Network Working Group
Internet-Draft
Intended status: Informational
Expires: 17 September 2025

J. Harvey
B. Kaliski
A. Fregly
S. Sheth
Verisign Labs
16 March 2025

Merkle Tree Ladder (MTL) Mode Signatures
draft-harvey-cfrg-mtl-mode-06

Abstract

This document provides an interoperable specification for Merkle tree ladder (MTL) mode, a technique for using an underlying signature scheme to authenticate an evolving series of messages. MTL mode can reduce the signature scheme's operational impact. Rather than signing messages individually, the MTL mode of operation signs structures called "Merkle tree ladders" that are derived from the messages to be authenticated. Individual messages are then authenticated relative to the ladder using a Merkle tree authentication path and the ladder is authenticated using the public key of the underlying signature scheme. The size and computational cost of the underlying signatures are thereby amortized across multiple messages, reducing the scheme's operational impact. The reduction can be particularly beneficial when MTL mode is applied to a post-quantum signature scheme that has a large signature size or computational cost. As an example, the document shows how to use MTL mode with the Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) specified in the draft FIPS 205. Like other Merkle tree techniques, MTL mode's security is based only on cryptographic hash functions, so the mode is quantum-safe based on the quantum-resistance of its cryptographic hash functions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 September 2025.

Table of Contents

1. Introduction

This document provides an interoperable specification for Merkle tree ladder (MTL) mode [MTL-MODE], a technique for using a signature scheme to authenticate an evolving series of messages that potentially can reduce the operational impact of the signature scheme.

MTL mode is a different way of using an underlying signature scheme. Instead of signing individual messages directly, MTL mode signs structures called "Merkle tree ladders" that are derived from the messages to be authenticated. Individual messages are then authenticated relative to the ladder using a Merkle tree authentication path (also called a Merkle proof). The operational impact of the signatures on the ladders is thus amortized across multiple messages. The remaining per-message cost consists of the overhead of computing and using the ladders and authentication paths.

The operational benefits of MTL mode are most evident in scenarios where verifiers are interested in a subset of messages among a large, evolving series of messages. Examples include authenticating web Public-Key Infrastructure certificates [RFC5280], Domain Name System Security Extensions (DNSSEC) records [RFC4033] and signed certificate timestamps [RFC9162]. MTL mode is not well suited to scenarios where a verifier is interested in authenticating a single, newly generated message. An example is a Transport Layer Security transcript hash [RFC8446]. In such scenarios, a ladder would need to be signed and verified for every message processed, so the operational impact would not be reduced. Two drafts on applying MTL mode to applications are not available as of this writing.
[I-D.harvey-cfrg-mtl-mode-considerations] provides design considerations for application designers on how to add Merkle Tree Ladder (MTL) Mode signatures into their applications.
[I-D.fregly-dnsop-slh-dsa-mtl-dnssec] describes how to apply SLH-DSA to DNSSEC. Additional drafts may be developed for other applications.

The mode is intended primarily for use with post-quantum signature schemes where the reduction of operational impact can be significant given their relatively large signature sizes. As an initial example, this document shows how to use MTL mode with SLH-DSA, the Stateless Hash-Based Digital Signature Algorithm selected by NIST for standardization of part of its post-quantum cryptography project [FIPS205]. (SLH-DSA is the standardized version of SPHINCS+ [SPHINCS-PLUS].) The design decision is motivated by three considerations: (1) SLH-DSA also is based on Merkle trees, and thus already has internal functions for computing leaf nodes and internal nodes; (2) SLH-DSA has a relatively large signature size and

computational cost, and therefore can benefit significantly from the reductions offered by MTL mode; and (3) hash-based techniques are well understood and offer a conservative choice for long-term security, alongside newer techniques from other families such as lattice-based cryptography. Future updates to this document may support other signature schemes.

1.1. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Preliminaries

2.1. Definitions

Node Set A set of nodes, each of which is part of a union of tree structures either as a leaf node whose value is the hash of a single data value, or an internal node whose value is the hash of two child nodes. The node set is acyclic, i.e., every node is either a leaf node or the ancestor of two or more leaf nodes, and no node is an ancestor of itself. Every node set has one or more root nodes, i.e., nodes that have no ancestors.

Rung A node from a node set that can be used to authenticate one or more leaf nodes within that node set.

Ladder A collection of one or more rungs that can be used to verify an authentication path.

Authentication Path The set of sibling hash values from a leaf hash value to a rung

Message A set of bytes that are intended to be signed and later verified.

2.2. Operators

Standard order of operations is assumed throughout this document.

The following mathematical operators are used:

** : a ** b denotes the value of a raised to the power of b
* : a * b denotes the product of a multiplied by b
/ : a / b denotes the quotient of a divided by b
+ : a + b denotes the sum of a and b when a and b are numbers
- : a - b denotes the difference of a and b
= : a = b denotes assigning the value of b to a

The following bitwise operators are used:

& : a & b denotes the bitwise AND of the unsigned integers a and b
| : a | b denotes the bitwise OR of the unsigned integers a and b
~ : ~a denotes the bitwise NOT of the unsigned integer a
>> : a >> i denotes a right bit shift (non-rotating) of a by i bit positions to the right.
<< : a << i denotes a left bit shift (non-rotating) of a by i bit positions to the left.

The following comparison operators are used:

== : a == b denotes the comparison between a and b to see if the two values are equal
<= : a <= b denotes the comparison between a and b to see if a is less than or equal to b
>= : a >= b denotes the comparison between a and b to see if a is greater than or equal to b
!= : a != b denotes the comparison between a and b to see if a is not equal to b

The following array notation is used:

The notation A[i] represents the ith element of array A.

The following byte string notation is used:

+ : a + b denotes the concatenation of values a and b when a and b are byte strings.

2.3. Functions

Given an unsigned integer x or a message byte string a, the following functions are defined:

lsb(x) returns the position of the least significant bit of x, where bit positions start at 1 and lsb(0) = 0.

msb(x) returns the position of the most significant bit of x.

bit_width (x) returns the number of 1 value bits in x. This corresponds to the popcnt instruction on Intel/AMD processors and the __builtin_popcount function in gcc.

octet(x) returns a single byte with value x (assuming 0 <= x <= 255).

OLEN(a) returns the length in bytes of a.

Example Function Outputs:

x	representation	lsb	msb	bit_width
0	00000000	0	0	0
1	00000001	1	1	1
2	00000010	2	2	1
3	00000011	1	2	2
4	00000100	3	3	1
5	00000101	1	3	2
6	00000110	2	3	2
7	00000111	1	3	3

2.4. Algorithm Style

The data structures and algorithms defined in this document are written to be runnable Python 3 code. The following styles have been applied to further make the code easy to read and follow:

* Classes and data structures are written in all uppercase (e.g. MTLNS)
* Constant values are also written as all uppercase with _ separating words (e.g. MTL_SIG_CONDENSED)
* Variables are written in all lowercase with _ separating words as needed (e.g. left_hash or tree)
* Functions are written in all lower case and proceeded by a comment block that highlights the input and output parameters.

3. General Model

The general model for MTL mode involves the following exchanges between a signer and a verifier. The signer is assumed to have a private key for an underlying signature scheme and the verifier is assumed to have the corresponding public key.

1. The signer maintains a dynamic data structure called a Merkle node set. The leaf nodes of the node set correspond to the messages that are being "signed" for later authentication and the internal nodes of the node sets are the hashes of two child nodes. Similar to a Merkle tree structure, ancestors authenticate or "cover" their descendants. A Merkle node set is more general than a Merkle tree in that more than one node can be a root node (i.e., a node without ancestors). For instance, a Merkle node set could include multiple trees.
2. As the node set evolves, the signer occasionally selects a set of nodes from the node set that collectively cover all the leaf nodes. Such a set is called a "ladder" and each node within the

set is called a "rung." The rungs are selected according to a "binary rung strategy" where each rung is the root of a perfect binary tree (see Section 6.5).

3. The signer signs each ladder with the private key of the underlying signature scheme. The signature on the ladder is the "MTL mode signature" of the set of messages covered by the ladder.
4. For each message of interest to a verifier, the signer provides the verifier a Merkle authentication path from the leaf node corresponding to the message to a rung in the then-current ladder. Similar to a Merkle tree structure, the authentication path includes the sibling hashes on the path from the leaf node to a rung on the ladder that covers the leaf node.
5. If the verifier has a ladder that is "compatible" with the authentication path, the verifier verifies the authentication path on the message relative to the ladder.
6. If not, the verifier requests the signed ladder that the authentication path was computed relative to. (Alternatively, the verifier may request a "full" signature on the message that includes both the authentication path and the signed ladder that it is computed relative to, which could be the current ladder. See Section 9.4 for a description of a full signature.)
7. The signer provides the signed ladder. (Or, alternatively, the signer provides a full signature including the authentication path together with a signed ladder.)
8. The verifier verifies the signature on the signed ladder and returns to Step 5.

The model can reduce the operational impact of the underlying signature scheme in two main ways. First, per Steps 2 and 3, the signer signs ladders only as needed, not necessarily every time a message is added to a message series. The signer thus potentially makes many fewer calls to the underlying signature generation operation and stores fewer signatures than if the messages were signed individually. Second, per Steps 6, 7 and 8, the verifier obtains and verifies signatures on ladders only as needed, not necessarily every time a message is authenticated. The signer thus potentially sends fewer signatures, and the verifier stores and verifies fewer signatures, than if the messages were signed individually.

4. Security Parameters, Cryptographic Functions and Address Scheme

MTL mode signatures are dependent on an underlying signature scheme, requiring proper domain separation and alignment of the security parameters and functions.

MTL mode, like pre-hashing, requires domain separation from the underlying signature scheme. Methods for MTL mode domain separation are defined in Section 4.1, following a scheme that is similar to what is specified by NIST in FIPS 205.

Because SLH-DSA is the initial recommended underlying signature scheme for MTL mode, this document specifies MTL mode using the SLH-DSA security parameter and abstract cryptographic functions that are substantially the same as the ones in [FIPS205]. The document also uses an extended version of the [FIPS205] address scheme with additional address types. One goal of this approach is to make it easier for developers who already have a SLH-DSA implementation to build MTL mode operations. Another goal is to makes it easier to ensure that MTL mode operations are cryptographically separated from SLH-DSA’s internal operations.

The cryptographic parameter is defined in Section 4.2. Domain separation and pre-hashing as they relate to MTL mode are discussed in Section 4.1. The cryptographic functions are defined in Section 4.3, Section 4.4 and Section 4.5. The address scheme is defined in Section 4.6. Finally, cryptographic separation and message index association are discussed in Section 4.7.

In an implementation, the parameter needs to be instantiated with an actual value and the abstract functions need to be instantiated with actual functions. Recommended instantiations when the underlying signature scheme is SLH-DSA are given in Section 10. Recommended instantiations for other underlying signature schemes may be added in updates to this document.

In the following, notation || indicates concatenation of byte strings for consistency with SLH-DSA, in contrast to the + notation used for byte string concatenation elsewhere in the document.

4.1. Domain separation and modes of operation

[FIPS205] specifies two ways to use SLH-DSA: "pure," where a message is input directly to the underlying signature scheme, and "pre-hash," where a hash of the message is the input. Both approaches also take an optional application-specified context string as input. MTL mode can be modeled as a third "mode of operation," where a ladder derived from one or more messages is the input. Expanding on [FIPS205], the input to the underlying signature scheme in this version of the document is formatted as follows:

```
octet(MTL_LADDER_SEP) || octet(OLEN(ctx)) ||
ctx || OID_MTL || ladder
```

where:

* MTL_LADDER_SEP is the integer 129 (a new domain separator value that provides easy visual separation in hexadecimal from the identifiers 0 and 1 in NIST's proposal)
* ctx is an application-specified context string at most 255 octets (bytes) long (default is the empty string)
* OID_MTL is the DER encoding of the object identifier of the selected instantiation of MTL mode (see Section 10 for the list of instantiations; the object identifiers are TBD)
* ladder is the Merkle tree ladder being signed or verified

The format above separates MTL mode inputs from pure inputs to the underlying signature scheme (which would have a first byte of 0) and from other pre-hash inputs (which would have a first byte of 1). The concatenation of the first four strings in the format (i.e., the strings preceding the ladder) is referred to as the "MTL ladder domain separator."

For domain separation between calls made by MTL mode operations and those made by the underlying signature scheme (in the case of SLH-DSA; see Section 4.7), the inputs to H_msg_mtl and PRF_msg in calls made by MTL mode operations are formatted as follows:

```
octet(MTL_MSG_SEP) || octet(OLEN(ctx)) || ctx || value
```

where:

* MTL_MSG_SEP is the integer 128 (a counterpart to MTL_LADDER_SEP's 129 above)
* ctx is an application-specified context string at most 255 octets (bytes) long (default is the empty string)
* value is the rest of the value being input to the function

The concatenation of the first three strings in the format (i.e., the strings preceding the value) is referred to as the "MTL message domain separator."

FOR DISCUSSION: Expanding on the pure and pre-hash domain separator formats is intended to make it easier for implementations of FIPS 205 to be extended to support MTL mode in a way that separates MTL mode from the pure and pre-hash modes. However, the approach introduces a new first byte value and new OIDs that implementations might not initially recognize. An alternative is to reuse the proposed pre-hash domain separator format and consider MTL mode itself as a more complex form of pre-hashing. Instead of a new byte, the alternative only introduces new OIDs. However, implementations might only expect OIDs that are associated with NIST-approved hash functions, not MTL

mode. Pre-hashing remains an active discussion topic on the NIST PQC mailing list as of this writing, and further updates may be made to this document as is appropriate.

4.2. Security parameter

MTL mode has one security parameter, the size in bytes of hash values, denoted n. The security parameter SHOULD be at least 16. Typical values of n are 16, 24 and 32 (see Section 10). The security parameter affects the difficulty of breaking MTL mode (see Section 15).

4.3. Randomized message digest function (H_msg_mtl)

MTL mode makes use of a variant of the randomized message digest function (i.e., keyed hash function) H_msg defined in SLH-DSA:

* H_msg_mtl(R_mtl, PK.seed, PK.root, M) -> md maps a n-byte randomizer, a n-byte public seed, a n-byte public root and a variable-length message to a n-byte hash value md.

H_msg_mtl differs from SLH-DSA's H_msg in that its output hash value is n bytes long rather than m bytes long (where m is a separate parameter in SLH-DSA). The inputs to H_msg_mtl have the following meanings:

* R_mtl is the randomizer for the message digest operation
* PK.seed the public seed from the public key
* PK.root is the public root from the public key
* M is the message being processed.

H_msg_mtl is used for computing data values from messages in MTL mode (see Section 5.1 and Section 5.2). (Note that when H_msg_mtl is called in these sections, M is the message being signed prepended with a MTL message domain separator - see Section 4.7 for discussion.)

4.4. Pseudorandom function (PRF_msg)

MTL mode also makes use of pseudorandom function PRF_msg defined in SLH-DSA:

* PRF_msg(SK.prf, opt_rand, M) -> R_mtl maps a n-byte secret PRF key, a n-byte optional random value, and a variable-length message to a n-byte randomizer R_mtl.

The inputs to PRF_msg have the following meanings:

* SK.prf is the secret PRF key from the private key.
* opt_rand depends on whether an implementation wants deterministic signing. If it does, then opt_rand SHOULD be a fixed value, e.g., all 0s or PK.seed. If not, then opt_rand MUST be a randomly generated value.
* M is the message being processed.

PRF_msg is used for computing randomizers for hashing messages in MTL mode (see Section 5.1 and Section 5.2). (Note that when PRF_msg is called in these sections, M is an ADRS value prepended with a domain separator, optionally followed by the message being signed - see Section 4.7 for discussion.)

4.5. Tweakable hash functions (F and H)

MTL mode makes use of the tweakable hash functions F and H defined in SLH-DSA:

* F(PK.seed, ADRS, M_1) -> md maps a n-byte public seed, a 32-byte address value and a n-byte message value to a n-byte hash value
* H(PK.seed, ADRS, M_1 || M_2) -> md maps a n-byte public seed, a 32-byte address value and the concatenation of two n-byte message values to a n-byte hash value

The inputs to F and H have the following meanings:

* PK.seed is the public seed from the public key. This value is a "tweak" to the hash function that separates uses of the function with different public keys (assuming different public keys have different public seeds, as they almost always will if the public seeds are generated at random).
* ADRS is the address associated with the call to the function. This value is another "tweak" that separates uses of the function for different purposes.
* M_1 (input to F) is a n-byte value being hashed.
* M_1 || M_2 (input to H) is the concatenation of two n-byte values being hashed together.

F is used for computing a leaf node from a data value in MTL mode (see Section 8.2.1). H is used for computing an internal node hash value from two child node hash values (see Section 8.2.2).

4.6. Address scheme

MTL mode extends the address scheme for function inputs defined in SLH-DSA, adding four address types.

As in SLH-DSA, the address is an eight-word (32-byte) value. The fifth word (byte positions 17-20) is the address type.

This document assigns identifiers 16-18 for new address types. The assignment provides easy visual separation in hexadecimal from the identifiers 0-6 used by SLH-DSA. The constants MTL_MSG, MTL_DATA, and MTL_TREE provide a descriptive alternative to the numbers.

For all three types, the first and second words MUST be 0 and the third and fourth words MUST be the series identifier SID associated with the MTL mode operation, padded on the left. The sixth, seventh and eighth words depend on the address type. Index values are represented as 4-byte unsigned integers in big endian notation.

* MTL Message Hash (type MTL_MSG = 16). This type is used in the address value that is prepended to a message when calling PRF_msg to compute a randomizer or when calling H_msg_mtl to compute a data value from a message. For this type, the sixth and seventh words MUST be 0 and the eighth word MUST be the message index (i.e., the leaf index of the corresponding leaf node).
* MTL Data Value (type MTL_DATA = 17). This type is used when calling F to compute a leaf node hash value from a data value. For this type, the sixth and seventh words MUST be 0 and the eighth word MUST be the leaf index.
* MTL Tree (type MTL_TREE = 18). This type is used when calling H to compute an internal node hash value from two child node hash values. For this type, the sixth word MUST be 0, the seventh word MUST be the left index associated with the internal node and the eighth word MUST be the right index associated with the internal node.

	Byte Positions						
Address Type	1-4	5-8	9-16	17-20	21-24	25-28	29-32
MTL Message Hash	0	0	SID	16	0	0	MsgID
MTL Data Value	0	0	SID	17	0	0	MsgID
MTL Tree	0	0	SID	18	0	Left	Right

Address values are represented with the ADRS class.

4.7. Cryptographic separation and message index association for H_msg_mtl and PRF_msg

The security proof for MTL mode in [MTL-MODE] assumes that calls to the function for computing data values from messages, i.e., to H_msg_mtl here, are cryptographically separated from calls made by SLH-DSA's internal operations. In addition, the security proof assumes that calls to this function also include the message index as input.

For the tweakable hash functions in SLH-DSA, cryptographic separation and message index association are achieved by taking an address value as input. However, H_msg in SLH-DSA doesn't take an address value as input, and for consistency, neither does H_msg_mtl.

This document takes the following approach to achieve cryptographic separation and message index association for calls to H_msg and H_msg_mtl:

* When calling H_msg_mtl to compute a data value from a message, see Section 5.1 and Section 5.2, a MTL message domain separator and an address value of type MTL_MSG are prepended to the message, where the address value includes the message index
* When calling the underlying signature scheme to sign a ladder or verify a signature on a ladder (see Section 4.1), a MTL ladder domain separator is prepended to the ladder

This document takes a comparable approach to achieve cryptographic separation and message index association for calls to PRF_msg. (Note that calls to PRF_msg from the MTL mode operations only take the domain separator and the address value in the input, not the message.)

Assuming that the underlying signature scheme passes the message to be signed directly to H_msg, as SLH-DSA does, the calls to H_msg_mtl from the MTL mode operations and to H_msg from SLH-DSA will involve values of M whose first bytes differ, providing cryptographic separation between calls by the MTL mode operations and by SLH-DSA. A comparable observation applies to calls to PRF_msg.

5. Computing Data Values from Messages

In MTL mode, variable-length messages are converted to fixed-length data values that can be processed by the MTL node set operations in the next section.

The conversion process involves a randomized message digest operation. The signer computes the randomizer and sends it to the verifier along with other information needed to authenticate the message.

The computation of the randomizer varies depending on whether the signer selects deterministic hashing or randomized hashing. (The choice follows a similar approach to SLH-DSA.)

5.1. Signer Operations

A MTL mode signer starts with a message M, a context string ctx, a series identifier SID, and a message index MsgID and computes a randomizer R_mtl and a data value with the following steps.

* Form an address value ADRS of type MTL_MSG from the series identifier and the message index as described in Section 4.6.
* Format a MTL message domain separator sep from the context string as follows:

sep = octet(MTL_MSG_SEP) || octet(OLEN(ctx)) || ctx

* With deterministic hashing, let opt_rand be the public seed PK.seed
* Generate a randomizer R_mtl using a secure random generator [RFC4086]. An implementation MAY use the following steps:
  - With randomized hashing, let opt_rand be a random n-byte value
  - Compute a randomizer by applying PRF_msg to a secret PRF key, SK.prf, the optional random value and the address value prepended with the domain separator

R_mtl = PRF_msg(SK.prf, opt_rand, sep || ADRS)

An implementation MAY either use the same or a different SK.prf value as is used in the underlying signature scheme. In either case, the SK.prf value MUST have been generated using a secure random generator and MUST be kept secret, e.g., as part of the private key. Using the same SK.prf value avoids the need to generate and manage an additional secret. However, using a different SK.prf value can be advantageous if the underlying signature scheme is implemented in a security module and the MTL mode operations are implemented outside the security module and do not have access to the underlying signature schemes SK.prf value.

An implementation MAY additionally include the message M in the input to the PRF_msg call by appending it to ADRS, i.e., by passing sep || ADRS || M as the third argument. However, this approach involves an additional pass over the message, which can be disadvantageous if the message is long.

* Compute the data value by applying H_msg_mtl to the randomizer, the public seed, the public root and the message prepended with the same address value

```
data_value = H_msg_mtl(R_mtl, PK.seed, PK.root, sep || ADRS || M)
```

5.2. Verifier Operations

A MTL mode verifier starts with a message M, a randomizer R_mtl, a context string ctx, a series identifier SID and a message index MsgID and recomputes the data value with the following steps.

* Form an address value ADRS of type MTL_MSG from the series identifier and the message index as described in Section 4.6.
* Format a MTL message domain separator sep from the context string as follows:

```
sep = octet(MTL_MSG_SEP) || octet(OLEN(ctx)) || ctx
```

* Compute the data value by applying H_msg_mtl to the randomizer, the public seed, the public root and the message prepended with the domain separator and the address value:

```
data_value = H_msg_mtl(R_mtl, PK.seed, PK.root, sep || ADRS || M)
```

6. MTL Node Sets

The core functionality that enables MTL mode is a set of hash-based nodes organized in an expanding tree-like structure. This allows for appending data values to an expanding data series, computing ladders and computing authentication paths from data values to ladder rungs. MTL node set operations provide the building blocks for authenticating messages when a signature scheme is operated in in MTL mode.

A MTL node set authenticates a series of data values. Each data value in the series has a unique index, a non-negative integer. In the MTL mode operations in this document, the index starts at 0 and is incremented by 1 after each new data value is appended. A data value is computed from a message to be authenticated via a randomized message digest operation, as described in the previous section.

Three data structures supporting MTL node sets are given in Section 7 and six MTL node set operations are given in Section 8. This section provides a general overview of the concepts behind those techniques.

6.1. Seeds and Series Identifiers

The hash operations in the MTL node set operations take a public seed and a series identifier input. The public seed separates the use of the hash operations with different public keys (assuming different public keys have different public seeds). The series identifier separates the use of the hash operations for different series for data values with the same public key.

Both the public seed and the series identifier are n-byte values where n is the security parameter.

6.2. Node Sets

A MTL node set has zero or more nodes each of the form <L,R,V> where:

* L is the node’s left index, a non-negative integer
* R is the node’s right index, a non-negative integer
* V is the node’s hash value

The pair (L,R) is the node’s index pair. A node set MUST NOT have more than one node with a given index pair.

The node with index pair (L,R) authenticates the data values with indexes between L and R inclusive. If L = R, the node is a leaf node (Section 6.3). If L < R, then it is an internal node (Section 6.4).

6.3. Leaf Nodes

A leaf node is a node where L = R. It has no child nodes. Its hash value is computed as

V = hash_leaf(seed, sid, L, data_value)

where hash_leaf is a hash function defined in Section 8.2.1, seed and sid are the public seed and series identifier and data_value is the data value associated with this leaf index.

Including the leaf index as an input to hash_leaf separates the use of the hash function for different leaf nodes.

A leaf node with a given index authenticates the data value with the corresponding index. It follows that if a node set has a leaf node with a given index, then the data series MUST have a data value with the same index.

6.4. Internal Nodes

An internal node is a node where L < R.

An internal node has two child nodes, called a left child and a right child. Its hash value is computed as

V = hash_int(seed, sid, L, R, left_hash, right_hash)

where hash_int is a hash function defined in Section 8.2.2, seed and sid are the public seed and the series identifier, left_hash is the left child's hash value and right_hash is the right child's hash value.

Including the left and right indexes as inputs to hash_int separates the use of the hash function for different internal nodes.

The left and right children are the nodes with index pairs (L,M-1) and (M,R) respectively where M, the middle index, is the unique integer between L+1 and R that is divisible by the largest power of two. The child index ranges are thus a partition of the internal node's index range. The middle index can be computed as follows:

```
power = msb(R-L)
M = R - mod(R, 2^(power+1))
if(M <= L):
    M = R - mod(R, 2^power)
```

An internal node with index pair (L,R) authenticates the data values with indexes between L and R inclusive. It follows that if a node set has an internal node with an index pair (L,R), then the data series MUST have data values with indexes L through R. In addition, the node set MUST have nodes with index pairs (L,M-1) and (M,R).

The following table gives examples of index pairs for internal nodes and their left and right child nodes. In the table, a leaf node is denoted with a single index. For instance, the index 4 is equivalent to the index pair (4,4).

Internal Node (L,R)	Left Child (L,M-1)	Right Child (M,R)
(0,3)	(0,1)	(2,3)
(4,5)	4	5
(0,5)	(0,3)	(4,5)
(0,31)	(0,15)	(16,31)
(0,2)	(0,1)	2

6.5. Ladders

A ladder is a subset of nodes that is used to authenticate data values. Each node in the ladder is called a rung.

In the MTL mode operations in this document, the subset is selected according to what is called the binary rung strategy. In this strategy, the index pairs for the rungs are based on the binary representation of the number of data values in the data series.

The rungs in the binary rung strategy are selected as follows. Let N be the number of data values in the data series, let B be the number of 1s bits in the binary representation of N. N can then be represented as the sum of B distinct binary powers.

$$N = 2^{v_1} + 2^{v_2} + \ldots + 2^{(v_B)}$$

where $v_1 > v_2 > \ldots > v_B$ are the bit positions of the 1s bits in the binary representation. The first rung has index pair $(0,2^{v_1}-1)$; it is the apex of a perfect binary tree of height v_1 and authenticates the first 2^{v_1} data values. The next rung has index pair $(2^{(v_1,2)}v_1+2^{v_2}-1)$; it is the apex of a perfect binary tree of height v_2 and authenticates the next 2^{v_2} data values. The process continues until the B-th rung, which has index pair $(N-2^{v_B},N-1)$ and authenticates the last 2^{v_B} data values.

A rung is said to cover the data values it authenticates, and a ladder is said to cover the data values that its rungs collectively authenticate. The ladder just described thus covers all N data values in the node set.

(Another way of visualizing the rungs is to consider the first rung as the apex of the largest perfect binary tree that can be formed from the data values, starting from the left; the second rung as the apex of the largest perfect binary tree than can be formed over the remaining data values; and so on. The sizes of the trees decrease from left to right.)

(The binary rung strategy can be contrasted with the typical "single-rung strategy" employed with Merkle trees, where a single rung of a node set is used to authenticate data values, i.e., the root node (0,N-1). When N is a power of 2, the single-rung strategy and the binary-rung strategy are the same.

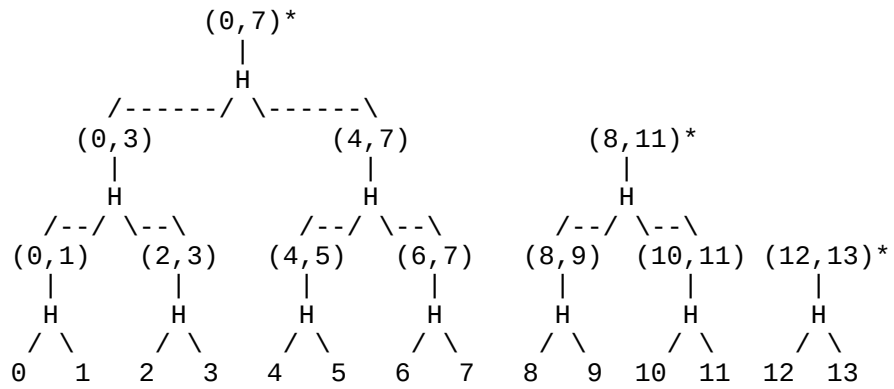
When the N-th data value is added to the node set, v_B+1 new nodes need to be computed to update the ladder: the leaf node with index (N-1,N-1) and the v_B internal nodes leading from the leaf node to the new ladder rung (N-2^v_B,N-1). The first B-1 ladder rungs in the new ladder are the same as in the previous ladder. Because 2^v_B is at most N, the number of new nodes computed is logarithmic in N, similar to ordinary Merkle tree constructions. Moreover, every node computed in the process is the apex of a perfect binary tree.

The minimum number of rungs in a ladder computed with the binary rung strategy is 1, in the case that the number of leaf nodes N is a power of 2. The maximum number is the least integer greater than or equal to log2(N), or equivalently ceiling(log2(n)). The actual number is bit_width(N).

The following table gives examples of ladders for values of N up to 14. As in the previous table, a leaf node is designated with a single index.

Number of Data Values N	Ladder Rungs
1	0
2	(0,1)
3	(0,1) 2
4	(0,3)
5	(0,3) 4
6	(0,3) (4,5)
7	(0,3) (4,5) 6
8	(0,7)
9	(0,7) 8
10	(0,7) (8,9)
11	(0,7) (8,9) 10
12	(0,7) (8,11)
13	(0,7) (8,11) 12
14	(0,7) (8,11) (12,13)
15	(0,7) (8,11) (12,13) 14
16	(0,15)
17	(0,15) 16
18	(0,15) (16,17)
19	(0,15) (16,17) 18

The following figure shows a node set with 14 nodes where the rungs are computed according to the binary rung strategy. The internal node hash function is denoted H and the leaf node hash function is not shown. The rungs are marked with asterisks (*).



6.6. Authentication Paths

An authentication path is the set of sibling node hash values encountered along the path from a leaf node to a target rung that covers a data value.

Target rungs can be any of the successive ancestors of the leaf node in the node set. Because each rung is the apex of a perfect binary tree, the sibling nodes encountered follow the structure of the binary tree.

For example, in the figure above, the sibling nodes encountered in the path from leaf node 6 to the rung (0,7) are the nodes with indexes 7, (4,5) and (0,3). The authentication path for the data value with index 6 includes the hash values at those nodes. This data value can be authenticated by recomputing leaf node 6 from the data value using hash_leaf, recomputing internal nodes (6,7), (4,7) and (0,7) from the sibling node hash values and previously computed hash values using hash_int, and then comparing the result to the rung hash value.

The minimum number of sibling nodes in an authentication path computed with the binary rung strategy is 0, in the case that the leaf node is the last leaf added and the number of leaf nodes N is odd. The maximum number is the greatest integer less than or equal to log2(N) , or equivalently floor(log2(N)). The actual number is bit_width(R-L) where (L,R) is the index pair of the rung covering the leaf node.

6.7. Backward Compatibility

An authentication path is initially computed relative to the current ladder in the MTL mode operations in this document. The target rung for the authentication path is thus the unique rung in the ladder that covers the data value to be authenticated. When an authentication path is verified, however, it can be verified relative to any of the successive ancestors of the leaf node corresponding to the data value up to and including the target rung.

Continuing the example above, the authentication path for the data value at index 6 can be verified relative to any ladder that includes a rung with index 6, (6,7), (4,7) and/or (0,7). In this case, the ladder covering the first six data values could also be used, because it includes a rung with index 6.

More generally, if an authentication path for the data value at leaf_index is computed relative to a ladder that covers the first N data values, the authentication path can also be authenticated relative to any binary rung strategy ladder that covers the first N' data values if the following condition is met:

leaf_index <= N' <= N.

The first inequality ensures that the ladder covers the data value; the second ensures that the authentication path can reach the ladder.

This property of "backward compatibility" with previous ladders is attractive because it provides a way for a verifier to use an old ladder to authenticate a new authentication path, thereby potentially reducing the number of times that the verifier needs to get a new ladder.

To facilitate this property, the following "compatibility check" can be applied. Let (L,R) be a rung in a ladder and let leaf_index be the index of the data value. The rung can be used to authenticate the data value if the following conditions hold:

* L <= leaf_index <= R, ensuring the leaf index is covered by the rung
* (L = 0 or degree <= lsb(L)-1) and R-L+1 = 2^degree, where degree = lsb(R-L+1)-1, ensuring the rung is indeed an apex of a perfect binary tree in the binary rung strategy
* lsb(R-L+1) is less than or equal to the number of sibling hash values in the authentication path, ensuring the authentication path can reach the rung

If a ladder has a rung that passes this check, then the ladder is compatible with the authentication path. If not, then the verifier needs to get a new ladder.

7. Data Structures

MTL mode operations use three well-defined data structures to represent elements described in the previous section. These structures are byte strings with number values represented in big endian notation. The data structures provide interoperability so that a verifier on one platform can read and verify the data provided by a signer on another platform.

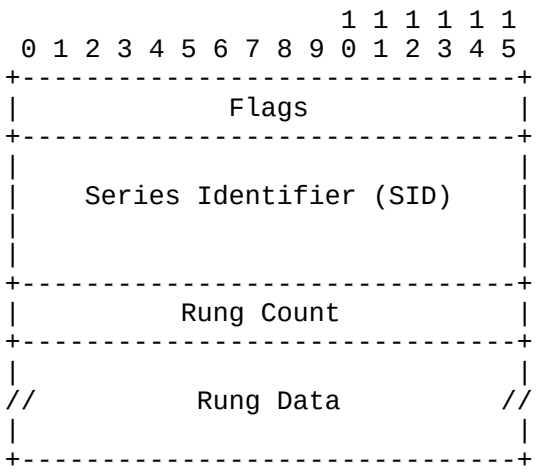
7.1. Ladder

A ladder data structure consists of four base components:

* flags, a 2-byte string providing future extensibility; the initial value for this field MUST be 0
* sid, the series identifier, a 8-byte string
* rung_count, the number of rungs in the ladder, a positive integer between 1 and 2^16-1
* rungs, one or more rung data structures

The rung data structure is described in the next section.

The byte representation of the ladder is the concatenation of the binary encodings of the fields using big endian representation of the integers:

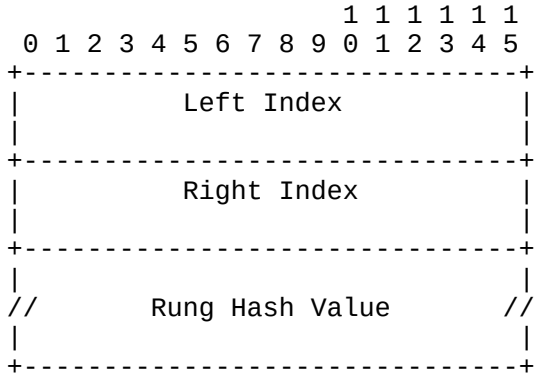


7.2. Rung

A rung data structure consists of three base components:

* left_index, the left index of the rung, a non-negative integer
* right_index, the right index of the rung, a non-negative integer
* hash, the rung hash value, a n-byte string

The byte representation of the rung is the concatenation of the binary encodings of the fields using big endian representation of the integers:

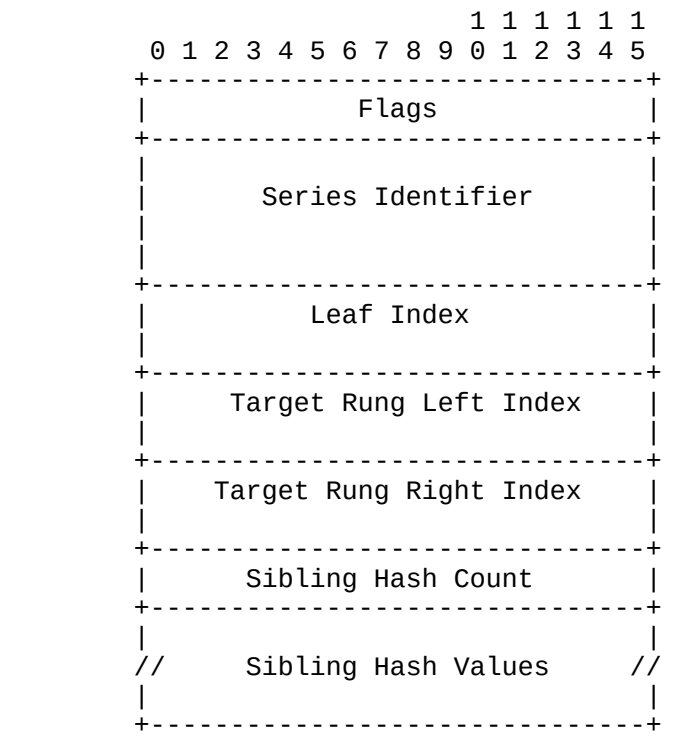


7.3. Authentication Path

An authentication path data structure consists of seven base components:

* flags, a 2-byte string providing future extensibility; it MUST be 0 for this version of the document
* sid, the series identifier, a 8-byte string
* leaf_index, the leaf index of the data value being authenticated, a non-negative integer
* sibling_hash_count, the number of sibling hash values in the authentication path, a non-negative integer between 0 and 2^16-1
* sibling_hashes, zero or more sibling hash values, each a n-byte string
* rung_left, the left index of the target rung, a non-negative integer
* rung_right, the right index of the target rung, a non-negative integer

The byte representation of the authentication path is the concatenation of the binary encodings of the fields using a 2-byte big endian representation for the node count and 4-byte big endian representations for the indexes:



8. MTL Node Set Operations

This section defines six operations supporting the use of MTL node sets to authenticate messages.

The first four, mtl_initns, mtl_append, mtl_ladder and mtl_authpath can be used by a signer to initialize a node set, add data values to it, obtain the current ladder, and obtain an authentication path relative to the current ladder. Each uses a MTL node set object to maintain the state of the node set.

The other two, mtl_rung and mtl_verify, can be used by a verifier to select a ladder rung that can be used to authenticate a data value and to authenticate the data value relative to the rung.

For the MTL mode operations in this version of the document, the following constraints apply:

* the public seed MUST be a n-byte string
* the series identifier MUST be a 8-byte string
* the various node indexes (leaf_index, left_index, right_index, etc.) MUST be non-negative integers between 0 and 2^32-1 (so they can be represented as 4-byte strings in big endian notation)
* the data value MUST be a n-byte string
* the various hash values (leaf_hash, left_hash, right_hash, internal_hash, etc.) MUST be a n-byte strings

8.1. MTL Node Set Object

MTL node set objects consist of four base components: a public seed, a series identifier, a leaf node count and a dynamic array of node hash values.

Consistent with the definition of a node set in Section 6.2, the array is indexed by two values. The hash value for the leaf node with index leaf_index is stored at hashes[leaf_index, leaf_index] and the hash value for the internal node with index pair (left_index, right_index) is stored at hashes[left_index, right_index]. The organization of the array is up to the implementation.

A new MTLNS node set object initially has a specified public seed and series identifier, a node count of 0 and an empty array of hash values.

8.2. MTL Node Set Hash Operations

As discussed in Section 6.3 and Section 6.4, MTL mode node sets are constructed using two hash operations hash_leaf and hash_int. The hash operations are specified in terms of the SLH-DSA abstract cryptographic functions and the address scheme in Section 4.

8.2.1. Hashing a Data Value to Produce a Leaf Node Hash Value (Function: hash_leaf)

hash_leaf is a supporting function that hashes a data value to produce a leaf node. The operation takes a public seed, a series identifier, a leaf index and a data value as input and returns a leaf node hash value.

The operation uses the MTL_DATA address type and the abstract cryptographic function F.

```
   ########################################################################
   # Algorithm 1: Hashing a Data Value to Produce a Leaf Node.
   ########################################################################
   # Input: seed, public seed for the node set (associated with
   #     public key)
   # Input: sid, series identifier for the node set
   # Input: leaf_index of the leaf node hash value
   #     being computed
   # Input: data_value corresponding to the leaf node
   # Output: leaf_hash, leaf node hash value

   def hash_leaf(seed, sid, leaf_index, data_value):
       dataADRS = spx.ADRS()
       dataADRS.setType(spx.ADRS.MTL_DATA)
       dataADRS.setSID(sid)
       dataADRS.setLeafIndex(leaf_index)

       leaf_hash = spx.F(seed, dataADRS.bytes(), data_value)

       return leaf_hash
```

8.2.2. Hashing Two Child Nodes to Produce an Internal Node

hash_int is a supporting function that hashes two child nodes to produce an internal node. The operation takes a public seed, a series identifier, a left index, a right index, a left child hash value and a right child hash value as input and returns an internal node hash value.

The operation uses the MTL_TREE address type and the abstract cryptographic function H.

```
   ########################################################################
   # Algorithm 2: Hashing Two Child Nodes to Produce an Internal Node.
   ########################################################################
   # Input: seed, public seed for the node set (associated with
   #     public key)
   # Input: sid, series identifier for the node set
   # Input: left_index, left index of the internal node
   # Input: right_index, right index of the internal node
   # Input: left_hash, left child hash value
   # Input: right_hash, right child hash value
   # Output: internal_hash, internal node hash value

   def hash_int(seed, sid, left_index, right_index, left_hash,
                right_hash):
       mtlnsADRS = spx.ADRS()
       mtlnsADRS.setType(spx.ADRS.MTL_TREE)
       mtlnsADRS.setSID(sid)
       mtlnsADRS.setLeftRightIndexes(left_index, right_index)

       internal_hash = spx.H(seed, mtlnsADRS.bytes(), left_hash,
                             right_hash)

       return internal_hash
```

8.3. Initializing a MTL Node Set (Function: mtl_initns)

mtl_initns initializes a new MTL node set. The operation takes a public seed and a series identifier as input and returns a new MTL node set object.

```
   ########################################################################
   # Algorithm 3: Initializing a MTL Node Set.
   ########################################################################
   # Input: seed, public seed for the node set (associated with
   #     public key)
   # Input: sid, series identifier for the node set
   # Output: node_set, new MTL node set object

   def mtl_initns(seed, sid):
       node_set = MTLNS(seed, sid)
       return node_set
```

8.4. Appending a Data Value to a MTL Node Set (Function: mtl_append)

mtl_append appends a data value to a MTL node set, adding a leaf node and internal nodes as needed to produce a new ladder covering the expanded series of data values. The operation takes a data value and a MTL node set object as input and returns the new leaf index. The MTL node set object is updated in place.

mtl_append maintains the node set in a way that can produce ladders and authentication paths with the binary rung strategy.

The operation has two primary steps. First, a new leaf node is computed from the data value and added to the node set. Second, new internal nodes are computed from other nodes (if needed) and added to the node set to produce a new ladder covering the expanded series of data values.

```
######################################################################
# Algorithm 4: Appending a Data Value to a MTL Node Set.
######################################################################
# Input: data_value to append to the node set
# Input: node_set, MTL node set object (updated in place)
# Output: leaf_index assigned to the data value

def mtl_append(data_value, node_set):
    leaf_index = node_set.count
    node_set.count = leaf_index + 1

    seed = node_set.seed
    sid = node_set.sid

    # Compute and store the leaf node hash value
    node_set.hashes[leaf_index,leaf_index] = hash_leaf(seed, sid,
                leaf_index, data_value)

    # Compute and store additional internal node hash values
    for i in range(1,lsb(leaf_index+1)):
        left_index = leaf_index - 2**i + 1
        mid_index = leaf_index - 2**(i-1) + 1
        node_set.hashes[left_index, leaf_index] = hash_int(seed,
                sid, left_index, leaf_index,
                node_set.hashes[left_index, mid_index - 1],
                node_set.hashes[mid_index, leaf_index])

    return leaf_index
```

8.5. Computing an Authentication Path (Function: mtl_authpath)

mtl_authpath computes an authentication path for the data value at a specified leaf index relative to the current ladder for a MTL node set. The operation takes a leaf index and a node set object as input and returns an authentication path from the leaf node to its associated rung in the node set’s current ladder.

mtl_authpath produces the authentication path with the binary rung strategy.

The operation has two primary steps. First, the current ladder rung covering the specified leaf index is selected. Second, the sibling hash values from the leaf node to the rung are concatenated to produce the authentication path.

```
   ######################################################################
   # Algorithm 5: Computing an Authentication Path for a Data Value.
   ######################################################################
   # Input: leaf_index, leaf node index of the data value to
   #     authenticate
   # Input: node_set, MTL node set object encompassing the specified
   #     leaf node
   # Output: auth_path, authentication path from the leaf node to
   #     the associated rung

   def mtl_authpath(leaf_index, node_set):
       left_index = 0
       sibling_hashes  = []
       flags = 0

       # Check that the leaf is part of this node set
       if(leaf_index < 0) or (leaf_index >= node_set.count):
           return None # Leaf is outside of node set

       # Find the rung index pair covering the leaf index
       for i in range(msb(node_set.count),-1,-1):
           if node_set.count & (1 << i):
               right_index = left_index + 2**i - 1
               if leaf_index <= right_index:
                   break
               left_index = right_index + 1

       # Concatenate the sibling nodes from the leaf to the rung
       for i in range(0, bit_width(right_index-left_index)):
           if leaf_index & (1<<i):
               sibling_left = (~(2**i-1) & leaf_index) - 2**i
           else:
               sibling_left = (~(2**i-1) & leaf_index) + 2**i
           sibling_right = sibling_left + 2**i -1
           sibling_hashes.append(node_set.hashes[sibling_left,
                                                 sibling_right])

       auth_path =  MTL_AUTH_PATH(flags, leaf_index, node_set.sid,
                           sibling_hashes, left_index, right_index)
       return auth_path
```

8.6. Computing the Merkle Tree Ladder for a Node Set (Function: mtl_ladder)

mtl_ladder computes the Merkle tree ladder for a node set. It takes a node set object as input and returns the ladder.

mtl_ladder produces the ladder with the binary rung strategy.

The operation has one primary steps: the current ladder rungs are concatenated to produce the ladder.

```
######################################################################
# Algorithm 6: Computing a Merkle Tree Ladder for a Node Set.
######################################################################
# Input: node_set, MTL node set object
# Output: ladder, Merkle tree ladder for this node set

def mtl_ladder(node_set):
    left_index = 0
    rungs = []
    flags = 0

    # Concatenate the rungs in the node set
    for i in range(msb(node_set.count),-1,-1):
        if node_set.count &(1 << i):
            right_index = left_index + 2**i - 1
            rungs.append(RUNG(left_index, right_index,
                        node_set.hashes[left_index,right_index]))
            left_index = right_index + 1

    ladder = MTL_LADDER(flags, node_set.sid, rungs)
    return ladders
```

8.7. Selecting a Ladder Rung (Function: mtl_rung)

mtl_rung selects a ladder rung associated with an authentication path. It takes a ladder and an authentication path as input and returns the associated rung of the lowest degree that can be used to verify the authentication path if the ladder has one, or None if not.

mtl_rung supports authentication paths produced with the binary rung strategy.

The operation has two primary steps. First, the authentication path is checked to confirm that its target rung index pair is compatible with the binary rung. Second, the ladder rungs are searched for the compatible rung of lowest degree that can be used to verify the authentication path.

```
######################################################################
# Algorithm 7: Selecting a Ladder Rung.
######################################################################
# Input: auth_path, authentication path from the leaf node to
#     a rung in the ladder
# Input: ladder, Merkle tree ladder to authenticate relative to
# Output: assoc_rung, the rung in the ladder associated with the
```

```
   #     authentication path or None

   def mtl_rung(auth_path, ladder):

       # Check that authentication path and ladder have same SID
       if(auth_path.sid != ladder.sid):
           return None

       leaf_index = auth_path.leaf_index
       sibling_hash_count = auth_path.sibling_hash_count

       # Check that authentication path is a binary rung strategy path
       left_index = leaf_index & ~(2**sibling_hash_count-1)
       right_index = left_index + (2**sibling_hash_count-1)
       if((auth_path.rung_left != left_index) or
          (auth_path.rung_right != right_index)):
           return None

       # Find associated rung with lowest degree, if present
       assoc_rung = None
       # Minimum degree is updated after first rung is found
       min_degree = -1

       for rung in ladder.rungs:
           # Check if rung index pair would be encountered in
           #     evaluating authentication path for leaf node
           left_index = rung.left_index
           right_index = rung.right_index
           if((left_index <= leaf_index) and
              (right_index >= leaf_index)):
               degree = lsb(right_index-left_index+1)-1
               if(((degree <= lsb(left_index)-1) or
                   (lsb(left_index) == 0)) and
                  (right_index-left_index+1 == 2**degree) and
                  (degree <= sibling_hash_count)):
                   if((assoc_rung == None) or
                      (degree < min_degree)):
                       assoc_rung = rung
                       min_degree = degree

       return assoc_rung
```

8.8. Verifying an Authentication Path (Function: mtl_verify)

mtl_verify verifies an authentication path for a data value relative to a rung. It takes a public seed, a data value and a rung as input and returns a Boolean value indicating whether the data value is successfully authenticated.

mtl_verify supports authentication paths produced with the binary rung strategy.

The operation has two primary steps. First, a leaf node hash value is computed from the data value using hash_leaf. If the leaf node index matches the rung index pair, the leaf node hash value is compared to the rung hash value. Second, internal node hash values are computed as needed from the leaf node hash value and successive sibling hash values in the authentication path using hash_int. Along the way, if the internal node index pair matches the rung index pair, then the internal hash value is compared to the rung hash value.

```
######################################################################
# Algorithm 8: Verifying an Authentication Path.
######################################################################
# Input: seed value for this operation (associated with public key)
# Input: data_value to authenticate
# Input: auth_path, (presumed) authentication path from corresponding
#     leaf node to rung of ladder covering leaf node
# Input: assoc_rung, Merkle tree rung to authenticate relative to
# Output: result, a Boolean indicating whether the data value is
#     successfully authenticated

def mtl_verify(seed, data_value, auth_path, assoc_rung):

    sid = auth_path.sid
    leaf_index = auth_path.leaf_index
    sibling_hash_count = auth_path.sibling_hash_count

    # Recompute leaf node hash value
    target_hash = hash_leaf(seed, sid, leaf_index, data_value)

    # Compare leaf node hash value to associated rung hash value if
    #     index pairs match
    if ((leaf_index == assoc_rung.left_index) and
        (leaf_index == assoc_rung.right_index)):
        return target_hash == assoc_rung.hash

    # Recompute internal node hash values and compare to
    #    associated rung hash value if index pairs match
    for i in range(1, sibling_hash_count+1):
        left_index  = leaf_index & ~(2**i-1)
        right_index = left_index + 2**i -1
        mid_index   = left_index + 2**(i-1)

        sibling_hash = auth_path.sibling_hash[i-1]
        if leaf_index < mid_index:
            target_hash = hash_int(seed, sid, left_index,
```

```
                                    right_index, target_hash,
                                    sibling_hash)
            else:
                target_hash = hash_int(seed, sid, left_index,
                                    right_index, sibling_hash,
                                    target_hash)

            # Break if associated rung reached
            if((left_index == assoc_rung.left_index) and
               (right_index == assoc_rung.right_index)):
                return target_hash == assoc_rung.hash

        return False
```

9. Signing and Verifying Messages in MTL Mode

Descriptions of the signer's and the verifier's operations in a typical application based on MTL mode are given in Section 9.1 and Section 9.2. Section 9.3 discusses how to identify ladders to facilitate interoperability. Representations of full and condensed MTL mode signatures are given in Section 9.4 and Section 9.5.

9.1. Signing Messages

A signer MUST perform the following or an equivalent set of operations to sign messages in MTL mode.

The first step is performed once per key pair:

1. Generate a public / private key pair for an underlying signature scheme, where the public key includes a public seed and a public root and the private key includes a public seed, a public root, and a secret PRF key.

The second step is performed once per series of messages to be signed:

2. Initialize a node set for the series from a public seed and a series identifier using mtl_initns. The message index for the message series is set to 0 in this step.

The third and fourth steps are performed once per message to be signed in a message series:

3. Compute a randomizer and a data value from the message, the context string, the series identifier and the message index for the message series as described in Section 5.1. Note that a domain separator of type MTL_MSG_SEP is prepended to the inputs to functions in Section 5.1. (See Section 4.7 for further discussion.)

4. Append the data value to the node set for the message series using mtl_append. The message index for the message series is incremented in this step.

The fifth and sixth steps are performed whenever the signer wants to produce a new signed ladder for the message series. The signer could do so after each new message is added, or after a new batch of new messages is added.

5. Compute the current ladder for the node set using mtl_ladder.

6. Format a domain separator sep from the context string as follows:

sep = octet(MTL_LADDER_SEP) || octet(OLEN(ctx)) || ctx || OID_MTL

Sign the ladder prepended with the domain separator using the private key of the underlying signature scheme. (See Section 4.7 for further discussion).

The seventh step is performed whenever the signer wants to provide a signed ladder to a requester, e.g., upon request by a verifier. (This step may not be needed if the signer supports the alternative of providing a full signature including the authentication path and a ladder.)

7. Provide the signed ladder associated with a specified ladder identifier.

The eighth step is performed whenever the signer wants to compute a new authentication path for a message relative to the current ladder for the message series. The signer could do so after each new message is added, after a batch of new messages is added, and/or later, as needed, to update the authentication paths for older messages so that they are relative to the current ladder.

8. Compute an authentication path for the data value at a specified message index relative to the current ladder using mtl_authpath.

The ninth step is performed whenever the signer wants to provide authentication information to a requester, e.g., in conjunction with a message to be authenticated.

9. Provide the authentication path and the randomizer associated with a message to be authenticated, e.g., in a condensed signature. The signer MAY also provide an explicit ladder identifier for the ladder that the authentication path was computed relative to - see Section 9.3. Alternatively, the signer may offer the option of requesting a full signature that includes the authentication path, the randomizer and a signed ladder.

9.2. Verifying Signatures

A verifier MUST perform the following or an equivalent set of operations to verify signatures in MTL mode:

The first step is performed once per key pair by each verifier:

1. Obtain the signer’s public key for the underlying signature scheme, where the public key includes a public seed and a public root.

The second, third, fourth and fifth steps is performed as needed for each message to be authenticated:

2. Obtain the authentication path and the randomizer associated with the message to be authenticated, e.g., in a condensed signature. The verifier MAY also obtain an explicit ladder identifier for the ladder that the authentication path was computed relative to - see Section 9.3.

3. Determine whether any of ladders held by the verifier includes a rung compatible with the authentication path, e.g., using mtl_rung. If not, then proceed to Step 6 and return here.

4. Re-compute a data value from the message, the context string, the series identifier in the authentication path and the message index in the authentication path and the randomizer as described in Section 5.2. Note that a domain separator of type MTL_MSG_SEP is prepended to the message prior to calling the functions in Section 5.2. (See Section 4.7 for further discussion).

5. Verify the authentication path for the data value at a specified message index relative to the compatible rung using mtl_verify.

The sixth and seventh steps are performed when the verifier doesn’t have a ladder compatible with an authentication path.

6. Obtain the signed ladder associated with a specified ladder identifier. Alternatively, the verifier may request a full signature including an authentication path and the signed ladder that it is computed relative to.

7. Format a domain separator sep from the context string and the MTL mode object identifier as follows:

```
sep = octet(MTL_LADDER_SEP) || octet(OLEN(ctx)) || ctx || OID_MTL
```

Verify the signature on the ladder prepended with the domain separator using the public key of the underlying signature scheme. (See Section 4.7 for further discussion).

9.3. Ladder identifiers

To facilitate interoperability, an application SHOULD have a way for signers and verifiers to identify a specific signed ladder that a verifier is interested in obtaining.

Potential approaches include:

* Identifying the ladder based on a public key identifier and information in the authentication path itself, i.e., the series identifier and the target index pair. This combination is sufficient to identify the public key, the series of data values (and thus the MTL node set), and the ladder of interest (given the target index pair, with the binary rung strategy).
* Identifying the ladder with a URI or other explicit identifier that refers to a location where the signed ladder is stored or to the signed ladder itself. This URI can be conveyed together with the authentication path in an application.
* Specifying interest in a ladder implicitly by setting a flag in the request for a message and its associated authentication path. When the flag is not set, the message and authentication path would be returned (producing a condensed signature - see Section 9.5). When the flag is set, the message, the signed ladder is also would be returned (producing a full signature - see Section 9.4).

The approach MAY be protocol-specific, e.g., the approach used for identifying MTL mode ladders associated with DNSSEC signatures MAY be different than the one used for MTL mode ladders associated with certificates.

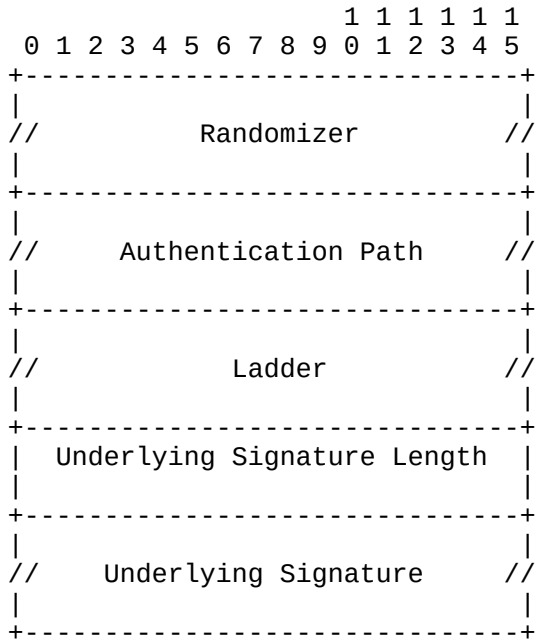
9.4. Full Signatures

An application MAY convey a "full" signature - including a randomizer, an authentication path, a ladder and its signature - with the following data structure. A full signature is convenient as a "drop-in" for a conventional signature because it can be verified on its own. However, it includes the overhead of the underlying signature on the ladder.

A full MTL mode signature data structure consists of five base components:

* R_mtl, the randomizer, a n-byte string
* auth_path, the authentication path Section 7.3
* ladder, the ladder Section 7.1
* sig_len, the length in bytes of the underlying signature on the ladder, a positive integer between 1 and 2^32-1 (so it can be represented as 4-byte string in big endian notation)
* sig, the underlying signature on the ladder, a scheme-specific string

The byte representation of the full MTL mode signature is the concatenation of the binary encodings of the fields, using a 4-byte big endian representation for the signature length.



Note that the last three components are the same as a signed ladder (Section 9.6).

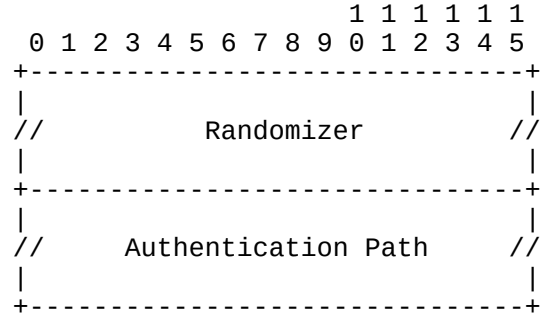
9.5. Condensed Signatures

An application MAY convey a "condensed" signature - including a randomizer and an authentication path but not a ladder and its signature - with the following data structure. A condensed signature is convenient for reducing the size impact of the ladder signature. However, it requires the verifier to obtain the ladder from a separate source.

A condensed MTL mode signature data structure consists of two base components:

* R_mtl, the randomizer, a n-byte string
* auth_path, the authentication path (Section 7.3)

The byte representation of the condensed MTL mode signature is the concatenation of the binary encodings of the fields.

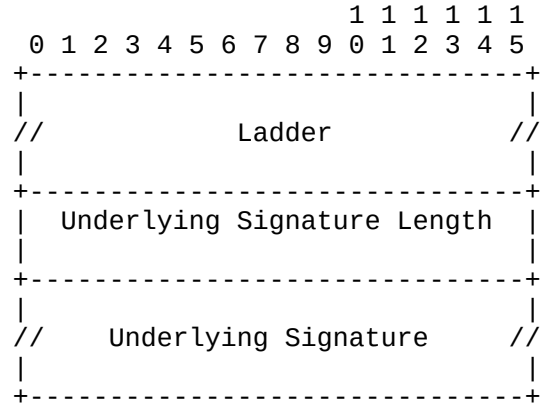


9.6. Signed Ladders

A signed ladder data structure consists of three base components:

* ladder, the ladder (Section 7.1)
* sig_len, the length in bytes of the underlying signature on the ladder, a positive integer between 1 and 2^32-1 (so it can be represented as 4-byte string in big endian notation)
* sig, the underlying signature on the ladder, a scheme-specific string

The byte representation of the signed ladder is the concatenation of the binary encodings of the fields, using a 4-byte big endian representation for the signature length.



10. SLH-DSA in MTL Mode

SLH-DSA-MTL applies MTL mode to the underlying signature scheme SLH-DSA. This document supports 12 instantiations corresponding to the six instantiations in [FIPS205] based on the SHAKE hash function [FIPS202] and the six instantiations based on the SHA2 hash functions [FIPS180-4].

The names of the SLH-DSA-MTL instantiations follow a similar convention to the SLH-DSA instantiations:

* SLH-DSA-MTL-{hash}-{bitsec}{opt}-{variant}

The components of the name are as follows:

* {hash} is the underlying hash function. For this version of the document, it MUST be "SHAKE" or "SHA2", corresponding to the underlying hash function.

* {bitsec} is the target bit security level. It MUST be "128", "192" or "256". The target bit security level is the security parameter n times by 8. The corresponding NIST security strength categories for these bit security levels are 1, 3 and 5.

* {opt} is the optimization goal. It MUST be "s" or "f". As discussed in [FIPS205], the "s" optimization results in smaller signature sizes, while the "f" optimization results in faster signing operations.

SLH-DSA-MTL with a given set of components is based on the underlying signature scheme SLH-DSA with the same components. SLH-DSA-MTL-{hash}-{bitsec}{opt} may thus be read as "MTL mode of SLH-DSA-{hash}-{bitsec}{opt}".

The three components may be chosen independently of one another. Given that there are two choices of {hash}, three choices of {bitsec}, and two choices of {opt}, the total number of instantiations is 2*3*2 = 12. The table below lists the names of the 12 supported instantiations, their associated security parameter n, and their NIST security categories.

As in SLH-DSA itself, the instantiations of the abstract cryptographic functions in the MTL mode operations depend on the underlying hash function and the security parameter. The function definitions for each instantiation are given in the following subsections. With the exception of H_msg_mtl, which is new, the instantiations of the functions are the same as in SLH-DSA and are repeated here for completeness. Recall that the parameters to these functions should be provided as defined in Section 5 of this draft.

SLH-DSA-MTL Instantiation	Security Parameter n	NIST Category
SLH-DSA-MTL-SHAKE-128s	16	1
SLH-DSA-MTL-SHAKE-128f	16	1
SLH-DSA-MTL-SHAKE-192s	24	3
SLH-DSA-MTL-SHAKE-192f	24	3
SLH-DSA-MTL-SHAKE-256s	32	5
SLH-DSA-MTL-SHAKE-256f	32	5
SLH-DSA-MTL-SHA2-128s	16	1
SLH-DSA-MTL-SHA2-128f	16	1
SLH-DSA-MTL-SHA2-192s	24	3
SLH-DSA-MTL-SHA2-192f	24	3
SLH-DSA-MTL-SHA2-256s	32	5
SLH-DSA-MTL-SHA2-256f	32	5

10.1. SHAKE instantiations

The SHAKE instantiations employ the SHAKE256 hash function [FIPS202].

10.1.1. Randomized message digest function (H_msg_mtl)

The randomized message digest function H_msg_mtl (see Section 4.3) is defined the same as the function H_msg in SLH-DSA except that H_msg_mtl's output is 8n bits (n bytes) rather than 8m bits (m bytes):

H_msg_mtl(R, PK.seed, PK.root, M) = SHAKE256(R || PK.seed || PK.root || M, 8n)

10.1.2. Pseudorandom function (PRF_msg)

The pseudorandom function PRF_msg (see Section 4.4) is defined the same as in SLH-DSA:

PRF_msg(SK.prf, OptRand, M) = SHAKE256(SK.prf || OptRand || M, 8n)

10.1.3. Tweakable hash functions (F and H)

The tweakable hash functions F and H (see Section 4.5) are defined the same as in SLH-DSA:

F(PK.seed, ADRS, M_1) = SHAKE256(PK.seed || ADRS || M_1, 8n)

H(PK.seed, ADRS, M_1, M_2) = SHAKE256(PK.seed || ADRS || M_1 || M_2, 8n)

10.2. SHA2 instantiations

The SHA2 instantiations employ the SHA2-256 and/or SHA2-512 hash functions [FIPS186-4], the HMAC-SHA2-256 and/or HMAC-SHA2-512 message authentication codes (pseudorandom functions) [FIPS198-1] and the MGF1-SHA2-256 and/or MGF1-SHA2-512 mask generation functions [RFC8017]. (Here and in the following, SHA-X denotes SHA2-256 when n = 16 and SHA2-512 when n = 24 or 32.)

10.2.1. Randomized message digest function (H_msg_mtl)

The randomized message digest function H_msg_mtl (see Section 4.3) is defined the same as the function H_msg in SLH-DSA except that its output is n bytes rather than m bytes:

H_msg_mtl(R, PK.seed, PK.root, M) = MGF1-SHA-X(R || PK.seed || SHA-X(R || PK.seed || PK.root || M), n).

10.2.2. Pseudorandom function (PRF_msg)

The pseudorandom function PRF_msg (see Section 4.4) is defined the same as in SLH-DSA:

PRF_msg(SK.prf, OptRand, M) = HMAC-SHA-X(SK.prf, OptRand || M)

10.2.3. Tweakable hash functions (F and H)

The tweakable hash functions F and H (see Section 4.5) are defined the same as in SLH-DSA:

F(PK.seed, ADRS, M_1) = SHA2-256(BlockPad(PK.seed) || ADRS^c || M_1)

H(PK.seed, ADRS, M_1, M_2) = SHA-X(BlockPad(PK.seed) || ADRS^c || (M_1 ||M_2))

11. Calculating Maximum Signature Sizes

Parameters required for calculation:

* n = Security parameter for the underlying signature scheme (Section 10)
* USS = Size of underlying signature (Table 1 SLH-DSA parameter sets in [FIPS205])
* N = Number of messages in message series

Calculations:

Max Condensed Signature Size = n + 24 + (n * floor(log2N)) (Section 6.6, Section 7.3, Section 9.5)
Max Signed Ladder Size = 16 + ((8 + n) * ceiling(log2(N))) + USS (Section 6.5, Section 7.1, Section 9.6)
Max Full Signature Size = Max Condensed Signature Size + Max Signed Ladder Size (Section 9.4)

12. Related Work

The binary rung strategy appears under different names in other cryptographic constructions based on Merkle trees. Champine defines a binary numeral tree [BIN-NUM-TREES] with similar structure (the successive perfect binary subtrees are called eigentrees). Other similarMerkle tree-based constructions include Crosby and Wallach’s history trees [HISTORY-TREE], Todd’s Merkle mountain ranges [MERKLE_MOUNTAIN], and Reyzin and Yakoubov’s cryptographic accumulator [CRYPTO-ACC], which achieves an "old-accumulator compatibility" property comparable to the backward compatibility

property described here for the binary rung strategy. Certificate transparency logs take advantage of Merkle trees to show the existence or non-existence of a certificate as published by a certification authority [RFC9162]. Benjamin, O'Brien, and Westerbaan [I-D.davidben-tls-merkle-tree-certs] propose using authentication paths to a limited lifetime Merkle tree produced by a certificate transparency service as certificate signatures. Each Merkle tree is constructed over a fixed time window in this approach, and the authentication paths are constructed relative to the single root of the Merkle tree, which is like a single-rung Merkle tree ladder.

13. IANA Considerations

This document makes no requests of IANA. However, a future version of this document may request that IANA register any or all of the following:

* flag values for the ladder and authentication path data structures;
* object identifiers for the various instantiations of MTL mode combined with underlying signature schemes;
* the domain separator types MTL_MSG_SEP and possibly MTL_LADDER_SEP; and
* the address types MTL_MSG, MTL_DATA, and MTL_TREE.

Because the domain separator types build on NIST's proposed domain separation scheme, the request for domain separator types may depend on the existence of a registry for domain separation type. Similarly, because the address types build on the SLH-DSA address scheme, that request may depend on the existence of a registry for SLH-DSA address types in conjunction with the adoption of SLH-DSA by the IETF.

14. Implementation Status

For testing purposes, a reference implementation of MTL mode is available in C. The MTL library can be found at https://github.com/verisign/MTL (https://github.com/verisign/MTL) and includes example tools for key generation, signing, and verifying messages with an MTL message series.

15. Security Considerations

Implementers MUST use a secure random generator [RFC4086].

Implementers MUST select a security parameter consistent with their security requirements.

Implementers MUST also select cryptographic functions that are generally accepted for their intended security strength category and use within MTL mode. Similar to SLH-DSA, the desired properties for the cryptographic functions in MTL mode are that PRF_msg is a pseudorandom function and F and H are multi-target, multi-function second preimage resistant function families. The desired property for H_msg_mtl is that it is an extended target collision resistant function with nonce (where the nonce is provided as the message index in the prepended address value).

Because MTL mode is an add-on to an underlying signature scheme, implementers MUST ensure adequate cryptographic separation between MTL mode's uses of cryptographic functions and the use of cryptographic functions by the underlying signature scheme. The operations in Section 5.1, Section 5.2, Section 9.1, and Section 9.2 including the domain separators, and the proposed instantiations in Section 10 were selected taking cryptographic separation between MTL mode and SLH-DSA into account. Other underlying signature schemes need to be evaluated separately.

To avoid unintended interactions between the different instantiations of MTL mode, a given key pair SHOULD only be used with a single instantiation of MTL mode.

The signer in MTL mode maintains a Merkle tree node set and is therefore stateful. Implementers SHOULD ensure that the node set is maintained accurately and is not at risk of being reset or repeated, or otherwise the security of MTL mode could be degraded. In particular, as discussed in [MTL-MODE], the reuse of state in MTL mode could provide additional target hash values for an adversary to match in an attack on the hash function, thereby weakening the provable security bounds. In contrast to hash-based signature schemes, however, the reuse of state in MTL mode does not reveal information about a private key that could directly lead to a signature forgery.

Similar to SLH-DSA, the security of MTL mode relies on a form of target collision resistance. Target collision resistance assumes that the message is hashed together with a randomizer that is produced with a secure random generator. An advantage of target collision resistance over basic collision resistance (without a randomizer) is that the bit security level associated with security parameter n can be achieved with an n-byte hash value rather than a 2n-byte hash value. This advantage reduces the size of the authentication paths and ladders in MTL mode, in a similar way that it reduces the size of the signatures in SLH-DSA. If the randomizer in the MTL mode operations is not produced securely, however, then the security of the MTL mode operations could be significantly at

risk. In particular, if an adversary can predict the randomizer, then an attacker can potentially perform a basic collision attack to find two messages that hash to the same hash value together with the predicted randomizer. Because the hash value is only n bytes, the implementation in this case would only have roughly 4n bits of security rather than the target 8n bits.

Section 5.1 suggests one primary approach to secure random generation, namely the use of a pseudorandom function PRF_msg. This is the same approach that SLH-DSA takes, but the inputs are different for the MTL mode operations.

In SLH-DSA, the PRF is applied to an optional external random value and the message being signed. Including the message ensures that the randomizer for one signature operation is cryptographically independent from the randomizer for another signature operation involving a different message, while maintaining the statelessness of SLH-DSA. If same message is signed twice, however, the same randomizer and ultimately the same signature will be generated (deterministic signing). Deterministic signing could increase the risk of side-channel attacks revealing information about the signer’s private key if the same computation is run multiple times. Including the optional random value avoids deterministic signing and mitigates this particular risk.

In the MTL mode operations, following the suggested example in Section 5.1, the PRF is applied to an optional external random value and an address value. Including the address value ensures that the randomizer for one signature operation is cryptographically independent from another signature operation, regardless of the message being signed. Here, the MTL mode operations are already stateful, and implementations are assumed to be able to maintain a unique address value. If the implementation is not able to maintain a unique address value, however, and uses the same address value more than once, the same randomizer will be generated when the address value is used again, making it predictable. This could lead to a basic collision attack. Including the optional random value avoids this predictability and mitigates the risk. Section 5.1 also suggests the option of including the message in the input in addition to the address value, which is another way to avoid the predictability of the randomizer if the same address value is used more than once (with a different message each time). As noted in Section 5.1, however, including the message involves an additional pass over the message (once for PRF_msg and once for H_msg_mtl), which can be disadvantageous if the message is long. For this reason, including the message is OPTIONAL.

This specification uses a domain separator format to distinguish between the use of an underlying signature scheme in MTL mode from its use in pure and pre-hash modes. The domain separator format is beneficial in an application that potentially supports MTL mode in addition to pure or more modes of operation for a given key pair. Other ways to protect against unintended interactions between the use of a key pair in different modes of operation include (a) supporting only a single mode of operation within the application, e.g., only MTL mode, and not allowing key pairs used within the application to be used in other applications; and (b) specifying the mode of operation for a key pair as part of key management information (e.g., via an algorithm identifier or key usage constraints).

Considerations for the optional context string are intended to be the same as those for the underlying signature scheme (if it supports one, as FIPS 205 does). Section 8.3 of [RFC8032] provides helpful guidance on the use of context strings.)

An adversary who learns a set of messages and their MTL mode signatures also learns the leaf nodes of the Merkle node set corresponding to these messages, the authentication paths in the signatures, and the signed ladders in the signatures. Such an adversary can also compute any nodes of the Merkle node set that depends on the nodes it has learned, and form other condensed and full signatures on the messages it has learned (see Section E.2.4 for discussion; the adversary can perform the same steps as an intermediary). Even with the ability to reconstruct the Merkle node set, however, assuming cryptographic security of MTL mode and the underlying signature scheme, an adversary cannot form signatures on messages that have not already been signed by the signer, as it does not have access to the signer’s private key.

The adversary also cannot form signatures on messages that have already been signed by the signer but that the adversary has not yet learned, because the adversary does not know and cannot predict the randomizers associated with those messages. Moreover, because of the lack of knowledge about the other messages’ randomizers, the adversary also cannot determine which messages have been signed based on the reconstructed Merkle node set, other than those whose randomizers it has already learned. Therefore, even though the Merkle tree node set can gradually be reconstructed publicly, the individual messages that have been signed by the signer remain private until the signer publishes them.

16. References

16.1. Normative References

[FIPS180-4]
National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", National Institute of Standards and Technology", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf>.

[FIPS186-4]
National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[FIPS198-1]
National Institute of Standards and Technology (NIST), "The Keyed-Hash Message Authentication Code (HMAC)", FIPS PUB 198-1, DOI 10.6028/NIST.FIPS.198-1, July 2008, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.198-1.pdf>.

[FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, DOI 10.6028/NIST.FIPS.202, August 2015, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

16.2. Informative References

[BIN-NUM-TREES]
Champine, L., "Streaming Merkle Proofs within Binary Numeral Trees", Cryptology ePrint Archive Paper 2021/038, 2021, <https://eprint.iacr.org/2021/038>.

[CRYPTO-ACC]
Reyzin, L. and S. Yakoubov, "Efficient data structures for tamper-evident logging", 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>.

[FIPS205]
National Institute of Standards and Technology (NIST), "Stateless Hash-Based Digital Signature Standard", FIPS PUB 205, DOI 10.6028/NIST.FIPS.205, 13 August 2024, <https://doi.org/10.6028/NIST.FIPS.205>.

[HISTORY-TREE]
Crosby, S. and D. Wallach, "Efficient data structures for tamper-evident logging", Proceedings of the 18th USENIX Security Symposium pp. 317-334. USENIX Association (2009), <https://dl.acm.org/doi/abs/10.5555/1855768.1855788>.

[I-D.davidben-tls-merkle-tree-certs]
Benjamin, D., O'Brien, D., and B. Westerbaan, "Merkle Tree Certificates for TLS", Work in Progress, Internet-Draft, draft-davidben-tls-merkle-tree-certs-04, 3 March 2025, <https://datatracker.ietf.org/api/v1/doc/document/draft-davidben-tls-merkle-tree-certs/>.

[I-D.fregly-dnsop-slh-dsa-mtl-dnssec]
Fregly, A., Harvey, J., Kaliski, B., and D. Wessels, "Stateless Hash-Based Signatures in Merkle Tree Ladder Mode (SLH-DSA-MTL) for DNSSEC", Work in Progress, Internet-Draft, draft-fregly-dnsop-slh-dsa-mtl-dnssec-03, 8 October 2024, <https://datatracker.ietf.org/doc/html/draft-fregly-dnsop-slh-dsa-mtl-dnssec-03>.

[I-D.harvey-cfrg-mtl-mode-considerations]
Harvey, J., Kaliski, B., Fregly, A., and S. Sheth, "Considerations for Integrating Merkle Tree Ladder (MTL) Mode Signatures into Applications", Work in Progress, Internet-Draft, draft-harvey-cfrg-mtl-mode-considerations-01, 21 February 2025, <https://datatracker.ietf.org/doc/html/draft-harvey-cfrg-mtl-mode-considerations-01>.

[MERKLE_MOUNTAIN]
Todd, P., "Merkle Mountain Ranges", <https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md>.

[MTL-MODE] Fregly, A., Harvey, J., Kaliski, B., and S. Sheth, "Merkle Tree Ladder Mode: Reducing the Size Impact of NIST PQC Signature Algorithms in Practice", in Rosulek, M. (editor), Lecture Notes in Computer Science VOLUME 13871, CT-RSA 2023 - Cryptographers Track at the RSA Conference pages 415-441, DOI 10.1007/978-3-031-30872-7_16, 2023, <https://eprint.iacr.org/2022/1730.pdf>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9162] Laurie, B., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", RFC 9162, DOI 10.17487/RFC9162, December 2021, <https://www.rfc-editor.org/info/rfc9162>.

[SPHINCS-PLUS]
Aumasson, J., Bernstein, D., Beullens, W., and E. al., "SPHINCS+ - Submission to the NIST post-quantum project, v3.1", 10 June 2022, <https://sphincs.org/data/sphincs+-r3.1-specification.pdf>.

Appendix A. Change Log

00: Initial draft of the document.

01: Fixed 10.2.3 Tweakable hash functions definitions. Fixed typo in Section 6.5. Added text to help clarify inputs to the H_msg_mtl and PRF_msg functions. Added reference to draft FIPS 205.

02: Updated algorithm IDs for alignment with draft FIPS 205. Fixed a typo in Sections 13 and 9.1.

03: Generalized how MTL mode randomizer is generated so that the message does not need to be an input to PRF_msg. Updated cryptographic separation to use "pre-hash" domain separator format for input to the underlying signature scheme for compatibility with NIST's recently proposed guidance for FIPS 204 and FIPS 205 pre-hashing. Added security considerations on randomizer generation and on message privacy. Other minor edits.

04: Updated document to align with FIPS-205 and remove SPHINCS+ references (aside from retaining historical commentary).

Acknowledgements

The authors would like to acknowledge the following individuals for their contributions to this document: Fitzgerald Marcelin.

Authors' Addresses

J. Harvey
Verisign Labs
12061 Bluemont Way
Reston
Email: jsharvey@verisign.com
URI: https://www.verisignlabs.com/

B. Kaliski
Verisign Labs
12061 Bluemont Way
Reston
Email: bkaliski@verisign.com
URI: https://www.verisignlabs.com/

A. Fregly
Verisign Labs
12061 Bluemont Way
Reston
Email: afregly@verisign.com
URI: https://www.verisignlabs.com/

S. Sheth
Verisign Labs
12061 Bluemont Way
Reston
Email: ssheth@verisign.com
URI: https://www.verisignlabs.com/