Hash-Based Signatures
draft-mcgrew-hash-sigs-06

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 6, 2017.

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 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 ith element in an array A is denoted as A[i].

. **Strings of w-bit elements**

   If S is a byte string, then byte(S, i) denotes its ith byte, where
   byte(S, 0) is the leftmost byte. In addition, bytes(S, i, j) denotes
   the range of bytes from the ith to the jth 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 ith, 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. The following fields can appear in a security string:

I - an identifier for the LMS public/private key pair. The length of this value varies based on the LMS parameter set and it MUST 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 from the identifier of any other LMS private key with probability close to one.

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

D_PRG = 0x05 in the recommended pseudorandom process for generating LMS private keys

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.

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.

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.

i - in the LM-OTS 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 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.

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,
the length LenS of the security string S, 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 | LenS | p   | ls |
+----------------------+--------+----+---+------+-----+----+
| LMOTS_SHA256_N32_W1  | SHA256 | 32 | 1 | 68   | 265 | 7  |
|                      |        |    |   |      |     |    |
| LMOTS_SHA256_N32_W2  | SHA256 | 32 | 2 | 68   | 133 | 6  |
|                      |        |    |   |      |     |    |
| LMOTS_SHA256_N32_W4  | SHA256 | 32 | 4 | 68   | 67  | 4  |
|                      |        |    |   |      |     |    |
| LMOTS_SHA256_N32_W8  | SHA256 | 32 | 8 | 68   | 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 a LenS-byte security string S. 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. if no security string S has been provided as input, then set S to
       a LenS-byte string generated uniformly at random

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

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

    5. return u32str(type) || S || 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 and S 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(S || tmp || u16str(i) || u8str(j) || D_ITER)
        }
        y[i] = tmp
      }
      K = H(S || y[0] || ... || y[p-1] || D_PBLC)
```

   5. return u32str(type) || S || 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 randomizer C and the security string S to the message, then appending D_MESG to the resulting string then computing its hash, 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 and S from the private key

   4. set C to a uniformly random n-byte string

   5. compute the array y as follows:
      Q = H(S || C || message || D_MESG)
      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(S || tmp || u16str(i) || u8str(j) || D_ITER)
        }
        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, S, 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 4 + LenS + n bytes long,
          return INVALID

       c. S = next LenS bytes of public key

       d. K = next n bytes of public key

    3. compute the public key candidate Kc from the signature,
       message, and the security string S 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 a Security String S

    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(S || C || message || D_MESG)
       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(S || tmp || u16str(i) || u8str(j) || D_ITER)
         }
         z[i] = tmp
       }
       Kc = H(S || z[0] || z[1] || ... || z[p-1] || D_PBLC)

    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 jth 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 MUST be the same as the hash function used within the LM-OTS system used to generate the leaves. This is required because both use the same I value, and hence must have the same length of I value (and the length of the I value is dependent on the hash function).

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_H24	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 qth element of OTS_PRIV[] is
   generated using Algorithm 0 with the security string S = 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) {
            S = I || q
            OTS_PRIV[q] = LM-OTS private key with security string S
          }

      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 ith 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 rth node is denoted as
   T[r] and is defined as

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

   a typecode indicating the particular LMS algorithm,

   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

   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) 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
   security string S = I || q

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

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

      sig[i] is the LMS signature of the ith 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.

   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[64];
  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 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 or 64 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 the
hashes computed by both LM and LM-OTS start with a fixed 64-byte I
value. The reason this size was selected was to allow an
implementation to compute the intermediate hash state after
processing I once (similar to the well-known optimization for HMAC),
and hence the majority of hashes computed during LM-OTS processing
can be performed using a single hash compression operation when using
SHA-256. Other hash functions, which may be used in future
specifications, can use a similar strategy, as long as I is long
enough that it is very unlikely to repeat if chosen uniformly at
random.

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 fifth version of this draft. It has the following
changes from previous versions:

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, as shown by Katz
[Katz16]. From Theorem 2 of that reference:

> For any adversary attacking the LMS scheme and making at most q
> hash queries, the probability the adversary forges a signature is
> at most $3*q/2^{(8*n)}$.

Here n is the number of bytes in the output of the hash function (as
defined in Section 4.1). The security of all of the the algorithms
and parameter sets defined in this note is roughly 128 bits, even
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].

The format of the inputs to the hash function H have the property
that each invocation of that function has an input that is distinct
from all others, with very high probability. This property is
important for a proof of security in the random oracle model. The
formats used during key generation and signing are

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

Because the suffixes D_ITER, D_PBLC, D_LEAF, D_INTR, and D_PRG are
distinct, the input formats ending with different suffixes are all
distinct. It remains to show the distinctness of the inputs for each
suffix.

The values of I and C are chosen uniformly at random from the set of
all n*8 bit strings. For n=32, it is highly likely that each value
of I and C will be distinct, even when 2^{96} such values are chosen.

For D_ITER, D_PBLC, and D_MESG, the value of S = I || u32str(q) is
distinct for each LMS leaf (or equivalently, for each q value). For
D_ITER, 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 S, 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 D_PRG,
for a given value of I, q and j are distinct for each invocation of H
(note that x_q[0] = SEED when j=0).

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 jth 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, because the hash operation dominates any
measure of 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", BCP 79, 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", BCP 79, 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.

   [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,

      D_PRG is a diversification constant, 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) || SEED || u16str(j) || D_PRG).

   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.

[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 public key
LMS type    00000005                           # LMS_SHA256_M32_H5
LMOTS_type  00000004                           # LMOTS_SHA256_N32_W8
I           a5f1da931d9acad25800936e78400a9f
            35e42c3026a95f52c3380dcec2cedc86
            67c3d6060c407aea9101c37298e38c31
            b54d8bb61a2c9668d01216814cc3788c
K           348ed79a731eabe47a3cd7ab603ef8de
            6db2e83eaa08fe742cdeb36e635590e2
----------------------------------------------
----------------------------------------------
```

Test Case 1 Message

```
----------------------------------------------
Message     54686520706f77657273206e6f742064  |The powers not d|
            656c65676174656420746f2074686520  |elegated to the |
            556e6974656420537461746573206279  |United States by|
            2074686520436f6e737469747574696f  | the Constitutio|
            6e2c206e6f722070726f686962697465  |n, nor prohibite|
            6420627920697420746f207468652053  |d by it to the S|
            74617465732c20617265207265736572  |tates, are reser|
            76656420746f20746865205374617465  |ved to the State|
            73207265737065637469766c792c20    |s respectively, |
            6f7220746f207468652070656f706c65  |or to the people|
            2e0a                              |..|
----------------------------------------------
```

Test Case 1 Signature

```
----------------------------------------------
HSS signature
Nspk        00000001
sig[0]:
----------------------------------------------
LMS signature
q           00000001
----------------------------------------------
LMOTS signature
LMOTS type  00000004                           # LMOTS_SHA256_N32_W8
C           c638b5aa5d3ebec1648986cff65a1b2e
            7213487c25c6fe15b1c859603f741e16
```

```
y[0]         b11e8ec40acfc44e74248c312cc8b027
             7fb992afb099f43cd69675b7bd6c22aa
y[1]         84ddb5ceade53f2097dae9b124be8773
             b275d470efa1038437378d8756092b17
y[2]         1bd8bac797db1a3e977f28e73aff1c3b
             94bd3dacca4af4384b6271742e25c841
y[3]         9a9d179629c2b966c0eb25a998243094
             d5f1a7185c0fdf0d9bf9dfa707cbae82
y[4]         545c4e5e2d86db1fad025f41e13276d0
             d28559d5ab81bd81fc97b63f914e1606
y[5]         ddd89cd611fe2a766f4e98d5932c1a27
             1d879592794f84e7decfcef6e9f00d0d
y[6]         2e20b82d50149fc5a5fe2a4c42e1dd10
             85e9a151c9bc11417b388a2b7018ec1a
y[7]         731c1077e54f8b8eba828d3a3462ed6c
             f340c7e8a93364df9174127a57463ea1
y[8]         ad3c122d9eb92e29dd97b1a0f9165a09
             c1f1f5eb4d0315d287fdcbff30a4fe15
y[9]         59eb238bb17c0583df83c5aac1cf5a85
             d72c12e2522090b5a130c4e580687b97
y[10]        62d897571b95c3c61d7dac8168a60a1e
             c1c38879129d30c99ecccf51edd0699e
y[11]        170b88ba98253729134e00e81e523f82
             ef5eaba611a10c3955eb0548918cd103
y[12]        fc40ee27c672af4fbc42f314cb1fc0c1
             5d42a6372bbe83b22f9334629b4af452
y[13]        00b60c768eb1cb888220ee2c4f08ba59
             bbb4b7793a5651e3dd10ef4b0bb5ed24
y[14]        9740e05d35f8670ff6271c5503a6be87
             7561f9e6f4c81e1b903e5048b20b5fb2
y[15]        dad7f51142c23faa4ecd2774b2e25fee
             73a93f02466c3fb9d80b10e4becf7d81
y[16]        1b6a0f4590231de56e0275466790feb0
             26f15e65c26dc45beb908afdba13e560
y[17]        46cac18acb86b10f96a5fcb59b07999c
             04f6febe461220c544dcc8328767c5a0
y[18]        01e434d65bc787ffd952f1404496f3f1
             dd91260e929c60c2725bde980438e591
y[19]        c0eb0788c2d40a867028f1109b80f6a3
             32c4c54ef39078df71a89dda43053c36
y[20]        c13d2ffb54c5b236d32eb07ea08ea3eb
             147fca0367512330736781d028756e53
y[21]        2b4e109b812789d44079e8f3c7833362
             4c0b5255b14057404168710a802cedd1
y[22]        b39be11a52cfbb522b17e796004ae6a7
             0c17aee15eb0d8f8239c5c95d3143633
y[23]        92d30c6c2268f27eeb0f64ff46312e47
             8ca388c37d895d1850f8abb5ac4f4d62
```

```
y[24]       39eac305ec8fd13a4a1f537b46e71d26
            3ee4ff2066256b8f1facf42d90e439a2
y[25]       2511733d1c27a3a76fd6d34b8c2d6c98
            419756af39148825a60c0bab0dc5e44d
y[26]       eb282478ecde2460b045e0b4f1649b23
            24eb21570d2804ebb331fef94b6a09d4
y[27]       f6139d54e2ec15b5c770ae0dda018748
            82f0a04e8d61d7f7985668fad9295aa8
y[28]       b851fa7a223c9bd8b7badb46ba7a6474
            e269f0261693af2589f2ba948616946d
y[29]       7d9e09f8c2d2311884469b0910990cc1
            952eba6dcf6ffbd7fe348c79698b9e74
y[30]       01f370a89c4de025393ccdd6ea4278e3
            07dd69025a77ad13f91d55dd8b11d320
y[31]       9b10acf760ca29f58866836dfbc00e1c
            790d63bac8cdea86408df23a7c780259
y[32]       db23d2482b65f2f4f5613660ef7a27e1
            a4cc4cd695fe7cd52be2c5f1a7140a38
y[33]       59f431952579592822aa15389fffb05d
            3528f92b91a8f376a5af2cb61fd8d2c5
---------------------------------------------
LMS type    00000005                            # LMS_SHA256_M32_H5
path[0]     76b85fb075704d6cd66c6d9c48c512ad
            5a41e84ef199ff2d07300400357a032d
path[1]     ef12462838a0fe139bb8b429eeb4e76e
            09b704611bdbb30c107db13076e52ee6
path[2]     055b20ae2af30d52b9e0d1194b979b5f
            897f23437a33c0f3099a4fe0f79662b8
path[3]     1fbd4cbf61a92e5eb45fa68358410cb7
            812540c560ed7bd2256cc912a80f5260
path[4]     6b60e09d773b729d806ace549227b376
            2fa7a55942b07a77b165e0d729899617
pub[0]:
---------------------------------------------
LMS public key
LMS type    00000005                            # LMS_SHA256_M32_H5
LMOTS_type  00000004                            # LMOTS_SHA256_N32_W8
I           9fc3084bbea5e6d31af8586bc14d8154
            f5532b14745e196dcadd820aa11ea137
            f06a326778eeb875c6035934ee6470ae
            8bfa18f1a1d36e1553f28aa87b878006
K           2d7920997295fc74ad49ea4c5ad6735e
            1e967c966766924b799e734ae922989a
---------------------------------------------
final_signature:
---------------------------------------------
LMS signature
q           00000009
```

```
--------------------------------------------
LMOTS signature
LMOTS type  00000004                       # LMOTS_SHA256_N32_W8
C           8c721faaa063d1c0a5acef3cc83b4f3a
            a3c3863586030c2fb1abdbbff08baf34
y[0]        36a7fc7f0287f1fc10ca471502bae902
            bed6be97b576ef330e119bc93f043811
y[1]        d5de1e0a4431f850d1d264bf880628aa
            9f53c66a23b3f87075651dfc4a05de3e
y[2]        bc8a1addc634dc1f38f27dbfee708169
            78007e9400618586b715c15ca153a1fa
y[3]        1d3a4711354893db705500d8d2b4ae98
            3fc358de7817ba6da1baaee64e670f43
y[4]        7fe3675543c548d8e3b23430b86dfb16
            27164c4b953086bc544ebcbef54c9437
y[5]        f79837dcc32e158f7858c5ad3c09628c
            b1715ae69c3489cf617527956385f7c9
y[6]        bf1a7365629691b10499e39405b07edc
            3464fd71170af8e50e06f644778b337e
y[7]        42b3a15affcd482de83dc1d408cfdf4a
            2b0e4566a09eaaae8269a0695c00b1a7
y[8]        3e482cf25b44d65474276cfc34f7991d
            15cb1defb2236fa7b697362cd9e6d1e0
y[9]        5dd1342b137d7d3a54374dba7ba5741e
            1aaa2831ff62dfdf52b8aee2559fb27c
y[10]       aebe546a5006b857692c32f0f6a8386d
            96646631e953942126d7793715245caa
y[11]       1704d819e50f2a2ec6c1271ed47db819
            b8ea3529a343818ec58c14206bbb5eea
y[12]       681897efa723779ffd970ee4d8841bee
            c87cf9cc14a5369d3196a3331e057be4
y[13]       e7b4c26fa6e74c916cd73be77406812d
            7dd1258e14dcf4ebb2b137d5f9a1d628
y[14]       e4d661b240c0c6f75e954e1872c2d135
            cb0b758c270b42193ab9838c360c8dc5
y[15]       43b7dfd7e6d49778f3eeb328ddb57078
            f24610b710ba20a01fccdec1f3f02763
y[16]       776ddbd8c82e25f6ab0f46cd1f776ffc
            00c1c55ef5f2429ad12501a8ad876901
y[17]       1d51dee1851abc129fa99aae096d1da1
            8acb95f7f78b5adeaaa4d4ea53984b1a
y[18]       a562394d39c479b93fea1db213e3685a
            8a9368b16fd4b3086729f61ec3d65ff8
y[19]       f4f634d430522606761ee1ad522f5a86
            573c5e7b0f6aeb90d1bdfb0cdec61272
y[20]       52b4b07683a59441377899e9558f5181
            56318c83fb6a9c1c0a49b43d3ae08dec
y[21]       221d0f3bc0230d9c080e06bddfce2f12
```

```
                    3b0bc012644aed82f4d565564461d814
      y[22]         62c401a74d41959720dd05dc717d3bcd
                    2790ddd2af0e4d6214990b0fee5fdaed
      y[23]         8af103391e6edceb8d08554249092ebe
                    949f8b1671ceabb7f6a991163da95372
      y[24]         0b384b59c8589030165bb90917b9a9a7
                    9462eecf5f6196280d23129011ddbd5e
      y[25]         4c99f50a7ae2cf8debc7d0034c39eb3f
                    33b67889073c62b7fbcccadc4921763c
      y[26]         512a485d8cc78f80a783a84348e17411
                    7a4e3716319316a2eb42c014a54616e8
      y[27]         40156b0d511f8762c3d2a0a3946e0b6f
                    993320206c930980cd6a9751e57c62dc
      y[28]         aa1cf6303ca775d71a91629bd904ac20
                    35226dc9d5b653dcd30673738374829f
      y[29]         f57d72293c0f1b3666004667248881bd
                    9338b59b049f4e0091f5d39879fca9b6
      y[30]         6c0d4b4eb19d9e63fef18f5657974ff4
                    d36bf23055dcb6ed4f7e5ce1ad04bfac
      y[31]         e91630344345eea1470efb49e4854411
                    8a09561d498e90a50c8d68c3e726d15b
      y[32]         f20871eaa508b929a5210bc027c92038
                    07a94c1cae545a97baf6dd961eddb72f
      y[33]         5fd33572aae2da10093c3600e26ead7e
                    eaa9e1dce4f253985f4f922b77057535
      ---------------------------------------------
      LMS type      00000005                          # LMS_SHA256_M32_H5
      path[0]       e89d230cd37998a27929b8ac966a76c6
                    73ae712267ab51ee82c754dc583efb34
      path[1]       a6f3e4f96984891c7bbc80468a88aedd
                    e5e6661e32d84c106f5353d660092428
      path[2]       affef3d925d9f0da2b7a5bbafc5099e2
                    169b29695c69a425bab93ece3fcfa376
      path[3]       75c32f006ef4599340508179caa9da3c
                    574b16721535ce74b1e287e507aab414
      path[4]       0ea5e46102296e0bb564d99520b5593f
                    25c07a581408d453ce99d615f565ebc2
```

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