Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: April 22, 2017

A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua GmbH
A. Mohaisen
SUNY Buffalo

October 19, 2016

XMSS: Extended Hash-Based Signatures
draft-irtf-cfrg-xmss-hash-based-signatures-07

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 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 April 22, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 3
1.1. CFRG Note on Post-Quantum Cryptography 5
1.2. Conventions Used In This Document 5
2. Notation 5
2.1. Data Types 6
2.2. Functions 6
2.3. Operators 6
2.4. Integer to Byte Conversion 7
2.5. Hash Function Address Scheme 7
2.6. Strings of Base w Numbers 10
2.7. Member Functions 11
3. Primitives 12
3.1. WOTS+ One-Time Signatures 12
3.1.1. WOTS+ Parameters 12
3.1.1.1. WOTS+ Functions 13
3.1.2. WOTS+ Chaining Function 13
3.1.3. WOTS+ Private Key 14
3.1.4. WOTS+ Public Key 15
3.1.5. WOTS+ Signature Generation 15
3.1.6. WOTS+ Signature Verification 17
3.1.7. Pseudorandom Key Generation 18
4. Schemes 18
4.1. XMSS: eXtended Merkle Signature Scheme 18
4.1.1. XMSS Parameters 19
4.1.2. XMSS Hash Functions 19
4.1.3. XMSS Private Key 20
4.1.4. Randomized Tree Hashing 20
4.1.5. L-Trees 21
4.1.6. TreeHash 22
4.1.7. XMSS Key Generation 23
4.1.8. XMSS Signature 24

4.1.9. XMSS Signature Generation 25
4.1.10. XMSS Signature Verification 27
4.1.11. Pseudorandom Key Generation 29
4.1.12. Free Index Handling and Partial Private Keys 29
4.2. XMSS^MT: Multi-Tree XMSS 30
4.2.1. XMSS^MT Parameters 30
4.2.2. XMSS^MT Key generation 30
4.2.3. XMSS^MT Signature 33
4.2.4. XMSS^MT Signature Generation 34
4.2.5. XMSS^MT Signature Verification 36
4.2.6. Pseudorandom Key Generation 37
4.2.7. Free Index Handling and Partial Private Keys 38
5. Parameter Sets 38
5.1. WOTS+ Parameters 39
5.2. XMSS Parameters 40
5.3. XMSS^MT Parameters 41
6. Rationale 43
7. IANA Considerations 44
8. Security Considerations 48
8.1. Security Proofs 48
8.2. Minimal Security Assumptions 50
8.3. Post-Quantum Security 50
9. Acknowledgements 50
10. References 50
10.1. Normative References 50
10.2. Informative References 51
Appendix A. WOTS+ XDR Formats 52
Appendix B. XMSS XDR Formats 53
Appendix C. XMSS^MT XDR Formats 58
Appendix D. Changed since draft-irtf-cfrg-xmss-hash-based-signatures-06 65
Authors' Addresses 66

1. Introduction

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

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

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

Some progress has already been made toward standardizing and introducing hash-based signatures. McGrew and Curcio have published an Internet-Draft [DC16] specifying the Lamport-Diffie-Winternitz-Merkle (LDWM) scheme, also taking into account subsequent adaptations by Leighton and Micali. Independently, Buchmann, Dahmen and Huelsing have proposed XMSS [BDH11], the eXtended Merkle Signature Scheme, offering better efficiency and a modern security proof. Very recently, the stateless hash-based signature scheme SPHINCS was introduced [BHH15], with the intent of being easier to deploy in current applications. A reasonable next step toward introducing hash-based signatures would be to complete the specifications of the basic algorithms - LDWM, XMSS, SPHINCS and/or variants [Kaliski15].

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

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

This document is structured as follows. Notation is introduced in Section 2. Section 3 describes the WOTS+ signature system. MTS schemes are defined in Section 4: the eXtended Merkle Signature Scheme (XMSS) in Section 4.1, and its Multi-Tree variant (XMSS^MT) in Section 4.2. Parameter sets are described in Section 5. Section 6 describes the rationale behind choices in this note. The IANA registry for these signature systems is described in Section 7. Finally, security considerations are presented in Section 8.

1.1. CFRG Note on Post-Quantum Cryptography

All post-quantum algorithms documented by CFRG are today considered ready for experimentation and further engineering development (e.g. to establish the impact of performance and sizes on IETF protocols). However, at the time of writing, we do not have significant deployment experience with such algorithms.

Many of these algorithms come with specific restrictions, e.g. change of classical interface or less cryptanalysis of proposed parameters than established schemes. CFRG has consensus that all documents describing post-quantum technologies include the above paragraph and a clear additional warning about any specific restrictions, especially as those might affect use or deployment of the specific scheme. That guidance may be changed over time via document updates.

Addition for XMSS:

CFRG consensus is that we are confident in the cryptographic security of the signature schemes described in this document against quantum computers, given the current state of the research community's knowledge about quantum algorithms. Indeed, we are confident that the security of a significant part of the Internet could be made dependent on the signature schemes defined in this document, if developers take care of the following.

In contrast to traditional signature schemes, the signature schemes described in this document are stateful, meaning the secret key changes over time. If a secret key state is used twice, no cryptographic security guarantees remain. This is a new property that most developers will not be familiar with and requires careful handling of secret keys. Developers should not use the schemes described here except in systems that prevent the reuse of secret key states.

1.2. 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. We assume big-endian representation for any data types or structures.

2.2. Functions

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

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

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

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

2.3. 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.

++ : a++ denotes incrementing a by 1, i.e., a = a + 1.

-- : a-- denotes decrementing a by 1, i.e., a = a - 1.

<< : a << b denotes a logical left shift, i.e., a * 2^b.

>> : a >> b denotes a logical right shift, i.e. floor(a / 2^b).

The standard order of operations is used when evaluating arithmetic expressions.

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

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

A AND B denotes the bitwise logical conjunction operation.

A XOR B denotes the bitwise logical exclusive disjunction operation.

When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation. Similarly, B << i denotes the logical left-shift operation.

If X is an 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].

2.4. Integer to Byte Conversion

If x and y are non-negative integers, we define Z = toByte(x, y) to be the y-byte string containing the binary representation of x in big-endian byte-order.

2.5. Hash Function Address Scheme

The schemes described in this document randomize each hash function call. This means that aside of the initial message digest, for each hash function call a different key and different bitmask is used. These values are pseudorandomly generated using a pseudorandom function that takes a key SEED and a 32-byte address ADRS as input and outputs a n-byte value, where n is the security parameter. Here we explain the structure of address ADRS and propose setter methods to manipulate the address. We explain the generation of the addresses in the following sections where they are used.

The schemes in the next two sections use two kinds of hash functions parameterized by security parameter n. For the hash tree constructions, a hash function that maps an n-byte key and 2n-byte inputs to n-byte outputs is used. To randomize this function, 3n bytes are needed - n bytes for the key and 2n bytes for a bitmask.

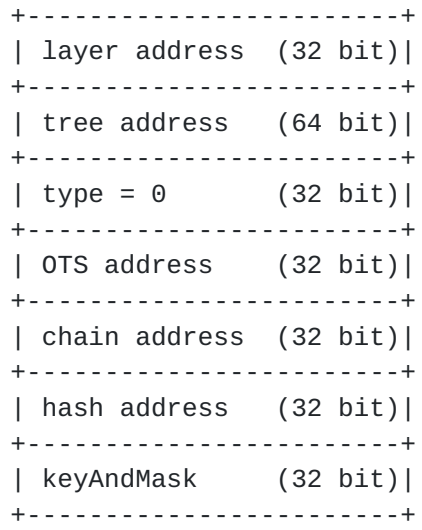
For the one-time signature scheme constructions, a hash function that maps n-byte keys and n-byte inputs to n-byte outputs is used. To randomize this function, 2n bytes are needed - n bytes for the key and n bytes for a bitmask. Consequently, three addresses are needed for the first function and two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

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

The structure of an address complies with word borders, with a word being 32 bits long in this context. Only the tree address is too long to fit a single word but matches a double word. An address is structured as follows. It always starts with a layer address of one word in the most significant bits, followed by a tree address of two words. Both addresses are needed for the multi-tree variant (see Section 4.2) and describe the position of a tree within a multi-tree. They are therefore set to zero in case of single-tree applications. For multi-tree hash-based signatures the layer address describes the height of a tree within the multi-tree starting from height zero for trees at the bottom layer. The tree address describes the position of a tree within a layer of a multi-tree starting with index zero for the leftmost tree. The next word defines the type of the address. It is set to 0 for an OTS address, to 1 for an L-tree address, and to 2 for a hash tree address. Whenever the type word of an address is changed, all following words should be initialized with 0 to prevent non-zero values in unused padding words.

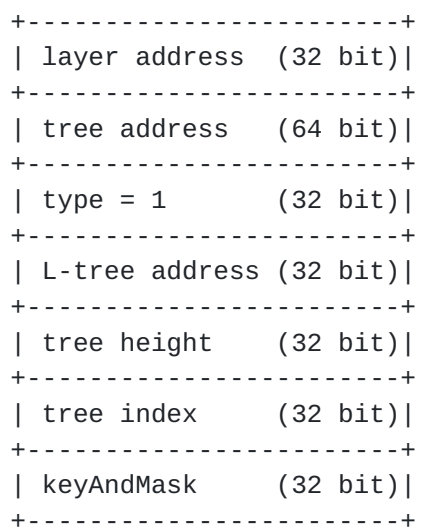
We first describe the OTS address case. In this case, the type word is followed by an OTS address word that encodes the index of the OTS key pair within the tree. The next word encodes the chain address followed by a word that encodes the address of the hash function call within the chain. The last word, called keyAndMask, is used to generate two different addresses for one hash function call. The word is set to zero to generate the key. To generate the n-byte bitmask, the word is set to one.

An OTS hash address



We now discuss the L-tree case, which means that the type word is set to one. In that case the type word is followed by an L-tree address word that encodes the index of the leaf computed with this L-tree. The next word encodes the height of the node inside the L-tree and the following word encodes the index of the node at that height, inside the L-tree. This time, the last word, keyAndMask, is used to generate three different addresses for one function call. The word is set to zero to generate the key. To generate the most significant n bytes of the 2n-byte bitmask, the word is set to one. The least significant bytes are generated using the address with the word set to two.

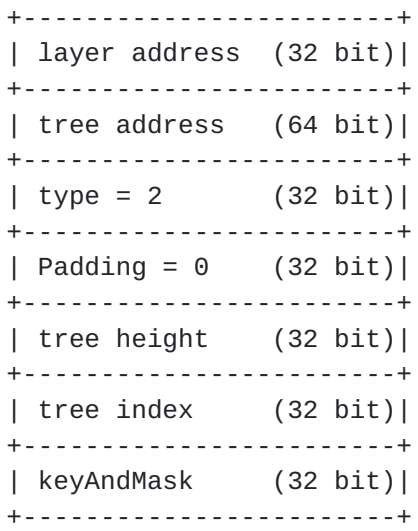
An L-tree address



We now describe the remaining type for the main tree hash addresses. In this case the type word is set to two, followed by a zero padding

of one word. The next word encodes the height of the tree node to be computed within the tree, followed by a word that encodes the index of this node at that height. As for the L-tree addresses, the last word, keyAndMask, is used to generate three different addresses for one function call. The word is set to zero to generate the key. To generate the most significant n bytes of the 2n-byte bitmask, the word is set to one. The least significant bytes are generated using the address with the word set to two.

A hash tree address



All fields within these addresses encode unsigned integers. When describing the generation of addresses we use setter-methods that take positive integers and set the bits of a field to the binary representation of that integer of the length of the field. We furthermore assume that the setType() method sets the four words following the type word to zero.

2.6. Strings of Base w Numbers

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

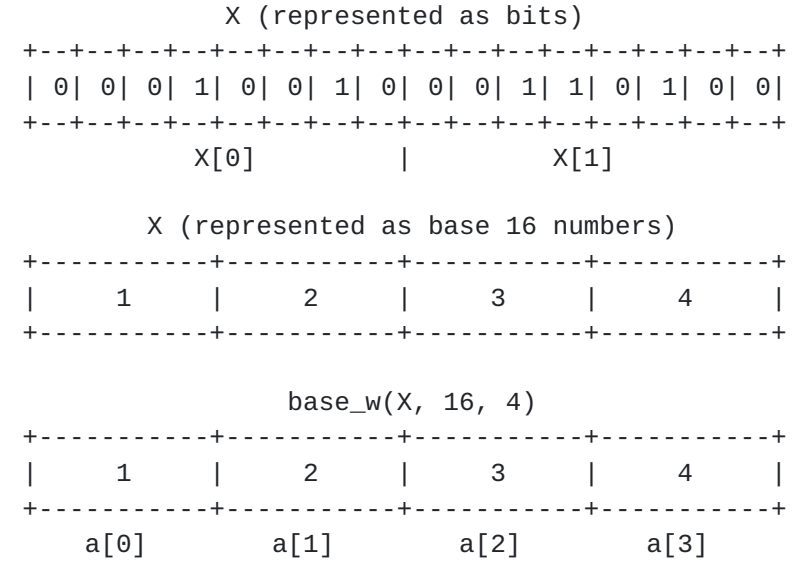
Algorithm 1: base_w

```
Input: len_X-byte string X, int w, output length out_len
Output: out_len int array basew

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

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

For example, if X is the (big-endian) byte string 0x1234, then base_w(X, 16, 4) returns the array a = {1, 2, 3, 4}.



2.7. Member Functions

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

Accordingly, setX(PK, X, Y) sets value X in PK to the value hold by Y. Since camelCase is used for member function names, a value z may be referred to as Z in the function name, e.g. getZ.

3. Primitives

3.1. WOTS+ One-Time Signatures

This section describes the WOTS+ one-time signature system, in a version similar to [Huelsing13]. WOTS+ is a one-time signature scheme; while a private key can be used to sign any message, each private key MUST be used only once to sign a single message. In particular, if a private 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. A description of the algorithms for key generation, signing and verification follows. Finally, pseudorandom key generation is discussed.

3.1.1. WOTS+ Parameters

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

n : the message length as well as the length of a private key, public key, or signature element in bytes.

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

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

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

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

WOTS+ parameters are implicitly included in algorithm inputs as needed.

3.1.1.1. WOTS+ Functions

The WOTS+ algorithm uses a keyed cryptographic hash function F. F accepts and returns byte strings of length n using keys of length n. Security requirements on F are discussed in Section 8. In addition, WOTS+ uses a pseudorandom function PRF. PRF takes as input an n-byte key and a 32-byte index and generates pseudorandom outputs of length n. Security requirements on PRF are discussed in Section 8.

3.1.2. WOTS+ Chaining Function

The chaining function (Algorithm 2) computes an iteration of F on an n-byte input using outputs of PRF. It takes an OTS hash address as input. This address will have the first six 32-bit words set to encode the address of this chain. In each iteration, PRF is used to generate a key for F and a bitmask that is XORed to the intermediate result before it is processed by F. In the following, ADRS is a 32-byte OTS hash address as specified in Section 2.5 and SEED is an n-byte string. To generate the keys and bitmasks, PRF is called with SEED as key and ADRS as input. The chaining function takes as input an n-byte string X, a start index i, a number of steps s, as well as ADRS and SEED. The chaining function returns as output the value obtained by iterating F for s times on input X, using the outputs of PRF.

Algorithm 2: chain - Chaining Function

```
Input: Input string X, start index i, number of steps s,
seed SEED, address ADRS
Output: value of F iterated s times on X

if ( s == 0 ) {
  return X;
}
if ( (i + s) > w - 1 ) {
  return NULL;
}
byte[n] tmp = chain(X, i, s - 1, SEED, ADRS);

ADRS.setHashAddress(i + s - 1);
ADRS.setKeyAndMask(0);
KEY = PRF(SEED, ADRS);
ADRS.setKeyAndMask(1);
BM = PRF(SEED, ADRS);

tmp = F(KEY, tmp XOR BM);
return tmp;
```

3.1.3. WOTS+ Private Key

The private key in WOTS+, denoted by sk (s for secret), is a length len array of n-byte strings. This private key MUST be only used to sign at most 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: WOTS_genSK - Generating a WOTS+ Private Key

```
Input: /
Output: WOTS+ private key sk

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

3.1.4. WOTS+ Public Key

A WOTS+ key pair defines a virtual structure that consists of len hash chains of length w. The len n-byte strings in the private 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 private key, the public key is also a length len array of n-byte strings. To compute the hash chain, the chaining function (Algorithm 2) is used. An OTS hash address ADRS and a seed SEED have to be provided by the calling algorithm. This address will encode the address of the WOTS+ key pair within a greater structure. Hence, a WOTS+ algorithm MUST NOT manipulate any other parts of ADRS than the last three 32-bit words. Please note that the SEED used here is public information also available to a verifier. The following pseudocode (Algorithm 4) describes an algorithm for generating the public key pk, where sk is the private key.

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

```
Input: WOTS+ private key sk, address ADRS, seed SEED
Output: WOTS+ public key pk

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

3.1.5. WOTS+ Signature Generation

A WOTS+ signature is a length len array of n-byte strings. The WOTS+ signature is generated by mapping a message to len integers between 0 and w - 1. To this end, the message is transformed into len_1 base w numbers using the base_w function defined in Section 2.6. Next, a checksum is computed and appended to the transformed message as len_2 base w numbers using the base_w function. Each of the base w integers is used to select a node from a different hash chain. The signature is formed by concatenating the selected nodes. An OTS hash address ADRS and a seed SEED have to be provided by the calling algorithm. This address will encode the address of the WOTS+ key pair within a greater structure. Hence, a WOTS+ algorithm MUST NOT manipulate any other parts of ADRS than the last three 32-bit words. Please note that the SEED used here is public information also available to a verifier. 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

```
  Input: Message M, WOTS+ private key sk, address ADRS, seed SEED
  Output: WOTS+ signature sig

  csum = 0;

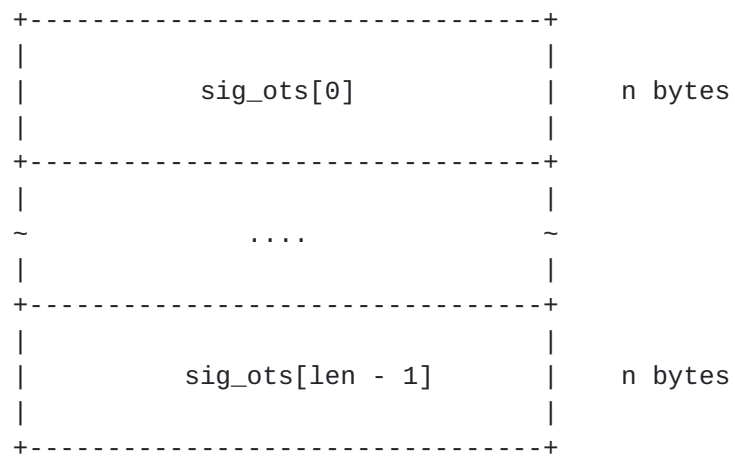
  // convert message to base w
  msg = base_w(M, w, len_1);

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

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

The data format for a signature is given below.

WOTS+ Signature



3.1.6. WOTS+ Signature Verification

In order to verify a signature sig on a message M, the verifier computes a WOTS+ public key value from the signature. This can be done by "completing" the chain computations starting from the signature values, using the base w values of the message hash and its checksum. This step, called WOTS_pkFromSig, is described below in Algorithm 6. The result of WOTS_pkFromSig is then compared to the given public key. If the values are equal, the signature is accepted. Otherwise, the signature MUST be rejected. An OTS hash address ADRS and a seed SEED have to be provided by the calling algorithm. This address will encode the address of the WOTS+ key pair within a greater structure. Hence, a WOTS+ algorithm MUST NOT manipulate any other parts of ADRS than the last three 32-bit words. Please note that the SEED used here is public information also available to a verifier.

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

```
Input: Message M, WOTS+ signature sig, address ADRS, seed SEED
Output: 'Temporary' WOTS+ public key tmp_pk

csum = 0;

// convert message to base w
msg = base_w(M, w, len_1);

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

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

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

3.1.7. Pseudorandom Key Generation

An implementation MAY use a cryptographically secure pseudorandom method to generate the private 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 private key elements from a random n-byte string is that only this n-byte string needs to be stored instead of the full private key. The key can be regenerated when needed. The suggested method from [BDH11] can be described using PRF. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This string S is stored as private key. The private key elements are computed as sk[i] = PRF(S, toByte(i, 32)) whenever needed. Please note that this seed S MUST be different from the seed SEED used to randomize the hash function calls. Also, this seed S MUST be kept secret.

4. Schemes

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

4.1. XMSS: eXtended Merkle Signature Scheme

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

To generate a key pair that can be used to sign 2^h messages, a tree of height h is used. XMSS is a stateful signature scheme, meaning that the private key changes with every signature generation. To prevent one-time private 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 private key contains an index that is updated with every signature generation, such that it contains the index of the next unused WOTS+ key pair.

A signature consists of the index of the used WOTS+ key pair, the WOTS+ signature on the message and the so-called authentication path. The latter is a vector of tree nodes that allow a verifier to compute a value for the root of the tree starting from a WOTS+ signature. A verifier computes the root value and compares it to the respective value in the XMSS public key. If they match, the signature is valid. The XMSS private key consists of all WOTS+ private keys and the current 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 the message digest as well as of each node

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

There are 2^h leaves in the tree.

For XMSS and XMSS^MT, private and public keys are denoted by SK (S for secret) and PK. For WOTS+, private and public keys are denoted by sk (s for secret) and pk, respectively. XMSS and XMSS^MT signatures are denoted by Sig. WOTS+ signatures are denoted by sig.

XMSS and XMSS^MT parameters are implicitly included in algorithm inputs as needed.

4.1.2. XMSS Hash Functions

Besides the cryptographic hash function F and the pseudorandom function PRF required by WOTS+, XMSS uses two more functions:

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

A cryptographic hash function H_msg. H_msg accepts 3n-byte keys and byte strings of arbitrary length and returns an n-byte string.

4.1.3. XMSS Private Key

An XMSS private key SK contains 2^h WOTS+ private keys, the leaf index idx of the next WOTS+ private key that has not yet been used, SK_PRF, an n-byte key to generate pseudorandom values for randomized message hashing, the n-byte value root, which is the root node of the tree and SEED, the n-byte public seed used to pseudorandomly generate bitmasks and hash function keys. Although root and SEED formally would be considered only part of the public key, they are needed e.g. for signature generation and hence are also required for functions that do not take the public key as input.

The leaf index idx is initialized to zero when the XMSS private key is created. The key SK_PRF MUST be sampled from a secure source of randomness that follows the uniform distribution. The WOTS+ private keys MUST be generated as described in Section 3.1. To reduce the private key size, a cryptographic pseudorandom method MAY be used as discussed at the end of this section. SEED is generated as a uniformly random n-byte string. Although SEED is public, it is critical for security that it is generated using a good entropy source. The root node is generated as described below in the section on key generation (Section 4.1.7). That section also contains an example algorithm for combined private and public key generation.

For the following algorithm descriptions, the existence of a method getWOTS_SK(SK, i) is assumed. This method takes as inputs an XMSS private key SK and an integer i and outputs the i^th WOTS+ private key of SK.

4.1.4. Randomized Tree Hashing

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

Algorithm 7: RAND_HASH

```
  Input:  n-byte value LEFT, n-byte value RIGHT, seed SEED,
          address ADRS
  Output: n-byte randomized hash

  ADRS.setKeyAndMask(0);
  KEY = PRF(SEED, ADRS);
  ADRS.setKeyAndMask(1);
  BM_0 = PRF(SEED, ADRS);
  ADRS.setKeyAndMask(2);
  BM_1 = PRF(SEED, ADRS);

  return H(KEY, (LEFT XOR BM_0) || (RIGHT XOR BM_1));
```

4.1.5. L-Trees

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

Algorithm 8: ltree

```
  Input: WOTS+ public key pk, address ADRS, seed SEED
  Output: n-byte compressed public key value pk[0]

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

4.1.6. TreeHash

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine treeHash (Algorithm 9) accepts an XMSS private key SK (including seed SEED), an unsigned integer s (the start index), an unsigned integer t (the target node height), and an address ADRS that encodes the address of the containing tree. For the height of a node within a tree counting starts with the leaves at height zero. The treeHash algorithm returns the root node of a tree of height t with the leftmost leaf being the hash of the WOTS+ pk with index s. It is REQUIRED that s % 2^t = 0, i.e. that the leaf at index s is a left most leaf of a sub-tree of height t. Otherwise the hash-addressing scheme fails. The treeHash algorithm described here uses a stack holding up to (t - 1) nodes, with the usual stack functions push() and pop(). We furthermore assume that the height of a node (an unsigned integer) is stored alongside a node's value (an n-byte string) on the stack.

Algorithm 9: treeHash

```
  Input: XMSS private key SK, start index s, target node height t,
         address ADRS
  Output: n-byte root node - top node on Stack

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

4.1.7. XMSS Key Generation

The XMSS key pair is computed as described in XMSS_keyGen (Algorithm 10). The XMSS public key PK consists of the root of the binary hash tree and the seed SEED, both also stored in SK. The root is computed using treeHash. For XMSS, there is only a single main tree. Hence, the used address is set to the all-zero string in the beginning. Note that we do not define any specific format or handling for the XMSS private key SK by introducing this algorithm. It relates to requirements described earlier and simply shows a basic but very inefficient example to initialize a private key.

Algorithm 10: XMSS_keyGen - Generate an XMSS key pair

```
Input: /
Output: XMSS private key SK, XMSS public key PK

// Example initialization for SK-specific contents
idx = 0;
for ( i = 0; i < 2^h; i++ ) {
  wots_sk[i] = WOTS_genSK();
}
initialize SK_PRF with a uniformly random n-byte string;
setSK_PRF(SK, SK_PRF);

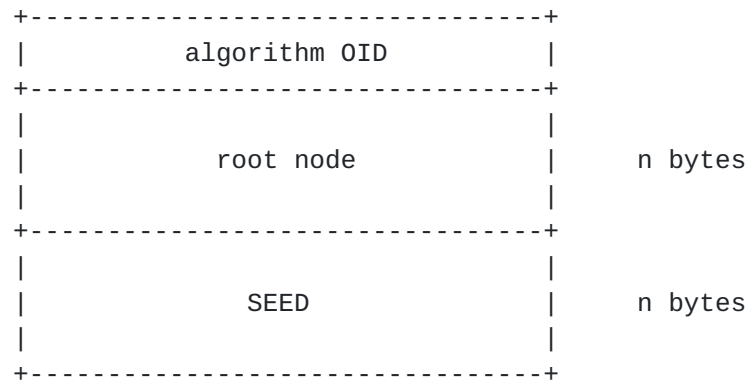
// Initialization for common contents
initialize SEED with a uniformly random n-byte string;
setSEED(SK, SEED);
setWOTS_SK(SK, wots_sk));
ADRS = toByte(0, 32);
root = treeHash(SK, 0, h, ADRS);

SK = idx || wots_sk || SK_PRF || root || SEED;
PK = root || SEED;
return (SK || PK);
```

The above is just an example algorithm. It is strongly RECOMMENDED to use pseudorandom key generation to reduce the private key size. Public and private key generation MAY be interleaved to save space. Especially, when a pseudorandom method is used to generate the private key, generation MAY be done when the respective WOTS+ key pair is needed by treeHash.

The format of an XMSS public key is given below.

XMSS Public Key



4.1.8. XMSS Signature

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

- the index idx_sig of the used WOTS+ key pair (4 bytes),
- a byte string r used for randomized message hashing (n bytes),
- a WOTS+ signature sig_ots (len * n bytes),
- the so-called authentication path 'auth' for the leaf associated with the used WOTS+ key pair (h * n bytes).

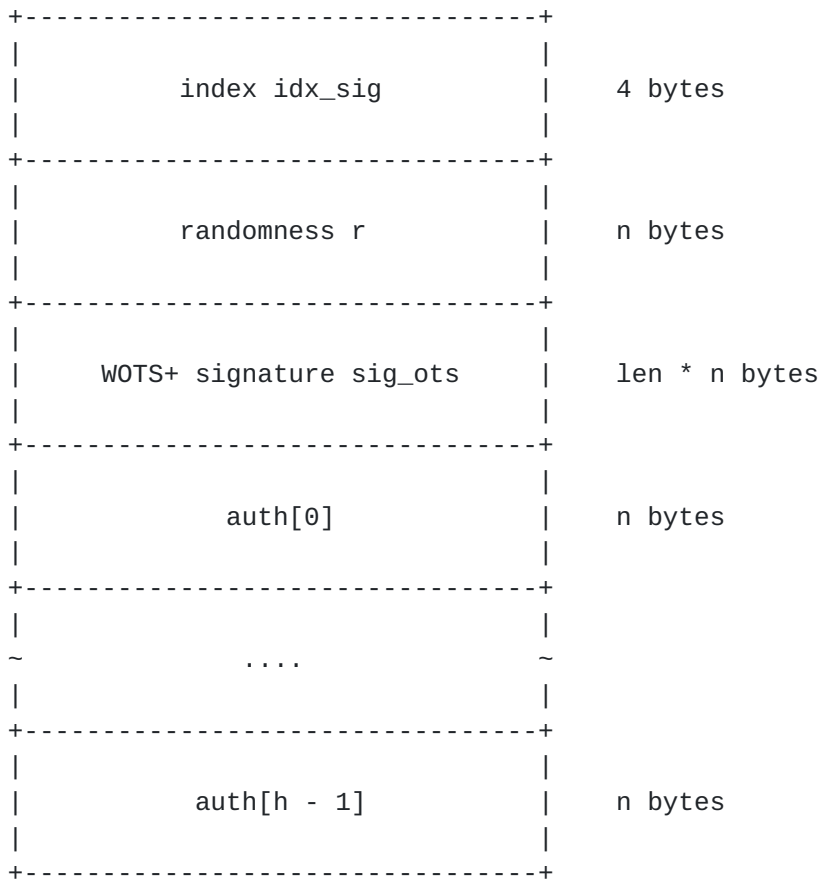
The authentication path is an array of h n-byte strings. It contains the siblings of the nodes on the path from the used leaf to the root. It does not contain the nodes on the path itself. These nodes are needed by a verifier to compute a root node for the tree from the WOTS+ public key. A node Node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

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

The computation of the authentication path is discussed in Section 4.1.9.

The data format for a signature is given below.

XMSS Signature



4.1.9. XMSS Signature Generation

To compute the XMSS signature of a message M with an XMSS private key, the signer first computes a randomized message digest using a random value r, idx_sig, the index of the WOTS+ key pair to be used, and the root value from the public key as key. Then a WOTS+ signature of the message digest is computed using the next unused WOTS+ private key. Next, the authentication path is computed. Finally, the private 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 the respective nodes. The least storage-intensive alternative is to recompute all nodes for each signature online using the treeHash algorithm (Algorithm 9). There exist several algorithms in between, with different time/storage trade-

offs. For an overview, see [BDS09]. A further approach can be found in [KMN14]. 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. The following version of buildAuth is just given for completeness. It is a simple example for understanding, but extremely inefficient. The use of one of the alternative algorithms is strongly RECOMMENDED.

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

(Example) buildAuth - Compute the authentication path for the i^th WOTS+ key pair

```
  Input: XMSS private key SK, WOTS+ key pair index i, ADRS
  Output: Authentication path auth

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

We split the description of the signature generation into two main algorithms. The first one, treeSig (Algorithm 11), generates the main part of an XMSS signature and is also used by the multi-tree version XMSS^MT. XMSS_sign (Algorithm 12) calls treeSig but handles message compression before and the private key update afterwards.

The algorithm treeSig (Algorithm 11) described below calculates the WOTS+ signature on an n-byte message and the corresponding authentication path. treeSig takes as inputs an n-byte message M', an XMSS private key SK, and an address ADRS. It returns the concatenation of the WOTS+ signature sig_ots and authentication path auth.

Algorithm 11: treeSig - Generate a WOTS+ signature on a message with corresponding authentication path

```
Input: n-byte message M', XMSS private key SK, ADRS
Output: Concatenation of WOTS+ signature sig_ots and
        authentication path auth

idx_sig = getIdx(SK);
auth = buildAuth(SK, idx_sig, ADRS);
ADRS.setType(0);   // Type = OTS hash address
ADRS.setOTSAddress(idx_sig);
sig_ots = WOTS_sign(getWOTS_SK(SK, idx_sig),
                    M', getSEED(SK), ADRS);
Sig = (sig_ots || auth);
return Sig;
```

The algorithm XMSS_sign (Algorithm 12) described below calculates an updated private key SK and a signature on a message M. XMSS_sign takes as inputs a message M of arbitrary length, and an XMSS private key SK. It returns the byte string containing the concatenation of the updated private key SK and the signature Sig.

Algorithm 12: XMSS_sign - Generate an XMSS signature and update the XMSS private key

```
Input: Message M, XMSS private key SK
Output: Updated SK, XMSS signature Sig

idx_sig = getIdx(SK);
ADRS = toByte(0, 32);
byte[n] r = PRF(getSK_PRF(SK), toByte(idx_sig, 32));
byte[n] M' = H_msg(r || getRoot(SK) || (toByte(idx_sig, n)), M);
Sig = (idx_sig || r || treeSig(M', SK, ADRS));
setIdx(SK, idx_sig + 1);
return (SK || Sig);
```

4.1.10. XMSS Signature Verification

An XMSS signature is verified by first computing the message digest using randomness r, index idx_sig, the root from PK and message M. Then the used WOTS+ public key pk_ots is computed from the WOTS+ signature using WOTS_pkFromSig. The WOTS+ public key in turn is used to compute the corresponding leaf using an L-tree. The leaf, together with index idx_sig and authentication path auth is used to compute an alternative root value for the tree. 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.

As for signature generation, we split verification into two parts to allow for reuse in the XMSS^MT description. The steps also needed for XMSS^MT are done by the function XMSS_rootFromSig (Algorithm 13). XMSS_verify (Algorithm 14) calls XMSS_rootFromSig as a subroutine and handles the XMSS-specific steps.

The main part of XMSS signature verification is done by the function XMSS_rootFromSig (Algorithm 13) described below. XMSS_rootFromSig takes as inputs an index idx_sig, a WOTS+ signature sig_ots, an authentication path auth, an n-byte message M', seed SEED, and address ADRS. XMSS_rootFromSig returns an n-byte string holding the value of the root of a tree defined by the input data.

Algorithm 13: XMSS_rootFromSig - Compute a root node from a tree signature

```
  Input: index idx_sig, WOTS+ signature sig_ots, authentication path
         auth, n-byte message M', seed SEED, address ADRS
  Output: n-byte root value node[0]

  ADRS.setType(0);   // Type = OTS hash address
  ADRS.setOTSAddress(idx_sig);
  pk_ots = WOTS_pkFromSig(sig_ots, M', SEED, ADRS);
  ADRS.setType(1);   // Type = L-tree address
  ADRS.setLTreeAddress(idx_sig);
  byte[n][2] node;
  node[0] = ltree(pk_ots, SEED, ADRS);
  ADRS.setType(2);   // Type = hash tree address
  ADRS.setTreeIndex(idx_sig);
  for ( k = 0; k < h; k++ ) {
    ADRS.setTreeHeight(k);
    if ( (floor(idx_sig / (2^k)) % 2) == 0 ) {
      ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
      node[1] = RAND_HASH(node[0], auth[k], SEED, ADRS);
    } else {
      ADRS.setTreeIndex(ADRS.getTreeIndex() - 1 / 2);
      node[1] = RAND_HASH(auth[k], node[0], SEED, ADRS);
    }
    node[0] = node[1];
  }
  return node[0];
```

The full XMSS signature verification is depicted below (Algorithm 14). It handles message compression, delegates the root computation to XMSS_rootFromSig, and compares the result to the value in the public key. XMSS_verify takes an XMSS signature Sig, a message M, and an XMSS public key PK. XMSS_verify returns true if and only if

Sig is a valid signature on M under public key PK. Otherwise, it returns false.

Algorithm 14: XMSS_verify - Verify an XMSS signature using the corresponding XMSS public key and a message

```
  Input: XMSS signature Sig, message M, XMSS public key PK
  Output: Boolean

  ADRS = toByte(0, 32);
  byte[n] M' = H_msg(r || getRoot(PK) || (toByte(idx_sig, n)), M);

  byte[n] node = XMSS_rootFromSig(idx_sig, sig_ots, auth, M',
                                  getSEED(PK), ADRS);
  if ( node == getRoot(PK) ) {
    return true;
  } else {
    return false;
  }
```

4.1.11. Pseudorandom Key Generation

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

For XMSS a similar method than the one used for WOTS+ can be used. The suggested method from [BDH11] can be described using PRF. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This seed S MUST NOT be confused with the public seed SEED. The seed S MUST be independent of SEED and as it is the main secret, it MUST be kept secret. This seed S is used to generate an n-byte value S_ots for each WOTS+ key pair. The n-byte value S_ots can then be used to compute the respective WOTS+ private key using the method described in Section 3.1.7. The seeds for the WOTS+ key pairs are computed as S_ots[i] = PRF(S, toByte(i, 32)) where i is the index of the WOTS+ key pair. An advantage of this method is that a WOTS+ key can be computed using only len + 1 evaluations of PRF when S is given.

4.1.12. Free Index Handling and Partial Private Keys

Some applications might require to work with partial private keys or copies of private 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 private keys or copies of a private 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, a so-called hypertree. 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.

4.2.2. XMSS^MT Key generation

An XMSS^MT private key SK_MT (S for secret) consists of one reduced XMSS private key for each XMSS tree. These reduced XMSS private keys just contain the WOTS+ private keys corresponding to that XMSS key pair and no pseudorandom function key, no index, no public seed, no root node. Instead, SK_MT contains a single n-byte pseudorandom function key SK_PRF, a single (ceil(h / 8))-byte index idx_MT, a single n-byte seed SEED, and a single root value root which is the root of the single tree on the top layer. 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 reduced XMSS private keys MUST either be generated as described in Section 4.1.3 or using a cryptographic pseudorandom method as discussed at the end of this section. As for XMSS, the PRF key SK_PRF MUST be sampled from a secure source of randomness that follows the uniform distribution. SEED is generated as a uniformly random n-byte string. Although SEED is public, it is critical for security that it is generated using a good entropy source. The root is the root node of the single XMSS tree on the top layer. Its computation is explained below. As for XMSS, root and SEED are public information and would classically be considered part of the public key. However, as both are needed for signing, which only takes the private key, they are also part of SK_MT.

This document does not define any specific format for the XMSS^MT private key SK_MT as it is not required for interoperability. The algorithm descriptions below use a function getXMSS_SK(SK, x, y) that outputs the reduced private key of the x^th XMSS tree on the y^th layer.

The XMSS^MT public key PK_MT contains the root of the single XMSS tree on layer d - 1 and the seed SEED. These are the same values as in the private key SK_MT. The pseudorandom function PRF keyed with SEED is used to generate the bitmasks and keys for all XMSS trees. XMSSMT_keyGen (Algorithm 15) shows example pseudocode to generate SK_MT and PK_MT. The n-byte root node of the top layer tree is computed using treeHash. The algorithm XMSSMT_keyGen outputs an XMSS^MT private key SK_MT and an XMSS^MT public key PK_MT. The algorithm below gives an example of how the reduced XMSS private keys can be generated. However, any of the above mentioned ways is acceptable as long as the cryptographic strength of the used method matches or superseeds that of the used XMSS^MT parameter set.

Algorithm 15: XMSSMT_keyGen - Generate an XMSS^MT key pair

```
Input: /
Output: XMSS^MT private key SK_MT, XMSS^MT public key PK_MT

// Example initialization
idx_MT = 0;
setIdx(SK_MT, idx_MT);
initialize SK_PRF with a uniformly random n-byte string;
setSK_PRF(SK_MT, SK_PRF);
initialize SEED with a uniformly random n-byte string;
setSEED(SK_MT, SEED);

// generate reduced XMSS private keys
ADRS = toByte(0, 32);
for ( layer = 0; layer < d; layer++ ) {
   ADRS.setLayerAddress(layer);
   for ( tree = 0; tree <
         (1 << ((d - 1 - layer) * (h / d)));
         tree++ ) {
      ADRS.setTreeAddress(tree);
      for ( i = 0; i < 2^h; i++ ) {
         WOTS_genSK(wots_sk[i]);
      }
      setXMSS_SK(SK_MT, wots_sk, tree, layer);
   }
}

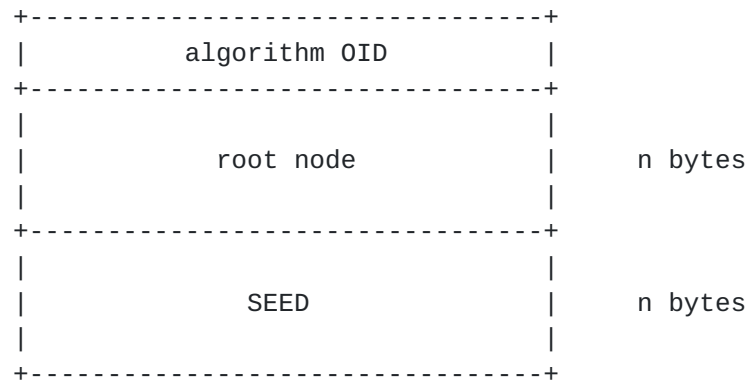
SK = getXMSS_SK(SK_MT, 0, d - 1);
setSEED(SK, SEED);
root = treeHash(SK, 0, h / d, ADRS);
setRoot(SK_MT, root);

PK_MT = (root || SEED);
return (SK_MT || PK_MT);
```

The above is just an example algorithm. It is strongly RECOMMENDED to use pseudorandom key generation to reduce the private key size. Public and private key generation MAY be interleaved to save space. Especially, when a pseudorandom method is used to generate the private key, generation MAY be delayed to the point when the respective WOTS+ key pair is needed by another algorithm.

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

XMSS^MT Public Key



4.2.3. XMSS^MT Signature

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

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

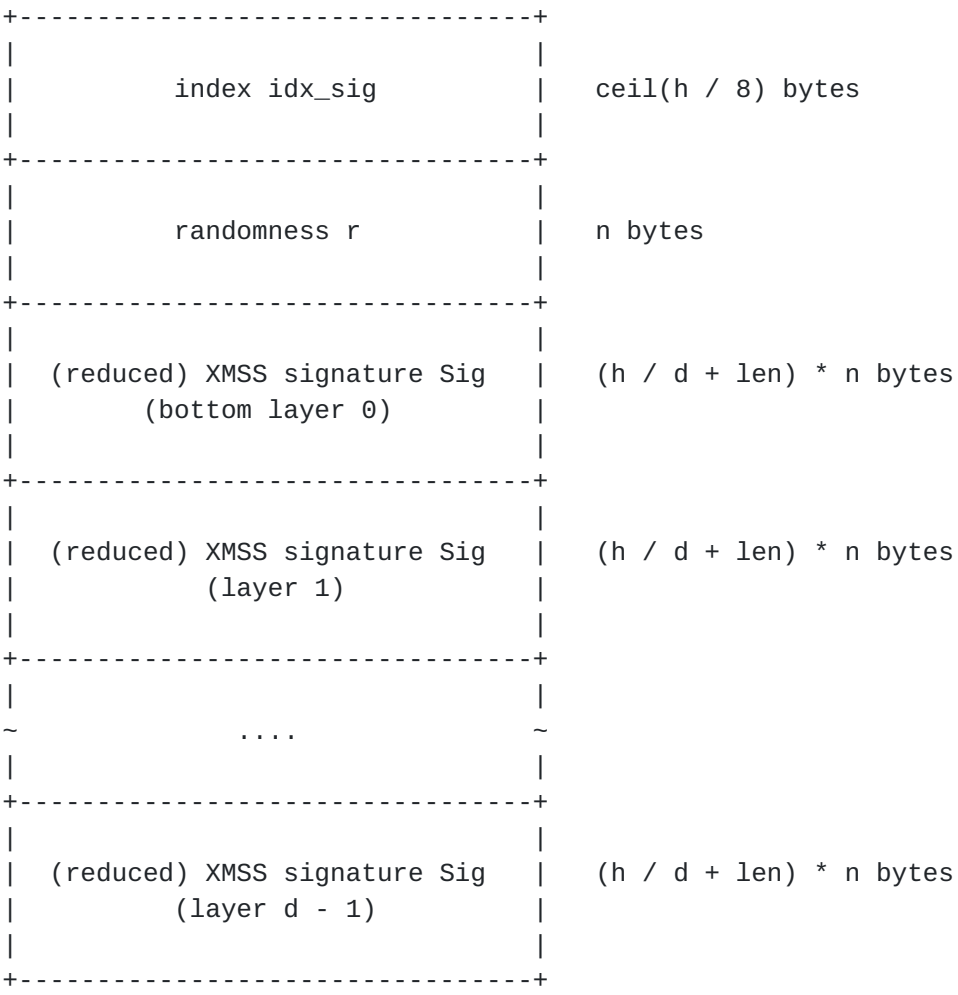
a byte string r used for randomized message hashing (n bytes),

d reduced XMSS signatures ((h / d + len) * n bytes each).

The reduced XMSS signatures only contain a WOTS+ signature sig_ots and an authentication path auth. They contain no index idx and no byte string r.

The data format for a signature is given below.

XMSS^MT signature



4.2.4. XMSS^MT Signature Generation

To compute the XMSS^MT signature Sig_MT of a message M using an XMSS^MT private key SK_MT, XMSSMT_sign (Algorithm 16) described below uses treeSig as defined in Section 4.1.9. First, the signature index is set to idx_sig. Next, PRF is used to compute a pseudorandom n-byte string r. This n-byte string, idx_sig, and the root node from PK_MT are then used to compute a randomized message digest of length n. 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 16: XMSSMT_sign - Generate an XMSS^MT signature and update the XMSS^MT private key

```
  Input: Message M, XMSS^MT private key SK_MT
  Output: Updated SK_MT, signature Sig_MT

  // Init
  ADRS = toByte(0, 32);
  SEED = getSEED(SK_MT);
  SK_PRF = getSK_PRF(SK_MT);
  idx_sig = getIdx(SK_MT);

  // Update SK_MT
  setIdx(SK_MT, idx_sig + 1);

  // Message compression
  byte[n] r = PRF(SK_PRF, toByte(idx_sig, 32));
  byte[n] M' = H_msg(r || getRoot(SK_MT) || (toByte(idx_sig, n)), M);

  // Sign
  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 || getXMSS_SK(SK_MT, idx_tree, 0) || SK_PRF
        || toByte(0, n) || SEED;
  ADRS.setLayerAddress(0);
  ADRS.setTreeAddress(idx_tree);
  Sig_tmp = treeSig(M', SK, ADRS);
  Sig_MT = Sig_MT || r || Sig_tmp;
  for ( j = 1; j < d; j++ ) {
     root = treeHash(SK, 0, h / d, ADRS);
     idx_leaf = (h / d) least significant bits of idx_tree;
     idx_tree = (h - j * (h / d)) most significant bits of idx_tree;
     SK = idx_leaf || getXMSS_SK(SK_MT, idx_tree, j) || SK_PRF
            || toByte(0, n) || SEED;
     ADRS.setLayerAddress(j);
     ADRS.setTreeAddress(idx_tree);
     Sig_tmp = treeSig(root, SK, ADRS);
     Sig_MT = Sig_MT || Sig_tmp;
  }
  return SK_MT || Sig_MT;
```

Algorithm 16 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].

4.2.5. XMSS^MT Signature Verification

XMSS^MT signature verification (Algorithm 17) can be summarized as d XMSS signature verifications with small changes. First, the message is hashed. The XMSS signatures are then all on n-byte values. Second, instead of comparing the computed root node to a given value, a signature on this root node 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. The function getXMSSSignature(Sig_MT, i) returns the ith reduced XMSS signature from the XMSS^MT signature Sig_MT. XMSSMT_verify takes as inputs an XMSS^MT signature Sig_MT, a message M and a public key PK_MT. XMSSMT_verify returns true if and only if Sig_MT is a valid signature on M under public key PK_MT. Otherwise, it returns false.

Algorithm 17: XMSSMT_verify - Verify an XMSS^MT signature Sig_MT on a message M using an XMSS^MT public key PK_MT

```
  Input: XMSS^MT signature Sig_MT, message M,
         XMSS^MT public key PK_MT
  Output: Boolean

  idx_sig = getIdx(Sig_MT);
  SEED = getSEED(PK_MT);
  ADRS = toByte(0, 32);

  byte[n] M' = H_msg(getR(Sig_MT) || getRoot(PK_MT)
                     || (toByte(idx_sig, n)), M);

  unsigned int idx_leaf
               = (h / d) least significant bits of idx_sig;
  unsigned int idx_tree
               = (h - h / d) most significant bits of idx_sig;
  Sig' = getXMSSSignature(Sig_MT, 0);
  ADRS.setLayerAddress(0);
  ADRS.setTreeAddress(idx_tree);
  byte[n] node = XMSS_rootFromSig(idx_leaf, getSig_ots(Sig'),
                                  getAuth(Sig'), M', SEED, ADRS);
  for ( j = 1; j < d; j++ ) {
     idx_leaf = (h / d) least significant bits of idx_tree;
     idx_tree = (h - j * h / d) most significant bits of idx_tree;
     Sig' = getXMSSSignature(Sig_MT, j);
     ADRS.setLayerAddress(j);
     ADRS.setTreeAddress(idx_tree);
     node = XMSS_rootFromSig(idx_leaf, getSig_ots(Sig'),
                             getAuth(Sig'), node, SEED, ADRS);
  }
  if ( node == getRoot(PK_MT) ) {
    return true;
  } else {
    return false;
  }
```

4.2.6. Pseudorandom Key Generation

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

For XMSS^MT a method similar to that for XMSS and WOTS+ can be used. The method uses PRF. 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 private key using the method described in Section 4.1.11. Let S[x][y] be the seed for the x^th XMSS private key on layer y. The seeds are computed as S[x][y] = PRF(PRF(S, toByte(y, 32)), toByte(x, 32)).

4.2.7. Free Index Handling and Partial Private Keys

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

5. Parameter Sets

This section provides a basic set of parameter sets which are assumed to cover most relevant applications. Parameter sets for two classical security levels are defined. Parameters with n = 32 provide a classical security level of 256 bits. Parameters with n = 64 provide a classical security level of 512 bits. Considering quantum-computer-aided attacks, these output sizes yield post-quantum security of 128 and 256 bits, respectively.

For the n = 32 and n = 64 settings, we give parameters that use SHA2-256, SHA2-512 as defined in [FIPS180], and the SHA3/Keccak-based extendable output functions SHAKE-128, SHAKE-256 as defined in [FIPS202]. The parameter sets using SHA2-256 are mandatory for deployment and therefore MUST be provided by any implementation. The remaining parameter sets specified in this document are OPTIONAL.

SHA2 does not provide a keyed-mode itself. To implement the keyed hash functions the following is used for SHA2 with n = 32:

F: SHA2-256(toByte(0, 32) || KEY || M),

H: SHA2-256(toByte(1, 32) || KEY || M),

H_msg: SHA2-256(toByte(2, 32) || KEY || M),

PRF: SHA2-256(toByte(3, 32) || KEY || M).

Accordingly, for SHA2 with n = 64 we use:

F: SHA2-512(toByte(0, 64) || KEY || M),

H: SHA2-512(toByte(1, 64) || KEY || M),

H_msg: SHA2-512(toByte(2, 64) || KEY || M),

PRF: SHA2-512(toByte(3, 64) || KEY || M).

The n-byte padding is used for two reasons. First, it is necessary that the internal compression function takes 2n-byte blocks but keys are n and 3n bytes long. Second, the padding is used to achieve independence of the different function families. Finally, for the PRF no full-fledged HMAC is needed as the message length is fixed. For that reason the simpler construction above suffices.

Similar constructions are used with SHA3. To implement the keyed hash functions the following is used for SHA3 with n = 32:

F: SHAKE128(toByte(0, 32) || KEY || M, 256),

H: SHAKE128(toByte(1, 32) || KEY || M, 256),

H_msg: SHAKE128(toByte(2, 32) || KEY || M, 256),

PRF: SHAKE128(toByte(3, 32) || KEY || M, 256).

Accordingly, for SHA3 with n = 64 we use:

F: SHAKE256(toByte(0, 64) || KEY || M, 512),

H: SHAKE256(toByte(1, 64) || KEY || M, 512),

H_msg: SHAKE256(toByte(2, 64) || KEY || M, 512),

PRF: SHAKE256(toByte(3, 64) || KEY || M, 512).

We use n-bytes for domain separation for consistency with the SHA2 implementations.

5.1. WOTS+ Parameters

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

Name	F / PRF	n	w	len
REQUIRED:				
WOTSP_SHA2-256_W16	SHA2-256	32	16	67
OPTIONAL:				
WOTSP_SHA2-512_W16	SHA2-512	64	16	131
WOTSP_SHAKE128_W16	SHAKE128	32	16	67
WOTSP_SHAKE256_W16	SHAKE256	64	16	131

Table 1

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

5.2. XMSS Parameters

To fully describe an XMSS signature method, the parameters n, w, and h, as well as the functions F, H, H_msg, and PRF MUST be specified. This section defines different XMSS signature systems, each of which is identified by a name. We define parameter sets that implement the functions using SHA2 and SHA3 for n = 32 and n = 64 as described above.

Name	Functions	n	w	len	h
REQUIRED:					
XMSS_SHA2-256_W16_H10	SHA2-256	32	16	67	10
XMSS_SHA2-256_W16_H16	SHA2-256	32	16	67	16
XMSS_SHA2-256_W16_H20	SHA2-256	32	16	67	20
OPTIONAL:					
XMSS_SHA2-512_W16_H10	SHA2-512	64	16	131	10
XMSS_SHA2-512_W16_H16	SHA2-512	64	16	131	16
XMSS_SHA2-512_W16_H20	SHA2-512	64	16	131	20
XMSS_SHAKE128_W16_H10	SHAKE128	32	16	67	10
XMSS_SHAKE128_W16_H16	SHAKE128	32	16	67	16
XMSS_SHAKE128_W16_H20	SHAKE128	32	16	67	20
XMSS_SHAKE256_W16_H10	SHAKE256	64	16	131	10
XMSS_SHAKE256_W16_H16	SHAKE256	64	16	131	16
XMSS_SHAKE256_W16_H20	SHAKE256	64	16	131	20

Table 2

The XDR formats for XMSS are listed in Appendix B.

5.3. XMSS^MT Parameters

To fully describe an XMSS^MT signature method, the parameters n, w, h, and d, as well as the functions F, H, H_msg, and PRF MUST be specified. This section defines several XMSS^MT signature systems, each of which is identified by a name. We define parameter sets that implement the functions using SHA2 and SHA3 for n = 32 and n = 64 as described above.

Name	Functions	n	w	len	h	d

REQUIRED:						
XMSSMT_SHA2-256_W16_H20_D2	SHA2-256	32	16	67	20	2
XMSSMT_SHA2-256_W16_H20_D4	SHA2-256	32	16	67	20	4
XMSSMT_SHA2-256_W16_H40_D2	SHA2-256	32	16	67	40	2
XMSSMT_SHA2-256_W16_H40_D4	SHA2-256	32	16	67	40	4
XMSSMT_SHA2-256_W16_H40_D8	SHA2-256	32	16	67	40	8
XMSSMT_SHA2-256_W16_H60_D3	SHA2-256	32	16	67	60	3
XMSSMT_SHA2-256_W16_H60_D6	SHA2-256	32	16	67	60	6
XMSSMT_SHA2-256_W16_H60_D12	SHA2-256	32	16	67	60	12
OPTIONAL:						
XMSSMT_SHA2-512_W16_H20_D2	SHA2-512	64	16	131	20	2
XMSSMT_SHA2-512_W16_H20_D4	SHA2-512	64	16	131	20	4
XMSSMT_SHA2-512_W16_H40_D2	SHA2-512	64	16	131	40	2
XMSSMT_SHA2-512_W16_H40_D4	SHA2-512	64	16	131	40	4
XMSSMT_SHA2-512_W16_H40_D8	SHA2-512	64	16	131	40	8
XMSSMT_SHA2-512_W16_H60_D3	SHA2-512	64	16	131	60	3
XMSSMT_SHA2-512_W16_H60_D6	SHA2-512	64	16	131	60	6
XMSSMT_SHA2-512_W16_H60_D12	SHA2-512	64	16	131	60	12
XMSSMT_SHAKE128_W16_H20_D2	SHAKE128	32	16	67	20	2
XMSSMT_SHAKE128_W16_H20_D4	SHAKE128	32	16	67	20	4
XMSSMT_SHAKE128_W16_H40_D2	SHAKE128	32	16	67	40	2
XMSSMT_SHAKE128_W16_H40_D4	SHAKE128	32	16	67	40	4
XMSSMT_SHAKE128_W16_H40_D8	SHAKE128	32	16	67	40	8
XMSSMT_SHAKE128_W16_H60_D3	SHAKE128	32	16	67	60	3

XMSSMT_SHAKE128_W16_H60_D6	SHAKE128	32	16	67	60	6
XMSSMT_SHAKE128_W16_H60_D12	SHAKE128	32	16	67	60	12
XMSSMT_SHAKE256_W16_H20_D2	SHAKE256	64	16	131	20	2
XMSSMT_SHAKE256_W16_H20_D4	SHAKE256	64	16	131	20	4
XMSSMT_SHAKE256_W16_H40_D2	SHAKE256	64	16	131	40	2
XMSSMT_SHAKE256_W16_H40_D4	SHAKE256	64	16	131	40	4
XMSSMT_SHAKE256_W16_H40_D8	SHAKE256	64	16	131	40	8
XMSSMT_SHAKE256_W16_H60_D3	SHAKE256	64	16	131	60	3
XMSSMT_SHAKE256_W16_H60_D6	SHAKE256	64	16	131	60	6
XMSSMT_SHAKE256_W16_H60_D12	SHAKE256	64	16	131	60	12

Table 3

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

6. Rationale

The goal of this note is to describe the WOTS+, XMSS and XMSS^MT algorithms following the scientific literature. Other signature methods are out of scope and may be an interesting follow-up work. The description is done in a modular way that allows to base a description of stateless hash-based signature algorithms like SPHINCS [BHH15] on it.

The draft slightly deviates from the scientific literature using a tweak that prevents multi-user / multi-target attacks against H_msg. To this end, the public key as well as the index of the used one-time key pair become part of the hash function key. Thereby we achieve a domain separation that forces an attacker to decide which hash value to attack.

For the generation of the randomness used for randomized message hashing, we apply a PRF, keyed with a secret value, to the index of the used one-time key pair instead of the message. The reason is that this requires to process the message only once instead of twice. For long messages this improves speed and simplifies implementations on resource constrained devices that cannot hold the entire message in storage.

We give one mandatory set of parameters using SHA2-256. The reasons are twofold. On the one hand, SHA2-256 is available on most platforms today and part of most cryptographic libraries. On the other hand, a 256-bit hash function leads parameters that provides 128 bit of security even against quantum-computer-aided attacks. A post-quantum security level of 256 bit seems overly conservative. However, to prepare for possible cryptanalytic breakthroughs, we also provide OPTIONAL parameter sets using the less common SHA2-512, SHAKE-256, and SHAKE-512 functions.

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

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

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

7. IANA Considerations

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

a short name, such as "XMSS_SHA2-256_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 [RFC5226].

The WOTS+ registry is as follows.

Name	Reference	Numeric Identifier
WOTSP_SHA2-256_W16	Section 5.1	0x01000001
WOTSP_SHA2-512_W16	Section 5.1	0x02000002
WOTSP_SHAKE128_W16	Section 5.1	0x03000003
WOTSP_SHAKE256_W16	Section 5.1	0x04000004

Table 4

The XMSS registry is as follows.

Name	Reference	Numeric Identifier
XMSS_SHA2-256_W16_H10	Section 5.2	0x01000001
XMSS_SHA2-256_W16_H16	Section 5.2	0x02000002
XMSS_SHA2-256_W16_H20	Section 5.2	0x03000003
XMSS_SHA2-512_W16_H10	Section 5.2	0x04000004
XMSS_SHA2-512_W16_H16	Section 5.2	0x05000005
XMSS_SHA2-512_W16_H20	Section 5.2	0x06000006
XMSS_SHAKE128_W16_H10	Section 5.2	0x07000007
XMSS_SHAKE128_W16_H16	Section 5.2	0x08000008
XMSS_SHAKE128_W16_H20	Section 5.2	0x09000009
XMSS_SHAKE256_W16_H10	Section 5.2	0x0a00000a
XMSS_SHAKE256_W16_H16	Section 5.2	0x0b00000b
XMSS_SHAKE256_W16_H20	Section 5.2	0x0c00000c

Table 5

The XMSS^MT registry is as follows.

Name	Reference	Numeric Identifier
XMSSMT_SHA2-256_W16_H20_D2	Section 5.3	0x01000001
XMSSMT_SHA2-256_W16_H20_D4	Section 5.3	0x02000002
XMSSMT_SHA2-256_W16_H40_D2	Section 5.3	0x03000003
XMSSMT_SHA2-256_W16_H40_D4	Section 5.3	0x04000004
XMSSMT_SHA2-256_W16_H40_D8	Section 5.3	0x05000005
XMSSMT_SHA2-256_W16_H60_D3	Section 5.3	0x06000006
XMSSMT_SHA2-256_W16_H60_D6	Section 5.3	0x07000007

XMSSMT_SHA2-256_W16_H60_D12	Section 5.3	0x08000008
XMSSMT_SHA2-512_W16_H20_D2	Section 5.3	0x09000009
XMSSMT_SHA2-512_W16_H20_D4	Section 5.3	0x0a00000a
XMSSMT_SHA2-512_W16_H40_D2	Section 5.3	0x0b00000b
XMSSMT_SHA2-512_W16_H40_D4	Section 5.3	0x0c00000c
XMSSMT_SHA2-512_W16_H40_D8	Section 5.3	0x0d00000d
XMSSMT_SHA2-512_W16_H60_D3	Section 5.3	0x0e00000e
XMSSMT_SHA2-512_W16_H60_D6	Section 5.3	0x0f00000f
XMSSMT_SHA2-512_W16_H60_D12	Section 5.3	0x01010101
XMSSMT_SHAKE128_W16_H20_D2	Section 5.3	0x02010102
XMSSMT_SHAKE128_W16_H20_D4	Section 5.3	0x03010103
XMSSMT_SHAKE128_W16_H40_D2	Section 5.3	0x04010104
XMSSMT_SHAKE128_W16_H40_D4	Section 5.3	0x05010105
XMSSMT_SHAKE128_W16_H40_D8	Section 5.3	0x06010106
XMSSMT_SHAKE128_W16_H60_D3	Section 5.3	0x07010107
XMSSMT_SHAKE128_W16_H60_D6	Section 5.3	0x08010108
XMSSMT_SHAKE128_W16_H60_D12	Section 5.3	0x09010109
XMSSMT_SHAKE256_W16_H20_D2	Section 5.3	0x0a01010a
XMSSMT_SHAKE256_W16_H20_D4	Section 5.3	0x0b01010b
XMSSMT_SHAKE256_W16_H40_D2	Section 5.3	0x0c01010c
XMSSMT_SHAKE256_W16_H40_D4	Section 5.3	0x0d01010d
XMSSMT_SHAKE256_W16_H40_D8	Section 5.3	0x0e01010e
XMSSMT_SHAKE256_W16_H60_D3	Section 5.3	0x0f01010f
XMSSMT_SHAKE256_W16_H60_D6	Section 5.3	0x01020201

XMSSMT_SHAKE256_W16_H60_D12	Section 5.3	0x02020202	

Table 6

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

8. Security Considerations

A signature system is considered secure if it prevents an attacker from forging a valid signature. More specifically, consider a setting in which an attacker gets a public key and can learn signatures on arbitrary messages of his choice. A signature system is secure if, even in this setting, the attacker can not produce a new 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. The given values for bit security were estimated according to [HRS16].

8.1. Security Proofs

A full security proof for the scheme described here can be found in [HRS16]. This proof shows 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 proof in [HRS16] considers a different initial message compression than the randomized hashing used here. We comment on this below. For the original schemes, these proofs show that an attacker has to break certain minimal security properties. In particular, it is not sufficient to break the collision resistance of the hash functions to generate a forgery.

More specifically, the requirements on the used functions are that F and H are post-quantum multi-function multi-target second-preimage resistant keyed functions, F fulfills an additional statistical requirement that roughly says that most images have at least two preimages, PRF is a post-quantum pseudorandom function, H_msg is a

post-quantum multi-target extended target collision resistant keyed hash function. For detailed definitions of these properties see [HRS16]. To give some intuition: Multi-function multi-target second preimage resistance is an extension of second preimage resistance to keyed hash functions, covering the case where an adversary succeeds if it finds a second preimage for one out of many values. The same holds for multi-target extended target collision resistance which just lacks the multi-function identifier as target collision resistance already considers keyed hash functions. The proof in [HRS16] splits PRF into two functions. When PRF is used for pseudorandom key generation or generation of randomness for randomized message hashing it is still considered a pseudorandom function. Whenever PRF is used to generate bitmasks and hash function keys it is modeled as a random oracle. This is due to technical reasons in the proof and an implementation using a pseudorandom function is secure.

The proof in [HRS16] considers classical randomized hashing for the initial message compression, i.e., H(r, M) instead of H(r || getRoot(PK) || index, M). While the classical randomized hashing used in [HRS16] allows to prove that it is not enough for an adversary to break the collision resistance of the underlying hash function, it turns out that an attacker could launch a multi-target attack even against multiple users at the same time. The reason is that the adversary attacking u users at the same time learns u * 2^h randomized hashes H(r_i_j || M_i_j) with signature index i in [0, 2^h - 1] and user index j in [0, u]. It suffices to find a single pair (r*, M*) such that H(r* || M*) = H(r_i_u || M_i_u) for one out of the u * 2^h learned hashes. Hence, an attacker can do a brute force search in time 2^n / u * 2^h instead of 2^n.

The indexed randomized hashing H(r || getRoot(PK) || toByte(idx, n), M) used in this work makes the hash function calls position- and user-dependent. This thwarts the above attack because each hash function evaluation during an attack can only target one of the learned randomized hash values. More specifically, an attacker now has to decide which index idx and which root value to use for each query. This can also be shown formally in the random oracle model.

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 and pseudorandom functions assuming conventional and quantum adversaries.

8.2. Minimal 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 a machine is an open conjecture. However, significant progress was made over the last few years in this regard. Hence, we consider it a matter of risk assessment to prepare for this case.

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 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 in [HRS16] that variants of Grover's algorithm are the optimal generic attacks against the security properties of hash functions required for the described scheme.

9. Acknowledgements

We would like to thank Johannes Braun, Peter Campbell, Scott Fluhrer, Burt Kaliski, Adam Langley, David McGrew, Rafael Misoczki, Sean Parkinson, Joost Rijneveld, and the Keccak team for their help and comments.

10. References

10.1. Normative References

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

[FIPS202] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS 202, 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <http://www.rfc-editor.org/info/rfc4506>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

10.2. Informative References

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

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

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

[DC16] McGrew, D. and M. Curcio, "Hash-based signatures", Work in Progress, draft-mcgrew-hash-sigs-04, March 2016.

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

[HRS16] Huelsing, A., Rijneveld, J., and F. Song, "Mitigating Multi-Target Attacks in Hash-based Signatures", Lecture Notes in Computer Science volume 9614. Public-Key Cryptography - PKC 2016, 2016.

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

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

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

[KMN14] Knecht, M., Meier, W., and C. Nicola, "A space- and time-efficient Implementation of the Merkle Tree Traversal Algorithm", Computing Research Repository (CoRR). arXiv:1409.4081, 2014.

[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. Note that this representation includes all optional parameter sets. The same applies for the XMSS and XMSS^MT formats below.

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

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

enum ots_algorithm_type {
  wotsp_reserved     = 0x00000000,
  wotsp_sha2-256_w16 = 0x01000001,
  wotsp_sha2-512_w16 = 0x02000002,
  wotsp_shake128_w16 = 0x03000003,
  wotsp_shake256_w16 = 0x04000004,
};
```

WOTS+ signatures are defined using XDR syntax as follows:

```
/* Byte strings */

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

union ots_signature switch (ots_algorithm_type type) {
  case wotsp_sha2-256_w16:
  case wotsp_shake128_w16:
    bytestring32 ots_sig_n32_len67[67];

  case wotsp_sha2-512_w16:
  case wotsp_shake256_w16:
    bytestring64 ots_sig_n64_len18[131];

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

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

```
union ots_pubkey switch (ots_algorithm_type type) {
  case wotsp_sha2-256_w16:
  case wotsp_shake128_w16:
    bytestring32 ots_pubk_n32_len67[67];

  case wotsp_sha2-512_w16:
  case wotsp_shake256_w16:
    bytestring64 ots_pubk_n64_len18[131];

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

Appendix B. XMSS XDR Formats

XMSS parameter sets are defined using XDR syntax as follows:

```
/* Byte strings */

typedef opaque bytestring4[4];

/* Definition of parameter sets */

enum xmss_algorithm_type {
  xmss_reserved         = 0x00000000,

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

  xmss_sha2-256_w16_h10 = 0x01000001,
  xmss_sha2-256_w16_h16 = 0x02000002,
  xmss_sha2-256_w16_h20 = 0x03000003,

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

  xmss_sha2-512_w16_h10 = 0x04000004,
  xmss_sha2-512_w16_h16 = 0x05000005,
  xmss_sha2-512_w16_h20 = 0x06000006,

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

  xmss_shake128_w16_h10 = 0x07000007,
  xmss_shake128_w16_h16 = 0x08000008,
  xmss_shake128_w16_h20 = 0x09000009,

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

  xmss_shake256_w16_h10 = 0x0a00000a,
  xmss_shake256_w16_h16 = 0x0b00000b,
  xmss_shake256_w16_h20 = 0x0c00000c,
};
```

XMSS signatures are defined using XDR syntax as follows:

```
/* Authentication path types */

union xmss_path switch (xmss_algorithm_type type) {
  case xmss_sha2-256_w16_h10:
  case xmss_shake128_w16_h10:
    bytestring32 path_n32_t10[10];
```

```
        case xmss_sha2-256_w16_h16:
        case xmss_shake128_w16_h16:
          bytestring32 path_n32_t16[16];

        case xmss_sha2-256_w16_h20:
        case xmss_shake128_w16_h20:
          bytestring32 path_n32_t20[20];

        case xmss_sha2-512_w16_h10:
        case xmss_shake256_w16_h10:
          bytestring64 path_n64_t10[10];

        case xmss_sha2-512_w16_h16:
        case xmss_shake256_w16_h16:
          bytestring64 path_n64_t16[16];

        case xmss_sha2-512_w16_h20:
        case xmss_shake256_w16_h20:
          bytestring64 path_n64_t20[20];

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

      /* Types for XMSS random strings */

      union random_string_xmss switch (xmss_algorithm_type type) {
        case xmss_sha2-256_w16_h10:
        case xmss_sha2-256_w16_h16:
        case xmss_sha2-256_w16_h20:
        case xmss_shake128_w16_h10:
        case xmss_shake128_w16_h16:
        case xmss_shake128_w16_h20:
          bytestring32 rand_n32;

        case xmss_sha2-512_w16_h10:
        case xmss_sha2-512_w16_h16:
        case xmss_sha2-512_w16_h20:
        case xmss_shake256_w16_h10:
        case xmss_shake256_w16_h16:
        case xmss_shake256_w16_h20:
          bytestring64 rand_n64;

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

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

```
    union xmss_ots_signature switch (xmss_algorithm_type type) {
      case xmss_sha2-256_w16_h10:
      case xmss_sha2-256_w16_h16:
      case xmss_sha2-256_w16_h20:
        wotsp_sha2-256_w16;

      case xmss_sha2-512_w16_h10:
      case xmss_sha2-512_w16_h16:
      case xmss_sha2-512_w16_h20:
        wotsp_sha2-512_w16;

      case xmss_shake128_w16_h10:
      case xmss_shake128_w16_h16:
      case xmss_shake128_w16_h20:
        wotsp_shake128_w16;

      case xmss_shake256_w16_h10:
      case xmss_shake256_w16_h16:
      case xmss_shake256_w16_h20:
        wotsp_shake256_w16;

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

    /* XMSS signature structure */

    struct xmss_signature {
      /* WOTS+ key pair index */
      bytestring4 idx_sig;
      /* Random string for randomized hashing */
      random_string_xmss rand_string;
      /* WOTS+ signature */
      xmss_ots_signature sig_ots;
      /* authentication path */
      xmss_path nodes;
    };
```

XMSS public keys are defined using XDR syntax as follows:

```
    /* Types for bitmask seed */

    union seed switch (xmss_algorithm_type type) {
      case xmss_sha2-256_w16_h10:
      case xmss_sha2-256_w16_h16:
      case xmss_sha2-256_w16_h20:
```

```
        case xmss_shake128_w16_h10:
        case xmss_shake128_w16_h16:
        case xmss_shake128_w16_h20:
          bytestring32 seed_n32;

        case xmss_sha2-512_w16_h10:
        case xmss_sha2-512_w16_h16:
        case xmss_sha2-512_w16_h20:
        case xmss_shake256_w16_h10:
        case xmss_shake256_w16_h16:
        case xmss_shake256_w16_h20:
          bytestring64 seed_n64;

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

      /* Types for XMSS root node */

      union xmss_root switch (xmss_algorithm_type type) {
        case xmss_sha2-256_w16_h10:
        case xmss_sha2-256_w16_h16:
        case xmss_sha2-256_w16_h20:
        case xmss_shake128_w16_h10:
        case xmss_shake128_w16_h16:
        case xmss_shake128_w16_h20:
          bytestring32 root_n32;

        case xmss_sha2-512_w16_h10:
        case xmss_sha2-512_w16_h16:
        case xmss_sha2-512_w16_h20:
        case xmss_shake256_w16_h10:
        case xmss_shake256_w16_h16:
        case xmss_shake256_w16_h20:
          bytestring64 root_n64;

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

      /* XMSS public key structure */

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

Appendix C. XMSS^MT XDR Formats

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

```
   /* Byte strings */

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

   /* Definition of parameter sets */

   enum xmssmt_algorithm_type {
     xmssmt_reserved             = 0x00000000,

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

     xmssmt_sha2-256_w16_h20_d2  = 0x01000001,
     xmssmt_sha2-256_w16_h20_d4  = 0x02000002,
     xmssmt_sha2-256_w16_h40_d2  = 0x03000003,
     xmssmt_sha2-256_w16_h40_d4  = 0x04000004,
     xmssmt_sha2-256_w16_h40_d8  = 0x05000005,
     xmssmt_sha2-256_w16_h60_d3  = 0x06000006,
     xmssmt_sha2-256_w16_h60_d6  = 0x07000007,
     xmssmt_sha2-256_w16_h60_d12 = 0x08000008,

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

     xmssmt_sha2-512_w16_h20_d2  = 0x09000009,
     xmssmt_sha2-512_w16_h20_d4  = 0x0a00000a,
     xmssmt_sha2-512_w16_h40_d2  = 0x0b00000b,
     xmssmt_sha2-512_w16_h40_d4  = 0x0c00000c,
     xmssmt_sha2-512_w16_h40_d8  = 0x0d00000d,
     xmssmt_sha2-512_w16_h60_d3  = 0x0e00000e,
     xmssmt_sha2-512_w16_h60_d6  = 0x0f00000f,
     xmssmt_sha2-512_w16_h60_d12 = 0x01010101,

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

     xmssmt_shake128_w16_h20_d2  = 0x02010102,
     xmssmt_shake128_w16_h20_d4  = 0x03010103,
     xmssmt_shake128_w16_h40_d2  = 0x04010104,
     xmssmt_shake128_w16_h40_d4  = 0x05010105,
     xmssmt_shake128_w16_h40_d8  = 0x06010106,
     xmssmt_shake128_w16_h60_d3  = 0x07010107,
     xmssmt_shake128_w16_h60_d6  = 0x08010108,
     xmssmt_shake128_w16_h60_d12 = 0x09010109,
```

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

        xmssmt_shake256_w16_h20_d2  = 0x0a01010a,
        xmssmt_shake256_w16_h20_d4  = 0x0b01010b,
        xmssmt_shake256_w16_h40_d2  = 0x0c01010c,
        xmssmt_shake256_w16_h40_d4  = 0x0d01010d,
        xmssmt_shake256_w16_h40_d8  = 0x0e01010e,
        xmssmt_shake256_w16_h60_d3  = 0x0f01010f,
        xmssmt_shake256_w16_h60_d6  = 0x01020201,
        xmssmt_shake256_w16_h60_d12 = 0x02020202,
      };
```

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

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

      union idx_sig_xmssmt switch (xmss_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake256_w16_h20_d4:
          bytestring3 idx3;

        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_sha2-512_w16_h40_d8:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake128_w16_h40_d8:
        case xmssmt_shake256_w16_h40_d2:
        case xmssmt_shake256_w16_h40_d4:
        case xmssmt_shake256_w16_h40_d8:
          bytestring5 idx5;

        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_sha2-512_w16_h60_d3:
```

```
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_shake128_w16_h60_d3:
        case xmssmt_shake128_w16_h60_d6:
        case xmssmt_shake128_w16_h60_d12:
        case xmssmt_shake256_w16_h60_d3:
        case xmssmt_shake256_w16_h60_d6:
        case xmssmt_shake256_w16_h60_d12:
          bytestring8 idx8;

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

      union random_string_xmssmt switch (xmssmt_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake128_w16_h40_d8:
        case xmssmt_shake128_w16_h60_d3:
        case xmssmt_shake128_w16_h60_d6:
        case xmssmt_shake128_w16_h60_d12:
          bytestring32 rand_n32;

        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_sha2-512_w16_h40_d8:
        case xmssmt_sha2-512_w16_h60_d3:
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake256_w16_h20_d4:
        case xmssmt_shake256_w16_h40_d2:
        case xmssmt_shake256_w16_h40_d4:
        case xmssmt_shake256_w16_h40_d8:
        case xmssmt_shake256_w16_h60_d3:
        case xmssmt_shake256_w16_h60_d6:
```

```
        case xmssmt_shake256_w16_h60_d12:
          bytestring64 rand_n64;

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

      /* Type for reduced XMSS signatures */

      union xmss_reduced (xmss_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake128_w16_h60_d6:
          bytestring32 xmss_reduced_n32_t77[77];

        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake128_w16_h40_d8:
        case xmssmt_shake128_w16_h60_d12:
          bytestring32 xmss_reduced_n32_t72[72];

        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake128_w16_h60_d3:
          bytestring32 xmss_reduced_n32_t87[87];

        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake256_w16_h40_d4:
        case xmssmt_shake256_w16_h60_d6:
          bytestring64 xmss_reduced_n32_t141[141];

        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_sha2-512_w16_h40_d8:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_shake256_w16_h20_d4:
        case xmssmt_shake256_w16_h40_d8:
        case xmssmt_shake256_w16_h60_d12:
          bytestring64 xmss_reduced_n32_t136[136];
```

```
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_sha2-512_w16_h60_d3:
        case xmssmt_shake256_w16_h40_d2:
        case xmssmt_shake256_w16_h60_d3:
          bytestring64 xmss_reduced_n32_t151[151];

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

      /* xmss_reduced_array depends on d */

      union xmss_reduced_array (xmss_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake256_w16_h40_d2:
          xmss_reduced xmss_red_arr_d2[2];

        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_sha2-512_w16_h60_d3:
        case xmssmt_shake128_w16_h60_d3:
        case xmssmt_shake256_w16_h60_d3:
          xmss_reduced xmss_red_arr_d3[3];

        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake256_w16_h20_d4:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake256_w16_h40_d4:
          xmss_reduced xmss_red_arr_d4[4];

        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_shake128_w16_h60_d6:
        case xmssmt_shake256_w16_h60_d6:
          xmss_reduced xmss_red_arr_d6[6];

        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-512_w16_h40_d8:
        case xmssmt_shake128_w16_h40_d8:
```

```
        case xmssmt_shake256_w16_h40_d8:
          xmss_reduced xmss_red_arr_d8[8];

        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_shake128_w16_h60_d12:
        case xmssmt_shake256_w16_h60_d12:
          xmss_reduced xmss_red_arr_d12[12];

        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;
        /* Array of d reduced XMSS signatures */
        xmss_reduced_array;
      };
```

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

```
      /* Types for bitmask seed */

      union seed switch (xmssmt_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake128_w16_h60_d6:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake128_w16_h40_d8:
        case xmssmt_shake128_w16_h60_d12:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake128_w16_h60_d3:
          bytestring32 seed_n32;
```

```
        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_sha2-512_w16_h40_d8:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_sha2-512_w16_h60_d3:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake256_w16_h40_d4:
        case xmssmt_shake256_w16_h60_d6:
        case xmssmt_shake256_w16_h20_d4:
        case xmssmt_shake256_w16_h40_d8:
        case xmssmt_shake256_w16_h60_d12:
        case xmssmt_shake256_w16_h40_d2:
        case xmssmt_shake256_w16_h60_d3:
          bytestring64 seed_n64;

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

      /* Types for XMSS^MT root node */

      union xmssmt_root switch (xmssmt_algorithm_type type) {
        case xmssmt_sha2-256_w16_h20_d2:
        case xmssmt_sha2-256_w16_h20_d4:
        case xmssmt_sha2-256_w16_h40_d2:
        case xmssmt_sha2-256_w16_h40_d4:
        case xmssmt_sha2-256_w16_h40_d8:
        case xmssmt_sha2-256_w16_h60_d3:
        case xmssmt_sha2-256_w16_h60_d6:
        case xmssmt_sha2-256_w16_h60_d12:
        case xmssmt_shake128_w16_h20_d2:
        case xmssmt_shake128_w16_h20_d4:
        case xmssmt_shake128_w16_h40_d2:
        case xmssmt_shake128_w16_h40_d4:
        case xmssmt_shake128_w16_h40_d8:
        case xmssmt_shake128_w16_h60_d3:
        case xmssmt_shake128_w16_h60_d6:
        case xmssmt_shake128_w16_h60_d12:
          bytestring32 root_n32;

        case xmssmt_sha2-512_w16_h20_d2:
        case xmssmt_sha2-512_w16_h20_d4:
        case xmssmt_sha2-512_w16_h40_d2:
        case xmssmt_sha2-512_w16_h40_d4:
        case xmssmt_sha2-512_w16_h40_d8:
```

```
        case xmssmt_sha2-512_w16_h60_d3:
        case xmssmt_sha2-512_w16_h60_d6:
        case xmssmt_sha2-512_w16_h60_d12:
        case xmssmt_shake256_w16_h20_d2:
        case xmssmt_shake256_w16_h20_d4:
        case xmssmt_shake256_w16_h40_d2:
        case xmssmt_shake256_w16_h40_d4:
        case xmssmt_shake256_w16_h40_d8:
        case xmssmt_shake256_w16_h60_d3:
        case xmssmt_shake256_w16_h60_d6:
        case xmssmt_shake256_w16_h60_d12:
          bytestring64 root_n64;

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

      /* XMSS^MT public key structure */

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

Appendix D. Changed since draft-irtf-cfrg-xmss-hash-based-signatures-06

1: Added the IETF/CFRG disclaimer on post-quantum cryptography.

2: Throughout the document we now use the term "private key" consistently instead of mixing it with "secret key". The naming of sk, SK, and SK_MT stays the same (a correspondent note is given).

3: (In|De)crementing and shift operators are now explicitely defined in section 2.3

4: Corrected the order of the input for the WOTS+ chaining function (3.1.2).

5: In section 4.1.7 we now use the definition conform assignment wots_sk[i] = WOTS_genSK() instead of WOTS_genSK(wots_sk[i]).

6: Some minor textual corrections were done.

Authors' Addresses

Andreas Huelsing
TU Eindhoven
P.O. Box 513
Eindhoven 5600 MB
NL

Email: ietf@huelsing.net

Denis Butin
TU Darmstadt
Hochschulstrasse 10
Darmstadt 64289
DE

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

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

Email: ietf@gazdag.de

Aziz Mohaisen
SUNY Buffalo
323 Davis Hall
Buffalo, NY 14260
US

Phone: +1 716 645-1592
Email: mohaisen@buffalo.edu