Hash-Based Signatures
draft-mcgrew-hash-sigs-15

Abstract

This note describes a digital signature system based on cryptographic hash functions, following the seminal work in this area of Lamport, Diffie, Winternitz, and Merkle, as adapted by Leighton and Micali in 1995. It specifies a one-time signature scheme and a general signature scheme. These systems provide asymmetric authentication without using large integer mathematics and can achieve a high security level. They are suitable for compact implementations, are relatively simple to implement, and naturally resist side-channel attacks. Unlike most other signature systems, hash-based signatures would still be secure even if it proves feasible for an attacker to build a quantum computer.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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 July 11, 2019.
Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................ 3
   1.1. CFRG Note on Post-Quantum Cryptography ........ 5
1.2. Intellectual Property ............................... 6
    1.2.1. Disclaimer ................................ 6
1.3. Conventions Used In This Document ................ 6
2. Interface ........................................... 6
3. Notation ............................................ 7
   3.1. Data Types ..................................... 7
      3.1.1. Operators ................................ 7
      3.1.2. Functions ............................... 8
      3.1.3. Strings of w-bit elements ............... 8
   3.2. Typecodes ..................................... 9
   3.3. Notation and Formats ............................ 9
4. LM-OTS One-Time Signatures .......................... 12
   4.1. Parameters .................................... 12
   4.2. Private Key ................................... 14
   4.3. Public Key .................................... 15
   4.4. Checksum ..................................... 15
   4.5. Signature Generation ........................... 16
   4.6. Signature Verification ........................ 17
5. Leighton–Micali Signatures .......................... 19
   5.1. Parameters .................................... 20
   5.2. LMS Private Key ............................... 21
   5.3. LMS Public Key ................................ 21
   5.4. LMS Signature ................................ 22
      5.4.1. LMS Signature Generation .................. 24
      5.4.2. LMS Signature Verification ............... 24
6. Hierarchical signatures ............................ 26
   6.1. Key Generation ................................ 29
   6.2. Signature Generation ........................... 30
   6.3. Signature Verification ........................ 32
1. Introduction

One-time signature systems, and general purpose signature systems built out of one-time signature systems, have been known since 1979 [Merkle79], were well studied in the 1990s [USPTO5432852], and have benefited from renewed attention in the last decade. The characteristics of these signature systems are small private and public keys and fast signature generation and verification, but large signatures and moderately slow key generation (in comparison with RSA and ECDSA). Private keys can be made very small by appropriate key generation, for example, as described in Appendix A. In recent years there has been interest in these systems because of their post-quantum security and their suitability for compact verifier implementations.

This note describes the Leighton and Micali adaptation [USPTO5432852] of the original Lamport-Diffie-Winternitz-Merkle one-time signature system [Merkle79] [C:Merkle87][C:Merkle89a][C:Merkle89b] and general signature system [Merkle79] with enough specificity to ensure interoperability between implementations.

A signature system provides asymmetric message authentication. The key generation algorithm produces a public/private key pair. A message is signed by a private key, producing a signature, and a message/signature pair can be verified by a public key. A One-Time
Signature (OTS) system can be used to sign one message securely, but will become insecure if more than one is signed with the same public/private key pair. An N-time signature system can be used to sign N or fewer messages securely. A Merkle tree signature scheme is an N-time signature system that uses an OTS system as a component.

In the Merkle scheme, a binary tree of height h is used to hold $2^h$ OTS key pairs. Each interior node of the tree holds a value which is the hash of the values of its two children nodes. The public key of the tree is the value of the root node (a recursive hash of the OTS public keys), while the private key of the tree is the collection of all the OTS private keys, together with the index of the next OTS private key to sign the next message with.

In this note we describe the Leighton-Micali Signature (LMS) system, which is a variant of the Merkle scheme, and a Hierarchical Signature System (HSS) built on top of it that can efficiently scale to larger numbers of signatures. In order to support signing a large number of messages on resource constrained systems, the Merkle tree can be subdivided into a number of smaller trees. Only the bottom-most tree is used to sign messages, while trees above that are used to sign the public keys of their children. For example, in the simplest case with 2 levels with both levels consisting of height h trees, the root tree is used to sign $2^h$ trees with $2^h$ OTS key pairs, and each second level tree has $2^h$ OTS key pairs, for a total of $2^{(2h)}$ bottom level key pairs, and so can sign $2^{(2h)}$ messages. The advantage of this scheme is that only the active trees need to be instantiated, which saves both time (for key generation) and space (for key storage). On the other hand, using a multilevel signature scheme increases the size of the signature, as well as the signature verification time.

This note is structured as follows. Notes on postquantum cryptography are discussed in Section 1.1. Intellectual Property issues are discussed in Section 1.2. The notation used within this note is defined in Section 3, and the public formats are described in Section 3.3. The LM-OTS signature system is described in Section 4, and the LMS and HSS N-time signature systems are described in Section 5 and Section 6, respectively. Sufficient detail is provided to ensure interoperability. The rationale for the design decisions is given in Section 7. The IANA registry for these signature systems is described in Section 8. Security considerations are presented in Section 9. Comparison with another hash based signature algorithm (XMSS) is in Section 10.

This document represents the rough consensus of the CFRG.
1.1. CFRG Note on Post-Quantum Cryptography

All post-quantum algorithms documented by the Crypto Forum Research Group (CFRG) are today considered ready for experimentation and further engineering development (e.g., to establish the impact of performance and sizes on IETF protocols). However, at the time of writing, we do not have significant deployment experience with such algorithms.

Many of these algorithms come with specific restrictions, e.g., change of classical interface or less cryptanalysis of proposed parameters than established schemes. CFRG has consensus that all documents describing post-quantum technologies include the above paragraph and a clear additional warning about any specific restrictions, especially as those might affect use or deployment of the specific scheme. That guidance may be changed over time via document updates.

Additionally, for LMS:

CFRG consensus is that we are confident in the cryptographic security of the signature schemes described in this document against quantum computers, given the current state of the research community’s knowledge about quantum algorithms. Indeed, we are confident that the security of a significant part of the Internet could be made dependent on the signature schemes defined in this document, if developers take care of the following.

In contrast to traditional signature schemes, the signature schemes described in this document are stateful, meaning the secret key changes over time. If a secret key state is used twice, no cryptographic security guarantees remain. In consequence, it becomes feasible to forge a signature on a new message. This is a new property that most developers will not be familiar with and requires careful handling of secret keys. Developers should not use the schemes described here except in systems that prevent the reuse of secret key states.

Note that the fact that the schemes described in this document are stateful also implies that classical APIs for digital signatures cannot be used without modification. The API MUST be able to handle a secret key state; in particular, this means that the API MUST allow to return an updated secret key state.
1.2. Intellectual Property

This draft is based on U.S. patent 5,432,852, which was issued over twenty years ago and is thus expired.

1.2.1. Disclaimer

This document is not intended as legal advice. Readers are advised to consult with their own legal advisers if they would like a legal interpretation of their rights.

The IETF policies and processes regarding intellectual property and patents are outlined in [RFC3979] and [RFC4879] and at https://datatracker.ietf.org/ipr/about.

1.3. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Interface

The LMS signing algorithm is stateful; it modifies and updates the private key as a side effect of generating a signature. Once a particular value of the private key is used to sign one message, it MUST NOT be used to sign another.

The key generation algorithm takes as input an indication of the parameters for the signature system. If it is successful, it returns both a private key and a public key. Otherwise, it returns an indication of failure.

The signing algorithm takes as input the message to be signed and the current value of the private key. If successful, it returns a signature and the next value of the private key, if there is such a value. After the private key of an N-time signature system has signed N messages, the signing algorithm returns the signature and an indication that there is no next value of the private key that can be used for signing. If unsuccessful, it returns an indication of failure.

The verification algorithm takes as input the public key, a message, and a signature, and returns an indication of whether or not the signature and message pair is valid.

A message/signature pair is valid if the signature was returned by the signing algorithm upon input of the message and the private key
corresponding to the public key; otherwise, the signature and message pair is not valid with probability very close to one.

3. Notation

3.1. Data Types

Bytes and byte strings are the fundamental data types. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string with a length of three. An array of byte strings is an ordered set, indexed starting at zero, in which all strings have the same length.

Unsigned integers are converted into byte strings by representing them in network byte order. To make the number of bytes in the representation explicit, we define the functions u8str(X), u16str(X), and u32str(X), which take a non-negative integer X as input and return one, two, and four byte strings, respectively. We also make use of the function strTou32(S), which takes a four-byte string S as input and returns a non-negative integer; the identity u32str(strTou32(S)) = S holds for any four-byte string S.

3.1.1. Operators

When a and b are real numbers, mathematical operators are defined as follows:

\(^\) : a ^ b denotes the result 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 remainder of the integer division of a by b (with a and b being restricted to integers in this case)

\(+\) : a + b denotes the sum of a and b

\(-\) : a - b denotes the difference of a and b

\(\text{AND}\) : a AND b denotes the bitwise AND of the two nonnegative integers a and b (represented in binary notation)

The standard order of operations is used when evaluating arithmetic expressions.
When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation by i bit positions. Similarly, B << i denotes the logical left-shift operation.

If S and T are byte strings, then S || T denotes the concatenation of S and T. If S and T are equal length byte strings, then S AND T denotes the bitwise logical and operation.

The i-th element in an array A is denoted as A[i].

3.1.2. Functions

If r is a non-negative real number, then we define the following functions:

ceil(r) : returns the smallest integer greater than or equal to r

floor(r) : returns the largest integer less than or equal to r

lg(r) : returns the base-2 logarithm of r

3.1.3. Strings of w-bit elements

If S is a byte string, then byte(S, i) denotes its i-th byte, where the index starts at 0 at the left. Hence, byte(S, 0) is the leftmost byte of S, byte(S, 1) is the second byte at the left and (assuming S is n bytes long) byte(S, n-1) is the rightmost byte of S. In addition, bytes(S, i, j) denotes the range of bytes from the i-th to the j-th byte, inclusive. For example, if S = 0x02040608, then byte(S, 0) is 0x02 and bytes(S, 1, 2) is 0x0406.

A byte string can be considered to be a string of w-bit unsigned integers; the correspondence is defined by the function coef(S, i, w) as follows:

If S is a string, i is a positive integer, and w is a member of the set {1, 2, 4, 8}, then coef(S, i, w) is the i-th, w-bit value, if S is interpreted as a sequence of w-bit values. That is,

\[
\text{coef}(S, i, w) = (2^w - 1) \text{ AND } \left( \text{byte}(S, \text{floor}(i \times w / 8)) >> (8 - (w \times (i \% (8 / w)) + w)) \right)
\]
For example, if $S$ is the string 0x1234, then $\text{coef}(S, 7, 1)$ is 0 and $\text{coef}(S, 0, 4)$ is 1.

\[
S \text{ (represented as bits)} \\
+-----------------------------+ \\
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | \\
+-----------------------------+ \\
^ \\
| coef(S, 7, 1) \\
\]

\[
S \text{ (represented as four-bit values)} \\
+-----------------------------+ \\
| 1 | 2 | 3 | 4 | \\
+-----------------------------+ \\
^ \\
| coef(S, 0, 4) \\
\]

The return value of $\text{coef}$ is an unsigned integer. If $i$ is larger than the number of $w$-bit values in $S$, then $\text{coef}(S, i, w)$ is undefined, and an attempt to compute that value MUST raise an error.

### 3.2. Typecodes

A typecode is an unsigned integer that is associated with a particular data format. The format of the LM-OTS, LMS, and HSS signatures and public keys all begin with a typecode that indicates the precise details used in that format. These typecodes are represented as four-byte unsigned integers in network byte order; equivalently, they are XDR enumerations (see Section 3.3).

### 3.3. Notation and Formats

The signature and public key formats are formally defined using the External Data Representation (XDR) [RFC4506] in order to provide an unambiguous, machine readable definition. For clarity, we also include a private key format as well, though consistency is not needed for interoperability and an implementation MAY use any private key format. Though XDR is used, these formats are simple and easy to parse without any special tools. An illustration of the layout of data in these objects is provided below. The definitions are as follows:

```c
/* one-time signatures */
```
enum lmots_algorithm_type {
    lmots_reserved = 0,
    lmots_sha256_n32_w1 = 1,
    lmots_sha256_n32_w2 = 2,
    lmots_sha256_n32_w4 = 3,
    lmots_sha256_n32_w8 = 4
};

typedef opaque bytestring32[32];

struct lmots_signature_n32_p265 {
    bytestring32 C;
    bytestring32 y[265];
};

struct lmots_signature_n32_p133 {
    bytestring32 C;
    bytestring32 y[133];
};

struct lmots_signature_n32_p67 {
    bytestring32 C;
    bytestring32 y[67];
};

struct lmots_signature_n32_p34 {
    bytestring32 C;
    bytestring32 y[34];
};

union lmots_signature switch (lmots_algorithm_type type) {
    case lmots_sha256_n32_w1:
        lmots_signature_n32_p265 sig_n32_p265;
    case lmots_sha256_n32_w2:
        lmots_signature_n32_p133 sig_n32_p133;
    case lmots_sha256_n32_w4:
        lmots_signature_n32_p67  sig_n32_p67;
    case lmots_sha256_n32_w8:
        lmots_signature_n32_p34  sig_n32_p34;
    default:
        void;  /* error condition */
};

/* hash based signatures (hbs) */

enum lms_algorithm_type {
    lms_reserved = 0,

lms_sha256_n32_h5 = 5,
lms_sha256_n32_h10 = 6,
lms_sha256_n32_h15 = 7,
lms_sha256_n32_h20 = 8,
lms_sha256_n32_h25 = 9,
);

/* leighton-micali signatures (lms) */
union lms_path switch (lms_algorithm_type type) {
    case lms_sha256_n32_h5:
        bytestring32 path_n32_h5[5];
    case lms_sha256_n32_h10:
        bytestring32 path_n32_h10[10];
    case lms_sha256_n32_h15:
        bytestring32 path_n32_h15[15];
    case lms_sha256_n32_h20:
        bytestring32 path_n32_h20[20];
    case lms_sha256_n32_h25:
        bytestring32 path_n32_h25[25];
    default:
        void; /* error condition */
};

struct lms_signature {
    unsigned int q;
    lmots_signature lmots_sig;
    lms_path nodes;
};

struct lms_key_n32 {
    lmots_algorithm_type ots_alg_type;
    opaque I[16];
    opaque K[32];
};

union lms_public_key switch (lms_algorithm_type type) {
    case lms_sha256_n32_h5:
    case lms_sha256_n32_h10:
    case lms_sha256_n32_h15:
    case lms_sha256_n32_h20:
    case lms_sha256_n32_h25:
        lms_key_n32 z_n32;
    default:
        void; /* error condition */
};

/* hierarchical signature system (hss) */
struct hss_public_key {
    unsigned int L;
    lms_public_key pub;
};

struct signed_public_key {
    lms_signature sig;
    lms_public_key pub;
}

struct hss_signature {
    signed_public_key signed_keys<7>;
    lms_signature sig_of_message;
};

4. LM-OTS One-Time Signatures

This section defines LM-OTS signatures. The signature is used to validate the authenticity of a message by associating a secret private key with a shared public key. These are one-time signatures; each private key MUST be used at most one time to sign any given message.

As part of the signing process, a digest of the original message is computed using the cryptographic hash function H (see Section 4.1), and the resulting digest is signed.

In order to facilitate its use in an N-time signature system, the LM-OTS key generation, signing, and verification algorithms all take as input parameters I and q. The parameter I is a 16 byte string, which indicates which Merkle tree this LM-OTS is used with. The parameter q is a 32 bit integer which indicates the leaf of the Merkle tree where the OTS public key appears. These parameters are used as part of the security string, as listed in Section 7.1. When the LM-OTS signature system is used outside of an N-time signature system, the value I MAY be used to differentiate this one time signatures from others; however the value q MUST be set to the all-zero value.

4.1. Parameters

The signature system uses the parameters n and w, which are both positive integers. The algorithm description also makes use of the internal parameters p and ls, which are dependent on n and w. These parameters are summarized as follows:

- n : the number of bytes of the output of the hash function
w : the width (in bits) of the Winternitz coefficients; that is, the number of bits from the hash or checksum that is used with a single Winternitz chain. It is a member of the set \{ 1, 2, 4, 8 \}

p : the number of n-byte string elements that make up the LM-OTS signature. This is a function of n and w; the values for the defined parameter sets are listed in Table 1; it can also be computed by the algorithm given in Appendix B.

ls : the number of left-shift bits used in the checksum function Cksm (defined in Section 4.4)

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an n-byte string

For more background on the cryptographic security requirements on H, see the Section 9.

The value of n is determined by the hash function selected for use as part of the LM-OTS algorithm; the choice of this value has a strong effect on the security of the system. The parameter w determines the length of the Winternitz chains computed as a part of the OTS signature (which involve \(2^w-1\) invocations of the hash function); it has little effect on security. Increasing w will shorten the signature, but at a cost of a larger computation to generate and verify a signature. The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix B. A table illustrating various combinations of n, w, p and ls, along with the resulting signature length, is provided in Table 1.

The value of w describes a space/time trade-off; increasing the value of w will cause the signature to shrink (by decreasing the value of p) while increasing the amount of time needed to perform operations with it (generate the public key, generate and verify the signature); in general, the LM-OTS signature is \(4+n(p+1)\) bytes long, and public key generation will take \(p(2^w-1)+1\) hash computations (and signature generation and verification will take approximately half that on average).
<table>
<thead>
<tr>
<th>Parameter Set Name</th>
<th>H</th>
<th>n</th>
<th>w</th>
<th>p</th>
<th>ls</th>
<th>sig_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMOTS_SHA256_N32_W1</td>
<td>SHA256</td>
<td>32</td>
<td>1</td>
<td>265</td>
<td>7</td>
<td>8516</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W2</td>
<td>SHA256</td>
<td>32</td>
<td>2</td>
<td>133</td>
<td>6</td>
<td>4292</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W4</td>
<td>SHA256</td>
<td>32</td>
<td>4</td>
<td>67</td>
<td>4</td>
<td>2180</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W8</td>
<td>SHA256</td>
<td>32</td>
<td>8</td>
<td>34</td>
<td>0</td>
<td>1124</td>
</tr>
</tbody>
</table>

Here SHA256 denotes the SHA-256 hash function defined in NIST standard [FIPS180].

4.2. Private Key

The format of the LM-OTS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that the private key may consist of a typecode indicating the particular LM-OTS algorithm, an array x[] containing p n-byte strings, and the 16-byte string I and the 4 byte string q. This private key MUST be used to sign (at most) one message. The following algorithm shows pseudocode for generating a private key.

Algorithm 0: Generating a Private Key

1. retrieve the values of q and I (the 16-byte identifier of the LMS public/private keypair) from the LMS tree that this LM-OTS private key will be used with
2. set type to the typecode of the algorithm
3. set n and p according to the typecode and Table 1
4. compute the array x as follows:
   for (i = 0; i < p; i = i + 1) {
      set x[i] to a uniformly random n-byte string
   }
5. return u32str(type) || I || u32str(q) || x[0] || x[1] || ... || x[p-1]

An implementation MAY use a pseudorandom method to compute x[i], as suggested in [Merkle79], page 46. The details of the pseudorandom method do not affect interoperability, but the cryptographic strength...
MUST match that of the LM-OTS algorithm. Appendix A provides an example of a pseudorandom method for computing the LM-OTS private key.

4.3. Public Key

The LM-OTS public key is generated from the private key by iteratively applying the function $H$ to each individual element of $x$, for $2^w - 1$ iterations, then hashing all of the resulting values.

The public key is generated from the private key using the following algorithm, or any equivalent process.

Algorithm 1: Generating a One Time Signature Public Key From a Private Key

1. set type to the typecode of the algorithm
2. set the integers $n$, $p$, and $w$ according to the typecode and Table 1
3. determine $x$, $I$ and $q$ from the private key
4. compute the string $K$ as follows:
   for ($i = 0; i < p; i = i + 1$) {
     $\text{tmp} = x[i]$
     for ($j = 0; j < 2^w - 1; j = j + 1$) {
       $\text{tmp} = H(I || u32str(q) || u16str(i) || u8str(j) || \text{tmp})$
     }
     $y[i] = \text{tmp}$
   }
   $K = H(I || u32str(q) || u16str(D_PBLCT) || y[0] || ... || y[p-1])$
5. return $\text{u32str}(\text{type}) || I || \text{u32str}(q) || K$

where $D_PBLCT$ is the fixed two byte value 0x8080, which is used to distinguish the last hash from every other hash in this system.

The public key is the value returned by Algorithm 1.

4.4. Checksum

A checksum is used to ensure that any forgery attempt that manipulates the elements of an existing signature will be detected. This checksum is needed because an attacker can freely advance any of the Winternitz chains. That is, if this checksum were not present, then an attacker who could find a hash that has every digit larger than the valid hash could replace it (and adjust the Winternitz chains). The security property that it provides is detailed in
Section 9. The checksum function Cksm is defined as follows, where S denotes the n-byte string that is input to that function, and the value sum is a 16-bit unsigned integer:

Algorithm 2: Checksum Calculation

```plaintext
sum = 0
for ( i = 0; i < (n*8/w); i = i + 1 ) {
    sum = sum + (2^w - 1) - coef(S, i, w)
}
return (sum << ls)
```

ls is the parameter that shifts the significant bits of the checksum into the positions that will actually be used by the coef function when encoding the digits of the checksum. The actual ls parameter is a function of the n and w parameters; the values for the currently defined parameter sets is shown in table 1. It is calculated by the algorithm given in Appendix B.

Because of the left-shift operation, the rightmost bits of the result of Cksm will often be zeros. Due to the value of p, these bits will not be used during signature generation or verification.

4.5. Signature Generation

The LM-OTS signature of a message is generated by first prepending the LMS key identifier I, the LMS leaf identifier q, the value D_MESG (0x8181) and the randomizer C to the message, then computing the hash, and then concatenating the checksum of the hash to the hash itself, then considering the resulting value as a sequence of w-bit values, and using each of the w-bit values to determine the number of times to apply the function H to the corresponding element of the private key. The outputs of the function H are concatenated together and returned as the signature. The pseudocode for this procedure is shown below.
Algorithm 3: Generating a One Time Signature From a Private Key and a Message

1. set type to the typecode of the algorithm
2. set n, p, and w according to the typecode and Table 1
3. determine x, I and q from the private key
4. set C to a uniformly random n-byte string
5. compute the array y as follows:
   \[ Q = H(I \| u32str(q) \| u16str(D_MESG) \| C \| message) \]
   for (i = 0; i < p; i = i + 1) {
     a = coef(Q \| Cksm(Q), i, w)
     tmp = x[i]
     for (j = 0; j < a; j = j + 1) {
       tmp = H(I \| u32str(q) \| u16str(i) \| u8str(j) \| tmp)
     }
     y[i] = tmp
   }
6. return u32str(type) \| C \| y[0] \| ... \| y[p-1]

Note that this algorithm results in a signature whose elements are intermediate values of the elements computed by the public key algorithm in Section 4.3.

The signature is the string returned by Algorithm 3. Section 3.3 specifies the typecode and more formally defines the encoding and decoding of the string.

4.6. Signature Verification

In order to verify a message with its signature (an array of n-byte strings, denoted as y), the receiver must "complete" the chain of iterations of H using the w-bit coefficients of the string resulting from the concatenation of the message hash and its checksum. This computation should result in a value that matches the provided public key.
Algorithm 4a: Verifying a Signature and Message Using a Public Key

1. if the public key is not at least four bytes long, return INVALID

2. parse pubtype, I, q, and K from the public key as follows:
   a. pubtype = strTou32(first 4 bytes of public key)
   b. set n according to the pubkey and Table 1; if the public key is not exactly 24 + n bytes long, return INVALID
   c. I = next 16 bytes of public key
   d. q = strTou32(next 4 bytes of public key)
   e. K = next n bytes of public key

3. compute the public key candidate Kc from the signature, message, pubtype and the identifiers I and q obtained from the public key, using Algorithm 4b. If Algorithm 4b returns INVALID, then return INVALID.

4. if Kc is equal to K, return VALID; otherwise, return INVALID
Algorithm 4b: Computing a Public Key Candidate $K_c$ from a Signature, Message, Signature Typecode $pubtype$, and identifiers $I$, $q$

1. if the signature is not at least four bytes long, return INVALID

2. parse sigtype, $C$, and $y$ from the signature as follows:
   a. sigtype = strTou32(first 4 bytes of signature)
   b. if sigtype is not equal to pubtype, return INVALID
   c. set $n$ and $p$ according to the pubtype and Table 1; if the signature is not exactly $4 + n \times (p+1)$ bytes long, return INVALID
   d. $C = \text{next } n \text{ bytes of signature}$
   e. $y[0] = \text{next } n \text{ bytes of signature}$
      $y[1] = \text{next } n \text{ bytes of signature}$
      $\ldots$
      $y[p-1] = \text{next } n \text{ bytes of signature}$

3. compute the string $K_c$ as follows
   
   $Q = H(I || \text{u32str}(q) || \text{u16str}(\text{D_MESG}) || C || \text{message})$

   for ( $i = 0; i < p; i = i + 1$ ) {
      $a = \text{coef}(Q || Cksm(Q), i, w)$
      $tmp = y[i]$
      for ( $j = a; j < 2^w - 1; j = j + 1$ ) {
         $tmp = H(I || \text{u32str}(q) || \text{u16str}(i) || \text{u8str}(j) || tmp)$
      }
      $z[i] = tmp$
   }

   $K_c = H(I || \text{u32str}(q) || \text{u16str}(\text{D_PBLC}) ||
      z[0] || z[1] || \ldots || z[p-1])$

4. return $K_c$

5. Leighton-Micali Signatures

The Leighton–Micali Signature (LMS) method can sign a potentially large but fixed number of messages. An LMS system uses two cryptographic components: a one-time signature method and a hash function. Each LMS public/private key pair is associated with a perfect binary tree, each node of which contains an $m$-byte value, where $m$ is the output length of the hash function. Each leaf of the tree contains the value of the public key of an LM-OTS public/private key pair. The value contained by the root of the tree is the LMS public key. Each interior node is computed by applying the hash function to the concatenation of the values of its children nodes.
Each node of the tree is associated with a node number, an unsigned integer that is denoted as node_num in the algorithms below, which is computed as follows. The root node has node number 1; for each node with node number \( N < 2^h \) (where \( h \) is the height of the tree), its left child has node number \( 2N \), while its right child has node number \( 2N+1 \). The result of this is that each node within the tree will have a unique node number, and the leaves will have node numbers \( 2^h, (2^h)+1, (2^h)+2, \ldots, (2^h)+(2^h)-1 \). In general, the \( j \)-th node at level \( i \) has node number \( 2^i + j \). The node number can conveniently be computed when it is needed in the LMS algorithms, as described in those algorithms.

5.1. Parameters

An LMS system has the following parameters:

\[
\begin{align*}
  h & : \text{the height of the tree, and} \\
  m & : \text{the number of bytes associated with each node.}
\end{align*}
\]

\( H \) : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an \( m \)-byte string.

There are \( 2^h \) leaves in the tree.

The overall strength of the LMS signatures is governed by the weaker of the hash function used within the LM-OTS and the hash function used within the LMS system. In order to minimize the risk, these two hash functions SHOULD be the same (so that an attacker could not take advantage of the weaker hash function choice).

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Name} & \text{H} & \text{m} & \text{h} \\
\hline
\text{LMS_SHA256_M32_H5} & \text{SHA256} & 32 & 5 \\
\text{LMS_SHA256_M32_H10} & \text{SHA256} & 32 & 10 \\
\text{LMS_SHA256_M32_H15} & \text{SHA256} & 32 & 15 \\
\text{LMS_SHA256_M32_H20} & \text{SHA256} & 32 & 20 \\
\text{LMS_SHA256_M32_H25} & \text{SHA256} & 32 & 25 \\
\hline
\end{array}
\]

Table 2
5.2. LMS Private Key

The format of the LMS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that it may consist of an array OTS_PRIV[] of $2^h$ LM-OTS private keys, and the leaf number q of the next LM-OTS private key that has not yet been used. The q-th element of OTS_PRIV[] is generated using Algorithm 0 with the identifiers I, q. The leaf number q is initialized to zero when the LMS private key is created. The process is as follows:

Algorithm 5: Computing an LMS Private Key.

1. determine h and m from the typecode and Table 2.
2. set I to a uniformly random 16-byte string
3. compute the array OTS_PRIV[] as follows:
   for ( q = 0; q < $2^h$; q = q + 1) {
     OTS_PRIV[q] = LM-OTS private key with identifiers I, q
   }
4. q = 0

An LMS private key MAY be generated pseudorandomly from a secret value, in which case the secret value MUST be at least m bytes long, be uniformly random, and MUST NOT be used for any other purpose than the generation of the LMS private key. The details of how this process is done do not affect interoperability; that is, the public key verification operation is independent of these details. Appendix A provides an example of a pseudorandom method for computing an LMS private key.

The signature generation logic uses q as the next leaf to use, hence step 4 starts it off at the left-most one. Because the signature process increments q after the signature operation, the first signature will have q=0.

5.3. LMS Public Key

An LMS public key is defined as follows, where we denote the public key final hash value (namely, the K value computed in Algorithm 1) associated with the i-th LM-OTS private key as OTS_PUB_HASH[i], with i ranging from 0 to $(2^h)-1$. Each instance of an LMS public/private key pair is associated with a balanced binary tree, and the nodes of that tree are indexed from 1 to $2^{(h+1)}-1$. Each node is associated with an m-byte string, and the string for the r-th node is denoted as T[r] and is defined as
if \( r \geq 2^h \):
    \[ H(I \| u32str(r) \| u16str(D\_LEAF) \| OTS\_PUB\_HASH[r-2^h]) \]
else
    \[ H(I \| u32str(r) \| u16str(D\_INTR) \| T[2*r] \| T[2*r+1]) \]

where D\_LEAF is the fixed two byte value 0x8282, and D\_INTR is the fixed two byte value 0x8383, both of which are used to distinguish this hash from every other hash in this system.

When we have \( r \geq 2^h \), then we are processing a leaf node (and thus hashing only a single LM-OTS public key). When we have \( r < 2^h \), then we are processing an internal node, that is, a node with two child nodes that we need to combine.

The LMS public key is the string

\[ u32str(type) \| u32str(otstype) \| I \| T[1] \]

Section 3.3 specifies the format of the type variable. The value otstype is the parameter set for the LM-OTS public/private keypairs used. The value I is the private key identifier, and is the value used for all computations for the same LMS tree. The value T[1] can be computed via recursive application of the above equation, or by any equivalent method. An iterative procedure is outlined in Appendix C.

### 5.4. LMS Signature

An LMS signature consists of

- the number \( q \) of the leaf associated with the LM-OTS signature, as a four-byte unsigned integer in network byte order,
- an LM-OTS signature,
- a typecode indicating the particular LMS algorithm,
- an array of \( h \) m-byte values that is associated with the path through the tree from the leaf associated with the LM-OTS signature to the root.

Symbolically, the signature can be represented as

\[ u32str(q) \| lmots\_signature \| u32str(type) \| path[0] \| path[1] \| path[2] \| ... \| path[h-1] \]

Section 3.3 specifies the typecode and more formally defines the format. The array for a tree with height \( h \) will have \( h \) values and...
contains the values of the siblings of (that is, is adjacent to) the nodes on the path from the leaf to the root, where the sibling to node A is the other node which shares node A’s parent. In the signature, 0 is counted from the bottom level of the tree, and so path[0] is the value of the node adjacent to leaf node q; path[1] is the second level node that is adjacent to leaf node q’s parent, and so up the tree until we get to path[h-1], which is the value of the next-to-the-top level node that leaf node q does not reside in.

Below is a simple example of the authentication path for h=3 and q=2. The leaf marked OTS is the one time signature which is used to sign the actual message. The nodes on the path from the OTS public key to the root are marked with a *, while the nodes that are used within the path array are marked with a **. The values in the path array are those nodes which are siblings of the nodes on the path; path[0] is the leaf** node that is adjacent to the OTS public key (which is the start of the path); path[1] is the T[4]** node which is the sibling of the second node T[5]* on the path, and path[2] is the T[3]** node which is the sibling of the third node T[2]* on the path.

```
  Root
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
  T[2]*                        T[3]**
  |                           |
  ----------------------------|
  |                        |
  T[4]**                   T[5]*
  |                        |
  ---------                -------
  |                   |
  leaf    leaf          OTS  leaf**    leaf    leaf    leaf    leaf
```

The idea behind this authentication path is that it allows us to validate the OTS hash with using h path array values and hash computations. What the verifier does is recompute the hashes up the path; first, he hashes the given OTS and path[0] value, giving a tentative T[5]’ value. Then, he hashes his path[1] and tentative T[5]’ value to get a tentative T[2]’ value. Then, he hashes that and the path[2] value to get a tentative Root’ value. If that value is the known public key of the Merkle tree, then we can assume that the value T[2]’ he got was the correct T[2] value in the original tree, and so the T[5]’ value he got was the correct T[5] value in the original tree, and so the OTS public key is the same as in the original, and hence is correct.
5.4.1. LMS Signature Generation

To compute the LMS signature of a message with an LMS private key, the signer first computes the LM-OTS signature of the message using the leaf number of the next unused LM-OTS private key. The leaf number q in the signature is set to the leaf number of the LMS private key that was used in the signature. Before releasing the signature, the leaf number q in the LMS private key MUST be incremented, to prevent the LM-OTS private key from being used again. If the LMS private key is maintained in nonvolatile memory, then the implementation MUST ensure that the incremented value has been stored before releasing the signature. The issue this tries to prevent is a scenario where a) we generate a signature, using one LM-OTS private key, and release it to the application, b) before we update the nonvolatile memory, we crash, and c) we reboot, and generate a second signature using the same LM-OTS private key; with two different signatures using the same LM-OTS private key, someone could potentially generate a forged signature of a third message.

The array of node values in the signature MAY be computed in any way. There are many potential time/storage tradeoffs that can be applied. The fastest alternative is to store all of the nodes of the tree and set the array in the signature by copying them; pseudocode to do so appears in Appendix D. The least storage intensive alternative is to recompute all of the nodes for each signature. Note that the details of this procedure are not important for interoperability; it is not necessary to know any of these details in order to perform the signature verification operation. The internal nodes of the tree need not be kept secret, and thus a node-caching scheme that stores only internal nodes can sidestep the need for strong protections.

Several useful time/storage tradeoffs are described in the ‘Small-Memory LM Schemes’ section of [USPTO5432852].

5.4.2. LMS Signature Verification

An LMS signature is verified by first using the LM-OTS signature verification algorithm (Algorithm 4b) to compute the LM-OTS public key from the LM-OTS signature and the message. The value of that public key is then assigned to the associated leaf of the LMS tree, then the root of the tree is computed from the leaf value and the array path[] as described in Algorithm 6 below. If the root value matches the public key, then the signature is valid; otherwise, the signature fails.
Algorithm 6: LMS Signature Verification

1. if the public key is not at least eight bytes long, return INVALID

2. parse pubtype, I, and T[1] from the public key as follows:
   a. pubtype = strTou32(first 4 bytes of public key)
   b. ots_typecode = strTou32(next 4 bytes of public key)
   c. set m according to pubtype, based on Table 2
   d. if the public key is not exactly 24 + m bytes long, return INVALID
   e. I = next 16 bytes of the public key
   f. T[1] = next m bytes of the public key

3. compute the LMS Public Key Candidate Tc from the signature, message, identifier, pubtype and ots_typecode using Algorithm 6a.

4. if Tc is equal to T[1], return VALID; otherwise, return INVALID

Algorithm 6a: Computing an LMS Public Key Candidate from a Signature, Message, Identifier, and algorithm typecode

1. if the signature is not at least eight bytes long, return INVALID

2. parse sigtype, q, lmots_signature, and path from the signature as follows:
   a. q = strTou32(first 4 bytes of signature)
   b. otssigtype = strTou32(next 4 bytes of signature)
   c. if otssigtype is not the OTS typecode from the public key, return INVALID
   d. set n, p according to otssigtype and Table 1; if the signature is not at least 12 + n * (p + 1) bytes long, return INVALID
   e. lmots_signature = bytes 4 through 7 + n * (p + 1) of signature
   f. sigtype = strTou32(bytes 8 + n * (p + 1)) through
11 + n * (p + 1) of signature

f. if sigtype is not the LM typecode from the public key, return INVALID

g. set m, h according to sigtype and Table 2

h. if q >= 2^h or the signature is not exactly 12 + n * (p + 1) + m * h bytes long, return INVALID

i. set path as follows:
   path[0] = next m bytes of signature
   path[1] = next m bytes of signature
   ...
   path[h-1] = next m bytes of signature

3. Kc = candidate public key computed by applying Algorithm 4b to the signature lmots_signature, the message, and the identifiers I, q

4. compute the candidate LMS root value Tc as follows:
   node_num = 2^h + q
   tmp = H(I || u32str(node_num) || u16str(D_LEAF) || Kc)
   i = 0
   while (node_num > 1) {
       if (node_num is odd):
           tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||path[i]||tmp)
       else:
           tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||tmp||path[i])
       node_num = node_num/2
       i = i + 1
   }
   Tc = tmp

5. return Tc

6. Hierarchical signatures

In scenarios where it is necessary to minimize the time taken by the public key generation process, a Hierarchical N-time Signature System (HSS) can be used. This hierarchical scheme, which we describe in this section, uses the LMS scheme as a component. In HSS, we have a sequence of L LMS trees, where the public key for the first LMS tree is included in the public key of the HSS system, and where each LMS private key signs the next LMS public key, and where the last LMS private key signs the actual message. For example, if we have a three level hierarchy (L=3), then to sign a message, we would have:
The first LMS private key (level 0) signs a level 1 LMS public key.

The second LMS private key (level 1) signs a level 2 LMS public key.

The third LMS private key (level 2) signs the message.

The root of the level 0 LMS tree is contained in the HSS public key.

To verify the LMS signature, we would verify all the signatures:

We would verify that the level 1 LMS public key is correctly signed by the level 0 signature.

We would verify that the level 2 LMS public key is correctly signed by the level 1 signature.

We would verify that the message is correctly signed by the level 2 signature.

We would accept the HSS signature only if all the signatures validated.

During the signature generation process, we sign messages with the lowest (level L-1) LMS tree. Once we have used all the leafs in that tree to sign messages, we would discard it, generate a fresh LMS tree, and sign it with the next (level L-2) LMS tree (and when that is used up, recursively generate and sign a fresh level L-2 LMS tree).

HSS, in essence, utilizes a tree of LMS trees. There is a single LMS tree at level 0 (the root). Each LMS tree (actually, the private key corresponding to the LMS tree) at level i is used to sign 2^h objects (where h is the height of trees at level i). If i < L-1, then each object will be another LMS tree (actually, the public key) at level i+1; if i = L-1, we’ve reached the bottom of the HSS tree, and so each object is a message from the application. The HSS public key contains the public key of the LMS tree at the root, and an HSS signature is associated with a path from the root of the HSS tree to the leaf.

Compared to LMS, HSS has a much reduced public key generation time, as only the root tree needs to be generated prior to the distribution of the HSS public key. For example, a L=3 tree (with h=10 at each level) would have 1 level 0 LMS tree, 2^10 level 1 LMS trees (with each such level 1 public key signed by one of the 1024 level 0 OTS public keys), and 2^20 level 2 LMS trees. Only 1024 OTS public keys
need to be computed to generate the HSS public key (as you need to compute only the level 0 LMS tree to compute that value; you can, of course, decide to compute the initial level 1 and level 2 LMS trees). And, the $2^{20}$ level 2 LMS trees can jointly sign a total of over a billion messages. In contrast, a single LMS tree that could sign a billion messages would require a billion OTS public keys to be computed first (even if $h=30$ were allowed in a supported parameter set).

Each LMS tree within the hierarchy is associated with a distinct LMS public key, private key, signature, and identifier. The number of levels is denoted $L$, and is between one and eight, inclusive. The following notation is used, where $i$ is an integer between 0 and $L-1$ inclusive, and the root of the hierarchy is level 0:

- $prv[i]$ is the current LMS private key of the $i$-th level.
- $pub[i]$ is the current LMS public key of the $i$-th level, as described in Section 5.3.
- $sig[i]$ is the LMS signature of public key $pub[i+1]$ generated using the private key $prv[i]$.

It is expected that the above arrays are maintained for the course of the HSS key. The contents of the $prv[]$ array MUST be kept private; the $pub[]$ and $sig[]$ array may be revealed, should the implementation find that convenient.

In this section, we say that an $N$-time private key is exhausted when it has generated $N$ signatures, and thus it can no longer be used for signing.

For $i > 0$, the values $prv[i]$, $pub[i]$ and (for all values of $i$) $sig[i]$ will be updated over time, as private keys are exhausted, and replaced by newer keys.

When these keys pairs are updated (or initially generated before the first message is signed), then the LMS key generation processes outlined in sections Section 5.2 and Section 5.3 are performed. If the generated key pairs are for level $i$ of the HSS hierarchy, then we store the public key in $pub[i]$ and the private key in $prv[i]$. In addition, if $i > 0$, then we sign the generated public key with the LMS private key at level $i-1$, placing the signature into $sig[i-1]$. When the LMS key pair are generated, the key pair and the corresponding identifier MUST be generated independently of all other keypairs.
HSS allows \( L=1 \), in which case the HSS public key and signature formats are essentially the LMS public key and signature formats, prepended by a fixed field. Since HSS with \( L=1 \) has very little overhead compared to LMS, all implementations MUST support HSS in order to maximize interoperability.

We specifically allow different LMS levels to use different parameter sets. For example, the 0-th LMS public key (the root) may use the LMS_SHA256_M32_H15 parameter set, while the 1-th public key may use LMS_SHA256_M32_H10. There are practical reasons to allow this; for one, the signer may decide to store parts of the 0-th LMS tree (that it needs to construct while computing the public key) to accelerate later operations. As the 0-th tree is never updated, these internal nodes will never need to be recomputed. In addition, during the signature generation operation, almost all the operations involved with updating the authentication path occurs with the bottom (\( L-1 \)th) LMS public key; hence it may be useful to make the tree that implements that to be shorter.

A close reading of the HSS verification pseudocode would show that it would allow the parameters of the non-top LMS public keys to change over time; for example, the signer might initially have the 1-th LMS public key to use LMS_SHA256_M32_H10, but when that tree is exhausted, the signer might replace it with LMS_SHA256_M32_H15 LMS public key. While this would work with the example verification pseudocode, the signer MUST NOT change the parameter sets for a specific level. This prohibition is to support verifiers that may keep state over the course of several signature verifications.

6.1. Key Generation

The public key of the HSS scheme consists of the number of levels \( L \), followed by \( \text{pub}[0] \), the public key of the top level.

The HSS private key consists of \( \text{prv}[0], \ldots, \text{prv}[L-1] \), along with the associated \( \text{pub}[0], \ldots, \text{pub}[L-1] \) and \( \text{sig}[0], \ldots, \text{sig}[L-2] \) values. As stated earlier, the values of the \( \text{pub}[] \) and \( \text{sig}[] \) arrays need not be kept secret, but may be revealed. The value of \( \text{pub}[0] \) does not change (and, except for the index \( q \), the value of \( \text{prv}[0] \) need not change), though the values of \( \text{pub}[i] \) and \( \text{prv}[i] \) are dynamic for \( i > 0 \), and are changed by the signature generation algorithm.

During the key generation, the public and private keys are initialized. Here is some pseudocode that explains the key generation logic.
Algorithm 7: Generating an HSS keypair

1. generate an LMS key pair, as specified in sections 5.2 and 5.3, placing the private key into priv[0], and the public key into pub[0]

2. for i = 1 to L-1 do {
   generate an LMS key pair, placing the private key into priv[i] and the public key into pub[i]
   sig[i-1] = lms_signature( pub[i], priv[i-1] )
}

3. return u32str(L) || pub[0] as the public key, and the priv[], pub[] and sig[] arrays as the private key

In the above algorithm, each LMS public/private keypair generated MUST be generated independently.

Note that the value of the public key does not depend on the execution of step 2. As a result, an implementation may decide to delay step 2 until later, for example, during the initial signature generation operation.

6.2. Signature Generation

To sign a message using an HSS keypair, the following steps are performed:

If prv[L-1] is exhausted, then determine the smallest integer d such that all of the private keys prv[d], prv[d+1], ..., prv[L-1] are exhausted. If d is equal to zero, then the HSS key pair is exhausted, and it MUST NOT generate any more signatures. Otherwise, the key pairs for levels d through L-1 must be regenerated during the signature generation process, as follows. For i from d to L-1, a new LMS public and private key pair with a new identifier is generated, pub[i] and prv[i] are set to those values, then the public key pub[i] is signed with prv[i-1], and sig[i-1] is set to the resulting value.

The message is signed with prv[L-1], and the value sig[L-1] is set to that result.

The value of the HSS signature is set as follows. We let signed_pub_key denote an array of octet strings, where

signed_pub_key[i] = sig[i] || pub[i+1], for i between 0 and Nspk-1, inclusive, where Nspk = L-1 denotes the number of signed public
keys. Then the HSS signature is u32str(Nspk) || signed_pub_key[0] || ... || signed_pub_key[Nspk-1] || sig[Nspk].

Note that the number of signed_pub_key elements in the signature is indicated by the value Nspk that appears in the initial four bytes of the signature.

Here is some pseudocode of the above logic

Algorithm 8: Generating an HSS signature

1. If the message-signing key prv[L-1] is exhausted, regenerate that key pair, together with any parent key pairs that might be necessary.
   If the root key pair is exhausted, then the HSS key pair is exhausted and it MUST NOT generate any more signatures.

   d = L
   while (prv[d-1].q == 2^(prv[d-1].h)) {
       d = d - 1
       if (d == 0)
           return FAILURE
   }
   while (d < L) {
       create lms keypair pub[d], prv[d]
       sig[d-1] = lms_signature( pub[d], prv[d-1] )
       d = d + 1
   }

2. sign the message
   sig[L-1] = lms_signature( msg, prv[L-1] )

3. Create the list of signed public keys
   i = 0;
   while (i < L-1) {
       signed_pub_key[i] = sig[i] || pub[i+1]
       i = i + 1
   }

4. return u32str(L-1) || signed_pub_key[0] || ... || signed_pub_key[L-2] || sig[L-1]

In the specific case of L=1, the format of an HSS signature is

u32str(0) || sig[0]

In the general case, the format of an HSS signature is
6.3. Signature Verification

To verify a signature $S$ and message using the public key $pub$, the following steps are performed:

The signature $S$ is parsed into its components as follows:

$\text{Nspk} = \text{strTou32}(\text{first four bytes of } S)$

\begin{verbatim}
if $\text{Nspk} + 1$ is not equal to the number of levels $L$ in $pub$:
    return INVALID
for ($i = 0; i < \text{Nspk}; i = i + 1$) {
    siglist[$i$] = next LMS signature parsed from $S$
    publist[$i$] = next LMS public key parsed from $S$
}
\end{verbatim}

siglist[\text{Nspk}] = next LMS signature parsed from $S$

key = $pub$

\begin{verbatim}
for ($i = 0; i < \text{Nspk}; i = i + 1$) {
    sig = siglist[$i$]
    msg = publist[$i$]
    if (lms_verify(msg, key, sig) != VALID):
        return INVALID
    key = msg
}
\end{verbatim}

return lms_verify(message, key, siglist[\text{Nspk}])

Since the length of an LMS signature cannot be known without parsing it, the HSS signature verification algorithm makes use of an LMS signature parsing routine that takes as input a string consisting of an LMS signature with an arbitrary string appended to it, and returns both the LMS signature and the appended string. The latter is passed on for further processing.

6.4. Parameter Set Recommendations

As for guidance as to the number of LMS level, and the size of each, any discussion of performance is implementation specific. In general, the sole drawback for a single LMS tree is the time it takes
to generate the public key; as every LM-OTS public key needs to be
generated, the time this takes can be substantial. For a two level
tree, only the top level LMS tree and the initial bottom level LMS
tree needs to be generated initially (before the first signature is
generated); this will in general be significantly quicker.

To give a general idea on the trade-offs available, we include some
measurements taken with the github.com/cisco/hash-sigs LMS
implementation, taken on a 3.3 GHz Xeon processor, with threading
enabled. We tried various parameter sets, all with \( W=8 \) (which
minimizes signature size, while increasing time). These are here to
give a guideline as to what’s possible; for the computational time,
your mileage may vary, depending on the computing resources you have.
The machine these tests were performed on does not have the SHA-256
extensions; you could possibly do significantly better.

<table>
<thead>
<tr>
<th>ParmSet</th>
<th>KeyGenTime</th>
<th>SigSize</th>
<th>KeyLifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6 sec</td>
<td>1616</td>
<td>30 seconds</td>
</tr>
<tr>
<td>20</td>
<td>3 min</td>
<td>1776</td>
<td>16 minutes</td>
</tr>
<tr>
<td>25</td>
<td>1.5 hour</td>
<td>1936</td>
<td>9 hours</td>
</tr>
<tr>
<td>15/10</td>
<td>6 sec</td>
<td>3172</td>
<td>9 hours</td>
</tr>
<tr>
<td>15/15</td>
<td>6 sec</td>
<td>3332</td>
<td>12 days</td>
</tr>
<tr>
<td>20/10</td>
<td>3 min</td>
<td>3332</td>
<td>12 days</td>
</tr>
<tr>
<td>20/15</td>
<td>3 min</td>
<td>3492</td>
<td>1 year</td>
</tr>
<tr>
<td>25/10</td>
<td>1.5 hour</td>
<td>3492</td>
<td>1 year</td>
</tr>
<tr>
<td>25/15</td>
<td>1.5 hour</td>
<td>3652</td>
<td>34 years</td>
</tr>
</tbody>
</table>

Table 3

ParmSet: this is the height of the Merkle tree(s); parameter sets
listed as a single integer have \( L=1 \), and consist a single Merkle tree
of that height; parameter sets with \( L=2 \) are listed as \( x/y \), with \( x \)
being the height of the top level Merkle tree, and \( y \) being the bottom
level.

KeyGenTime: the measured key generation time; that is, the time
needed to generate the public private key pair.
SigSize: the size of a signature (in bytes)

KeyLifetime: the lifetime of a key, assuming we generated 1000 signatures per second. In practice, we’re not likely to get anywhere close to 1000 signatures per second sustained; if you have a more appropriate figure for your scenario, this column is pretty easy to recompute.

As for signature generation or verification times, those are moderately insensitive to the above parameter settings (except for the Winternitz setting, and the number of Merkle trees for verification). Tests on the same machine (without multithreading) gave approximately 4msec to sign a short message, 2.6msec to verify; these tests used a two level ParmSet; a single level would approximately halve the verification time. All times can be significantly improved (by perhaps a factor of 8) by using a parameter set with W=4; however that also about doubles the signature size.

7. Rationale

The goal of this note is to describe the LM-OTS, LMS and HSS algorithms following the original references and present the modern security analysis of those algorithms. Other signature methods are out of scope and may be interesting follow-on work.

We adopt the techniques described by Leighton and Micali to mitigate attacks that amortize their work over multiple invocations of the hash function.

The values taken by the identifier I across different LMS public/private key pairs are chosen randomly in order to improve security. The analysis of this method in [Fluhrer17] shows that we do not need uniqueness to ensure security; we do need to ensure that we don’t have a large number of private keys that use the same I value. By randomly selecting 16 byte I values, the chance that, out of 2^64 private keys, 4 or more of them will use the same I value is negligible (that is, has probability less than 2^-128).

The reason 16 bytes I values were selected was to optimize the Winternitz hash chain operation. With the current settings, the value being hashed is exactly 55 bytes long (for a 32 byte hash function), which SHA-256 can hash in a single hash compression operation. Other hash functions may be used in future specifications; all the ones that we will be likely to support (SHA-512/256 and the various SHA-3 hashes) would work well with a 16-byte I value.
The signature and public key formats are designed so that they are relatively easy to parse. Each format starts with a 32-bit enumeration value that indicates the details of the signature algorithm and provides all of the information that is needed in order to parse the format.

The Checksum Section 4.4 is calculated using a non-negative integer "sum", whose width was chosen to be an integer number of w-bit fields such that it is capable of holding the difference of the total possible number of applications of the function H as defined in the signing algorithm of Section 4.5 and the total actual number. In the case that the number of times H is applied is 0, the sum is \((2^w - 1) \times (8n/w)\). Thus for the purposes of this document, which describes signature methods based on \(H = SHA256\) \((n = 32\text{ bytes})\) and \(w = \{1, 2, 4, 8\}\), the sum variable is a 16-bit non-negative integer for all combinations of \(n\) and \(w\). The calculation uses the parameter \(ls\) defined in Section 4.1 and calculated in Appendix B, which indicates the number of bits used in the left-shift operation.

7.1. Security String

To improve security against attacks that amortize their effort against multiple invocations of the hash function, Leighton and Micali introduce a "security string" that is distinct for each invocation of that function. Whenever this process computes a hash, the string being hashed will start with a string formed from the below fields. These fields will appear in fixed locations in the value we compute the hash of, and so we list where in the hash these fields would be present. These fields that make up this string are:

I - a 16-byte identifier for the LMS public/private key pair. It MUST be chosen uniformly at random, or via a pseudorandom process, at the time that a key pair is generated, in order to minimize the probability that any specific value of I be used for a large number of different LMS private keys. This is always bytes 0-15 of the value being hashed.

r - in the LMS N-time signature scheme, the node number r associated with a particular node of a hash tree is used as an input to the hash used to compute that node. This value is represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

q - in the LMS N-time signature scheme, each LM-OTS signature is associated with the leaf of a hash tree, and q is set to the leaf number. This ensures that a distinct value of q is used for each distinct LM-OTS public/private key pair. This value is
represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

D - a domain separation parameter, which is a two byte identifier that takes on different values in the different contexts in which the hash function is invoked. D occurs in bytes 20, 21 of the value being hashed and takes on the following values:

- D_PBLC = 0x8080 when computing the hash of all of the iterates in the LM-OTS algorithm
- D_MESG = 0x8181 when computing the hash of the message in the LM-OTS algorithms
- D_LEAF = 0x8282 when computing the hash of the leaf of an LMS tree
- D_INTR = 0x8383 when computing the hash of an interior node of an LMS tree

i - a value between 0 and 264; this is used in the LM-OTS scheme, when either computing the iterations of the Winternitz chain, or when using the suggested LM-OTS private key generation process. It is represented as a 16-bit (two-byte) unsigned integer in network byte order. If present, it occurs at bytes 20, 21 of the value being hashed.

j - in the LM-OTS scheme, j is the iteration number used when the private key element is being iteratively hashed. It is represented as an 8-bit (one byte) unsigned integer and is present if i is a value between 0 and 264. If present, it occurs at bytes 22 to 21+n of the value being hashed.

C - an n-byte randomizer that is included with the message whenever it is being hashed to improve security. C MUST be chosen uniformly at random, or via a pseudorandom process. It is present if D=D_MESG, and it occurs at bytes 22 to 21+n of the value being hashed.

8. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create two registries: one for OTS signatures, which includes all of the LM-OTS signatures as defined in Section 4, and one for Leighton-Micali Signatures, as defined in Section 5.
Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient detail that interoperability between independent implementations is possible. IANA MUST verify that all applications for additions to these registries have first been reviewed by the IRTF Crypto Forum Research Group (CFRG).

Each entry in the registry contains the following elements:

- a short name, such as "LMS_SHA256_M32_H10",
- a positive number, and
- a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

The numbers between 0xDDDDDDDD (decimal 3,722,304,989) and 0xFFFFFFFF (decimal 4,294,967,295) inclusive, will not be assigned by IANA, and are reserved for private use; no attempt will be made to prevent multiple sites from using the same value in different (and incompatible) ways [RFC2434].

The LM-OTS registry is as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Numeric Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td>0x00000000</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W1</td>
<td>Section 4</td>
<td>0x00000001</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W2</td>
<td>Section 4</td>
<td>0x00000002</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W4</td>
<td>Section 4</td>
<td>0x00000003</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W8</td>
<td>Section 4</td>
<td>0x00000004</td>
</tr>
</tbody>
</table>

Table 4

The LMS registry is as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Numeric Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
An IANA registration of a signature system does not constitute an endorsement of that system or its security.

The LM-OTS and the LMS registries currently occupy a disjoint set of values. This coincidence is a historical accident; the correctness of the system does not depend on this. IANA is not required to maintain this situation.

9. Security Considerations

The hash function H MUST have second preimage resistance: it must be computationally infeasible for an attacker that is given one message M to be able to find a second message M' such that H(M) = H(M').

The security goal of a signature system is to prevent forgeries. A successful forgery occurs when an attacker who does not know the private key associated with a public key can find a message (distinct from all previously signed ones) and signature that is valid with that public key (that is, the Signature Verification algorithm applied to that signature and message and public key will return VALID). Such an attacker, in the strongest case, may have the ability to forge valid signatures for an arbitrary number of other messages.

LMS is provably secure in the random oracle model, where the hash compression function is considered the random oracle, as shown by [Fluhrer17]. Corollary 1 of that paper states:

If we have no more than 2^64 randomly chosen LMS private keys, allow the attacker access to a signing oracle and a SHA-256 hash
compression oracle, and allow a maximum of $2^{120}$ hash compression computations, then the probability of an attacker being able to generate a single forgery against any of those LMS keys is less than $2^{-129}$.

Many of the objects within the public key and the signature start with a typecode. A verifier MUST check each of these typecodes, and a verification operation on a signature with an unknown type, or a type that does not correspond to the type within the public key MUST return INVALID. The expected length of a variable-length object can be determined from its typecode, and if an object has a different length, then any signature computed from the object is INVALID.

9.1. Hash Formats

The format of the inputs to the hash function $H$ have the property that each invocation of that function has an input that is repeated by a small bounded number of other inputs (due to potential repeats of the I value), and in particular, will vary somewhere in the first 23 bytes of the value being hashed. This property is important for a proof of security in the random oracle model.

The formats used during key generation and signing (including the recommended pseudorandom key generation procedure in Appendix A):

$I \mid u32str(q) \mid u16str(i) \mid u8str(j) \mid tmp$
$I \mid u32str(q) \mid u16str(D_PBL) \mid y[0] \mid \ldots \mid y[p-1]$
$I \mid u32str(q) \mid u16str(D_MES) \mid C \mid message$
$I \mid u32str(r) \mid u16str(D_LEA) \mid OTS_PUB_HASH[r-2^h]$
$I \mid u32str(r) \mid u16str(D_INTR) \mid T[2*r] \mid T[2*r+1]$
$I \mid u32str(q) \mid u16str(i) \mid u8str(0xff) \mid SEED$

Each hash type listed is distinct; at locations 20, 21 of the value being hashed, there exists either a fixed value D_PBL, D_MES, D_LEA, D_INTR, or a 16 bit value i. These fixed values are distinct from each other, and are large (over 32768), while the 16 bit values of i are small (currently no more than 265; possibly being slightly larger if larger hash functions are supported); hence the range of possible values of i will not collide any of the D_PBL, D_MES, D_LEA, D_INTR identifiers. The only other collision possibility is the Winternitz chain hash colliding with the recommended pseudorandom key generation process; here, at location 22 of the value being hashed, the Winternitz chain function has the value $u8str(j)$, where $j$ is a value between 0 and 254, while location 22 of the recommended pseudorandom key generation process has value 255.

For the Winternitz chaining function, D_PBL, and D_MES, the value of $I \mid u32str(q)$ is distinct for each LMS leaf (or equivalently, for
each q value). For the Winternitz chaining function, the value of u16str(i) || u8str(j) is distinct for each invocation of H for a given leaf. For D_PBLC and D_MESG, the input format is used only once for each value of q, and thus distinctness is assured. The formats for D_INTR and D_Leaf are used exactly once for each value of r, which ensures their distinctness. For the recommended pseudorandom key generation process, for a given value of I, q and j are distinct for each invocation of H.

The value of I is chosen uniformly at random from the set of all 128 bit strings. If 2^64 public keys are generated (and hence 2^64 random I values), there is a nontrivial probability of a duplicate (which would imply duplicate prefixes). However, there will be an extremely high probability there will not be a four-way collision (that is, any I value used for four distinct LMS keys; probability < 2^-132), and hence the number of repeats for any specific prefix will be limited to at most 3. This is shown (in [Fluhrer17]) to have only a limited effect on the security of the system.

9.2. Stateful signature algorithm

The LMS signature system, like all N-time signature systems, requires that the signer maintain state across different invocations of the signing algorithm, to ensure that none of the component one-time signature systems are used more than once. This section calls out some important practical considerations around this statefulness. These issues are discussed in greater detail in [STMGMT].

In a typical computing environment, a private key will be stored in non-volatile media such as on a hard drive. Before it is used to sign a message, it will be read into an application’s Random Access Memory (RAM). After a signature is generated, the value of the private key will need to be updated by writing the new value of the private key into non-volatile storage. It is essential for security that the application ensure that this value is actually written into that storage, yet there may be one or more memory caches between it and the application. Memory caching is commonly done in the file system, and in a physical memory unit on the hard disk that is dedicated to that purpose. To ensure that the updated value is written to physical media, the application may need to take several special steps. In a POSIX environment, for instance, the O_SYNC flag (for the open() system call) will cause invocations of the write() system call to block the calling process until the data has been written to the underlying hardware. However, if that hardware has its own memory cache, it must be separately dealt with using an operating system or device specific tool such as hdparm to flush the on-drive cache, or turn off write caching for that drive. Because these details vary across different operating systems and devices,
this note does not attempt to provide complete guidance; instead, we call the implementer’s attention to these issues.

When hierarchical signatures are used, an easy way to minimize the private key synchronization issues is to have the private key for the second level resident in RAM only, and never write that value into non-volatile memory. A new second level public/private key pair will be generated whenever the application (re)starts; thus, failures such as a power outage or application crash are automatically accommodated. Implementations SHOULD use this approach wherever possible.

9.3. Security of LM-OTS Checksum

To show the security of LM-OTS checksum, we consider the signature $y$ of a message with a private key $x$ and let $h = H(message)$ and $c = \text{Cksm}(H(message))$ (see Section 4.5). To attempt a forgery, an attacker may try to change the values of $h$ and $c$. Let $h'$ and $c'$ denote the values used in the forgery attempt. If for some integer $j$ in the range 0 to $u$, where $u = \text{ceil}(8*n/w)$ is the size of the range that the checksum value can cover, inclusive,

\[ a' = \text{coef}(h', j, w), \]
\[ a = \text{coef}(h, j, w), \]
\[ a' > a \]

then the attacker can compute $F^{a'}(x[j])$ from $F^a(x[j]) = y[j]$ by iteratively applying function $F$ to the $j$-th term of the signature an additional $(a' - a)$ times. However, as a result of the increased number of hashing iterations, the checksum value $c'$ will decrease from its original value of $c$. Thus a valid signature’s checksum will have, for some number $k$ in the range $u$ to $(p-1)$, inclusive,

\[ b' = \text{coef}(c', k, w), \]
\[ b = \text{coef}(c, k, w), \]
\[ b' < b \]

Due to the one-way property of $F$, the attacker cannot easily compute $F^{b'}(x[k])$ from $F^b(x[k]) = y[k]$. 
10. Comparison with other work

The eXtended Merkle Signature Scheme (XMSS) [XMSS], [RFC8391] is similar to HSS in several ways. Both are stateful hash based signature schemes, and both use a hierarchical approach, with a Merkle tree at each level of the hierarchy. XMSS signatures are slightly shorter than HSS signatures, for equivalent security and an equal number of signatures.

HSS has several advantages over XMSS. HSS operations are roughly four times faster than the comparable XMSS ones, when SHA256 is used as the underlying hash. This occurs because the hash operation done as a part of the Winternitz iterations dominates performance, and XMSS performs four compression function invocations (two for the PRF, two for the F function) while HSS needs only perform one. Additionally, HSS is somewhat simpler (as each hash invocation is just a prefix followed by the data being hashed).

11. Acknowledgements

Thanks are due to Chirag Shroff, Andreas Huelsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, Russ Housley, Philip Lafrance, Alexander Truskovsky, Mark Peruzel for constructive suggestions and valuable detailed review. We especially acknowledge Jerry Solinas, Laurie Law, and Kevin Igoe, who pointed out the security benefits of the approach of Leighton and Micali [USPTO5432852] and Jonathan Katz, who gave us security guidance, and Bruno Couillard and Jim Goodman for an especially thorough review.

12. References

12.1. Normative References


12.2. Informative References


Appendix A. Pseudorandom Key Generation

An implementation MAY use the following pseudorandom process for generating an LMS private key.

SEED is an m-byte value that is generated uniformly at random at the start of the process,

I is LMS key pair identifier,

q denotes the LMS leaf number of an LM-OTS private key,

x_q denotes the x array of private elements in the LM-OTS private key with leaf number q,

i is the index of the private key element, and

H is the hash function used in LM-OTS.

The elements of the LM-OTS private keys are computed as:

\[ x_q[i] = H(I || u32str(q) || u16str(i) || u8str(0xff) || SEED). \]

This process stretches the m-byte random value SEED into a (much larger) set of pseudorandom values, using a unique counter in each invocation of H. The format of the inputs to H are chosen so that they are distinct from all other uses of H in LMS and LM-OTS. A careful reader will note that this is similar to the hash we perform
when iterating through the Winternitz chain; however in that chain, the iteration index will vary between 0 and 254 maximum (for W=8), while the corresponding value in this formula is 255. This algorithm is included in the proof of security in [Fluhrer17] and hence this method is safe when used within the LMS system; however any other cryptographically secure method of generating private keys would also be safe.

Appendix B. LM-OTS Parameter Options

The LM-OTS one time signature method uses several internal parameters, which are a function of the selected parameter set. These internal parameters set:

p  - This is the number of independent Winternitz chains used in the signature; it will be the number of w-bit digits needed to hold the n-bit hash (u in the below equations), along with the number of digits needed to hold the checksum (v in the below equations)

ls  - This is the size of the shift needed to move the checksum so that it appears in the checksum digits

ls is needed because, while we express the checksum internally as a 16 bit value, we don’t always express all 16 bits in the signature; for example, if w=4, we might use only the top 12 bits. Because we read the checksum in network order, this means that, without the shift, we’ll use the higher order bits (which may be always 0), and omit the lower order bits (where the checksum value actually resides). This shift is here to ensure that the parts of the checksum we need to express (for security) actually contribute to the signature; when multiple such shifts are possible, we take the minimal value.
The parameters $ls$, and $p$ are computed as follows:

\[ u = \text{ceil}(8n/w) \]
\[ v = \text{ceil}\left(\frac{\text{floor}(\text{lg}((2^w - 1) * u)) + 1}{w}\right) \]
\[ ls = 16 - (v * w) \]
\[ p = u + v \]

Here $u$ and $v$ represent the number of $w$-bit fields required to contain the hash of the message and the checksum byte strings, respectively. And as the value of $p$ is the number of $w$-bit elements of\((H(\text{message}) || Cksm(H(\text{message})))\), it is also equivalently the number of byte strings that form the private key and the number of byte strings in the signature. The value 16 in the $ls$ computation of $ls$ corresponds to the 16 bits value used for the sum variable in Algorithm 2 in Section 4.4

A table illustrating various combinations of $n$ and $w$ with the associated values of $u$, $v$, $ls$, and $p$ is provided in Table 6.

<table>
<thead>
<tr>
<th>Hash Length in Bytes ($n$)</th>
<th>Winternitz Parameter ($w$)</th>
<th>$w$-bit Elements in Hash ($u$)</th>
<th>$w$-bit Elements in Checksum ($v$)</th>
<th>Left Shift ($ls$)</th>
<th>Total Number of $w$-bit Elements ($p$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1</td>
<td>256</td>
<td>9</td>
<td>7</td>
<td>265</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>128</td>
<td>5</td>
<td>6</td>
<td>133</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>64</td>
<td>3</td>
<td>4</td>
<td>67</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>32</td>
<td>2</td>
<td>0</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 6

Appendix C. An iterative algorithm for computing an LMS public key

The LMS public key can be computed using the following algorithm or any equivalent method. The algorithm uses a stack of hashes for data. It also makes use of a hash function with the typical init/update/final interface to hash functions; the result of the invocations hash_init(), hash_update(N[1]), hash_update(N[2]), ..., hash_update(N[n]), $v = \text{hash\_final}()$, in that order, is identical to that of the invocation of $H(N[1] || N[2] || \ldots || N[n])$. 

Generating an LMS Public Key from an LMS Private Key

for ( \( i = 0; i < 2^h; i = i + 1 \) ) {
    \( r = i + \text{num}_\text{lmots_keys}; \)
    \( \text{temp} = H(I || \text{u32str}(r) || \text{u16str(D_LEAF)} || \text{OTS_PUB_HASH}[i]) \)
    \( j = i; \)
    while (\( j \% 2 == 1 \)) {
        \( r = (r - 1)/2; \)
        \( j = (j-1) / 2; \)
        \( \text{left_side} = \text{pop(data stack)}; \)
        \( \text{temp} = H(I || \text{u32str}(r) || \text{u16str(D_INTR)} || \text{left_side} || \text{temp}) \)
        \( \text{push temp onto the data stack} \)
    }
    \( \text{public_key} = \text{pop(data stack)} \)
}

Note that this pseudocode expects that all \( 2^h \) leaves of the tree have equal depth; that is, \( \text{num}_\text{lmots_keys} \) to be a power of 2. The maximum depth of the stack will be \( h-1 \) elements, that is, a total of \( (h-1) \times n \) bytes; for the currently defined parameter sets, this will never be more than 768 bytes of data.

Appendix D. Method for deriving authentication path for a signature

The LMS signature consists of \( \text{u32str}(q) || \text{lmots_signature} || \text{u32str(type)} || \text{path}[0] || \text{path}[1] || \ldots || \text{path}[h-1] \). This appendix shows one method of constructing this signature, assuming that the implementation has stored the \( T[] \) array that was used to construct the public key. Note that this is not the only possible method; other methods exist which don’t assume that you have the entire \( T[] \) array in memory. To construct a signature, you perform the following algorithm:
Generating an LMS Signature

1. set type to the typecode of the LMS algorithm

2. extract h from the typecode according to table 2

3. create the LM-OTS signature for the message:
   ots_signature = lmots_sign(message, LMS_PRIV[q])

4. compute the array path as follows:
   i = 0
   r = 2^h + q
   while (i < h) {
     temp = (r / 2^i) xor 1
     path[i] = T[temp]
     i = i + 1
   }

5. S = u32str(q) || ots_signature || u32str(type) ||
     path[0] || path[1] || ... || path[h-1]

6. q = q + 1

7. return S

where ‘xor’ is the bitwise exclusive-or operation, and / is integer division (that is, rounded down to an integer value)

Appendix E. Example Implementation

An example implementation can be found online at https://github.com/cisco/hash-sigs.

Appendix F. Test Cases

This section provides test cases that can be used to verify or debug an implementation. This data is formatted with the name of the elements on the left, and the value of the elements on the right, in hexadecimal. The concatenation of all of the values within a public key or signature produces that public key or signature, and values that do not fit within a single line are listed across successive lines.
### Test Case 1 Public Key

<table>
<thead>
<tr>
<th>HSS public key</th>
<th>levels</th>
<th>00000002</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>LMS type</th>
<th>00000005</th>
<th># LM_SHA256_M32_H5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMOTS type</td>
<td>00000004</td>
<td># LMOTS_SHA256_N32_W8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>61a5d57d37f5e46bfb7520806b07a1b8</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>50650e3b31fe4a773ea29a07f09cf2ea</td>
</tr>
<tr>
<td></td>
<td>30e579f0df58ef8e298da0434cb2b878</td>
</tr>
</tbody>
</table>

### Test Case 1 Message

<table>
<thead>
<tr>
<th>Message</th>
<th>54686520706f77657273206e6f742064</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>656e656761746564207466f207466520</td>
</tr>
<tr>
<td></td>
<td>556e6974656420537461746573206279</td>
</tr>
<tr>
<td></td>
<td>20746865204366e73746974756496f</td>
</tr>
<tr>
<td></td>
<td>6e2c206e6f722070726f686962697465</td>
</tr>
<tr>
<td></td>
<td>64206279206974207466f207468652053</td>
</tr>
<tr>
<td></td>
<td>74617465732c20617265207265736572</td>
</tr>
<tr>
<td></td>
<td>766554207466f20746865205374617465</td>
</tr>
<tr>
<td></td>
<td>7320726573706563746976656c792c20</td>
</tr>
<tr>
<td></td>
<td>6f72207466f207468652070656f706c65</td>
</tr>
<tr>
<td></td>
<td>2e0a</td>
</tr>
</tbody>
</table>

### Test Case 1 Signature

#### HSS signature

<table>
<thead>
<tr>
<th>Nspk</th>
<th>00000001</th>
</tr>
</thead>
</table>

#### LMS signature

<table>
<thead>
<tr>
<th>q</th>
<th>00000005</th>
</tr>
</thead>
</table>

#### LMOTS signature

<table>
<thead>
<tr>
<th>LMOTS type</th>
<th>00000004</th>
<th># LMOTS_SHA256_N32_W8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>d32b56671d7eb98833c49b433c272586</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc4a1c8a8970528ffa04b966f9426eb9</td>
<td></td>
</tr>
<tr>
<td>y[0]</td>
<td>965a25b7d37f196b9073f3d4a232f6b6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9128ec45146f86292f9dff9610a7bf95</td>
<td></td>
</tr>
<tr>
<td>y[1]</td>
<td>a64c7f60f626162043f86c70324b770</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7f5b4a8a6e191147be866d48878a0</td>
<td></td>
</tr>
</tbody>
</table>
\[\begin{align*}
y[26] & = \text{ab8f5c612ad0b729a1d059d02bf}e18e \\
y[27] & = \text{ec0f3f3f1039a1788b0cf80f4f88}4 \\
y[28] & = \text{4f1f4ab49b99feafdc}b71a50e2f7d6 \\
y[29] & = \text{9b066609c37282d2f06737c7d}7 \\
y[30] & = \text{b7c878c9411cafc507a34a00f4cf}077 \\
y[31] & = \text{d76f7ac99367095a7e936af97f}4 \\
y[32] & = \text{41b64457c54d65ef6500c59df}b69a \\
y[33] & = \text{b0f3f79cd893d314168499889}fbc0 \\
\end{align*}\]

\begin{align*}
\text{LMS type} & = 00000005 \quad \# \text{LM_SHA256_M32_H5} \\
\text{path[0]} & = \text{d8b88112f9200a5e50c42d61b5342c} \\
\text{path[1]} & = \text{129ac6eda8396f357b5a0387c5ce97} \\
\text{path[2]} & = \text{12f5d5be400b49e4501e859f885bf}073 \\
\text{path[3]} & = \text{b5971115aa39ef8d8564b90282c316} \\
\text{path[4]} & = \text{4cca1848cf7da59cc2b3d0692d2}a2 \\
\end{align*}

\begin{align*}
\text{LMS public key} \\
\text{LMS type} & = 00000005 \quad \# \text{LM_SHA256_M32_H5} \\
\text{LMOTS type} & = 00000004 \quad \# \text{LMOTS_SHA256_N32_W8} \\
I & = \text{d2f14ff6346af964569f7d6cb8}80a1b6 \\
K & = \text{6c5004917da6eafe4d9ef6c6407b3db0} \\
\end{align*}

\begin{align*}
\text{final_signature:} \\
\text{LMS signature} \\
q & = 0000000a \\
\text{LMOTS signature} \\
\text{LMOTS type} & = 00000004 \quad \# \text{LMOTS_SHA256_N32_W8} \\
C & = \text{0703c491e7558b35011ece3592eaa5}da \\
y[0] & = \text{95cae05b99e35dfff7d7b057e020998} \\
y[1] & = \text{9bc042da4b4526506485c66d0c19b31} \\
\end{align*}
y[2] 6a120c5612344258b85efdb7db1db9e1
y[3] 865a73caf96557eb39ed3e3f426933ac
y[4] 37f9de2d60113c23f846df26fa942008
y[5] a698994c0827d90e86d43ed0f7bf4bfc
db09b86a373b98288b7094ad81a0185ac
y[6] 100e4f2c5fc38c003c1ab6fe4a79eb2f
y[7] 5ebe48f5847159b8ada03586e65ad9c
y[8] 969f6aecbfe44cf356888a7b15a3ff07
y[9] 4f771760269c04884e1faa329fbf4
ey[10] e61af23ae7fa5d49a5dfcf43c4c26c
ey[11] e8aea2ce8a2990d7a7b75710847dabf
y[12] bead2b253acccc1ac0cecf346cb90fb
y[13] 044beee4fac2603a442bdf7e507243b7
y[14] 319c9944b1586e899d43c17f91bccc8
y[15] 690dbf5b28386b2315f3d36ef2ea3ac
y[16] f30b2b51f48b71b003dfb08249484201
y[17] 0081262a00000480dcbbca93d6f6b6f5c
y[18] 1c0a55e48a0e729f9184fcb1407c3152
y[19] 9db268f6fe5032a363c9801306837fa
y[20] fabdf957fd97eafc80db1d65e435d0e2
y[21] dfd836a2b354023924b6fb7e48bc0b3
ey[22] ed95ee6a4c2d402f4d734c8dc26f3a5c
y[23] 91825daef01ea3ce3c8e3328d0a77d6c
y[24] 57034f2877cbb0f0e1c9a7cbdc828627
y[25] 205e4737b84b58376551d44cl23c3215
cy[26] c812a0970789c83de516d6ad787271963
y[27] 327f0a5fbb6b5907dec02c9a90934af5
y[28] a1c63b72c82653605d1dce51596b3c2
y[29] b4569668f2e3b382007497557692caac
y[30] 4d57b5de9f5569b2aad0137fd47f47e
y[31] 664fcb6d4b971f5b3e07acaceda9ac130e
y[32] 9f38182de994cfff192ec0e82f6d4cb7
y[33] f3fe00812589b7a7ce5154405463301
y[34] 6b84a59bec66191c6c0b37dd1450ed4
y[35] f2d8b584410ceda8025f5d2dd2d12d017
y[36] 6fc1cf2cc06fa8c8bbed4d944e71339e
y[37] ce780fd025b4d1ec34ebbf9fd4270a322
y[38] 4e019fcb444474d482fd2d6e75efb203
y[39] 89cc10cd60abb54c47ede93e08c114e
db04117d714dc1d525e11bed8756192f
y[40] 929d15462b939ff3f52f2252da2ed64d
y[41] 8f0e88818b1efa2c7b08c8794fb1b214
y[42] aa233db3162833141ea4383f1a6f120b
e1db82ce3630b3429114463157a64e91
y[43] 234d475e2f79cbf05e4db6a9407d7c6

[Page 52]
bff7d1190b5c4d6aad2831db61274993
y[26] 715a0182c7dc8089e32c8531deed4f74 31c07c02195eab2ef91efb5613c37af7
y[27] ae0c066babc6936900e1dd26eddc0d2 16c781d56e4ce4e73303fa73007ff7b9
y[28] 49ef23be2aa4dbf25206fe45c20dd888 395b2526391a7249964a4156beac8082
y[29] 12858792bf8e74cba49dee5e8812e019 da87454bff9e847ed83db07af3137430
y[30] 82f880a278f682c2bd0ad6887cb59f65 2e155987d61bfbf6a88d36ee93b6072e6
y[31] 656d9ccbaae3d655852e38de63a2dcf8 058dc9fbf2ab3d3b3539eb77b248a66
y[32] 1091d05eb62f297774fe6053598457c c61908318de4b826f0fc86d4bb117d33
y[33] e865a805009cc2918d9c2f840c4da43 a703ad9f5b806163d7161696b50adc
--------------------------------------------
LMS type 00000005                         # LM_SHA256_M32_H5
path[0] d5c0d1beb06048ed6fe2ef26cef305 b3ed633941ebc8b3bec978754cdd60
path[1] e1920ada52f3d055b5031cee6192520 d6a51154851ce7fd4484a39fae2ab
path[2] 2335525f484e9b40d4a4969394843b dcf6d14c48e8015e0ab92662c05c6e9
path[3] f90b65a7a620168999f32bdf368e5e3 ec9cb70ac7b8399003f175c40885081a
path[4] 9ab3034911fe125631051df0408b39e 6b0bde790911e8978ba07dd56c73e7ee
--------------------------------------------
Test Case 2 Private Key
--------------------------------------------
(note: procedure in Appendix A is used)
SEED 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
I 08fabd4a2091ff0a8cbe4834e74534
--------------------------------------------
Test Case 2 Public Key

HSS public key
levels 00000002
--------------------------------------------
LMS type 00000006 # LM_SHA256_M32_H10
LMOTS type 00000003 # LMOTS_SHA256_N32_W4
I d08fabd4a2091ff0a8cb4ed834e74534
K 32a58885cd9ba0431235466bff9651c6
c92124404d45fa53cf161c28f1ad5a8e
--------------------------------------------
Test Case 2 Message

Message 54686520656e756d65726174696620 The enumeration
696e2074686520646573746974657320 The enumeration
in the Constitution, of certain
7269676874732c207368617274657320 age others retail
6e6564206279207468652070656669 the people
65206252697467687465732072657461696e6c3e...
--------------------------------------------
Test Case 2 Signature

HSS signature
Nspk 00000001
sig[0]:
--------------------------------------------
LMS signature
q 00000003
--------------------------------------------
LMOTS signature
LMOTS type 00000003 # LMOTS_SHA256_N32_W4
C 3d46be88660f8f215d3f96408a7a64cf1c4da02b63a55f62c666ef5707a914ce
y[0] 0674e8cb7a55f0c48d484f31f3aa4af9719a74f22cf823b94431d01c9262ea76
y[1] bb71226d279700ec81c9e95fb11a0d10d0652f9a5796e265ae17737c44eb8c59
y[2] 4508e126a9a7870bf4360820bde9a01d9693779e416828e75bddd7d8c70d50a
y[51] 8240027afd9d52a79b647c90c2709e06
0ed70f87299dd798d68f4fadd3da6c51
y[52] d839f851f986f7840b964ebe73f8cc4c
1572538ec6bc131034ca2894eb736b3b
y[53] da93df5f6fa6f6c0f3ee34336288414
940355fb54d3dfdd03633ae108f3de3e
y[54] bc85a3ff51eefea3bc2cf27e1658f178
9e612c83d0f5df6f7cd071930e2946
y[55] beeeecaa04dccea9f97786001475e0294
bc2852f62eb5d39bb9fbee75916e0e4
y[56] 4a662ecae37ede27e9d6eadfede8f8b2
b2d4bcbf96fa6d6f7321fb0e701f4d4
y[57] 29c2f4dcd153a2742574126e5eacc77
686acfe63ee48f423766e0f4c466810a9
y[58] 05ff5453cc99897b56bc55d4b991114
2f65043f2d744eeb935ba7f4ef23cf80
y[59] cc5a8a3353d619d781e7454826df720e
ec82e06034c4469b5f0c4aa8787752e
y[60] 057fa3419b5bb0e25d309814e1cbeb316
322da8f69931cf42fad3f3bce6ded5b
y[61] 8bfc3d20a2148861b2afc14562dd27f
12897ab0f0685288dccc5c4982f8260268
y[62] 46a24bf77e383c7aaccab1ab692b29ed8
c018a65f3dc2b87ff619a633c41bf4ad
y[63] b1c78725cf1f8f922f6009787b1964247
d0f136b1bc61a45b573c59a16d089917b
y[64] d4a8b66f0d95c581279a139be9dfc6e
98a470a0bceca191f4e476f9370021cb
y[65] c05518a7efd35d89d8577c990a5e1996
1ba6203c959c19829ba7a797c1cccb4b
y[66] 294546454fa5388a23a22e805a5ca35f
956598884bda678615f8c28af5d6a61a

LMS type    00000006                         # LM_SHA256_M32_H10
path[0]     b326493310353ced3876db9d23714818
1b7173bc7cd042cef4db4be94d2e58cd21
path[1]     a769db4657a103279b8ef3a629ca84e
e8361729a9c50e51f45581741cf808315
path[2]     7028a48538ecdd3b38d3d5d26226465
95c4fb73a525a5ed2c30524ebb1d8cc8
path[3]     2e0c19bc4977c6898ff95fd301b00ba
e7169c5e93c6a552456bf96e9d075e3
path[4]     83bb7543c675842baefc7c7db88483b3
276c29d4f0a341c2d406e40d4653b7e4
path[7]     d045851acf6a0a0ea9c710b805cced46
           35ee8c107362f0f8c8d80c14d0ac49c51
path[8]     6703d26d14752fe34c0d2c4247581c1
           8c2cf4de48e9ce94987c88869caebe4
path[9]     a415e291fd107d21dcf084b1582082
           49f28f4f7c7e931ba7b3bd0d824a4570
--------------------------------------------

LMS public key
LMS type    00000005                         # LM_SHA256_M32_H5
LMOTS type  00000004                         # LMOTS_SHA256_N32_W8
I           215f83b7ccb9acbcd88db97d0d4dc2b
K           a1cd035833e0e90059603f26e07ad2aa
d152338e7a5e5984bced5f7bb4eba40b7
--------------------------------------------

final_signature:
--------------------------------------------

LMS signature
q           00000004
--------------------------------------------

LMOTS signature
LMOTS type  00000004                         # LMOTS_SHA256_N32_W8
C           0eb1ed54a2460d512388cad533138d24
           0534e97b1e82d33bd927d021dfe24eb
y[0]        11b364902369f8515b0189e50c0098
           850ac343a77b3638319c347d7310269d
y[1]        3b7714fa406b8c35b021d54d4fdada7b
           9c954db506799e72aaf58c5aae7aca
y[2]        057aa0e2e74e7d5cfd17a0823429db629
           65b7d563c57b4cecc942cc85e29c1dad
y[3]        83cabc8b4d61aacc457f336e6a10b6632
           3f5887bf3523dfcaddee1538053bfa98d
y[4]        c6bf59d6a4e36b5eb2a9c6572a60
           67ce7c3279e9039b3bea6a1edc7fdd9c3
y[5]        df927aade10c1c9f2d5ff446450d2a39
           98d09f6202b5e07c3f97d2458c693d3c
y[6]        81906439787a7f4d64e97e3f1c4a08a
           7c5bc03f5d568c2c172907e0b7e5b
y[7]        2f19014375a6043d56e5d253471f4ee
           cfc62575fb66ff37edfa249d6edca1a9
y[8]        f7979f5a3c55a066700f45863f04b6c
           8a58cfd431241e002d02c021747cbf1
y[9]        8b6363ae547c17713689f317835cb0e
f4303df4304d6af0da44f4af7800
y[10]       bc7a5c8a5abdb12dc718b559b74cab9
           090e33c58a955300981c420c4da8fffd
y[11]       67df540890a062fe40da8b2c1c548ce
           d22473219c534911d48ccaabfb71bc71
y[12]       862f4a24ebd376d288fd4e6fb06ed870
y[13]       5787c5fedc813cd2697e5b1aac1ced45
y[14]       767b14ce884099aeebb601a93559a8e89
y[15]       3e143d1c395bc326da821d79a9ed41dc
y[16]       fbe549147f71c092f4f3ac522b5cc572
y[17]       9070650487bae9b5b5671ecc9ccc2ce5
y[18]       1ead87acc01985268521222fb9057df7e
y[19]       d41810b5ef04df7cc67368c90f573b1a
y[20]       c2ce956c365ed38e893ce7b2fabe15d36
y[21]       85a3df2fa3d4cc098fa57dd60d29754
y[22]       a8ade980ad0f93f6787075c3f680a2ba
y[23]       19368c61d1af52ab7e21f416be09d2a
y[24]       8d64c3d3d8582968c283902229f85ae
e297e717c0948d4a233b5db658d3d7
y[25]       7bf04ff3ff8fa5e383a48574802ed
y[26]       545bbe7a6b47535333537d3706067640
y[27]       135a7ce517279cd683039747d218647c
y[28]       86e097b0daa2872d54b8f3e508598762
y[29]       9547b830d8118161b65079fe7bc59a99
e93c7380e3e70b7138fe5d9be255150
y[30]       2b698e09ae193972f27d40f38dea264a
y[31]       0126637d74ae492a62499fa103436d3
eb0d4029ac712bfc7a5eacbd75186d6
y[32]       4fe903a5ae65527cd65bbd4e9925ca2
y[33]       4fd7214d617c150544e423f450c99ce
y[34]       51ac8005d33acad741fbed3b17b7266a4
y[35]       a3bb86da7ebad8b101e5c79e9a207
y[36]       852cf91249ef480619ff2af8cabca831
y[37]       25d1faa94ccb0a039a906f683bf3f47a97
c871df513e510a7a25f283b196075778
y[38]       496152a91c2bf9da76ebe089f4654877
y[39]       f2d586ae7149c40e6b3edeb2b5c7e8
y[40]       2429b9e8cb4834c83464f079995332e4
y[41]       b3c8f5a12bb4b86c1f740d45dc6c1f79
y[42]       952c0b7420df525e37c15377b5f09843
y[43]       19c3993921e5ccd97e097592064530d3
y[44]       3de3afad5733cbe7703c52962b62f7734
y[45]       2efbf5a04755b0b3c9c7b3c28463e84c
y[46]       aa2de3f1dc2b9baaacc7aa646e44b5
c0f16044df38fbad2f9647b3a8389a13
y[47]       982f2e370c0f78ed7024c84db34e36b
y[48]       46cc76460a690cc86c302457dd1cde1
y[49]       97ec8075e8b393d542075134e2a17ee
y[50]       70a7a187075d03ae3c853c6f60729ba4

-------------

LMS type                  00000005 # LM_SHA256_M32_H5
path[0]       4def1f6965bda6c676c5a4dc7c35f97f8
                2cb0e31c68d04f1dad96314ff09e6b3d
path[1]       e96aeeee300df68bf1bca9f6c58e0323

path[2] 36cd819aaf578744e50d1357a0e42867
        04d341aa0a337b19fe4bc43c2e79964d
    4f351089f2e0e41c7c43ae0d49e7f404
path[3] b0f75be80ea3af098c9752420a8ac0ea
    2bbbf4ebea05238ae0d8ce63f0c6e5
path[4] e4041d95398a6f7f3e0ee97cc1591849
    d4ed236338b147abde9f51ef9fd4e1c1

Authors’ Addresses

David McGrew
Cisco Systems
13600 Dulles Technology Drive
Herndon, VA  20171
USA

Email: mcgrew@cisco.com

Michael Curcio
Cisco Systems
7025-2 Kit Creek Road
Research Triangle Park, NC  27709-4987
USA

Email: micurcio@cisco.com

Scott Fluhrer
Cisco Systems
170 West Tasman Drive
San Jose, CA
USA

Email: sfluhrer@cisco.com