4: Secure Hash Standard. NIST (2015). https://doi.org/10.6028/NIST.FIPS.180-4.
63. 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 2024/01/02.
64. 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 2024/01/02.

Appendices

A Hash Function Instantiations

MTL mode uses three hash functions as noted in Section 2:

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

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 three 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 mode can use the same underlying hash function as the underlying signature scheme or a different hash function.

In the following, let H be a cryptographic hash function with security level at least ℓ (e.g., SHA-256 [62] for the case $\ell = 128$). We assume the hash function output is represented as an octet string as per the hash function's specification, e.g., Section 3 of [62]; $hLen$ denotes the length of the octet string (e.g., $hLen = 32$ for SHA-256). 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 [24]). We take a similar approach for MTL mode's uses and propose

$$H_{\text{msg}}(SID, i, m, c) := H(SID \parallel [i]_4 \parallel D_{\text{MTLM}} \parallel c \parallel m)\langle 1{:}\,\ell/8\,\rangle;$$
$$H_{\text{leaf}}(SID, i, d) := H(SID \parallel [i]_4 \parallel D_{\text{MTLL}} \parallel d)\langle 1{:}\,\ell/8\,\rangle; \text{ and}$$
$$H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) := H(SID \parallel [L]_4 \parallel D_{\text{MTLI}} \parallel [R]_4 \parallel V_{\text{left}} \parallel V_{\text{right}})\langle 1{:}\,\ell/8\,\rangle.$$

To align with HSS/LMS's formats, we place these constraints on our MTL mode implementation: SID must be a 16-octet string and i, L and R must be at most $2^{32} - 1$. We suggest $D_{\text{MTLM}} = 0\text{x}9090$, $D_{\text{MTLL}} = 0\text{x}9191$, and $D_{\text{MTLI}} = 0\text{x}9292$, 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 $\ell_c = 2\ell$ bits, the same length as HSS/LMS's own message randomizer (see Section 7.1 of [24]), even though a shorter ℓ_c is sufficient (see Section 5.3 above). When $\ell = 128$ and H is SHA-256, the input to the hash function in H_{leaf} is at most 38 octets long, which, after padding, fits within a single SHA-256 compression function call. H_{int} takes two calls, matching its counterpart in HSS/LMS.

With these instantiations, up to $2^{32} - 1$ messages can be associated with a given series identifier and up to $2^{160} - 2^{128}$ 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 [24]). Note that while our inputs generally follow HSS/LMS's formats, our outputs are half the size, $\ell/8$ vs. $hLen$ octets following the security proof in Section 5.

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 [25]). Following this approach, we propose

$$H_{\text{msg}}(SID, i, m, c) := H(D_{\text{MTLM}} \parallel SID \parallel [i]_4 \parallel c \parallel m)\langle 1{:}\,\ell/8\rangle;$$
$$H_{\text{leaf}}(SID, i, d) := H(D_{\text{MTLL}} \parallel SID \parallel [i]_4 \parallel d)\langle 1{:}\,\ell/8\rangle; \text{ and}$$
$$H_{\text{int}}(SID, L, R, V_{\text{left}}, V_{\text{right}}) := H(\,D_{\text{MTLI}} \parallel SID \parallel [L]_4 \parallel [R]_4 \parallel V_{\text{left}} \parallel V_{\text{right}})\langle 1{:}\,\ell/8\rangle.$$

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 and i, L and R to be at most $2^{32} - 1$. However, the instantiations could be redefined with different lengths. As above, with these constraints up to $2^{160} - 2^{128}$ 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 [25]).

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$^+$ (now being standardized as FIPS 205 [7]). 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 [23]). 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), given that SPHINCS$^+$ is stateless and therefore doesn't have a built-in limit on the number of messages that can be signed. A detailed "SPHINCS$^+$-friendly" instantiation along similar lines as described here is proposed in [59].

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 [22]). 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 (now being standardized as FIPS 204 [8]) also uses a hash function for several purposes (see Section 5.3 of [21]). Like FALCON, it doesn't directly provide a way to separate other uses from its own. 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 [15], where a design goal is to limit each of the adversary's hash function queries to a specific context.

B Binary Rung Strategy Operations

We now give example pseudocode for the Merkle tree ladder operations in the binary rung strategy described in Section 3, which is the basis for the MTL mode operations in Section 4. 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, 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 which 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 are the index pair 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 $(L, R) \to V$, 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 $(L, R) \to Z$ 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 [63]. 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 $2R - B(R)$ [64]. 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 $B(R)$ perfect, adjacent binary trees spanning the R leaf nodes. A perfect 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 hash values 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 $T.\Pi$. 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

INITNODESET$(SID) \longrightarrow T$ returns a new node set T associated with the series identifier SID.

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

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

B.3 Leaf Node Addition

ADDLEAFNODE$(T, d) \longrightarrow \langle \Lambda_N \rangle$ adds a leaf node corresponding to a data value d to the node set T and returns the current ladder Λ_N where N is the current leaf node count.

1. Set $SID := T.SID$.
2. Set $i := T.N + 1$.
3. Set $T.N := i$.

4. Compute $V := H_{\text{leaf}}(SID, i, d)$.
5. Set $T.V[i{:}\,i] := V$, adding the new leaf node to the node set.
6. Write $i = \sum_{j=1}^{B} 2^{\nu_j}$ where the ν_j are the indexes of the ones bits in the binary representation of i from highest to lowest.
7. For k from 1 to ν_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.
8. Create a new empty array Λ_N.
9. Set $R := 0$.
10. For j from 1 to B do:
    a. Set $L := R + 1$ and $R := R + 2^{\nu_j}$.
    b. Set $\Lambda_N[j] := T.V[L{:}\,R]$, adding this rung hash value to the ladder.
11. Return Λ_N, which will be an array of B hash values.

Step 7 computes the new ladder rung $[i - 2^{\nu_B} + 1{:}\,i]$ from leaf to ladder. As also noted in Appendix C, Step 7 computes this rung from the last ν_B rungs of Λ_{N-1}, so the rung is their ancestor. Step 10 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 7a selects the rung $T.\Lambda[B + \nu_B - k]$ from the current ladder. Instead of storing the new hash value, Step 7b copies the selected rung to the current authentication path by setting $T.\Pi[k] := T.\Lambda[B + \nu_B - k]$. A new step before Step 8 then copies the newly computed final rung to the current ladder by setting $T.\Lambda[B] := V$. In addition, instead of retrieving $T.V[L{:}\,R]$ from the set of node hash values, Step 10b sets $\Lambda_N[j] := T.\Lambda[j]$. Steps 5, 9 and 10a are omitted.

B.4 Authentication Path Construction

GETAUTHPATH$(T, i) \longrightarrow \Pi_{i,N}$ returns the authentication path $\Pi_{i,N}$ from the i^{th} leaf node in the node set T relative to the current ladder. The operation requires that $1 \leq i \leq N$.

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 perfect 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.5 Authentication Path Verification

CHECKAUTHPATH$(SID, i, N, N', d, \Pi_{i,N}, \Lambda_{N'}) \longrightarrow b$ verifies that the i^{th} leaf node corresponds to the data value d using an authentication path $\Pi_{i,N}$ from the i^{th} leaf node relative to the N^{th} ladder Λ_N, and the N'^{th} ladder $\Lambda_{N'}$. It returns $b =$ TRUE if the authentication path is valid and $b =$ FALSE otherwise. The operation requires that $1 \leq i \leq N' \leq N$.

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.

C Proof of Backward Compatibility Property

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

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

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 $\Lambda_{N'}$ for every N' such that $1 \leq N' \leq N$.

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. CHECKAUTHPATH can therefore verify $\Pi_{i,N}$ using $\Lambda_{N'}$. ■

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.

We consider three of the parameterizations given in Section 7 as examples:

- SPHINCS$^+$-128s
- HSS/LMS (ParmSet 25/15)
- XMSS^MT (SHA2_20/2_256)

All three parameterizations 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 would be roughly 86% of its initial (i.e., uncondensed) signature size. The HSS/LMS and XMSS^MT examples have two; their condensed signature sizes would be roughly 50% of their uncondensed signature sizes.

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

In the XMSS^MT example, the top-layer tree has 2^{10} 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 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 HMS/LMS example, the top-layer tree has 2^{25} leaf nodes, however, so the transition point is much higher. 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 may be lower in practice because not every top-layer leaf will necessarily be involved in the first K signatures, especially for HSS/LMS and XMSS^MT which exhaust each top-layer leaf node before moving to the next one. If we want to reduce further, faster, and for non-hash-based signature schemes, however, we need a different approach such as MTL mode.

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:

- *Initialization.* RECONSTINIT$(pk) \longrightarrow st$ returns a new reconstitution state st relative to the public key pk.
- *Condensed signature incorporation.* ADDCONDENSEDSIG$(st, \varsigma, \chi) \longrightarrow \langle i, b \rangle$ incorporates a condensed signature ς associated with a handle χ into the state st and returns the signature index i for this signature and a flag b indicating whether a new reference value is needed.
- *Reference value incorporation.* ADDREFVAL$(st, \chi, \upsilon) \longrightarrow st'$ incorporates a reference value υ associated with a handle χ into the state st.
- *Reconstituted signature production.* GETRECONSTSIG$(st, i) \longrightarrow \langle \sigma' \rangle$ reconstitutes the i^{th} signature in the state st and returns the reconstituted signature σ'.

By returning the flag, ADDCONDENSEDSIG automates the process of determining whether a suitable reference value is available mentioned in Section 6. 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).

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 backward compatibility property of the binary rung strategy. 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 backward 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 $\upsilon_{N_0^{(1)}}$, $\upsilon_{N_0^{(2)}}$, ..., 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 series identifier SID 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 also the unique integer with this property between i and N' and that $[R - 2^v + 1 : R]$ is 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 $[R - 2^{\nu} + 1 : R]$ 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^{ν} is the largest power of 2 dividing R, we have $\gamma_1 = 2^{\nu_B} = 2^{\nu}$.

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 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.