

Withdrawn Draft

Warning Notice

The attached draft document has been withdrawn, and is provided solely for historical purposes. It has been superseded by the document identified below.

Withdrawal Date October 29, 2020

Original Release Date December 11, 2019

Superseding Document

Status Final

Series/Number NIST Special Publication (SP) 800-208

Title Recommendation for Stateful Hash-Based Signature Schemes

Publication Date October 2020

DOI <https://doi.org/10.6028/NIST.SP.800-208>

CSRC URL <https://csrc.nist.gov/publications/detail/sp/800-208/final>

Additional Information

2

3 **Recommendation for Stateful**
4 **Hash-Based Signature Schemes**

5

6 David A. Cooper
7 Daniel C. Apon
8 Quynh H. Dang
9 Michael S. Davidson
10 Morris J. Dworkin
11 Carl A. Miller
12

13

14

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

17

18

19 **C O M P U T E R S E C U R I T Y**

20

21
22

23
24
25

26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

Draft 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-draft>

December 2019



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

45
46
47
48
49
50
51
52

53

Authority

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

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

67 National Institute of Standards and Technology Special Publication 800-208
68 Natl. Inst. Stand. Technol. Spec. Publ. 800-208, 54 pages (December 2019)
69 CODEN: NSPUE2

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

72 Certain commercial entities, equipment, or materials may be identified in this document in order to describe an
73 experimental procedure or concept adequately. Such identification is not intended to imply recommendation or
74 endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best
75 available for the purpose.

76 There may be references in this publication to other publications currently under development by NIST in accordance
77 with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies,
78 may be used by federal agencies even before the completion of such companion publications. Thus, until each
79 publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For
80 planning and transition purposes, federal agencies may wish to closely follow the development of these new
81 publications by NIST.

82 Organizations are encouraged to review all draft publications during public comment periods and provide feedback to
83 NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at
84 <https://csrc.nist.gov/publications>.

85

86 **Public comment period: *December 11, 2019* through *February 28, 2020***

87 National Institute of Standards and Technology
88 Attn: Computer Security Division, Information Technology Laboratory
89 100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
90 Email: pqc-comments@nist.gov

91 All comments are subject to release under the Freedom of Information Act (FOIA).

92

Reports on Computer Systems Technology

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

103

Abstract

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

108

Keywords

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

110

Document Conventions

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

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

117 The terms “may” and “need not” indicate a course of action permissible within the limits of the
118 publication.

119 The terms “can” and “cannot” indicate a possibility and capability, whether material, physical or
120 causal.

121

Conformance Testing

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

128

Note to Reviewers

129 Sections 4 and 5 specify the parameter sets that are approved by this recommendation for LMS,
130 HSS, XMSS, and XMSS^{MT}. Given the large number of parameter sets specified in these two
131 sections, NIST would like feedback on whether there would be a benefit in reducing the number
132 of parameter sets that are approved, and if so, which ones should be removed.

133 While this recommendation does not allow cryptographic modules to export private keying
134 material, Section 7 describes a way in which a single key pair can be created with the one-time
135 keys being spread across multiple cryptographic modules. The method described in Section 7
136 involves creating a 2-level HSS or XMSS^{MT} tree where the one-time keys associated with each of
137 the bottom-level trees can be created on a different cryptographic module.

138 NIST believes that it would be possible to create a one-level XMSS or LMS tree in which the
139 one-time keys are not all created and stored on the same cryptographic module. Key generation
140 would be more complicated to implement, though, as would be the steps that end users would
141 have to perform during the key generation process. However, a one-level tree would result in
142 shorter signatures.

143 NIST would like feedback on whether there is a need to be able to create one-level XMSS or
144 LMS keys in which the one-time keys are not all created and stored on the same cryptographic
145 module even though such an option would be more complicated to implement and use than the
146 two-level option that is already described in the draft.

147

Call for Patent Claims

148 This public review includes a call for information on essential patent claims (claims whose use
149 would be required for compliance with the guidance or requirements in this Information
150 Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be
151 directly stated in this ITL Publication or by reference to another publication. This call also
152 includes disclosure, where known, of the existence of pending U.S. or foreign patent applications
153 relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

154 ITL may require from the patent holder, or a party authorized to make assurances on its behalf,
155 in written or electronic form, either:

- 156 a) assurance in the form of a general disclaimer to the effect that such party does not hold and
157 does not currently intend holding any essential patent claim(s); or
- 158 b) assurance that a license to such essential patent claim(s) will be made available to applicants
159 desiring to utilize the license for the purpose of complying with the guidance or requirements
160 in this ITL draft publication either:
- 161 i) under reasonable terms and conditions that are demonstrably free of any unfair
162 discrimination; or
- 163 ii) without compensation and under reasonable terms and conditions that are demonstrably
164 free of any unfair discrimination.

165 Such assurance shall indicate that the patent holder (or third party authorized to make assurances
166 on its behalf) will include in any documents transferring ownership of patents subject to the
167 assurance, provisions sufficient to ensure that the commitments in the assurance are binding on
168 the transferee, and that the transferee will similarly include appropriate provisions in the event of
169 future transfers with the goal of binding each successor-in-interest.

170 The assurance shall also indicate that it is intended to be binding on successors-in-interest
171 regardless of whether such provisions are included in the relevant transfer documents.

172 Such statements should be addressed to: pqc-comments@nist.gov

173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205

Table of Contents

1 Introduction 1

 1.1 Intended Applications for Stateful HBS Schemes 1

 1.2 The Importance of the Proper Maintenance of State 1

 1.3 Outline of Text..... 2

2 Glossary of Terms, Acronyms, and Mathematical Symbols 4

 2.1 Terms and Definitions 4

 2.2 Acronyms 4

 2.3 Mathematical Symbols 5

3 General Discussion..... 6

 3.1 One-Time Signature Systems 6

 3.2 Merkle Trees 7

 3.3 Two-Level Trees 8

 3.4 Prefixes and Bitmasks 9

4 Leighton-Micali Signatures (LMS) Parameter Sets 10

 4.1 LMS with SHA-256..... 10

 4.2 LMS with SHA-256/192..... 11

 4.3 LMS with SHAKE256/256 12

 4.4 LMS with SHAKE256/192 12

5 eXtended Merkle Signature Scheme (XMSS) Parameter Sets 14

 5.1 XMSS and XMSS^{MT} with SHA-256 14

 5.2 XMSS and XMSS^{MT} with SHA-256/192 15

 5.3 XMSS and XMSS^{MT} with SHAKE256/256..... 16

 5.4 XMSS and XMSS^{MT} with SHAKE256/192..... 17

6 Random Number Generation for Keys and Signatures 19

 6.1 LMS and HSS Random Number Generation Requirements 19

 6.2 XMSS and XMSS^{MT} Random Number Generation Requirements 19

7 Distributed Multi-Tree Hash-Based Signatures 20

 7.1 HSS 20

 7.2 XMSS^{MT} 20

 7.2.1 Modified XMSS Key Generation and Signature Algorithms 21

 7.2.2 XMSS^{MT} External Device Operations 22

206 **8 Conformance** **24**

207 8.1 Key Generation and Signature Generation 24

208 8.2 Signature Verification 24

209 **9 Security Considerations** **25**

210 9.1 One-Time Signature Key Reuse 25

211 9.2 Fault Injection Resistance 25

212 9.3 Hash Collisions 26

213 **References** **27**

214

215 **List of Appendices**

216 **Appendix A— LMS XDR Syntax Additions** **30**

217 **Appendix B— XMSS XDR Syntax Additions** **34**

218 B.1 WOTS⁺ 34

219 B.2 XMSS 34

220 B.3 XMSS^{MT} 37

221 **Appendix C— Provable Security Analysis** **43**

222 C.1 The Random Oracle Model 43

223 C.2 The Quantum Random Oracle Model 43

224 C.3 LMS Security Proof 43

225 C.4 XMSS Security Proof 44

226 C.5 Comparison of the Security Models and Proofs of LMS and XMSS 45

227

228 **List of Figures**

229 Figure 1: A Sample Winternitz chain 6

230 Figure 2: A Sample Winternitz Signature 7

231 Figure 3: A Merkle Hash Tree 7

232 Figure 4: A Two-Level Merkle Tree 8

233 Figure 5: XMSS Hash Computation with Prefix and Bitmask 9

234

235 **List of Tables**

236 Table 1: LM-OTS parameter sets for SHA-256 10

237 Table 2: LMS parameter sets for SHA-256 11

238 Table 3: LM-OTS parameter sets for SHA-256/192 11

239 Table 4: LMS parameter sets for SHA-256/192 11

240 Table 5: LM-OTS parameter sets for SHAKE256/256..... 12

241 Table 6: LMS parameter sets for SHAKE256/256..... 12

242 Table 7: LM-OTS parameter sets for SHAKE256/192..... 12

243 Table 8: LMS parameter sets for SHAKE256/192..... 13

244 Table 9: WOTS⁺ parameter sets 14

245 Table 10: XMSS parameter sets for SHA-256..... 14

246 Table 11: XMSS^{MT} parameter sets for SHA-256 15

247 Table 12: XMSS parameter sets for SHA-256/192..... 15

248 Table 13: XMSS^{MT} parameter sets for SHA-256/192 16

249 Table 14: XMSS parameter sets for SHAKE256/256 16

250 Table 15: XMSS^{MT} parameter sets for SHAKE256/256..... 17

251 Table 16: XMSS parameter sets for SHAKE256/192 17

252 Table 17: XMSS^{MT} parameter sets for SHAKE256/192..... 18

253

254 **1 Introduction**

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

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

271 **1.1 Intended Applications for Stateful HBS Schemes**

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

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

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

285 **1.2 The Importance of the Proper Maintenance of State**

286 In a stateful HBS scheme, a key pair consists of a large set of one-time signature (OTS) key
287 pairs. An HBS key pair may contain thousands, millions, or billions of OTS keys, and the signer
288 needs to ensure that no individual OTS key is ever used to sign more than one message. If an
289 attacker were able to obtain digital signatures for two different messages created using the same
290 OTS key, then it would become computationally feasible for that attacker to forge signatures on
291 arbitrary messages [13]. Therefore, as described in [8], when a stateful HBS scheme is
292 implemented, extreme care needs to be taken in order to ensure that no OTS key is ever reused.

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

298 **1.3 Outline of Text**

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

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

- 329
- 330
- Appendix C, *Provable Security Analysis*, provides information about the security proofs that are available for LMS and XMSS. This section is *informative*.

331 **2 Glossary of Terms, Acronyms, and Mathematical Symbols**332 **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.

333
334 **2.2 Acronyms**

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

EEPROM	Electronically erasable programmable read-only memory
EUUF-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

336

337

2.3 Mathematical Symbols

SHA-256(M)	SHA-256 hash function as specified in [3]
SHA-256/192(M)	$T_{192}(\text{SHA-256}(M))$, 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]
SHAKE256/192(M)	SHAKE256(M , 192), where SHAKE256 is specified in Section 6.2 of [5]
$T_{192}(X)$	A truncation function that outputs the most significant (i.e., leftmost) 192 bits of the input bit string X

338

39 **3 General Discussion**

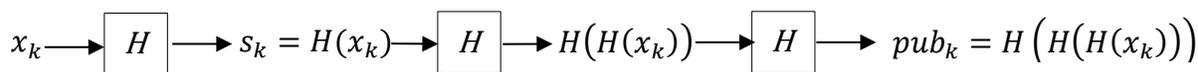
340 At a high level, XMSS and LMS are very similar. They each consist of two components—a one-
341 time signature (OTS) scheme and a method for creating a single, long-term public key from a
342 large set of OTS public keys. A brief explanation of OTS schemes and the method for creating a
343 long-term public key from a large set of OTS public keys can be found in Sections 3 and 4 of
344 [14].

345 **3.1 One-Time Signature Systems**

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

349 A hash chain is created by first randomly generating a secret value, x , which is the private key.
350 The size of x should generally correspond to the targeted strength of the scheme. So for the
351 parameter sets approved by this recommendation, x will be either 192 or 256 bits in length. The
352 public key, pub , is then created by applying the hash function, H , to the secret $b - 1$ times,
353 $H^{b-1}(x)$. Figure 1 shows an example of a hash chain for the k th digit of a digest where b is 4.

354 The k th digit of the digest, N_k , is signed by applying the hash function, H , to the private key N_k
355 times, $H^{N_k}(x_k)$. In Figure 1, N_k is 1, and so the signature is $s_k = H^1(x_k) = H(x_k)$. The
356 signature can be verified by checking that $pub_k = H^{b-1-N_k}(s_k)$. So in Figure 1, the signature
357 can be verified by checking that $pub_k = H^{4-1-1}(s_k) = H^2(s_k) = H(H(s_k))$.



358

Figure 1: A sample Winternitz chain

359 As noted in [14], simply signing the individual digits of the digest is not sufficient as an attacker
360 would be able to generate valid signatures for other message digests. For example, given $s_k =$
361 $H(x_k)$, as in Figure 1, an attacker would be able to generate a signature for a message digest with
362 a k th digit of 2 by applying H to s_k once or to a message digest with a k th digit of 3 by applying
363 H to s_k twice. An attacker could not, however, generate a signature for a message digest with a
364 k th digit of 0 as this would require finding some value y such that $H(y) = s_k$, which would not
365 be feasible as long as H is preimage resistant.

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

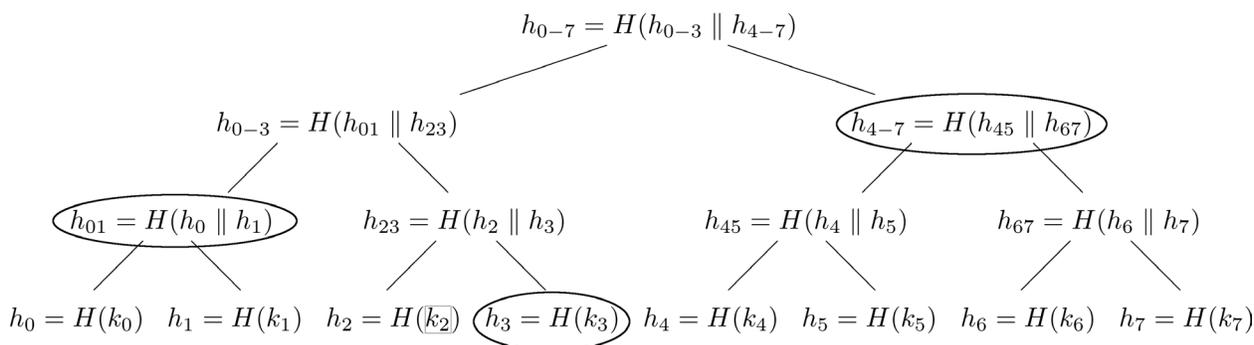
373 Figure 2 shows an example of a signature for a 32-bit message digest using $b = 16$. The digest is
 374 written as eight hexadecimal digits, and a separate hash chain is used to sign each digit with each
 375 hash chain having its own private key.¹

	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	$H^{15}(x_0)$	$H^{15}(x_1)$	$H^{15}(x_2)$	$H^{15}(x_3)$	$H^{15}(x_4)$	$H^{15}(x_5)$	$H^{15}(x_6)$	$H^{15}(x_7)$	$H^{15}(x_8)$	$H^{15}(x_9)$

376 **Figure 2: A sample Winternitz signature**

377 **3.2 Merkle Trees**

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



387 **Figure 3: A Merkle Hash Tree**

388 Figure 3 depicts a hash tree containing eight OTS public keys. The eight keys are each hashed to
 389 form the leaves of the tree, and the eight leaf values are hashed in pairs to create the next level up
 390 in the tree. These four hash values are again hashed in pairs to create h_{0-3} and h_{4-7} , which are
 391

¹ 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.

392 hashed together to create the long-term public key, h_{0-7} . In order for an entity that had already
 393 received h_{0-7} in a secure manner to verify a message signed using k_2 , the signer would need to
 394 provide h_3 , h_{01} , and h_{4-7} in addition to k_2 . The verifier would compute $h'_2 = H(k_2)$, $h'_{23} =$
 395 $H(h'_2 || h_3)$, $h'_{0-3} = H(h_{01} || h'_{23})$, 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
 396 may be used to verify the message signature.

397 3.3 Two-Level Trees

398 Both [1] and [2] define single tree as well as multi-tree variants of their signature schemes. In an
 399 instance that involves two levels of trees, as shown in Figure 4, the OTS keys that form the
 400 leaves of the top-level tree sign the roots of the trees at the bottom level, and the OTS keys that
 401 form the leaves of the bottom-level trees are used to sign the messages. The root of the top-level
 402 tree is the public key for the signature scheme.²

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

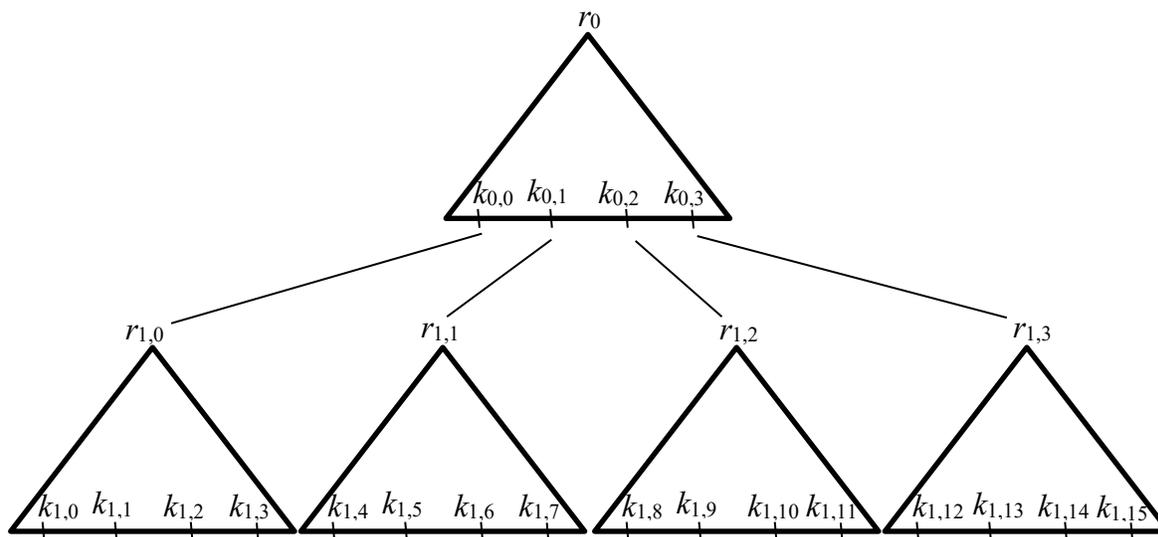


Figure 4: A two-level Merkle tree

410 While there are benefits in the use of a two-level tree, it results in larger signatures and slower
 411 signature verification as each message signature will need to include two OTS signatures. For
 412 example, if a message were signed using OTS key $k_{1,6}$ in Figure 4, the signature would need to

² 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.

413 include the signature on $r_{1,1}$ using $k_{0,1}$ in addition to the signature on the message using $k_{1,6}$.

414 3.4 Prefixes and Bitmasks

415 In order to strengthen the security of the schemes in both XMSS and LMS whenever a value is
 416 hashed, a prefix is prepended to the value that is hashed. For example, when computing the
 417 public key for a Winternitz signature from the private key in LMS as described in Section 3.1,
 418 rather than just computing $pub_k = H^3(x_k) = H(H(H(x_k)))$ the public key is computed as
 419 $pub_k = H(p_3 || H(p_2 || H(p_1 || x_k)))$, where p_1 , p_2 , and p_3 are each different values. The
 420 prefix is formed by concatenating together various pieces of information, including a unique
 421 identifier for the long-term public key and an indicator of the purpose of the hash (e.g.,
 422 Winternitz chain or Merkle tree). If the hash is part of a Winternitz chain, then the prefix also
 423 includes the number of the OTS key, which digit of the digest or checksum is being signed, and
 424 where in the chain the hash appears. The goal is to ensure that every single hash that is computed
 425 within the LMS scheme uses a different prefix.

426 XMSS generates its prefixes in a similar way. The information described above is used to form
 427 an address, which uniquely identifies where a particular hash invocation occurs within the
 428 scheme. This address is then hashed along with a unique identifier for the long-term public key
 429 (SEED) to create the prefix.

430 Unlike LMS, XMSS also uses bitmasks. In addition to creating the prefix, a slightly different
 431 address is also hashed along with the SEED to create a bitmask. The bitmask is then exclusive-
 432 Ored with the input before the input is hashed along with the prefix. Figure 5 illustrates an
 433 example of this computation. In [1], the hash function is referred to as H, H_msg, F, or PRF,
 434 depending on where it is being used. However, in each case it is the same function, just with a
 435 different prefix prepended in order to ensure separation between the uses.

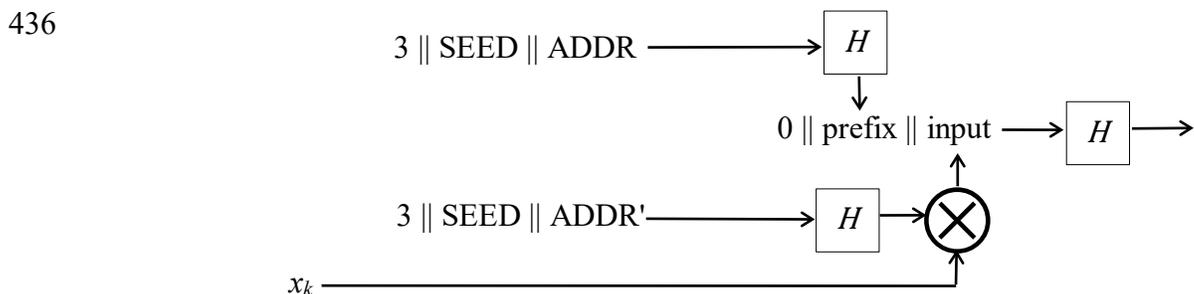


Figure 5: XMSS hash computation with prefix and bitmask

437 **4 Leighton-Micali Signatures (LMS) Parameter Sets**

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

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

446 When generating a key pair for an HSS instance, the requirements specified in the previous
 447 paragraph apply to each LMS tree in the instance. If the HSS instance has more than one level,
 448 then the hash function used for the tree at level 0 **shall** be used for every LMS tree at every other
 449 level. For each level, the same LMS and LM-OTS parameter sets **shall** be used for every LMS
 450 tree at that level.

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

454 Extensions to the XDR syntax in Section 3.3 of [2] needed to support the parameter sets defined
 455 in Sections 4.2 through 4.4 of this document are specified in Appendix A.

456 **4.1 LMS with SHA-256**

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

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

LM-OTS Parameter Sets	Numeric Identifier	n	w	p	ls	sig len
LMOTS_SHA256_N32_W1	0x00000001	32	1	265	7	8516
LMOTS_SHA256_N32_W2	0x00000002	32	2	133	6	4292
LMOTS_SHA256_N32_W4	0x00000003	32	4	67	4	2180
LMOTS_SHA256_N32_W8	0x00000004	32	8	34	0	1124

460

461

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

462

4.2 LMS with SHA-256/192

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

466

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

LM-OTS Parameter Sets	Numeric Identifier	n	w	p	ls	sig_len
LMOTS_SHA256_N24_W1	TBD	24	1	200	8	4828
LMOTS_SHA256_N24_W2	TBD	24	2	101	6	2452
LMOTS_SHA256_N24_W4	TBD	24	4	51	4	1252
LMOTS_SHA256_N24_W8	TBD	24	8	26	0	652

467

468

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

469

470 **4.3 LMS with SHAKE256/256**

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

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

LM-OTS Parameter Sets	Numeric Identifier	n	w	p	ls	sig_len
LMOTS_SHAKE_N32_W1	TBD	32	1	265	7	8516
LMOTS_SHAKE_N32_W2	TBD	32	2	133	6	4292
LMOTS_SHAKE_N32_W4	TBD	32	4	67	4	2180
LMOTS_SHAKE_N32_W8	TBD	32	8	34	0	1124

474
475

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

476
477

4.4 LMS with SHAKE256/192

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

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

LM-OTS Parameter Sets	Numeric Identifier	n	w	p	ls	sig_len
LMOTS_SHAKE_N24_W1	TBD	24	1	200	8	4828
LMOTS_SHAKE_N24_W2	TBD	24	2	101	6	2452
LMOTS_SHAKE_N24_W4	TBD	24	4	51	4	1252
LMOTS_SHAKE_N24_W8	TBD	24	8	26	0	652

481

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

482

483 **5 eXtended Merkle Signature Scheme (XMSS) Parameter Sets**

484 The XMSS and XMSS^{MT} algorithms are described in RFC 8391 [1]. This Special Publication
 485 approves the use of XMSS and XMSS^{MT} with four different hash functions: SHA-256, SHA-
 486 256/192, SHAKE256/256, and SHAKE256/192 (see Section 2.3).³ The parameter sets that use
 487 SHA-256 are defined in RFC 8391 [1]. The parameter sets that use SHA-256/192,
 488 SHAKE256/256, and SHAKE256/192 are defined below.

489 The WOTS⁺ parameters corresponding to the use of each of these hash functions is specified in
 490 the following table.

491 **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	TBD	See Section 5.2	24	16	51
WOTSP-SHAKE256_256	TBD	See Section 5.3	32	16	67
WOTSP-SHAKE256_192	TBD	See Section 5.4	24	16	51

492
 493 The XMSS and XMSS^{MT} parameter sets that are approved for use by this Special Publication are
 494 specified in Sections 5.1 through 5.4. The parameters n, w, len, h, and d specified in the tables
 495 are defined in Sections 3.1.1, 4.1.1, and 4.2.1 of [1].

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

498 **5.1 XMSS and XMSS^{MT} with SHA-256**

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

502 **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	0x00000002	32	16	67	20

³ The parameter sets specified in RFC 8391 [1] that use SHAKE128, SHAKE256, and SHA-512 are not approved for use by this Special Publication.

503

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

504

505 For the parameter sets in this section, the functions F, H, H_msg, and PRF are as defined in
506 Section 5.1 of [1] for SHA2 with n = 32.

507 5.2 XMSS and XMSS^{MT} with SHA-256/192

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

511

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

Parameter Sets	Numeric Identifier	n	w	len	h
XMSS-SHA2_10_192	TBD	24	16	51	10
XMSS-SHA2_16_192	TBD	24	16	51	16
XMSS-SHA2_20_192	TBD	24	16	51	20

512

513

514

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

Parameter Sets	Numeric Identifier	n	w	len	h	d
XMSSMT-SHA2_20/2_192	TBD	24	16	51	20	2
XMSSMT-SHA2_20/4_192	TBD	24	16	51	20	4
XMSSMT-SHA2_40/2_192	TBD	24	16	51	40	2
XMSSMT-SHA2_40/4_192	TBD	24	16	51	40	4
XMSSMT-SHA2_40/8_192	TBD	24	16	51	40	8
XMSSMT-SHA2_60/3_192	TBD	24	16	51	60	3
XMSSMT-SHA2_60/6_192	TBD	24	16	51	60	6
XMSSMT-SHA2_60/12_192	TBD	24	16	51	60	12

515

516 For the parameter sets in this section, the functions F, H, H_msg, and PRF are defined as
517 follows:

- 518 • F: $T_{192}(\text{SHA-256}(\text{toByte}(0, 4) \parallel \text{KEY} \parallel \text{M}))$
- 519 • H: $T_{192}(\text{SHA-256}(\text{toByte}(1, 4) \parallel \text{KEY} \parallel \text{M}))$
- 520 • H_msg: $T_{192}(\text{SHA-256}(\text{toByte}(2, 4) \parallel \text{KEY} \parallel \text{M}))$
- 521 • PRF: $T_{192}(\text{SHA-256}(\text{toByte}(3, 4) \parallel \text{KEY} \parallel \text{M}))$

522 5.3 XMSS and XMSS^{MT} with SHAKE256/256

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

526

Table 14: XMSS parameter sets for SHAKE256/256

Parameter Sets	Numeric Identifier	n	w	len	h
XMSS-SHAKE256_10_256	TBD	32	16	67	10
XMSS-SHAKE256_16_256	TBD	32	16	67	16
XMSS-SHAKE256_20_256	TBD	32	16	67	20

527

528

529

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

Parameter Sets	Numeric Identifier	n	w	len	h	d
XMSSMT-SHAKE256_20/2_256	TBD	32	16	67	20	2
XMSSMT-SHAKE256_20/4_256	TBD	32	16	67	20	4
XMSSMT-SHAKE256_40/2_256	TBD	32	16	67	40	2
XMSSMT-SHAKE256_40/4_256	TBD	32	16	67	40	4
XMSSMT-SHAKE256_40/8_256	TBD	32	16	67	40	8
XMSSMT-SHAKE256_60/3_256	TBD	32	16	67	60	3
XMSSMT-SHAKE256_60/6_256	TBD	32	16	67	60	6
XMSSMT-SHAKE256_60/12_256	TBD	32	16	67	60	12

530

531 For the parameter sets in this section, the functions F, H, H_msg, and PRF are defined as
532 follows:

- 533 • F: SHAKE256(toByte(0, 32) || KEY || M, 256)
- 534 • H: SHAKE256(toByte(1, 32) || KEY || M, 256)
- 535 • H_msg: SHAKE256(toByte(2, 32) || KEY || M, 256)
- 536 • PRF: SHAKE256(toByte(3, 32) || KEY || M, 256)

537 5.4 XMSS and XMSS^{MT} with SHAKE256/192

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

541

Table 16: XMSS parameter sets for SHAKE256/192

Parameter Sets	Numeric Identifier	n	w	len	h
XMSS-SHAKE256_10_192	TBD	24	16	51	10
XMSS-SHAKE256_16_192	TBD	24	16	51	16
XMSS-SHAKE256_20_192	TBD	24	16	51	20

542

543

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

Parameter Sets	Numeric Identifier	n	w	len	h	d
XMSSMT-SHAKE256_20/2_192	TBD	24	16	51	20	2
XMSSMT-SHAKE256_20/4_192	TBD	24	16	51	20	4
XMSSMT-SHAKE256_40/2_192	TBD	24	16	51	40	2
XMSSMT-SHAKE256_40/4_192	TBD	24	16	51	40	4
XMSSMT-SHAKE256_40/8_192	TBD	24	16	51	40	8
XMSSMT-SHAKE256_60/3_192	TBD	24	16	51	40	3
XMSSMT-SHAKE256_60/6_192	TBD	24	16	51	40	6
XMSSMT-SHAKE256_60/12_192	TBD	24	16	51	40	12

544

545 For the parameter sets in this section, the functions F, H, H_msg, and PRF are defined as
546 follows:

547

- F: SHAKE256(toByte(0, 4) || KEY || M, 192)

548

- H: SHAKE256(toByte(1, 4) || KEY || M, 192)

549

- H_msg: SHAKE256(toByte(2, 4) || KEY || M, 192)

550

- PRF: SHAKE256(toByte(3, 4) || KEY || M, 192)

551

552 **6 Random Number Generation for Keys and Signatures**

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

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

557 **6.1 LMS and HSS Random Number Generation Requirements**

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

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

566 If more than one LMS instance is being created (e.g., for an HSS instance), then a separate key
567 pair identifier, I , and SEED (if using the pseudorandom key generation method) **shall** be
568 generated for each LMS instance.

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

572 **6.2 XMSS and XMSS^{MT} Random Number Generation Requirements**

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

576 The private n -byte strings in the WOTS⁺ private keys ($sk[i]$ in Section 3.1.3 of [1]) **shall** be
577 generated using the pseudorandom key generation method specified in Section 3.1.7 of [1]:
578 $sk[i, j] = \text{PRF}(S_ots[j], \text{toByte}(i, 32))$, where PRF is as defined in Section 5 for the parameter set
579 being used. The private seed, $S_ots[j]$, for each WOTS⁺ private key, j , **shall** be as specified in
580 Section 4.1.11 of [1]: $S_ots[j] = \text{PRF}(S_XMSS, \text{toByte}(j, 32))$, where PRF is as defined in Section
581 5 for the parameter set being used. The private seed, S_XMSS , **shall** be generated using an
582 **approved** random bit generator [6] where the instantiation of the random bit generator supports
583 at least $8n$ bits of security strength. If more than one XMSS key pair is being created within a
584 cryptographic module (including XMSS keys that belong to a single XMSS^{MT} instance), then a
585 separate random S_XMSS **shall** be generated for each XMSS key pair.

7 Distributed Multi-Tree Hash-Based Signatures

587 If a digital signature key will be used to generate signatures over a long period of time and
588 replacing the public key would be difficult, then storing the private key in multiple places to
589 protect against loss will be necessary. In the case of most digital signature schemes, this just
590 involves making copies of the private key. However, in the case of stateful HBS schemes, simply
591 copying the private key would create a risk of OTS key reuse. An alternative that avoids this risk
592 is to have multiple cryptographic modules that each generate their own OTS keys and then create
593 a single instance that includes all of the public keys from all of the modules.

594 While it would also be possible to have one cryptographic module generate all of the OTS keys
595 and then distribute different OTS keys to each of the other cryptographic modules, doing so is
596 not an option for cryptographic modules conforming to this recommendation. Due to the risks
597 associated with copying OTS keys, this recommendation prohibits exporting private keying
598 material (Section 8).

599 The easiest way to have OTS keys on multiple cryptographic modules without exporting private
600 keys is to use HSS or XMSS^{MT} with two levels of trees where each tree is instantiated on a
601 different cryptographic module. First, a top-level LMS or XMSS key pair would be created in a
602 cryptographic module. The top level's OTS keys would only be used to sign the roots of other
603 trees. Then, bottom-level LMS or XMSS key pairs would be created in other cryptographic
604 modules, and the public keys from those key pairs (i.e., the roots of their Merkle trees) would be
605 signed by OTS keys of the top-level key pair. The OTS keys of the bottom-level key pairs would
606 be used to sign ordinary messages. The number of bottom-level key pairs that could be created
607 would only be limited by the number of OTS keys in the top-level key pair.

608 7.1 HSS

609 In the case of HSS, the scheme described above can be implemented using multiple
610 cryptographic modules that each implement LMS without modifications. The top-level LMS
611 public key can be converted to an HSS public key by an external, non-cryptographic device. This
612 device can also submit the public keys of the bottom-level LMS keys to be signed by the top-
613 level LMS key. In HSS, the operation for signing the root of a lower-level tree is the same as the
614 operation for signing an ordinary message. Finally, this external device can submit ordinary
615 messages to cryptographic modules holding the bottom-level LMS keys for signing and then
616 combine the resulting LMS signatures with the top-level key's signature on the bottom-level
617 LMS public key in order to create the HSS signature for the ordinary messages (see Algorithm 8
618 and Algorithm 9 in [2]).

619 7.2 XMSS^{MT}

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

627 signed.

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

635 Section 7.2.2 describes the steps that an external, non-cryptographic device needs to perform in
636 order to implement XMSS^{MT} key and signature generation using a set of cryptographic modules
637 that implement the algorithms in Section 7.2.1. While Algorithms 10' and 12' in Section 7.2.1
638 have been designed to work with XMSS^{MT} instances that have more than two layers, the
639 algorithms in Section 7.2.2 assume that an XMSS^{MT} instance with exactly two layers is being
640 created.

641 7.2.1 Modified XMSS Key Generation and Signature Algorithms

642 Algorithm 10': $\text{XMSS}'_{\text{keyGen}}$

```

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

648 // Example initialization for SK-specific contents
649 idx = t * 2^(h / d);
650 for ( i = 0; i < 2^(h / d); i++ ) {
651     wots_sk[i] = WOTS_genSK();
652 }

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

658 // SEED needs to be generated for the top-level XMSS key.
659 // For all other XMSS keys, the value needs to be copied from
660 // the top-level XMSS key.
661 if ( L = d - 1 ) {
662     Initialize SEED with an n-byte string using an approved
663     random bit generator [6], where the instantiation of the
664     random bit generator supports at least 8n bits of security
665     strength.
666 } else {

```

```

667     SEED = getSEED(PK_MT);
668   }
669   setSEED(SK, SEED);
670   setWOTS_SK(SK, wots_sk);
671   ADRS = toByte(0, 32);
672   ADRS.setLayerAddress(L);
673   ADRS.setTreeAddress(t);
674   root = treeHash(SK, 0, h / d, ADRS);

675   // The "root" value in SK needs to be the root of the top-level
676   // XMSS tree, as this is the value used when hashing the message
677   // to be signed.
678   if ( L = d - 1 ) {
679     SK = L || t || idx || wots_sk || SK_PRF || root || SEED
680   } else {
681     SK = L || t || idx || wots_sk || SK_PRF || getRoot(PK_MT) || SEED
682   }
683   PK = OID || root || SEED

684 Algorithm 12': XMSS'_sign

685   Input: Message M
686   Output: signature Sig

687   idx_sig = getIdx(SK);
688   setIdx(SK, idx_sig + 1);
689   L = getLayerAddress(SK);
690   t = getTreeAddress(SK);
691   ADRS = toByte(0, 32);
692   ADRS.setLayerAddress(L);
693   ADRS.setTreeAddress(t);

694   if ( L > 0 ) {
695     // M must be the n-byte root from an XMSS public key
696     byte[n] r = 0 // n-byte string of zeros
697     byte[n] M' = M
698   } else {
699     byte[n] r = PRF(getSK_PRF(SK), toByte(idx_sig, 32));
700     byte[n] M' = H_msg(r || getRoot(SK) || (toByte(idx_sig, n)), M);
701   }
702   idx_leaf = idx_sig - t * 2^(h / d);
703   Sig = idx_sig || r || treeSig(M', SK, idx_leaf, ADRS);

704 7.2.2 XMSSMT External Device Operations
705 XMSSMT external device keygen

706   Input: No input

```

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

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

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

716 // If the public key on the bottom-level tree was created using
717 // a tree address of t, then its root needs to be signed by OTS
718 // key t of the top-level tree. If it wasn't, then try again.
719 if ( getIdx(SigPK[t]) ≠ t ) {
720 t = getIdx(SigPK[t]) + 1;
721 PK[t] = XMSS'_keygen(0, t, PK_MT);
722 SigPK[t] = XMSS'_sign(getRoot(PK[t]));
723 }
724 t = t + 1;
725 }

726 XMSS^MT external device sign

727 Input: Message M
728 Output: signature Sig

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

731 idx_sig = getIdx(Sig_tmp);
732 t = (h / d) most significant bits of idx_sig;

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

736 **8 Conformance**

737 **8.1 Key Generation and Signature Generation**

738 Cryptographic modules implementing signature generation for a parameter set **shall** also
739 implement key generation for that parameter set. Implementations of the key generation and
740 signature algorithms in this document **shall** only be validated for use within hardware
741 cryptographic modules. The cryptographic modules **shall** be validated to provide FIPS 140-2 or
742 FIPS 140-3 [19] Level 3 or higher physical security, and the operational environment **shall** be
743 *limited*.⁴ In addition, a cryptographic module implementing the key generation or signature
744 algorithms **shall** only operate in an **approved** mode of operation and **shall not** implement a
745 bypass mode. The cryptographic module **shall not** allow for the export of private keying
746 material.

747 In order to prevent the possible reuse of an OTS key, when the cryptographic module accepts a
748 request to sign a message, the cryptographic module **shall** update the state of the private key in
749 non-volatile storage before exporting a signature value or accepting another request to sign a
750 message.

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

757 Cryptographic modules implementing XMSS key and signature generation **shall** implement
758 Algorithm 10 and Algorithm 12 from [1] for at least one of the XMSS parameter sets in Section
759 5. Cryptographic modules supporting implementation of XMSS^{MT} key and signature generation
760 **shall** implement Algorithm 10' and Algorithm 12' from Section 7.2.1 of this document for at
761 least one of the XMSS^{MT} parameter sets in Section 5. Cryptographic modules implementing
762 XMSS or XMSS^{MT} key and signature generation **shall** generate random data in accordance with
763 Section 6.2.

764 **8.2 Signature Verification**

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

769 Cryptographic modules implementing XMSS signature verification **shall** implement Algorithm
770 14 of [1] for at least one of the parameter sets in Section 5. Cryptographic modules implementing
771 XMSS^{MT} signature verification **shall** implement Algorithm 17 of [1] for at least one of the
772 parameter sets in Section 5.

⁴ See Section 4.6 of FIPS 140-2 [19].

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 non-volatile memory to mark an OTS key as unavailable before the corresponding signature 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 as the write operation will be cached with the actual write to non-volatile memory taking place later. If the cryptographic module loses power or crashes before the write to non-volatile memory, 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 starts up.

Some hardware cryptographic modules implement monotonic counters, which are guaranteed to increase 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 Fault Injection Resistance

Fault injection attacks involve the intentional introduction of an error at some point during the execution of an algorithm, such as by varying the voltage supplied to a device executing the algorithm, causing it to produce the wrong output, and providing the attacker with additional information. These attacks are most relevant for users of embedded cryptographic devices where an adversary may have physical access to the signing device and thus can control its operations.

Fault injection attacks have been shown to be effective against hash-based signatures, though they are more severe when used against stateless schemes like SPHINCS and its variants [9][10]. With hash-based signatures, the attack works by forcing the cryptographic device to sign two different messages with the same OTS key. The attack takes advantage of the schemes where multiple levels of Merkle trees are used and the roots of lower-level trees are signed using a one-time signature (XMSS^{MT} and HSS) [10]. In some cases, the signatures on these roots are recomputed each time a message is signed. Under normal circumstances, this is acceptable since it just involves using an OTS key multiple times to sign the same message. However, by injecting a fault that introduces an error in the computation of the Merkle tree root at any of the non-top layers, an attacker can cause the device to sign a different message under the same key. With both a valid and a faulty signature, the attacker can "graft" a new subtree into the hierarchy

814 and produce universal forgeries.

815 The faulted signature remains a valid signature, so checking that the signature verifies is
816 insufficient to detect or prevent this attack. The only reliable way to prevent this attack is to
817 compute each one-time signature once, cache the result, and output it whenever needed. When
818 implementing multiple levels of trees as described in Section 7, this is the only option since no
819 cryptographic module will use any OTS more than once. If multiple levels of trees are
820 implemented within a single cryptographic module, it is recommended to cache a single, one-
821 time signature per layer of subtrees, refreshing them when a new subtree is used for signing [10].
822 While this prevents an attacker from learning about the secret key when a corrupted signature is
823 cached, it does result in the cached one-time signature being incorrect and thus prevents the
824 hash-based signature scheme from working.

825 **9.3 Hash Collisions**

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

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

837 LMS and XMSS both use randomized hashing. When a message is presented to be signed, a
838 random value is created and prepended to the message, and the hash function is applied to this
839 expanded message to produce the message digest. Prepending the random value makes it
840 infeasible for anyone other than the signer to find a generic collision as finding a collision would
841 require predicting the randomizing value. The randomized hashing process does not, however,
842 impact the ability for a signer to create a generic collision since the signer, knowing the private
843 key, could choose the random value to prepend to the message.

844 The 196-bit hash functions in this recommendation, SHA-256/196 and SHAKE256/196, offer
845 significantly less resistance to generic collision searches than their 256-bit counterparts. In
846 particular, a collision of the 196-bit functions may be found as the number of sampled inputs
847 approaches 2^{96} , as opposed to 2^{128} for the 256-bit functions, and it may be possible for a signer
848 with access to an extremely large amount of computing resources to sample 2^{96} inputs.

849 Consequently, one tradeoff for the use of 196-bit hash functions in LMS and XMSS is the
850 weakening of the verifier's assurance that the signer will not be able to change the message once
851 the signature is revealed. This possibility does not affect the formal security properties of the
852 schemes because it remains the case that only the signer could produce a valid signature on a
853 message.

854 **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] Castelnuovi 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, Hülsing 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] Hülsing A, Rijneveld J, Song F (2015) Mitigating Multi-Target Attacks in Hash-based Signatures. *Cryptology ePrint Archive*, Report 2015/1256. <https://eprint.iacr.org/2015/1256>
- [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>

Appendix A—LMS XDR Syntax Additions

857 In order to support the LM-OTS and LMS parameter sets defined in Sections 4.2 through 4.4, the
858 XDR syntax in Section 3.3 of [2] is extended as follows.

```

859     /* one-time signatures */
860
861     enum lmots_algorithm_type {
862         lmots_sha256_n24_w1 = TBD,
863         lmots_sha256_n24_w2 = TBD,
864         lmots_sha256_n24_w4 = TBD,
865         lmots_sha256_n24_w8 = TBD,
866         lmots_shake_n32_w1  = TBD,
867         lmots_shake_n32_w2  = TBD,
868         lmots_shake_n32_w4  = TBD,
869         lmots_shake_n32_w8  = TBD,
870         lmots_shake_n24_w1  = TBD,
871         lmots_shake_n24_w2  = TBD,
872         lmots_shake_n24_w4  = TBD,
873         lmots_shake_n24_w8  = TBD
874     };
875
876     typedef opaque bytestring24[24];
877
878     struct lmots_signature_n24_p200 {
879         bytestring24 C;
880         bytestring24 y[200];
881     };
882
883     struct lmots_signature_n24_p101 {
884         bytestring24 C;
885         bytestring24 y[101];
886     };
887
888     struct lmots_signature_n24_p51 {
889         bytestring24 C;
890         bytestring24 y[51];
891     };
892
893     struct lmots_signature_n24_p26 {
894         bytestring24 C;
895         bytestring24 y[26];
896     };
897
898     union lmots_signature switch (lmots_algorithm_type type) {
899         case lmots_sha256_n24_w1:
900             lmots_signature_n24_p200 sig_n24_p200;

```

```

901     case lmots_sha256_n24_w2:
902         lmots_signature_n24_p101 sig_n24_p101;
903     case lmots_sha256_n24_w4:
904         lmots_signature_n24_p51  sig_n24_p51;
905     case lmots_sha256_n24_w8:
906         lmots_signature_n24_p26  sig_n24_p26;
907     case lmots_shake_n32_w1:
908         lmots_signature_n32_p265 sig_n32_p265;
909     case lmots_shake_n32_w2:
910         lmots_signature_n32_p133 sig_n32_p133;
911     case lmots_shake_n32_w4:
912         lmots_signature_n32_p67  sig_n32_p67;
913     case lmots_shake_n32_w8:
914         lmots_signature_n32_p34  sig_n32_p34;
915     case lmots_shake_n24_w1:
916         lmots_signature_n24_p200 sig_n24_p200;
917     case lmots_shake_n24_w2:
918         lmots_signature_n24_p101 sig_n24_p101;
919     case lmots_shake_n24_w4:
920         lmots_signature_n24_p51  sig_n24_p51;
921     case lmots_shake_n24_w8:
922         lmots_signature_n24_p26  sig_n24_p26;
923 };
924
925 /* hash-based signatures (hbs) */
926
927 enum lms_algorithm_type {
928     lms_sha256_n24_h5   = TBD,
929     lms_sha256_n24_h10 = TBD,
930     lms_sha256_n24_h15 = TBD,
931     lms_sha256_n24_h20 = TBD,
932     lms_sha256_n24_h25 = TBD,
933     lms_shake_n32_h5   = TBD,
934     lms_shake_n32_h10 = TBD,
935     lms_shake_n32_h15 = TBD,
936     lms_shake_n32_h20 = TBD,
937     lms_shake_n32_h25 = TBD,
938     lms_shake_n24_h5   = TBD,
939     lms_shake_n24_h10 = TBD,
940     lms_shake_n24_h15 = TBD,
941     lms_shake_n24_h20 = TBD,
942     lms_shake_n24_h25 = TBD
943 };
944
945 /* leighton-micali signatures (lms) */
946
947 union lms_path switch (lms_algorithm_type type) {

```

```

948     case lms_sha256_n24_h5:
949     case lms_shake_n24_h5:
950         bytestring24 path_n24_h5[5];
951     case lms_sha256_n24_h10:
952     case lms_shake_n24_h10:
953         bytestring24 path_n24_h10[10];
954     case lms_sha256_n24_h15:
955     case lms_shake_n24_h15:
956         bytestring24 path_n24_h15[15];
957     case lms_sha256_n24_h20:
958     case lms_shake_n24_h20:
959         bytestring24 path_n24_h20[20];
960     case lms_sha256_n24_h25:
961     case lms_shake_n24_h25:
962         bytestring24 path_n24_h25[25];
963
964     case lms_shake_n32_h5:
965         bytestring32 path_n32_h5[5];
966     case lms_shake_n32_h10:
967         bytestring32 path_n32_h10[10];
968     case lms_shake_n32_h15:
969         bytestring32 path_n32_h15[15];
970     case lms_shake_n32_h20:
971         bytestring32 path_n32_h20[20];
972     case lms_shake_n32_h25:
973         bytestring32 path_n32_h25[25];
974 };
975
976 struct lms_key_n24 {
977     lmots_algorithm_type ots_alg_type;
978     opaque I[16];
979     opaque K[24];
980 };
981
982 union lms_public_key_switch (lms_algorithm_type type) {
983     case lms_sha256_n24_h5:
984     case lms_sha256_n24_h10:
985     case lms_sha256_n24_h15:
986     case lms_sha256_n24_h20:
987     case lms_sha256_n24_h25:
988     case lms_shake_n24_h5:
989     case lms_shake_n24_h10:
990     case lms_shake_n24_h15:
991     case lms_shake_n24_h20:
992     case lms_shake_n24_h25:
993         lms_key_n24 z_n24;
994

```

```
995     case lms_shake_n32_h5:
996         case lms_shake_n32_h10:
997             case lms_shake_n32_h15:
998                 case lms_shake_n32_h20:
999                     case lms_shake_n32_h25:
1000                         lms_key_n32 z_n32;
1001 };
1002
```

1003 Appendix B—XMSS XDR Syntax Additions

1004 In order to support the XMSS parameter sets defined in Sections 5.2 through 5.4, the XDR
1005 syntax in Appendices A, B, and C of [1] is extended as follows.

1006 B.1 WOTS⁺

```

1007     /* ots_algorithm_type identifies a particular
1008        signature algorithm */
1009
1010     enum ots_algorithm_type {
1011         wotsp-sha2_192      = TBD,
1012         wotsp-shake256_256 = TBD,
1013         wotsp-shake256_192 = TBD,
1014     };
1015
1016     /* Byte strings */
1017
1018     typedef opaque bytestring24[24];
1019
1020     union ots_signature switch (ots_algorithm_type type) {
1021
1022         case wotsp-sha2_192:
1023         case wotsp-shake256_192:
1024             bytestring24 ots_sig_n24_len51[51];
1025
1026         case wotsp-shake256_256:
1027             bytestring32 ots_sig_n32_len67[67];
1028     };
1029
1030     union ots_pubkey switch (ots_algorithm_type type) {
1031         case wotsp-sha2_192:
1032         case wotsp-shake256_192:
1033             bytestring24 ots_pubk_n24_len51[51];
1034
1035         case wotsp-shake256_256:
1036             bytestring32 ots_pubk_n32_len67[67];
1037     };

```

1038 B.2 XMSS

```

1039     /* Definition of parameter sets */
1040
1041     enum xmss_algorithm_type {
1042         xmss-sha2_10_192    = TBD,
1043         xmss-sha2_16_192    = TBD,
1044         xmss-sha2_20_192    = TBD,
1045

```

```

1046     xmss-shake256_10_256 = TBD,
1047     xmss-shake256_16_256 = TBD,
1048     xmss-shake256_20_256 = TBD,
1049
1050     xmss-shake256_10_192 = TBD,
1051     xmss-shake256_16_192 = TBD,
1052     xmss-shake256_20_192 = TBD,
1053 };
1054
1055 /* Authentication path types */
1056
1057 union xmss_path switch (xmss_algorithm_type type) {
1058     case xmss-sha2_10_192:
1059     case xmss-shake256_10_192:
1060         bytestring24 path_n24_t10[10];
1061
1062     case xmss-shake256_10_256:
1063         bytestring32 path_n32_t10[10];
1064
1065     case xmss-sha2_16_192:
1066     case xmss-shake256_16_192:
1067         bytestring24 path_n24_t16[16];
1068
1069     case xmss-shake256_16_256:
1070         bytestring32 path_n32_t16[16];
1071
1072     case xmss-sha2_20_192:
1073     case xmss-shake256_20_192:
1074         bytestring24 path_n24_t20[20];
1075
1076     case xmss-shake256_20_256:
1077         bytestring32 path_n32_t20[20];
1078 };
1079
1080 /* Types for XMSS random strings */
1081
1082 union random_string_xmss switch (xmss_algorithm_type type) {
1083     case xmss-sha2_10_192:
1084     case xmss-sha2_16_192:
1085     case xmss-sha2_20_192:
1086     case xmss-shake256_10_192:
1087     case xmss-shake256_16_192:
1088     case xmss-shake256_20_192:
1089         bytestring24 rand_n24;
1090
1091     case xmss-shake256_10_256:
1092     case xmss-shake256_16_256:

```

```

1093     case xmss-shake256_20_256:
1094         bytestring32 rand_n32;
1095     };
1096
1097     /* Corresponding WOTS+ type for given XMSS type */
1098
1099     union xmss_ots_signature switch (xmss_algorithm_type type) {
1100         case xmss-sha2_10_192:
1101         case xmss-sha2_16_192:
1102         case xmss-sha2_20_192:
1103             wotsp-sha2_192;
1104
1105         case xmss-shake256_10_256:
1106         case xmss-shake256_16_256:
1107         case xmss-shake256_20_256:
1108             wotsp-shake256_256;
1109
1110         case xmss-shake256_10_192:
1111         case xmss-shake256_16_192:
1112         case xmss-shake256_20_192:
1113             wotsp-shake256_192;
1114     };
1115
1116     /* Types for bitmask seed */
1117
1118     union seed switch (xmss_algorithm_type type) {
1119         case xmss-sha2_10_192:
1120         case xmss-sha2_16_192:
1121         case xmss-sha2_20_192:
1122         case xmss-shake256_10_192:
1123         case xmss-shake256_16_192:
1124         case xmss-shake256_20_192:
1125             bytestring24 seed_n24;
1126
1127         case xmss-shake256_10_256:
1128         case xmss-shake256_16_256:
1129         case xmss-shake256_20_256:
1130             bytestring32 seed_n32;
1131     };
1132
1133     /* Types for XMSS root node */
1134
1135     union xmss_root switch (xmss_algorithm_type type) {
1136         case xmss-sha2_10_192:
1137         case xmss-sha2_16_192:
1138         case xmss-sha2_20_192:
1139         case xmss-shake256_10_192:

```

```

1140     case xmss-shake256_16_192:
1141     case xmss-shake256_20_192:
1142         bytestring24 root_n24;
1143
1144     case xmss-shake256_10_256:
1145     case xmss-shake256_16_256:
1146     case xmss-shake256_20_256:
1147         bytestring32 root_n32;
1148 };

```

1149 **B.3 XMSS^{MT}**

```

1150     /* Definition of parameter sets */
1151
1152     enum xmssmt_algorithm_type {
1153
1154         xmssmt-sha2_20/2_192      = TBD,
1155         xmssmt-sha2_20/4_192      = TBD,
1156         xmssmt-sha2_40/2_192      = TBD,
1157         xmssmt-sha2_40/4_192      = TBD,
1158         xmssmt-sha2_40/8_192      = TBD,
1159         xmssmt-sha2_60/3_192      = TBD,
1160         xmssmt-sha2_60/6_192      = TBD,
1161         xmssmt-sha2_60/12_192     = TBD,
1162
1163         xmssmt-shake256_20/2_256  = TBD,
1164         xmssmt-shake256_20/4_256  = TBD,
1165         xmssmt-shake256_40/2_256  = TBD,
1166         xmssmt-shake256_40/4_256  = TBD,
1167         xmssmt-shake256_40/8_256  = TBD,
1168         xmssmt-shake256_60/3_256  = TBD,
1169         xmssmt-shake256_60/6_256  = TBD,
1170         xmssmt-shake256_60/12_256 = TBD,
1171
1172         xmssmt-shake256_20/2_192  = TBD,
1173         xmssmt-shake256_20/4_192  = TBD,
1174         xmssmt-shake256_40/2_192  = TBD,
1175         xmssmt-shake256_40/4_192  = TBD,
1176         xmssmt-shake256_40/8_192  = TBD,
1177         xmssmt-shake256_60/3_192  = TBD,
1178         xmssmt-shake256_60/6_192  = TBD,
1179         xmssmt-shake256_60/12_192 = TBD,
1180     };
1181
1182     /* Type for XMSSMT key pair index */
1183     /* Depends solely on h */
1184

```

```

1185     union idx_sig_xmssmt switch (xmss_algorithm_type type) {
1186         case xmssmt-sha2_20/2_192:
1187         case xmssmt-sha2_20/4_192:
1188         case xmssmt-shake256_20/2_256:
1189         case xmssmt-shake256_20/4_256:
1190         case xmssmt-shake256_20/2_192:
1191         case xmssmt-shake256_20/4_192:
1192             bytestring3 idx3;
1193
1194         case xmssmt-sha2_40/2_192:
1195         case xmssmt-sha2_40/4_192:
1196         case xmssmt-sha2_40/8_192:
1197         case xmssmt-shake256_40/2_256:
1198         case xmssmt-shake256_40/4_256:
1199         case xmssmt-shake256_40/8_256:
1200         case xmssmt-shake256_40/2_192:
1201         case xmssmt-shake256_40/4_192:
1202         case xmssmt-shake256_40/8_192:
1203             bytestring5 idx5;
1204
1205         case xmssmt-sha2_60/3_192:
1206         case xmssmt-sha2_60/6_192:
1207         case xmssmt-sha2_60/12_192:
1208         case xmssmt-shake256_60/3_256:
1209         case xmssmt-shake256_60/6_256:
1210         case xmssmt-shake256_60/12_256:
1211         case xmssmt-shake256_60/3_192:
1212         case xmssmt-shake256_60/6_192:
1213         case xmssmt-shake256_60/12_192:
1214             bytestring8 idx8;
1215     };
1216
1217     union random_string_xmssmt switch (xmssmt_algorithm_type type) {
1218         case xmssmt-sha2_20/2_192:
1219         case xmssmt-sha2_20/4_192:
1220         case xmssmt-sha2_40/2_192:
1221         case xmssmt-sha2_40/4_192:
1222         case xmssmt-sha2_40/8_192:
1223         case xmssmt-sha2_60/3_192:
1224         case xmssmt-sha2_60/6_192:
1225         case xmssmt-sha2_60/12_192:
1226         case xmssmt-shake256_20/2_192:
1227         case xmssmt-shake256_20/4_192:
1228         case xmssmt-shake256_40/2_192:
1229         case xmssmt-shake256_40/4_192:
1230         case xmssmt-shake256_40/8_192:
1231         case xmssmt-shake256_60/3_192:

```

```

1232     case xmssmt-shake256_60/6_192:
1233     case xmssmt-shake256_60/12_192:
1234         bytestring24 rand_n24;
1235
1236     case xmssmt-shake256_20/2_256:
1237     case xmssmt-shake256_20/4_256:
1238     case xmssmt-shake256_40/2_256:
1239     case xmssmt-shake256_40/4_256:
1240     case xmssmt-shake256_40/8_256:
1241     case xmssmt-shake256_60/3_256:
1242     case xmssmt-shake256_60/6_256:
1243     case xmssmt-shake256_60/12_256:
1244         bytestring32 rand_n32;
1245 };
1246
1247 /* Type for reduced XMSS signatures */
1248
1249 union xmss_reduced (xmss_algorithm_type type) {
1250     case xmssmt-sha2_20/2_192:
1251     case xmssmt-sha2_40/4_192:
1252     case xmssmt-sha2_60/6_192:
1253     case xmssmt-shake256_20/2_192:
1254     case xmssmt-shake256_40/4_192:
1255     case xmssmt-shake256_60/6_192:
1256         bytestring24 xmss_reduced_n24_t61[61];
1257
1258     case xmssmt-sha2_20/4_192:
1259     case xmssmt-sha2_40/8_192:
1260     case xmssmt-sha2_60/12_192:
1261     case xmssmt-shake256_20/4_192:
1262     case xmssmt-shake256_40/8_192:
1263     case xmssmt-shake256_60/12_192:
1264         bytestring24 xmss_reduced_n24_t56[56];
1265
1266     case xmssmt-sha2_40/2_192:
1267     case xmssmt-sha2_60/3_192:
1268     case xmssmt-shake256_40/2_192:
1269     case xmssmt-shake256_60/3_192:
1270         bytestring24 xmss_reduced_n24_t71[71];
1271
1272     case xmssmt-shake256_20/2_256:
1273     case xmssmt-shake256_40/4_256:
1274     case xmssmt-shake256_60/6_256:
1275         bytestring32 xmss_reduced_n32_t77[77];
1276
1277     case xmssmt-shake256_20/4_256:
1278     case xmssmt-shake256_40/8_256:

```

```

1279     case xmssmt-shake256_60/12_256:
1280         bytestring32 xmss_reduced_n32_t72[72];
1281
1282     case xmssmt-shake256_40/2_256:
1283     case xmssmt-shake256_60/3_256:
1284         bytestring32 xmss_reduced_n32_t87[87];
1285 };
1286
1287 /* xmss_reduced_array depends on d */
1288
1289 union xmss_reduced_array (xmss_algorithm_type type) {
1290     case xmssmt-sha2_20/2_192:
1291     case xmssmt-sha2_40/2_192:
1292     case xmssmt-shake256_20/2_256:
1293     case xmssmt-shake256_40/2_256:
1294     case xmssmt-shake256_20/2_192:
1295     case xmssmt-shake256_40/2_192:
1296         xmss_reduced xmss_red_arr_d2[2];
1297
1298     case xmssmt-sha2_60/3_192:
1299     case xmssmt-shake256_60/3_256:
1300     case xmssmt-shake256_60/3_192:
1301         xmss_reduced xmss_red_arr_d3[3];
1302
1303     case xmssmt-sha2_20/4_192:
1304     case xmssmt-sha2_40/4_192:
1305     case xmssmt-shake256_20/4_256:
1306     case xmssmt-shake256_40/4_256:
1307     case xmssmt-shake256_20/4_192:
1308     case xmssmt-shake256_40/4_192:
1309         xmss_reduced xmss_red_arr_d4[4];
1310
1311     case xmssmt-sha2_60/6_192:
1312     case xmssmt-shake256_60/6_256:
1313     case xmssmt-shake256_60/6_192:
1314         xmss_reduced xmss_red_arr_d6[6];
1315
1316     case xmssmt-sha2_40/8_192:
1317     case xmssmt-shake256_40/8_256:
1318     case xmssmt-shake256_40/8_192:
1319         xmss_reduced xmss_red_arr_d8[8];
1320
1321     case xmssmt-sha2_60/12_192:
1322     case xmssmt-shake256_60/12_256:
1323     case xmssmt-shake256_60/12_192:
1324         xmss_reduced xmss_red_arr_d12[12];
1325 };

```

```

1326
1327     /* Types for bitmask seed */
1328
1329     union seed switch (xmssmt_algorithm_type type) {
1330         case xmssmt-sha2_20/2_192:
1331         case xmssmt-sha2_20/4_192:
1332         case xmssmt-sha2_40/2_192:
1333         case xmssmt-sha2_40/4_192:
1334         case xmssmt-sha2_40/8_192:
1335         case xmssmt-sha2_60/3_192:
1336         case xmssmt-sha2_60/6_192:
1337         case xmssmt-sha2_60/12_192:
1338         case xmssmt-shake256_20/2_192:
1339         case xmssmt-shake256_20/4_192:
1340         case xmssmt-shake256_40/2_192:
1341         case xmssmt-shake256_40/4_192:
1342         case xmssmt-shake256_40/8_192:
1343         case xmssmt-shake256_60/3_192:
1344         case xmssmt-shake256_60/6_192:
1345         case xmssmt-shake256_60/12_192:
1346             bytestring24 seed_n24;
1347
1348         case xmssmt-shake256_20/2_256:
1349         case xmssmt-shake256_20/4_256:
1350         case xmssmt-shake256_40/2_256:
1351         case xmssmt-shake256_40/4_256:
1352         case xmssmt-shake256_40/8_256:
1353         case xmssmt-shake256_60/3_256:
1354         case xmssmt-shake256_60/6_256:
1355         case xmssmt-shake256_60/12_256:
1356             bytestring32 seed_n32;
1357
1358     };
1359
1360     /* Types for XMSS^MT root node */
1361
1362     union xmssmt_root switch (xmssmt_algorithm_type type) {
1363         case xmssmt-sha2_20/2_192:
1364         case xmssmt-sha2_20/4_192:
1365         case xmssmt-sha2_40/2_192:
1366         case xmssmt-sha2_40/4_192:
1367         case xmssmt-sha2_40/8_192:
1368         case xmssmt-sha2_60/3_192:
1369         case xmssmt-sha2_60/6_192:
1370         case xmssmt-sha2_60/12_192:
1371         case xmssmt-shake256_20/2_192:
1372         case xmssmt-shake256_20/4_192:

```

```
1373     case xssmt-shake256_40/2_192:
1374     case xssmt-shake256_40/4_192:
1375     case xssmt-shake256_40/8_192:
1376     case xssmt-shake256_60/3_192:
1377     case xssmt-shake256_60/6_192:
1378     case xssmt-shake256_60/12_192:
1379         bytestring24 root_n24;
1380
1381     case xssmt-shake256_20/2_256:
1382     case xssmt-shake256_20/4_256:
1383     case xssmt-shake256_40/2_256:
1384     case xssmt-shake256_40/4_256:
1385     case xssmt-shake256_40/8_256:
1386     case xssmt-shake256_60/3_256:
1387     case xssmt-shake256_60/6_256:
1388     case xssmt-shake256_60/12_256:
1389         bytestring32 root_n32;
1390 };
1391
```

Appendix C—Provable Security Analysis

1393 This appendix briefly summarizes the formal security model and proofs of security of the LMS
1394 and XMSS signature schemes and provides a short discussion comparing these models and
1395 proofs.

C.1 The Random Oracle Model

1397 In the *random oracle model* (ROM), there is a publicly accessible random oracle that both the
1398 user and the adversary can send queries to and receive responses from at any time. A random
1399 oracle H is a hypothetical, *interactive* black-box algorithm that obeys the following rules:

- 1400 1. Every time the algorithm H receives a new input string s , it generates an output t
1401 uniformly at random from its output space and returns the response t . The algorithm H
1402 then records the pair (s, t) for future use.
- 1403 2. If the algorithm H is ever queried in the future with some prior input s , it will always
1404 return the same output t according to its recorded memory.

1405 Alternatively, the random oracle H can be described as a non-interactive but *exponentially large*
1406 look-up table initialized with truly random outputs t for each possible input string s .

1407 To say that a cryptographic security proof is done in the random oracle model means that every
1408 use of a particular function (for example, in the case here, the compression function that is used
1409 to perform hashes) is replaced by a query to the random oracle H . This simplifies security claims
1410 as, for example, it becomes easy to prove upper bounds on the likelihood of producing a second
1411 preimage within a fixed number of queries to H . On the other hand, (compression) functions in
1412 the real world are neither interactive nor have exponentially large descriptions, so they cannot
1413 truly behave like a random oracle.

1414 It is therefore desirable to have a cryptographic security proof that avoids using the random
1415 oracle model. However, this often leads to less efficient cryptographic systems, or it is not yet
1416 known how to perform a proof without appealing to the random oracle model, or both. So, as a
1417 matter of real-world pragmatism, the ROM is commonly used.

C.2 The Quantum Random Oracle Model

1419 The *quantum random oracle model* (QROM) is similar to the ROM, except it is additionally
1420 assumed that all parties (in particular, the adversary) have quantum computers and can query the
1421 random oracle H in superposition. (In the real world, the random oracle H is still instantiated as a
1422 compression function or similar, as per the cryptosystem's specification.) While this complicates
1423 security claims as compared to the ROM, it more accurately models the power of an adversary
1424 that has access to a large-scale quantum device for its cryptanalysis when attacking a real-world
1425 scheme.

C.3 LMS Security Proof

1427 In [11], the author considers a particular experiment in the random oracle model in which the

1428 adversary is given a series of strings with prefixes (in a randomly chosen but structured manner)
1429 and hash targets. The attacker's goal is to find one more string that has the same prefix and hash
1430 target as any of its input strings. The author proves an upper bound on the adversary's ability to
1431 compute first or second preimages from these strings (by querying the compression function
1432 modeled as a random oracle).

1433 Then, the author reduces the problem of forging a signature in LMS to this stated experiment,
1434 concluding that the same upper bounds apply to the problem of producing forgeries against
1435 LMS. This random oracle model proof critically depends on the randomness of the prefixes used
1436 in LMS, which means that LMS in the real world critically depends on the pseudorandomness of
1437 the prefixes.

1438 Further, in [15], the same proof is carried out in the QROM.

1439 **C.4 XMSS Security Proof**

1440 In [12], a security analysis for the *original* (academic publication) version of XMSS is given
1441 under the following assumptions:

- 1442 1. The function family $\{f_k\}$ used to construct Winternitz signatures is pseudorandom. This
1443 means that if the bit string k is chosen uniformly at random, then an adversary given
1444 black-box access to the function f_k cannot distinguish this black box from a random
1445 function within a polynomial number of queries (except with negligible probability).
- 1446 2. The hash function family $\{h_k\}$ is second preimage-resistant. This means that if bit strings
1447 k and m are chosen uniformly at random, then an adversary given k and m cannot
1448 construct $m' \neq m$ such that $h_k(m') = h_k(m)$ in polynomial time (except with negligible
1449 probability).

1450 The proof in [12] asserts that if both of these assumptions are true, then XMSS is existentially
1451 unforgeable under adaptive chosen message attacks (EUF-CMA) in the standard model.

1452 However, in the *current* version of XMSS^{MT} [1], the security analysis differs somewhat. In the
1453 standard model, [17] shows that XMSS^{MT} is EUF-CMA. Further, [16] shows that XMSS^{MT} is
1454 post-quantum existentially unforgeable under adaptive chosen message attacks with respect to
1455 the QROM.

1456 In a little more detail, the current version of XMSS uses two types of assumptions:

- 1457 1. A standard model assumption – that the hash function h_k , used for the one-time
1458 signatures and tree node computations, is post-quantum, multi-function, multi-target
1459 preimage-resistant.
- 1460 2. A (quantum) random oracle model assumption – that the pseudorandom function f_k , used
1461 to generate pseudorandom values for randomized hashing and computing bitmasks as
1462 blinding keys, may be validly modeled as a quantum random oracle H .

1463 C.5 Comparison of the Security Models and Proofs of LMS and XMSS

1464 Generally speaking, both LMS and XMSS are supported by sound security proofs under
1465 commonly used cryptographic hardness assumptions. That is, if these cryptographic assumptions
1466 are true, then both schemes are provably shown to be existentially unforgeable under chosen
1467 message attack, even against an adversary that has access to a large-scale quantum computer for
1468 use in its forgery attack.

1469 The main difference between these schemes' security analyses comes down to the use (and the
1470 degree of use) of the random oracle or quantum random oracle models. Along these lines, the
1471 difference between the (standard model/real world) cryptographic assumption that some function
1472 family $\{f_k\}$ is pseudorandom and the use of the random oracle model is briefly pointed out. For a
1473 function f_k to be a pseudorandom function in the real world, it should be the case that the bit
1474 string k used as the key to the function remains private, meaning that it is not in the view of the
1475 adversary at any point of the security experiment. On the other hand, a random oracle H achieves
1476 the same pseudorandomness (or even randomness) properties of a pseudorandom function f_k , but
1477 there is no key k necessarily associated with the random oracle. Indeed, all inputs to the random
1478 oracle H may be known to all parties and, in particular, to the adversary. Therefore, using the
1479 random oracle model clearly involves making a stronger assumption about the (limits of the)
1480 cryptanalytic power of the adversary.

1481 That said, a security proof is either *entirely* a "real world proof," which does not use the random
1482 oracle model, or it appeals to the random oracle methodology in some manner. The security
1483 analysis of the current version of XMSS only uses the random oracle H when performing
1484 randomized hashing and computing bitmasks, whereas LMS uses the random oracle H to a
1485 greater degree (modeling the compression function as a random oracle). However, it remains the
1486 case that both schemes in their modern form are ultimately proven secure using the ROM and
1487 QROM.

1488 Therefore, the cryptographic hardness assumptions made by LMS and XMSS in order to achieve
1489 existential unforgeability under chosen message attack (EUF-CMA) may be viewed as
1490 substantially similar and worthy of essentially equal confidence. As such, the practitioner's
1491 decision to deploy one scheme or the other should primarily depend on other factors, such as the
1492 efficiency demands for a given deployment environment or the other security considerations
1493 enumerated earlier in this document.