NIST Special Publication 800-208

Recommendation for Stateful Hash-Based Signature Schemes

David A. Cooper
Daniel C. Apon
Quynh H. Dang
Michael S. Davidson
Morris J. Dworkin
Carl A. Miller

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-208

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-208

Recommendation for Stateful Hash-Based Signature Schemes

David A. Cooper
Daniel C. Apon
Quynh H. Dang
Michael S. Davidson
Morris J. Dworkin
Carl A. Miller
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-208

October 2020

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Under Secretary of Commerce for Standards and Technology

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-208
Natl. Inst. Stand. Technol. Spec. Publ. 800-208, 59 pages (October 2020)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-208

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: pqc-comments@nist.gov

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

This recommendation specifies two algorithms that can be used to generate a digital signature, both of which are stateful hash-based signature schemes: the Leighton-Micali Signature (LMS) system and the eXtended Merkle Signature Scheme (XMSS), along with their multi-tree variants, the Hierarchical Signature System (HSS) and multi-tree XMSS (XMSS^{MT}).

Keywords

cryptography; digital signatures; hash-based signatures; public-key cryptography.

Document Conventions

The terms "**shall**" and "**shall not**" indicate requirements to be followed strictly in order to conform to the publication and from which no deviation is permitted.

The terms "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable without mentioning or excluding others, that a certain course of action is preferred but not necessarily required, or that (in the negative form) a certain possibility or course of action is discouraged but not prohibited.

The terms "may" and "need not" indicate a course of action permissible within the limits of the publication.

The terms "can" and "cannot" indicate a possibility and capability, whether material, physical, or causal.

Conformance Testing

Conformance testing for implementations of the functions that are specified in this publication will be conducted within the framework of the Cryptographic Algorithm Validation Program (CAVP) and the Cryptographic Module Validation Program (CMVP). The requirements that apply to these implementations are indicated by the word "**shall**." Some of these requirements may be out-of-scope for CAVP or CMVP validation testing and, thus, are the responsibility of entities using, implementing, installing, or configuring applications that incorporate this Recommendation.

Patent Disclosure Notice

NOTICE: The Information Technology Laboratory (ITL) has requested that holders of patent claims whose use may be required for compliance with the guidance or requirements of this publication disclose such patent claims to ITL. However, holders of patents are not obligated to respond to ITL calls for patents and ITL has not undertaken a patent search in order to identify which, if any, patents may apply to this publication.

As of the date of publication and following call(s) for the identification of patent claims whose use may be required for compliance with the guidance or requirements of this publication, no such patent claims have been identified to ITL.

No representation is made or implied by ITL that licenses are not required to avoid patent infringement in the use of this publication.

Table of Contents

List of Appendices

List of Figures

List of Tables

1 Introduction

This publication supplements FIPS 186 [4] by specifying two additional digital signature schemes, both of which are stateful hash-based signature (HBS) schemes: the Leighton-Micali Signature (LMS) system [2] and the eXtended Merkle Signature Scheme (XMSS) [1], along with their multi-tree variants, the Hierarchical Signature System (HSS) and multi-tree XMSS (XMSS^{MT}). All of the digital signature schemes specified in FIPS 186 will be broken if large-scale quantum computers are ever built. The security of the stateful HBS schemes in this publication depends only on the security of the underlying hash functions—in particular, the infeasibility of finding a preimage or a second preimage—and it is believed that the security of hash functions will not be broken by the development of large-scale quantum computers [20].

This recommendation specifies profiles of LMS, HSS, XMSS, and XMSS^{MT} that are appropriate for use by the U.S. Federal Government. This publication approves the use of some but not all of the parameter sets defined in [1] and [2] and also defines some new parameter sets. The approved parameter sets use 192- or 256-bit outputs with either SHA-256 [3] or SHAKE256 [5]. This recommendation requires that key and signature generation be performed in hardware cryptographic modules that do not allow secret keying material to be exported, even in encrypted form.

1.1 Intended Applications for Stateful HBS Schemes

NIST is in the process of developing standards for post-quantum secure digital signature schemes [7] that can be used as replacements for the schemes that are specified in [4]. Stateful HBS schemes are not suitable for general use because they require careful state management that is often difficult to assure, as summarized in Section 1.2 and described in detail in [8].

Instead, stateful HBS schemes are primarily intended for applications with the following characteristics: 1) it is necessary to implement a digital signature scheme in the near future; 2) the implementation will have a long lifetime; and 3) it would not be practical to transition to a different digital signature scheme once the implementation has been deployed.

An application that may fit this profile is the authentication of firmware updates for constrained devices. Some constrained devices that will be deployed in the near future will be in use for decades. These devices will need to have a secure mechanism for receiving firmware updates, and it may not be practical to change the code for verifying signatures on updates once the devices have been deployed.

1.2 The Importance of the Proper Maintenance of State

In a stateful HBS scheme, an HBS private key consists of a large set of one-time signature (OTS) private keys. The signer needs to ensure that no individual OTS key is ever used to sign more than one message. If an attacker were able to obtain digital signatures for two different messages that were created using the same OTS key, then it would become computationally feasible for that attacker to forge signatures on arbitrary messages [13]. Therefore, as described in [8], when a stateful HBS scheme is implemented, extreme care needs to be taken in order to ensure that no OTS key is ever reused.

In order to obtain assurance that OTS keys are not reused, the signing process should be performed in a highly controlled environment. As described in [8], there are many ways in which seemingly routine operations could lead to the risk of one-time key reuse. The conformance requirements imposed in Section 8.1 on cryptographic modules that implement stateful HBS schemes are intended to help prevent one-time key reuse.

1.3 Outline of Text

The remainder of this document is divided into the following sections and appendices:

- Section 2, *Glossary of Terms, Acronyms, and Mathematical Symbols*, defines the terms, acronyms, and mathematical symbols used in this document. This section is *informative*.
- Section 3, *General Discussion*, gives a conceptual explanation of the elements used in stateful hash-based signature schemes (including hash chains, Merkle trees, and hash prefixes). This section may be used as either a high-level overview of stateful hash-based signature schemes or as an introduction to the detailed descriptions of LMS and XMSS provided in [1] and [2]. This section is *informative*.
- Section 4, *Leighton-Micali Signatures (LMS) Parameter Sets*, describes the parameter sets that are approved for use by this Special Publication with LMS and HSS.
- Section 5, *eXtended Merkle Signature Scheme (XMSS) Parameter Sets*, describes the parameter sets that are approved for use by this Special Publication with XMSS and XMSS^{MT}.
- Section 6, *Random Number Generation for Keys and Signatures*, states how the random data used in XMSS and LMS must be generated.
- Section 7, *Distributed Multi-Tree Hash-Based Signatures*, provides recommendations for distributing the implementation of a single HSS or XMSS^{MT} instance over multiple cryptographic modules.
- Section 8, *Conformance*, specifies requirements for cryptographic algorithm and module validation that are specific to modules that implement the algorithms in this document.
- Section 9, *Security Considerations*, enumerates security risks in various scenarios for stateful HBS schemes (with a focus on the problem of key reuse) and describes steps that should be taken to maximize the security of an implementation. This section is *informative*.
- Appendix A, *LMS XDR Syntax Additions*, describes additions that are required for the External Data Representation (XDR) syntax for LMS in order to support the new parameter sets specified in this document.
- Appendix B, *XMSS XDR Syntax Additions*, describes additions that are required for the XDR syntax for XMSS and XMSS^{MT} in order to support the new parameter sets specified in this document.

- Appendix C, *Provable Security Analysis*, provides information about the security proofs that are available for LMS and XMSS. This section is *informative*.

2 Glossary of Terms, Acronyms, and Mathematical Symbols

2.1 Terms and Definitions

approved	FIPS-**approved** or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST Recommendation or 2) adopted in a FIPS or NIST Recommendation and specified either (a) in an appendix to the FIPS or NIST Recommendation or (b) in a document referenced by the FIPS or NIST Recommendation.

2.2 Acronyms

Selected acronyms and abbreviations used in this publication are defined below.

EEPROM	Electronically erasable programmable read-only memory
EUF-CMA	Existential unforgeability under adaptive chosen message attacks
FIPS	Federal Information Processing Standard
HBS	Hash-based signature
HSS	Hierarchical Signature Scheme
IRTF	Internet Research Task Force
LM-OTS	Leighton-Micali One-Time Signature
LMS	Leighton-Micali signature
NIST	National Institute of Standards and Technology
OTS	One-time signature
QROM	Quantum random oracle model
RAM	Random access memory
RFC	Request for Comments
ROM	Random oracle model
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm KECCAK
SP	Special Publication

VM	Virtual machine
WOTS$^+$	Winternitz One-Time Signature Plus
XDR	External Data Representation
XMSS	eXtended Merkle Signature Scheme
XMSS^{MT}	Multi-tree XMSS

2.3 Mathematical Symbols

SHA-256(M)	SHA-256 hash function as specified in [3].
SHA-256/192(M)	T_{192}(SHA-256(M)), the most significant (i.e., leftmost) 192 bits of the SHA-256 hash of M.
SHAKE256/256(M)	SHAKE256(M, 256), where SHAKE256 is specified in Section 6.2 of [5]. The output length is 256 bits.
SHAKE256/192(M)	SHAKE256(M, 192), where SHAKE256 is specified in Section 6.2 of [5]. The output length is 192 bits.
$T_{192}(X)$	A truncation function that outputs the most significant (i.e., leftmost) 192 bits of the input bit string X.
n	The number of bytes in the output of a hash function.
m	In LMS, the number of bytes associated with each node of a Merkle tree.
w	1. In XMSS, the length of a Winternitz chain. A single Winternitz chain uses $\log_2(w)$ bits from the hash or checksum. 2. In LMS, the number of bits from the hash or checksum used in a single Winternitz chain. The length of a Winternitz chain is 2^w. (Note that using a Winternitz parameter of $w = 4$ in LMS would be comparable to using a parameter of $w = 16$ in XMSS.)
p	The number of n-byte string elements in an LM-OTS private key, public key, and signature.
Len	The number of n-byte string elements in a WOTS+ private key, public key, and signature.
H	In LMS and XMSS, the height of the tree. In XMSS^{MT}, the total height of the multi-tree (the trees at each level have a height of H/D).

D	The number of levels of trees in XMSS^{MT}.
L	The number of levels of trees in HSS.
I	A 16-byte string used in LMS as a key pair identifier.
C	In LMS, the *n*-byte randomizer used for randomized message hashing.
R	In XMSS, the *n*-byte randomizer used for randomized message hashing.
SK_PRF	An *n*-byte key used to pseudorandomly generate the randomizer *r*.
S_XMSS	A secret random value used for pseudorandom key generation in XMSS.
ADRS	A 32-byte data structure used in XMSS when generating prefixes (keys) and bitmasks.
SEED	1. In XMSS, the public, random, unique identifier for the long-term key. 2. In LMS, a secret random value used for pseudorandom key generation.

3 General Discussion

At a high level, XMSS and LMS are very similar. They each consist of two components—a one-time signature (OTS) scheme and a method for creating a single, long-term public key from a large set of OTS public keys. A brief explanation of OTS schemes and the method for creating a long-term public key from a large set of OTS public keys can be found in Sections 3 and 4 of [14].

3.1 One-Time Signature Systems

Both LMS and XMSS make use of variants of the Winternitz signature scheme. In the Winternitz signature scheme, the message to be signed is hashed to create a digest; the digest is encoded as a base b number;[1] and then each digit of the digest is signed using a hash chain, as follows.

A hash chain is created by first randomly generating a secret value, x, which is the private key. The size of x should generally correspond to the targeted security strength of the scheme. So, for the parameter sets approved by this recommendation, x will be either 192 or 256 bits in length. The public key, pub, is then created by applying the hash function, H, to the secret $b - 1$ times, $H^{b-1}(x)$. Figure 1 shows an example of a hash chain for the kth digit of a digest where b is 4.

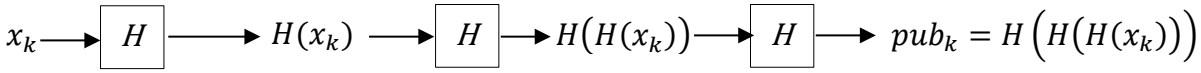

Figure 1: A sample Winternitz chain for *b* = 4

The kth digit of the digest, N_k, is signed by applying the hash function, H, to the private key N_k times, $H^{N_k}(x_k)$. Figure 2 shows an example of a signature for the kth digit of the digest created using the Winternitz chain in Figure 1 when N_k is 1. As shown, the signature is $s_k = H^1(x_k) = H(x_k)$. Figure 2 also shows how the signature, s_k, can be verified. The hash function, H, is applied to s_k twice, and if the resulting value is the same as the public key, pub_k, then the signature is valid. In general, the signature for the kth digit of a digest can be verified by checking that $pub_k = H^{b-1-N_k}(s_k)$.

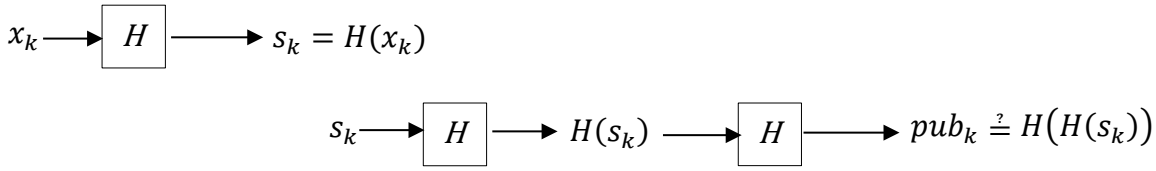

Figure 2: A sample Winternitz signature generation and verification

As noted in [14], simply signing the individual digits of the digest is not sufficient because an

[1] The base b is referred to as the Winternitz parameter in this publication. RFC 8391 [1] specifies that the Winternitz parameter, denoted by w there, may be either 4 or 16 but only specifies parameter sets for $w = 16$. In RFC 8554 [2], the term "Winternitz parameter," also denoted by w, refers to a different but related quantity: the number of bits of the digest that is encoded by b. RFC 8554 specifies that w may be 1, 2, 4, or 8, which corresponds to a b of 2, 4, 16, or 256, respectively.

attacker would be able to generate valid signatures for other message digests. For example, given $s_k = H(x_k)$, as in Figure 2, an attacker would be able to generate a signature for a message digest with a kth digit of 2 by applying H to s_k once or to a message digest with a kth digit of 3 by applying H to s_k twice. An attacker could not, however, generate a signature for a message digest with a kth digit of 0 as this would require finding some value y such that $H(y) = s_k$, which would not be feasible as long as H is preimage-resistant.

In order to protect against the above attack, the Winternitz signature scheme computes a checksum of the message digest and signs the checksum along with the digest. For an n-digit message digest, the checksum is computed as $\sum_{k=0}^{n-1}(b - 1 - N_k)$. The checksum is designed so that the value is non-negative, and any increase in a digit in the message digest will result in the checksum becoming smaller. This prevents an attacker from creating an effective forgery from a message signature since the attacker can only increase values within the message digest and cannot decrease values within the checksum.

Figure 3 shows an example of a signature for a 32-bit message digest using $b = 16$. The digest is written as eight hexadecimal digits, and a separate hash chain is used to sign each digit with each hash chain having its own private key.[2]

	Digest								Checksum	
Digest	6	3	F	1	E	9	0	B	3	D
Private Key	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Signature	$H^6(x_0)$	$H^3(x_1)$	$H^{15}(x_2)$	$H(x_3)$	$H^{14}(x_4)$	$H^9(x_5)$	x_6	$H^{11}(x_7)$	$H^3(x_8)$	$H^{13}(x_9)$
Public Key	$H^{15}(x_0)$	$H^{15}(x_1)$	$H^{15}(x_2)$	$H^{15}(x_3)$	$H^{15}(x_4)$	$H^{15}(x_5)$	$H^{15}(x_6)$	$H^{15}(x_7)$	$H^{15}(x_8)$	$H^{15}(x_9)$

Figure 3: A sample Winternitz signature

3.2 Merkle Trees

While a single, long-term public key could be created from a large set of OTS public keys by simply concatenating the keys together, the resulting public key would be unacceptably large. XMSS and LMS instead use Merkle hash trees [18], which allow for the long-term public key to be very short in exchange for requiring a small amount of additional information to be provided with each OTS key. To create a hash tree, the OTS public keys are hashed once to form the leaves of the tree, and these hashes are then hashed together in pairs to form the next level up. Those hash values are then hashed together in pairs, the resulting hash values are hashed together, and so on until all of the public keys have been used to generate a single hash value (the root of the tree), which will be used as the long-term public key.

[2] If SHA-256 were used as the hash function, then the message digest would be encoded as 64 hexadecimal digits, and the checksum would be encoded as three hexadecimal digits.

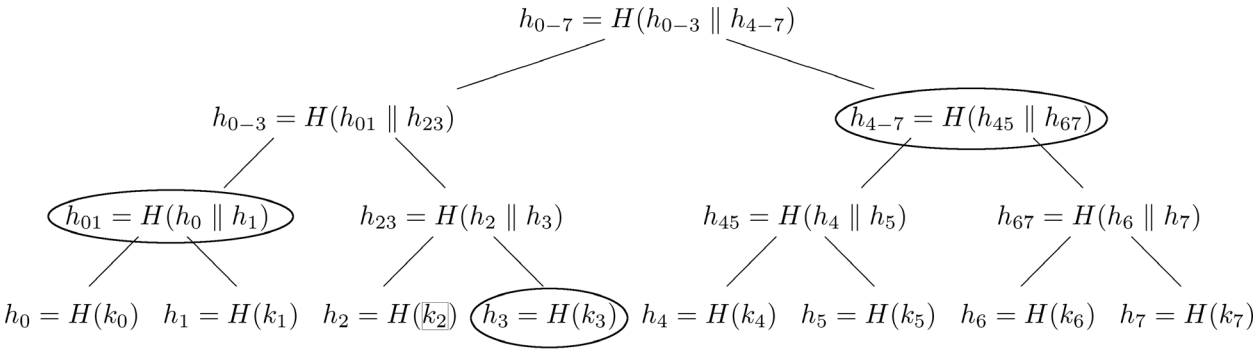

Figure 4: A Merkle Hash Tree

Figure 4 depicts a hash tree that contains eight OTS public keys ($k_0 \ldots k_7$). The eight keys are each hashed to form the leaves of the tree ($h_0 \ldots h_7$), and the eight leaf values are hashed in pairs to create the next level up in the tree ($h_{01}, h_{23}, h_{45}, h_{67}$). These four hash values are again hashed in pairs to create $h_{0–3}$ and $h_{4–7}$, which are hashed together to create the long-term public key, $h_{0–7}$. In order for an entity that had already received $h_{0–7}$ in a secure manner to verify a message signed using k_2, the signer would need to provide h_3, h_{01}, and $h_{4–7}$ in addition to k_2. The verifier would compute $h'_2 = H(k_2)$, $h'_{23} = H(h'_2||h_3)$, $h'_{0-3} = H(h_{01}||h'_{23})$, and $h'_{0-7} = H(h'_{0-3}||h_{4-7})$. If h'_{0-7} is the same as $h_{0–7}$, then k_2 may be used to verify the message signature.

3.3 Two-Level Trees

Both [1] and [2] define single tree and multi-tree variants of their signature schemes. In an instance that involves two levels of trees, as shown in Figure 5, the OTS keys that form the leaves of the top-level tree sign the roots of the trees at the bottom level, and the OTS keys that form the leaves of the bottom-level trees are used to sign the messages. The root of the top-level tree is the long-term public key for the signature scheme.[3]

[3] While this section only describes two-level trees, HSS allows for up to eight levels of trees, and XMSS^{MT} allows for up to 12 levels of trees.

As described in Section 7, the use of two levels of trees can make it easier to distribute OTS keys across multiple cryptographic modules in order to protect against private key loss. A set of OTS keys can be created in one cryptographic module, and the root of the Merkle tree formed from these keys can be published as the public key for the signature scheme. OTS keys can then be created on multiple other cryptographic modules with a separate Merkle tree created for the OTS keys of each of the other cryptographic modules, and a different OTS key from the first cryptographic module can be used to sign each of the roots of the other cryptographic modules.

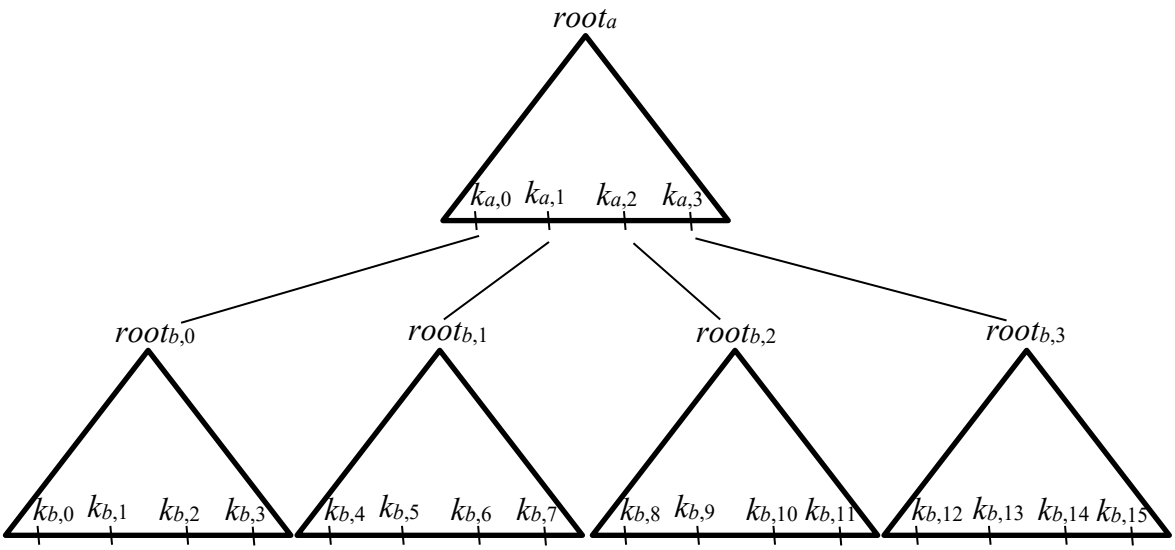

Figure 5: A two-Level Merkle tree

While there are benefits in the use of a two-level tree, it results in larger signatures and slower signature verification as each message signature will need to include two OTS signatures. For example, if a message were signed using OTS key $k_{b,6}$ in Figure 5, the signature would need to include the signature on $root_{b,1}$ using $k_{a,1}$ in addition to the signature on the message using $k_{b,6}$.

3.4 Prefixes and Bitmasks

In order to strengthen the security of the schemes in both XMSS and LMS, a prefix is prepended whenever a value is hashed. For example, when computing the public key for a Winternitz signature from the private key in LMS as described in Section 3.1, rather than just computing $pub_k = H^3(x_k) = H\left(H\left(H(x_k)\right)\right)$, the public key is computed as $pub_k = H\left(p_3 \,||\, H\left(p_2 \,||\, H(p_1 \,||\, x_k)\right)\right)$, where p_1, p_2, and p_3 are each different prefix values. The prefix is formed by concatenating together various pieces of information, including a unique identifier for the long-term public key and an indicator of the purpose of the hash (e.g., Winternitz chain or Merkle tree). If the hash is part of a Winternitz chain, then the prefix also includes the number of the OTS key, which digit of the digest or checksum is being signed, and where in the chain the hash appears. The goal is to ensure that every single hash that is computed within the LMS scheme uses a different prefix.

XMSS generates its prefixes in a similar way. The information described above is used to form an address, which uniquely identifies where a particular hash invocation occurs within the

scheme. This address is then hashed along with a unique identifier (SEED) for the long-term public key to create the prefix.

Unlike LMS, XMSS also uses bitmasks. Similar to the generation of the prefix, a slightly different address is hashed along with the SEED to create a bitmask, which is exclusive-ORed with the value to be hashed (the input). The prefix and masked input value are concatenated as specified in [1] to form the input to the hash function.[4] Figure 6 illustrates an example of this computation when the input is a Winternitz chaining value. In [1], the hash function is referred to as H, H_msg, F, or PRF, depending on where it is being used. However, in each case, it is the same function, just with a different prefix prepended in order to ensure separation between the uses.

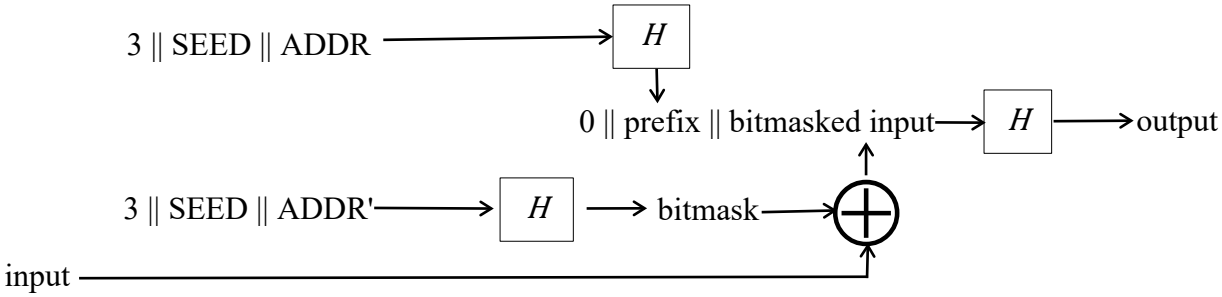

Figure 6: XMSS hash computation with prefix and bitmask

[4] When hashing values within a Merkle tree, two bitmasks are created, one for the left input and one for the right input, and the two bitmasked values are concatenated along with the prefix.

4 Leighton-Micali Signatures (LMS) Parameter Sets

The LMS and HSS algorithms are described in RFC 8554 [2]. This Special Publication approves the use of LMS and HSS with four different hash functions: SHA-256, SHA-256/192, SHAKE256/256, and SHAKE256/192 (see Section 2.3). The parameter sets that use SHA-256 are defined in RFC 8554 [2]. The parameter sets that use SHA-256/192, SHAKE256/256, and SHAKE256/192 are defined in this publication.

When generating a key pair for an LMS instance, each LM-OTS key in the system **shall** use the same parameter set, and the hash function used for the LMS system **shall** be the same as the hash function used in the LM-OTS keys. The height of the tree (h) **shall** be 5, 10, 15, 20, or 25.

When generating a key pair for an HSS instance, the requirements specified in the previous paragraph apply to each LMS tree in the instance. If the HSS instance has more than one level, then the hash function used for the tree at level 0 **shall** be used for every LMS tree at every other level. For each level, the same LMS and LM-OTS parameter sets **shall** be used for every LMS tree at that level. Different LMS and LM-OTS parameter sets may be used at different levels, as long as all chosen parameter sets use the same hash function.

The LMS and LM-OTS parameter sets that are approved for use by this Special Publication are specified in tables in Sections 4.1 through 4.4. The parameters n, w, p, m, and h specified in the tables are defined in Sections 4.1 and 5.1 of [2].

Extensions to the XDR syntax in Section 3.3 of [2] needed to support the parameter sets defined in Sections 4.2 through 4.4 of this document are specified in Appendix A. The numeric identifiers for these parameter sets are marked as "TBD" since they had not yet been assigned at the time this document was published. Once they are assigned, the numeric identifiers may be found at https://www.iana.org/assignments/leighton-micali-signatures/leighton-micali-signatures.xhtml.

4.1 LMS with SHA-256

When generating LMS or HSS key pairs using SHA-256, the LMS and LM-OTS parameter sets **shall** be selected from the following two tables, which come from Sections 4 and 5 of [2].

Table 1: LM-OTS parameter sets for SHA-256

LM-OTS Parameter Sets	Numeric Identifier	*n*	*w*	*P*
LMOTS_SHA256_N32_W1	0x00000001	32	1	265
LMOTS_SHA256_N32_W2	0x00000002	32	2	133
LMOTS_SHA256_N32_W4	0x00000003	32	4	67
LMOTS_SHA256_N32_W8	0x00000004	32	8	34

Table 2: LMS parameter sets for SHA-256

LMS Parameter Sets	Numeric Identifier	*m*	*h*
LMS_SHA256_M32_H5	0x00000005	32	5
LMS_SHA256_M32_H10	0x00000006	32	10
LMS_SHA256_M32_H15	0x00000007	32	15
LMS_SHA256_M32_H20	0x00000008	32	20
LMS_SHA256_M32_H25	0x00000009	32	25

4.2 LMS with SHA-256/192

When generating LMS or HSS key pairs using SHA-256/192, the LMS and LM-OTS parameter sets **shall** be selected from the following two tables.

Table 3: LM-OTS parameter sets for SHA-256/192

LM-OTS Parameter Sets	Numeric Identifier	*n*	*w*	*P*
LMOTS_SHA256_N24_W1	TBD	24	1	200
LMOTS_SHA256_N24_W2	TBD	24	2	101
LMOTS_SHA256_N24_W4	TBD	24	4	51
LMOTS_SHA256_N24_W8	TBD	24	8	26

Table 4: LMS parameter sets for SHA-256/192

LMS Parameter Sets	Numeric Identifier	*m*	*h*
LMS_SHA256_M24_H5	TBD	24	5
LMS_SHA256_M24_H10	TBD	24	10
LMS_SHA256_M24_H15	TBD	24	15
LMS_SHA256_M24_H20	TBD	24	20
LMS_SHA256_M24_H25	TBD	24	25

4.3 LMS with SHAKE256/256

When generating LMS or HSS key pairs using SHAKE256/256, the LMS and LM-OTS parameter sets **shall** be selected from the following two tables.

Table 5: LM-OTS parameter sets for SHAKE256/256

LM-OTS Parameter Sets	Numeric Identifier	*n*	*w*	*p*
LMOTS_SHAKE_N32_W1	TBD	32	1	265
LMOTS_SHAKE_N32_W2	TBD	32	2	133
LMOTS_SHAKE_N32_W4	TBD	32	4	67
LMOTS_SHAKE_N32_W8	TBD	32	8	34

Table 6: LMS parameter sets for SHAKE256/256

LMS Parameter Sets	Numeric Identifier	*m*	*h*
LMS_ SHAKE_M32_H5	TBD	32	5
LMS_ SHAKE_M32_H10	TBD	32	10
LMS_ SHAKE_M32_H15	TBD	32	15
LMS_ SHAKE_M32_H20	TBD	32	20
LMS_ SHAKE_M32_H25	TBD	32	25

4.4 LMS with SHAKE256/192

When generating LMS or HSS key pairs using SHAKE256/192, the LMS and LM-OTS parameter sets **shall** be selected from the following two tables.

Table 7: LM-OTS parameter sets for SHAKE256/192

LM-OTS Parameter Sets	Numeric Identifier	*n*	*w*	*p*
LMOTS_SHAKE_N24_W1	TBD	24	1	200
LMOTS_SHAKE_N24_W2	TBD	24	2	101
LMOTS_SHAKE_N24_W4	TBD	24	4	51
LMOTS_SHAKE_N24_W8	TBD	24	8	26

Table 8: LMS parameter sets for SHAKE256/192

LMS Parameter Sets	Numeric Identifier	*m*	*h*
LMS_ SHAKE_M24_H5	TBD	24	5
LMS_ SHAKE_M24_H10	TBD	24	10
LMS_ SHAKE_M24_H15	TBD	24	15
LMS_ SHAKE_M24_H20	TBD	24	20
LMS_ SHAKE_M24_H25	TBD	24	25

5 eXtended Merkle Signature Scheme (XMSS) Parameter Sets

The XMSS and XMSS^{MT} algorithms are described in RFC 8391 [1]. This Special Publication approves the use of XMSS and XMSS^{MT} with four different hash functions: SHA-256, SHA-256/192, SHAKE256/256, and SHAKE256/192 (see Section 2.3).[5] The parameter sets that use SHA-256 are defined in RFC 8391 [1]. The parameter sets that use SHA-256/192, SHAKE256/256, and SHAKE256/192 are defined below.

The WOTS$^+$ parameters that correspond to the use of each of these hash functions are specified in the following table.

Table 9: WOTS$^+$ parameter sets

Parameter Sets	Numeric Identifier	F / PRF	*n*	*w*	*len*
WOTSP-SHA2_256	0x00000001	See Section 5.1	32	16	67
WOTSP-SHA2_192	0x00000005	See Section 5.2	24	16	51
WOTSP-SHAKE256_256	0x00000006	See Section 5.3	32	16	67
WOTSP-SHAKE256_192	0x00000007	See Section 5.4	24	16	51

The XMSS and XMSS^{MT} parameter sets that are approved for use by this Special Publication are specified in Sections 5.1 through 5.4. The parameters *n*, *w*, *len*, *h*, and *d* specified in the tables are defined in Sections 3.1.1, 4.1.1, and 4.2.1 of [1].

Extensions to the XDR syntax in Appendices A, B, and C of [1] needed to support the parameter sets defined in Sections 5.2 through 5.4 of this document are specified in Appendix B.

5.1 XMSS and XMSS^{MT} with SHA-256

When generating XMSS or XMSS^{MT} key pairs using SHA-256, the parameter sets **shall** be selected from the following two tables, which come from Section 5 of [1]. Each of these uses the WOTSP-SHA2_256 parameter set.

Table 10: XMSS parameter sets for SHA-256

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*
XMSS-SHA2_10_256	0x00000001	32	16	67	10
XMSS-SHA2_16_256	0x00000002	32	16	67	16
XMSS-SHA2_20_256	0x00000003	32	16	67	20

[5] The parameter sets specified in RFC 8391 [1] that use SHAKE128, SHAKE256, and SHA-512 are not approved for use by this Special Publication.

Table 11: XMSS^{MT} parameter sets for SHA-256

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*	*d*
XMSSMT-SHA2_20/2_256	0x00000001	32	16	67	20	2
XMSSMT-SHA2_20/4_256	0x00000002	32	16	67	20	4
XMSSMT-SHA2_40/2_256	0x00000003	32	16	67	40	2
XMSSMT-SHA2_40/4_256	0x00000004	32	16	67	40	4
XMSSMT-SHA2_40/8_256	0x00000005	32	16	67	40	8
XMSSMT-SHA2_60/3_256	0x00000006	32	16	67	60	3
XMSSMT-SHA2_60/6_256	0x00000007	32	16	67	60	6
XMSSMT-SHA2_60/12_256	0x00000008	32	16	67	60	12

For the parameter sets in this section, the functions F, H, H_msg, and PRF are as defined in Section 5.1 of [1] for SHA2 with $n = 32$. The function PRF_{keygen}, which is used for key generation as specified in Section 6.2, is defined as follows:

$$PRF_{keygen}(KEY, M): SHA\text{-}256(toByte(4, 32) \,||\, KEY \,||\, M).$$

5.2 XMSS and XMSS^{MT} with SHA-256/192

When generating XMSS or XMSS^{MT} key pairs using SHA-256/192, the parameter sets **shall** be selected from the following two tables. Each of these uses the WOTSP-SHA2_192 parameter set.

Table 12: XMSS parameter sets for SHA-256/192

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*
XMSS-SHA2_10_192	0x0000000D	24	16	51	10
XMSS-SHA2_16_192	0x0000000E	24	16	51	16
XMSS-SHA2_20_192	0x0000000F	24	16	51	20

Table 13: XMSS^{MT} parameter sets for SHA-256/192

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*	*d*
XMSSMT-SHA2_20/2_192	0x00000021	24	16	51	20	2
XMSSMT-SHA2_20/4_192	0x00000022	24	16	51	20	4
XMSSMT-SHA2_40/2_192	0x00000023	24	16	51	40	2
XMSSMT-SHA2_40/4_192	0x00000024	24	16	51	40	4
XMSSMT-SHA2_40/8_192	0x00000025	24	16	51	40	8
XMSSMT-SHA2_60/3_192	0x00000026	24	16	51	60	3
XMSSMT-SHA2_60/6_192	0x00000027	24	16	51	60	6
XMSSMT-SHA2_60/12_192	0x00000028	24	16	51	60	12

For the parameter sets in this section, the functions F, H, H_msg, PRF, and PRF_{keygen} are defined as follows:

- F(KEY, M): T_{192}(SHA-256(toByte(0, 4) || KEY || M))
- H(KEY, M): T_{192}(SHA-256(toByte(1, 4) || KEY || M))
- H_msg(KEY, M): T_{192}(SHA-256(toByte(2, 4) || KEY || M))
- PRF(KEY, M): T_{192}(SHA-256(toByte(3, 4) || KEY || M))
- PRF_{keygen}(KEY, M): T_{192}(SHA-256(toByte(4, 4) || KEY || M))

5.3 XMSS and XMSS^{MT} with SHAKE256/256

When generating XMSS or XMSS^{MT} key pairs using SHAKE256/256, the parameter sets **shall** be selected from the following two tables. Each of these uses the WOTSP-SHAKE256_256 parameter set.

Table 14: XMSS parameter sets for SHAKE256/256

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*
XMSS-SHAKE256_10_256	0x00000010	32	16	67	10
XMSS-SHAKE256_16_256	0x00000011	32	16	67	16
XMSS-SHAKE256_20_256	0x00000012	32	16	67	20

Table 15: XMSS^{MT} parameter sets for SHAKE256/256

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*	*d*
XMSSMT-SHAKE256_20/2_256	0x00000029	32	16	67	20	2
XMSSMT-SHAKE256_20/4_256	0x0000002A	32	16	67	20	4
XMSSMT-SHAKE256_40/2_256	0x0000002B	32	16	67	40	2
XMSSMT-SHAKE256_40/4_256	0x0000002C	32	16	67	40	4
XMSSMT-SHAKE256_40/8_256	0x0000002D	32	16	67	40	8
XMSSMT-SHAKE256_60/3_256	0x0000002E	32	16	67	60	3
XMSSMT-SHAKE256_60/6_256	0x0000002F	32	16	67	60	6
XMSSMT-SHAKE256_60/12_256	0x00000030	32	16	67	60	12

For the parameter sets in this section, the functions F, H, H_msg, PRF, and PRF_{keygen} are defined as follows:

- F(KEY, M): SHAKE256(toByte(0, 32) || KEY || M, 256)
- H(KEY, M): SHAKE256(toByte(1, 32) || KEY || M, 256)
- H_msg(KEY, M): SHAKE256(toByte(2, 32) || KEY || M, 256)
- PRF(KEY, M): SHAKE256(toByte(3, 32) || KEY || M, 256)
- PRF_{keygen}(KEY, M): SHAKE256(toByte(4, 32) || KEY || M, 256)

5.4 XMSS and XMSS^{MT} with SHAKE256/192

When generating XMSS or XMSS^{MT} key pairs using SHAKE256/192, the parameter sets **shall** be selected from the following two tables. Each of these uses the WOTSP-SHAKE256_192 parameter set.

Table 16: XMSS parameter sets for SHAKE256/192

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*
XMSS-SHAKE256_10_192	0x00000013	24	16	51	10
XMSS-SHAKE256_16_192	0x00000014	24	16	51	16
XMSS-SHAKE256_20_192	0x00000015	24	16	51	20

Table 17: XMSS^{MT} parameter sets for SHAKE256/192

Parameter Sets	Numeric Identifier	*n*	*w*	*len*	*h*	*d*
XMSSMT-SHAKE256_20/2_192	0x00000031	24	16	51	20	2
XMSSMT-SHAKE256_20/4_192	0x00000032	24	16	51	20	4
XMSSMT-SHAKE256_40/2_192	0x00000033	24	16	51	40	2
XMSSMT-SHAKE256_40/4_192	0x00000034	24	16	51	40	4
XMSSMT-SHAKE256_40/8_192	0x00000035	24	16	51	40	8
XMSSMT-SHAKE256_60/3_192	0x00000036	24	16	51	60	3
XMSSMT-SHAKE256_60/6_192	0x00000037	24	16	51	60	6
XMSSMT-SHAKE256_60/12_192	0x00000038	24	16	51	60	12

For the parameter sets in this section, the functions F, H, H_msg, PRF, and PRF_{keygen} are defined as follows:

- F(KEY, M): SHAKE256(toByte(0, 4) || KEY || M, 192)
- H(KEY, M): SHAKE256(toByte(1, 4) || KEY || M, 192)
- H_msg(KEY, M): SHAKE256(toByte(2, 4) || KEY || M, 192)
- PRF(KEY, M): SHAKE256(toByte(3, 4) || KEY || M, 192)
- PRF_{keygen}(KEY, M): SHAKE256(toByte(4, 4) || KEY || M, 192)

6 Random Number Generation for Keys and Signatures

This section specifies requirements for the generation of random data that apply in addition to the requirements that are specified in [2] for LMS and HSS and in [1] for XMSS and XMSS^{MT}.

Note: Variables and notations used in this section are defined in the relevant documents mentioned above.

6.1 LMS and HSS Random Number Generation Requirements

The LMS key pair identifier, *I*, **shall** be generated using an **approved** random bit generator (see the SP 800-90 series of publications [6]), where the instantiation of the random bit generator supports at least 128 bits of security strength.

The *n*-byte private elements of the LM-OTS private keys (x[i] in Section 4.2 of [2]) **shall** be generated using the pseudorandom key generation method specified in Appendix A of [2]. The same SEED value **shall** be used to generate every private element in a single LMS instance, and SEED **shall** be generated using an **approved** random bit generator [6], where the instantiation of the random bit generator supports at least 8*n* bits of security strength.

If more than one LMS instance is being created (e.g., for an HSS instance), then a separate key pair identifier, *I*, and SEED **shall** be generated for each LMS instance.

When generating a signature, the *n*-byte randomizer *C* (see Section 4.5 of [2]) **shall** be generated using an **approved** random bit generator [6], where the instantiation of the random bit generator supports at least 8*n* bits of security strength.

6.2 XMSS and XMSS^{MT} Random Number Generation Requirements

The *n*-byte values *SK_PRF* and *SEED* **shall** be generated using an **approved** random bit generator (see the SP 800-90 series of publications [6]), where the instantiation of the random bit generator supports at least 8*n* bits of security strength.

The private *n*-byte strings in the WOTS$^+$ private keys (sk[i] in Section 3.1.3 of [1]) **shall** be generated using the pseudorandom key generation method specified in Algorithm 10' in Section 7.2.1: sk[*j*] = $\text{PRF}_{\text{keygen}}$(*S_XMSS*, *SEED* || *ADRS*), where $\text{PRF}_{\text{keygen}}$ is as defined in Section 5 for the parameter set being used.[6] The private seed, *S_XMSS*, **shall** be generated using an **approved** random bit generator [6], where the instantiation of the random bit generator supports at least 8*n* bits of security strength. If more than one XMSS key pair is being created within a cryptographic module (including XMSS keys that belong to a single XMSS^{MT} instance), then a separate random *S_XMSS* **shall** be generated for each XMSS key pair.

[6] For an XMSS key that is not part of an XMSS^{MT} instance, $d = 1$, $L = 0$, and $t = 0$.

7 Distributed Multi-Tree Hash-Based Signatures

If a digital signature key will be used to generate signatures over a long period of time and replacing the public key would be difficult, then it will be necessary to prepare for the possibility that a cryptographic module holding the private key may fail during the key's lifetime. In the case of most digital signature schemes, a common solution is to make copies of the private key. However, in the case of stateful HBS schemes, simply copying the private key would create a risk of OTS key reuse.

While it would be possible to have one cryptographic module generate all of the OTS keys and then distribute different OTS keys to each of the other cryptographic modules, doing so is not an option for cryptographic modules that conform to this recommendation: due to the risks associated with copying OTS keys, this recommendation prohibits exporting private keying material (Section 8).

One option would be to create multiple stateful HBS keys on different cryptographic modules and then configure clients to accept signatures created using any of these keys. These keys could be distributed to clients all at once or using a mechanism such as the Hash Of Root Key certificate extension [23], which provides a mechanism for distributing new public keys over time.

Another option would be to create a single stateful HBS key in which the OTS private keys are distributed across multiple cryptographic modules. The easiest way to have OTS keys on multiple cryptographic modules without exporting private keys is to use HSS or XMSS^{MT} with two levels of trees where the trees are instantiated on different cryptographic modules. First, a top-level LMS or XMSS key pair would be created in a cryptographic module. The top level's OTS keys would only be used to sign the roots of other trees. Then, bottom-level LMS or XMSS key pairs would be created, and the public keys from those key pairs (i.e., the roots of their Merkle trees) would be signed by OTS keys of the top-level key pair. The OTS keys of the bottom-level key pairs would be used to sign ordinary messages. The number of bottom-level key pairs that could be created would be limited only by the number of OTS keys in the top-level key pair.

As an example, suppose that an organization wishes to have a single XMSS^{MT} key with the OTS private keys being distributed across two cryptographic modules (in case one fails), and the organization has determined that, at most, 10 000 signatures will need to be generated over the lifetime of the XMSS^{MT} key. The organization could create a top-level XMSS key pair in one cryptographic module using the XMSSMT-SHA2_20/2_256 parameter set and then create 10 bottom-level XMSS keys in that same cryptographic module. An additional 10 bottom-level XMSS keys could be created in a second cryptographic module with all 20 of the bottom-level's keys being signed by OTS keys of the top-level key pair.

When working with distributed multi-tree hash-based signatures, the cryptographic module that holds the top-level tree is a potential single point of failure. Once this cryptographic module fails, it is no longer possible to sign additional bottom-level key pairs. Consequently, all of the bottom-level keys should be generated up front as part of the initial key generation ceremony. Once the top-level key has been used to sign all of the bottom-level keys, the top-level key is no

longer needed, as copies of the signatures created using OTS keys of the top-level key pair may be stored outside of the cryptographic module.

In order to avoid the top-level key being a single point of failure, the two options described above could be combined to create multiple distributed multi-tree HBS keys. Multiple top-level keys pairs would initially be created, each on a different cryptographic module, and clients would be configured to accept signatures created using any of these keys. Then, whenever a new bottom-level key needed to be created, it could be signed by any one of the top-level keys. This would allow for new bottom-level keys to be created as long as at least one of the cryptographic modules that contains a top-level key remained operational. Of course, the same level of care should be used when signing a bottom-level key as would be used during the initial key generation ceremony (or as would be used in making a copy of an RSA or ECDSA private key).

7.1 HSS

In the case of HSS, the distributed multi-tree scheme described above can be implemented using multiple cryptographic modules that individually implement LMS without modifications. The top-level LMS public key can be converted to an HSS public key by an external, non-cryptographic device. This device can also submit the public keys of the bottom-level LMS keys for signature by the top-level LMS key. In HSS, the operation for signing the root of a lower-level tree is the same as the operation for signing an ordinary message. Finally, this external device can submit ordinary messages to cryptographic modules that hold the bottom-level LMS keys for signing and then combine the resulting LMS signatures with the top-level key's signature on the bottom-level LMS public key in order to create the HSS signature for the ordinary messages (see Algorithm 7 and Algorithm 8 in [2]).

7.2 XMSS^{MT}

Distributing the implementation of an XMSS^{MT} instance across multiple cryptographic modules requires each cryptographic module to implement slightly modified versions of the XMSS key and signature generation algorithms provided in [1]. The modified versions of these algorithms are provided in Section 7.2.1. The modifications are primarily intended to ensure that each XMSS key uses the appropriate values for its layer and tree addresses when computing prefixes and bitmasks. The modifications also ensure that every XMSS key uses the same value for SEED and that the root of the top-level tree is used when computing the hashes of messages to be signed.

Note that while Algorithm 15 in [1] indicates that an XMSS^{MT} secret key has a single *SK_PRF* value that is shared by all of the XMSS secret keys, Algorithm 10' in Section 7.2.1 has each cryptographic module generate its own value for *SK_PRF*. While generating a different *SK_PRF* for each cryptographic module does not exactly align with the specification in [1], doing so does not affect either interoperability or security. *SK_PRF* is only used to pseudorandomly generate the value *r* in Algorithm 16, which is used for randomized hashing, and any secure method for generating random values could be used to generate *r*.

Section 7.2.2 describes the steps that an external, non-cryptographic device needs to perform in order to implement XMSS^{MT} key and signature generation using a set of cryptographic modules that implement the algorithms in Section 7.2.1. While Algorithms 10' and 12' in Section 7.2.1

have been designed to work with XMSS^{MT} instances that have more than two layers, the algorithms in Section 7.2.2 assume that an XMSS^{MT} instance with exactly two layers is being created.

7.2.1 Modified XMSS Key Generation and Signature Algorithms

```
Algorithm 10': XMSS'_keyGen

  // L needs to be in the range [0 … d-1]
  // t needs to be in the range [0 … 2^((d-1-L)(h/d)) - 1]
  Input: level L, tree t,
         public key of top-level tree PK_MT (if L ≠ d - 1)
  Output: XMSS public key PK

  Initialize S_XMSS with an n-byte string using an approved
  random bit generator [6], where the instantiation of the
  random bit generator supports at least 8n bits of security
  strength;

  // SEED needs to be generated for the top-level XMSS key.
  // For all other XMSS keys, the value needs to be copied from
  // the top-level XMSS key.
  if ( L = d – 1 ) {
    Initialize SEED with an n-byte string using an approved
    random bit generator [6], where the instantiation of the
    random bit generator supports at least 8n bits of security
    strength;
  } else {
    SEED = getSEED(PK_MT);
  }
  setSEED(SK, SEED);

  ADRS = toByte(0, 32);
  ADRS.setLayerAddress(L);
  ADRS.setTreeAddress(t);

  // Initialization for SK-specific contents
  idx = t * 2^(h / d);
  for ( i = 0; i < 2^(h / d); i++ ) {
    ADRS.setOTSAddress(i);
    // For each OTS key, i, generate the private key value for
    // chain in the OTS key.
    for ( j=0; j < len; j++) {
      ADRS.setChainAddress(j);
      sk[j] = PRF_keygen(S_XMSS, SEED || ADRS);
    }
    // Set the secret key for OTS key i to the array of len
    // private key values generated for that key.
```

```
    wots_sk[i] = sk;
  }
  setWOTS_SK(SK, wots_sk);

  Initialize SK_PRF with an n-byte string using an approved
  random bit generator [6], where the instantiation of the
  random bit generator supports at least 8n bits of security
  strength;
  setSK_PRF(SK, SK_PRF);

  root = treeHash(SK, 0, h / d, ADRS);

  setLayerAddress(SK, L);
  setTreeAddress(SK, t);
  setIdx(SK, idx);

  // The "root" value in SK needs to be the root of the top-level
  // XMSS tree, as this is the value used when hashing the message
  // to be signed.
  if ( L = d - 1 ) {
    setRoot(SK, root);
    SK = L || t || idx || wots_sk || SK_PRF || root || SEED;
  } else {
    setRoot(SK, getRoot(PK_MT));
    SK = L || t || idx || wots_sk || SK_PRF || getRoot(PK_MT) || SEED;
  }
  // The public key should be encoded using the XDR for
  // xmssmt_public_key in Appendix C.3 of [1] with the additions
  // specified in Appendix B.3 of this document.
  PK = OID || root || SEED;
  return PK;

Algorithm 12': XMSS'_sign

  Input: Message M
  Output: signature Sig

  idx_sig = getIdx(SK);
  setIdx(SK, idx_sig + 1);
  L = getLayerAddress(SK);
  t = getTreeAddress(SK);
  ADRS = toByte(0, 32);
  ADRS.setLayerAddress(L);
  ADRS.setTreeAddress(t);

  if ( L > 0 ) {
    // M must be the n-byte root from an XMSS public key
    byte[n] r = 0; // n-byte string of zeros
```

```
    byte[n] M' = M;
  } else {
    byte[n] r = PRF(getSK_PRF(SK), toByte(idx_sig, 32));
    byte[n] M' = H_msg(r || getRoot(SK) || (toByte(idx_sig, n)), M);
  }
  idx_leaf = idx_sig - t * 2^(h / d);
  Sig = idx_sig || r || treeSig(M', SK, idx_leaf, ADRS);
  return Sig;
```

7.2.2 XMSS^{MT} External Device Operations

```
XMSS^MT external device keygen

  Input: No input

  // Generate top-level key pair on a cryptographic module
  PK_MT = XMSS'_keyGen(1, 0, NULL);

  t = 0;
  for each bottom-level key pair to be created {
    // Generate bottom-level key pair on a cryptographic module
    PK[t] = XMSS'_keygen(0, t, PK_MT);

    // Submit root of bottom-level key pair's public key
    // to be signed by the top-level key pair.
    SigPK[t] = XMSS'_sign(getRoot(PK[t]));

    // If the public key on the bottom-level tree was created using
    // a tree address of t, then its root needs to be signed by OTS
    // key t of the top-level tree. If it was not, then try again.[7]
    while ( getIdx(SigPK[t]) ≠ t ) {
      t = getIdx(SigPK[t]) + 1;
      PK[t] = XMSS'_keygen(0, t, PK_MT);
      SigPK[t] = XMSS'_sign(getRoot(PK[t]));
    }
    t = t + 1;
  }

XMSS^MT external device sign

  Input: Message M
```

[7] While the signing cryptographic module should use its one-time keys sequentially, making it possible for the external device to determine in advance which one-time key will be used to sign the public key of the bottom-level tree, the external device cannot specify to the signing cryptographic module which one-time key it should use. So, there is a small chance that an internal glitch in the signing cryptographic module will cause it to skip over one or more key indices and sign the bottom-level's public key using an unexpected key index. While this event should be rare, if it does happen, the only option is to regenerate the bottom-level key pair, set the tree address to the next expected key index, and then try again.

```
Output: signature Sig

// Send XMSS'_sign() command to one of the bottom-level key pairs
Sig_tmp = XMSS'_sign(M);

idx_sig = getIdx(Sig_tmp);

// Determine which bottom-level tree was used to sign the message
// by extracting the most significant bits of idx_sig.
t = [idx_sig - (idx_sig mod 2^(h / d))]/ 2^(h / d));

// Append the signature of the signing key pair's root
// (just the output of treeSig, not idx_sig or r).
Sig = Sig_tmp || getSig(SigPK[t]);
return Sig;
```

8 Conformance

8.1 Key Generation and Signature Generation

Cryptographic modules that implement signature generation for a parameter set **shall** also implement key generation for that parameter set. Implementations of the key generation and signature algorithms in this document **shall** only be validated for use within hardware cryptographic modules. The cryptographic modules **shall** be validated to provide FIPS 140-2 or FIPS 140-3 [19] Level 3 or higher physical security, and the operational environment **shall** be *non-modifiable* or *limited*.[8] In addition, a cryptographic module that implements the key generation or signature algorithms **shall** only operate in an **approved** mode of operation and **shall not** implement a bypass mode. The cryptographic module **shall not** allow for the export of private keying material. The entropy source for any **approved** random bit generator [6] used in the implementation **shall** be located inside the cryptographic module's physical boundary.

In order to prevent the possible reuse of an OTS key, when the cryptographic module accepts a request to sign a message, the cryptographic module **shall** increment the leaf index of the private key (*q* in LMS, *idx* in XMSS, *idx_sig* in XMSS^{MT}) and **shall** store the incremented leaf index value in nonvolatile storage before exporting a signature value or accepting another request to sign a message. The cryptographic module **shall not** use an OTS key to generate a digital signature more than one time.[9]

Cryptographic modules that implement LMS key and signature generation **shall** support at least one of the LM-OTS parameter sets in Section 4. For each LM-OTS parameter set supported by a cryptographic module, the cryptographic module **shall** support at least one LMS parameter set from Section 4 that uses the same hash function as the LM-OTS parameter set. Cryptographic modules that implement LMS key and signature generation **shall** generate random data in accordance with Section 6.1.

Cryptographic modules that implement XMSS key and signature generation **shall** implement Algorithm 10 and Algorithm 12 from [1] for at least one of the XMSS parameter sets in Section 5. (The WOTS+ key generation method specified in Algorithm 10' in Section 7.2.1 **shall** be used.) Cryptographic modules that support implementation of XMSS^{MT} key and signature generation **shall** implement Algorithm 10' and Algorithm 12' from Section 7.2.1 of this document for at least one of the XMSS^{MT} parameter sets in Section 5. Cryptographic modules that implement XMSS or XMSS^{MT} key and signature generation **shall** generate random data in accordance with Section 6.2.

[8] See Section 4.6 of FIPS 140-2 [19] and Section 7.6 of ISO/IEC 19790.

[9] In some implementations of HSS or XMSS^{MT} (e.g., Algorithm 16 in [1]), the root of the LMS or XMSS tree used to create the signature is signed by its parent each time a signature is generated. This results in an OTS key being used to generate a digital signature more than once. While the OTS key is used more than once, the message being signed is the same, so the result is to just recreate the same signature (as long as the randomizer value is the same each time). However, as noted in [9] and [10], such implementations are vulnerable to fault injection attacks. Implementations compliant with this publication must sign the root of each tree only once. The resulting signature may be stored within the cryptographic module, or it may be exported from the cryptographic module for storage elsewhere.

8.2 Signature Verification

Cryptographic modules that implement LMS signature verification **shall** support at least one of the LM-OTS parameter sets in Section 4. For each LM-OTS parameter set supported by a cryptographic module, the cryptographic module **shall** support at least one LMS parameter set from Section 4 that uses the same hash function as the LM-OTS parameter set.

Cryptographic modules that implement XMSS signature verification **shall** implement Algorithm 14 of [1] for at least one of the parameter sets in Section 5. Cryptographic modules that implement XMSS^{MT} signature verification **shall** implement Algorithm 17 of [1] for at least one of the parameter sets in Section 5.

9 Security Considerations

9.1 One-Time Signature Key Reuse

Both LMS and XMSS are stateful signature schemes. If an attacker were able to obtain signatures for two different messages created using the same one-time signature (OTS) key, then it would become computationally feasible for that attacker to create forgeries [13]. As noted in [8], extreme care needs to be taken in order to avoid the risk that an OTS key will be reused accidentally. While the conformance requirements in Section 8.1 prevent many of the actions that could result in accidental OTS key reuse, cryptographic modules still need to be carefully designed to ensure that unexpected behavior cannot result in an OTS key being reused.

In order to avoid reuse of an OTS key, the state of the private key must be updated each time a signature is generated. If the private key is stored in non-volatile memory, then the state of the key must be updated in the nonvolatile memory to mark an OTS key as unavailable before the corresponding signature that was generated using the OTS key is exported. Depending on the environment, this can be nontrivial to implement. With many operating systems, simply writing the update to a file is not sufficient because the write operation will be cached with the actual write to non-volatile memory occurring later. If the cryptographic module loses power or crashes before the write to non-volatile memory occurs, then the state update will be lost. If a signature were exported after the write operation was issued, but before the update was written to non-volatile memory, there would be a risk that the OTS key would be used again after the cryptographic module restarts.

Some hardware cryptographic modules implement monotonic counters, which are guaranteed to increment each time the counter's value is read. When available, using the current value of a monotonic counter to determine which OTS key to use for a signature may be very helpful in avoiding unintentional reuse of an OTS key.

9.2 Hash Collisions

For LMS and XMSS, as for the other **approved** digital signature schemes [4], the signature generation algorithm is not applied directly to the message but to a *message digest* generated by the underlying hash function. The security of any signature scheme depends on the inability of an attacker to find distinct messages with the same message digest.

There are two ways that an attacker might find these distinct messages. The attacker could look for a message that has the same message digest as a message that has already been signed (a second preimage), or the attacker could look for any two messages that have the same message digest (a generic collision) and then try to get the private key holder (i.e., signer) to sign one of them [21]. Finding a second preimage is much more difficult than finding a generic collision, and it would be infeasible for an attacker to find a second preimage with any of the hash functions allowed by this recommendation.

LMS and XMSS both use randomized hashing. When a message is presented to be signed, a random value is created and prepended to the message, and the hash function is applied to this expanded message to produce the message digest. Prepending the random value makes it infeasible for anyone other than the signer to find a generic collision because finding a collision

would require predicting the randomizing value. The randomized hashing process does not, however, impact the ability of a signer to create a generic collision since the signer, knowing the private key, can choose the random value to prepend to the message.

The 192-bit hash functions described in this recommendation, SHA-256/192 and SHAKE256/192, offer significantly less resistance to generic collision searches than their 256-bit counterparts. In particular, a collision of the 192-bit functions may be found as the number of sampled inputs approaches 2^{96}, as opposed to 2^{128} for the 256-bit functions, and it may be possible for a signer with access to an extremely large amount of computing resources to sample 2^{96} inputs.

Consequently, one trade-off for the use of 192-bit hash functions in LMS and XMSS is the weakening of the verifier's assurance that the signer will not be able to change the message once the signature is revealed. This possibility does not affect the formal security properties of the schemes because it remains the case that only the signer could produce a valid signature on a message.

9.3 Revocation

Although procedures for the revocation of a compromised key are outside the scope of this publication, the implementation of any signature scheme, in principle, should include such a procedure [22]. For implementations of stateful hash-based signature schemes, which would be vulnerable in the event of OTS key reuse, revocation procedures would arguably be even more important.

In practice, however, procedures for revocation that are timely, efficient, and robust are often difficult to implement. For applications with the characteristics described in Section 1.1, the difficulties would likely be magnified.

References

[1] Huelsing A, Butin D, Gazdag S, Rijneveld J, Mohaisen A (2018) XMSS: eXtended Merkle Signature Scheme. (Internet Research Task Force (IRTF)), IRTF Request for Comments (RFC) 8391. https://doi.org/10.17487/RFC8391

[2] McGrew D, Curcio M, Fluhrer S (2019) Leighton-Micali Hash-Based Signatures. (Internet Research Task Force (IRTF)), IRTF Request for Comments (RFC) 8554. https://doi.org/10.17487/RFC8554

[3] National Institute of Standards and Technology (2015) Secure Hash Standard (SHS). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 180-4. https://doi.org/10.6028/NIST.FIPS.180-4

[4] National Institute of Standards and Technology (2013) Digital Signature Standard (DSS). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 186-4. https://doi.org/10.6028/NIST.FIPS.186-4

[5] National Institute of Standards and Technology (2015) SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. https://doi.org/10.6028/NIST.FIPS.202

[6] Special Publication 800-90 series:

Barker EB, Kelsey JM (2015) Recommendation for Random Number Generation Using Deterministic Random Bit Generators. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90A, Rev. 1. https://doi.org/10.6028/NIST.SP.800-90Ar1

Sönmez Turan M, Barker EB, Kelsey JM, McKay KA, Baish ML, Boyle M (2018) Recommendation for the Entropy Sources Used for Random Bit Generation. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B. https://doi.org/10.6028/NIST.SP.800-90B

Barker EB, Kelsey JM (2016) Recommendation for Random Bit Generator (RBG) Constructions. (National Institute of Standards and Technology, Gaithersburg, MD), (Second Draft) NIST Special Publication (SP) 800-90C. Available at https://csrc.nist.gov/publications/detail/sp/800-90c/draft

[7] National Institute of Standards and Technology (2019) *Post-Quantum Cryptography*. Available at https://csrc.nist.gov/projects/post-quantum-cryptography

[8] McGrew D, Kampanakis P, Fluhrer S, Gazdag S, Butin D, Buchmann J (2016) State Management for Hash-Based Signatures. *Cryptology ePrint Archive*, Report 2016/357. https://eprint.iacr.org/2016/357.pdf

[9] Genêt A, Kannwischer MJ, Pelletier H, McLauchlan A (2018) Practical Fault Injection Attacks on SPHINCS. *Cryptology ePrint Archive*, Report 2018/674. https://eprint.iacr.org/2018/674

[10] Castelnovi L, Martinelli A, Prest T (2018) Grafting trees: A fault attack against the SPHINCS framework. *Post-Quantum Cryptography - 9th International Conference (PQCrypto 2018)*, Lecture Notes in Computer Science 10786, pp 165–184. https://doi.org/10.1007/978-3-319-79063-3_8

[11] Fluhrer S (2017) Further Analysis of a Proposed Hash-Based Signature Standard. *Cryptology ePrint Archive*, Report 2017/553. https://eprint.iacr.org/2017/553.pdf

[12] Buchmann J, Dahmen E, Hulsing A (2011) XMSS – A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions. *Cryptology ePrint Archive*, Report 2011/484. https://eprint.iacr.org/2011/484.pdf

[13] Bruinderink LG, Hülsing A (2016) "Oops, I did it again" – Security of One-Time Signatures under Two-Message Attacks. *Cryptology ePrint Archive*, Report 2016/1042. https://eprint.iacr.org/2016/1042.pdf

[14] Perlner R, Cooper D (2009) Quantum Resistant Public Key Cryptography: A Survey. *8th Symposium on Identity and Trust on the Internet (IDtrust 2009)*, pp 85-93. https://doi.org/10.1145/1527017.1527028

[15] Eaton E (2017) Leighton-Micali Hash-Based Signatures in the Quantum Random-Oracle Model. *Cryptology ePrint Archive*, Report 2017/607. https://eprint.iacr.org/2017/607

[16] Bernstein DJ, Hülsing A, Kölbl S, Niederhagen R, Rijneveld J, Schwabe P (2019) The SPHINCS+ Signature Framework. *Cryptology ePrint Archive*, Report 2019/1086. https://eprint.iacr.org/2019/1086.pdf

[17] Malkin T, Micciancio D, Miner S (2002) Efficient generic forward-secure signatures with an unbounded number of time periods. *Advances in Cryptology — EUROCRYPT 2002*, Lecture Notes in Computer Science 2332, pp 400–417. https://doi.org/10.1007/3-540-46035-7_27

[18] Merkle RC (1979) *Security, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979. Available at https://www.merkle.com/papers/Thesis1979.pdf

[19] National Institute of Standards and Technology (2001) Security Requirements for Cryptographic Modules. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-2, Change Notice 2 December 03, 2002. https://doi.org/10.6028/NIST.FIPS.140-2

National Institute of Standards and Technology (2019) Security Requirements for Cryptographic Modules. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-3. https://doi.org/10.6028/NIST.FIPS.140-3

[20] Chen L, Jordan S, Liu Y-K, Moody D, Peralta R, Perlner RA, Smith-Tone D (2016) Report on Post-Quantum Cryptography. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 8105. https://doi.org/10.6028/NIST.IR.8105

[21] Sotirov A, Stevens M, Appelbaum J, Lenstra A, Molnar D, Osvik DA, de Weger B (2008) *MD5 considered harmful today: Creating a rogue CA certificate*. Available at https://www.win.tue.nl/hashclash/rogue-ca

[22] Special Publication 800-57 series:

Barker EB (2020) Recommendation for Key Management, Part 1: General. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 1, Rev. 5. https://doi.org/10.6028/NIST.SP.800-57pt1r5

Barker EB, Barker WC (2019) Recommendation for Key Management: Part 2 – Best Practices for Key Management Organizations. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 2, Rev. 1. https://doi.org/10.6028/NIST.SP.800-57pt2r1

Barker EB, Dang QH (2015) Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 3, Rev. 1. https://doi.org/10.6028/NIST.SP.800-57pt3r1

[23] Housley R (2019) Hash Of Root Key Certificate Extension. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8649. https://doi.org/10.17487/RFC8649

Appendix A—LMS XDR Syntax Additions

In order to support the LM-OTS and LMS parameter sets defined in Sections 4.2 through 4.4, the XDR syntax in Section 3.3 of [2] is extended as follows. For data structures of type `enum` or `union` below, the values or case statements specified in this appendix are to be added to the ones specified in Section 3.3 of [2].

The numeric identifiers for the parameter sets defined in Sections 4.2 through 4.4 are marked as "TBD," as they had not yet been assigned at the time this document was published. Once they are assigned, the numeric identifiers may be found at https://www.iana.org/assignments/leighton-micali-signatures/leighton-micali-signatures.xhtml.

```
/* one-time signatures */

enum lmots_algorithm_type {
  lmots_sha256_n24_w1 = TBD,
  lmots_sha256_n24_w2 = TBD,
  lmots_sha256_n24_w4 = TBD,
  lmots_sha256_n24_w8 = TBD,
  lmots_shake_n32_w1  = TBD,
  lmots_shake_n32_w2  = TBD,
  lmots_shake_n32_w4  = TBD,
  lmots_shake_n32_w8  = TBD,
  lmots_shake_n24_w1  = TBD,
  lmots_shake_n24_w2  = TBD,
  lmots_shake_n24_w4  = TBD,
  lmots_shake_n24_w8  = TBD
};

typedef opaque bytestring24[24];

struct lmots_signature_n24_p200 {
  bytestring24 C;
  bytestring24 y[200];
};

struct lmots_signature_n24_p101 {
  bytestring24 C;
  bytestring24 y[101];
};

struct lmots_signature_n24_p51 {
  bytestring24 C;
  bytestring24 y[51];
};

struct lmots_signature_n24_p26 {
```

```
  bytestring24 C;
  bytestring24 y[26];
};

union lmots_signature switch (lmots_algorithm_type type) {
 case lmots_sha256_n24_w1:
   lmots_signature_n24_p200 sig_n24_p200;
 case lmots_sha256_n24_w2:
   lmots_signature_n24_p101 sig_n24_p101;
 case lmots_sha256_n24_w4:
   lmots_signature_n24_p51  sig_n24_p51;
 case lmots_sha256_n24_w8:
   lmots_signature_n24_p26  sig_n24_p26;
 case lmots_shake_n32_w1:
   lmots_signature_n32_p265 sig_n32_p265;
 case lmots_shake_n32_w2:
   lmots_signature_n32_p133 sig_n32_p133;
 case lmots_shake_n32_w4:
   lmots_signature_n32_p67  sig_n32_p67;
 case lmots_shake_n32_w8:
   lmots_signature_n32_p34  sig_n32_p34;
 case lmots_shake_n24_w1:
   lmots_signature_n24_p200 sig_n24_p200;
 case lmots_shake_n24_w2:
   lmots_signature_n24_p101 sig_n24_p101;
 case lmots_shake_n24_w4:
   lmots_signature_n24_p51  sig_n24_p51;
 case lmots_shake_n24_w8:
   lmots_signature_n24_p26  sig_n24_p26;
};

/* hash-based signatures (hbs) */

enum lms_algorithm_type {
  lms_sha256_n24_h5  = TBD,
  lms_sha256_n24_h10 = TBD,
  lms_sha256_n24_h15 = TBD,
  lms_sha256_n24_h20 = TBD,
  lms_sha256_n24_h25 = TBD,
  lms_shake_n32_h5   = TBD,
  lms_shake_n32_h10  = TBD,
  lms_shake_n32_h15  = TBD,
  lms_shake_n32_h20  = TBD,
  lms_shake_n32_h25  = TBD,
  lms_shake_n24_h5   = TBD,
  lms_shake_n24_h10  = TBD,
  lms_shake_n24_h15  = TBD,
```

```
  lms_shake_n24_h20  = TBD,
  lms_shake_n24_h25  = TBD
};

/* leighton-micali signatures (lms) */

union lms_path switch (lms_algorithm_type type) {
 case lms_sha256_n24_h5:
 case lms_shake_n24_h5:
   bytestring24 path_n24_h5[5];
 case lms_sha256_n24_h10:
 case lms_shake_n24_h10:
   bytestring24 path_n24_h10[10];
 case lms_sha256_n24_h15:
 case lms_shake_n24_h15:
   bytestring24 path_n24_h15[15];
 case lms_sha256_n24_h20:
 case lms_shake_n24_h20:
   bytestring24 path_n24_h20[20];
 case lms_sha256_n24_h25:
 case lms_shake_n24_h25:
   bytestring24 path_n24_h25[25];

 case lms_shake_n32_h5:
   bytestring32 path_n32_h5[5];
 case lms_shake_n32_h10:
   bytestring32 path_n32_h10[10];
 case lms_shake_n32_h15:
   bytestring32 path_n32_h15[15];
 case lms_shake_n32_h20:
   bytestring32 path_n32_h20[20];
 case lms_shake_n32_h25:
   bytestring32 path_n32_h25[25];
};

struct lms_key_n24 {
  lmots_algorithm_type ots_alg_type;
  opaque I[16];
  opaque K[24];
};

union lms_public_key switch (lms_algorithm_type type) {
 case lms_sha256_n24_h5:
 case lms_sha256_n24_h10:
 case lms_sha256_n24_h15:
 case lms_sha256_n24_h20:
 case lms_sha256_n24_h25:
```

```
    case lms_shake_n24_h5:
    case lms_shake_n24_h10:
    case lms_shake_n24_h15:
    case lms_shake_n24_h20:
    case lms_shake_n24_h25:
         lms_key_n24 z_n24;

    case lms_shake_n32_h5:
    case lms_shake_n32_h10:
    case lms_shake_n32_h15:
    case lms_shake_n32_h20:
    case lms_shake_n32_h25:
         lms_key_n32 z_n32;
   };
```

Appendix B—XMSS XDR Syntax Additions

In order to support the XMSS parameter sets defined in Sections 5.2 through 5.4, the XDR syntax in Appendices A, B, and C of [1] is extended as follows. For data structures of type `enum` or `union` below, the values or case statements specified in this appendix are to be added to the ones specified in Appendices A, B, and C of [1].

B.1 WOTS+

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

enum ots_algorithm_type {
  wotsp-sha2_192     = 0x00000005,
  wotsp-shake256_256 = 0x00000006,
  wotsp-shake256_192 = 0x00000007,
};

/* Byte strings */

typedef opaque bytestring24[24];

union ots_signature switch (ots_algorithm_type type) {

  case wotsp-sha2_192:
  case wotsp-shake256_192:
    bytestring24 ots_sig_n24_len51[51];

  case wotsp-shake256_256:
    bytestring32 ots_sig_n32_len67[67];
};

union ots_pubkey switch (ots_algorithm_type type) {
  case wotsp-sha2_192:
  case wotsp-shake256_192:
    bytestring24 ots_pubk_n24_len51[51];

  case wotsp-shake256_256:
    bytestring32 ots_pubk_n32_len67[67];
};
```

B.2 XMSS

```
/* Definition of parameter sets */

enum xmss_algorithm_type {
  xmss-sha2_10_192      = 0x0000000D,
  xmss-sha2_16_192      = 0x0000000E,
```

```
  xmss-sha2_20_192      = 0x0000000F,

  xmss-shake256_10_256  = 0x00000010,
  xmss-shake256_16_256  = 0x00000011,
  xmss-shake256_20_256  = 0x00000012,

  xmss-shake256_10_192  = 0x00000013,
  xmss-shake256_16_192  = 0x00000014,
  xmss-shake256_20_192  = 0x00000015,
};

/* Authentication path types */

union xmss_path switch (xmss_algorithm_type type) {
  case xmss-sha2_10_192:
  case xmss-shake256_10_192:
    bytestring24 path_n24_t10[10];

  case xmss-shake256_10_256:
    bytestring32 path_n32_t10[10];

  case xmss-sha2_16_192:
  case xmss-shake256_16_192:
    bytestring24 path_n24_t16[16];

  case xmss-shake256_16_256:
    bytestring32 path_n32_t16[16];

  case xmss-sha2_20_192:
  case xmss-shake256_20_192:
    bytestring24 path_n24_t20[20];

  case xmss-shake256_20_256:
    bytestring32 path_n32_t20[20];
};

/* Types for XMSS random strings */

union random_string_xmss switch (xmss_algorithm_type type) {
  case xmss-sha2_10_192:
  case xmss-sha2_16_192:
  case xmss-sha2_20_192:
  case xmss-shake256_10_192:
  case xmss-shake256_16_192:
  case xmss-shake256_20_192:
    bytestring24 rand_n24;
```

```
    case xmss-shake256_10_256:
    case xmss-shake256_16_256:
    case xmss-shake256_20_256:
      bytestring32 rand_n32;
  };

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

  union xmss_ots_signature switch (xmss_algorithm_type type) {
    case xmss-sha2_10_192:
    case xmss-sha2_16_192:
    case xmss-sha2_20_192:
      wotsp-sha2_192;

    case xmss-shake256_10_256:
    case xmss-shake256_16_256:
    case xmss-shake256_20_256:
      wotsp-shake256_256;

    case xmss-shake256_10_192:
    case xmss-shake256_16_192:
    case xmss-shake256_20_192:
      wotsp-shake256_192;
  };

  /* Types for bitmask seed */

  union seed switch (xmss_algorithm_type type) {
    case xmss-sha2_10_192:
    case xmss-sha2_16_192:
    case xmss-sha2_20_192:
    case xmss-shake256_10_192:
    case xmss-shake256_16_192:
    case xmss-shake256_20_192:
      bytestring24 seed_n24;

    case xmss-shake256_10_256:
    case xmss-shake256_16_256:
    case xmss-shake256_20_256:
      bytestring32 seed_n32;
  };

  /* Types for XMSS root node */

  union xmss_root switch (xmss_algorithm_type type) {
    case xmss-sha2_10_192:
    case xmss-sha2_16_192:
```

```
    case xmss-sha2_20_192:
    case xmss-shake256_10_192:
    case xmss-shake256_16_192:
    case xmss-shake256_20_192:
      bytestring24 root_n24;

    case xmss-shake256_10_256:
    case xmss-shake256_16_256:
    case xmss-shake256_20_256:
      bytestring32 root_n32;
  };
```

B.3 XMSS^{MT}

```
  /* Definition of parameter sets */

  enum xmssmt_algorithm_type {

    xmssmt-sha2_20/2_192      = 0x00000021,
    xmssmt-sha2_20/4_192      = 0x00000022,
    xmssmt-sha2_40/2_192      = 0x00000023,
    xmssmt-sha2_40/4_192      = 0x00000024,
    xmssmt-sha2_40/8_192      = 0x00000025,
    xmssmt-sha2_60/3_192      = 0x00000026,
    xmssmt-sha2_60/6_192      = 0x00000027,
    xmssmt-sha2_60/12_192     = 0x00000028,

    xmssmt-shake256_20/2_256  = 0x00000029,
    xmssmt-shake256_20/4_256  = 0x0000002A,
    xmssmt-shake256_40/2_256  = 0x0000002B,
    xmssmt-shake256_40/4_256  = 0x0000002C,
    xmssmt-shake256_40/8_256  = 0x0000002D,
    xmssmt-shake256_60/3_256  = 0x0000002E,
    xmssmt-shake256_60/6_256  = 0x0000002F,
    xmssmt-shake256_60/12_256 = 0x00000030,

    xmssmt-shake256_20/2_192  = 0x00000031,
    xmssmt-shake256_20/4_192  = 0x00000032,
    xmssmt-shake256_40/2_192  = 0x00000033,
    xmssmt-shake256_40/4_192  = 0x00000034,
    xmssmt-shake256_40/8_192  = 0x00000035,
    xmssmt-shake256_60/3_192  = 0x00000036,
    xmssmt-shake256_60/6_192  = 0x00000037,
    xmssmt-shake256_60/12_192 = 0x00000038,
  };

  /* Type for XMSS^MT key pair index */
```

```
/* Depends solely on h */

union idx_sig_xmssmt switch (xmss_algorithm_type type) {
  case xmssmt-sha2_20/2_192:
  case xmssmt-sha2_20/4_192:
  case xmssmt-shake256_20/2_256:
  case xmssmt-shake256_20/4_256:
  case xmssmt-shake256_20/2_192:
  case xmssmt-shake256_20/4_192:
    bytestring3 idx3;

  case xmssmt-sha2_40/2_192:
  case xmssmt-sha2_40/4_192:
  case xmssmt-sha2_40/8_192:
  case xmssmt-shake256_40/2_256:
  case xmssmt-shake256_40/4_256:
  case xmssmt-shake256_40/8_256:
  case xmssmt-shake256_40/2_192:
  case xmssmt-shake256_40/4_192:
  case xmssmt-shake256_40/8_192:
    bytestring5 idx5;

  case xmssmt-sha2_60/3_192:
  case xmssmt-sha2_60/6_192:
  case xmssmt-sha2_60/12_192:
  case xmssmt-shake256_60/3_256:
  case xmssmt-shake256_60/6_256:
  case xmssmt-shake256_60/12_256:
  case xmssmt-shake256_60/3_192:
  case xmssmt-shake256_60/6_192:
  case xmssmt-shake256_60/12_192:
    bytestring8 idx8;
};

union random_string_xmssmt switch (xmssmt_algorithm_type type) {
  case xmssmt-sha2_20/2_192:
  case xmssmt-sha2_20/4_192:
  case xmssmt-sha2_40/2_192:
  case xmssmt-sha2_40/4_192:
  case xmssmt-sha2_40/8_192:
  case xmssmt-sha2_60/3_192:
  case xmssmt-sha2_60/6_192:
  case xmssmt-sha2_60/12_192:
  case xmssmt-shake256_20/2_192:
  case xmssmt-shake256_20/4_192:
  case xmssmt-shake256_40/2_192:
  case xmssmt-shake256_40/4_192:
```

```
    case xmssmt-shake256_40/8_192:
    case xmssmt-shake256_60/3_192:
    case xmssmt-shake256_60/6_192:
    case xmssmt-shake256_60/12_192:
      bytestring24 rand_n24;

    case xmssmt-shake256_20/2_256:
    case xmssmt-shake256_20/4_256:
    case xmssmt-shake256_40/2_256:
    case xmssmt-shake256_40/4_256:
    case xmssmt-shake256_40/8_256:
    case xmssmt-shake256_60/3_256:
    case xmssmt-shake256_60/6_256:
    case xmssmt-shake256_60/12_256:
      bytestring32 rand_n32;
  };

  /* Type for reduced XMSS signatures */

  union xmss_reduced (xmss_algorithm_type type) {
    case xmssmt-sha2_20/2_192:
    case xmssmt-sha2_40/4_192:
    case xmssmt-sha2_60/6_192:
    case xmssmt-shake256_20/2_192:
    case xmssmt-shake256_40/4_192:
    case xmssmt-shake256_60/6_192:
      bytestring24 xmss_reduced_n24_t61[61];

    case xmssmt-sha2_20/4_192:
    case xmssmt-sha2_40/8_192:
    case xmssmt-sha2_60/12_192:
    case xmssmt-shake256_20/4_192:
    case xmssmt-shake256_40/8_192:
    case xmssmt-shake256_60/12_192:
      bytestring24 xmss_reduced_n24_t56[56];

    case xmssmt-sha2_40/2_192:
    case xmssmt-sha2_60/3_192:
    case xmssmt-shake256_40/2_192:
    case xmssmt-shake256_60/3_192:
      bytestring24 xmss_reduced_n24_t71[71];

    case xmssmt-shake256_20/2_256:
    case xmssmt-shake256_40/4_256:
    case xmssmt-shake256_60/6_256:
      bytestring32 xmss_reduced_n32_t77[77];
```

```
    case xmssmt-shake256_20/4_256:
    case xmssmt-shake256_40/8_256:
    case xmssmt-shake256_60/12_256:
      bytestring32 xmss_reduced_n32_t72[72];

    case xmssmt-shake256_40/2_256:
    case xmssmt-shake256_60/3_256:
      bytestring32 xmss_reduced_n32_t87[87];
  };

  /* xmss_reduced_array depends on d */

  union xmss_reduced_array (xmss_algorithm_type type) {
    case xmssmt-sha2_20/2_192:
    case xmssmt-sha2_40/2_192:
    case xmssmt-shake256_20/2_256:
    case xmssmt-shake256_40/2_256:
    case xmssmt-shake256_20/2_192:
    case xmssmt-shake256_40/2_192:
      xmss_reduced xmss_red_arr_d2[2];

    case xmssmt-sha2_60/3_192:
    case xmssmt-shake256_60/3_256:
    case xmssmt-shake256_60/3_192:
      xmss_reduced xmss_red_arr_d3[3];

    case xmssmt-sha2_20/4_192:
    case xmssmt-sha2_40/4_192:
    case xmssmt-shake256_20/4_256:
    case xmssmt-shake256_40/4_256:
    case xmssmt-shake256_20/4_192:
    case xmssmt-shake256_40/4_192:
      xmss_reduced xmss_red_arr_d4[4];

    case xmssmt-sha2_60/6_192:
    case xmssmt-shake256_60/6_256:
    case xmssmt-shake256_60/6_192:
      xmss_reduced xmss_red_arr_d6[6];

    case xmssmt-sha2_40/8_192:
    case xmssmt-shake256_40/8_256:
    case xmssmt-shake256_40/8_192:
      xmss_reduced xmss_red_arr_d8[8];

    case xmssmt-sha2_60/12_192:
    case xmssmt-shake256_60/12_256:
    case xmssmt-shake256_60/12_192:
```

```
    xmss_reduced xmss_red_arr_d12[12];
};

/* Types for bitmask seed */

union seed switch (xmssmt_algorithm_type type) {
  case xmssmt-sha2_20/2_192:
  case xmssmt-sha2_20/4_192:
  case xmssmt-sha2_40/2_192:
  case xmssmt-sha2_40/4_192:
  case xmssmt-sha2_40/8_192:
  case xmssmt-sha2_60/3_192:
  case xmssmt-sha2_60/6_192:
  case xmssmt-sha2_60/12_192:
  case xmssmt-shake256_20/2_192:
  case xmssmt-shake256_20/4_192:
  case xmssmt-shake256_40/2_192:
  case xmssmt-shake256_40/4_192:
  case xmssmt-shake256_40/8_192:
  case xmssmt-shake256_60/3_192:
  case xmssmt-shake256_60/6_192:
  case xmssmt-shake256_60/12_192:
    bytestring24 seed_n24;

  case xmssmt-shake256_20/2_256:
  case xmssmt-shake256_20/4_256:
  case xmssmt-shake256_40/2_256:
  case xmssmt-shake256_40/4_256:
  case xmssmt-shake256_40/8_256:
  case xmssmt-shake256_60/3_256:
  case xmssmt-shake256_60/6_256:
  case xmssmt-shake256_60/12_256:
    bytestring32 seed_n32;

};

/* Types for XMSS^MT root node */

union xmssmt_root switch (xmssmt_algorithm_type type) {
  case xmssmt-sha2_20/2_192:
  case xmssmt-sha2_20/4_192:
  case xmssmt-sha2_40/2_192:
  case xmssmt-sha2_40/4_192:
  case xmssmt-sha2_40/8_192:
  case xmssmt-sha2_60/3_192:
  case xmssmt-sha2_60/6_192:
  case xmssmt-sha2_60/12_192:
```

```
    case xmssmt-shake256_20/2_192:
    case xmssmt-shake256_20/4_192:
    case xmssmt-shake256_40/2_192:
    case xmssmt-shake256_40/4_192:
    case xmssmt-shake256_40/8_192:
    case xmssmt-shake256_60/3_192:
    case xmssmt-shake256_60/6_192:
    case xmssmt-shake256_60/12_192:
      bytestring24 root_n24;

    case xmssmt-shake256_20/2_256:
    case xmssmt-shake256_20/4_256:
    case xmssmt-shake256_40/2_256:
    case xmssmt-shake256_40/4_256:
    case xmssmt-shake256_40/8_256:
    case xmssmt-shake256_60/3_256:
    case xmssmt-shake256_60/6_256:
    case xmssmt-shake256_60/12_256:
      bytestring32 root_n32;
  };
```

Appendix C—Provable Security Analysis

This appendix briefly summarizes the formal security model and proofs of security of the LMS and XMSS signature schemes and provides a short discussion comparing these models and proofs.

C.1 The Random Oracle Model

In the *random oracle model* (ROM), there is a publicly accessible random oracle that both the user and the adversary can send queries to and receive responses from at any time. A random oracle H is a hypothetical, *interactive* black-box algorithm that obeys the following rules:

1. Every time the algorithm H receives a new input string s, it generates an output t uniformly at random from its output space and returns the response t. The algorithm H then records the pair (s, t) for future use.
2. If the algorithm H is ever queried in the future with some prior input s, it will always return the same output t according to its recorded memory.

Alternatively, the random oracle H can be described as a non-interactive but *exponentially large* look-up table initialized with truly random outputs t for each possible input string s.

To say that a cryptographic security proof is done in the random oracle model means that every use of a particular function (e.g., in the case here, the compression function that is used to perform hashes) is replaced by a query to the random oracle H. This simplifies security claims because, for example, it becomes easy to prove upper bounds on the likelihood of producing a second preimage within a fixed number of queries to H. On the other hand, (compression) functions in the real world are neither interactive nor have exponentially large descriptions, so they cannot truly behave like a random oracle.

It is therefore desirable to have a cryptographic security proof that avoids using the random oracle model. However, this often leads to less efficient cryptographic systems, or it is not yet known how to perform a proof without appealing to the random oracle model, or both. So, as a matter of real-world pragmatism, the ROM is commonly used.

C.2 The Quantum Random Oracle Model

The *quantum random oracle model* (QROM) is similar to the ROM, except that it is additionally assumed that all parties (in particular, the adversary) have quantum computers and can query the random oracle H in superposition. (In the real world, the random oracle H is still instantiated as a compression function or something similar, in accordance with the cryptosystem's specification.) While this complicates security claims as compared to the ROM, it more accurately models the power of an adversary that has access to a large-scale quantum device for its cryptanalysis when attacking a real-world scheme.

C.3 LMS Security Proof

In [11], the author considers a particular experiment in the random oracle model in which the adversary is given a series of strings with prefixes (in a randomly chosen but structured manner) and hash targets. The attacker's goal is to find one more string that has the same prefix and hash target as any of its input strings. The author proves an upper bound on the adversary's ability to compute first or second preimages from these strings (by querying the compression function modeled as a random oracle).

Then, the author reduces the problem of forging a signature in LMS to this stated experiment, concluding that the same upper bounds apply to the problem of producing forgeries against LMS. This random oracle model proof critically depends on the randomness of the prefixes used in LMS, which means that LMS in the real world critically depends on the pseudorandomness of the prefixes.

Further, in [15], the same proof is carried out in the QROM.

C.4 XMSS Security Proof

In [12], a security analysis for the *original* (academic publication) version of XMSS is given under the following assumptions:

1. The function family $\{f_k\}$ used to construct Winternitz signatures is pseudorandom. This means that if the bit string k is chosen uniformly at random, then an adversary given black-box access to the function f_k cannot distinguish this black box from a random function within a polynomial number of queries (except with negligible probability).
2. The hash function family $\{h_k\}$ is second preimage-resistant. This means that if bit strings k and m are chosen uniformly at random, then an adversary given k and m cannot construct $m' \neq m$ such that $h_k(m') = h_k(m)$ in polynomial time (except with negligible probability).

The proof in [12] asserts that if both of these assumptions are true, then XMSS is existentially unforgeable under adaptive chosen message attacks (EUF-CMA) in the standard model.

However, in the *current* version of XMSS^{MT} [1], the security analysis differs somewhat. In the standard model, [17] shows that XMSS^{MT} is EUF-CMA. Further, [16] shows that XMSS^{MT} is post-quantum existentially unforgeable under adaptive chosen message attacks with respect to the QROM.

In a little more detail, the current version of XMSS uses two types of assumptions:

1. A standard model assumption that the hash function h_k, used for the one-time signatures and tree node computations, is post-quantum, multi-function, multi-target decisional second-preimage-resistant.

2. A (quantum) random oracle model assumption that the pseudorandom function f_k, used to generate pseudorandom values for randomized hashing and computing bitmasks as blinding keys, may be validly modeled as a quantum random oracle H.

C.5 Comparison of the Security Models and Proofs of LMS and XMSS

Generally speaking, both LMS and XMSS are supported by sound security proofs under commonly used cryptographic hardness assumptions. That is, if these cryptographic assumptions are true, then both schemes are provably shown to be existentially unforgeable under chosen message attack, even against an adversary that has access to a large-scale quantum computer for use in its forgery attack.

The main difference between these schemes' security analyses comes down to the use (and the degree of use) of the random oracle or quantum random oracle models. Along these lines, the difference between the (standard model/real world) cryptographic assumption that some function family $\{f_k\}$ is pseudorandom and the use of the random oracle model is briefly pointed out. For a function f_k to be a pseudorandom function in the real world, it should be the case that the bit string k used as the key to the function remains private, meaning that it is not in the view of the adversary at any point of the security experiment. On the other hand, a random oracle H achieves the same pseudorandomness (or even randomness) properties of a pseudorandom function f_k, but no key k is necessarily associated with the random oracle. Indeed, all inputs to the random oracle H may be known to all parties and, in particular, to the adversary. Therefore, using the random oracle model clearly involves making a stronger assumption about the (limits of the) cryptanalytic power of the adversary.

That said, a security proof is either *entirely* a "real-world proof," which does not use the random oracle model, or it appeals to the random oracle methodology in some manner. The security analysis of the current version of XMSS only uses the random oracle H when performing randomized hashing and computing bitmasks, whereas LMS uses the random oracle H to a greater degree (modeling the compression function as a random oracle). However, it remains the case that both schemes in their modern form are ultimately proven secure using the ROM and QROM.

Therefore, the cryptographic hardness assumptions made by LMS and XMSS in order to achieve existential unforgeability under chosen message attack (EUF-CMA) may be viewed as substantially similar and worthy of essentially equal confidence. As such, the practitioner's decision to deploy one scheme or the other should primarily depend on other factors, such as the efficiency demands for a given deployment environment or the other security considerations enumerated earlier in this document.