Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: September 24, 2015

A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua mbH
A. Mohaisen
Verisign Labs
March 23, 2015

XMSS: Extended Hash-Based Signatures
draft-huelsing-cfrg-hash-sig-xmss-00

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.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 24, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 3
  1.1. Conventions Used In This Document 5
2. Notation 5
  2.1. Data Types 5
  2.2. Operators 5
  2.3. Functions 6
  2.4. Strings of Base-w Numbers 6
  2.5. Member Functions 7
3. Primitives 8
  3.1. WOTS+ One-Time Signatures 8
    3.1.1. WOTS+ Parameters 8
      3.1.1.1. WOTS+ Hashing Functions 9
    3.1.2. WOTS+ Chaining Function 9
    3.1.3. WOTS+ Private Key 9
    3.1.4. WOTS+ Public Key 10
    3.1.5. WOTS+ Signature Generation 10
    3.1.6. WOTS+ Signature Verification 11
    3.1.7. Pseudorandom Key Generation 12
4. Schemes 12
  4.1. XMSS: eXtended Merkle Signature Scheme 13
    4.1.1. XMSS Parameters 13
    4.1.2. XMSS Hash Functions 14
    4.1.3. XMSS Private Key 14
    4.1.4. L-Trees 14
    4.1.5. TreeHash 15
    4.1.6. XMSS Public Key 15
    4.1.7. XMSS Signature 16
    4.1.8. XMSS Signature Generation 18
    4.1.9. XMSS Signature Verification 19
    4.1.10. Pseudorandom Key Generation 20
    4.1.11. Free Index Handling and Partial Secret Keys 21

4.2. XMSS^MT: Multi-Tree XMSS 21
4.2.1. XMSS^MT Parameters 21
4.2.2. XMSS Algorithms Without Message Hash 22
4.2.3. XMSS^MT Private Key 22
4.2.4. XMSS^MT Public Key 22
4.2.5. XMSS^MT Signature 23
4.2.6. XMSS^MT Signature Generation 24
4.2.7. XMSS^MT Signature Verification 25
4.2.8. Pseudorandom Key Generation 26
4.2.9. Free Index Handling and Partial Secret Keys 26
5. Parameter Sets 27
5.1. Zero Bitmasks 27
5.2. WOTS+ Parameters 28
5.3. XMSS Parameters 29
5.3.1. XMSS Parameters 29
5.3.1.1. XMSS Parameters with AES and SHA3 29
5.3.1.2. XMSS Parameters with SHA3 30
5.3.2. XMSS Parameters With Empty Bitmasks 31
5.4. XMSS^MT Parameters 32
5.4.1. XMSS^MT Parameters 32
5.4.1.1. XMSS^MT Parameters with AES and SHA3 32
5.4.1.2. XMSS^MT Parameters with SHA3 33
5.4.2. XMSS^MT Parameters With Empty Bitmasks 35
6. Rationale 38
7. IANA Considerations 38
8. Security Considerations 49
8.1. Security Proofs 50
8.2. Security Assumptions 51
8.3. Post-Quantum Security 51
9. Acknowledgements 51
10. References 51
10.1. Normative References 51
10.2. Informative References 52
Appendix A. WOTS+ XDR Formats 53
Appendix B. XMSS XDR Formats 55
Appendix C. XMSS^MT XDR Formats 65
Authors' Addresses 87

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 us to use 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 a 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 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, SPHINCS, a stateless hash-based signature scheme was introduced [BHH15], with the intent of being easier to deploy in current applications. A reasonable next step toward introducing hash signatures would seem 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 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 note describes a single-tree and a multi-tree variant of the eXtended Merkle Signature Scheme (XMSS) [BDH11]. 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 note is structured as follows. Notation is introduced in Section 2. Section 3 describes the WOTS+ signature system. Many time signature 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.

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 ith 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 ith byte, where X[0] is the leftmost byte. In addition, bytes(X, i, j) with i < j denotes the range of bytes from the ith to the jth byte in X, inclusively. For example, if X = 0x01020304, then X[0] is 0x01 and bytes(X, 1, 2) is 0x0203.

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 base-2 logarithm of x.

If x, y, and z are real numbers, then we define the functions max(x, y) and max(x, y, z) which return the maximum value of the set {x, y} and {x, y, z}, respectively.

2.4. 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 a m-byte string, w is a member of the set {4, 8, 16}, then base_w(X, w) outputs a length ceil(8m/lg(w)) array of integers between 0 and w - 1. In case lg(w)

does not divide 8 * m without a remainder, X is virtually padded with a sufficient amount of zero bits.

Algorithm 1: base_w(X, w)

```
i_byte = 0;
i_bit = 0;
for ( i=0; i < ceil(8m/lg(w)); i++ ){
  if( i_bit + lg(w) <= 8 ){
    basew[i] = ((X[i_byte] << i_bit) >> (8-lg(w))) AND (w-1);
    i_bit += lg(w);
    if ( i_bit == 8 ){
      i_bit = 0;
      i_byte = i_byte + 1;
    }
  } else {
    basew[i] = ((X[i_byte] << i_bit) >> (8-lg(w))) AND (w-1);
    i_byte = i_byte + 1;
    if ( i_byte < m ){
      basew[i] += (X[i_byte] >> (8-(i_bit + lg(w)-8))) AND (w-1);
      i_bit = i_bit + lg(w)-8;
    }
  }
}
return basew;
```

For example, if X is 0x1234, then base_w(X, 8) returns the array {0, 4, 4, 3, 2, 0}.

```
                    X (represented as bits)
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

                    X (padded with zeros)
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0| 0| 0|
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

              X (represented as base-w numbers)
      +--------+--------+--------+--------+--------+--------+
      |    0   |    4   |    4   |    3   |    2   |    0   |
      +--------+--------+--------+--------+--------+--------+
```

2.5. 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, as defined in [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, 8, 16}

The parameters are used to compute values l, l_1 and l_2:

l : the number of n-byte string elements in a WOTS+ secret key, public key, and signature. It is computed as l = l_1 + l_2, with l_1 = ceil(8m/lg(w)) and l_2 = floor(lg(l_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 often the length of a message digest. The parameter w can be chosen from the set {4,8,16}. 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, 8 and 16 since these values yield optimal trade-offs.

3.1.1.1. WOTS+ Hashing Functions

The WOTS+ algorithm uses a cryptographic hash function F. F accepts and returns byte strings of length n. Security requirements on F 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 a vector of n-byte strings called bitmasks. In each iteration, a bitmask is first XORed to an intermediate result before it is processed by F. In the following, bm is an array of at least w-2 n-byte strings (that contains the bitmasks). The chaining function takes as input an n-byte string X, a start index i, a number of steps s, and the bitmasks bm. The chaining function returns as output the value obtained by iterating F for s times on input X, using the bitmasks from bm starting at index i.

Algorithm 2: Chaining Function

```
if s is equal to 0 then
  return X;
end
if (i+s) > w-1 then
  return NULL;
end
byte[n] tmp = chain(X, i, s-1, bm);
tmp = F(tmp XOR bm[i+s-1]);
return tmp;
```

3.1.3. WOTS+ Private Key

The private key in WOTS+, denoted by sk, is a length l 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 < l; 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 l hash chains of length w. The l 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 l array of n-byte strings. To compute the hash chain, the chaining function (Algorithm 2) is used. The bitmasks have to be provided by the calling algorithm. The same bitmasks are used for all chains. 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 < l; i = i + 1 ) {
  pk[i] = chain(sk[i], 0, w-1, bm);
}
return pk;
```

3.1.5. WOTS+ Signature Generation

A WOTS+ signature is a length l array of n-byte strings. The WOTS+ signature is generated by mapping a message to l 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 2.4. Next, a checksum is computed and appended to the transformed message as base w numbers using base_w(). 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 < l_1; i = i + 1 ) {
  csum = csum + w - 1 - msg[i]
}
// Convert csum to base w
msg = msg || base_w(csum, w);
for ( i = 0; i < l; i = i + 1 ) {
  sig[i] = chain(sk[i], 0, msg[i], bm)
}
return sig
```

The data format for a signature is given below.

WOTS+ Signature

```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|           sig_ots[0]            |    n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|          sig_ots[l-1]           |    n bytes
|                                 |
+---------------------------------+
```

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 < l_1; i = i + 1 ) {
  csum = csum + w - 1 - msg[i]
}
// Convert csum to base w
msg = msg || base_w(csum, w);
for ( i = 0; i < l; i = i + 1 ) {
  tmp_pk[i] = chain(sig[i], msg[i], w-1-msg[i], bm)
}
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] uses a pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. The secret key elements are computed as sk[i] = G(S,i) whenever needed. The second parameter of G is i, represented as n-byte string in the common way. To implement G, an implementation MAY use the hash function F in PRF mode. When WOTS+ is used within XMSS or XMSS^MT, an implementation SHOULD use PRF_m, taking the first n bytes from the output.

4. Schemes

In this section, the extended Merkle signature scheme (XMSS) is described using WOTS+. XMSS comes in two flavours: 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 four cryptographic components: WOTS+ as OTS method, two additional cryptographic hash functions H and H_m, and a pseudorandom function PRF_m. 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 hash function H to the result. The value corresponding to the root of the XMSS tree forms the XMSS public key together with the bitmasks.

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. 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. XMSS uses num_bm = max{2 * (h + ceil(lg(l))), w - 2} bitmasks produced during key generation.

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 three more functions:

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

A cryptographic hash function H_m. H_m accepts 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.

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 ith WOTS+ secret key of SK.

4.1.4. 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 7) takes as input a WOTS+ public key pk and compresses it to a single n-byte value pk[0]. The algorithm uses the first (2 * ceil(log(l))) of the num_bm n-byte bitmasks bm.

Algorithm 7: ltree

```
unsigned int l' = l
unsigned int j = 0
while ( l' > 1 ) {
  for ( i = 0; i < floor(l' / 2); i = i + 1 ) {
    pk[i] = H((pk[2i] XOR bm[j]) || (pk[2i + 1] XOR bm[j + 1]))
  }
  if ( l' is equal to 1 % 2 ) {
    pk[floor(l' / 2) + 1] = pk[l']
  }
  l' = ceil(l' / 2)
  j = j + 2
}
return pk[0]
```

4.1.5. TreeHash

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine treeHash (Algorithm 8) accepts an XMSS secret key SK, an unsigned integer s (the start index), an unsigned integer h (the target node height) and the bitmasks bm. The treeHash algorithm returns the root node of a tree of height h with the leftmost leaf being the hash of the WOTS+ pk with index s. The treeHash algorithm uses a stack holding up to (h-1) n-byte strings, with the usual stack functions push() and pop().

Algorithm 8: treeHash

```
for ( i = 0; i < 2^h; i = i + 1 ) {
  pk = WOTS_genPK (getWOTS_SK(SK, s+i), bm)
  node = ltree(pk, bm)
  while ( Top node on Stack has same height h' as node ) {
    node = H((Stack.pop() XOR bm[2l + 2h']) ||
             (node XOR bm[2l + 2h' + 1]))
  }
  Stack.push(node)
}
return Stack.pop()
```

4.1.6. XMSS Public Key

The XMSS public key is computed as described in XMSS_genPK (Algorithm 9). The algorithm takes the num_bm n-byte bitmasks bm, 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 bitmasks bm.

Algorithm 9: XMSS_genPK - Generate an XMSS public key from an XMSS private key

```
for ( i = 0; i < num_bm; i = i + 1 ) {
   set bm[i] to a uniformly random n-byte string
}
root = treeHash(SK, 0, h, bm)
PK = root || bm
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

```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|            root node            |     n bytes
|                                 |
+---------------------------------+
|                                 |
|              bm[0]              |     n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|          bm[num_bm-1]           |     n bytes
|                                 |
+---------------------------------+
```

4.1.7. XMSS Signature

An XMSS signature is a (4 + m + (l + 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 hashing (m bytes),

a WOTS+ signature sig_ots (l * 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 xth 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 ith WOTS+ key pair, counting from zero, the jth authentication path node is

Node(j, floor(i / (2^j)) + 1) if floor(i / (2^j)) is even or

Node(j, floor(i / (2^j)) - 1) if floor(i / (2^j)) is odd.

Given an XMSS secret key SK and bitmasks bm, all nodes in a tree are determined. Their value is defined in terms of treeHash (Algorithm 8):

Node(x,y) = treeHash(SK, x * 2^y, y, bm).

The data format for a signature is given below.

XMSS Signature

```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|          index idx_sig          |    4 bytes
|                                 |
+---------------------------------+
|                                 |
|          randomness r           |    m bytes
|                                 |
+---------------------------------+
|                                 |
|     WOTS+ signature sig_ots     |    l * n bytes
|                                 |
+---------------------------------+
|                                 |
|             auth[0]             |    n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|            auth[h-1]            |    n bytes
|                                 |
+---------------------------------+
```

4.1.8. 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 10) 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 bitmasks bm. It returns the byte string containing the concatenation of the updated secret key SK and the signature Sig.

Algorithm 10: XMSS_sign - Generate an XMSS signature and update the XMSS secret key

```
idx_sig = getIdx(SK)
auth = buildAuth(SK, bm, idx_sig)
byte[m] r = PRF_m(getSK_PRF(SK), M)
byte[m] M' = H_m(r || M)
sig_ots = WOTS_sign(getWOTS_SK(SK, idx_sig), M', bm)
Sig = (idx_sig || r || sig_ots || auth)
setIdx(SK, idx_sig + 1)
return (SK || Sig)
```

4.1.9. 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, authentication path auth and bitmasks bm is used to compute an alternative root value for the tree. These first steps are done by XMSS_rootFromSig (Algorithm 11). 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 11) described below. XMSS_rootFromSig takes as inputs an XMSS signature Sig, a message M, and the bitmasks bm. XMSS_rootFromSig returns an n-byte string holding the value of the root of a tree defined by the input data.

Algorithm 11: XMSS_rootFromSig - Compute a root node using an XMSS signature, a message, and bitmasks bm

```
byte[m] M' = H_m(r || M)
pk_ots = WOTS_pkFromSig(sig_ots, M', bm)
byte[n][2] node
node[0] = ltree(pk_ots, bm)
for ( k = 1; k < h; k = k + 1 ) {
  if ( floor(i / (2^k)) % 2 is equal to 0 ) {
    node[1] = H((node[0] XOR bm[2l + 2k]) ||
                (auth[k - 1] XOR bm[2l + 2k + 1]))
  } else {
    node[1] = H((auth[k - 1] XOR bm[2l + 2k]) ||
                (node[0] XOR bm[2l + 2k + 1]))
  }
  node[0] = node[1]
}
return node[0]
```

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

Algorithm 12: 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, getBM(PK))
if ( node is equal to root in PK ) {
  return true
} else {
  return false
}
```

4.1.10. 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] uses a pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This seed S is used to generate an n-byte value S_ots for each WOTS+ key pair. This n-byte value 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 n-byte string in the common way. To implement G an implementation SHOULD use PRF_m, taking the first n bytes from the output. An advantage of this method is that a WOTS+ key can be computed using only l+1 evaluations of G when S is given.

4.1.11. 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 l_1, l_2 and l change for these layers. The parameters l_1_n, l_2_n, and l_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 SHOULD 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 xth XMSS tree on the yth 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 bitmasks. The same bitmasks are used for all XMSS tress. Algorithm 13 shows pseudocode to generate PK_MT. First, num_bm = max{ 2 * (h / d + ceil(lg(l))), 2 * (h / d + ceil(lg(l_n))), w - 2 } n-byte bitmasks bm are 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 13: XMSSMT_genPK - Generate an XMSS^MT public key from an XMSS^MT private key

```
for ( i = 0; i < num_bm; i = i + 1 ) {
   set bm[i] to a uniformly random n-byte string
}
root = treeHash(getXMSS_SK(SK_MT, 0, d - 1), 0, h / d, bm)
PK_MT = root || bm
return PK_MT
```

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

XMSS^MT Public Key

```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|            root node            |     n bytes
|                                 |
+---------------------------------+
|                                 |
|              bm[0]              |     n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|          bm[num_bm-1]           |     n bytes
|                                 |
+---------------------------------+
```

4.2.5. XMSS^MT Signature

An XMSS^MT signature Sig_MT is a byte string of length (ceil(h / 8) + m + (h + l + (d - 1) * l_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 hashing (m bytes),

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

d-1 reduced XMSS signatures with message length n ((h + l_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 l n-byte elements. The remaining reduced XMSS signatures contain a WOTS+ signature on an n-byte message and hence consist of l_n n-byte elements.

The data format for a signature is given below.

XMSS^MT signature

```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|          index idx_sig          |    ceil(h / 8) bytes
|                                 |
+---------------------------------+
|                                 |
|          randomness r           |    m bytes
|                                 |
+---------------------------------+
|                                 |
|  (reduced) XMSS signature Sig   |    (h + l) * n bytes
|        (bottom layer 0)         |
|                                 |
+---------------------------------+
|                                 |
|  (reduced) XMSS signature Sig   |    (h + l_n) * n bytes
|            (layer 1)            |
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|  (reduced) XMSS signature Sig   |    (h + l_n) * n bytes
|           (layer d-1)           |
|                                 |
+---------------------------------+
```

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 bitmasks bm, XMSSMT_sign (Algorithm 14) 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 14: XMSSMT_sign - Generate an XMSS^MT signature and update the XMSS^MT secret key

```
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)
Sig_tmp = XMSS_sign(M, SK, bm)
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, bm)
   idx_leaf = (h / d) least significant bits of idx_tree
   idx_tree = (h - j * (h / d)) most significant bytes of idx_tree
   SK = idx_leaf || SK_PRF || getXMSS_SK(SK_MT, idx_tree, j)
   Sig_tmp = XMSS_sign_wo_hash(root, SK, bm) with idx removed
   Sig_MT = Sig_MT || Sig_tmp
}
return SK_MT || Sig_MT
```

Algorithm 14 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 15) 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 15: 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)
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)
byte[n] node = XMSS_rootFromSig(Sig', M, getBm(PK_MT))
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)
   node = XMSS_rootFromSig_wo_hash(Sig', node, getBm(PK_MT))
}
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. 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 pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. 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 4.1.10. Let S[x][y] be the seed for the xth 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 n-byte string in the common way. To implement G an implementation SHOULD use PRF_m, taking the first n bytes from the output.

4.2.9. Free Index Handling and Partial Secret Keys

The content of Section 4.1.11 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 three classical security levels are defined: 128, 256 and 512 bits. Function output sizes are n = 16, 32 and 64 bytes and m = 32, 64, respectively. While m = n is used for n = 32 and n = 64, m = 32 is used for the n = 16 case. Considering quantum-computer-aided attacks, these output sizes yield post-quantum security of 64, 128 and 256 bits, respectively. The n = 16 parameter sets are included to encourage adoption in the pre-quantum era as they lead to smaller signatures and faster runtimes than other parameter sets. The n = 64 parameter sets are provided to support post-quantum scenarios.

For the n = 16 setting, this note only defines parameter sets with AES-based hash functions. The reason is that they benefit from hardware acceleration on many modern platforms. Let AES(K,M) denote evaluation of AES-128 with 128 bit key K and 128 bit message M. Define the 16-byte string IV = 0x000102030405060708091011121314151 6. Then F and H are implemented as

```
F(X) = AES(IV,X) XOR X

H(X) = AES( AES(IV, X1) XOR X1, X2) XOR X2
```

where X = X1 || X2, i.e. X1 denotes the most significant 16 bytes of X and X2 the least significant 16 bytes. For these parameter sets H_m is implemented as SHA3-256 and PRF_m as SHA3-256 in PRF/MAC mode.

For the n = m = 32 and n = m = 64 settings, all functions are implemented using SHA3-256 and SHA3-512, respectively.

5.1. Zero Bitmasks

For applications that require a very small public key this note additionally defines zero bitmasks parameter sets. For these parameter sets the bitmasks are set to an all-zero string. The XMSS and XMSS^MT public keys for these parameter sets contain no bitmasks. Instead, they only contain the single n-byte value holding the root node. When handling zero bitmasks parameter sets, implementations MAY internally use an all-zero string as bitmasks and stick to the same algorithms as for the other parameter sets. Implementations MAY omit the XOR with an all-zero bitmask. Zero bitmasks parameter sets are only defined for n = 32 and n = 64, as formal security reductions require the used hash functions to be collision-resistant in this case. Hence, the estimated classical security levels are 128 and 256 bits for n = 32 and n = 64 with zero bitmasks, respectively. The corresponding post-quantum security levels are approximately 85 and 170 bits, respectively.

5.2. WOTS+ Parameters

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

Name	F	m	n	w	l
WOTSP_AES128_M32_W4	AES128	32	16	4	133
WOTSP_AES128_M32_W8	AES128	32	16	8	90
WOTSP_AES128_M32_W16	AES128	32	16	16	67
WOTSP_SHA3-256_M32_W4	SHA3	32	32	4	133
WOTSP_SHA3-256_M32_W8	SHA3	32	32	8	90
WOTSP_SHA3-256_M32_W16	SHA3	32	32	16	67
WOTSP_SHA3-512_M64_W4	SHA3	64	64	4	261
WOTSP_SHA3-512_M64_W8	SHA3	64	64	8	175
WOTSP_SHA3-512_M64_W16	SHA3	64	64	16	131

Table 1

Here SHA3 denotes the NIST standard hash function, also known as Keccak [DRAFTFIPS202]. XDR formats for WOTS+ are listed in Appendix A.

5.3. 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 and PRF_m MUST be specified. This section defines several XMSS signature systems, each of which is identified by a name.

The XDR formats for XMSS are listed in Appendix B.

5.3.1. XMSS Parameters

We first define XMSS signature methods as described in Section 4.1. We define parameter sets that implement the functions using AES and SHA3 as described above as well as pure SHA3 parameter sets.

5.3.1.1. XMSS Parameters with AES and SHA3

The following XMSS signature methods implement the functions F, H, H_m and PRF_m using AES and SHA3 as described above.

Name	m	n	w	l	h
XMSS_AES128_M32_W4_H10	32	16	4	133	10
XMSS_AES128_M32_W4_H16	32	16	4	133	16
XMSS_AES128_M32_W4_H20	32	16	4	133	20
XMSS_AES128_M32_W8_H10	32	16	8	90	10
XMSS_AES128_M32_W8_H16	32	16	8	90	16
XMSS_AES128_M32_W8_H20	32	16	8	90	20
XMSS_AES128_M32_W16_H10	32	16	16	67	10
XMSS_AES128_M32_W16_H16	32	16	16	67	16
XMSS_AES128_M32_W16_H20	32	16	16	67	20

Table 2

5.3.1.2. XMSS Parameters with SHA3

The following XMSS signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h
XMSS_SHA3-256_M32_W4_H10	32	32	4	133	10
XMSS_SHA3-256_M32_W4_H16	32	32	4	133	16
XMSS_SHA3-256_M32_W4_H20	32	32	4	133	20
XMSS_SHA3-256_M32_W8_H10	32	32	8	90	10
XMSS_SHA3-256_M32_W8_H16	32	32	8	90	16
XMSS_SHA3-256_M32_W8_H20	32	32	8	90	20
XMSS_SHA3-256_M32_W16_H10	32	32	16	67	10
XMSS_SHA3-256_M32_W16_H16	32	32	16	67	16
XMSS_SHA3-256_M32_W16_H20	32	32	16	67	20
XMSS_SHA3-512_M64_W4_H10	64	64	4	261	10
XMSS_SHA3-512_M64_W4_H16	64	64	4	261	16
XMSS_SHA3-512_M64_W4_H20	64	64	4	261	20
XMSS_SHA3-512_M64_W8_H10	64	64	8	175	10
XMSS_SHA3-512_M64_W8_H16	64	64	8	175	16
XMSS_SHA3-512_M64_W8_H20	64	64	8	175	20
XMSS_SHA3-512_M64_W16_H10	64	64	16	131	10
XMSS_SHA3-512_M64_W16_H16	64	64	16	131	16
XMSS_SHA3-512_M64_W16_H20	64	64	16	131	20

Table 3

5.3.2. XMSS Parameters With Empty Bitmasks

We now define XMSS signature methods for the zero bitmasks special case described in Section 5.1. For this setting all signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h
XMSS_SHA3-256_M32_W4_H10_z	32	32	4	133	10
XMSS_SHA3-256_M32_W4_H16_z	32	32	4	133	16
XMSS_SHA3-256_M32_W4_H20_z	32	32	4	133	20
XMSS_SHA3-256_M32_W8_H10_z	32	32	8	90	10
XMSS_SHA3-256_M32_W8_H16_z	32	32	8	90	16
XMSS_SHA3-256_M32_W8_H20_z	32	32	8	90	20
XMSS_SHA3-256_M32_W16_H10_z	32	32	16	67	10
XMSS_SHA3-256_M32_W16_H16_z	32	32	16	67	16
XMSS_SHA3-256_M32_W16_H20_z	32	32	16	67	20
XMSS_SHA3-512_M64_W4_H10_z	64	64	4	261	10
XMSS_SHA3-512_M64_W4_H16_z	64	64	4	261	16
XMSS_SHA3-512_M64_W4_H20_z	64	64	4	261	20
XMSS_SHA3-512_M64_W8_H10_z	64	64	8	175	10
XMSS_SHA3-512_M64_W8_H16_z	64	64	8	175	16
XMSS_SHA3-512_M64_W8_H20_z	64	64	8	175	20
XMSS_SHA3-512_M64_W16_H10_z	64	64	16	131	10
XMSS_SHA3-512_M64_W16_H16_z	64	64	16	131	16
XMSS_SHA3-512_M64_W16_H20_z	64	64	16	131	20

Table 4

5.4. 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 and PRF_m MUST be specified. This section defines several XMSS^MT signature systems, each of which is identified by a name.

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

5.4.1. XMSS^MT Parameters

We first define XMSS^MT signature methods as described in Section 4.2. We define parameter sets that implement the functions using AES and SHA3 as described above as well as pure SHA3 parameter sets.

5.4.1.1. XMSS^MT Parameters with AES and SHA3

The following XMSS^MT signature methods implement the functions F, H, H_m and PRF_m using AES and SHA3 as described above.

Name	m	n	w	l	h	d
XMSSMT_AES128_M32_W4_H20_D2	32	16	4	133	20	2
XMSSMT_AES128_M32_W4_H20_D4	32	16	4	133	20	4
XMSSMT_AES128_M32_W4_H40_D2	32	16	4	133	40	2
XMSSMT_AES128_M32_W4_H40_D4	32	16	4	133	40	4
XMSSMT_AES128_M32_W4_H40_D8	32	16	4	133	40	8
XMSSMT_AES128_M32_W4_H60_D3	32	16	4	133	60	3
XMSSMT_AES128_M32_W4_H60_D6	32	16	4	133	60	6
XMSSMT_AES128_M32_W4_H60_D12	32	16	4	133	60	12
XMSSMT_AES128_M32_W8_H20_D2	32	16	8	90	20	2
XMSSMT_AES128_M32_W8_H20_D4	32	16	8	90	20	4
XMSSMT_AES128_M32_W8_H40_D2	32	16	8	90	40	2
XMSSMT_AES128_M32_W8_H40_D4	32	16	8	90	40	4

XMSSMT_AES128_M32_W8_H40_D8	32	16	8	90	40	8
XMSSMT_AES128_M32_W8_H60_D3	32	16	8	90	60	3
XMSSMT_AES128_M32_W8_H60_D6	32	16	8	90	60	6
XMSSMT_AES128_M32_W8_H60_D12	32	16	8	90	60	12
XMSSMT_AES128_M32_W16_H20_D2	32	16	16	67	20	2
XMSSMT_AES128_M32_W16_H20_D4	32	16	16	67	20	4
XMSSMT_AES128_M32_W16_H40_D2	32	16	16	67	40	2
XMSSMT_AES128_M32_W16_H40_D4	32	16	16	67	40	4
XMSSMT_AES128_M32_W16_H40_D8	32	16	16	67	40	8
XMSSMT_AES128_M32_W16_H60_D3	32	16	16	67	60	3
XMSSMT_AES128_M32_W16_H60_D6	32	16	16	67	60	6
XMSSMT_AES128_M32_W16_H60_D12	32	16	16	67	60	12

Table 5

5.4.1.2. XMSS^MT Parameters with SHA3

The following XMSS^MT signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h	d
XMSSMT_SHA3-256_M32_W4_H20_D2	32	32	4	133	20	2
XMSSMT_SHA3-256_M32_W4_H20_D4	32	32	4	133	20	4
XMSSMT_SHA3-256_M32_W4_H40_D2	32	32	4	133	40	2
XMSSMT_SHA3-256_M32_W4_H40_D4	32	32	4	133	40	4
XMSSMT_SHA3-256_M32_W4_H40_D8	32	32	4	133	40	8
XMSSMT_SHA3-256_M32_W4_H60_D3	32	32	4	133	60	3
XMSSMT_SHA3-256_M32_W4_H60_D6	32	32	4	133	60	6

XMSSMT_SHA3-256_M32_W4_H60_D12	32	32	4	133	60	12
XMSSMT_SHA3-256_M32_W8_H20_D2	32	32	8	90	20	2
XMSSMT_SHA3-256_M32_W8_H20_D4	32	32	8	90	20	4
XMSSMT_SHA3-256_M32_W8_H40_D2	32	32	8	90	40	2
XMSSMT_SHA3-256_M32_W8_H40_D4	32	32	8	90	40	4
XMSSMT_SHA3-256_M32_W8_H40_D8	32	32	8	90	40	8
XMSSMT_SHA3-256_M32_W8_H60_D3	32	32	8	90	60	3
XMSSMT_SHA3-256_M32_W8_H60_D6	32	32	8	90	60	6
XMSSMT_SHA3-256_M32_W8_H60_D12	32	32	8	90	60	12
XMSSMT_SHA3-256_M32_W16_H20_D2	32	32	16	67	20	2
XMSSMT_SHA3-256_M32_W16_H20_D4	32	32	16	67	20	4
XMSSMT_SHA3-256_M32_W16_H40_D2	32	32	16	67	40	2
XMSSMT_SHA3-256_M32_W16_H40_D4	32	32	16	67	40	4
XMSSMT_SHA3-256_M32_W16_H40_D8	32	32	16	67	40	8
XMSSMT_SHA3-256_M32_W16_H60_D3	32	32	16	67	60	3
XMSSMT_SHA3-256_M32_W16_H60_D6	32	32	16	67	60	6
XMSSMT_SHA3-256_M32_W16_H60_D12	32	32	16	67	60	12
XMSSMT_SHA3-512_M64_W4_H20_D2	64	64	4	261	20	2
XMSSMT_SHA3-512_M64_W4_H20_D4	64	64	4	261	20	4
XMSSMT_SHA3-512_M64_W4_H40_D2	64	64	4	261	40	2
XMSSMT_SHA3-512_M64_W4_H40_D4	64	64	4	261	40	4
XMSSMT_SHA3-512_M64_W4_H40_D8	64	64	4	261	40	8
XMSSMT_SHA3-512_M64_W4_H60_D3	64	64	4	261	60	3
XMSSMT_SHA3-512_M64_W4_H60_D6	64	64	4	261	60	6

XMSSMT_SHA3-512_M64_W4_H60_D12	64	64	4	261	60	12
XMSSMT_SHA3-512_M64_W8_H20_D2	64	64	8	175	20	2
XMSSMT_SHA3-512_M64_W8_H20_D4	64	64	8	175	20	4
XMSSMT_SHA3-512_M64_W8_H40_D2	64	64	8	175	40	2
XMSSMT_SHA3-512_M64_W8_H40_D4	64	64	8	175	40	4
XMSSMT_SHA3-512_M64_W8_H40_D8	64	64	8	175	40	8
XMSSMT_SHA3-512_M64_W8_H60_D3	64	64	8	175	60	3
XMSSMT_SHA3-512_M64_W8_H60_D6	64	64	8	175	60	6
XMSSMT_SHA3-512_M64_W8_H60_D12	64	64	8	175	60	12
XMSSMT_SHA3-512_M64_W16_H20_D2	64	64	16	131	20	2
XMSSMT_SHA3-512_M64_W16_H20_D4	64	64	16	131	20	4
XMSSMT_SHA3-512_M64_W16_H40_D2	64	64	16	131	40	2
XMSSMT_SHA3-512_M64_W16_H40_D4	64	64	16	131	40	4
XMSSMT_SHA3-512_M64_W16_H40_D8	64	64	16	131	40	8
XMSSMT_SHA3-512_M64_W16_H60_D3	64	64	16	131	60	3
XMSSMT_SHA3-512_M64_W16_H60_D6	64	64	16	131	60	6
XMSSMT_SHA3-512_M64_W16_H60_D12	64	64	16	131	60	12

Table 6

5.4.2. XMSS^MT Parameters With Empty Bitmasks

We now define XMSS^MT signature methods for the zero bitmasks special case described in Section 5.1. For this setting all signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h	d

XMSSMT_SHA3-256_M32_W4_H20_D2_z	32	32	4	133	20	2
XMSSMT_SHA3-256_M32_W4_H20_D4_z	32	32	4	133	20	4
XMSSMT_SHA3-256_M32_W4_H40_D2_z	32	32	4	133	40	2
XMSSMT_SHA3-256_M32_W4_H40_D4_z	32	32	4	133	40	4
XMSSMT_SHA3-256_M32_W4_H40_D8_z	32	32	4	133	40	8
XMSSMT_SHA3-256_M32_W4_H60_D3_z	32	32	4	133	60	3
XMSSMT_SHA3-256_M32_W4_H60_D6_z	32	32	4	133	60	6
XMSSMT_SHA3-256_M32_W4_H60_D12_z	32	32	4	133	60	12
XMSSMT_SHA3-256_M32_W8_H20_D2_z	32	32	8	90	20	2
XMSSMT_SHA3-256_M32_W8_H20_D4_z	32	32	8	90	20	4
XMSSMT_SHA3-256_M32_W8_H40_D2_z	32	32	8	90	40	2
XMSSMT_SHA3-256_M32_W8_H40_D4_z	32	32	8	90	40	4
XMSSMT_SHA3-256_M32_W8_H40_D8_z	32	32	8	90	40	8
XMSSMT_SHA3-256_M32_W8_H60_D3_z	32	32	8	90	60	3
XMSSMT_SHA3-256_M32_W8_H60_D6_z	32	32	8	90	60	6
XMSSMT_SHA3-256_M32_W8_H60_D12_z	32	32	16	67	60	12
XMSSMT_SHA3-256_M32_W16_H20_D2_z	32	32	16	67	20	2
XMSSMT_SHA3-256_M32_W16_H20_D4_z	32	32	16	67	20	4
XMSSMT_SHA3-256_M32_W16_H40_D2_z	32	32	16	67	40	2
XMSSMT_SHA3-256_M32_W16_H40_D4_z	32	32	16	67	40	4
XMSSMT_SHA3-256_M32_W16_H40_D8_z	32	32	16	67	40	8
XMSSMT_SHA3-256_M32_W16_H60_D3_z	32	32	16	67	60	3
XMSSMT_SHA3-256_M32_W16_H60_D6_z	32	32	16	67	60	6
XMSSMT_SHA3-256_M32_W16_H60_D12_z	32	32	16	67	60	12

XMSSMT_SHA3-512_M64_W4_H20_D2_z	64	64	4	261	20	2
XMSSMT_SHA3-512_M64_W4_H20_D4_z	64	64	4	261	20	4
XMSSMT_SHA3-512_M64_W4_H40_D2_z	64	64	4	261	40	2
XMSSMT_SHA3-512_M64_W4_H40_D4_z	64	64	4	261	40	4
XMSSMT_SHA3-512_M64_W4_H40_D8_z	64	64	4	261	40	8
XMSSMT_SHA3-512_M64_W4_H60_D3_z	64	64	4	261	60	3
XMSSMT_SHA3-512_M64_W4_H60_D6_z	64	64	4	261	60	6
XMSSMT_SHA3-512_M64_W4_H60_D12_z	64	64	4	261	60	12
XMSSMT_SHA3-512_M64_W8_H20_D2_z	64	64	8	175	20	2
XMSSMT_SHA3-512_M64_W8_H20_D4_z	64	64	8	175	20	4
XMSSMT_SHA3-512_M64_W8_H40_D2_z	64	64	8	175	40	2
XMSSMT_SHA3-512_M64_W8_H40_D4_z	64	64	8	175	40	4
XMSSMT_SHA3-512_M64_W8_H40_D8_z	64	64	8	175	40	8
XMSSMT_SHA3-512_M64_W8_H60_D3_z	64	64	8	175	60	3
XMSSMT_SHA3-512_M64_W8_H60_D6_z	64	64	8	175	60	6
XMSSMT_SHA3-512_M64_W8_H60_D12_z	64	64	8	175	60	12
XMSSMT_SHA3-512_M64_W16_H20_D2_z	64	64	16	131	20	2
XMSSMT_SHA3-512_M64_W16_H20_D4_z	64	64	16	131	20	4
XMSSMT_SHA3-512_M64_W16_H40_D2_z	64	64	16	131	40	2
XMSSMT_SHA3-512_M64_W16_H40_D4_z	64	64	16	131	40	4
XMSSMT_SHA3-512_M64_W16_H40_D8_z	64	64	16	131	40	8
XMSSMT_SHA3-512_M64_W16_H60_D3_z	64	64	16	131	60	3
XMSSMT_SHA3-512_M64_W16_H60_D6_z	64	64	16	131	60	6
XMSSMT_SHA3-512_M64_W16_H60_D12_z	64	64	16	131	60	12

Table 7

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 parameter w is constrained to powers of 2 to support simpler and more efficient implementations. Furthermore, w is restricted to the set {4, 8, 16}. No bigger values are included since the decrease in signature size then becomes less significant. The value w = 2 was not included since w = 4 leads to similar runtimes but a halved signature size. This is the case because while chains get twice as long, thereby increasing runtime, the number of chains is roughly halved. For instance, assuming m = n = 32, one obtains l = 38 for w = 2 and l = 19 for w = 4.

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_SHA3-512_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_AES128_M32_W4	Section 5.2	0x01000001
WOTSP_AES128_M32_W8	Section 5.2	0x02000002
WOTSP_AES128_M32_W16	Section 5.2	0x03000003
WOTSP_SHA3-256_M32_W4	Section 5.2	0x04000004
WOTSP_SHA3-256_M32_W8	Section 5.2	0x05000005
WOTSP_SHA3-256_M32_W16	Section 5.2	0x06000006
WOTSP_SHA3-512_M64_W4	Section 5.2	0x07000007
WOTSP_SHA3-512_M64_W8	Section 5.2	0x08000008
WOTSP_SHA3-512_M64_W16	Section 5.2	0x09000009

Table 8

The XMSS registry is as follows.

Name	Reference	Numeric Identifier
XMSS_SHA3-256_M32_W4_H10_Z	Section 5.3	0x01000001
XMSS_SHA3-256_M32_W4_H16_Z	Section 5.3	0x02000002
XMSS_SHA3-256_M32_W4_H20_Z	Section 5.3	0x03000003
XMSS_SHA3-256_M32_W8_H10_Z	Section 5.3	0x04000004
XMSS_SHA3-256_M32_W8_H16_Z	Section 5.3	0x05000005
XMSS_SHA3-256_M32_W8_H20_Z	Section 5.3	0x06000006
XMSS_SHA3-256_M32_W16_H10_Z	Section 5.3	0x07000007
XMSS_SHA3-256_M32_W16_H16_Z	Section 5.3	0x08000008
XMSS_SHA3-256_M32_W16_H20_Z	Section 5.3	0x09000009
XMSS_SHA3-512_M64_W4_H10_Z	Section 5.3	0x0a00000a
XMSS_SHA3-512_M64_W4_H16_Z	Section 5.3	0x0b00000b
XMSS_SHA3-512_M64_W4_H20_Z	Section 5.3	0x0c00000c
XMSS_SHA3-512_M64_W8_H10_Z	Section 5.3	0x0d00000d
XMSS_SHA3-512_M64_W8_H16_Z	Section 5.3	0x0e00000e
XMSS_SHA3-512_M64_W8_H20_Z	Section 5.3	0x0f00000f
XMSS_SHA3-512_M64_W16_H10_Z	Section 5.3	0x01010101
XMSS_SHA3-512_M64_W16_H16_Z	Section 5.3	0x02010102
XMSS_SHA3-512_M64_W16_H20_Z	Section 5.3	0x03010103
XMSS_AES128_M32_W4_H10	Section 5.3	0x04010104
XMSS_AES128_M32_W4_H16	Section 5.3	0x05010105
XMSS_AES128_M32_W4_H20	Section 5.3	0x06010106

XMSS_AES128_M32_W8_H10	Section 5.3	0x07010107
XMSS_AES128_M32_W8_H16	Section 5.3	0x08010108
XMSS_AES128_M32_W8_H20	Section 5.3	0x09010109
XMSS_AES128_M32_W16_H10	Section 5.3	0x0a01010a
XMSS_AES128_M32_W16_H16	Section 5.3	0x0b01010b
XMSS_AES128_M32_W16_H20	Section 5.3	0x0c01010c
XMSS_SHA3-256_M32_W4_H10	Section 5.3	0x0d01010d
XMSS_SHA3-256_M32_W4_H16	Section 5.3	0x0e01010e
XMSS_SHA3-256_M32_W4_H20	Section 5.3	0x0f01010f
XMSS_SHA3-256_M32_W8_H10	Section 5.3	0x01020201
XMSS_SHA3-256_M32_W8_H16	Section 5.3	0x02020202
XMSS_SHA3-256_M32_W8_H20	Section 5.3	0x03020203
XMSS_SHA3-256_M32_W16_H10	Section 5.3	0x04020204
XMSS_SHA3-256_M32_W16_H16	Section 5.3	0x05020205
XMSS_SHA3-256_M32_W16_H20	Section 5.3	0x06020206
XMSS_SHA3-512_M64_W4_H10	Section 5.3	0x07020207
XMSS_SHA3-512_M64_W4_H16	Section 5.3	0x08020208
XMSS_SHA3-512_M64_W4_H20	Section 5.3	0x09020209
XMSS_SHA3-512_M64_W8_H10	Section 5.3	0x0a02020a
XMSS_SHA3-512_M64_W8_H16	Section 5.3	0x0b02020b
XMSS_SHA3-512_M64_W8_H20	Section 5.3	0x0c02020c
XMSS_SHA3-512_M64_W16_H10	Section 5.3	0x0d02020d
XMSS_SHA3-512_M64_W16_H16	Section 5.3	0x0e02020e
XMSS_SHA3-512_M64_W16_H20	Section 5.3	0x0f02020f

```
 +------------------------------+-------------+--------------------+
```

Table 9

The XMSS^MT registry is as follows.

Name	Reference	Numeric Identifier
XMSSMT_SHA3-256_M32_W4_H20_D2_Z	Section 5.4	0x01000001
XMSSMT_SHA3-256_M32_W4_H20_D4_Z	Section 5.4	0x02000002
XMSSMT_SHA3-256_M32_W4_H40_D2_Z	Section 5.4	0x03000003
XMSSMT_SHA3-256_M32_W4_H40_D4_Z	Section 5.4	0x04000004
XMSSMT_SHA3-256_M32_W4_H40_D8_Z	Section 5.4	0x05000005
XMSSMT_SHA3-256_M32_W4_H60_D3_Z	Section 5.4	0x06000006
XMSSMT_SHA3-256_M32_W4_H60_D6_Z	Section 5.4	0x07000007
XMSSMT_SHA3-256_M32_W4_H60_D12_Z	Section 5.4	0x08000008
XMSSMT_SHA3-256_M32_W8_H20_D2_Z	Section 5.4	0x09000009
XMSSMT_SHA3-256_M32_W8_H20_D4_Z	Section 5.4	0x0a00000a
XMSSMT_SHA3-256_M32_W8_H40_D2_Z	Section 5.4	0x0b00000b
XMSSMT_SHA3-256_M32_W8_H40_D4_Z	Section 5.4	0x0c00000c
XMSSMT_SHA3-256_M32_W8_H40_D8_Z	Section 5.4	0x0d00000d

XMSSMT_SHA3-256_M32_W8_H60_D3_Z	Section 5.4	0x0e00000e
XMSSMT_SHA3-256_M32_W8_H60_D6_Z	Section 5.4	0x0f00000f
XMSSMT_SHA3-256_M32_W8_H60_D12_Z	Section 5.4	0x00010100
XMSSMT_SHA3-256_M32_W16_H20_D2_Z	Section 5.4	0x01010101
XMSSMT_SHA3-256_M32_W16_H20_D4_Z	Section 5.4	0x02010102
XMSSMT_SHA3-256_M32_W16_H40_D2_Z	Section 5.4	0x03010103
XMSSMT_SHA3-256_M32_W16_H40_D4_Z	Section 5.4	0x04010104
XMSSMT_SHA3-256_M32_W16_H40_D8_Z	Section 5.4	0x05010105
XMSSMT_SHA3-256_M32_W16_H60_D3_Z	Section 5.4	0x06010106
XMSSMT_SHA3-256_M32_W16_H60_D6_Z	Section 5.4	0x07010107
XMSSMT_SHA3-256_M32_W16_H60_D12_Z	Section 5.4	0x08010108
XMSSMT_SHA3-512_M64_W4_H20_D2_Z	Section 5.4	0x09010109
XMSSMT_SHA3-512_M64_W4_H20_D4_Z	Section 5.4	0x0a01010a
XMSSMT_SHA3-512_M64_W4_H40_D2_Z	Section 5.4	0x0b01010b
XMSSMT_SHA3-512_M64_W4_H40_D4_Z	Section 5.4	0x0c01010c
XMSSMT_SHA3-512_M64_W4_H40_D8_Z	Section 5.4	0x0d01010d

XMSSMT_SHA3-512_M64_W4_H60_D3_Z	Section 5.4	0x0e01010e
XMSSMT_SHA3-512_M64_W4_H60_D6_Z	Section 5.4	0x0f01010f
XMSSMT_SHA3-512_M64_W4_H60_D12_Z	Section 5.4	0x00020200
XMSSMT_SHA3-512_M64_W8_H20_D2_Z	Section 5.4	0x01020201
XMSSMT_SHA3-512_M64_W8_H20_D4_Z	Section 5.4	0x02020202
XMSSMT_SHA3-512_M64_W8_H40_D2_Z	Section 5.4	0x03020203
XMSSMT_SHA3-512_M64_W8_H40_D4_Z	Section 5.4	0x04020204
XMSSMT_SHA3-512_M64_W8_H40_D8_Z	Section 5.4	0x05020205
XMSSMT_SHA3-512_M64_W8_H60_D3_Z	Section 5.4	0x06020206
XMSSMT_SHA3-512_M64_W8_H60_D6_Z	Section 5.4	0x07020207
XMSSMT_SHA3-512_M64_W8_H60_D12_Z	Section 5.4	0x08020208
XMSSMT_SHA3-512_M64_W16_H20_D2_Z	Section 5.4	0x09020209
XMSSMT_SHA3-512_M64_W16_H20_D4_Z	Section 5.4	0x0a02020a
XMSSMT_SHA3-512_M64_W16_H40_D2_Z	Section 5.4	0x0b02020b
XMSSMT_SHA3-512_M64_W16_H40_D4_Z	Section 5.4	0x0c02020c
XMSSMT_SHA3-512_M64_W16_H40_D8_Z	Section 5.4	0x0d02020d

XMSSMT_SHA3-512_M64_W16_H60_D3_Z	Section 5.4	0x0e02020e
XMSSMT_SHA3-512_M64_W16_H60_D6_Z	Section 5.4	0x0f02020f
XMSSMT_SHA3-512_M64_W16_H60_D12_Z	Section 5.4	0x00030300
XMSSMT_AES128_M32_W4_H20_D2	Section 5.4	0x01030301
XMSSMT_AES128_M32_W4_H20_D4	Section 5.4	0x02030302
XMSSMT_AES128_M32_W4_H40_D2	Section 5.4	0x03030303
XMSSMT_AES128_M32_W4_H40_D4	Section 5.4	0x04030304
XMSSMT_AES128_M32_W4_H40_D8	Section 5.4	0x05030305
XMSSMT_AES128_M32_W4_H60_D3	Section 5.4	0x06030306
XMSSMT_AES128_M32_W4_H60_D6	Section 5.4	0x07030307
XMSSMT_AES128_M32_W4_H60_D12	Section 5.4	0x08030308
XMSSMT_AES128_M32_W8_H20_D2	Section 5.4	0x09030309
XMSSMT_AES128_M32_W8_H20_D4	Section 5.4	0x0a03030a
XMSSMT_AES128_M32_W8_H40_D2	Section 5.4	0x0b03030b
XMSSMT_AES128_M32_W8_H40_D4	Section 5.4	0x0c03030c
XMSSMT_AES128_M32_W8_H40_D8	Section 5.4	0x0d03030d

XMSSMT_AES128_M32_W8_H60_D3	Section 5.4	0x0e03030e
XMSSMT_AES128_M32_W8_H60_D6	Section 5.4	0x0f03030f
XMSSMT_AES128_M32_W8_H60_D12	Section 5.4	0x00040400
XMSSMT_AES128_M32_W16_H20_D2	Section 5.4	0x01040401
XMSSMT_AES128_M32_W16_H20_D4	Section 5.4	0x02040402
XMSSMT_AES128_M32_W16_H40_D2	Section 5.4	0x03040403
XMSSMT_AES128_M32_W16_H40_D4	Section 5.4	0x04040404
XMSSMT_AES128_M32_W16_H40_D8	Section 5.4	0x05040405
XMSSMT_AES128_M32_W16_H60_D3	Section 5.4	0x06040406
XMSSMT_AES128_M32_W16_H60_D6	Section 5.4	0x07040407
XMSSMT_AES128_M32_W16_H60_D12	Section 5.4	0x08040408
XMSSMT_SHA3-256_M32_W4_H20_D2	Section 5.4	0x09040409
XMSSMT_SHA3-256_M32_W4_H20_D4	Section 5.4	0x0a04040a
XMSSMT_SHA3-256_M32_W4_H40_D2	Section 5.4	0x0b04040b
XMSSMT_SHA3-256_M32_W4_H40_D4	Section 5.4	0x0c04040c
XMSSMT_SHA3-256_M32_W4_H40_D8	Section 5.4	0x0d04040d

XMSSMT_SHA3-256_M32_W4_H60_D3	Section 5.4	0x0e04040e
XMSSMT_SHA3-256_M32_W4_H60_D6	Section 5.4	0x0f04040f
XMSSMT_SHA3-256_M32_W4_H60_D12	Section 5.4	0x00050500
XMSSMT_SHA3-256_M32_W8_H20_D2	Section 5.4	0x01050501
XMSSMT_SHA3-256_M32_W8_H20_D4	Section 5.4	0x02050502
XMSSMT_SHA3-256_M32_W8_H40_D2	Section 5.4	0x03050503
XMSSMT_SHA3-256_M32_W8_H40_D4	Section 5.4	0x04050504
XMSSMT_SHA3-256_M32_W8_H40_D8	Section 5.4	0x05050505
XMSSMT_SHA3-256_M32_W8_H60_D3	Section 5.4	0x06050506
XMSSMT_SHA3-256_M32_W8_H60_D6	Section 5.4	0x07050507
XMSSMT_SHA3-256_M32_W8_H60_D12	Section 5.4	0x08050508
XMSSMT_SHA3-256_M32_W16_H20_D2	Section 5.4	0x09050509
XMSSMT_SHA3-256_M32_W16_H20_D4	Section 5.4	0x0a05050a
XMSSMT_SHA3-256_M32_W16_H40_D2	Section 5.4	0x0b05050b
XMSSMT_SHA3-256_M32_W16_H40_D4	Section 5.4	0x0c05050c
XMSSMT_SHA3-256_M32_W16_H40_D8	Section 5.4	0x0d05050d

XMSSMT_SHA3-256_M32_W16_H60_D3	Section 5.4	0x0e05050e
XMSSMT_SHA3-256_M32_W16_H60_D6	Section 5.4	0x0f05050f
XMSSMT_SHA3-256_M32_W16_H60_D12	Section 5.4	0x00060600
XMSSMT_SHA3-512_M64_W4_H20_D2	Section 5.4	0x01060601
XMSSMT_SHA3-512_M64_W4_H20_D4	Section 5.4	0x02060602
XMSSMT_SHA3-512_M64_W4_H40_D2	Section 5.4	0x03060603
XMSSMT_SHA3-512_M64_W4_H40_D4	Section 5.4	0x04060604
XMSSMT_SHA3-512_M64_W4_H40_D8	Section 5.4	0x05060605
XMSSMT_SHA3-512_M64_W4_H60_D3	Section 5.4	0x06060606
XMSSMT_SHA3-512_M64_W4_H60_D6	Section 5.4	0x07060607
XMSSMT_SHA3-512_M64_W4_H60_D12	Section 5.4	0x08060608
XMSSMT_SHA3-512_M64_W8_H20_D2	Section 5.4	0x09060609
XMSSMT_SHA3-512_M64_W8_H20_D4	Section 5.4	0x0a06060a
XMSSMT_SHA3-512_M64_W8_H40_D2	Section 5.4	0x0b06060b
XMSSMT_SHA3-512_M64_W8_H40_D4	Section 5.4	0x0c06060c
XMSSMT_SHA3-512_M64_W8_H40_D8	Section 5.4	0x0d06060d

XMSSMT_SHA3-512_M64_W8_H60_D3	Section 5.4	0x0e06060e
XMSSMT_SHA3-512_M64_W8_H60_D6	Section 5.4	0x0f06060f
XMSSMT_SHA3-512_M64_W8_H60_D12	Section 5.4	0x00070700
XMSSMT_SHA3-512_M64_W16_H20_D2	Section 5.4	0x01070701
XMSSMT_SHA3-512_M64_W16_H20_D4	Section 5.4	0x02070702
XMSSMT_SHA3-512_M64_W16_H40_D2	Section 5.4	0x03070703
XMSSMT_SHA3-512_M64_W16_H40_D4	Section 5.4	0x04070704
XMSSMT_SHA3-512_M64_W16_H40_D8	Section 5.4	0x05070705
XMSSMT_SHA3-512_M64_W16_H60_D3	Section 5.4	0x06070706
XMSSMT_SHA3-512_M64_W16_H60_D6	Section 5.4	0x07070707
XMSSMT_SHA3-512_M64_W16_H60_D12	Section 5.4	0x08070708

Table 10

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 the schemes 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. For the scheme without initial message compression, 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 a folklore that one can securely combine a secure signature scheme for fixed length messages with an initial message digest. It is easy to proof that an attacker either must break the security of the fixed-input-length signature scheme or the collision resistance of the used hash function. XMSS and XMSS^MT 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 that the resulting scheme is secure if the hash function is not collision-resistant but fulfills some weaker security properties.

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 Burt Kaliski, and David McGrew for their help.

10. References

10.1. Normative References

[DRAFTFIPS202]
National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Draft FIPS 202, 2014.

[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

[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", To appear. 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., "Shoring up the Infrastructure: A Strategy for Standardizing Hash Signatures", Post Quantum 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_aes128_m32_w4    = 0x01000001,
  wotsp_aes128_m32_w8    = 0x02000002,
  wotsp_aes128_m32_w16   = 0x03000003,
  wotsp_sha3-256_m32_w4  = 0x04000004,
  wotsp_sha3-256_m32_w8  = 0x05000005,
  wotsp_sha3-256_m32_w16 = 0x06000006,
  wotsp_sha3-512_m64_w4  = 0x07000007,
  wotsp_sha3-512_m64_w8  = 0x08000008,
  wotsp_sha3-512_m64_w16 = 0x09000009,
};
```

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_aes128_m32_w4:
  case wotsp_sha3-256_m32_w4:
    bytestring32 ots_sig_m32_l133[133];

  case wotsp_aes128_m32_w8:
  case wotsp_sha3-256_m32_w8:
    bytestring32 ots_sig_m32_l90[90];

  case wotsp_aes128_m32_w16:
  case wotsp_sha3-256_m32_w16:
    bytestring32 ots_sig_m32_l67[67];

  case wotsp_sha3-512_m64_w4:
    bytestring64 ots_sig_m64_l261[261];

  case wotsp_sha3-512_m64_w8:
    bytestring64 ots_sig_m64_l75[175];

  case wotsp_sha3-512_m64_w16:
    bytestring64 ots_sig_m64_l18[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_aes128_m32_w4:
  case wotsp_sha3-256_m32_w4:
    bytestring32 ots_pubk_m32_l133[133];

  case wotsp_aes128_m32_w8:
  case wotsp_sha3-256_m32_w8:
    bytestring32 ots_pubk_m32_l90[90];

  case wotsp_aes128_m32_w16:
  case wotsp_sha3-256_m32_w16:
    bytestring32 ots_pubk_m32_l67[67];

  case wotsp_sha3-512_m64_w4:
    bytestring64 ots_pubk_m64_l261[261];

  case wotsp_sha3-512_m64_w8:
    bytestring64 ots_pubk_m64_l75[175];

  case wotsp_sha3-512_m64_w16:
    bytestring64 ots_pubk_m64_l18[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];
typedef opaque bytestring16[16];

/* Definition of parameter sets */

enum xmss_algorithm_type {
  xmss_reserved                 = 0x00000000,

  /* Empty bitmasks */

  /* 128 bit classical security, 85 bit post-quantum security */

  xmss_sha3-256_m32_w4_h10_z  = 0x01000001,
  xmss_sha3-256_m32_w4_h16_z  = 0x02000002,
```

```
xmss_sha3-256_m32_w4_h20_z  = 0x03000003,

xmss_sha3-256_m32_w8_h10_z  = 0x04000004,
xmss_sha3-256_m32_w8_h16_z  = 0x05000005,
xmss_sha3-256_m32_w8_h20_z  = 0x06000006,

xmss_sha3-256_m32_w16_h10_z = 0x07000007,
xmss_sha3-256_m32_w16_h16_z = 0x08000008,
xmss_sha3-256_m32_w16_h20_z = 0x09000009,

/* 256 bit classical security, 170 bit post-quantum security */

xmss_sha3-512_m64_w4_h10_z  = 0x0a00000a,
xmss_sha3-512_m64_w4_h16_z  = 0x0b00000b,
xmss_sha3-512_m64_w4_h20_z  = 0x0c00000c,

xmss_sha3-512_m64_w8_h10_z  = 0x0d00000d,
xmss_sha3-512_m64_w8_h16_z  = 0x0e00000e,
xmss_sha3-512_m64_w8_h20_z  = 0x0f00000f,

xmss_sha3-512_m64_w16_h10_z = 0x01010101,
xmss_sha3-512_m64_w16_h16_z = 0x02010102,
xmss_sha3-512_m64_w16_h20_z = 0x03010103,

/* Non-empty bitmasks */

/* 128 bit classical security, 64 bit post-quantum security */

xmss_aes128_m32_w4_h10        = 0x04010104,
xmss_aes128_m32_w4_h16        = 0x05010105,
xmss_aes128_m32_w4_h20        = 0x06010106,

xmss_aes128_m32_w8_h10        = 0x07010107,
xmss_aes128_m32_w8_h16        = 0x08010108,
xmss_aes128_m32_w8_h20        = 0x09010109,

xmss_aes128_m32_w16_h10       = 0x0a01010a,
xmss_aes128_m32_w16_h16       = 0x0b01010b,
xmss_aes128_m32_w16_h20       = 0x0c01010c,

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

xmss_sha3-256_m32_w4_h10      = 0x0d01010d,
xmss_sha3-256_m32_w4_h16      = 0x0e01010e,
xmss_sha3-256_m32_w4_h20      = 0x0f01010f,

xmss_sha3-256_m32_w8_h10      = 0x01020201,
xmss_sha3-256_m32_w8_h16      = 0x02020202,
```

```
        xmss_sha3-256_m32_w8_h20      = 0x03020203,

        xmss_sha3-256_m32_w16_h10     = 0x04020204,
        xmss_sha3-256_m32_w16_h16     = 0x05020205,
        xmss_sha3-256_m32_w16_h20     = 0x06020206,

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

        xmss_sha3-512_m64_w4_h10      = 0x07020207,
        xmss_sha3-512_m64_w4_h16      = 0x08020208,
        xmss_sha3-512_m64_w4_h20      = 0x09020209,

        xmss_sha3-512_m64_w8_h10      = 0x0a02020a,
        xmss_sha3-512_m64_w8_h16      = 0x0b02020b,
        xmss_sha3-512_m64_w8_h20      = 0x0c02020c,

        xmss_sha3-512_m64_w16_h10     = 0x0d02020d,
        xmss_sha3-512_m64_w16_h16     = 0x0e02020e,
        xmss_sha3-512_m64_w16_h20     = 0x0f02020f,
      };
```

XMSS signatures are defined using XDR syntax as follows:

```
      /* Authentication path types */

      union xmss_path switch (xmss_algorithm_type type) {
        case xmss_sha3-256_m32_w4_h10_z:
        case xmss_sha3-256_m32_w8_h10_z:
        case xmss_sha3-256_m32_w16_h10_z:
        case xmss_sha3-256_m32_w4_h10:
        case xmss_sha3-256_m32_w8_h10:
        case xmss_sha3-256_m32_w16_h10:
          bytestring32 path_n32_t10[10];

        case xmss_sha3-256_m32_w4_h16_z:
        case xmss_sha3-256_m32_w8_h16_z:
        case xmss_sha3-256_m32_w16_h16_z:
        case xmss_sha3-256_m32_w4_h16:
        case xmss_sha3-256_m32_w8_h16:
        case xmss_sha3-256_m32_w16_h16:
          bytestring32 path_n32_t16[16];

        case xmss_sha3-256_m32_w4_h20_z:
        case xmss_sha3-256_m32_w8_h20_z:
        case xmss_sha3-256_m32_w16_h20_z:
        case xmss_sha3-256_m32_w4_h20:
```

```
        case xmss_sha3-256_m32_w8_h20:
        case xmss_sha3-256_m32_w16_h20:
          bytestring32 path_n32_t20[20];

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w4_h10:
        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
          bytestring64 path_n64_t10[10];

        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
          bytestring64 path_n64_t16[16];

        case xmss_sha3-512_m64_w4_h20_z:
        case xmss_sha3-512_m64_w8_h20_z:
        case xmss_sha3-512_m64_w16_h20_z:
        case xmss_sha3-512_m64_w4_h20:
        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 path_n64_t20[20];

        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w16_h10:
          bytestring16 path_n16_t10[10];

        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w16_h16:
          bytestring16 path_n16_t16[16];

        case xmss_aes128_m32_w4_h20:
        case xmss_aes128_m32_w8_h20:
        case xmss_aes128_m32_w16_h20:
          bytestring16 path_n16_t20[20];

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

      /* Types for XMSS random strings */
```

```
union random_string_xmss switch (xmss_algorithm_type type) {
  case xmss_sha3-256_m32_w4_h10_z:
  case xmss_sha3-256_m32_w4_h16_z:
  case xmss_sha3-256_m32_w4_h20_z:
  case xmss_sha3-256_m32_w8_h10_z:
  case xmss_sha3-256_m32_w8_h16_z:
  case xmss_sha3-256_m32_w8_h20_z:
  case xmss_sha3-256_m32_w16_h10_z:
  case xmss_sha3-256_m32_w16_h16_z:
  case xmss_sha3-256_m32_w16_h20_z:
  case xmss_sha3-256_m32_w4_h10:
  case xmss_sha3-256_m32_w4_h16:
  case xmss_sha3-256_m32_w4_h20:
  case xmss_sha3-256_m32_w8_h10:
  case xmss_sha3-256_m32_w8_h16:
  case xmss_sha3-256_m32_w8_h20:
  case xmss_sha3-256_m32_w16_h10:
  case xmss_sha3-256_m32_w16_h16:
  case xmss_sha3-256_m32_w16_h20:
  case xmss_aes128_m32_w4_h10:
  case xmss_aes128_m32_w4_h16:
  case xmss_aes128_m32_w4_h20:
  case xmss_aes128_m32_w8_h10:
  case xmss_aes128_m32_w8_h16:
  case xmss_aes128_m32_w8_h20:
  case xmss_aes128_m32_w16_h10:
  case xmss_aes128_m32_w16_h16:
  case xmss_aes128_m32_w16_h20:
    bytestring32 rand_m32;

  case xmss_sha3-512_m64_w4_h10_z:
  case xmss_sha3-512_m64_w4_h16_z:
  case xmss_sha3-512_m64_w4_h20_z:
  case xmss_sha3-512_m64_w8_h10_z:
  case xmss_sha3-512_m64_w8_h16_z:
  case xmss_sha3-512_m64_w8_h20_z:
  case xmss_sha3-512_m64_w16_h10_z:
  case xmss_sha3-512_m64_w16_h16_z:
  case xmss_sha3-512_m64_w16_h20_z:
  case xmss_sha3-512_m64_w4_h10:
  case xmss_sha3-512_m64_w4_h16:
  case xmss_sha3-512_m64_w4_h20:
  case xmss_sha3-512_m64_w8_h10:
  case xmss_sha3-512_m64_w8_h16:
  case xmss_sha3-512_m64_w8_h20:
  case xmss_sha3-512_m64_w16_h10:
  case xmss_sha3-512_m64_w16_h16:
  case xmss_sha3-512_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_sha3-256_m32_w4_h10_z:
        case xmss_sha3-256_m32_w4_h16_z:
        case xmss_sha3-256_m32_w4_h20_z:
          wotsp_sha3-256_m32_w4;

        case xmss_sha3-256_m32_w8_h10_z:
        case xmss_sha3-256_m32_w8_h16_z:
        case xmss_sha3-256_m32_w8_h20_z:
          wotsp_sha3-256_m32_w8;

        case xmss_sha3-256_m32_w16_h10_z:
        case xmss_sha3-256_m32_w16_h16_z:
        case xmss_sha3-256_m32_w16_h20_z:
          wotsp_sha3-256_m32_w16

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w4_h20_z:
          wotsp_sha3-512_m64_w4;

        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w8_h20_z:
          wotsp_sha3-512_m64_w8;

        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w16_h20_z:
          wotsp_sha3-512_m64_w16;

        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w4_h20:
          wotsp_aes128_m32_w4;

        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w8_h20:
          wotsp_aes128_m32_w8;
```

```
        case xmss_aes128_m32_w16_h10:
        case xmss_aes128_m32_w16_h16:
        case xmss_aes128_m32_w16_h20:
          wotsp_aes128_m32_w16;

        case xmss_sha3-256_m32_w4_h10:
        case xmss_sha3-256_m32_w4_h16:
        case xmss_sha3-256_m32_w4_h20:
          wotsp_sha3-256_m32_w4;

        case xmss_sha3-256_m32_w8_h10:
        case xmss_sha3-256_m32_w8_h16:
        case xmss_sha3-256_m32_w8_h20:
          wotsp_sha3-256_m32_w8;

        case xmss_sha3-256_m32_w16_h10:
        case xmss_sha3-256_m32_w16_h16:
        case xmss_sha3-256_m32_w16_h20:
          wotsp_sha3-256_m32_w16;

        case xmss_sha3-512_m64_w4_h10:
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w4_h20:
          wotsp_sha3-512_m64_w4;

        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w8_h20:
          wotsp_sha3-512_m64_w8;

        case xmss_sha3-512_m64_w16_h10:
        case xmss_sha3-512_m64_w16_h16:
        case xmss_sha3-512_m64_w16_h20:
          wotsp_sha3-512_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;
    };
```

When no bitmasks are used, XMSS public keys are defined using XDR syntax as follows:

```
    /* Types for XMSS root node */

    union xmss_root switch (xmss_algorithm_type type) {
      case xmss_sha3-256_m32_w4_h10_z:
      case xmss_sha3-256_m32_w4_h16_z:
      case xmss_sha3-256_m32_w4_h20_z:
      case xmss_sha3-256_m32_w8_h10_z:
      case xmss_sha3-256_m32_w16_h10_z:
      case xmss_sha3-256_m32_w8_h16_z:
      case xmss_sha3-256_m32_w16_h16_z:
      case xmss_sha3-256_m32_w8_h20_z:
      case xmss_sha3-256_m32_w16_h20_z:
        bytestring32 root_n32;

      case xmss_sha3-512_m64_w4_h10_z:
      case xmss_sha3-512_m64_w4_h16_z:
      case xmss_sha3-512_m64_w4_h20_z:
      case xmss_sha3-512_m64_w8_h10_z:
      case xmss_sha3-512_m64_w16_h10_z:
      case xmss_sha3-512_m64_w8_h16_z:
      case xmss_sha3-512_m64_w16_h16_z:
      case xmss_sha3-512_m64_w8_h20_z:
      case xmss_sha3-512_m64_w16_h20_z:
        bytestring64 root_n64;

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

    /* XMSS public key structure */

    struct xmss_public_key {
      xmss_root root;  /* Root node */
    };
```

When bitmasks are used, XMSS public keys are defined using XDR syntax as follows:

```
/* Types for XMSS bitmasks */

union xmss_bm switch (xmss_algorithm_type type) {
  case xmss_aes128_m32_w4_h10:
    bytestring16 bm_n16_bm36[36];

  case xmss_aes128_m32_w4_h16:
    bytestring16 bm_n16_bm48[48];

  case xmss_aes128_m32_w4_h20:
    bytestring16 bm_n16_bm56[56];

  case xmss_aes128_m32_w8_h10:
  case xmss_aes128_m32_w16_h10:
    bytestring16 bm_n16_bm34[34];

  case xmss_aes128_m32_w8_h16:
  case xmss_aes128_m32_w16_h16:
    bytestring16 bm_n16_bm46[46];

  case xmss_aes128_m32_w8_h20:
  case xmss_aes128_m32_w16_h20:
    bytestring16 bm_n16_bm54[54];

  case xmss_sha3-256_m32_w4_h10:
    bytestring32 bm_n32_bm36[36];

  case xmss_sha3-256_m32_w4_h16:
    bytestring32 bm_n32_bm48[48];

  case xmss_sha3-256_m32_w4_h20:
    bytestring32 bm_n32_bm56[56];

  case xmss_sha3-256_m32_w8_h10:
  case xmss_sha3-256_m32_w16_h10:
    bytestring32 bm_n32_bm34[34];

  case xmss_sha3-256_m32_w8_h16:
  case xmss_sha3-256_m32_w16_h16:
    bytestring32 bm_n32_bm46[46];

  case xmss_sha3-256_m32_w8_h20:
  case xmss_sha3-256_m32_w16_h20:
    bytestring32 bm_n32_bm54[54];

  case xmss_sha3-512_m64_w4_h10:
    bytestring64 bm_n64_bm38[38];
```

```
        case xmss_sha3-512_m64_w4_h16:
          bytestring64 bm_n64_bm50[50];

        case xmss_sha3-512_m64_w4_h20:
          bytestring64 bm_n64_bm58[58];

        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
          bytestring64 bm_n64_bm36[36];

        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
          bytestring64 bm_n64_bm48[48];

        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 bm_n64_bm56[56];

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

      /* Types for XMSS root node */

      union xmss_root switch (xmss_algorithm_type type) {
        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w4_h20:
        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w16_h10:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w16_h16:
        case xmss_aes128_m32_w8_h20:
        case xmss_aes128_m32_w16_h20:
          bytestring16 root_n16;

        case xmss_sha3-256_m32_w4_h10:
        case xmss_sha3-256_m32_w4_h16:
        case xmss_sha3-256_m32_w4_h20:
        case xmss_sha3-256_m32_w8_h10:
        case xmss_sha3-256_m32_w16_h10:
        case xmss_sha3-256_m32_w8_h16:
        case xmss_sha3-256_m32_w16_h16:
        case xmss_sha3-256_m32_w8_h20:
        case xmss_sha3-256_m32_w16_h20:
          bytestring32 root_n32;

        case xmss_sha3-512_m64_w4_h10:
```

```
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w4_h20:
        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 root_n64;

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

      /* XMSS public key structure */

      struct xmss_public_key {
        xmss_bm bm;  /* Bitmasks */
        xmss_root root;  /* Root node */
      };
```

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,

        /* Empty bitmasks */

        /* 128 bit classical security, 85 bit post-quantum security */

        xmssmt_sha3-256_m32_w4_h20_d2_z   = 0x01000001,
        xmssmt_sha3-256_m32_w4_h20_d4_z   = 0x02000002,
        xmssmt_sha3-256_m32_w4_h40_d2_z   = 0x03000003,
        xmssmt_sha3-256_m32_w4_h40_d4_z   = 0x04000004,
        xmssmt_sha3-256_m32_w4_h40_d8_z   = 0x05000005,
        xmssmt_sha3-256_m32_w4_h60_d3_z   = 0x06000006,
```

```
xmssmt_sha3-256_m32_w4_h60_d6_z   = 0x07000007,
xmssmt_sha3-256_m32_w4_h60_d12_z  = 0x08000008,

xmssmt_sha3-256_m32_w8_h20_d2_z   = 0x09000009,
xmssmt_sha3-256_m32_w8_h20_d4_z   = 0x0a00000a,
xmssmt_sha3-256_m32_w8_h40_d2_z   = 0x0b00000b,
xmssmt_sha3-256_m32_w8_h40_d4_z   = 0x0c00000c,
xmssmt_sha3-256_m32_w8_h40_d8_z   = 0x0d00000d,
xmssmt_sha3-256_m32_w8_h60_d3_z   = 0x0e00000e,
xmssmt_sha3-256_m32_w8_h60_d6_z   = 0x0f00000f,
xmssmt_sha3-256_m32_w8_h60_d12_z  = 0x00010100,

xmssmt_sha3-256_m32_w16_h20_d2_z  = 0x01010101,
xmssmt_sha3-256_m32_w16_h20_d4_z  = 0x02010102,
xmssmt_sha3-256_m32_w16_h40_d2_z  = 0x03010103,
xmssmt_sha3-256_m32_w16_h40_d4_z  = 0x04010104,
xmssmt_sha3-256_m32_w16_h40_d8_z  = 0x05010105,
xmssmt_sha3-256_m32_w16_h60_d3_z  = 0x06010106,
xmssmt_sha3-256_m32_w16_h60_d6_z  = 0x07010107,
xmssmt_sha3-256_m32_w16_h60_d12_z = 0x08010108,

/* 256 bit classical security, 170 bit post-quantum security */

xmssmt_sha3-512_m64_w4_h20_d2_z   = 0x09010109,
xmssmt_sha3-512_m64_w4_h20_d4_z   = 0x0a01010a,
xmssmt_sha3-512_m64_w4_h40_d2_z   = 0x0b01010b,
xmssmt_sha3-512_m64_w4_h40_d4_z   = 0x0c01010c,
xmssmt_sha3-512_m64_w4_h40_d8_z   = 0x0d01010d,
xmssmt_sha3-512_m64_w4_h60_d3_z   = 0x0e01010e,
xmssmt_sha3-512_m64_w4_h60_d6_z   = 0x0f01010f,
xmssmt_sha3-512_m64_w4_h60_d12_z  = 0x00020200,

xmssmt_sha3-512_m64_w8_h20_d2_z   = 0x01020201,
xmssmt_sha3-512_m64_w8_h20_d4_z   = 0x02020202,
xmssmt_sha3-512_m64_w8_h40_d2_z   = 0x03020203,
xmssmt_sha3-512_m64_w8_h40_d4_z   = 0x04020204,
xmssmt_sha3-512_m64_w8_h40_d8_z   = 0x05020205,
xmssmt_sha3-512_m64_w8_h60_d3_z   = 0x06020206,
xmssmt_sha3-512_m64_w8_h60_d6_z   = 0x07020207,
xmssmt_sha3-512_m64_w8_h60_d12_z  = 0x08020208,

xmssmt_sha3-512_m64_w16_h20_d2_z  = 0x09020209,
xmssmt_sha3-512_m64_w16_h20_d4_z  = 0x0a02020a,
xmssmt_sha3-512_m64_w16_h40_d2_z  = 0x0b02020b,
xmssmt_sha3-512_m64_w16_h40_d4_z  = 0x0c02020c,
xmssmt_sha3-512_m64_w16_h40_d8_z  = 0x0d02020d,
xmssmt_sha3-512_m64_w16_h60_d3_z  = 0x0e02020e,
xmssmt_sha3-512_m64_w16_h60_d6_z  = 0x0f02020f,
```

```
xmssmt_sha3-512_m64_w16_h60_d12_z = 0x00030300,

/* Non-empty bitmasks */

/* 128 bit classical security, 64 bit post-quantum security */

xmssmt_aes128_m32_w4_h20_d2         = 0x01030301,
xmssmt_aes128_m32_w4_h20_d4         = 0x02030302,
xmssmt_aes128_m32_w4_h40_d2         = 0x03030303,
xmssmt_aes128_m32_w4_h40_d4         = 0x04030304,
xmssmt_aes128_m32_w4_h40_d8         = 0x05030305,
xmssmt_aes128_m32_w4_h60_d3         = 0x06030306,
xmssmt_aes128_m32_w4_h60_d6         = 0x07030307,
xmssmt_aes128_m32_w4_h60_d12        = 0x08030308,

xmssmt_aes128_m32_w8_h20_d2         = 0x09030309,
xmssmt_aes128_m32_w8_h20_d4         = 0x0a03030a,
xmssmt_aes128_m32_w8_h40_d2         = 0x0b03030b,
xmssmt_aes128_m32_w8_h40_d4         = 0x0c03030c,
xmssmt_aes128_m32_w8_h40_d8         = 0x0d03030d,
xmssmt_aes128_m32_w8_h60_d3         = 0x0e03030e,
xmssmt_aes128_m32_w8_h60_d6         = 0x0f03030f,
xmssmt_aes128_m32_w8_h60_d12        = 0x00040400,

xmssmt_aes128_m32_w16_h20_d2        = 0x01040401,
xmssmt_aes128_m32_w16_h20_d4        = 0x02040402,
xmssmt_aes128_m32_w16_h40_d2        = 0x03040403,
xmssmt_aes128_m32_w16_h40_d4        = 0x04040404,
xmssmt_aes128_m32_w16_h40_d8        = 0x05040405,
xmssmt_aes128_m32_w16_h60_d3        = 0x06040406,
xmssmt_aes128_m32_w16_h60_d6        = 0x07040407,
xmssmt_aes128_m32_w16_h60_d12       = 0x08040408,

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

xmssmt_sha3-256_m32_w4_h20_d2       = 0x09040409,
xmssmt_sha3-256_m32_w4_h20_d4       = 0x0a04040a,
xmssmt_sha3-256_m32_w4_h40_d2       = 0x0b04040b,
xmssmt_sha3-256_m32_w4_h40_d4       = 0x0c04040c,
xmssmt_sha3-256_m32_w4_h40_d8       = 0x0d04040d,
xmssmt_sha3-256_m32_w4_h60_d3       = 0x0e04040e,
xmssmt_sha3-256_m32_w4_h60_d6       = 0x0f04040f,
xmssmt_sha3-256_m32_w4_h60_d12      = 0x00050500,

xmssmt_sha3-256_m32_w8_h20_d2       = 0x01050501,
xmssmt_sha3-256_m32_w8_h20_d4       = 0x02050502,
xmssmt_sha3-256_m32_w8_h40_d2       = 0x03050503,
xmssmt_sha3-256_m32_w8_h40_d4       = 0x04050504,
```

```
        xmssmt_sha3-256_m32_w8_h40_d8       = 0x05050505,
        xmssmt_sha3-256_m32_w8_h60_d3       = 0x06050506,
        xmssmt_sha3-256_m32_w8_h60_d6       = 0x07050507,
        xmssmt_sha3-256_m32_w8_h60_d12      = 0x08050508,

        xmssmt_sha3-256_m32_w16_h20_d2      = 0x09050509,
        xmssmt_sha3-256_m32_w16_h20_d4      = 0x0a05050a,
        xmssmt_sha3-256_m32_w16_h40_d2      = 0x0b05050b,
        xmssmt_sha3-256_m32_w16_h40_d4      = 0x0c05050c,
        xmssmt_sha3-256_m32_w16_h40_d8      = 0x0d05050d,
        xmssmt_sha3-256_m32_w16_h60_d3      = 0x0e05050e,
        xmssmt_sha3-256_m32_w16_h60_d6      = 0x0f05050f,
        xmssmt_sha3-256_m32_w16_h60_d12     = 0x00060600,

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

        xmssmt_sha3-512_m64_w4_h20_d2       = 0x01060601,
        xmssmt_sha3-512_m64_w4_h20_d4       = 0x02060602,
        xmssmt_sha3-512_m64_w4_h40_d2       = 0x03060603,
        xmssmt_sha3-512_m64_w4_h40_d4       = 0x04060604,
        xmssmt_sha3-512_m64_w4_h40_d8       = 0x05060605,
        xmssmt_sha3-512_m64_w4_h60_d3       = 0x06060606,
        xmssmt_sha3-512_m64_w4_h60_d6       = 0x07060607,
        xmssmt_sha3-512_m64_w4_h60_d12      = 0x08060608,

        xmssmt_sha3-512_m64_w8_h20_d2       = 0x09060609,
        xmssmt_sha3-512_m64_w8_h20_d4       = 0x0a06060a,
        xmssmt_sha3-512_m64_w8_h40_d2       = 0x0b06060b,
        xmssmt_sha3-512_m64_w8_h40_d4       = 0x0c06060c,
        xmssmt_sha3-512_m64_w8_h40_d8       = 0x0d06060d,
        xmssmt_sha3-512_m64_w8_h60_d3       = 0x0e06060e,
        xmssmt_sha3-512_m64_w8_h60_d6       = 0x0f06060f,
        xmssmt_sha3-512_m64_w8_h60_d12      = 0x00070700,

        xmssmt_sha3-512_m64_w16_h20_d2      = 0x01070701,
        xmssmt_sha3-512_m64_w16_h20_d4      = 0x02070702,
        xmssmt_sha3-512_m64_w16_h40_d2      = 0x03070703,
        xmssmt_sha3-512_m64_w16_h40_d4      = 0x04070704,
        xmssmt_sha3-512_m64_w16_h40_d8      = 0x05070705,
        xmssmt_sha3-512_m64_w16_h60_d3      = 0x06070706,
        xmssmt_sha3-512_m64_w16_h60_d6      = 0x07070707,
        xmssmt_sha3-512_m64_w16_h60_d12     = 0x08070708,
      };
```

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_sha3-256_m32_w4_h20_d2_z:
  case xmssmt_sha3-256_m32_w4_h20_d4_z:
  case xmssmt_sha3-256_m32_w8_h20_d2_z:
  case xmssmt_sha3-256_m32_w8_h20_d4_z:
  case xmssmt_sha3-256_m32_w16_h20_d2_z:
  case xmssmt_sha3-256_m32_w16_h20_d4_z:
  case xmssmt_sha3-512_m64_w4_h20_d2_z:
  case xmssmt_sha3-512_m64_w4_h20_d4_z:
  case xmssmt_sha3-512_m64_w8_h20_d2_z:
  case xmssmt_sha3-512_m64_w8_h20_d4_z:
  case xmssmt_sha3-512_m64_w16_h20_d2_z:
  case xmssmt_sha3-512_m64_w16_h20_d4_z:
  case xmssmt_aes128_m32_w4_h20_d2:
  case xmssmt_aes128_m32_w4_h20_d4:
  case xmssmt_aes128_m32_w8_h20_d2:
  case xmssmt_aes128_m32_w8_h20_d4:
  case xmssmt_aes128_m32_w16_h20_d2:
  case xmssmt_aes128_m32_w16_h20_d4:
  case xmssmt_sha3-256_m32_w4_h20_d2:
  case xmssmt_sha3-256_m32_w4_h20_d4:
  case xmssmt_sha3-256_m32_w8_h20_d2:
  case xmssmt_sha3-256_m32_w8_h20_d4:
  case xmssmt_sha3-256_m32_w16_h20_d2:
  case xmssmt_sha3-256_m32_w16_h20_d4:
  case xmssmt_sha3-512_m64_w4_h20_d2:
  case xmssmt_sha3-512_m64_w4_h20_d4:
  case xmssmt_sha3-512_m64_w8_h20_d2:
  case xmssmt_sha3-512_m64_w8_h20_d4:
  case xmssmt_sha3-512_m64_w16_h20_d2:
  case xmssmt_sha3-512_m64_w16_h20_d4:
    bytestring3 idx3;

  case xmssmt_sha3-256_m32_w4_h40_d2_z:
  case xmssmt_sha3-256_m32_w4_h40_d4_z:
  case xmssmt_sha3-256_m32_w4_h40_d8_z:
  case xmssmt_sha3-256_m32_w8_h40_d2_z:
  case xmssmt_sha3-256_m32_w8_h40_d4_z:
  case xmssmt_sha3-256_m32_w8_h40_d8_z:
  case xmssmt_sha3-256_m32_w16_h40_d2_z:
  case xmssmt_sha3-256_m32_w16_h40_d4_z:
  case xmssmt_sha3-256_m32_w16_h40_d8_z:
  case xmssmt_sha3-512_m64_w4_h40_d2_z:
  case xmssmt_sha3-512_m64_w4_h40_d4_z:
  case xmssmt_sha3-512_m64_w4_h40_d8_z:
```

```
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
          bytestring5 idx5;

        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
```

```
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring8 idx8;

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

      union random_string_xmssmt switch (xmssmt_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h20_d2:
```

```
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h20_d2:
```

```
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 rand_m32;

        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
```

```
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_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_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
          bytestring16 xmss_reduced_n16_t88[88];

        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
          bytestring16 xmss_reduced_n16_t108[108];

        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
          bytestring16 xmss_reduced_n16_t128[128];

        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d4:
          bytestring16 xmss_reduced_n16_t66[66];
```

```
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
          bytestring16 xmss_reduced_n16_t86[86];

        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
          bytestring16 xmss_reduced_n16_t106[106];

        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
          bytestring16 xmss_reduced_n16_t55[55];

        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
          bytestring16 xmss_reduced_n16_t75[75];

        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
          bytestring16 xmss_reduced_n16_t95[95];

        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
          bytestring32 xmss_reduced_n32_t153[153];

        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
          bytestring32 xmss_reduced_n32_t173[173];

        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
          bytestring32 xmss_reduced_n32_t193[193];

        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
```

```
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d4:
          bytestring32 xmss_reduced_n32_t110[110];

        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
          bytestring32 xmss_reduced_n32_t130[130];

        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
          bytestring32 xmss_reduced_n32_t150[150];

        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
          bytestring32 xmss_reduced_n32_t87[87];

        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
          bytestring32 xmss_reduced_n32_t107[107];

        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 xmss_reduced_n32_t127[127];

        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
          bytestring64 xmss_reduced_n64_t281[281];
```

```
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
          bytestring64 xmss_reduced_n64_t301[301];

        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
          bytestring64 xmss_reduced_n64_t321[321];

        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
          bytestring64 xmss_reduced_n64_t195[195];

        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
          bytestring64 xmss_reduced_n64_t215[215];

        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
          bytestring64 xmss_reduced_n64_t235[235];

        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
          bytestring64 xmss_reduced_n64_t151[151];

        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
```

```
          bytestring64 xmss_reduced_n64_t171[171];

        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_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_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d2:
          xmss_reduced_others xmss_red_arr_d2[1];
```

```
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d3:
          xmss_reduced_others xmss_red_arr_d3[2];

        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w4_h20_d4:
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d4:
          xmss_reduced_others xmss_red_arr_d4[3];
```

```
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d6:
          xmss_reduced_others xmss_red_arr_d6[5];

        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w16_h40_d8:
          xmss_reduced_others xmss_red_arr_d8[7];

        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h60_d12:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h60_d12:
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h60_d12:
```

```
        case xmssmt_sha3-512_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;
      };
```

When no bitmasks are used, XMSS^MT public keys are defined using XDR syntax as follows:

```
      /* Types for XMSS^MT root node */

      union xmssmt_root switch (xmssmt_algorithm_type type) {
        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
```

```
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
          bytestring32 root_n32;

        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
          bytestring64 root_n64;

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

      /* XMSS^MT public key structure */

      struct xmssmt_public_key {
        xmssmt_root root;  /* Root node */
      };
```

When bitmasks are used, XMSS^MT public keys are defined using XDR syntax as follows:

```
      /* Types for XMSS^MT bitmasks */
```

```
union xmssmt_bm switch (xmssmt_algorithm_type type) {
  case xmssmt_aes128_m32_w4_h20_d2:
  case xmssmt_aes128_m32_w4_h40_d4:
  case xmssmt_aes128_m32_w4_h60_d6:
    bytestring16 bm_n16_t36[36];

  case xmssmt_aes128_m32_w4_h60_d3:
  case xmssmt_aes128_m32_w4_h40_d2:
    bytestring16 bm_n16_t36[56];

  case xmssmt_aes128_m32_w4_h20_d4:
  case xmssmt_aes128_m32_w4_h40_d8:
  case xmssmt_aes128_m32_w4_h60_d12:
    bytestring16 bm_n16_t26[26];

  case xmssmt_aes128_m32_w8_h20_d2:
  case xmssmt_aes128_m32_w8_h40_d4:
  case xmssmt_aes128_m32_w8_h60_d6:
  case xmssmt_aes128_m32_w16_h20_d2:
  case xmssmt_aes128_m32_w16_h40_d4:
  case xmssmt_aes128_m32_w16_h60_d6:
    bytestring16 bm_n16_t34[34];

  case xmssmt_aes128_m32_w8_h20_d4:
  case xmssmt_aes128_m32_w8_h40_d8:
  case xmssmt_aes128_m32_w8_h60_d12:
  case xmssmt_aes128_m32_w16_h20_d4:
  case xmssmt_aes128_m32_w16_h40_d8:
  case xmssmt_aes128_m32_w16_h60_d12:
    bytestring16 bm_n16_t24[24];

  case xmssmt_aes128_m32_w8_h40_d2:
  case xmssmt_aes128_m32_w8_h60_d3:
  case xmssmt_aes128_m32_w16_h40_d2:
  case xmssmt_aes128_m32_w16_h60_d3:
    bytestring16 bm_n16_t54[54];

  case xmssmt_sha3-256_m32_w4_h20_d2:
  case xmssmt_sha3-256_m32_w4_h40_d4:
  case xmssmt_sha3-256_m32_w4_h60_d6:
    bytestring32 bm_n32_t36[36];

  case xmssmt_sha3-256_m32_w4_h20_d4:
  case xmssmt_sha3-256_m32_w4_h40_d8:
  case xmssmt_sha3-256_m32_w4_h60_d12:
    bytestring32 bm_n32_t26[26];

  case xmssmt_sha3-256_m32_w4_h40_d2:
```

```
        case xmssmt_sha3-256_m32_w4_h60_d3:
          bytestring32 bm_n32_t56[56];

        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h60_d6:
          bytestring32 bm_n32_t34[34];

        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 bm_n32_t24[24];

        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h60_d3:
          bytestring32 bm_n32_t54[54];

        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h60_d6:
          bytestring64 bm_n64_t38[38];

        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d12:
          bytestring64 bm_n64_t28[28];

        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h60_d3:
          bytestring64 bm_n64_t58[58];

        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h60_d6:
          bytestring64 bm_n64_t36[36];

        case xmssmt_sha3-512_m64_w8_h20_d4:
```

```
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring64 bm_n64_t26[26];

        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h60_d3:
          bytestring64 bm_n64_t56[56];

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

      /* Types for XMSS^MT root node */

      union xmssmt_root switch (xmssmt_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
          bytestring16 root_n16;

        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
```

```
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 root_n32;

        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
```

```
          bytestring64 root_n64;

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

      /* XMSS^MT public key structure */

      struct xmssmt_public_key {
        xmssmt_bm bm;  /* Bitmasks */
        xmssmt_root root;  /* Root node */
      };
```

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 mbH
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