Hash-Based Signatures
draft-mcgrew-hash-sigs-01

Abstract

   This note describes a digital signature system based on cryptographic
   hash functions, following the seminal work in this area. It
   specifies a one-time signature scheme based on the work of Lamport,
   Diffie, Winternitz, and Merkle (LDWM), and specifies a general
   signature system using a Merkle tree. These systems provide
   asymmetric authentication without using large integer mathematics and
   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.

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, and have benefited from
   renewed development in the last decade. The characteristics of these
   signature systems are small private and public keys and low
   computational cost, but large signatures. In recent years there has
   been interest in these systems because of their post-quantum security
   (see Section 8.3) and their suitability for compact implementations.

   This note describes the original Lamport-Diffie-Winternitz-Merkle
   (LDWM) one-time signature system (following Merkle 1979 but also
   using a technique from Merkle's later work [C:Merkle87][C:
   Merkle89a][C:Merkle89b]) and Merkle tree signature system (following
   Merkle 1979) with enough specificity to ensure interoperability
   between implementations. While the specification of the algorithms
   follows these early references, the security considerations makes use
   of more recent security analyses (especially Buchmann, Dahmen, Ereth,
   Hulsing, and Ruckert 2011).

   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. A general signature system can be used to sign multiple
   messages. The Merkle Tree Signatures (MTS) is a general signature
   system that uses an OTS system as a component. In principle the MTS
   can be used with any OTS system, but in this note we describe its use
   with the LDWM system.

   This note is structured as follows. Notation is introduced in
   Section 2. The LDWM signature system is described in Section 3, and
   the Merkle tree signature system is described in Section 4.
   Sufficient detail is provided to ensure interoperability. Appendix B
   describes test considerations and contains test cases that can be
   used to validate an implementation. The IANA registry for these
   signature systems is described in Section 7. Security considerations
   are presented in Section 8.

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. Notation

2.1. Data Types

   Bytes and byte strings are the fundamental data type. 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.

   A byte can be considered as a sequence of eight bits, a sequence of
   four duets (two bit unsigned integers), or a sequence of two quartets
   (four bit unsigned integers). A bit has value 0 or 1. A duet has a
   value between 0 and 3, and a quartet has value between 0 and 15
   decimal (F hexadecimal).

   The correspondence between bytes, bits, duets, and quartets is shown
   below, where X denotes the value of the byte:

```
              Byte (8-bit element)
            +-----------+-----------+
            |           X           |
            +-----------+-----------+

             Bits (1-bit elements)
            +--+--+--+--+--+--+--+--+
            |b0|b1|b2|b3|b4|b5|b6|b7|
            +--+--+--+--+--+--+--+--+

             Duets (2-bit elements)
            +-----+-----+-----+-----+
            | d0  | d1  | d2  | d3  |
            +-----+-----+-----+-----+

            Quartets (4-bit elements)
            +-----------+-----------+
            |    q0     |    q1     |
            +-----------+-----------+
```
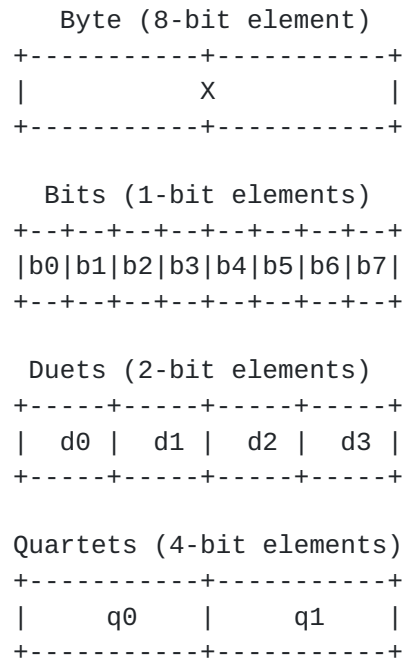
   $X = 128*b0 + 64*b1 + 32*b2 + 16*b3 + 8*b4 + 4*b5 + 2*b6 + b7$
   $X = 64*d0 + 16*d1 + 4*d2 + d3$
   $X = 16*q0 + q1$

For example, the following diagram shows how the byte with decimal
value 27 (hexadecimal value 0x1b) is represented with bits, duets,
and quartets.

```
                  Byte (8-bit element)
             +-----------+-----------+
             |          27           |
             +-----------+-----------+

               Bits (1-bit elements)
             +--+--+--+--+--+--+--+--+
             | 0| 0| 0| 1| 1| 0| 1| 1|
             +--+--+--+--+--+--+--+--+

              Duets (2-bit elements)
             +-----+-----+-----+-----+
             |  0  |  1  |  2  |  3  |
             +-----+-----+-----+-----+

             Quartets (4-bit elements)
             +-----------+-----------+
             |     1     |     b     |
             +-----------+-----------+
```

2.1.1. 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. For example, if S = 0x0204ff, then
byte(S, 0) is 0x02.

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 Quartets)
        +-----------+-----------+-----------+-----------+
        |     1     |     2     |     3     |     4     |
        +-----------+-----------+-----------+-----------+
              ^
              |
         coef(S, 0, 4)
```
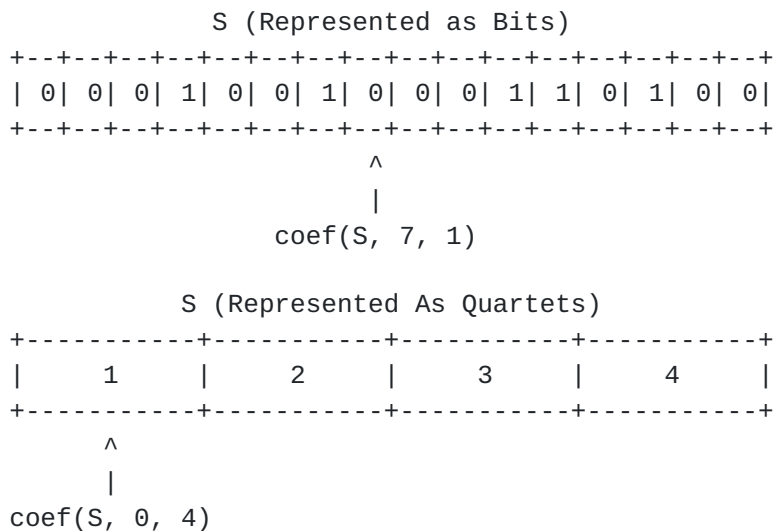
The return value of coef is an unsigned integer. If i is larger than
the number of w-bit values in S, then coef(S, i, w) is undefined, and
an attempt to compute that value should raise an error.

2.2. Operators

When a and b are 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 a divided 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 ith byte string in an array A is denoted as A[i].

2.3. 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

   When F is a function that takes m-byte strings (i.e. byte strings of
   length m) as input and returns m-byte strings as output, we denote
   the repeated applications of F with itself a non-negative, integral
   number of times i as F^i.

   Thus for any byte string x,

      F^i(x) = / F(F^(i-1)(x)) for i > 0
               \ x   for i = 0.

   For example, F^2(x) = F(F(x)).

3.   **LDWM One-Time Signatures**

   This section defines LDWM 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.

   Note that in order to constrain what constitutes the "message" and
   establish fixed sizes for the signature, a digest of the original
   message is computed using a collision-resistant hash function, H (see
   Section 3.2).

3.1.   **Parameters**

   The signature system uses the parameters m, n, and w; they are all
   positive integers. The algorithm description also uses the value p.
   These parameters are summarized as follows:

      m : the length in bytes of each element of an LDWM signature

      n : the length in bytes of result of the hash function

      w : the Winternitz parameter; it is a member of the set
      { 1, 2, 4, 8 }

      p : the number of m-byte string elements that make up the LDWM
      signature; it is equivalent to the number of w-bit elements of
      (H(message) || C(H(message))) as well as the number of n-byte
      strings that form the private key; it can be specified in terms of
      u and v, as shown in Appendix A.

      ls : the number of bits the checksum function C will left-shift
      the sum total of hashing operations required by the verifier
      before returning a result

   The values of m and n are determined by the functions selected for
   use as part of the LDWM algorithm. They are chosen to ensure an
   appropriate level of security. 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
   a direct result of the choices of n and w. Appendix A describes how
   p and ls depend on the other parameters. A table illustrating
   various combinations of n, w, p, and ls is provided in Table 4.

3.2. Hashing Functions

The LDWM algorithm requires a robust one-way function to underpin the
signature generation and verification. Let this be defined as a hash
function H that has inputs of any integral byte length and has n-byte
outputs.

In addition, let F be defined as a one-way function that has m-byte
inputs and m-byte outputs.

3.3. Signature Methods

To fully describe a LDWM signature method, the parameters m, n, and
w, as well as the functions H and F MUST be specified. This section
defines several LDWM signature systems, each of which is identified
by a name. Values for p and ls are provided as a convenience.

Name	H	F	m	n	w	p	ls
LDWM_SHA256_M20_W1	SHA256	SHA256-20	20	32	1	265	7
LDWM_SHA256_M20_W2	SHA256	SHA256-20	20	32	2	133	6
LDWM_SHA256_M20_W4	SHA256	SHA256-20	20	32	4	67	4
LDWM_SHA256_M20_W8	SHA256	SHA256-20	20	32	8	34	0

Table 1

Here SHA256 denotes the NIST standard hash function. SHA256-20
denotes that hash function with its final output truncated to 20
bytes.

3.4. Private Key

The LDWM private key must be 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

      for i=0,i<p,i++
        set x[i] to a uniformly random value
      endfor
      return x

   Note that one possible implementation may consist of a single random
   input to a suitable key derivation function.

3.5. Public Key

   The LDWM public key is generated from the private key through a
   series of hashing operations using the functions F and H. Its value
   is the hash (using H) of the concatenation of the elements of an
   array y. The content of y is generated by iteratively hashing (using
   F) each element of array x, (2^w - 1) times. The following algorithm
   shows pseudocode for generating the public key.

   Algorithm 1: Generating a Public Key From a Private Key

      e = 2^w - 1
      for i=0,i<p,i++
        y[i] = F^e(x[i])
      endfor
      return H(y[0] || y[1] || ... || y[p-1])

3.6. Checksum

   A checksum is used to prevent the manipulation of an existing
   signature in an attempt to produce a new signature for a different
   message outside of the normal signing process. The security property
   is detailed in Section 8.

   The checksum value is calculated using a non-negative integer, sum,
   whose width is sized 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 F as defined in the signing algorithm of
   Section 3.7 and the total actual number. In the worst case (i.e. the
   actual number of times F 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 }, let sum be a 16-bit non-negative integer for
   all combinations of n and w. The checksum function C is defined as
   follows where S is a byte string provided to the function as an
   argument.

Algorithm 2: Checksum Calculation

```
sum = 0
for i=0,i<u,i++
  sum = sum + (2^w - 1) - coef(S, i, w)
endfor
return (sum << ls)
```

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

> Implementation Note: Based on the previous fact, the implementation may therefore choose to optimize the width of sum to (v * w) bits and set ls to 0. The rationale for this is given that (2^w - 1) * ceil(8*n/w) is the maximum value of sum and the value of (2^w - 1) is represented by w bits, the result of repeatedly adding two w-bit numbers a total of u = ceil(8*n/w) times requires at most (floor(lg(u)) + w) bits. Dividing by w and taking the next largest integer gives the total required number of w-bit fields and gives (ceil(floor(lg(u)) / w) + 1), or v. Thus sum requires a minimum width of (v * w) bits and no left-shift operation is performed.

3.7. Signature Generation

The LDWM signature is generated by using the values of the w-bit fields that compose the hash of the message and its checksum to determine the number of times to apply the function F to the elements of the private key. This signature is provided by the signer to the verifier along with the message and the public key.

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

```
V = ( H(message) || C(H(message)) )
for i=0,i<p,i++
  a = coef(V, i, w)
  y[i] = F^a(x[i])
endfor
return ( 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 3.5.

3.8. Signature Verification

In order to verify a message with its signature (denoted as y', an
array of m-byte strings), the receiver must "complete" the series of
applications of F using the value of the message hash and its
checksum. This should result in a computation of the public key that
matches the public key provided.

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

```
V = ( H(message) || C(H(message)) )
for i=0,i<p,i++
  a = (2^w - 1) - coef(V, i, w)
  z[i] = F^a(y'[i])
endfor
if public key is equal to H(z[0] || z[1] || ... || z[p-1])
  return 1 (message signature is valid)
else
  return 0 (message signature is invalid)
```

3.9. Notes

A future version of this specification may define a method for
computing the signature of a very short message in which the hash is
not applied to the message during the signature computation. That
would allow the signatures to have reduced size.

3.10. Formats

The signature format is defined using XDR [RFC4506] as follows:

```
/*
 * ldwm_algorithm_type identifies a particular signature algorithm
 */
enum ldwm_algorithm_type {
  ldwm_reserved = 0,
  ldwm_sha256_m20_w1  = 1,
  ldwm_sha256_m20_w2  = 2,
  ldwm_sha256_m20_w4  = 3,
  ldwm_sha256_m20_w8  = 4
};

/*
 * basic data types; b16 is a string of 16 bytes, and so on
 */
typedef opaque b16[16];
typedef opaque b20[20];
typedef opaque b24[24];
typedef opaque b28[28];
typedef opaque b32[32];

/*
 * arrays
 */
typedef b20 b20_array_265[265];
typedef b20 b20_array_133[133];
typedef b20 b20_array_67[67];
typedef b20 b20_array_34[34];

union ldwm_signature switch (ldwm_algorithm_type type) {
 case ldwm_sha256_m20_w1:
   b20_array_265 y265;
 case ldwm_sha256_m20_w2:
   b20_array_133 y133;
 case ldwm_sha256_m20_w4:
   b20_array_67 y67;
 case ldwm_sha256_m20_w8:
   b20_array_34 y34;
 default:
   void;    /* error condition */
};
```

Though the data formats are formally defined by XDR, we describe the
format as well as a convenience to the reader. An example of the
format of an ldwm_signature is illustrated below, for

ldwm_sha256_m20_w1, which has 265 elements, each of which is a 20-
octet string. An ldwm_signature always consists of a 32-bit unsigned
integer that indicates the ldwm_algorithm_type followed by an array
of equal-length octet strings. The number of octets in each octet
string, and the number of elements in the array, are determined by
the ldwm_algorithm_type field. A receiver MUST check the
ldwm_algorithm_type field, and a verification operation on a
signature with an unknown ldwm_algorithm_type MUST return FAIL.

```
            +--------------------------------+
            |        ldwm_algorithm_type     |
            +--------------------------------+
            |                                |
            |              y265[0]           |
            |                                |
            |                                |
            +--------------------------------+
            |                                |
            |              y265[1]           |
            |                                |
            |                                |
            |                                |
            +--------------------------------+
            |                                |
            ~              ....              ~
            |                                |
            +--------------------------------+
            |                                |
            |             y265[264]          |
            |                                |
            |                                |
            |                                |
            +--------------------------------+
```
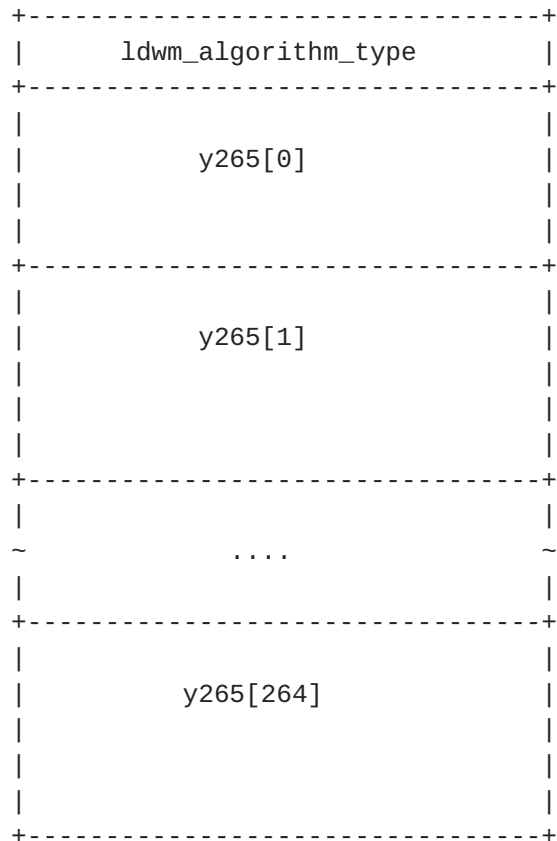
[4](#). Merkle Tree Signatures

   Merkle Tree Signatures (MTS) are a method for signing a large but
   fixed number of messages. An MTS system uses two cryptographic
   components: a one-time signature method and a collision-resistant
   hash function. Each MTS public/private key pair is associated with a
   k-way tree, each node of which contains an n-byte value. Each leaf
   of the tree contains the value of the public key of an LDWM public/
   private key pair. The value contained by the root of the tree is the
   MTS public key. Each interior node is computed by applying the hash
   function to the concatenation of the values of its children nodes.

   An MTS system has the following parameters:

      k : the number of children nodes of an interior node,

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

      n : the number of bytes associated with each node.

   There are k^h leaves in the tree.

[4.1](#). Private Key

   An MTS private key consists of k^h LDWM private keys and the leaf
   number of the next LDWM private key that has not yet been used. The
   leaf number is initialized to zero when the MTS private key is
   created.

   An MTS private key MAY be generated pseudorandomly from a secret
   value, in which case the secret value MUST be n bytes long, be
   uniformly random, and MUST NOT be used for any other purpose than the
   generation of the MTS 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.

[4.2](#). MTS Public Key

   An MTS public key is defined as follows, where we denote the public
   key associated with the ith LDWM private key as ldwm_public_key(i).

   The MTS public key can be computed using the following algorithm or
   any equivalent method. The algorithm uses a stack of hashes and a
   separate stack of integers which keeps track of the level of the
   tree.

```
   for i = 0 to num_ldwm_keys by steps of k
     level = 0
     for j = 0 to k-1
       push ldwm_public_key(i+j) on the data stack
       push level on the integer stack
     endfor
     while the height of the stack is at least k
       if the top k elements on the integer stack are equal
         pop the top k elements of the data stack
         pop the top k elements of the integer stack,
           and set the variable "level" to be their value
         hash the top k elements of the data stack
         push the hash result on the data stack
         push (level+1) on to the integer stack
       endif
     endwhile
   endfor
```

   Note that the stack never gets bigger than the logarithm of the
   number of LDWM public keys, so for typical parameters it will have
   something like 20 32-byte elements.

4.3. MTS Signature

   An MTS signature consists of

      an LDWM signature,

      a node number that identifies the leaf node associated with the
      signature, and

      an array of values that is associated with the path through the
      tree from the leaf associated with the LDWM signature to the root.

   The array of values contains 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 branching number k and
   height h will have (k-1)*h values. The first (k-1) values are the
   siblings of the leaf, the next (k-1) values are the siblings of the
   parent of the leaf, and so on.

4.3.1. MTS Signature Generation

   To compute the MTS signature of a message with an MTS private key,
   the signer first computes the LDWM signature of the message using the
   leaf number of the next unused LDWM private key. Before releasing
   the signature, the leaf number in the MTS private key MUST be
   incremented to prevent the LDWM private key from being used again.

The node number in the signature is set to the leaf number of the MTS
private key that was used in the signature.

The array of node values MAY be computed in any way. There are many
potential time/storage tradeoffs. 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 public
key verification algorithm.

4.4. MTS Signature Verification

An MTS signature is verified by first using the LDWM signature
verification algorithm to compute the LDWM public key from the LDWM
signature and the message. The value of the leaf associated with the
LDWM signature is assigned to the public key. Then the root of the
tree is computed from the leaf value and the node array as described
below. If the root value matches the public key, then the signature
is valid; otherwise, the signature fails.

An efficient way to compute the root from the leaf and the node array
is as follows, where n is initially the node number:

```
  v = leaf
  step = 0
  for i=0 to h-1 by steps of 1
    position = n % k
    hash_init()
    for j=0 to position-1 by steps of 1
      hash_update(node[step + j])
    endfor
    hash_update(v)
    for j=position to (k-1) by steps of 1
      hash_update(node[step + j])
    endfor
    v = hash_final()
    n = floor(n/k)
    step = step + (k-1)
  endfor
```

This algorithm uses 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]).

This algorithm works because the leaves of the MTS tree are numbered starting at zero. Therefore leaf n is in the position (n % k) in the highest level of the tree.

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 MTS public key.

A full test example that combines the LDWM OTS and MTS algorithms is given in Appendix B.

4.5. MTS Formats

   MTS signatures and public keys are defined using XDR syntax as
   follows:

```
enum mts_algorithm_type {
  mts_reserved      = 0,
  mts_sha256_k2_h20 = 1,
  mts_sha256_k4_h10 = 2,
  mts_sha256_k8_h7  = 3,
  mts_sha256_k16_h5 = 4
};

union mts_path switch (mts_algorithm_type type) {
 case mts_sha256_k2_h20:
   b32 t20[20];
 case mts_sha256_k4_h10:
   b32 t30[30];
 case mts_sha256_k8_h7:
   b32 t49[49];
 case mts_sha256_k16_h5:
   b32 t75[75];
 default:
   void;      /* error condition */
};

struct mts_signature_t {
  ldwm_signature ldwm_sig;
  unsigned int signature_leaf_number;
  mts_path nodes;
};

union mts_public_key switch (mts_algorithm_type type) {
 case mts_sha256_k2_h20:
 case mts_sha256_k4_h10:
 case mts_sha256_k8_h7:
 case mts_sha256_k16_h5:
   b32 z;
 default:
   void;      /* error condition */
};
```

5. Rationale

   The goal of this note is to describe the LDWM and MTS 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.

   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.

6. History

   This is the initial version of this draft.

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

7. IANA Considerations

   The Internet Assigned Numbers Authority (IANA) is requested to create
   two registries: one for LDWM signatures, as defined in Section 3, and
   one for Merkle Tree 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 "MTS_SHA256_K16_H5", that starts with the
      strings "MTS" and "LDWM" for the MTS and LDWM registries,
      respectively,

      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 positive 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 32,768 (binary 1000000000000000) and 65,535
   (binary 1111111111111111) 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 LDWM registry is as follows.

```
+--------------------+-----------+-------------------+
| Name               | Reference | Numeric Identifier |
+--------------------+-----------+-------------------+
| LDWM_SHA256_M20_W1 | Section 3 |         1         |
|                    |           |                   |
| LDWM_SHA256_M20_W2 | Section 3 |         2         |
|                    |           |                   |
| LDWM_SHA256_M20_W4 | Section 3 |         3         |
|                    |           |                   |
| LDWM_SHA256_M20_W8 | Section 3 |         4         |
+--------------------+-----------+-------------------+
```

Table 2

The MTS registry is as follows.

```
+------------------+-----------+-------------------+
| Name             | Reference | Numeric Identifier |
+------------------+-----------+-------------------+
| MTS_SHA256_K2_H20 | Section 4 |         1         |
|                  |           |                   |
| MTS_SHA256_K4_H10 | Section 4 |         2         |
|                  |           |                   |
| MTS_SHA256_K8_H7  | Section 4 |         3         |
|                  |           |                   |
| MTS_SHA256_K16_H5 | Section 4 |         4         |
+------------------+-----------+-------------------+
```

Table 3

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

8. 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").

   LDWM signatures rely on the fact that, given an m-byte string y, it
   is prohibitively expensive to compute a value x such that $F^i(x) = y$
   for any i. Informally, F is said to be a "one-way" function.

8.1. Security of LDWM Checksum

   To show the security of LDWM checksum, we consider the signature y of
   a message with a private key x and let h = H(message) and
   c = C(H(message)) (see Section 3.7). To attempt a forgery, an
   attacker can 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 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 this action, the
   checksum will decrease, and 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 $F^b(x[k]) = y[k]$.

8.2. Security Conjectures

   LDWM and MTS signatures have a minimum of security conjectures. In
   particular, their security does not rely on the computational
   difficulty of factoring composites with large prime factors (as does

RSA) or the difficulty of computing the discrete logarithm in a
finite field (as does DSA) or an elliptic curve group (as does
ECDSA). All of these signature schemes rely on the security of the
hash function that they use, but with LDWM and MTS, the security of
the hash function is sufficient.

8.3. Post-Quantum Security

A post-quantum cryptosystem is a system that is secure against
quantum computers that have more than a trivial number of quantum
bits. It is open to conjecture whether or not it is feasible to
build such a machine.

The LDWM and Merkle signature systems are post-quantum secure if they
are used with an appropriate underlying hash function, in which the
size of m and n are double what they would be otherwise, in order to
protect against quantum square root attacks due to Grover's
algorithm. In contrast, the signature systems in wide use (RSA, DSA,
and ECDSA) are not post-quantum secure.

9. Acknowledgements

   Thanks are due to Chirag Shroff for constructive feedback.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
           Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an
           IANA Considerations Section in RFCs", BCP 26, RFC 2434,
           October 1998.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard",
           STD 67, RFC 4506, May 2006.

10.2. 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.

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

Appendix A. LDWM Parameter Options

   A table illustrating various combinations of n, w, p, and ls is
   provided in Table 4.

   The parameters p and ls are computed as follows:

     u = ceil(8*n/w)
     v = ceil(floor(lg(u)) / w) + 1
     p = u + v
     ls = (number of bits in sum) - (v * w)

   Here u and v represent the number of w-bit fields required to contain
   the hash of the message message and the checksum byte strings,
   respectively. For a further explanation of the values of v and sum,
   see Section 3.6.

Hash Length in Bytes (n)	Winternitz Parameter (w)	w-bit Elements in Checksum	Left Shift (ls)	Number of Elements (p)
20	1	8	8	168
20	2	4	8	84
20	4	3	4	43
20	8	2	0	22
32	1	9	7	265
32	2	5	6	133
32	4	3	4	67
32	8	2	0	34
48	1	9	7	393
48	2	5	6	197
48	4	3	4	99
48	8	2	0	50
64	1	10	6	522
64	2	5	6	261
64	4	3	4	131
64	8	2	0	66

Table 4

Appendix B. Example Data for Testing

   As with all cryptosystems, implementations of LDWM signatures and
   Merkle signatures need to be tested before they are used. This
   section contains sample data generated from the signing and
   verification operations of software that implements the algorithms
   described in this document.

B.1. Parameters

   The example contained in this section demonstrates the calculations
   of LDWM_SHA256_M20_W4 using a Merkle Tree Signature of degree 4 and
   height 2. This corresponds to the following parameter values:

```
           +----+----+---+----+---+---+---+
           |  m |  n | w |  p | l | k | h |
           +----+----+---+----+---+---+---+
           | 20 | 32 | 4 | 67 | 4 | 4 | 2 |
           +----+----+---+----+---+---+---+
```

                             Table 5

   The non-standard size of the Merkle tree (h = 2) has been selected
   specifically for this example to reduce the amount of data presented.

B.2. Key Generation

   The LDWM algorithm does not define a required method of key
   generation. This is left to the implementer. The selected method,
   however, must satisfy the requirement that the public/private key
   pairs of the one-time signatures are unique. In addition, all LDWM
   key pairs must be generated in advance in order to calculate the
   value of the Merkle public key.

   For the test data presented here, a summary of the key generation
   method is as follows:

   1. MTS Private Key - Set mts_private_key to a pseudorandomly
       generated n-byte value.

   2. OTS Private Keys - Use the mts_private_key as a key derivation
       key input to some key derivation function, thereby producing n^k
       derived keys. Then use each derived key as an input to the same
       function again to further derive p elements of n-bytes each.
       This accomplishes the result of Algorithm 0 of Section 3.4 for
       each leaf of the Merkle tree.

3. OTS Public Keys - For each OTS private key, calculate the
    corresponding OTS public key as in Algorithm 1 of Section 3.5.

4. MTS Public Key - Each OTS public key is the value of a leaf on
    the Merkle tree. Calculate the MTS public key using the
    pseudocode algorithm of Section 4.2.

The above steps result in the following data values associated with
the first leaf of the Merkle tree, leaf 0.

```
+------------------------------------------------------------------+
|                        MTS Private Key                           |
+------------------------------------------------------------------+
|                 0x0f677ff1b4cbf10baec89959f051f203               |
|                   3371492da02f62dd61d6fbd1cee1bd14               |
+------------------------------------------------------------------+
```

                              Table 6

```
+----------------+-------------------------------------------------+
|  Key Element   |         OTS Private Key 0 Element (x[i])         |
|   Index (i)    |                                                 |
+----------------+-------------------------------------------------+
|       0        |          0xbfb757383fb08d324629115a84daf00b      |
|                |            188d5695303c83c184e1ec7a501c431f      |
|                |                                                 |
|       1        |          0x7ce628fb82003a2829aab708432787d0      |
|                |            fc735a29d671c7d790068b453dc8c913      |
|                |                                                 |
|       2        |          0x8174929461329d15068a4645a34412bd      |
|                |            446d4c9e757463a7d5164efd50e05c93      |
|                |                                                 |
|       3        |          0xf283f3480df668de4daa74bb0e4c5531      |
|                |            5bc00f7d008bb6311e59a5bbca910fd7      |
|                |                                                 |
|       4        |          0xe62708eaf9c13801622563780302a068      |
|                |            0ba9d39c078daa5ebc3160e1d80a1ea7      |
|                |                                                 |
|       5        |          0x1f002efad2bfb4275e376af7138129e3      |
|                |            3e88cf7512ec1dcdc7df8d5270bc0fd7      |
|                |                                                 |
|       6        |          0x8ed5a703e9200658d18bc4c05dd0ca8a      |
|                |            356448a26f3f4fe4e0418b52bd6750a2      |
|                |                                                 |
|       7        |          0xc74e56d61450c5387e86ddad5a8121c8      |
|                |            8b1bc463e64f248a1f1d91d950957726      |
|                |                                                 |
```

8		0x629f18b6a2a4ea65fff4cf758b57333f e1d34af05b1cd7763696899c9869595f
9		0x1741c31fdbb4864712f6b17fadc05d45 926c831c7a755b7d7af57ac316ba6c2a
10		0xe59a7b81490c5d1333a9cdd48b9cb364 56821517a3a13cb7a8ed381d4d5f3545
11		0x3ba97fe8b2967dd74c8b10f31fc5f527 a23b89c1266202a4d7c281e1f41fa020
12		0xa262a9287cc979aaa59225d75df51b82 57b92e780d1ab14c4ac3ecdac58f1280
13		0x9dfe0af1a3d9064338d96cb8eae88baa 6a69265538873b4c17265fa9d573bcff
14		0xde9c5c6a5c6a274eabe90ed2a8e6148c 720196d237a839aaf5868af8da4d0829
15		0x5de81ec17090a82cb722f616362d3808 30f04841191e44f1f81b9880164b14cd
16		0xc0d047000604105bad657d9fa2f9ef10 1cfd9490f4668b700d738f2fa9e1d11a
17		0xf45297ef310941e1e855f97968129bb1 73379193919f7b0fee9c037ae507c2d2
18		0x46ef43a877f023e5e66bbcd4f06b839f 3bfb2b64de25cd67d1946b0711989129
19		0x46e2a599861bd9e8722ad1b55b8f0139 305fcf8b6077d545d4488c4bcb652f29
20		0xe1ad4d2d296971e4b0b7a57de305779e 82319587b58d3ef4daeb08f630bd5684
21		0x7a07fa7aed97cb54ae420a0e6a58a153 38110f7743cab8353371f8ca710a4409
22		0x40601f6c4b35362dd4948d5687b5cb6b 5ec8b2ec59c2f06fd50f8919ebeaae92
23		0xa061b0ba9f493c4991be5cd3a9d15360 a9eb94f6f7adc28dddf174074f3df3c4

24	0xcf1546a814ff16099cebf1fe0db1ace5
	1c272fda9846fbb535815924b0077fa4
25	0xcbb06f13155ce4e56c85a32661c90142
	8b630a4c37ea5c7062156f07f6b3efff
26	0x1181ee7fc03342415094e36191eb450a
	11cdea9c6f6cdc34de79cee0ba5bf230
27	0xe9f1d429b343bb897881d2a19ef363cd
	1ab4117cbaad54dc292b74b8af9f5cf2
28	0x87f34b2551ef542f579fa65535c5036f
	80eb83be4c898266ffc531da2e1a9122
29	0x9b4b467852fe33a03a872572707342fd
	ddeae64841225186babf353fa2a0cd09
30	0x19d58cd240ab5c80be6ddf5f60d18159
	2dca2be40118c1fdd46e0f14dffbcc7d
31	0x5c9ad386547ba82939e49c9c74a8eccf
	1cea60aa327b5d2d0a66b1ca48912d6d
32	0xf49083e502400ffae9273c6de92a301e
	7bda1537cab085e5adfa9eb746e8eca9
33	0x4074e1812d69543ce3c1ce706f6e0b45
	f5f26f4ef39b34caa709335fd71e8fc0
34	0x1256612b0ca8398e97b247ae564b74b1
	3839b3b1cf0a0dd8ba629a2c58355f84
35	0xbab3989f00fd2c327bbfb35a218cc3ce
	49d6b34cbf8b6e8919e90c4eff400ca9
36	0x96b52a5d395a5615b73dae65586ac5c8
	7f9dd3b9b3f82dbf509b5881f0643fa8
37	0x5d05ca4c644e1c41ccdaedbd2415d4f0
	9b4a1b940b51fe823dff7617b8ee8304
38	0xd96aab95ef6248e235d91d0f23b64727
	a6675adfc64efea72f6f8b4a47996c0d
39	0xfd9c384d52d3ac27c4f4898fcc15e83a
	c182f97ea63f7d489283e2cc7e6ed180

	40		0xc86eaed6a9e3fbe5b262c1fa1f099f7c
			35ece71d9e467fab7a371dbcf400b544
	41		0xf462b3719a2ed8778155638ff814dbf4
			2b107bb5246ee3dd82abf97787e6a69e
	42		0x014670912e3eb74936ebb64168b447e4
			2522b57c2540ac4b49b9ae356c01eca6
	43		0x2b411096e0ca16587830d3acd673e858
			863fedc4cea046587cba0556d2bf9884
	44		0xa73917c74730582e8e1815b8a07b1896
			2ac05e500e045676be3f1495fcfa18ca
	45		0xa4ab61e6962fe39a255dbf8a46d25110
			0d127fab08db59512653607bda24302c
	46		0x9b910ca516413f376b9eba4b0d571b22
			253c2a9646131ac9a2af5f615f7322b8
	47		0xfc1b4ce627c77ad35a21ea9ded2cce91
			b3758a758224e35cf2918153a513d64c
	48		0xc1902d8e8c02d9442581d7e053a2798a
			a84d77a74b6e7f2cc5096d50646c890f
	49		0xb3f47e2e8e2dcdd890ea00934b9d8234
			830dbc4a30ac996b144f12b3e463c77f
	50		0x8188d1ecfc6ae6118911f2b9b3a6c7a1
			e5f909aa8b5c0aab8c69f1a7d436c307
	51		0xca42d985974c7b870bc76494604eff49
			2676c942c6cb7c75d4938805885dd054
	52		0xbe58851ebe566057e1ee16b8c604a473
			4c373af622660b2a82357ac6effb4566
	53		0xc22d493f7a5642fceba2404dbefa8f95
			6323fac87fac425f6de8d23c9e8b20ca
	54		0x1a76c1ffa467906173fd0245b0cd6639
			e6013ca79c4ed92426ee69ff5beeac0b
	55		0xbc6c0cb7808f379af1b7b7327436ad65
			c05458f2d0a6923c333e5129c4c99671

56	0xfbb04488c3c088dc5e63d13e6a701036
	6109ca4c5f4b0a8d37780187e2e9930e
57	0xaec10811569d4d72e3a1baf71a886b75
	eba6dc07ed027af0b2beffa71f9b43c8
58	0xf5529be3b7a19212e8baa970d2420bf4
	123f678267f96c1c3ef26ab610cb0061
59	0x172ba1ba0b701eeafe00692d1eb90181
	8ccaefaeb8f799395da81711766d1f43
60	0xfe1f8c15825208f3a21346b894b3d94e
	4f3aa29cbc194a7b2c8a810c4c509042
61	0x2e81c66cc914ea1b0fa5942fe9780d54
	8c0b330e3bf73f0cb0bda4bc9c9e6ff4
62	0xfc3453aec5cc19a6a4bda4bc25931604
	704bf4386cd65780c6e73214c1da85ba
63	0x4e8000c587dc917888e7e3d817672c0a
	ef812788cc8579afa7e9b2e566309003
64	0xba667ca0e44a8601a0fde825d4d2cf1b
	b9cf467041e04af84c9d0cd9fd8dc784
65	0x4965db75f81c8a596680753ce70a94c6
	156253bb426947de1d7662dd7e05e9a8
66	0x2c23cc3e5ca37dec279c506101a3d8d9
	f1e4f99b2a33741b59f8bddba7455419

Table 7

Using the value of the OTS private key above, the corresponding
public key is given below. Intermediate values of the SHA-256-20
function F^(2^w - 1)(x[i]) are provided in Table 13.

OTS Public Key 0
0x2db55a72075fcfab5aedbef77bf6b371
dfb489d6e61ad2884a248345e6910618

Table 8

Following the creation of all OTS public/private key pairs, the OTS
public keys in Table 14 are used to determine the MTS public key
below. Intermediate values of the interior nodes of the Merkle tree
are provided in Table 15.

```
+---------------------------------------------------------------------+
|                         MTS Public Key                              |
+---------------------------------------------------------------------+
|                 0x6610803d9a3546fb0a7895f6a4a0cfed                  |
|                 3a07d45e51d096e204b018e677453235                    |
+---------------------------------------------------------------------+
```

                                Table 9

B.3. Signature Generation

In order to test signature generation, a text file containing the
content "Hello world!\n", where '\n' represents the ASCII line feed
character, was created and signed. A raw hex dump of the file
contents is shown in the table below.

```
+-------------------------------+-----------------------------------+
|    Hexadecimal Byte Values    |        ASCII Representation        |
|                               |        ('.' is substituted for     |
|                               |        non-printing characters)    |
+-------------------------------+-----------------------------------+
| 0x48 0x65 0x6c 0x6c 0x6f 0x20 |          Hello world!.             |
| 0x77 0x6f 0x72 0x6c 0x64 0x21 |                                    |
|              0x0a             |                                    |
+-------------------------------+-----------------------------------+
```

                                Table 10

The SHA256 hash of the text file is provided below.

```
+---------------------------------------------------------------------+
|          SHA256 Hash of Signed File (H("Hello world!\n"))           |
+---------------------------------------------------------------------+
|                 0x0ba904eae8773b70c75333db4de2f3ac4                 |
|                 5a8ad4ddba1b242f0b3cfc199391dd81c                   |
+---------------------------------------------------------------------+
```

                                Table 11

This value was subsequently used in Algorithm 3 of Section 3.7 to
create the one-time signature of the message. Algorithm 2 of
Section 3.6 was applied to calculate a checksum of 0x1cc. The
resulting signature is shown in the following table.

OTS Element Index (i)	Function Iteration Count (a = coef(H(msg) || C(H(msg)), i, w))	OTS Element (y[i] = F^a(x[i]))
0	0	0xbfb757383fb08d324629115a84daf00b188d5695
1	11	0x4af079e885ddfd3245f29778d265e868a3bfeaa4
2	10	0xfbad1928bfc57b22bcd949192452293d07d6b9ad
3	9	0xb98063e184b4cb949a51e1bb76d99d4249c0b448
4	0	0xe62708eaf9c13801622563780302a0680ba9d39c
5	4	0x39343cba3ffa6d75074ce89831b3f3436108318c
6	14	0xfe08aa73607aec5664188a9dacdc34a295588c9a
7	10	0xd3346382119552d1ceb92a78597a00c956372bf0
8	14	0xf1dd245ec587c0a7a1b754cc327b27c839a6e46a
9	8	0xa5f158adc1decaf0c1edc1a3a5d8958d726627b5
10	7	0x06d2990f62f22f0c943a418473678e3ffdbff482
11	7	0xf3390b8d6e5229ae9c5d4c3f45e10455d8241a49
12	3	0x22dd5f9d3c89180caa0f695203d8cf90f3c359be
13	11	0x67999c4043f95de5f07d82b741347a3eb6ac0c25
14	7	0xc4ffe472d48adeb37c7360da70711462013b7a4e
15	0	0x5de81ec17090a82cb722f616362d380830f04841
16	12	0x2f892c824af65cc749f912a36dfa8ade2e4c3fd1
17	7	0xb644393e8030924403b594fb5cacd8b2d28862e2
18	5	0x31b8d2908911dbbf5ba1f479a854808945d9e948
19	3	0xa9a02269d24eb8fed6fb86101cbd0d8977219fb1

20	3	0xe4aae6e6a9fe1b0d5099513f170c111dee95714d
21	3	0xd79c16e7f2d4dd790e28bab0d562298c864e31e9
22	13	0xc29678f0bb4744597e04156f532646c98a0b42e8
23	11	0x57b31d75743ff0f9bcf2db39d9b6224110b8d27b
24	4	0x0a336d93aac081a2d849c612368b8cbb2fa9563a
25	13	0x917be0c94770a7bb12713a4bae801fb3c1c43002
26	14	0x91586feaadcf691b6cb07c16c8a2ed0884666e84
27	2	0xdd4e4b720fb2517c4bc6f91ccb8725118e5770c6
28	15	0x491f6ec665f54c4b3cffaa02ec594d31e6e26c0e
29	3	0x4f5a082c9d9c9714701de0bf426e9f893484618c
30	10	0x11f7017313f0c9549c5d415a8abc25243028514d
31	12	0x6839a994fccb9cb76241d809146906a3d13f89f1
32	4	0x71cd1d9163d7cd563936837c61d97bb1a5337cc0
33	5	0x77c9034ffc0f9219841aa8e1edbfb62017ef9fd1
34	10	0xad9f6034017d35c338ac35778dd6c4c1abe4472a
35	8	0x4a1c396b22e4f5cc2428045b36d13737c4007515
36	10	0x98cb57b779c5fd3f361cd5debc243303ae5baefd
37	13	0x29857298f274d6bf595eadc89e5464ccf9608a6c
38	4	0x95e35a26815a3ae9ad84a24464b174a29364da18
39	13	0x4afeb3b95b5b333759c0acdd96ce3f26314bb22b
40	13	0x325a37ee5e349b22b13b54b24be5145344e7b8f3
41	11	0x4f772c93f56fd6958ce135f02847996c67e1f2ef
42	10	0xd4f6d91c577594060be328b013c9e9b0e8a2e5d8
43	1	0x717e1a81c325cdccacb6e9fd9e92dd3e1bb84ae8

44	11	0x1dd363724ec66c090a1228dfa1cd3d9cc806f346
45	2	0x64b4110476dd0beea78714c5ab71278818792cfa
46	4	0xe22290e740056a144af50f0b10962b5bcc18fc82
47	2	0x34fd87046a183f4732a52bb7805ce207eebdafc5
48	15	0xbd2fdc5e4e8d0ed7c48c1bad9c2f7793fc2c9303
49	0	0xb3f47e2e8e2dcdd890ea00934b9d8234830dbc4a
50	11	0xcd29719c56cdb507030e6132132179e5807e1d3b
51	3	0xf9edb9b301916217de0d746a0542316bebe9e806
52	12	0x7a3801cbfe0cafed863d81210c1ec721eede49e5
53	15	0x5caba3ec960efa210f5f3e1c22c567ca475ef3ec
54	12	0xf911b5d148e1b03fe6983c53411f76ea78772379
55	1	0x06da2baa75c6ef752bf59f3812fa042ff8181209
56	9	0x2b29f5aa2f34af51a78a5fac586004f749c6e6dc
57	9	0x55e033ababac0845cc9142e24f9ef0a641c51cbe
58	3	0xb62d207bb700071fba8a68312ca204ce4d994c33
59	9	0x551d5c00fad905bdb99c4f70ec7590a10d3ff8ca
60	1	0x0d03b1845b5f8838d735142f185f9cf8f8d2db6c
61	13	0x3b5d9e49e7ede41cd9aa5a09f72a0384fd4ff511
62	13	0xa766b0278d14a9b7d32bf0307c0737a8ecf82ab1
63	8	0xca85296f354e6e3d2a96ab497c01e5ccd4530cf1
64	1	0x7bb29db7dd8aaaf1cd11487cea0d13730edb1df3
65	12	0x547ef341b3cf3208753bb1b62d85a4e3fc2cffe0
66	12	0xb890e1a99da4b2e0a9dde42f82f92d0946327cee

Table 12

Finally, based on the fact that the message is the first to be signed
by the Merkle tree (i.e. using leaf node 0), the values of the leaf
and interior nodes that compose the authentication path from leaf to
root are determined as described in Section 4.3. These values are
marked with an asterisk ('*') in Table 14 and Table 15.

B.4. Signature Verification

The signature verification step was provided the following items:

1. OTS = (y[0] || y[1] || ... || y[p-1]) - from Table 12.

2. Authentication Path = concatenation of (k-1)*h Merkle tree node
    values - from Table 14 and Table 15.

3. Message Number = leaf number of Merkle tree.

4. Merkle Public Key = root of Merkle tree - from Table 9.

Using Algorithm 4 of Section 3.8 as a start, the potential OTS public
key was calculated from the value of the OTS. Since the actual OTS
public key was not provided to the verifier, the calculated key was
checked for validity using the pseudocode algorithm of Section 4.4
and the provided values of the Authentication Path and Message
Number. Since the message was valid, the calculated value of the
root matched the Merkle public key. Otherwise, verification would
have failed.

B.5. Intermediate Calculation Values

Key Element Index (i)	SHA256-20 Result for w = 4 ($F^{15}(x[i])$)
0	0x6eff4b0c224874ecc4e4f4500da53dbe2a030e45
1	0x58ac2c6c451c7779d67efefdb12e5c3d85475a94
2	0xb1f3e42e29c710d69268eed1bbdb7f5a500b7937
3	0x51d28e573aac2b84d659abb961c32c465e911b55
4	0xa0ed62bccac5888f5000ca6a01e5ffefd442a1c6
5	0x44da9e145666322422c1e2b5e21627e05aeb4367
6	0x04e7ff9213c2655f28364f659c35d3086d7414e1

```
|               7           | 0x414cdb3215408b9722a02577eeb71f9e016e4251 |
|                           |                                            |
|               8           | 0xa3ab06b90a2b20f631175daa9454365a4f408e9e |
|                           |                                            |
|               9           | 0xe38acfd3c0a03faa82a0f4aeac1a7c04983fad25 |
|                           |                                            |
|              10           | 0xd95a289094ccce8ad9ff1d5f9e38297f9bb306ff |
|                           |                                            |
|              11           | 0x593d148b22e33c32f18b66340bdaffceb3ad1a55 |
|                           |                                            |
|              12           | 0x16b53fbea11dc7ab70c8336ec3c23881ae5d51bf |
|                           |                                            |
|              13           | 0xa639ca0cf871188cadd0020832c4f06e6ebd5f98 |
|                           |                                            |
|              14           | 0xe3ab3e0c5ad79d6c8c2a7e9a79856d4380941fe0 |
|                           |                                            |
|              15           | 0x8368c2933dabcde69c373867a9bf2dc78df97bea |
|                           |                                            |
|              16           | 0xe3609fca11545da156a7779ae565b1e3c87902c0 |
|                           |                                            |
|              17           | 0xab029e62c7011772dc0589d79fad01aacf8d2177 |
|                           |                                            |
|              18           | 0xa8310f1c27c1aa481192de07d4397b8c4716e25f |
|                           |                                            |
|              19           | 0xdbdbb14dbd9a5f03c1849af24b69b9e3f80faca2 |
|                           |                                            |
|              20           | 0x1a17399d555dec07d3d4f6d54b2b87d2bcaa398b |
|                           |                                            |
|              21           | 0xf81c66cc522bfb203232e44d0003ed65d2462867 |
|                           |                                            |
|              22           | 0x202a625b8c5f22de6ea081af6da077cf5c63202f |
|                           |                                            |
|              23           | 0x2e080f3591f5ff3d5de39c2698846cc107a09816 |
|                           |                                            |
|              24           | 0xa1d9c78c22f9810e3b7db2d59ad9f5fdd259f4d4 |
|                           |                                            |
|              25           | 0x658eeb85ebe0f4542c4d32dced2d7226929266b2 |
|                           |                                            |
|              26           | 0x67fae1a784f919577afc091504d82d31b4ba9fc7 |
|                           |                                            |
|              27           | 0xfc39fb43677fb2d433a6292f19c6e7320279655a |
|                           |                                            |
|              28           | 0x491f6ec665f54c4b3cffaa02ec594d31e6e26c0e |
|                           |                                            |
|              29           | 0x17cec813a5781409b11d2e4a85f62301c2fd8873 |
|                           |                                            |
|              30           | 0xc578eb105454d900c053eb55833db607aa5757e0 |
|                           |                                            |
```

	31		0xaed094323290a41fd4b546919620e2f6b23916c8	
	32		0x192b5a87b5124dc287e06cdd4ec7c0c11f67dda6	
	33		0x4e9e2bdc1b0204d1ceeb68fb4159e752c40b9608	
	34		0xf34c57ad9ce45d67fd32dc2737e6263bcc5cc61f	
	35		0xf73bd27d376186310f83cc66e72060aeaccde371	
	36		0xeea482511acd8be783e9be42b48799653b222db4	
	37		0xa2e53196fec8676065b8f32b3e8498e66a4af3cf	
	38		0x670c98185157e1b28d38f7dafb00796b434c8316	
	39		0x441afbb265b93595389aaa66325de792f343f209	
	40		0x7b6c50d20b5edc0bc90eb4b289770514cbc8d547	
	41		0xfde6e862a7ba3534893a3e630e209a24be590b1e	
	42		0xc59611200c20b2e73dfb24c84cedf4792d6daf10	
	43		0x66e3527bee88373d18f91b230b53b569361f0a15	
	44		0xd0fd79c7116198e689275fec9b4c46f4aac73293	
	45		0x65f07406ad4241e7cf4174c5f284267292cdbc32	
	46		0x7b1b5535d45f46542e2b876245b66ea83cde3d8f	
	47		0x7a11620934eb0eb17e10e4a8bbd52aa4b020da0e	
	48		0xbd2fdc5e4e8d0ed7c48c1bad9c2f7793fc2c9303	
	49		0x00432602437252a0622a30676dbaaef3023328b9	
	50		0x09a9c4b25034466a5acd7ff681af1c27e8f97577	
	51		0x4b31481d52aa5e1a261064bbd87ea46479a6be23	
	52		0xaca2ad4aa1264618ab633bf11cbca3cc8fa43091	
	53		0x5caba3ec960efa210f5f3e1c22c567ca475ef3ec	
	54		0x353e3ffcedfd9500141921cf2aebc2e111364dad	

```
|            55          | 0xe1c498c32169c869174ccf2f1e71e7202f45fba7 |
|                        |                                            |
|            56          | 0x5b8519a40d4305813936c7c00a96f5b4ceb603f1 |
|                        |                                            |
|            57          | 0x3b942ae6a6bd328d08804ade771a0775bb3ff8f8 |
|                        |                                            |
|            58          | 0x6f3be60ee1c34372599b8d634be72e168453bf10 |
|                        |                                            |
|            59          | 0xf700c70bac24db0aab1257940661f5b57da6e817 |
|                        |                                            |
|            60          | 0x85ccf60624b13663a290fa808c6bbecaf89523cd |
|                        |                                            |
|            61          | 0xd049be55ab703c44f42167d5d9e939c830df960f |
|                        |                                            |
|            62          | 0xd27a178ccc3b364c7e03d2266093a0d1dfdd9d51 |
|                        |                                            |
|            63          | 0xd73c53fdddbe196b9ab56fcc5c9a4a57ad868cd1 |
|                        |                                            |
|            64          | 0xb59a70a7372f0c121fa71727baaf6588eccec400 |
|                        |                                            |
|            65          | 0x9b5bf379f989f9a499799c12a3202db58b084eed |
|                        |                                            |
|            66          | 0xccabf40f3c1dacf114b5e5f98a73103b4c1f9b55 |
+-----------------------+--------------------------------------------+
```

                              Table 13

MTS Leaf (Level 2) Node Number	OTS Public Key (H(x[0] || x[1] || ... || x[p-1]))	Member of Authentication Path of Message 0
0	0x2db55a72075fcfab5aedbef77bf6b371 dfb489d6e61ad2884a248345e6910618	
1	0x8c6c6a1215bfe7fda10b7754e73cd984 a64823b1ab9d5f50feda6b151c0fee6d	*
2	0xc1fb91de68b3059c273e53596108ec7c f39923757597fe86439e91ce1c25fc84	*
3	0x1b511189baee50251335695b74d67c40 5a04eddaa79158a9090cc7c3eb204cbf	*
4	0xf3bcf088ccf9d00338b6c87e8f822da6 8ec471f88d1561193b3c017d20b3c971	
5	0x40584c059e6cc72fb61f7bd1b9c28e73 c689551e6e7de6b0b9b730fab9237531	
6	0x1b1d09de1ca16ca890036e018d7e73de b39b07de80c19dcc5e55a699f021d880	
7	0x83a82632acaac5418716f4f357f5007f 719d604525dbe1831c09a2ead9400a52	
8	0xccb8b2a1d60f731b5f51910eb427e211 96090d5cd2a077f33968b425301e3fbd	
9	0x616767ebf3c1f3ec662d8c57c630c6ae b31853fd40a18c3d831f5490610c1f16	
10	0x5a4b3e157b66327c75d7f01304d188e2 cecd1b6168240c11a01775d581b01fb6	
11	0xf25744b8a1c2184ba38521801bf4727c 407b85eb5aef8884d8fbb1c12e2f6108	
12	0xaf8189f51874999162890f72e0ef25e6 f76b4ab94dc53569bdd66507f5ab0d8e	
13	0x96251e396756686645f35cd059da329f 7083838d56c9ccacebbaf8486af18844	

14	0x773d5206e40065d3553c3c2ed2500122 e3ee6fd2c91f35a57f084dc839aab1fc		
15	0xcda7fae67ce2c3ed29ce426fdcd3f2a8 eb699e47a67a52f1c94e89726ffe97fa		

Table 14

MTS Interior (Level 1) Node Number	Node Value (H(child_0 || child_1 || ... || child_k-1))	Member of Authentication Path of Message 0
0	0xb6a310deb55ed48004133ece2aebb25e d74defb77ebd8d63c79a42b5b4191b0c	
1	0x71a0c8b767ade2c97ebac069383e4dfb a1c06d5fd3f69a775711ea6470747664	*
2	0x91109fa97662dc88ae63037391ac2650 f6c664ac2448b54800a1df748953af31	*
3	0xd277fb8c89689525f90de567068d6c93 565df3588b97223276ef8e9495468996	*

Table 15

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