Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2016

A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua GmbH
A. Mohaisen
Verisign Labs

July 3, 2015

XMSS: Extended Hash-Based Signatures
draft-irtf-cfrg-xmss-hash-based-signatures-01

Abstract

This note describes the eXtended Merkle Signature Scheme (XMSS), a hash-based digital signature system. It follows existing descriptions in scientific literature. The note specifies the WOTS+ one-time signature scheme, a single-tree (XMSS) and a multi-tree variant (XMSS^MT) of XMSS. Both variants use WOTS+ as a main building block. XMSS provides cryptographic digital signatures without relying on the conjectured hardness of mathematical problems. Instead, it is proven that it only relies on the properties of cryptographic hash functions. XMSS provides strong security guarantees and, besides some special instantiations, is even secure when the collision resistance of the underlying hash function is broken. It is suitable for compact implementations, relatively simple to implement, and naturally resists side-channel attacks. Unlike most other signature systems, hash-based signatures withstand attacks using quantum computers.

Table of Contents

1. Introduction

A (cryptographic) digital signature scheme provides asymmetric message authentication. The key generation algorithm produces a key pair consisting of a private and a public key. A message is signed using a private key to produce a signature. A message/signature pair can be verified using a public key. A One-Time Signature (OTS) scheme allows using a key pair to sign exactly one message securely. A many-time signature system can be used to sign multiple messages.

One-Time Signature schemes, and Many-Time Signature (MTS) schemes composed of them, were proposed by Merkle in 1979 [Merkle79]. They were well-studied in the 1990s and have regained interest from 2006 onwards because of their resistance against quantum-computer-aided

attacks. These kinds of signature schemes are called hash-based signature schemes as they are built out of a cryptographic hash function. Hash-based signature schemes generally feature small private and public keys as well as fast signature generation and verification but large signatures and relatively slow key generation. In addition, they are suitable for compact implementations that benefit various applications and are naturally resistant to most kinds of side-channel attacks.

Some progress has already been made toward standardizing and introducing hash-based signatures. McGrew and Curcio have published an Internet-Draft [DC14] specifying the "textbook" Lamport-Diffie-Winternitz-Merkle (LDWM) scheme based on early publications. Independently, Buchmann, Dahmen and Huelsing have proposed XMSS [BDH11], the eXtended Merkle Signature Scheme, offering better efficiency and a modern security proof. Very recently, the stateless hash-based signature scheme SPHINCS was introduced [BHH15], with the intent of being easier to deploy in current applications. A reasonable next step toward introducing hash-based signatures would be to complete the specifications of the basic algorithms - LDWM, XMSS, SPHINCS and/or variants [Kaliski15].

The eXtended Merkle Signature Scheme (XMSS) [BDH11] is the latest stateful hash-based signature scheme. It has the smallest signatures out of such schemes and comes with a multi-tree variant that solves the problem of slow key generation. Moreover, it can be shown that XMSS is secure, making only mild assumptions on the underlying hash function. Especially, it is not required that the cryptographic hash function is collision-resistant for the security of XMSS.

This document describes a single-tree and a multi-tree variant of XMSS. It also describes WOTS+, a variant of the Winternitz OTS scheme introduced in [Huelsing13] that is used by XMSS. The schemes are described with enough specificity to ensure interoperability between implementations.

This document is structured as follows. Notation is introduced in Section 2. Section 3 describes the WOTS+ signature system. MTS schemes are defined in Section 4: the eXtended Merkle Signature Scheme (XMSS) in Section 4.1, and its Multi-Tree variant (XMSS^MT) in Section 4.2. Parameter sets are described in Section 5. Section 6 describes the rationale behind choices in this note. The IANA registry for these signature systems is described in Section 7. Finally, 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 types. A byte is a sequence of eight bits. 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 of length 3. An array of byte strings is an ordered, indexed set starting with index 0 in which all byte strings have identical length. If not stated or handled otherwise, we assume big-endian representation of data types.

2.2. Operators

When a and b are integers, 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 and b. This operator is sometimes used implicitly in the absence of ambiguity, as in usual mathematical notation.

/ : a / b denotes the quotient of a by b.

% : a % b denotes the non-negative 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.

Arrays are used in the common way, where the i^th element of an array A is denoted A[i]. Byte strings are treated as arrays of bytes where necessary: If X is a byte string, then X[i] denotes its i^th byte, where X[0] is the leftmost byte.

If A and B are byte strings of equal length, then:

A AND B denotes the bitwise logical conjunction operation.

A XOR B denotes the bitwise logical exclusive disjunction 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 X is a x-byte string and Y a y-byte string, then X || Y denotes the concatenation of X and Y, with X || Y = X[0] ... X[x-1] Y[0] ... Y[y-1].

2.3. Functions

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

ceil(x) : returns the smallest integer greater or equal than x.

floor(x) : returns the largest integer less or equal than x.

lg(x) : returns the logarithm to base 2 of x.

2.4. Integer to Byte Conversion

If x and y are non-negative integers, we define Z = toByte(x,y) the y-byte representation of x as the y-byte string that contains the binary representation of x padded with zeros in the most significant bit positions in little endian byte-order.

2.5. Hash Function Address Scheme

The schemes described in this document randomize each hash function call. This means that aside of the initial message digest, for each hash function call a different key and different bitmask is used. These values are pseudorandomly generated using a pseudorandom generator that takes a seed S and a 16-byte address A. The latter is used to select the A-th n-byte block from the PRG output where n is the security parameter. Here we explain the structure of address A. We explain the construction of the addresses in the following sections where they are used.

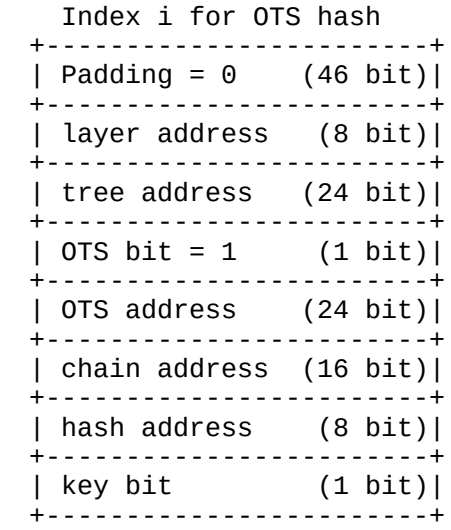
The schemes in the next two sections use two kinds of hash functions parameterized by security parameter n. For the hash tree constructions a hash function that maps 2n-byte inputs and an n-byte

key to n-byte outputs is used. To randomize this function, 3n bytes are needed - n bytes for the key and 2n bytes for a bitmask. For the one-time signature scheme constructions a hash function that maps n-byte inputs and n-byte keys to n-byte outputs is used. To randomize this function, 2n bytes are needed - n bytes for the key and n bytes for a bitmask. Consequently, three addresses are needed for the first function and two addresses for the second one.

There are three different address formats for the different use cases. One format for the hashes used in one-time signature schemes, one for hashes used within the main Merkle-tree construction, and one for hashes used in the L-trees. The latter being used to compress one-time public keys. All these formats share as much format as possible. In the following we describe these formats in detail.

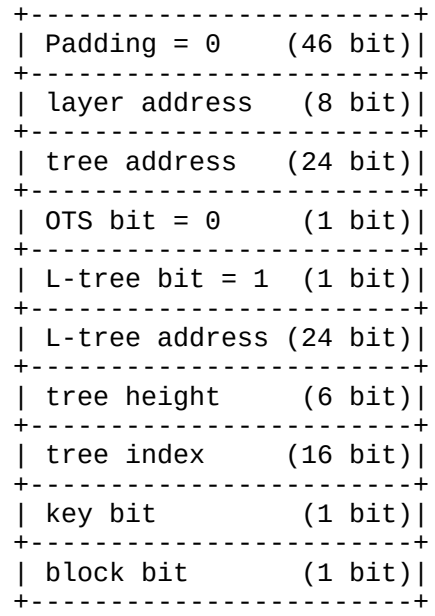
An address is structured as follows. It always starts with 46 zero bits in the most significant bits. These are followed by a layer address of 8 bits, and a tree address of 24 bits. The next bit decides whether it is an OTS construction or a hash tree address. This OTS bit is set to zero for a tree hash address and it is set to one for an OTS hash address.

We first describe the OTS address case as the hash tree case again splits into two cases. In this case, the OTS bit is followed by a 24-bit OTS address that encodes the index of the OTS key pair within a tree. The next 16 bits encode the chain address followed by 8 bits that encode the address of the hash function call within a chain. The key bit is used to generate two different addresses for one hash function call. The bit is set to one to generate the key. To generate the n-byte bitmask, the key bit is set to zero.

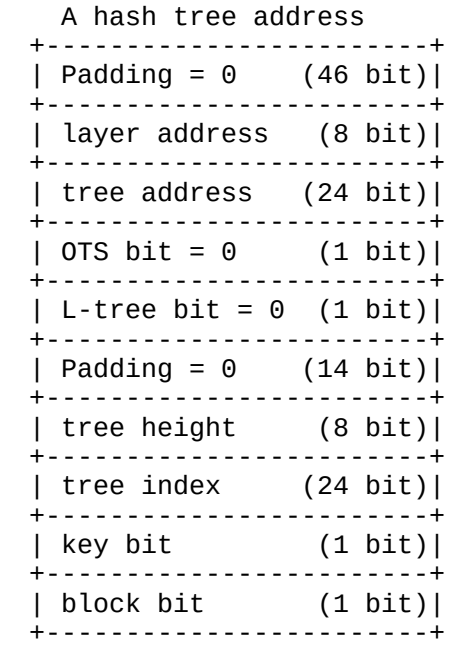


Now we describe the hash tree address case. This case again splits into two. The OTS bit is followed by an L-tree bit. This bit is set to zero in case of an L-tree and set to one for main tree nodes. We first discuss the L-tree case. In this case the L-tree bit is followed by a 24 bit L-tree address, encoding the index of the leaf computed with this L-tree. The next 6 bits encode the height of the node inside the L-tree and the following 16 bit encode the index of the node at that height, inside the L-tree. The last two bits are used to generate three different addresses for one node. The first of these bits is set to one to generate the key. In that case the last bit is always zero. To generate the 2n-byte bitmask, the key bit is set to zero. The most significant n bytes are generated using the address with the last bit zero. The least significant bytes are generated using the address with the last bit set to one.

An L-tree address



We now describe the remaining format for the main tree hash addresses. In this case the L-tree bit is set to zero and followed by 14 zero bits padding as there are less hash tree addresses required. The next 8 bits encode the height of the tree node to be computed within the tree, followed by 24 bits that encode the index of this node at that height. The last two bits are used to generate three different addresses for one node as described for the L-tree case. The first of these bits is set to one to generate the key. In that case the last bit is always zero. To generate the 2n-byte bitmask, the key bit is set to zero. The most significant n bytes are generated using the address with the last bit zero. The least significant bytes are generated using the address with the last bit set to one.



All fields within these addresses encode unsigned integers. When describing the generation of addresses we use setter-methods that take positive integers and set the bits of a field to the binary representation of that integer of the length of the field. We also assume that setting the L-tree bit to zero, does also set the (second) padding block to zero.

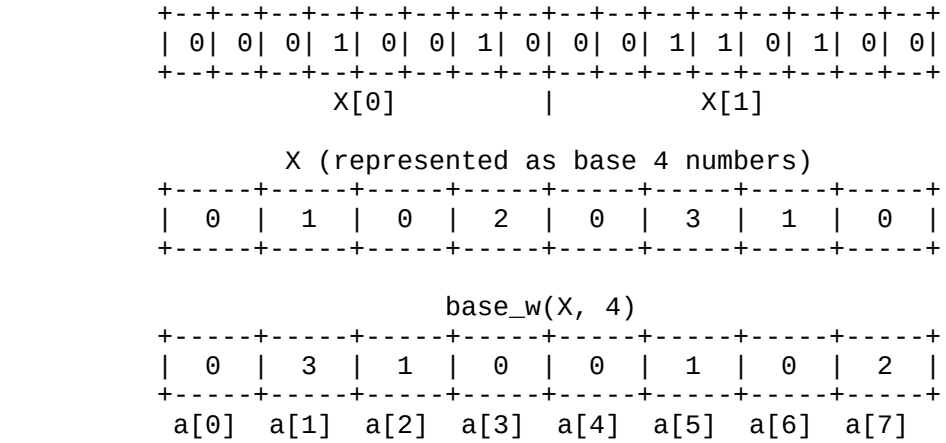
2.6. Strings of Base w Numbers

A byte string can be considered as a string of base w numbers, i.e. integers in the set {0, ... , w - 1}. The correspondence is defined by the function base_w(X, w) as follows. If X is an m-byte string, w is a member of the set {4, 16}, then base_w(X, w) outputs a length 8 * m / lg(w) array of integers between 0 and w - 1.

Algorithm 1: base_w(X, w)

```
    int in = 0;
    int out = 0;
    unsigned int total = 0;
    int bits = 0;
    int consumed;

    for ( consumed = 0; consumed < 8 * m; consumed += lg(w) ) {
        if ( bits == 0 ) {
            total = X[m - 1 - in];
            in++;
            bits += 8;
        }
        bits -= lg(w);
        basew[out] = (total >> bits) AND (w - 1);
        out++;
    }
    return basew;
```

For example, if X is the (big endian) byte string 0x1234, then base_w(X, 4) returns the array a = {0, 3, 1, 0, 0, 1, 0, 2}.



2.7. Member Functions

To simplify algorithm descriptions, we assume the existence of member functions. If a complex data structure like a public key PK contains a value X then getX(PK) returns the value of X for this public key. Accordingly, setX(PK, X, Y) sets value X in PK to the value hold by Y.

3. Primitives

3.1. WOTS+ One-Time Signatures

This section describes the WOTS+ one-time signature system, in a version similar to [Huelsing13]. WOTS+ is a one-time signature scheme; while a private key can be used to sign any message, each private key MUST be used only once to sign a single message. In particular, if a secret key is used to sign two different messages, the scheme becomes insecure.

The section starts with an explanation of parameters. Afterwards, the so-called chaining function, which forms the main building block of the WOTS+ scheme, is explained. It follows a description of the algorithms for key generation, signing and verification. Finally, pseudorandom key generation is discussed.

3.1.1. WOTS+ Parameters

WOTS+ uses the parameters m, n, and w; they all take positive integer values. These parameters are summarized as follows:

m : the message length in bytes

n : the length, in bytes, of a secret key, public key, or signature element

w : the Winternitz parameter; it is a member of the set {4, 16}

The parameters are used to compute values len, len_1 and len_2:

len : the number of n-byte string elements in a WOTS+ secret key, public key, and signature. It is computed as len = len_1 + len_2, with len_1 = ceil(8m/lg(w)) and len_2 = floor(lg(len_1*(w-1))/lg(w)) + 1

The value of n is determined by the cryptographic hash function used for WOTS+. The hash function is chosen to ensure an appropriate level of security. The value of m is the input length that can be processed by the signing algorithm. It is often the length of a message digest. The parameter w can be chosen from the set {4, 16}. A larger value of w results in shorter signatures but slower overall signing operations; it has little effect on security. Choices of w are limited to the values 4 and 16 since these values yield optimal trade-offs and easy implementation.

3.1.1.1. WOTS+ Functions

The WOTS+ algorithm uses a keyed cryptographic hash function F. F accepts and returns byte strings of length n using keys of length n. Security requirements on F are discussed in Section 8. In addition, WOTS+ uses a pseudorandom generator G. G takes as input an n-byte key and a 16-byte index and generates pseudorandom outputs of length n. Security requirements on G are discussed in Section 8.

3.1.2. WOTS+ Chaining Function

The chaining function (Algorithm 2) computes an iteration of F on an n-byte input using outputs of G. It takes a hash function address as input. This address will have the first 119 bits set to encode the address of this chain. In each iteration, one output of G is used as key for F and a second output is XORed to the intermediate result before it is processed by F. In the following, ADRS is a 16-byte hash function address as specified in Section 2.5 and SEED is an n-byte string, both used to generate the outputs of G. The chaining function takes as input an n-byte string X, a start index i, a number of steps s, as well as ADRS and SEED. The chaining function returns as output the value obtained by iterating F for s times on input X, using the outputs of G.

Algorithm 2: Chaining Function

```
if ( s is equal to 0 ) {
  return X;
}
if ( (i+s) > w-1 ) {
  return NULL;
}
byte[n] tmp = chain(X, i, s-1, SEED, ADRS);
ADRS.setHashAddress(i+s-1);
ADRS.setKeyBit(0);
BM = G(SEED, ADRS);
ADRS.setKeyBit(1);
KEY = G(SEED, ADRS);
tmp = F(KEY, tmp XOR BM);
return tmp;
```

3.1.3. WOTS+ Private Key

The private key in WOTS+, denoted by sk, is a length len array of n-byte strings. This private key MUST be only used to sign exactly one message. Each n-byte string MUST either be selected randomly from the uniform distribution or using a cryptographically secure pseudorandom procedure. In the latter case, the security of the used

procedure MUST at least match that of the WOTS+ parameters used. For a further discussion on pseudorandom key generation see the end of this section. The following pseudocode (Algorithm 3) describes an algorithm for generating sk.

Algorithm 3: Generating a WOTS+ Private Key

```
for ( i = 0; i < len; i = i + 1 ) {
  set sk[i] to a uniformly random n-byte string;
}
return sk;
```

3.1.4. WOTS+ Public Key

A WOTS+ key pair defines a virtual structure that consists of len hash chains of length w. The len n-byte strings in the secret key each define the start node for one hash chain. The public key consists of the end nodes of these hash chains. Therefore, like the secret key, the public key is also a length len array of n-byte strings. To compute the hash chain, the chaining function (Algorithm 2) is used. A hash function address ADRS and a seed SEED has to be provided by the calling algorithm. This address will encode the address of the WOTS+ key pair within a greater structure. Hence, a WOTS+ algorithm MUST NOT manipulate any other fields of ADRS than chain address, hash address and key bit. Please note that the SEED used here is public information also available to a verifier. The following pseudocode (Algorithm 4) describes an algorithm for generating the public key pk, where sk is the private key.

Algorithm 4 (WOTS_genPK): Generating a WOTS+ Public Key From a Private Key

```
for ( i = 0; i < len; i = i + 1 ) {
  ADRS.setChainAddress(i);
  pk[i] = chain(sk[i], 0, w-1, SEED, ADRS);
}
return pk;
```

3.1.5. WOTS+ Signature Generation

A WOTS+ signature is a length len array of n-byte strings. The WOTS+ signature is generated by mapping a message to len integers between 0 and w - 1. To this end, the message is transformed into base w numbers using the base_w function defined in Section 2.6. Next, a checksum is computed and appended to the transformed message as len_2 base w numbers using the base_w function. Each of the base w integers is used to select a node from a different hash chain. The signature is formed by concatenating the selected nodes. The

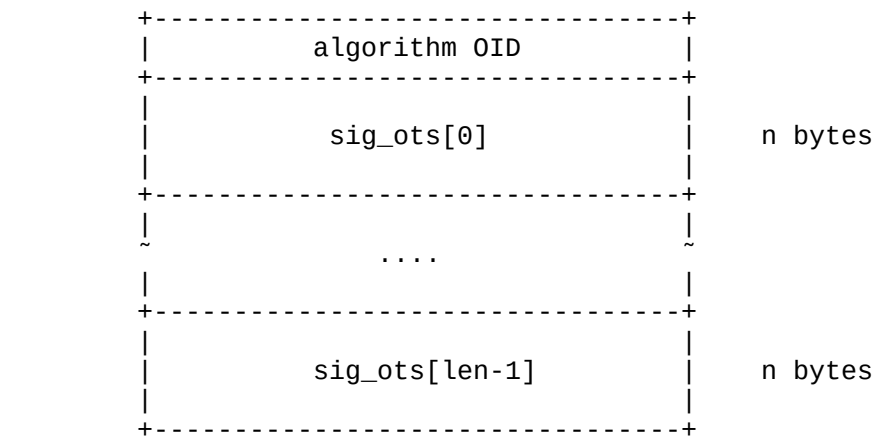
pseudocode for signature generation is shown below (Algorithm 5), where M is the message and sig is the resulting signature.

Algorithm 5 (WOTS_sign): Generating a signature from a private key and a message

```
csum = 0;

// convert message to base w
msg = base_w(M,w);

// compute checksum
for ( i = 0; i < len_1; i = i + 1 ) {
      csum = csum + w - 1 - msg[i];
}

// Convert csum to base w
csum = csum << ( 8 - ( len_2 % 8 ));
len_2_bytes = ceil( len_2 / 8 );
msg = msg || base_w(toByte(csum, len_2_bytes), w);
for ( i = 0; i < len; i = i + 1 ) {
     ADRS.setChainAddress(i);
     sig[i] = chain(sk[i], 0, msg[i], SEED, ADRS);
}
return sig;
```

The data format for a signature is given below.

WOTS+ Signature



3.1.6. WOTS+ Signature Verification

In order to verify a signature sig on a message M, the verifier computes a WOTS+ public key value from the signature. This can be done by "completing" the chain computations starting from the signature values, using the base w values of the message hash and its checksum. This step, called WOTS_pkFromSig, is described below in Algorithm 6. The result of WOTS_pkFromSig is then compared to the given public key. If the values are equal, the signature is accepted. Otherwise, the signature is rejected.

Algorithm 6 (WOTS_pkFromSig): Computing a WOTS+ public key from a message and its signature

```
csum = 0;

// convert message to base w
msg = base_w(M,w);

// compute checksum
for ( i = 0; i < len_1; i = i + 1 ) {
      csum = csum + w - 1 - msg[i];
}

// Convert csum to base w
csum = csum << ( 8 - ( len_2 % 8 ));
len_2_bytes = ceil( len_2 / 8 );
msg = msg || base_w(toByte(csum, len_2_bytes), w);
for ( i = 0; i < len; i = i + 1 ) {
     ADRS.setChainAddress(i);
     tmp_pk[i] = chain(sig[i], msg[i], w-1-msg[i], SEED, ADRS);
}
return tmp_pk;
```

Note: XMSS uses WOTS_pkFromSig to compute a public key value and delays the comparison to a later point.

3.1.7. Pseudorandom Key Generation

An implementation MAY use a cryptographically secure pseudorandom method to generate the secret key from a single n-byte value. For example, the method suggested in [BDH11] and explained below MAY be used. Other methods MAY be used. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used WOTS+ parameters.

The advantage of generating the secret key elements from a random n-byte string is that only this n-byte string needs to be stored

instead of the full secret key. The key can be regenerated when needed. The suggested method from [BDH11] can be described using G. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This string S is stored as secret key. The secret key elements are computed as sk[i] = G'(S, toByte(i,16)) whenever needed. Please note that this seed S MUST be different from the seed SEED used to randomize the hash function calls. Also, this seed S MUST be kept secret.

4. Schemes

In this section, the eXtended Merkle Signature Scheme (XMSS) is described using WOTS+. XMSS comes in two flavors: First, a single-tree variant (XMSS) and second a multi-tree variant (XMSS^MT). Both allow combining a large number of WOTS+ key pairs under a single small public key. The main ingredient added is a binary hash tree construction. XMSS uses a single hash tree while XMSS^MT uses a tree of XMSS key pairs.

4.1. XMSS: eXtended Merkle Signature Scheme

XMSS is a method for signing a potentially large but fixed number of messages. It is based on the Merkle signature scheme. XMSS uses five cryptographic components: WOTS+ as OTS method, two additional cryptographic hash functions H and H_m, a pseudorandom function PRF_m, and a pseudorandom generator G. One of the main advantages of XMSS with WOTS+ is that it does not rely on the collision resistance of the used hash functions but on weaker properties. Each XMSS public/private key pair is associated with a perfect binary tree, every node of which contains an n-byte value. Each tree leaf contains a special tree hash of a WOTS+ public key value. Each non-leaf tree node is computed by first concatenating the values of its child nodes, computing the XOR with a bitmask, and applying the keyed hash function H to the result. The bitmasks and the keys for the hash function H are generated from a (public) seed that is part of the public key using the pseudorandom generator G. The value corresponding to the root of the XMSS tree forms the XMSS public key together with the seed.

To generate a key pair that can be used to sign 2^h messages, a tree of height h is used. XMSS is a stateful signature scheme, meaning that the secret key changes after every signature. To prevent one-time secret keys from being used twice, the WOTS+ key pairs are numbered from 0 to (2^h)-1 according to the related leaf, starting from index 0 for the leftmost leaf. The secret key contains an index that is updated after every signature, such that it contains the index of the next unused WOTS+ key pair.

A signature consists of the index of the used WOTS+ key pair, the WOTS+ signature on the message and the so-called authentication path. The latter is a vector of tree nodes that allow a verifier to compute a value for the root of the tree starting from a WOTS+ signature. A verifier computes the root value and compares it to the respective value in the XMSS public key. If they match, the signature is valid. The XMSS secret key consists of all WOTS+ secret keys and the actual index. To reduce storage, a pseudorandom key generation procedure, as described in [BDH11], MAY be used. The security of the used method MUST at least match the security of the XMSS instance.

4.1.1. XMSS Parameters

XMSS has the following parameters:

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

n : the length in bytes of each node

m : the length of the message digest

w : the Winternitz parameter as defined for WOTS+ in Section 3.1

There are N = 2^h leaves in the tree.

For XMSS and XMSS^MT, secret and public keys are denoted by SK and PK. For WOTS+, secret and public keys are denoted by sk and pk, respectively. XMSS and XMSS^MT signatures are denoted by Sig. WOTS+ signatures are denoted by sig.

4.1.2. XMSS Hash Functions

Besides the cryptographic hash function F required by WOTS+, XMSS uses four more functions:

A cryptographic hash function H. H accepts n-byte keys and byte strings of length (2 * n) and returns an n-byte string.

A cryptographic hash function H_m. H_m accepts m-byte keys and byte strings of arbitrary length and returns an m-byte string.

A pseudorandom function PRF_m. PRF_m accepts byte strings of arbitrary length and an m-byte key and returns an m-byte string.

A pseudorandom generator G. G takes as input an n-byte key and a 16-byte index and generates pseudorandom outputs of length n.

4.1.3. XMSS Private Key

An XMSS private key contains N = 2^h WOTS+ private keys, the leaf index idx of the next WOTS+ private key that has not yet been used and SK_PRF, an m-byte key for the PRF. The leaf index idx is initialized to zero when the XMSS private key is created. The PRF key SK_PRF MUST be sampled from a secure source of randomness that follows the uniform distribution. The WOTS+ secret keys MUST be generated as described in Section 3.1. To reduce the secret key size, a cryptographic pseudorandom method MAY be used as discussed at the end of this section. For the following algorithm descriptions, the existence of a method getWOTS_SK(SK,i) is assumed. This method takes as inputs an XMSS secret key SK and an integer i and outputs the i^th WOTS+ secret key of SK.

4.1.4. Randomized Tree Hashing

To improve readability we introduce a function RAND_HASH(LEFT, RIGHT, SEED, ADRS) that does the randomized hashing. It takes as input two n-byte values LEFT and RIGHT that represent the left and the right half of the hash function input, the seed SEED for G and the address ADRS of this hash function call. RAND_HASH first uses G with SEED and ADRS to generate a key KEY and n-byte bitmasks BM_0, BM_1. Then it returns the randomized hash H(KEY, (LEFT XOR BM_0)||(RIGHT XOR BM_1)).

Algorithm 7: RAND_HASH

```
ADRS.setKeyBit(0);
ADRS.setBlockBit(0);
BM_0 = G(SEED, ADRS);
ADRS.setBlockBit(1);
BM_1 = G(SEED, ADRS);
ADRS.setKeyBit(1);
ADRS.setBlockBit(0);
KEY = G(SEED, ADRS);
return H(KEY, (LEFT XOR BM_0) || (RIGHT XOR BM_1));
```

4.1.5. L-Trees

To compute the leaves of the binary hash tree, a so-called L-tree is used. An L-tree is an unbalanced binary hash tree, distinct but similar to the main XMSS binary hash tree. The algorithm ltree (Algorithm 8) takes as input a WOTS+ public key pk and compresses it to a single n-byte value pk[0]. Towards this end it also takes an address ADRS as input that encodes the address of the L-tree. The algorithm uses G and the seed SEED generated during public key generation.

Algorithm 8: ltree

```
unsigned int len' = len;
unsigned int j = 0;
ADRS.setTreeHeight(0);
while ( len' > 1 ) {
  for ( i = 0; i < floor(len' / 2); i = i + 1 ) {
    ADRS.setTreeIndex(i);
    pk[i] = RAND_HASH(pk[2i], pk[2i + 1], SEED, ADRS);
  }
  if ( len' % 2 == 1 ) {
    pk[floor(len' / 2) + 1] = pk[len'];
  }
  len' = ceil(len' / 2);
  ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
}
return pk[0];
```

4.1.6. TreeHash

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine treeHash (Algorithm 9) accepts an XMSS secret key SK, an unsigned integer s (the start index), an unsigned integer t (the target node height), a seed SEED, and an address ADRS that encodes the address of the containing tree. For the height of a node within a tree counting starts with the leaves at height zero. The treeHash algorithm returns the root node of a tree of height t with the leftmost leaf being the hash of the WOTS+ pk with index s. It is REQUIRED that s % 2^t = 0, i.e. that the leaf at index s is a left most leaf of a sub-tree of height t. Otherwise the hash-addressing scheme fails. The treeHash algorithm uses a stack holding up to (t-1) n-byte strings, with the usual stack functions push() and pop().

Algorithm 9: treeHash

```
if( s % (1 << t) != 0 ) return -1;
for ( i = 0; i < 2^t; i = i + 1 ) {
  ADRS.setOTSBit(1);
  ADRS.setOTSAddress(s+i);
  pk = WOTS_genPK (getWOTS_SK(SK, s+i), SEED, ADRS);
  ADRS.setOTSBit(0);
  ADRS.setLTreeBit(1);
  ADRS.setLTreeAddress(s+i);
  node = ltree(pk, SEED, ADRS);
  ADRS.setLTreeBit(0);
  ADRS.setTreeHeight(0);
  ADRS.setTreeIndex(i+s);
  while ( Top node on Stack has same height t' as node ) {
     ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
     node = RAND_HASH(Stack.pop(), node, SEED, ADRS);
     ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
  }
  Stack.push(node);
}
return Stack.pop();
```

4.1.7. XMSS Public Key

The XMSS public key is computed as described in XMSS_genPK (Algorithm 10). The algorithm takes the XMSS secret key SK, and the tree height h. The XMSS public key PK consists of the root of the binary hash tree and the seed SEED. SEED is generated as a uniformly random n-byte string. Although SEED is public, it is important that it is generated using a good entropy source. The root is computed using treeHash. For XMSS, there is only a single main tree. Hence, the used address is set to the all-zero-string.

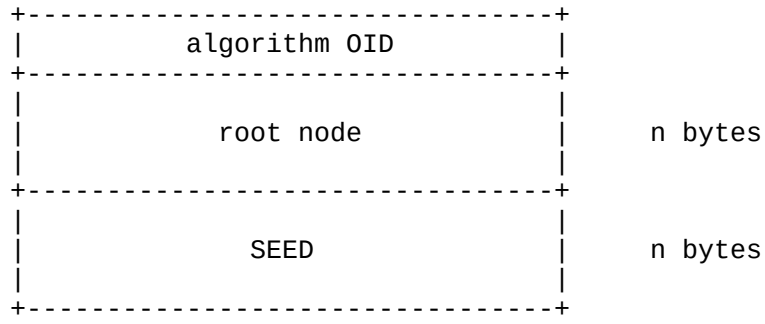
Algorithm 10: XMSS_genPK - Generate an XMSS public key from an XMSS private key

```
set SEED to a uniformly random n-byte string;
ADRS = toByte(0,16);
root = treeHash(SK, 0, h, SEED, ADRS);
PK = root || SEED;
return PK;
```

Public and private key generation MAY be interleaved to save space. Especially, when a pseudorandom method is used to generate the secret key, generation MAY be done when the respective WOTS+ key pair is needed by treeHash.

The format of an XMSS public key is given below.

XMSS Public Key



4.1.8. XMSS Signature

An XMSS signature is a (4 + m + (len + h) * n)-byte string consisting of

the index idx_sig of the used WOTS+ key pair (4 bytes),

a byte string r used for randomized message hashing (m bytes),

a WOTS+ signature sig_ots (len * n bytes),

the so-called authentication path 'auth' for the leaf associated with the used WOTS+ key pair (h * n bytes).

The authentication path is an array of h n-byte strings. It contains the siblings of the nodes on the path from the used leaf to the root. It does not contain the nodes on the path itself. These nodes are needed by a verifier to compute a root node for the tree from the WOTS+ public key. A node Node is addressed by its position in the tree. Node(x,y) denotes the x^th node on level y with x = 0 being the leftmost node on a level. The leaves are on level 0, the root is on level h. An authentication path contains exactly one node on every layer 0 <= x <= h-1. For the i^th WOTS+ key pair, counting from zero, the j^th authentication path node is

Node(j, floor(i / (2^j)) XOR 1)

Given an XMSS secret key SK and seed SEED, all nodes in a tree are determined. Their value is defined in terms of treeHash(Algorithm 9). Hence, one can compute the authentication path:

```
  ADRS = toByte(0, 16);
  for (j = 0; j < h; j++) {
    k = floor(i / (2^j)) XOR 1;
    auth[j] = treeHash(SK, k * 2^j, j, SEED, ADRS);
    }
```

The data format for a signature is given below.

XMSS Signature

4.1.9. XMSS Signature Generation

To compute the XMSS signature of a message M with an XMSS private key, the signer first computes a randomized message digest. Then a WOTS+ signature of the message is computed using the next unused WOTS+ private key. Next, the authentication path is computed. Finally, the secret key is updated, i.e. idx is incremented. An implementation MUST NOT output the signature before the updated private key.

The node values of the authentication path MAY be computed in any way. This computation is assumed to be performed by the subroutine

buildAuth for the function XMSS_sign, as below. The fastest alternative is to store all tree nodes and set the array in the signature by copying them, respectively. The least storage-intensive alternative is to recompute all nodes for each signature online. There exist several algorithms in between, with different time/ storage trade-offs. For an overview see [BDS09]. Note that the details of this procedure are not relevant to interoperability; it is not necessary to know any of these details in order to perform the signature verification operation. As a consequence, buildAuth is not specified here.

The algorithm XMSS_sign (Algorithm 11) described below calculates an updated secret key SK and a signature on a message M. XMSS_sign takes as inputs a message M of an arbitrary length, an XMSS secret key SK and seed SEED. It returns the byte string containing the concatenation of the updated secret key SK and the signature Sig.

Algorithm 11: XMSS_sign - Generate an XMSS signature and update the XMSS secret key

```
idx_sig = getIdx(SK);
ADRS = toByte(0,16);
auth = buildAuth(SK, idx_sig, SEED, ADRS);
byte[m] r = PRF_m(getSK_PRF(SK), M);
byte[m] M' = H_m(r, M);
ADRS.setOTSBit(0);
ADRS.setOTSAddress(idx_sig);
sig_ots = WOTS_sign(getWOTS_SK(SK, idx_sig), M', SEED, ADRS);
Sig = (idx_sig || r || sig_ots || auth);
setIdx(SK, idx_sig + 1);
return (SK || Sig);
```

4.1.10. XMSS Signature Verification

An XMSS signature is verified by first computing the message digest using randomness r and a message M. Then the used WOTS+ public key pk_ots is computed from the WOTS+ signature using WOTS_pkFromSig. The WOTS+ public key in turn is used to compute the corresponding leaf using an L-tree. The leaf, together with index idx_sig and authentication path auth is used to compute an alternative root value for the tree. These first steps are done by XMSS_rootFromSig (Algorithm 12). The verification succeeds if and only if the computed root value matches the one in the XMSS public key. In any other case it MUST return fail.

The main part of XMSS signature verification is done by the function XMSS_rootFromSig (Algorithm 12) described below. XMSS_rootFromSig takes as inputs an XMSS signature Sig, a message M, and seed SEED.

XMSS_rootFromSig returns an n-byte string holding the value of the root of a tree defined by the input data.

Algorithm 12: XMSS_rootFromSig - Compute a root node using an XMSS signature, a message, and seed SEED

```
byte[m] M' = H_m(r, M);
ADRS = toByte(0,16);
ADRS.setOTSBit(1);
ADRS.setOTSAddress(idx_sig);
pk_ots = WOTS_pkFromSig(sig_ots, M', SEED, ADRS);
ADRS.setOTSBit(0);
ADRS.setLTreeBit(1);
ADRS.setLTreeAddress(idx_sig);
byte[n][2] node;
node[0] = ltree(pk_ots, SEED, ADRS);
ADRS.setLTreeBit(0);
ADRS.setTreeIndex(idx_sig);
for ( k = 0; k < h; k = k + 1 ) {
  ADRS.setTreeHeight(k);
  if ( floor(idx_sig / (2^k)) % 2 is equal to 0 ) {
    ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
    node[1] = RAND_HASH(node[0], auth[k], SEED, ADRS);
  } else {
    ADRS.setTreeIndex(ADRS.getTreeIndex() - 1 / 2);
    node[1] = RAND_HASH(auth[k], node[0], SEED, ADRS);
  }
  node[0] = node[1];
}
return node[0];
```

The full XMSS signature verification is depicted below. XMSS^MT uses only XMSS_rootFromSig and delegates the comparison to a later comparison of data depending on its output.

Algorithm 13: XMSS_verify - Verify an XMSS signature using an XMSS signature, the corresponding XMSS public key and a message

```
byte[n] node = XMSS_rootFromSig(Sig, M, getSEED(PK));
if ( node is equal to root in PK ) {
  return true;
} else {
  return false;
}
```

4.1.11. Pseudorandom Key Generation

An implementation MAY use a cryptographically secure pseudorandom method to generate the XMSS secret key from a single n-byte value. For example, the method suggested in [BDH11] and explained below MAY be used. Other methods MAY be used. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used XMSS parameters.

For XMSS a similar method than the one used for WOTS+ can be used. The suggested method from [BDH11] can be described using G. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This seed S MUST NOT be confused with the public seed SEED. The seed S MUST be independent of SEED and as it is the main secret, it MUST be kept secret. This seed S is used to generate an n-byte value S_ots for each WOTS+ key pair. The n-byte value S_ots can then be used to compute the respective WOTS+ secret key using the method described in Section 3.1.7. The seeds for the WOTS+ key pairs are computed as S_ots[i] = G(S,i). The second parameter of G is the index i of the WOTS+ key pair, represented as 16-byte string in the common way. An advantage of this method is that a WOTS+ key can be computed using only len + 1 evaluations of G when S is given.

4.1.12. Free Index Handling and Partial Secret Keys

Some applications might require to work with partial secret keys or copies of secret keys. Examples include delegation of signing rights / proxy signatures, and load balancing. Such applications MAY use their own key format and MAY use a signing algorithm different from the one described above. The index in partial secret keys or copies of a secret key MAY be manipulated as required by the applications. However, applications MUST establish means that guarantee that each index and thereby each WOTS+ key pair is used to sign only a single message.

4.2. XMSS^MT: Multi-Tree XMSS

XMSS^MT is a method for signing a large but fixed number of messages. It was first described in [HRB13]. It builds on XMSS. XMSS^MT uses a tree of several layers of XMSS trees. The trees on top and intermediate layers are used to sign the root nodes of the trees on the respective layer below. Trees on the lowest layer are used to sign the actual messages. All XMSS trees have equal height.

Consider an XMSS^MT tree of total height h that has d layers of XMSS trees of height h / d. Then layer d - 1 contains one XMSS tree,

layer d - 2 contains 2^(h / d) XMSS trees, and so on. Finally, layer 0 contains 2^(h - h / d) XMSS trees.

4.2.1. XMSS^MT Parameters

In addition to all XMSS parameters, an XMSS^MT system requires the number of tree layers d, specified as an integer value that divides h without remainder. The same tree height h / d and the same Winternitz parameter w are used for all tree layers.

All the trees on higher layers sign root nodes of other trees which are n-byte strings. Hence, no message compression is needed and WOTS+ is used to sign the root nodes themselves instead of their hash values. Hence the WOTS+ message length for these layers is n not m. Accordingly, the values of len_1, len_2 and len change for these layers. The parameters len_1_n, len_2_n, and len_n denote the respective values computed using n as message length for WOTS+.

4.2.2. XMSS Algorithms Without Message Hash

As all XMSS trees besides those on layer 0 are used to sign short fixed length messages, the initial message hash can be omitted. In the description below XMSS_sign_wo_hash and XMSS_rootFromSig_wo_hash are versions of XMSS_sign and XMSS_rootFromSig, respectively, that omit the initial message hash. They are obtained by setting M' = M in the above algorithms. Accordingly, the evaluations of H_m and PRF_m MUST be omitted. This also means that no randomization element r for the message hash is required. XMSS signatures generated by XMSS_sign_wo_hash and verified by XMSS_rootFromSig_wo_hash MUST NOT contain a value r.

4.2.3. XMSS^MT Private Key

An XMSS^MT private key SK_MT consists of one reduced XMSS private key for each XMSS tree. These reduced XMSS private keys contain no pseudorandom function key and no index. Instead, SK_MT contains a single m-byte pseudorandom function key SK_PRF and a single (ceil(h / 8))-byte index idx_MT. The index is a global index over all WOTS+ key pairs of all XMSS trees on layer 0. It is initialized with 0. It stores the index of the last used WOTS+ key pair on the bottom layer, i.e. a number between 0 and 2^h - 1.

The algorithm descriptions below uses a function getXMSS_SK(SK, x, y) that outputs the reduced secret key of the x^th XMSS tree on the y^th layer.

4.2.4. XMSS^MT Public Key

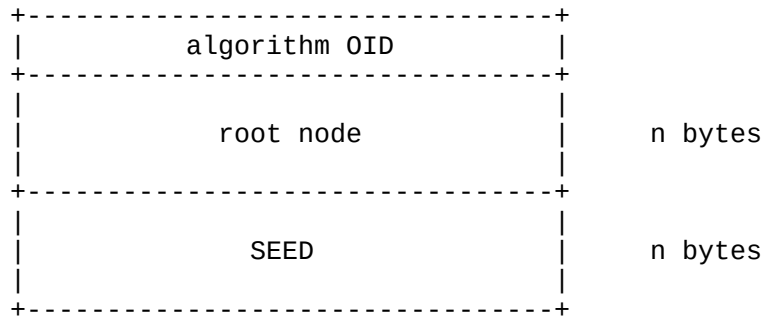
The XMSS^MT public key PK_MT contains the root of the single XMSS tree on layer d-1 and the seed SEED. The pseudorandom generator G is used with SEED to generate the bitmasks and keys for all XMSS trees. Algorithm 14 shows pseudocode to generate PK_MT. First, the n-byte SEED is chosen uniformly at random. The n-byte root node of the top layer tree is computed using treeHash. The algorithm XMSSMT_genPK takes the XMSS^MT secret key SK_MT as an input and outputs an XMSS^MT public key PK_MT.

Algorithm 14: XMSSMT_genPK - Generate an XMSS^MT public key from an XMSS^MT private key

```
set SEED to a uniformly random n-byte string;
ADRS = toByte(0,16);
ADRS.setLayerAddress(d-1);
root = treeHash(getXMSS_SK(SK_MT, 0, d - 1), 0, h / d, SEED, ADRS);
PK_MT = root || SEED;
return PK_MT;
```

The format of an XMSS^MT public key is given below.

XMSS^MT Public Key



4.2.5. XMSS^MT Signature

An XMSS^MT signature Sig_MT is a byte string of length (ceil(h / 8) + m + (h + len + (d - 1) * len_n) * n). It consists of

the index idx_sig of the used WOTS+ key pair on the bottom layer (ceil(h / 8) bytes),

a byte string r used for randomized message hashing (m bytes),

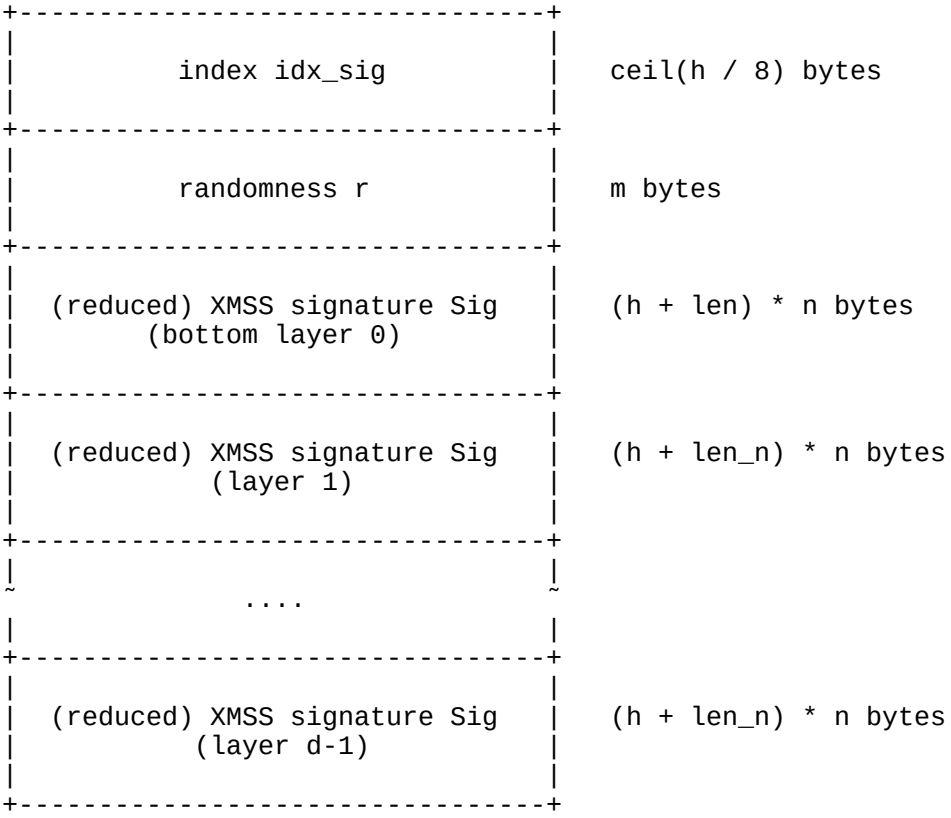
one reduced XMSS signature ((h + len) * n bytes),

d-1 reduced XMSS signatures with message length n ((h + len_n) * n bytes).

The reduced XMSS signatures contain no index idx and no byte string r. They only contain a WOTS+ signature sig_ots and an authentication path auth. The first reduced XMSS signature contains a WOTS+ signature that consists of len n-byte elements. The remaining reduced XMSS signatures contain a WOTS+ signature on an n-byte message that consists of len_n n-byte elements.

The data format for a signature is given below.

XMSS^MT signature



4.2.6. XMSS^MT Signature Generation

To compute the XMSS^MT signature Sig_MT of a message M using an XMSS^MT private key SK_MT and seed SEED, XMSSMT_sign (Algorithm 15) described below uses XMSS_sign and XMSS_sign_wo_hash as defined in

Section 4.2.2. First, the signature index is set to idx. Next, PRF_m is used to compute a pseudorandom m-byte string r. This m-byte string is then used to compute a randomized message digest of length m. The message digest is signed using the WOTS+ key pair on the bottom layer with absolute index idx. The authentication path for the WOTS+ key pair is computed as well as the root of the containing XMSS tree. The root is signed by the parent XMSS tree. This is repeated until the top tree is reached.

Algorithm 15: XMSSMT_sign - Generate an XMSS^MT signature and update the XMSS^MT secret key

```
ADRS = toByte(0,16);
SK_PRF = getSK_PRF(SK_MT);
idx_sig = getIdx(SK_MT);
setIdx(SK_MT, idx_sig + 1);
Sig_MT = idx_sig;
unsigned int idx_tree = (h - h / d) most significant bits of idx_sig;
unsigned int idx_leaf = (h / d) least significant bits of idx_sig;
SK = idx_leaf || SK_PRF || getXMSS_SK(SK_MT, idx_tree, 0);
ADRS.setLayerAddress(0);
ADRS.setTreeAddress(idx_tree);
Sig_tmp = XMSS_sign(M, SK, SEED, ADRS);
Sig_tmp = Sig_tmp without idx;
Sig_MT = Sig_MT || Sig_tmp;
for ( j = 1; j < d; j = j + 1 ) {
   root = treeHash(SK, 0, h / d, SEED, ADRS);
   idx_leaf = (h / d) least significant bits of idx_tree;
   idx_tree = (h - j * (h / d)) most significant bits of idx_tree;
   SK = idx_leaf || SK_PRF || getXMSS_SK(SK_MT, idx_tree, j);
   ADRS.setLayerAddress(j);
   ADRS.setTreeAddress(idx_tree);
   Sig_tmp = XMSS_sign_wo_hash(root, SK, SEED, ADRS)
        with idx removed;
   Sig_MT = Sig_MT || Sig_tmp;
}
return SK_MT || Sig_MT;
```

Algorithm 15 is only one method to compute XMSS^MT signatures. Especially, there exist time-memory trade-offs that allow to reduce the signing time to less than the signing time of an XMSS scheme with tree height h / d. These trade-offs prevent certain values from being recomputed several times by keeping a state and distribute all computations over all signature generations. Details can be found in [Huelsing13a].

4.2.7. XMSS^MT Signature Verification

XMSS^MT signature verification (Algorithm 16) can be summarized as d XMSS signature verifications with small changes. First, only the message is hashed. The remaining XMSS signatures are on the root nodes of trees which have a fixed length. Second, instead of comparing the computed root node to a given value, a signature on the root is verified. Only the root node of the top tree is compared to the value in the XMSS^MT public key. XMSSMT_verify uses XMSS_rootFromSig and XMSS_rootFromSig_wo_hash. XMSSMT_verify takes as inputs an XMSS^MT signature Sig^MT, a message M and a public key PK_MT. It outputs a boolean.

Algorithm 16: XMSSMT_verify - Verify an XMSS^MT signature Sig_MT on a message M using an XMSS^MT public key PK_MT

```
idx = getIdx(Sig_MT);
SEED = getSEED(PK_MT);
ADRS = toByte(0,16);
unsigned int idx_leaf = (h / d) least significant bits of idx;
unsigned int idx_tree = (h - h / d) most significant bits of idx;
Sig' = leaf || setR(Sig_MT) || getXMSSSignature(Sig, 0);
ADRS.setLayerAddress(0);
ADRS.setTreeAddress(idx_tree);
byte[n] node = XMSS_rootFromSig(Sig', M, SEED, ADRS);
for ( j = 1; j < d; j = j + 1 ) {
   idx_leaf = (h / d) least significant bytes of idx_tree;
   idx_tree = (h - j * h / d) most significant bytes of idx_tree;
   Sig' = idx_leaf || getXMSSSignature(Sig, j);
   ADRS.setLayerAddress(j);
   ADRS.setTreeAddress(idx_tree);
   node = XMSS_rootFromSig_wo_hash(Sig', node, SEED, ADRS);
}
if ( node is equal to getRoot(PK_MT) ) {
  return true;
} else {
  return false;
}
```

4.2.8. Pseudorandom Key Generation

Like for XMSS, an implementation MAY use a cryptographically secure pseudorandom method to generate the XMSS^MT secret key from a single n-byte value. For example, the method explained below MAY be used. Other methods MAY be used, too. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used XMSS parameters.

For XMSS^MT a method similar to that for XMSS and WOTS+ can be used. The method uses a G as pseudorandom generator. During key generation a uniformly random n-byte string S_MT is sampled from a secure source of randomness. This seed S_MT is used to generate one n-byte value S for each XMSS key pair. This n-byte value can be used to compute the respective XMSS secret key using the method described in Section 4.1.11. Let S[x][y] be the seed for the x^th XMSS secret key on layer y. The seeds are computed as S[x][y] = G(G(S, y), x). The second parameter of G is the index x (resp. level y), represented as 16-byte string in the common way.

4.2.9. Free Index Handling and Partial Secret Keys

The content of Section 4.1.12 also applies to XMSS^MT.

5. Parameter Sets

This note provides a first basic set of parameter sets which are assumed to cover most relevant applicants. Parameter sets for two classical security levels are defined: 256 and 512 bits. Function output sizes are n = m = 32 and 64 bytes. Considering quantum-computer-aided attacks, these output sizes yield post-quantum security of 128 and 256 bits, respectively.

For the n = m = 32 and n = m = 64 settings, we give parameters that use SHA2-256 and SHA2-512 as defined in [FIPS180], respectively, and ChaCha20 as defined in [RFC7539]. SHA2 does not provide a keyed-mode itself. To implement a keyed hash-function, SHA2-256(toByte(0,32) || KEY || M) and SHA2-512(toByte(0,64) || KEY || M) are used. This construction is used for the functions F, H, and H_m. To implement PRF_m, HMAC-SHA2-256 and HMAC-SHA2-512 are used, respectively. The pseudorandom generator G for n=32 is implemented as ChaCha20 using SEED as key, the most significant 12 bytes of the address input as nonce and the least significant 4 bytes as counter. The output consists of the first 32 bytes of the key stream. The pseudorandom generator G for n=64 is implemented as HMAC-SHA2-512.

5.1. WOTS+ Parameters

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

Name	F	G	m	n	w	len
WOTSP_SHA2-256_M32_W16	SHA-2	ChaCha20	32	32	16	67
WOTSP_SHA2-512_M64_W16	SHA-2	SHA-2	64	64	16	131

Table 1

The implementation of the single functions is done as described above. XDR formats for WOTS+ are listed in Appendix A.

5.2. XMSS Parameters

To fully describe an XMSS signature method, the parameters m, n, w, and h, as well as the functions F, H, H_m, PRF_m, and G MUST be specified. This section defines different XMSS signature systems, each of which is identified by a name. We define parameter sets that implement the functions using SHA2 and ChaCha20 for n = 32 and only SHA2 for n=64 as described above.

Name	m	n	w	len	h
XMSS_SHA2-256_M32_W16_H10	32	32	16	67	10
XMSS_SHA2-256_M32_W16_H16	32	32	16	67	16
XMSS_SHA2-256_M32_W16_H20	32	32	16	67	20
XMSS_SHA2-512_M64_W16_H10	64	64	16	131	10
XMSS_SHA2-512_M64_W16_H16	64	64	16	131	16
XMSS_SHA2-512_M64_W16_H20	64	64	16	131	20

Table 2

The XDR formats for XMSS are listed in Appendix B.

5.3. XMSS^MT Parameters

To fully describe an XMSS^MT signature method, the parameters m, n, w, h, and d, as well as the functions F, H, H_m, PRF_m, and G MUST be specified. This section defines several XMSS^MT signature systems, each of which is identified by a name. We define parameter sets that

implement the functions using SHA2 and ChaCha20 for n = 32 and only SHA2 for n=64 as described above.

Name	m	n	w	len	h	d
XMSSMT_SHA2-256_M32_W16_H20_D2	32	32	16	67	20	2
XMSSMT_SHA2-256_M32_W16_H20_D4	32	32	16	67	20	4
XMSSMT_SHA2-256_M32_W16_H40_D2	32	32	16	67	40	2
XMSSMT_SHA2-256_M32_W16_H40_D4	32	32	16	67	40	4
XMSSMT_SHA2-256_M32_W16_H40_D8	32	32	16	67	40	8
XMSSMT_SHA2-256_M32_W16_H60_D3	32	32	16	67	60	3
XMSSMT_SHA2-256_M32_W16_H60_D6	32	32	16	67	60	6
XMSSMT_SHA2-256_M32_W16_H60_D12	32	32	16	67	60	12
XMSSMT_SHA2-512_M64_W16_H20_D2	64	64	16	131	20	2
XMSSMT_SHA2-512_M64_W16_H20_D4	64	64	16	131	20	4
XMSSMT_SHA2-512_M64_W16_H40_D2	64	64	16	131	40	2
XMSSMT_SHA2-512_M64_W16_H40_D4	64	64	16	131	40	4
XMSSMT_SHA2-512_M64_W16_H40_D8	64	64	16	131	40	8
XMSSMT_SHA2-512_M64_W16_H60_D3	64	64	16	131	60	3
XMSSMT_SHA2-512_M64_W16_H60_D6	64	64	16	131	60	6
XMSSMT_SHA2-512_M64_W16_H60_D12	64	64	16	131	60	12

Table 3

XDR formats for XMSS^MT are listed in Appendix C.

6. Rationale

The goal of this note is to describe the WOTS+, XMSS and XMSS^MT algorithms following the scientific literature. Other signature methods are out of scope and may be an interesting follow-on work.

The description is done in a modular way that allows to base a description of stateless hash-based signature algorithms like SPHINCS [BHH15] on it.

The draft slightly deviates from the scientific literature using a tweak that prevents multi-target attacks against the underlying hash-function. The security assumptions for this tweak are discussed in Section 8. The main difference to literature is that security now relies either on the random oracle model or some other seemingly natural heuristic assumptions.

We suggest the value w = 16 for the Winternitz parameter. No bigger values are included since the decrease in signature size then becomes less significant. Furthermore, the value w = 16 considerably simplifies the implementations of some of the algorithms. Please note that we do allow w = 4, but limit the specified parameter sets to w = 16 for efficiency reasons.

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

The enumeration values used in this note are palindromes, which have the same byte representation in either host order or network order. This fact allows an implementation to omit the conversion between byte order for those enumerations. Note however that the idx field used in XMSS and XMSS^MT signatures and secret keys must be properly converted to and from network byte order; this is the only field that requires such conversion. There are 2^32 XDR enumeration values, 2^16 of which are palindromes, which is adequate for the foreseeable future. If there is a need for more assignments, non-palindromes can be assigned.

7. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create three registries: one for WOTS+ signatures as defined in Section 3, one for XMSS signatures and one for XMSS^MT signatures; the latter two being defined in Section 4. For the sake of clarity and convenience, the first sets of WOTS+, XMSS, and XMSS^MT parameter sets are defined in Section 5. Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient details to make interoperability between independent implementations possible. Each entry in the registry contains the following elements:

a short name, such as "XMSS_SHA2-512_M64_W16_H20",

a positive number, and

a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

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

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

The WOTS+ registry is as follows.

Name	Reference	Numeric Identifier
WOTSP_SHA2-256_M32_W16	Section 5.1	0x01000001
WOTSP_SHA2-512_M64_W16	Section 5.1	0x02000002

Table 4

The XMSS registry is as follows.

Name	Reference	Numeric Identifier
XMSS_SHA2-256_M32_W16_H10	Section 5.2	0x01000001
XMSS_SHA2-256_M32_W16_H16	Section 5.2	0x02000002
XMSS_SHA2-256_M32_W16_H20	Section 5.2	0x03000003
XMSS_SHA2-512_M64_W16_H10	Section 5.2	0x04000004
XMSS_SHA2-512_M64_W16_H16	Section 5.2	0x05000005
XMSS_SHA2-512_M64_W16_H20	Section 5.2	0x06000006

Table 5

The XMSS^MT registry is as follows.

Name	Reference	Numeric Identifier
XMSSMT_SHA2-256_M32_W16_H20_D2	Section 5.3	0x01000001
XMSSMT_SHA2-256_M32_W16_H20_D4	Section 5.3	0x02000002
XMSSMT_SHA2-256_M32_W16_H40_D2	Section 5.3	0x03000003
XMSSMT_SHA2-256_M32_W16_H40_D4	Section 5.3	0x04000004
XMSSMT_SHA2-256_M32_W16_H40_D8	Section 5.3	0x05000005
XMSSMT_SHA2-256_M32_W16_H60_D3	Section 5.3	0x06000006
XMSSMT_SHA2-256_M32_W16_H60_D6	Section 5.3	0x07000007
XMSSMT_SHA2-256_M32_W16_H60_D12	Section 5.3	0x08000008
XMSSMT_SHA2-512_M64_W16_H20_D2	Section 5.3	0x09000009
XMSSMT_SHA2-512_M64_W16_H20_D4	Section 5.3	0x0a00000a
XMSSMT_SHA2-512_M64_W16_H40_D2	Section 5.3	0x0b00000b
XMSSMT_SHA2-512_M64_W16_H40_D4	Section 5.3	0x0c00000c
XMSSMT_SHA2-512_M64_W16_H40_D8	Section 5.3	0x0d00000d
XMSSMT_SHA2-512_M64_W16_H60_D3	Section 5.3	0x0e00000e
XMSSMT_SHA2-512_M64_W16_H60_D6	Section 5.3	0x0f00000f
XMSSMT_SHA2-512_M64_W16_H60_D12	Section 5.3	0x01010101

Table 6

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

8. Security Considerations

A signature system is considered secure if it prevents an attacker from forging a valid signature. More specifically, consider a setting in which an attacker gets a public key and can learn signatures on arbitrary messages of his choice. A signature system

is secure if, even in this setting, the attacker can not produce a message signature pair of his choosing such that the verification algorithm accepts.

Preventing an attacker from mounting an attack means that the attack is computationally too expensive to be carried out. There exist various estimates when a computation is too expensive to be done. For that reason, this note only describes how expensive it is for an attacker to generate a forgery. Parameters are accompanied by a bit security value. The meaning of bit security is as follows. A parameter set grants b bits of security if the best attack takes at least 2^(b-1) bit operations to achieve a success probability of 1/2. Hence, to mount a successful attack, an attacker needs to perform 2^b bit operations on average. How the given values for bit security were estimated is described below.

8.1. Security Proofs

There exist formal security proofs for schemes very similar to those described here in the literature [Huelsing13a]. These proofs show that an attacker has to break at least one out of certain security properties of the used hash functions and PRFs to forge a signature. The proofs in [Huelsing13a] do not consider the initial message compression and the extended randomized hashing used here. For the original schemes, these proofs show that an attacker has to break certain minimal security properties. In particular, it is not sufficient to break the collision resistance of the hash functions to generate a forgery.

It is folklore that one can securely combine a secure signature scheme for fixed length messages with an initial message digest. It is easy to prove that an attacker either must break the security of the fixed-input-length signature scheme or the collision resistance of the used hash function. The descriptions of XMSS and XMSS^MT in this note use a known trick to prevent the applicability of collision attacks. Namely, the schemes use a randomized message hash. For technical reasons, it is not possible to formally prove in the standard model that the resulting scheme is secure if the hash function is not collision-resistant but fulfills some weaker security properties. However, in the random oracle model such a proof is trivial.

While the basic randomized hashing used in the original descriptions of the schemes allows to prove that it is not enough for an adversary to break the collision resistance of the underlying hash function. However, it turns out that an attacker could launch a multi-target second-preimage attack. The (simplified) reason is that the adversary learns in the order of 2^h hash function input-output pairs

and it suffices to find a second-preimage for one out of those. Hence, an attacker can do a brute force search until he finds an input that matches one of the given outputs.

The extended randomized hashing used here makes the hash function calls position dependent. Hence, the above attack does not work anymore because each hash function evaluation during an attack can only target one output value. This can also be shown formally.

The given bit security values were estimated based on the complexity of the best known generic attacks against the required security properties of the used hash functions and PRFs.

8.2. Security Assumptions

The security assumptions made to argue for the security of the described schemes are minimal. Any signature algorithm that allows arbitrary size messages relies on the security of a cryptographic hash function. For the schemes described here this is already sufficient to be secure. In contrast, common signature schemes like RSA, DSA, and ECDSA additionally rely on the conjectured hardness of certain mathematical problems.

8.3. Post-Quantum Security

A post-quantum cryptosystem is a system that is secure against attackers with access to a reasonably sized quantum computer. At the time of writing this note, whether or not it is feasible to build such machine is an open conjecture. However, significant progress was made over the last few years in this regard.

In contrast to RSA, DSA, and ECDSA, the described signature systems are post-quantum-secure if they are used with an appropriate cryptographic hash function. In particular, for post-quantum security, the size of m and n must be twice the size required for classical security. This is in order to protect against quantum square root attacks due to Grover’s algorithm. It has been shown that Grover’s algorithm is optimal for finding preimages and collisions.

9. Acknowledgements

We would like to thank Scott Fluhrer, Burt Kaliski, Adam Langley, David McGrew, and Sean Parkinson for their help and comments.

10. References

10.1. Normative References

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

[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.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, May 2015.

10.2. Informative References

[BDH11] Buchmann, J., Dahmen, E., and A. Huelsing, "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions", Lecture Notes in Computer Science volume 7071. Post-Quantum Cryptography, 2011.

[BDS09] Buchmann, J., Dahmen, E., and M. Szydlo, "Hash-based Digital Signature Schemes", Book chapter Post-Quantum Cryptography, Springer, 2009.

[BHH15] Bernstein, D., Hopwood, D., Huelsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., and Z. Wilcox-O'Hearn, "SPHINCS: Practical Stateless Hash-Based Signatures", Lecture Notes in Computer Science volume 9056. Advances in Cryptology - EUROCRYPT, 2015.

[DC14] McGrew, D. and M. Curcio, "Hash-based signatures", draft-mcgrew-hash-sigs-02 (work in progress), July 2014.

[HRB13] Huelsing, A., Rausch, L., and J. Buchmann, "Optimal Parameters for XMSS^MT", Lecture Notes in Computer Science volume 8128. CD-ARES, 2013.

[Huelsing13]
Huelsing, A., "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes", Lecture Notes in Computer Science volume 7918. Progress in Cryptology - AFRICACRYPT, 2013.

[Huelsing13a]
Huelsing, A., "Practical Forward Secure Signatures using Minimal Security Assumptions", PhD thesis TU Darmstadt, 2013.

[Kaliski15]
Kaliski, B., "Panel: Shoring up the Infrastructure: A Strategy for Standardizing Hash Signatures", NIST Workshop on Cybersecurity in a Post-Quantum World, 2015.

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

Appendix A. WOTS+ XDR Formats

The WOTS+ signature and public key formats are formally defined using XDR [RFC4506] in order to provide an unambiguous, machine readable definition. Though XDR is used, these formats are simple and easy to parse without any special tools. To avoid the need to convert to and from network / host byte order, the enumeration values are all palindromes.

WOTS+ parameter sets are defined using XDR syntax as follows:

```
/* ots_algorithm_type identifies a particular
   signature algorithm */

enum ots_algorithm_type {
  wotsp_reserved         = 0x00000000,
  wotsp_sha2-256_m32_w16 = 0x01000001,
  wotsp_sha2-512_m64_w16 = 0x02000002,
};
```

WOTS+ signatures are defined using XDR syntax as follows:

```
/* Byte strings */

typedef opaque bytestring32[32];
typedef opaque bytestring64[64];

union ots_signature switch (ots_algorithm_type type) {
  case wotsp_sha2-256_m32_w16:
    bytestring32 ots_sig_m32_len67[67];

  case wotsp_sha2-512_m64_w16:
    bytestring64 ots_sig_m64_len18[131];

  default:
    void;   /* error condition */
};
```

WOTS+ public keys are defined using XDR syntax as follows:

```
union ots_pubkey switch (ots_algorithm_type type) {
  case wotsp_sha2-256_m32_w16:
    bytestring32 ots_pubk_m32_len67[67];

  case wotsp_sha2-512_m64_w16:
    bytestring64 ots_pubk_m64_len18[131];

  default:
    void;   /* error condition */
};
```

Appendix B. XMSS XDR Formats

XMSS parameter sets are defined using XDR syntax as follows:

```
/* Byte strings */

typedef opaque bytestring4[4];

/* Definition of parameter sets */

enum xmss_algorithm_type {
  xmss_reserved             = 0x00000000,

  /* 256 bit classical security, 128 bit post-quantum security */

  xmss_sha2-256_m32_w16_h10 = 0x01000001,
  xmss_sha2-256_m32_w16_h16 = 0x02000002,
  xmss_sha2-256_m32_w16_h20 = 0x03000003,

  /* 512 bit classical security, 256 bit post-quantum security */

  xmss_sha2-512_m64_w16_h10 = 0x04000004,
  xmss_sha2-512_m64_w16_h16 = 0x05000005,
  xmss_sha2-512_m64_w16_h20 = 0x06000006,
};
```

XMSS signatures are defined using XDR syntax as follows:

```
/* Authentication path types */

union xmss_path switch (xmss_algorithm_type type) {
  case xmss_sha2-256_m32_w16_h10:
    bytestring32 path_n32_t10[10];

  case xmss_sha2-256_m32_w16_h16:
    bytestring32 path_n32_t16[16];

  case xmss_sha2-256_m32_w16_h20:
    bytestring32 path_n32_t20[20];

  case xmss_sha2-512_m64_w16_h10:
    bytestring64 path_n64_t10[10];

  case xmss_sha2-512_m64_w16_h16:
    bytestring64 path_n64_t16[16];

  case xmss_sha2-512_m64_w16_h20:
```

```
      bytestring64 path_n64_t20[20];

  default:
    void;     /* error condition */
};

/* Types for XMSS random strings */

union random_string_xmss switch (xmss_algorithm_type type) {
  case xmss_sha2-256_m32_w16_h10:
  case xmss_sha2-256_m32_w16_h16:
  case xmss_sha2-256_m32_w16_h20:
    bytestring32 rand_m32;

  case xmss_sha2-512_m64_w16_h10:
  case xmss_sha2-512_m64_w16_h16:
  case xmss_sha2-512_m64_w16_h20:
    bytestring64 rand_m64;

  default:
    void;     /* error condition */
};

/* Corresponding WOTS+ type for given XMSS type */

union xmss_ots_signature switch (xmss_algorithm_type type) {
  case xmss_sha2-256_m32_w16_h10:
  case xmss_sha2-256_m32_w16_h16:
  case xmss_sha2-256_m32_w16_h20:
    wotsp_sha2-256_m32_w16;

  case xmss_sha2-512_m64_w16_h10:
  case xmss_sha2-512_m64_w16_h16:
  case xmss_sha2-512_m64_w16_h20:
    wotsp_sha2-512_m64_w16;

  default:
    void;     /* error condition */
};

/* XMSS signature structure */

struct xmss_signature {
  /* WOTS+ key pair index */
  bytestring4 idx_sig;
  /* Random string for randomized hashing */
  random_string_xmss rand_string;
  /* WOTS+ signature */
```

```
        xmss_ots_signature sig_ots;
        /* authentication path */
        xmss_path nodes;
      };
```

XMSS public keys are defined using XDR syntax as follows:

```
/* Types for bitmask seed */

union seed switch (xmss_algorithm_type type) {
  case xmss_sha2-256_m32_w16_h10:
  case xmss_sha2-256_m32_w16_h16:
  case xmss_sha2-256_m32_w16_h20:
    bytestring32 seed_n32;

  case xmss_sha2-512_m64_w16_h10:
  case xmss_sha2-512_m64_w16_h16:
  case xmss_sha2-512_m64_w16_h20:
    bytestring64 seed_n64;

  default:
    void;     /* error condition */
};

/* Types for XMSS root node */

union xmss_root switch (xmss_algorithm_type type) {
  case xmss_sha2-256_m32_w16_h10:
  case xmss_sha2-256_m32_w16_h16:
  case xmss_sha2-256_m32_w16_h20:
    bytestring32 root_n32;

  case xmss_sha2-512_m64_w16_h10:
  case xmss_sha2-512_m64_w16_h16:
  case xmss_sha2-512_m64_w16_h20:
    bytestring64 root_n64;

  default:
    void;     /* error condition */
};

/* XMSS public key structure */

struct xmss_public_key {
  xmss_root root;  /* Root node */
  seed SEED;  /* Seed for bitmasks */
};
```

Appendix C. XMSS^MT XDR Formats

XMSS^MT parameter sets are defined using XDR syntax as follows:

```
/* Byte strings */

typedef opaque bytestring3[3];
typedef opaque bytestring5[5];
typedef opaque bytestring8[8];

/* Definition of parameter sets */

enum xmssmt_algorithm_type {
  xmssmt_reserved                 = 0x00000000,

  /* 256 bit classical security, 128 bit post-quantum security */

  xmssmt_sha2-256_m32_w16_h20_d2  = 0x01000001,
  xmssmt_sha2-256_m32_w16_h20_d4  = 0x02000002,
  xmssmt_sha2-256_m32_w16_h40_d2  = 0x03000003,
  xmssmt_sha2-256_m32_w16_h40_d4  = 0x04000004,
  xmssmt_sha2-256_m32_w16_h40_d8  = 0x05000005,
  xmssmt_sha2-256_m32_w16_h60_d3  = 0x06000006,
  xmssmt_sha2-256_m32_w16_h60_d6  = 0x07000007,
  xmssmt_sha2-256_m32_w16_h60_d12 = 0x08000008,

  /* 512 bit classical security, 256 bit post-quantum security */

  xmssmt_sha2-512_m64_w16_h20_d2  = 0x09000009,
  xmssmt_sha2-512_m64_w16_h20_d4  = 0x0a00000a,
  xmssmt_sha2-512_m64_w16_h40_d2  = 0x0b00000b,
  xmssmt_sha2-512_m64_w16_h40_d4  = 0x0c00000c,
  xmssmt_sha2-512_m64_w16_h40_d8  = 0x0d00000d,
  xmssmt_sha2-512_m64_w16_h60_d3  = 0x0e00000e,
  xmssmt_sha2-512_m64_w16_h60_d6  = 0x0f00000f,
  xmssmt_sha2-512_m64_w16_h60_d12 = 0x01010101,
};
```

XMSS^MT signatures are defined using XDR syntax as follows:

```
/* Type for XMSS^MT key pair index */
/* Depends solely on h */

union idx_sig_xmssmt switch (xmss_algorithm_type type) {
  case xmssmt_sha2-256_m32_w16_h20_d2:
```

```
        case xmssmt_sha2-256_m32_w16_h20_d4:
        case xmssmt_sha2-512_m64_w16_h20_d2:
        case xmssmt_sha2-512_m64_w16_h20_d4:
          bytestring3 idx3;

        case xmssmt_sha2-256_m32_w16_h40_d2:
        case xmssmt_sha2-256_m32_w16_h40_d4:
        case xmssmt_sha2-256_m32_w16_h40_d8:
        case xmssmt_sha2-512_m64_w16_h40_d2:
        case xmssmt_sha2-512_m64_w16_h40_d4:
        case xmssmt_sha2-512_m64_w16_h40_d8:
          bytestring5 idx5;

        case xmssmt_sha2-256_m32_w16_h60_d3:
        case xmssmt_sha2-256_m32_w16_h60_d6:
        case xmssmt_sha2-256_m32_w16_h60_d12:
        case xmssmt_sha2-512_m64_w16_h60_d3:
        case xmssmt_sha2-512_m64_w16_h60_d6:
        case xmssmt_sha2-512_m64_w16_h60_d12:
          bytestring8 idx8;

        default:
          void;     /* error condition */
      };

      union random_string_xmssmt switch (xmssmt_algorithm_type type) {
        case xmssmt_sha2-256_m32_w16_h20_d2:
        case xmssmt_sha2-256_m32_w16_h20_d4:
        case xmssmt_sha2-256_m32_w16_h40_d2:
        case xmssmt_sha2-256_m32_w16_h40_d4:
        case xmssmt_sha2-256_m32_w16_h40_d8:
        case xmssmt_sha2-256_m32_w16_h60_d3:
        case xmssmt_sha2-256_m32_w16_h60_d6:
        case xmssmt_sha2-256_m32_w16_h60_d12:
          bytestring32 rand_m32;

        case xmssmt_sha2-512_m64_w16_h20_d2:
        case xmssmt_sha2-512_m64_w16_h20_d4:
        case xmssmt_sha2-512_m64_w16_h40_d2:
        case xmssmt_sha2-512_m64_w16_h40_d4:
        case xmssmt_sha2-512_m64_w16_h40_d8:
        case xmssmt_sha2-512_m64_w16_h60_d3:
        case xmssmt_sha2-512_m64_w16_h60_d6:
        case xmssmt_sha2-512_m64_w16_h60_d12:
          bytestring64 rand_m64;

        default:
          void;     /* error condition */
```

```
};

struct xmss_reduced_bottom {
  xmss_ots_signature sig_ots; /* WOTS+ signature */
  xmss_path nodes;            /* authentication path */
};

/* Type for individual reduced XMSS signatures on higher layers */

union xmss_reduced_others (xmss_algorithm_type type) {
  case xmssmt_sha2-256_m32_w16_h20_d2:
  case xmssmt_sha2-256_m32_w16_h20_d4:
    bytestring32 xmss_reduced_n32_t87[87];

  case xmssmt_sha2-256_m32_w16_h40_d2:
  case xmssmt_sha2-256_m32_w16_h40_d4:
  case xmssmt_sha2-256_m32_w16_h40_d8:
    bytestring32 xmss_reduced_n32_t107[107];

  case xmssmt_sha2-256_m32_w16_h60_d3:
  case xmssmt_sha2-256_m32_w16_h60_d6:
  case xmssmt_sha2-256_m32_w16_h60_d12:
    bytestring32 xmss_reduced_n32_t127[127];

  case xmssmt_sha2-512_m64_w16_h20_d2:
  case xmssmt_sha2-512_m64_w16_h20_d4:
    bytestring64 xmss_reduced_n64_t151[151];

  case xmssmt_sha2-512_m64_w16_h40_d2:
  case xmssmt_sha2-512_m64_w16_h40_d4:
  case xmssmt_sha2-512_m64_w16_h40_d8:
    bytestring64 xmss_reduced_n64_t171[171];

  case xmssmt_sha2-512_m64_w16_h60_d3:
  case xmssmt_sha2-512_m64_w16_h60_d6:
  case xmssmt_sha2-512_m64_w16_h60_d12:
    bytestring64 xmss_reduced_n64_t191[191];

  default:
    void;     /* error condition */
};

/* xmss_reduced_array depends on d */

union xmss_reduced_array (xmss_algorithm_type type) {
  case xmssmt_sha2-256_m32_w16_h20_d2:
  case xmssmt_sha2-512_m64_w16_h20_d2:
  case xmssmt_sha2-256_m32_w16_h40_d2:
```

```
        case xmssmt_sha2-512_m64_w16_h40_d2:
          xmss_reduced_others xmss_red_arr_d2[1];

        case xmssmt_sha2-256_m32_w16_h60_d3:
        case xmssmt_sha2-512_m64_w16_h60_d3:
          xmss_reduced_others xmss_red_arr_d3[2];

        case xmssmt_sha2-256_m32_w16_h20_d4:
        case xmssmt_sha2-512_m64_w16_h20_d4:
        case xmssmt_sha2-256_m32_w16_h40_d4:
        case xmssmt_sha2-512_m64_w16_h40_d4:
          xmss_reduced_others xmss_red_arr_d4[3];

        case xmssmt_sha2-256_m32_w16_h60_d6:
        case xmssmt_sha2-512_m64_w16_h60_d6:
          xmss_reduced_others xmss_red_arr_d6[5];

        case xmssmt_sha2-256_m32_w16_h40_d8:
        case xmssmt_sha2-512_m64_w16_h40_d8:
          xmss_reduced_others xmss_red_arr_d8[7];

        case xmssmt_sha2-256_m32_w16_h60_d12:
        case xmssmt_sha2-512_m64_w16_h60_d12:
          xmss_reduced_others xmss_red_arr_d12[11];

        default:
          void;     /* error condition */
      };

      /* XMSS^MT signature structure */

      struct xmssmt_signature {
        /* WOTS+ key pair index */
        idx_sig_xmssmt idx_sig;
        /* Random string for randomized hashing */
        random_string_xmssmt randomness;
        /* Reduced bottom layer XMSS signature */
        xmss_reduced_bottom;
        /* Array of reduced XMSS signatures with message length n */
        xmss_reduced_array;
      };
```

XMSS^MT public keys are defined using XDR syntax as follows:

```
      /* Types for bitmask seed */
```

```
union seed switch (xmssmt_algorithm_type type) {
  case xmssmt_sha2-256_m32_w16_h20_d2:
  case xmssmt_sha2-256_m32_w16_h40_d4:
  case xmssmt_sha2-256_m32_w16_h60_d6:
  case xmssmt_sha2-256_m32_w16_h20_d4:
  case xmssmt_sha2-256_m32_w16_h40_d8:
  case xmssmt_sha2-256_m32_w16_h60_d12:
  case xmssmt_sha2-256_m32_w16_h40_d2:
  case xmssmt_sha2-256_m32_w16_h60_d3:
    bytestring32 seed_n32;

  case xmssmt_sha2-512_m64_w16_h20_d2:
  case xmssmt_sha2-512_m64_w16_h40_d4:
  case xmssmt_sha2-512_m64_w16_h60_d6:
  case xmssmt_sha2-512_m64_w16_h20_d4:
  case xmssmt_sha2-512_m64_w16_h40_d8:
  case xmssmt_sha2-512_m64_w16_h60_d12:
  case xmssmt_sha2-512_m64_w16_h40_d2:
  case xmssmt_sha2-512_m64_w16_h60_d3:
    bytestring64 seed_n64;

  default:
    void;     /* error condition */
};

/* Types for XMSS^MT root node */

union xmssmt_root switch (xmssmt_algorithm_type type) {
  case xmssmt_sha2-256_m32_w16_h20_d2:
  case xmssmt_sha2-256_m32_w16_h20_d4:
  case xmssmt_sha2-256_m32_w16_h40_d2:
  case xmssmt_sha2-256_m32_w16_h40_d4:
  case xmssmt_sha2-256_m32_w16_h40_d8:
  case xmssmt_sha2-256_m32_w16_h60_d3:
  case xmssmt_sha2-256_m32_w16_h60_d6:
  case xmssmt_sha2-256_m32_w16_h60_d12:
    bytestring32 root_n32;

  case xmssmt_sha2-512_m64_w16_h20_d2:
  case xmssmt_sha2-512_m64_w16_h20_d4:
  case xmssmt_sha2-512_m64_w16_h40_d2:
  case xmssmt_sha2-512_m64_w16_h40_d4:
  case xmssmt_sha2-512_m64_w16_h40_d8:
  case xmssmt_sha2-512_m64_w16_h60_d3:
  case xmssmt_sha2-512_m64_w16_h60_d6:
  case xmssmt_sha2-512_m64_w16_h60_d12:
    bytestring64 root_n64;
```

```
  default:
    void;     /* error condition */
};

/* XMSS^MT public key structure */

struct xmssmt_public_key {
  xmssmt_root root;  /* Root node */
  seed SEED;  /* Seed for bitmasks */
};
```

Appendix D. Changed since draft-irtf-cfrg-xmss-hash-based-signatures-00

Added keying of hash functions with pseudorandomly generated keys for all hashes but the message hash. A Hash Function Address Scheme is introduced that for. For further information please take a look at sections Section 2.5 and Section 8.

Replaced bitmasks by pseudorandomly generated values.

Removed zero bitmasks, as we now only store a small seed (n bytes) for bitmask generation, which is pretty small compared to the bitmask solution before.

Removed w = 8 to reduce huge number of parameter sets. Simplified algorithms (like base_w) as we don’t need to support the w = 8 case now.

Removed w = 4 from the suggested parameter sets.

Changed ’l’ to ’len, ’l_1’ to ’len_1’ and ’l_2’ to ’len_2’ to avoid confusion between the characters ’l’ and ’1’.

Changed appearances of "is equal" or "mod" to % operator for better readability.

Removed the OID in the XMSS and XMSS^MT signatures. This was redundant, since the OID is already part of the public key in both cases.

Replaced SHA-3 by SHA-2 in light of more widespread usage and faster implementations for SHA-2.

Fixed the log notation in the Schemes section.

Removed AES-based parameter sets.

Adapted XDR according to the changes.

Removed definitions like max() from notation, when no longer needed in this document.

Authors' Addresses

Andreas Huelsing
TU Eindhoven
P.O. Box 513
Eindhoven 5600 MB
The Netherlands

Email: a.t.huelsing@tue.nl

Denis Butin
TU Darmstadt
Hochschulstrasse 10
Darmstadt 64289
Germany

Email: dbutin@cdc.informatik.tu-darmstadt.de

Stefan-Lukas Gazdag
genua GmbH
Domagkstrasse 7
Kirchheim bei Muenchen 85551
Germany

Email: stefan-lukas_gazdag@genua.eu

Aziz Mohaisen
Verisign Labs
12061 Bluemont Way
Reston, VA 20190

Phone: +1 703 948-3200
Email: amohaisen@verisign.com