Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: December 22, 2017

D. McGrew
M. Curcio
S. Fluhrer
Cisco Systems

June 20, 2017

Hash-Based Signatures
draft-mcgrew-hash-sigs-07

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.

Status of This Memo

Copyright Notice

Table of Contents

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 relatively slow key generation. 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 at most one message securely, but cannot securely sign more than one. 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 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. We denote the one-time signature scheme incorporate in LMS as LM-OTS. This note is structured as follows. Notation is introduced in Section 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 public formats are described in Section 7. The rationale for design decisions are given in Section 8. The IANA registry for these signature systems is described in Section 10. Security considerations are presented in Section 12.

1.1. 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 are valid.

A message/signature pair are 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 are 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

+ : a + b denotes the sum of a and b

- : a - b denotes the difference of a and b

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. 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. Strings of w-bit elements

If S is a byte string, then byte(S, i) denotes its i^th byte, where byte(S, 0) is the leftmost byte. 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,

```
    coef(S, i, w) = (2^w - 1) AND
                    ( byte(S, floor(i * w / 8)) >>
                      (8 - (w * (i % (8 / w)) + w)) )
```

For example, if S is the string 0x1234, then coef(S, 7, 1) is 0 and coef(S, 0, 4) is 1.

```
                      S (represented as bits)
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                 ^
                                 |
                           coef(S, 7, 1)


                 S (represented as four-bit values)
          +-----------+-----------+-----------+-----------+
          |     1     |     2     |     3     |     4     |
          +-----------+-----------+-----------+-----------+
                ^
                |
          coef(S, 0, 4)
```

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

3.2. 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 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 hash.

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 separate parameter) will be bytes 16-19 of the hash.

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 separate parameter) will be bytes 16-19 of the hash.

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 hash, 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. The value is also the index of the LM-OTS private key element upon which H is being applied. It is represented as a 16-bit (two byte) unsigned integer in network byte order.

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 D is a value between 0 and 264. If present, it occurs at byte 22 of the hash.

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 22-53 of the hash.

3.3. Functions

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

ceil(r) : returns the smallest integer larger than r

floor(r) : returns the largest integer smaller than r

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

3.4. 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 7).

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 a diversification parameter q. When the LM-OTS signature system is used outside of an N-time signature system, this value SHOULD 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 (number of bits) of the Winternitz coefficients; 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

ls : the number of left-shift bits used in the checksum function Cksm (defined in Section 4.5).

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 12.

The value of n is determined by the functions 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 is provided in Table 1.

4.2. Parameter Sets

To fully describe a LM-OTS signature method, the parameters n and w, as well as the function H, MUST be specified. This section defines several LM-OTS methods, each of which is identified by a name. The values for p and ls are provided as a convenience.

Name	H	n	w	p	ls
LMOTS_SHA256_N32_W1	SHA256	32	1	265	7
LMOTS_SHA256_N32_W2	SHA256	32	2	133	6
LMOTS_SHA256_N32_W4	SHA256	32	4	67	4
LMOTS_SHA256_N32_W8	SHA256	32	8	34	0

Table 1

Here SHA256 denotes the NIST standard hash function [FIPS180].

4.3. Private Key

The LM-OTS private key consists 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. set type to the typecode of the algorithm

2. set n and p according to the typecode and Table 1

3. compute the array x as follows:
```
for ( i = 0; i < p; i = i + 1 ) {
  set x[i] to a uniformly random n-byte string
}
```

4. 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 LM-OTS private key.

4.4. 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 ) {
     tmp = x[i]
     for ( j = 0; j < 2^w - 1; j = j + 1 ) {
        tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
     }
     y[i] = tmp
   }
   K = H(I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1])

5. return u32str(type) || I || u32str(q) || K
```

The public key is the value returned by Algorithm 1.

4.5. Checksum

A checksum is used to ensure that any forgery attempt that manipulates the elements of an existing signature will be detected. The security property that it provides is detailed in Section 12. 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

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

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.

4.6. Signature Generation

The LM-OTS signature of a message is generated by first prepending the LM key identifier I, the LM leaf identifier q, the value D_MESG 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 4.4.

The signature is the string returned by Algorithm 3. Section 7 specifies the typecode and more formally defines the encoding and decoding of the string.

4.7. 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. if pubtype is not equal to sigtype, return INVALID

   c. 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, 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 Kc from a Signature, Message, Signature Typecode Type , 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 Type, return INVALID

   c. set n and p according to the sigtype and Table 1;  if the
   signature is not exactly 4 + n * (p+1) bytes long, return INVALID

   d. C = next n bytes of signature

   e.  y[0] = next n bytes of signature
       y[1] = next n bytes of signature
       ...
     y[p-1] = next n bytes of signature

3. compute the string Kc 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 = y[i]
     for ( j = a; j < 2^w - 1; j = j + 1 ) {
        tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
     }
     z[i] = tmp
   }
   Kc = H(I || u32str(q) || u16str(D_PBLC) || z[0] || z[1] || ... || z[p-1])

4. return Kc
```

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. 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, its left child has node number 2*N, while its right child has node number 2*N+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, ..., (2^h)+(2^h)-1. In general, the j^th node at level L has node number 2^L + 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:

h : the height (number of levels - 1) in the tree, and

m : the number of bytes associated with each node.

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an m-byte string. H SHOULD be the same as in Section 4.1, but MAY be different.

There are 2^h leaves in the tree. The hash function used within the LMS system SHOULD be the same as the hash function used within the LM-OTS system used to generate the leaves.

Name	H	m	h
LMS_SHA256_M32_H5	SHA256	32	5
LMS_SHA256_M32_H10	SHA256	32	10
LMS_SHA256_M32_H15	SHA256	32	15
LMS_SHA256_M32_H20	SHA256	32	20
LMS_SHA256_M32_H25	SHA256	32	25

Table 2

5.2. LMS Private Key

An LMS private key consists 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. 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
       }

   3. 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.

5.3. LMS Public Key

An LMS public key is defined as follows, where we denote the public key associated with the i^th LM-OTS private key as OTS_PUB[i], with i ranging from 0 to (2^h)-1. Each instance of an LMS public/private key pair is associated with a perfect 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

```
T[r]=/ H(I||u32str(r)||u16str(D_LEAF)||OTS_PUB[r-2^h])   if r >= 2^h,
     \ H(I||u32str(r)||u16str(D_INTR)||T[2*r]||T[2*r+1]) otherwise.
```

The LMS public key is the string u32str(type) || I || T[1]. Section 7 specifies the format of the type variable. The value I is the private key identifier (whose length is denoted by the parameter set), 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, and

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) || ots_signature || u32str(type) || path[0] || path[1] || ... || path[h-1]. Section 7 specifies the typecode and more formally defines the format. The array of values contains the siblings of the nodes on the path from the leaf to the root but does not contain the nodes on the path themselves. The array for a tree with height h will have h values. The first value is the sibling of the leaf, the next value is the sibling of the parent of the leaf, and so on up the path to the root.

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 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. 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].

5.5. 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 four 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. if the public key is not exactly 4 + LenI + m bytes long, return INVALID

   c. I = next LenI bytes of the public key

   d. T[1] = next m bytes of the public key

6. compute the candidate LMS root value Tc from the signature, message, identifier and pubtype using Algorithm 6b.

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

Algorithm 6b: 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, ots_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. ots_signature = bytes 8 through 8 + n * (p + 1) - 1 of signature

f. sigtype = strTou32(4 bytes of signature at location 8 + n * (p + 1))

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
```

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

6. 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
```

7. 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. Leighton and Micali describe a scheme in which an

LMS public key is used to sign a second LMS public key, which is then distributed along with the signatures generated with the second public key [USPTO5432852]. This hierarchical scheme, which we describe in this section, uses an LMS scheme as a component. HSS, in essence, utilizes a tree of LMS trees, in which 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 one of its leaves. 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.

Each level of 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 LMS private key of the i^th level,

pub[i] is the LMS public key of the i^th level (which includes the identifier I as well as the key value K),

sig[i] is the LMS signature of the i^th level,

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.

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.

6.1. Key Generation

When an HSS key pair is generated, the key pair for each level MUST have its own identifier I.

To generate an HSS private and public key pair, new LMS private and public keys are generated for prv[i] and pub[i] for i=0, ... , L-1. These key pairs, and their identifiers, MUST be generated independently. All of the information of the leaf level L-1, including the private key, MUST NOT be stored in nonvolatile memory. Letting Nnv denote the lowest level for which prv[Nnv] is stored in nonvolatile memory, there are Nnv nonvolatile levels, and L-Nnv volatile levels. For security, Nnv should be as close to one as possible (see Section 12.1).

The public key of the HSS scheme is consists of the number of levels L, followed by pub[0], the public key of the top level.

The HSS private key consists of prv[0], ... , prv[L-1]. The values pub[0] and prv[0] do not change, though the values of pub[i] and prv[i] are dynamic for i > 0, and are changed by the signature generation algorithm.

6.2. Signature Generation

To sign a message using the private key prv, 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.

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

```
u32str(Nspk) || signed_pub_key[0] || ... || signed_pub_key[Nspk-1] ||
sig[Nspk]
```

which is equivalent to

u32str(Nspk) || sig[0] || pub[1] || ... || sig[Nspk-1] || pub[Nspk] || sig[Nspk]

6.3. Signature Verification

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

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

L' = strTou32(first four bytes of S)
if L' is not equal to the number of levels L in pub:
   return INVALID
for (i = 0; i < L; i = i + 1) {
   siglist[i] = next LMS signature parsed from S
   publist[i] = next LMS public key parsed from S
}
siglist[L-1] = next LMS signature parsed from S

key = pub
for (i = 0; i < L; i = i + 1) {
   sig = siglist[i]
   msg = publist[i]
   if (lms_verify(msg, key, sig) != VALID):
       return INVALID
   key = msg
return lms_verify(message, key, siglist[L-1])
```

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.

7. 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:

```
/* one-time signatures */

enum ots_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 ots_signature switch (ots_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 hbs_algorithm_type {
  hbs_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 (hbs_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;
  ots_signature lmots_sig;
  lms_path nodes;
};

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

union hbs_public_key switch (hbs_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;
  hbs_public_key pub;
};

struct signed_public_key {
  hbs_signature sig;
  hbs_public_key pub;
}

struct hss_signature {
  signed_public_key signed_keys<7>;
  hbs_signature sig_of_message;
};
```

Many of the objects 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.

8. 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 this size was 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 4.5 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 4.6 and the total actual number. In the case that the number of times H is applied is 0, the sum is (2^w - 1) * (8*n/w). Thus for the purposes of this document, which describes signature methods based on H = SHA256 (n = 32 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.

9. History

This is the seventh version of this draft. It has the following changes from previous versions:

Version 06

Modified the order of the values that were hashed to make it easier to prove security.

Decreased the size of the I LMS public key identifier to 16 bytes.

Version 05

Clarified the L=1 specific case.

Extended the parameter sets to include an H=25 option

A large number of corrections and clarifications

Added a comparison to XMSS and SPHINCS, and citations to those algorithms and to the recent Security Standardization Research 2016 publications on the security of LMS and on the state management in hash-based signatures.

Version 04

Specified that, in the HSS method, the I value was computed from the I value of the parent LM tree. Previous versions had the I value extracted from the public key (which meant that all LM trees of a particular level and public key used the same I value)

Changed the length of the I field based on the parameter set. As noted in the Rationale section, this allows an implementation to compute SHA256 n=32 based parameter sets significantly faster.

Modified the XDR of an HSS signature not to use an array of LM signatures; LM signatures are variable length, and XDR doesn't support arrays of variable length structures.

Changed the LMS registry to be in a consistent order with the LM-OTS parameter sets. Also, added LMS parameter sets with height 15 trees

Previous versions

In Algorithms 3 and 4, the message was moved from the initial position of the input to the function H to the final position, in the computation of the intermediate variable Q. This was done to improve security by preventing an attacker that can find a collision in H from taking advantage of that fact via the forward chaining property of Merkle-Damgard.

The Hierarchical Signature Scheme was generalized slightly so that it can use more than two levels.

Several points of confusion were corrected; these had resulted from incomplete or inconsistent changes from the Merkle approach of the earlier draft to the Leighton-Micali approach.

This section is to be removed by the RFC editor upon publication.

10. 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 3, and one for Leighton-Micali Signatures, as defined in Section 4. 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. 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.

Requests to add an entry to the registry MUST include the name and the reference. The number is assigned by IANA. Submitters SHOULD have their requests reviewed by the IRTF Crypto Forum Research Group (CFRG) at cfrg@ietf.org. Interested applicants that are unfamiliar with IANA processes should visit http://www.iana.org.

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.

Name	Reference	Numeric Identifier
LMOTS_SHA256_N32_W1	Section 4	0x00000001
LMOTS_SHA256_N32_W2	Section 4	0x00000002
LMOTS_SHA256_N32_W4	Section 4	0x00000003
LMOTS_SHA256_N32_W8	Section 4	0x00000004

Table 3

The LMS registry is as follows.

Name	Reference	Numeric Identifier
LMS_SHA256_M32_H5	Section 5	0x00000005
LMS_SHA256_M32_H10	Section 5	0x00000006
LMS_SHA256_M32_H15	Section 5	0x00000007
LMS_SHA256_M32_H20	Section 5	0x00000008
LMS_SHA256_M32_H25	Section 5	0x00000009

Table 4

An IANA registration of a signature system does not constitute an endorsement of that system or its security.

11. Intellectual Property

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

11.1. 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.

12. 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 and signature that are 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.

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 are

```
I || u32str(q) || u16str(i) || u8str(j) || tmp
I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1]
I || u32str(q) || u16str(D_MESG) || C || message
I || u32str(r) || u16str(D_LEAF) || OTS_PUB[r-2^h]
I || u32str(r) || u16str(D_INTR) || T[2*r] || T[2*r+1]
I || u32str(q) || u16str(j) || u8str(0xff) || SEED
```

Each hash type listed is distinct; at locations 20, 21 of each hash, there exists either a fixed value D_PBLC, D_MESG, D_LEAF, D_INTR, or a 16 bit value (i or j). These fixed values are distinct from each other, and large (over 32768), while the 16 bit values are small (currently no more than 265; possibly being slightly larger if larger hash functions are supported; hence the hash invocations with i/j will not collide any of the D_PBLC, D_MESG, D_LEAF, D_INTR hashes. The only other collision possibility is the Winternitz chain hash colliding with the recommended pseudorandom key generation process; here, at location 22, 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_PBLC, and D_MESG, the value of I || 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

pseuddorandom 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 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 3. This can be shown (in [Fluhrer17]) to have only a limited ebffect on the security of the system.

12.1. 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.

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 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.

12.2. 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 = Cksm(H(message)) (see Section 4.6). 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 = ceil(8*n/w) is the size of the range that the checksum value can over), inclusive,

a' = coef(h', j, w),

a = coef(h, j, w), and

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' = coef(c', k, w),

b = coef(c, k, w), and

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].

13. Comparison with other work

The eXtended Merkle Signature Scheme (XMSS) [XMSS] 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) where HSS need only perform one. Additionally, HSS is somewhat simpler, and it admits a single-level tree in a simple way (as described in Section 6.2).

Another advantage of HSS is the fact that it can use a stateless hash-based signature scheme in its non-volatile levels, while continuing to use LMS in its volatile levels, and thus realize a hybrid stateless/stateful scheme as described in [STMGMT]. While we conjecture that hybrid schemes will offer lower computation times and signature sizes than purely stateless schemes, the details are outside the scope of this note. HSS is therefore amenable to future extensions that will enable it to be used in environments in which a purely stateful scheme would be too brittle.

SPHINCS [SPHINCS] is a purely stateless hash based signature scheme. While that property benefits security, its signature sizes and generation times are an order of magnitude (or more) larger than those of HSS, making it more difficult to adopt in some practical scenarios.

14. Acknowledgements

Thanks are due to Chirag Shroff, Andreas Huelsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, Russ Housley and Philip Lafrance 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.

15. References

15.1. Normative References

[FIPS180] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS 180-4, March 2012.

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

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <http://www.rfc-editor.org/info/rfc2434>.

[RFC3979] Bradner, S., Ed., "Intellectual Property Rights in IETF Technology", RFC 3979, DOI 10.17487/RFC3979, March 2005, <http://www.rfc-editor.org/info/rfc3979>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <http://www.rfc-editor.org/info/rfc4506>.

[RFC4879] Narten, T., "Clarification of the Third Party Disclosure Procedure in RFC 3979", RFC 4879, DOI 10.17487/RFC4879, April 2007, <http://www.rfc-editor.org/info/rfc4879>.

[USPTO5432852] Leighton, T. and S. Micali, "Large provably fast and secure digital signature schemes from secure hash functions", U.S. Patent 5,432,852, July 1995.

15.2. Informative References

[C:Merkle87] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Lecture Notes in Computer Science crypto87vol, 1988.

[C:Merkle89a] Merkle, R., "A Certified Digital Signature", Lecture Notes in Computer Science crypto89vol, 1990.

[C:Merkle89b] Merkle, R., "One Way Hash Functions and DES", Lecture Notes in Computer Science crypto89vol, 1990.

[Fluhrer17] Fluhrer, S., "Further analysis of a proposed hash-based signature standard", EPrint http://eprint.iacr.org/2017/553.pdf, 2017.

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 28th ACM Symposium on the Theory of Computing p. 212, 1996.

[Katz16] Katz, J., "Analysis of a proposed hash-based signature standard", Security Standardization Research (SSR) Conference http://www.cs.umd.edu/~jkatz/papers/HashBasedSigs-SSR16.pdf, 2016.

[Merkle79] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

[SPHINCS] Bernstein, D., Hopwood, D., Hulsing, A., Lange, T., Niederhagen, R., Papachristadoulou, L., Schneider, M., Schwabe, P., and Z. Wilcox-O'Hearn, "SPHINCS: Practical Stateless Hash-Based Signatures.", Annual International Conference on the Theory and Applications of Cryptographic Techniques Springer., 2015.

[STMGMT] McGrew, D., Fluhrer, S., Kampanakis, P., Gazdag, S., Butin, D., and J. Buchmann, "State Management for Hash-based Signatures.", Security Standardization Resarch (SSR) Conference 224., 2016.

[XMSS] Buchmann, J., Dahmen, E., and . Andreas Hulsing, "XMSS-a practical forward secure signature scheme based on minimal security assumptions.", International Workshop on Post-Quantum Cryptography Springer Berlin., 2011.

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,

j is an 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[j] = H(I || u32str(q) || u16str(j) || 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 cryptographical secure method of generating private keys would also be safe.

Appendix B. LM-OTS Parameter Options

A table illustrating various combinations of n and w with the associated values of u, v, ls, and p is provided in Table 5.

The parameters u, v, ls, and p are computed as follows:

```
u = ceil(8*n/w)
v = ceil((floor(lg((2^w - 1) * u)) + 1) / w)
ls = (number of bits in sum) - (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. The "number of bits in sum" is defined according to Section 4.5. And as the value of p is the number of w-bit elements of (H(message) || Cksm(H(message))), it is also equivalently the number of byte strings that form the private key and the number of byte strings in the signature.

Hash Length in Bytes (n)	Winternitz Parameter (w)	w-bit Elements in Hash (u)	w-bit Elements in Checksum (v)	Left Shift (ls)	Total Number of w-bit Elements (p)
16	1	128	8	8	137
16	2	64	4	8	68
16	4	32	3	4	35
16	8	16	2	0	18
32	1	256	9	7	265
32	2	128	5	6	133
32	4	64	3	4	67
32	8	32	2	0	34

Table 5

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 = hash_final(), in that order, is identical to that of the invocation of H(N[1] || N[2] || ... || N[n]).

Generating an LMS Public Key From an LMS Private Key

```
  for ( i = 0; i < num_lmots_keys; i = i + 1 ) {
    r = i + num_lmots_keys;
    temp = H(I || OTS_PUBKEY[i] || u32str(r) || D_LEAF)
    j = i;
    while (j % 2 == 1) {
      r = (r - 1)/2; j = (j-1) / 2;
      left_size = pop(data stack);
      temp = H(I || left_side || temp || u32str(r) || D_INTR)
    }
    push temp onto the data stack
 }
 public_key = pop(data stack)
```

Note that this pseudocode expects that all 2^h leaves of the tree have equal depth; that is, num_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)*n bytes; for the currently defined parameter sets, this will never be more than 768 bytes of data.

Appendix D. Example Implementation

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

Appendix E. 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

```
--------------------------------------------
HSS public key
levels      00000002
--------------------------------------------
LMS type    00000005                         # LM_SHA256_M32_H5
LMOTS type  00000004                         # LMOTS_SHA256_N32_W8
I           61a5d57d37f5e46bfb7520806b07a1b8
K           50650e3b31fe4a773ea29a07f09cf2ea
            30e579f0df58ef8e298da0434cb2b878
--------------------------------------------
--------------------------------------------
```

Test Case 1 Message

```
--------------------------------------------
Message     54686520706f77657273206e6f742064  |The powers not d|
            656c65676174656420746f2074686520  |elegated to the |
            556e6974656420537461746573206279  |United States by|
            20746865204366e737469747574696f  | the Constitutio|
            6e2c206e6f722070726f686962697465  |n, nor prohibite|
            64206279206974207466f207468652053  |d by it to the S|
            74617465732c20617265207265736572  |tates, are reser|
            76656420746f20746865205374617465  |ved to the State|
            73207265737065637469766656c792c20  |s respectively, |
            6f7220746f2074686520706565f706c65  |or to the people|
            2e0a                              |..|
--------------------------------------------
```

Test Case 1 Signature

```
--------------------------------------------
HSS signature
Nspk        00000001
sig[0]:
--------------------------------------------
LMS signature
q           00000005
--------------------------------------------
LMOTS signature
LMOTS type  00000004                         # LMOTS_SHA256_N32_W8
C           d32b56671d7eb98833c49b433c272586
            bc4a1c8a8970528ffa04b966f9426eb9
y[0]        965a25bfd37f196b9073f3d4a232feb6
            9128ec45146f86292f9dff9610a7bf95
y[1]        a64c7f60f6261a62043f86c70324b770
            7f5b4a8a6e19c114c7be866d488778a0
```

```
   y[2]         e05fd5c6509a6e61d559cf1a77a970de
                927d60c70d3de31a7fa0100994e162a2
   y[3]         582e8ff1b10cd99d4e8e413ef469559f
                7d7ed12c838342f9b9c96b83a4943d16
   y[4]         81d84b15357ff48ca579f19f5e71f184
                66f2bbef4bf660c2518eb20de2f66e3b
   y[5]         14784269d7d876f5d35d3fbfc7039a46
                2c716bb9f6891a7f41ad133e9e1f6d95
   y[6]         60b960e7777c52f060492f2d7c660e14
                71e07e72655562035abc9a701b473ecb
   y[7]         c3943c6b9c4f2405a3cb8bf8a691ca51
                d3f6ad2f428bab6f3a30f55dd9625563
   y[8]         f0a75ee390e385e3ae0b906961ecf41a
                e073a0590c2eb6204f44831c26dd768c
   y[9]         35b167b28ce8dc988a3748255230cef9
                9ebf14e730632f27414489808afab1d1
   y[10]        e783ed04516de012498682212b078105
                79b250365941bcc98142da13609e9768
   y[11]        aaf65de7620dabec29eb82a17fde35af
                15ad238c73f81bdb8dec2fc0e7f93270
   y[12]        1099762b37f43c4a3c20010a3d72e2f6
                06be108d310e639f09ce7286800d9ef8
   y[13]        a1a40281cc5a7ea98d2adc7c7400c2fe
                5a101552df4e3cccfd0cbf2ddf5dc677
   y[14]        9cbbc68fee0c3efe4ec22b83a2caa3e4
                8e0809a0a750b73ccdcf3c79e6580c15
   y[15]        4f8a58f7f24335eec5c5eb5e0cf01dcf
                4439424095fceb077f66ded5bec73b27
   y[16]        c5b9f64a2a9af2f07c05e99e5cf80f00
                252e39db32f6c19674f190c9fbc506d8
   y[17]        26857713afd2ca6bb85cd8c107347552
                f30575a5417816ab4db3f603f2df56fb
   y[18]        c413e7d0acd8bdd81352b2471fc1bc4f
                1ef296fea1220403466b1afe78b94f7e
   y[19]        cf7cc62fb92be14f18c2192384ebceaf
                8801afdf947f698ce9c6ceb696ed70e9
   y[20]        e87b0144417e8d7baf25eb5f70f09f01
                6fc925b4db048ab8d8cb2a661ce3b57a
   y[21]        da67571f5dd546fc22cb1f97e0ebd1a6
                5926b1234fd04f171cf469c76b884cf3
   y[22]        115cce6f792cc84e36da58960c5f1d76
                0f32c12faef477e94c92eb75625b6a37
   y[23]        1efc72d60ca5e908b3a7dd69fef02491
                50e3eebdfed39cbdc3ce9704882a2072
   y[24]        c75e13527b7a581a556168783dc1e975
                45e31865ddc46b3c957835da252bb732
   y[25]        8d3ee2062445dfb85ef8c35f8e1f3371
                af34023cef626e0af1e0bc017351aae2
```

```
   y[26]        ab8f5c612ead0b729a1d059d02bfe18e
                fa971b7300e882360a93b025ff97e9e0
   y[27]        eec0f3f3f13039a17f88b0cf808f4884
                31606cb13f9241f40f44e537d302c64a
   y[28]        4f1f4ab949b9feefadcb71ab50ef27d6
                d6ca8510f150c85fb525bf25703df720
   y[29]        9b6066f09c37280d59128d2f0f637c7d
                7d7fad4ed1c1ea04e628d221e3d8db77
   y[30]        b7c878c9411cafc5071a34a00f4cf077
                38912753dfce48f07576f0d4f94f42c6
   y[31]        d76f7ce973e9367095ba7e9a3649b7f4
                61d9f9ac1332a4d1044c96aefee67676
   y[32]        401b64457c54d65fef6500c59cdfb69a
                f7b6dddfcb0f086278dd8ad0686078df
   y[33]        b0f3f79cd893d314168648499898fbc0
                ced5f95b74e8ff14d735cdea968bee74
   --------------------------------------------
   LMS type     00000005                         # LM_SHA256_M32_H5
   path[0]      d8b8112f9200a5e50c4a262165bd342c
                d800b8496810bc716277435ac376728d
   path[1]      129ac6eda839a6f357b5a04387c5ce97
                382a78f2a4372917eefcbf93f63bb591
   path[2]      12f5dbe400bd49e4501e859f885bf073
                6e90a509b30a26bfac8c17b5991c157e
   path[3]      b5971115aa39efd8d564a6b90282c316
                8af2d30ef89d51bf14654510a12b8a14
   path[4]      4cca1848cf7da59cc2b3d9d0692dd2a2
                0ba3863480e25b1b85ee860c62bf5136
   --------------------------------------------
   LMS public key
   LMS type     00000005                         # LM_SHA256_M32_H5
   LMOTS type   00000004                         # LMOTS_SHA256_N32_W8
   I            d2f14ff6346af964569f7d6cb880a1b6
   K            6c5004917da6eafe4d9ef6c6407b3db0
                e5485b122d9ebe15cda93cfec582d7ab
   --------------------------------------------
   final_signature:
   --------------------------------------------
   LMS signature
   q            0000000a
   --------------------------------------------
   LMOTS signature
   LMOTS type   00000004                         # LMOTS_SHA256_N32_W8
   C            0703c491e7558b35011ece3592eaa5da
                4d918786771233e8353bc4f62323185c
   y[0]         95cae05b899e35dffd71705470620998
                8ebfdf6e37960bb5c38d7657e8bffeef
   y[1]         9bc042da4b4525650485c66d0ce19b31
```

```
                7587c6ba4bffcc428e25d08931e72dfb
   y[2]         6a120c5612344258b85efdb7db1db9e1
                865a73caf96557eb39ed3e3f426933ac
   y[3]         9eeddb03a1d2374af7bf771855774562
                37f9de2d60113c23f846df26fa942008
   y[4]         a698994c0827d90e86d43e0df7f4bfcd
                b09b86a373b98288b7094ad81a0185ac
   y[5]         100e4f2c5fc38c003c1ab6fea479eb2f
                5ebe48f584d7159b8ada03586e65ad9c
   y[6]         969f6aecbfe44cf356888a7b15a3ff07
                4f771760b26f9c04884ee1faa329fbf4
   y[7]         e61af23aee7fa5d4d9a5dfcf43c4c26c
                e8aea2ce8a2990d7ba7b57108b47dabf
   y[8]         beadb2b25b3cacc1ac0cef346cbb90fb
                044beee4fac2603a442bdf7e507243b7
   y[9]         319c9944b1586e899d431c7f91bcccc8
                690dbf59b28386b2315f3d36ef2eaa3c
   y[10]        f30b2b51f48b71b003dfb08249484201
                043f65f5a3ef6bbd61ddfee81aca9ce6
   y[11]        0081262a00000480dcbc9a3da6fbef5c
                1c0a55e48a0e729f9184fcb1407c3152
   y[12]        9db268f6fe50032a363c9801306837fa
                fabdf957fd97eafc80dbd165e435d0e2
   y[13]        dfd836a28b354023924b6fb7e48bc0b3
                ed95eea64c2d402f4d734c8dc26f3ac5
   y[14]        91825daef01eae3c38e3328d00a77dc6
                57034f287ccb0f0e1c9a7cbdc828f627
   y[15]        205e4737b84b58376551d44c12c3c215
                c812a0970789c83de51d6ad787271963
   y[16]        327f0a5fbb6b5907dec02c9a90934af5
                a1c63b72c82653605d1dcce51596b3c2
   y[17]        b45696689f2eb382007497557692caac
                4d57b5de9f5569bc2ad0137fd47fb47e
   y[18]        664fcb6db4971f5b3e07aceda9ac130e
                9f38182de994cff192ec0e82fd6d4cb7
   y[19]        f3fe00812589b7a7ce51544045643301
                6b84a59bec6619a1c6c0b37dd1450ed4
   y[20]        f2d8b584410ceda8025f5d2d8dd0d217
                6fc1cf2cc06fa8c82bed4d944e71339e
   y[21]        ce780fd025bd41ec34ebff9d4270a322
                4e019fcb444474d482fd2dbe75efb203
   y[22]        89cc10cd600abb54c47ede93e08c114e
                db04117d714dc1d525e11bed8756192f
   y[23]        929d15462b939ff3f52f2252da2ed64d
                8fae88818b1efa2c7b08c8794fb1b214
   y[24]        aa233db3162833141ea4383f1a6f120b
                e1db82ce3630b3429114463157a64e91
   y[25]        234d475e2f79cbf05e4db6a9407d72c6
```

```
               bff7d1198b5c4d6aad2831db61274993
   y[26]       715a0182c7dc8089e32c8531deed4f74
               31c07c02195eba2ef91efb5613c37af7
   y[27]       ae0c066babc69369700e1dd26eddc0d2
               16c781d56e4ce47e3303fa73007ff7b9
   y[28]       49ef23be2aa4dbf25206fe45c20dd888
               395b2526391a724996a44156beac8082
   y[29]       12858792bf8e74cba49dee5e8812e019
               da87454bff9e847ed83db07af3137430
   y[30]       82f880a278f682c2bd0ad6887cb59f65
               2e155987d61bbf6a88d36ee93b6072e6
   y[31]       656d9ccbaae3d655852e38deb3a2dcf8
               058dc9fb6f2ab3d3b3539eb77b248a66
   y[32]       1091d05eb6e2f297774fe6053598457c
               c61908318de4b826f0fc86d4bb117d33
   y[33]       e865aa805009cc2918d9c2f840c4da43
               a703ad9f5b5806163d7161696b5a0adc
   --------------------------------------------
   LMS type    00000005                         # LM_SHA256_M32_H5
   path[0]     d5c0d1bebb06048ed6fe2ef2c6cef305
               b3ed633941ebc8b3bec9738754cddd60
   path[1]     e1920ada52f43d055b5031cee6192520
               d6a5115514851ce7fd448d4a39fae2ab
   path[2]     2335b525f484e9b40d6a4a969394843b
               dcf6d14c48e8015e08ab92662c05c6e9
   path[3]     f90b65a7a6201689999f32bfd368e5e3
               ec9cb70ac7b8399003f175c40885081a
   path[4]     09ab3034911fe125631051df0408b394
               6b0bde790911e8978ba07dd56c73e7ee
```

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