Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: September 22, 2016

D. McGrew
M. Curcio
Cisco Systems

March 21, 2016

Hash-Based Signatures
draft-mcgrew-hash-sigs-04

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

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 http://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 September 22, 2016.

Copyright Notice

Copyright (c) 2016 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 (http://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

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 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. An example implementation is given in an appendix.

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 exactly 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. We denote the one-time signature scheme that it incorporates 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 N-time signature system is described in Section 5. Sufficient detail is provided to ensure interoperability. The IANA registry for these signature systems is described in Section 10. Security considerations are presented in Section 11.

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; 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 return one, two, and four byte values, respectively.

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.

If A and B are bytes, then A AND B denotes the bitwise logical and operation.

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.

The i^th byte string 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 H, Leighton and Micali introduce a "security string" that is distinct for each invocation of H. The following fields can appear in a security string:

I - an identifier for the private key. This value is 31 bytes long, and it MUST be distinct from all other such identifiers. It SHOULD be chosen uniformly at random, or via a pseudorandom process, at the time that a key pair is generated, in order to ensure that it will be distinct with probability close to one, but it MAY be a structured identifier.

D - a domain separation parameter, which is a single byte that takes on different values in the different algorithms in which H is invoked. D takes on the following values:

D_ITER = 0x00 in the iterations of the LM-OTS algorithms

D_PBLC = 0x01 when computing the hash of all of the iterates in the LM-OTS algorithm

D_MESG = 0x02 when computing the hash of the message in the LM-OTS algorithms

D_LEAF = 0x03 when computing the hash of the leaf of an LMS tree

D_INTR = 0x04 when computing the hash of an interior node of an LMS tree

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 with a cryptographic strength that matches or exceeds that of the LM-OTS algorithm itself.

i - in the LM-OTS one-time signature scheme, i is the index of the 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 one-time signature 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.

q - in the LM-OTS one-time signature scheme, q is a diversification string provided as input. In the LMS N-time signature scheme, each LM-OTS signature is associated with the leaf of a tree, and q is set to the leaf number (as described below). This ensures that a distinct value of q is used for each distinct LM-OTS public/private keypair. q is represented as a four byte string.

r - in the LMS N-time signature scheme, the node number r associated with a particular node of the 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.

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

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 only 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 4.2), 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 Winternitz parameter; 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 4.6).

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 can be chosen to set the number of bytes in the signature; it has little effect on security. Note however, that there is a larger computational cost to generate and verify a shorter signature. The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix A. A table illustrating various combinations of n, w, p, and ls is provided in Table 1.

4.2. Hashing Functions

The LM-OTS algorithm uses a hash function H that accepts byte strings of any length, and returns an n-byte string.

4.3. Signature Methods

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 signature systems, each of which is identified by a name. 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
LMOTS_SHA256_N16_W1	SHA256-16	16	1	68	8
LMOTS_SHA256_N16_W2	SHA256-16	16	2	68	8
LMOTS_SHA256_N16_W4	SHA256-16	16	4	35	4
LMOTS_SHA256_N16_W8	SHA256-16	16	8	18	0

Table 1

Here SHA256 denotes the NIST standard hash function [FIPS180]. SHA256-16 denotes the SHA256 hash function with its final output truncated to return the leftmost 16 bytes.

4.4. Private Key

The LM-OTS private key consists of an array of size p containing n-byte strings. Let x denote the private key. This private key must be used to sign one and only one message. It must therefore be unique from all other private keys. The following algorithm shows pseudocode for generating x.

Algorithm 0: Generating a Private Key

```
set type to the typecode of the algorithm
set n and p according to the typecode and Table 1
for ( i = 0; i < p; i = i + 1 ) {
  set x[i] to a uniformly random n-byte string
}
return u32str(type) || 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.

4.5. 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.

Each public/private key pair is associated with a single identifier I. This string MUST be 31 bytes long, and be generated as described in Section 3.2. It MUST be generated by a uniform random or pseudorandom process during the LM-OTS key pair generation, unless a structured identifier is provided as an input to the algorithm.

The diversification parameter q is an input to the algorithm, as described in Section 3.2. (In the LMS scheme, this parameter is set to the leaf number, as each LM-OTS key pair is associated with the leaf of a tree.)

The following algorithm shows pseudocode for generating the public key, where the array x is the private key.

Algorithm 1: Generating a Public Key From a Private Key

```
set type to the typecode of the algorithm
set n and p according to the typecode and Table 1
for ( i = 0; i < p; i = i + 1 ) {
  tmp = x[i]
  for ( j = 0; j < 2^w - 1; j = j + 1 ) {
     tmp = H(tmp || I || q || u16str(i) || u8str(j) || D_ITER)
  }
  y[i] = tmp
}
return H(I || q || y[0] || y[1] || ... || y[p-1] || D_PBLC)
```

The public key the value returned by Algorithm 1.

4.6. 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 11. The checksum function Cksm is defined as follows, where S denotes the 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 < u; 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.

4.7. Signature Generation

The LM-OTS signature of a message is generated by first appending the randomizer C, the identifier string I, the diversification string q, and D_MESG to the message, then using H to compute the hash of the resulting string, 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 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.

The identifier string I and diversification string q are the same as in Section 4.5.

Algorithm 3: Generating a Signature From a Private Key and a Message

```
set type to the typecode of the algorithm
set n and p according to the typecode and Table 1
set C to a uniformly random n-byte string
Q = H(C || I || q || D_MESG || 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(tmp || I || q || u16str(i) || u8str(j) || D_ITER)
  }
  y[i] = tmp
}
return u32str(type) || C || q || y[0] || y[1] || ... || 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 4.5.

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.

4.8. 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 series of applications of H using the w-bit values of the message hash and its checksum. This computation should result in a value that matches the provided public key.

Algorithm 4: Verifying a Signature and Message Using a Public Key

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

set type by applying uint32str() to the first four bytes of the
signature

if the type computed from the signature is not equal to the
type of the public key, return INVALID

set n and p according to the type and Table 1

if the signature is not exactly 8 + n * (p+1) bytes long, return
INVALID

parse C, q, and y from the signature as follows:
   type = first 4 bytes
   C = next n bytes
   q = next four bytes
   y[0] = next n bytes
   y[1] = next n bytes
   ...
   y[p-1] = next n bytes
Q = H(C || I || q || D_MESG || message)
for ( i = 0; i < p; i = i + 1 ) {
  a = (2^w - 1) - coef(Q || Cksm(Q), i, w)
  tmp = y[i]
  for ( j = a+1; j < 2^w - 1; j = j + 1 ) {
     tmp = H(tmp || I || q || u16str(i) || u8str(j) || D_ITER)
  }
  z[i] = tmp
}
candidate = H(I || q || z[0] || z[1] || ... || z[p-1] || D_PBLC)
if (candidate = public_key)
  return VALID
else
  return INVALID
```

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 n-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.

An LMS system has the following parameters:

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

n : the number of bytes associated with each node.

There are 2^h leaves in the tree.

5.1. LMS Private Key

An LMS private key consists of 2^h one-time signature private keys and the leaf number of the next LM-OTS private key that has not yet been used. The leaf number is initialized to zero when the LMS private key is created.

An LMS private key MAY be generated pseudorandomly from a secret value, in which case the secret value MUST be at least n 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.

5.2. 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_PUBKEY[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 n-byte string, and the string for the rth node is denoted as T[r] and is defined as

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

The LMS public key is the string u32str(type) || I || T[1]. Section 7 specifies the typecode and more formally defines the format. 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 B.

5.3. LMS Signature

An LMS signature consists of

a typecode indicating the particular LMS algorithm,

an LM-OTS signature, and

an array of 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(type) || lmots_signature || path[0] || path[1] || ... || path[p-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 itself. 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.

5.3.1. 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. Before releasing the signature, the leaf number in the LMS private key MUST be incremented to prevent the LM-OTS private key from being used again. The node number in the signature is set to the leaf number of the LMS private key that was used in the signature. Then the signature and the LMS private key are returned.

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.

One useful time/storage tradeoff is described in Column 19 of [USPTO5432852].

5.4. LMS Signature Verification

An LMS signature is verified by first using the LM-OTS signature verification algorithm 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 node array (path[]) as described below. If the root value matches the public key, then the signature is valid; otherwise, the signature fails.

Algorithm 5: LMS Signature Verification

set pubtype by applying uint32str() to the first four bytes of the public key

set sigtype by applying uint32str() to the first four bytes of the signature

if pubtype is not equal to sigtype, return INVALID; otherwise, continue

identify the height h of the tree from pubtype

parse q from the LM-OTS signature in the LMS signature

parse the path from the LMS signature

parse the LMS public key value from the lms_public_key

```
tmp = candidate public key computed from LM-OTS signature and message
tmp = H(tmp || I || u32str(node_num) || D_LEAF)
i = 0
n = node number = 2^h + q
while (node_num > 1) {
    if (node_num is odd):
        tmp = H(path[i] || tmp || I || u32str(node_num/2) || D_INTR)
    else:
        tmp = H(tmp || path[i] || I || u32str(node_num/2) || D_INTR)
    node_num = node_num/2
    i = i + 1
if (tmp == LMS public key vaule)
  return 1  // message/signature pair is valid
else
  return 0  // message/signature pair is invalid
```

Upon completion, v contains the value of the root of the LMS tree for comparison.

The verifier MAY cache interior node values that have been computed during a successful signature verification for use in subsequent signature verifications. However, any implementation that does so MUST make sure any nodes that are cached during a signature verification process are deleted if that process does not result in a successful match between the root of the tree and the LMS public key.

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 scheme 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. Each level is associated with an LMS public key, private key, and signature. The number of levels denoted L, and is between two and eight, inclusive. The following notation is used, where i is an integer between 1 and L inclusive:

prv[i] is the private key of the ith level,

pub[i] is the public key of the ith level,

sig[i] is the signature of the ith level, and

info[i] is lms_key_info, a structure containing information describing the public key of the ith level, including the LMS algorithm type, OTS algorithm type, and the Identifier I.

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

6.1. Key Generation

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=1, ..., L. These key pairs MUST be generated independently, and each key pair MUST use a distinct Identity I.

The public key of the HSS scheme is pub[1], the public key of the first level, followed by the lms_key_info structures for the remaining levels.

The HSS private key consists of prv[1], ... , prv[L]. The values pub[1] and prv[1] do not change, though the values of pub[i] and prv[i] are dynamic for i > 1, 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:

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

The value of the HSS signature is set to sig[1] || ... || sig[L].

If prv[L] is exhausted, then the key pair associated with that level is regenerated as follows. A new LMS public and private key pair with the same algorithm parameters is generated, and pub[L] and prv[L] are set to those values. pub[L] is signed with prv[L-1], and sig[L-1] is set to the resulting value. If prv[L-1] is exhausted, then the regeneration process is applied to that key, and so on for L-1 up to 2.

6.3. Signature Verification

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

The signature sig is parsed into its components sig[1] || ... || sig[L].

The signature sig[L] and message is verified using the public key pub[L]. If verification fails, then an indication of failure is returned. Otherwise, processing continues as follows.

The signature sig[L-1] of the "message" pub[L] is verified using the public key pub[L-1]. If verification fails, then an indication of failure is returned. Otherwise, this process is repeated for all levels from L-1 down to 2.

The signature sig[1] of the "message" pub[2] is verified using the value of the HSS public key. If verification fails, then an indication of failure is returned. Otherwise, an indication of success is returned.

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 signature primitives
 */
enum ots_algorithm_type {
  ots_reserved         = 0,
  lmots_sha256_n16_w1  = 1,
  lmots_sha256_n16_w2  = 2,
  lmots_sha256_n16_w4  = 3,
  lmots_sha256_n16_w8  = 4,
  lmots_sha256_n32_w1  = 5,
  lmots_sha256_n32_w2  = 6,
  lmots_sha256_n32_w4  = 7,
  lmots_sha256_n32_w8  = 8
};

typedef opaque bytestring4[4];
typedef opaque bytestring16[16];
typedef opaque bytestring32[32];

struct lmots_signature_n16_p265 {
  bytestring16 C;
  bytestring4  q;
  bytestring16 y[265];
};

struct lmots_signature_n16_p133 {
  bytestring16 C;
  bytestring4  q;
  bytestring16 y[133];
};

struct lmots_signature_n16_p67 {
  bytestring16 C;
  bytestring4  q;
  bytestring16 y[67];
```

```
  };

  struct lmots_signature_n16_p34 {
    bytestring16 C;
    bytestring4  q;
    bytestring16 y[34];
  };

  struct lmots_signature_n32_p265 {
    bytestring32 C;
    bytestring4  q;
    bytestring32 y[265];
  };

  struct lmots_signature_n32_p133 {
    bytestring32 C;
    bytestring4  q;
    bytestring32 y[133];
  };

  struct lmots_signature_n32_p67 {
    bytestring32 C;
    bytestring4  q;
    bytestring32 y[67];
  };

  struct lmots_signature_n32_p34 {
    bytestring32 C;
    bytestring4  q;
    bytestring32 y[34];
  };

  union ots_signature switch (ots_algorithm_type type) {
   case lmots_sha256_n16_w1:
     lmots_signature_n16_p256 sig_n16_p265;
   case lmots_sha256_n16_w2:
     lmots_signature_n16_p133 sig_n16_p133;
   case lmots_sha256_n16_w4:
     lmots_signature_n16_p67  sig_n16_p67;
   case lmots_sha256_n16_w8:
     lmots_signature_n16_p43  sig_n16_p34;
   case lmots_sha256_n32_w1:
     lmots_signature_n32_p256 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_p43  sig_n32_p34;
   default:
     void;   /* error condition */
  };

  union ots_private_key switch (ots_algorithm_type type) {
   case lmots_sha256_n16_w1:
   case lmots_sha256_n16_w2:
   case lmots_sha256_n16_w4:
   case lmots_sha256_n16_w8:
        bytestring16 x16;
   case lmots_sha256_n32_w1:
   case lmots_sha256_n32_w2:
   case lmots_sha256_n32_w4:
   case lmots_sha256_n32_w8:
        bytestring32 x32;
   default:
     void;   /* error condition */
   };

  /* leighton micali signature (lms) data types  */

  enum lms_algorithm_type {
    lms_reserved       = 0,
    lms_sha256_n32_h20 = 1,
    lms_sha256_n32_h10 = 2,
    lms_sha256_n32_h5  = 3,
    lms_sha256_n16_h20 = 4,
    lms_sha256_n16_h10 = 5,
    lms_sha256_n16_h5  = 6
  };

  union lms_path switch (lms_algorithm_type type) {
   case lms_sha256_n32_h20:
     bytestring32 path_n32_h20[20];
   case lms_sha256_n32_h10:
     bytestring32 path_n32_h10[10];
   case lms_sha256_n32_h5:
     bytestring32 path_n32_h5[5];
   case lms_sha256_n16_h20:
     bytestring16 path_n16_h20[20];
   case lms_sha256_n16_h10:
     bytestring16 path_n16_h10[10];
   case lms_sha256_n16_h5:
     bytestring16 path_n16_h5[5];
   default:
     void;     /* error condition */
  };
```

```
  struct lms_signature {
    ots_signature lmots_sig;
    lms_path nodes;
  };

  struct lms_key_n16 {
    ots_algorithm_type ots_alg_type;
    opaque I[31];
    opaque value[16];
  };

  struct lms_key_n32 {
    ots_algorithm_type ots_alg_type;
    opaque I[31];
    opaque value[32];
  };

  union lms_public_key switch (lms_algorithm_type type) {
   case lms_sha256_n32_h20:
   case lms_sha256_n32_h10:
   case lms_sha256_n32_h5:
        lms_key_n32 z_n32;
   case lms_sha256_n16_h20:
   case lms_sha256_n16_h10:
   case lms_sha256_n16_h5:
        lms_key_n16 z_n16;
    default:
     void;     /* error condition */
  };

  struct lms_key_info {
    lms_algorithm_type lms_alg_type;
    ots_algorithm_type ots_alg_type;
    opaque I[31];
  };

  union lms_private_key switch (lms_algorithm_type type) {
   case lms_sha256_n32_h20:
   case lms_sha256_n32_h10:
   case lms_sha256_n32_h5:
        lms_key_n32 z_n32;
   case lms_sha256_n16_h20:
   case lms_sha256_n16_h10:
   case lms_sha256_n16_h5:
        lms_key_n16 z_n16;
    default:
     void;     /* error condition */
  };
```

```
/*
 * hierarchical signature scheme (hss)
 */

struct hss_public_key {
  lms_public_key pub;
  lms_key_info   info<7>;
};

struct hss_private_key {
  lms_private_key pub<8>;
};

struct hss_signature {
  lms_signature sig<8>;   /* maximum of eight levels */
};

/*
 * hss_data_type and hss_data provide high level data types for all of
 * the objects that might need to be stored on a file system, as a
 * convenience to the implementer
 */
enum hss_data_type {
  reserved             = 0,
  type_lms_public_key  = 1,
  type_lms_private_key = 2,
  type_hss_public_key  = 3,
  type_hss_private_key = 4,
  type_hss_signature   = 5
};

union hss_data switch (hss_data_type type) {
 case type_lms_public_key:
   lms_public_key lms_public_key_data;
 case type_lms_private_key:
   lms_private_key lms_private_key_data;
 case type_hss_public_key:
   hss_public_key hss_public_key_data;
 case type_hss_private_key:
   hss_private_key hss_private_key_data;
 case type_hss_signature:
   hss_signature hss_signature_data;
 default:
   void;
};
```

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 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.

The layout of the data inside of public keys, signatures, and private keys is illustrated below, using the following notation. Each line describes a single object, and indentation is used to show that an object is contained in another object. Some of these objects do not appear explicitly in the data format, as they are merely logical groupings. Objects that do appear explicitly are indicated by an asterisk (*). The lengths of some objects is variable, and some object names are incomplete (because more than one name might appear), so this diagriam is meant as a conceptual aid only, and not a precise definition.

```
hss_public_key
   * hss_algorithm_type
     lms_public_key
        * lms_algorithm_type
          lms_public_key_n
             * ots_algorithm_type
             * I
             * value
     lms_key_info
        * lms_algorithm_type
        * ots_algorithm_type
        * I
```

```
hss_private_key
   * hss_algorithm_type
     lms_private_key
        * lms_algorithm_type
          lms_public_key_n
             * ots_algorithm_type
             * I
             * value
     lms_private_key
        * lms_algorithm_type
          lms_public_key_n
             * ots_algorithm_type
             * I
             * value
```

```
   hss_signature
      * hss_algorithm_type
        lms_public_key
           * lms_algorithm_type
             lms_key_n
                * ots_algorithm_type
                * I
                * value
        lms_signature
           ots_signature
              * ots_algorithm_type
              * C
              * q
              * y[p]
           lms_path
              * lms_algorithm_type
              * path[h]
        lms_signature
           ots_signature
              * ots_algorithm_type
              * C
              * q
              * y[p]
           lms_path
              * lms_algorithm_type
              * path[h]
```

8. Rationale

The goal of this note is to describe the LM-OTS and LMS 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 required to be distinct in order to improve security. That distinctness ensures the uniqueness of the inputs to H across all of those public/private key pair instances, which is important for provable security in the random oracle model. The length of I is set at 31 bytes so that randomly chosen values of I will be distinct with probability at least 1 - 1/2^128 as long as there are 2^60 or fewer instances of LMS public/private key pairs.

The sizes of the parameters in the security string are such that, for n=16, the LM-OTS iterates a 55-byte value (that is, the string that is input to H() during the iteration over j during signature generation and verification is 55 bytes long). Thus, when SHA-256 is used as the function H, only a single invocation of its compression function is needed.

The signature and public key formats are designed so that they are easy to parse. Each format starts with a 32-bit enumeration value that indicates all of the details of the signature algorithm and hence defines all of the information that is needed in order to parse the format.

The Checksum Section 4.6 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 4.7 and the total actual number. In the worst case (i.e. the actual number of times H is iteratively applied is 0), the sum is (2^w - 1) * ceil(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 A, which indicates the number of bits used in the left-shift operation.

9. History

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

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 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_n32_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. These number assignments SHOULD use the smallest available palindromic number. 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_N16_W1	Section 4	0x00000001
LMOTS_SHA256_N16_W2	Section 4	0x00000002
LMOTS_SHA256_N16_W4	Section 4	0x00000003
LMOTS_SHA256_N16_W8	Section 4	0x00000004
LMOTS_SHA256_N32_W1	Section 4	0x00000005
LMOTS_SHA256_N32_W2	Section 4	0x00000006
LMOTS_SHA256_N32_W4	Section 4	0x00000007
LMOTS_SHA256_N32_W8	Section 4	0x00000008

Table 2

The LMS registry is as follows.

Name	Reference	Numeric Identifier
LMS_SHA256_N32_H20	Section 5	0x00000001
LMS_SHA256_N32_H10	Section 5	0x00000002
LMS_SHA256_N32_H5	Section 5	0x00000003
LMS_SHA256_N16_H20	Section 5	0x00000004
LMS_SHA256_N16_H10	Section 5	0x00000005
LMS_SHA256_N16_H5	Section 5	0x00000006

Table 3

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

11. Security Considerations

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.

LM-OTS and LMS are provably secure in the random oracle model, as shown by Katz [Katz15]. From Theorem 8 of that reference:

> For any adversary attacking arbitrarily many instances of the one-time signature scheme, and making at most q hash queries, the probability with which the adversary is able to forge a signature with respect to any of the instances is at most q2^(1-8n).

Here n is the number of bytes in the output of the hash function (as defined in Section 4.1). Thus, the security of the algorithms defined in this note can be roughly described as follows. For a security level of roughly 128 bits, assuming that there are no quantum computers, use n=16 by selecting an algorithm identifier with N16 in its name. For a security level of roughly 128 bits, assuming that there are quantum computers that can compute the input to an arbitrary function with computational cost equivalent to the square root of the size of the domain of that function [Grover96], use n=32 by selecting an algorithm identifier with N32 in its name.

11.1. 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.

11.2. 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 4.7). 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-1), 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 $F^b(x[k]) = y[k]$.

12. Acknowledgements

Thanks are due to Chirag Shroff, Andreas Hulsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, Russ Housley, and Scott Fluhrer for constructive suggestions and valuable detailed review. We esepcially 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.

13. References

13.1. 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>.

[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>.

[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.

13.2. 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.

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

[Katz15] Katz, J., "Analysis of a proposed hash-based signature standard", Contribution to

IRTF http://www.cs.umd.edu/~jkatz/papers/HashBasedSigs.pdf, 2015.

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

Appendix A. 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 4.

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 4.6. 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 4

Appendix B. 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 and a separate stack of integers to keep track of the level of the tree. 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 + 2 ) {
  level = 0;
  for ( j = 0; j < 2; j = j + 1 ) {
    r = node number
    push H(OTS_PUBKEY[i+j] || I || u32str(r) || D_LEAF) onto data stack
    push level onto the integer stack
  }
  while ( height of the integer stack >= 2 ) {
    if level of the top 2 elements on the integer stack are equal {
      hash_init()
      siblings = ""
      repeat ( 2 ) {
        siblings = (pop(data stack) || siblings)
        level = pop(integer stack)
      }
      hash_update(siblings)
      r = node number
      hash_update(I || u32str(r) || D_INTR)
      push hash_final() onto the data stack
      push (level + 1) onto the integer stack
    }
  }
}
public_key = pop(data stack)
```

Note that this pseudocode expects that all 2^h leaves of the tree have equal depth. Neither stack ever contains more than h+1 elements. For typical parameters, these stacks will hold around 512 bytes of data.

Appendix C. Example implementation

--- Pending Revision ---

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