LMS vs XMSS: Comparion of two Hash-Based Signature Standards

Panos Kampanakis, Scott Fluhrer

Cisco Systems, USA
{panosk, sfluhrer}@cisco.com

Abstract. Quantum computing poses challenges to public key signatures as we know them today. LMS and XMSS are two hash based signature schemes that have been proposed in the IETF as quantum secure. Both schemes are based on well-studied hash trees, but their similarities and differences have not yet been discussed. In this work, we attempt to compare the two standards. We compare their security assumptions and quantify their signature and public key sizes. We also address the computation overhead they introduce. Our goal is to provide a clear understanding of the schemes' similarities and differences for implementers and protocol designers to be able to make a decision as to which standard to chose.

1 Introduction

1.1 HBS

Recent interest in quantum computer research has increased concerns about the challenges quantum computers would pose, were they developed in practice, to public key crypto algorithms. More specifically Shor's algorithm [1, 2] would break signature algorithms (RSA, DSA, ECDSA) in polynomial time and thus render today's digital signatures useless. To address this concern, Hash Based Signatures (HBS) are schemes [3–9] that have been studied for a long time and proven to be secure against quantum algorithms.

One method of implementing an HBS would be to use Merkle trees [10] along with a One Time Signature (OTS) scheme. An OTS is a signature scheme with a private key used to sign a message and the corresponding public key is used to verify the OTS signature. The caveat is that a private key can be used only once to sign a message. An HBS tree is a binary Merkle tree whose leafs are the OTS public keys, and each internal node in the tree consists of the hash of its two children. The root of the tree (Fig. 1) is the public key of the Merkle construction.

The HBS signature consists of the OTS signature, the corresponding public key leaf and the nodes that form the path from the leaf to the root of the tree. For a verifier to verify an HBS signature, he first computes what the OTS public key would be if the signature was valid; then, he takes that putative value, and tries to authenticate it via the Merkle tree by using the tree path included in the signature. If the final hash is equal to the Merkle public key, that is, the expected value of the root node, then he knows that the computed OTS public key really was the right one, and that the Merkle part of the signature validates. Fig.1 gives an example of a Merkle tree. In this example we sign the actual message with the OTS private key corresponding to the public key leaf L3. Along with the OTS signature, we also include the values of the public key L2, and the values of the internal nodes N4 and N3. The verifier would reconstruct (based on the OTS signature of the message) the value of L3, combine that with the value of L2 to reconstruct N5, combine that with N4 to form N2, and finally combine that with N3, to reconstruct the expected root value. If that's the same as the public key, then the signature was generated by the legitimate signer.

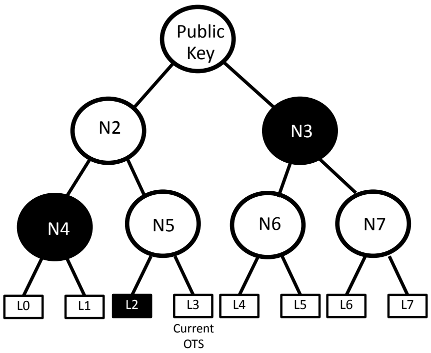

Fig. 1: HBS tree with the OTS public key leaves and the trusted root, with the authentication path for L3 highlighted

The maximum number of messages an HBS tree can sign is $2^{H_{tree}}$, where H_{tree} is the height of the tree. For a tree of height $H_{tree} = 40$ or more, key generation would be completely impractical, as the entire tree would need to be generated in order to form the root. To keep the key generation time practical, while allowing for a large number of signatures, the tree can be broken into a multi-level tree. Fig. 2 shows a multi-level HBS tree where the bottom level trees sign the messages and the tree at the level above signs the root of the tree below. In other words, the root of of each tree is signed by the OTS of the tree at the level above. The maximum number of signed messages of the multi-level tree is $2^{H_{tree1}+H_{tree2}+...+H_{treeZ}}$, where $H_{tree1}, H_{tree2}, ..., H_{treeZ}$ are the heights of the subtrees at level $1, 2, ..., Z$. With a multi-level tree the height of each tree remains reasonable. Key generation can be done in time $O(2^{tree1} + 2^{tree2} + ... + 2^{treeZ})$, which is considerably smaller than the number of signatures that can be generated. Signing and verifying can also be done with a relatively small cost increase. On the other hand, multi-level trees do increase the length of the signature.signature consists of the OTS signature, and now we include an OTS signature for each tree level.

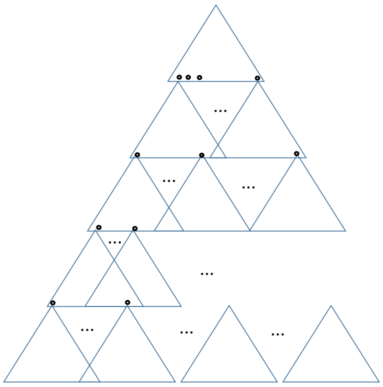

Fig. 2: Multi-level HBS tre where the root of a tree is signed by the OTS of the tree above.

The security of a Merkle-tree hash based signature (including multi-level trees) is based on the strength of the hash function it uses. To forge a signature, someone must either generate a hash preimage, or a second preimage (assuming you use randomized hashing for the initial message, which both LMS and XMSS do).

One Proposed Standard: LMS [11] is a quantum secure HBS scheme proposed in the IETF. As a component it uses a scheme proposed by Leighton and Michali in which an LMS public key is used to sign a second LMS public key, which is then distributed along with the signatures generated with the second public key [4]. LMS uses the Winternitz one-time signature (WOTS) as its OTS (LM-OTS). HSS, the multi-level version of LMS, supports up to eight independent levels of trees (each configurable to be a different height and Winternitz parameter). LMS currently supports only SHA-256 as its hash function, with other hash functions likely to be added later.

LMS uses a distinct prefix and suffix for every hash it performs. This is to avoid multi-collision attacks. These are situations where the attacker has a number of hash targets (such a values from the Merkle tree of valid signatures it has seen), and 'wins' if he is able to find a preimage for any one of them. The idea is that the attacker selects an image, hashes it, and checks to see if the hash value happens to be any of the values he's looking for. However, such a preimage would actually allow the attacker to generate a forgery only if it is a plausible image in the context of the signature scheme. By ensuring that an image would be a plausible image only in one place in one signature tree (by assigning different prefixes and suffixes for each use), this means that any image he generates could be a plausible preimage for only one place in the signature scheme; hence any such attack is effectively limited to only one hash target.

A Second Proposed Standard: XMSS [12] is a second draft proposed in the IETF. XMSS leverages an HBS tree and an OTS to sign messages. The OTS it uses is a variant of WOTS (WOTS+ [13]) that eliminates the requirement for a collision resistant hash function. Additionally, XMSS uses an L-tree structure [8] to shrink the OTS public key. Moreover, it uses blinding masks for the WOTS+ Winternitz chains and every node calculation in the HBS and L-trees. The blinding keys are generated pseudo-randomly for every tree node while generating or verifying a signature. The multi-level version of XMSS is defined as XMSS^{MT}. Both schemes can sign arbitrary messages by using a target collision resistant hash function [14]. XMSS supports SHA-256, SHA-512 and SHAKE as hash functions. The hash functions could be extended later.

1.2 Notation

LMS and XMSS use different notation to specify the same parameters. For this work, we will use a common notation for various parameters which is summarized in Table 1. Table 1 also gives the corresponding notations from the LMS and XMSS drafts.

Symbols	Meaning
w	The Winternitz parameter, which is the base that the initial hash is interpreted. In XMSS, this is the value w. In LMS, this is the value 2^w.
n	The length of the hash (in bytes). In XMSS, this is the value n. In LMS, both n and m stand for this.
p	The number of Winternitz chains used in a single OTS operation. In XMSS, this is the value *len*. In LMS, this is the value p.
h	The height of a single Merkle tree. Both LMS and XMSS notate this as h when dealing with a single level tree.
$\Sigma\ p$	This is the total number of Winternitz chains used in all levels of a multilevel tree. In XMSS^{MT}, this is the value $d \cdot len$. In HSS, this is the sum of all the p values across all the levels.
$\Sigma\ h$	This is the total height of all the Merkle trees. In XMSS^{MT}, this is the value h. In HSS, this is the sum of all the h values across all the levels.
d	This is the number of Merkle trees in the multi-level tree. In XMSS^{MT}, this is the value d. In HSS, this is L.

Table 1: Meanings for parameters used in this paper.

In the rest of this paper, Section 2 presents the differences of XMSS and LMS in terms of security proofs and assumptions, public key, signature size and processing. Section 3 offers a discussion on the applicability of each scheme and how the comparisons presented should weigh in choosing LMS or XMSS to provide quantum secure signatures.

2 Comparison

Full disclosure This paper has been co-written by the designers of the LMS system. We have tried to be impartial; however we would note that we have opinions on what is important in a hash based signature algorithm, and (not surprisingly) LMS was designed to function well according to the criteria that were considered important.

As published, the two schemes have different restrictions on the parameter sets that they allow. With LMS, you are limited to the SHA-256 hash function. With XMSS, you are limited to Winternitz parameter $w = 16$ (the draft claims support for $w = 4$ as well, however no defined parameter set allows it), and in the multi-level version of XMSS, each Merkle tree is required to have the same height, hash function and Winternitz parameter. In both cases, it would be simple to extend the allowable parameters to cover the parameter sets allowed by the other. In the comparison of the two schemes below, we will restrict ourselves to parameter sets that are supported by both.

2.1 Security model

Even though the two schemes share many similarities, it is important to identify differences related to their assumptions and security proofs which will help us distinguish the usecases that make them more favorable.

LMS has been proven [15] secure in the Random Oracle Model (ROM) by modeling the hash function as a Random Oracle (RO). As hash functions cannot be considered truly random, [16] proves LMS secure in the ROM when the Merkle-Damgård hash compression function is modeled as the RO. [17] also proves it secure in the Quantum Random Oracle Model (QROM). The original XMSS scheme is proven secure in the standard model [7]. The IETF version of multi-level XMSS^{MT} [12] has been proven forward secure and existentially unforgeable under chosen message attacks (EU-CMA) in the standard model [18] and post-quantum existentially unforgeable under adaptive chosen message attacks with respect to the QROM [9].

The assumption that LMS makes is that the hash compression functions of the hashes used in LM-OTS and the tree node calculations are a RO. The assumptions that XMSS makes is that the hash function used for the OTS and the tree node calculations are post-quantum multi-function multi-target second-preimage resistant. The current IETF draft version of XMSS [12] uses a PRF to generate the pseudorandom values for randomized hashing and the bitmasks used as blinding keys. This PRF is assumed to be a pseudorandom function modeled as a RO when used to generate bitmasks. Moreover, XMSS' message compression hash is assumed to be a multi-target extended target collision resistant keyed hash [9]. The message compression function in LMS is again a hash with a compression function modeled as a RO for Merkle-Damgard hashes (i.e. SHA2). Clearly LMS has stronger security assumptions than XMSS, but readers should not ignore that the XMSS IETF draft introduces a ROM for the PRF function.

2.2 Sizes

One important disadvantage of post-quantum signatures is their size. Stateful hash based signatures schemes have large signatures that make them impractical in some scenarios. Stateless signature methods have even larger signature sizes. It is important to evaluate the practical signature sizes of LMS and XMSS to see how they compare at the same security level. The public key size is also important for the verifier.

Public Key The public key sizes for LMS and XMSS and their corresponding multi-level version are shown in Table 2. Currently, the size of the public keys depends on only the hash function. If we consider the SHA-256 version of LMS and XMSS ($n = 32$), we see that XMSS^{MT}public keys are slightly bigger but of very similar length as the HSS public keys.

Signature The signature key sizes for LMS and XMSS and their corresponding multi-level version are shown in Table 2. Even though it is not obvious at first, the signature sizes of LMS are slightly bigger that XMSS, but not significantly.

	Public Key	Signature
LMS	$24 + n$	$12 + n(p + h + 1)$
XMSS	$4 + 2n$	$4 + n(p + h + 1)$
HSS	$28 + n$	$(36d + 2nd - n - 20) + n(\Sigma p + \Sigma h)$
XMSS^{MT}	$4 + 2n$	$\lceil \Sigma h/8 \rceil + n(\Sigma p + \Sigma h + 1)$

Table 2: Sizes (in bytes) of HBS schemes based on scheme parameters.

To quantify the public key and signature sizes differences between LMS and XMSS, we use Table 3. The table summarizes sizes for the two schemes that can sign the same amount of total messages, at the same security level with the same OTS signature length. Table 3a includes the sizes for LMS with LMS_SHA256_M32_H10 and LMOTS_SHA256_N32_W4. The equivalent XMSS parameter sets are XMSS_SHA2-256_W16_H10 and XMSSMT_SHA2-256_W16_H20_D2. Both schemes can sign 2^{10} messages and their multi-level version can sign 2^{20} messages. Table 3b shows the sizes for LMS parameters LMS_SHA256_M32_H20 and LMOTS_SHA256_N32_W4 and XMSS parameters XMSS_SHA2-256_W16_H20 and XMSSMT_SHA2-256_W16_H40_D2. Similarly, Tables 3c, 3d provide the public key and signature sizes for more (2^{60}) total multi-level signed messages and a tree height of $h = 20$ and 10 respectively (XMSS parameter set XMSSMT_SHA2-256_W16_H60_3 and XMSSMT_SHA2-256_W16_H60_6). As we can see, for comparable security parameters, LMS has only 8 more bytes of signature size. In terms of their multi-level tree versions HSS has 2%-5% larger signatures (depending on the number of tree levels). Regarding public key sizes, LMS has slightly smaller than XMSS public keys, but both remain relatively small.

If we examine the reasons for the difference in sizes, it turns out the largest difference is due to the HSS approach of explicitly embedding the internal Merkle

	Public Key	Signature
LMS	56	2508
XMSS	68	2500
HSS	60	5076
XMSS^{MT}	68	4963

(a) $w = 16$, $p = 67$, 2^{10} LMS / XMSS and 2^{20} HSS / XMSS^{MT}total messages (2 levels)

	Public Key	Signature
LMS	56	2828
XMSS	68	2820
HSS	60	5716
XMSS^{MT}	68	5605

(b) $w = 16$, $p = 67$, 2^{20} LMS / XMSS and 2^{40} HSS / XMSS^{MT}total messages (2 levels)

	Public Key	Signature
HSS	60	8600
XMSS^{MT}	68	8392

(c) $w = 16$, $p = 67$, 2^{60} HSS / XMSS^{MT}total messages (3 levels)

	Public Key	Signature
HSS	60	15533
XMSS^{MT}	68	14824

(d) $w = 16$, $p = 67$, 2^{60} HSS / XMSS^{MT}total messages (6 levels)

Table 3: Sizes (in B) of HBS scheme for various parameters and $n = m = 32$.

public keys within the signature. The HSS choice does make it simpler and potentially more extensible, but at the cost of having $h - 1$ public keys within each signature. On the other hand, LMS currently supports $w = 256$ ($w = 8$ in the notation that LMS uses); this setting would reduce the size of the signature at the cost of computational overhead (that is still less than the XMSS equivalent parameter set).

2.3 Performance - Experimental Results

HBS signatures require multiple hash function calculations by both the signer and verifier. Similarly, generating the OTS public key involves calculating the Winternitz chain multiple times. It turns out that the majority of the computation by both the signer and the verifiier is spent performing the hash computations during the Winternitz one-time signature. To verify this, we instrumented implementations of both XMSS and LMS. The LMS implementation is written in C by the LMS authors [19]. It supports all the parameter sets listed in the LMS draft. It also supports multi-threading to accelerate operations. For the test, multi-threading was disabled. The XMSS implementation is a C implementation written by Andreas Hülsing and Joost Rijneveld. We modified the implementation to support the hash pre-computation optimization of the PRF function which gave a noticeable speedup. Additionally, both implementations use OpenSSL to perform the hash computations, hence the quality of that part of the implementation was equivalent. Both implementations allow time/memory trade-offs that affected the signature operations; these trade-offs modified the number of OTS public key generations required during a signature operation.

We tried to select trade-offs that resulted in similar number of OTS public key operations.

We first measured how much time was spent performing the hash compression operation for the OTS computations for each scheme. For this measurement, we include the hash computations required for compressing the OTS public key into an n-byte value. The results of this measurement are given in Table 4. In all cases, at least 85% of the time was spent performing the hash compression computation. Because the schemes can be implemented to perform the same number and types of OTS operations and the hash compressions in each OTS operation are identical, we can perform a realistic comparison by comparing the number of hash compression operations for each scheme[1].

Operation	Hash Compression for OTS	Other Operations
LMS Key Gen	94.4%	5.6%
XMSS Key Gen	88.1%	11.9%
LMS Sig Gen	93.6%	6.4%
XMS Sig Gen	88.0%	12.0%
LMS Sig Ver	90.1%	9.9%
XMSS Sig Ver	85.8%	14.2%

Table 4: Percentage of time spent doing hash compression operations during an OTS computation.

To evaluate the number of hash compressions, we can break the OTS operation itself into three types of sub-operations; namely:

- Generate a secret value at the beginning of the Winternitz chain. Our model will be that it can be done with 1 hash compression operation.
- Iterate to the next value of the Winternitz chain. LM-OTS can perform this with 1 hash compression operation. WOTS+ requires 4.
- Combine the final values of the Winternitz chain into a single n-byte sized value. LM-OTS takes all the final values, and hashes them together as a single message. This requires $\lceil \frac{p}{2} \rceil$ hash compression operations. WOTS+ uses an L-tree construction. An L-tree has $p-1$ internal nodes, and each node requires 6 hash compression operations giving a total of $6(p-1)$ hash compression operations.

An OTS public key generation, generates p secrets, performs $p(w-1)$ Winternitz iterations, and then combines those final values. Thus, from the OTS sub-operation breakdown we understand that the OTS public generation of

[1] These numbers indicate that the XMSS implementation tested has significantly more overhead than the LMS implementation; however it appears to be a difference between the implementations, rather than the schemes themselves.

XMSS includes 4 times more hash compressions than LMS. As LMS/XMSS key generation requires OTS public key generation almost exclusively, we would expect an XMSS key generation to perform approximately 4 times slower than an LMS key generation.

An OTS signature operation generates p secret values, and then performs an expected[2] $p(w-1)/2$ Winternitz iterations. As LMS/XMSS signature generation typically performs 1 OTS signature operation, and (for multilevel schemes) several OTS public key generation operations, by using the OTS sub-operation breakdown we would expect the XMSS signature operation to be between 3.5 to 4 times slower than the corresponding LMS operation.

An OTS signature verification also iterates an expected $p(w-1)/2$ times, and then combines those values. As LMS/XMSS signature verification performs only OTS verification operations, by using the OTS sub-operation breakdown we would expect that an XMSS signature verification to perform approximately 5 times slower than an LMS signature verification of equivalent parameters. If we sum up the analysis for LMS and XMSS, using the SHA-256 hash function ($n = 32$) and Winternitz value $w = 16, p = 67$, we obtain the total hash compression operations given in Table 5. As we can see the LMS OTS operations are 3-5 times less than XMSS. Thus, since the majority of key generation, signature and verification is spent on the OTS operations as explained above, we expect that XMSS will be performing 3-5 times slower than LMS.

Operation	LMS	XMSS	XMSS / LMS ratio
OTS PK Gen	603.5	2473	4.10
OTS Sign	318.25	1072	3.37
OTS Verify	285.25	1401	4.91

Table 5: Number of hash compression operations expected for various OTS operations SHA-256, $w = 16$.

To experimentally confirm our performance analysis, we measured the overall performance on the LMS and XMSS implementations mentioned previously. We ran the tests using the equivalent of the XMSS^{MT}parameter sets XMSSMT_SHA2-256_W16_H20_D2 and XMSSMT_SHA2-256_W16_H40_D2. For each test, we generated a public key, signed 4096 short (100 byte messages), and then verified the 4096 signatures generated[3]. As we can see in Table 6

[2] This is not precisely accurate, as the checksum digits cannot be expected to average $(w-1)/2$; however it is close.

[3] Both implementations allow time/memory trade-offs in the signature algorithm (where one could increase the amount of memory used to reduce the number of OTS computations required); because the two different implementations used different algorithms, we could not come up with trade-offs with the same number of expected

the ratios measured are mostly in line with our analysis. The XMSS signature operations are somewhat more expensive than expected, but well within the range we would expect from implementation details.

Operation	LMS	XMSS	XMSS / LMS ratio
	XMSSMT_SHA2-256_W16_H20_D2		
PK Gen	0.89 s	3.26 s	3.66
Sign	1.21 ms	4.72 ms	3.90
Verify	0.339 ms	1.76 ms	5.19
	XMSSMT_SHA2-256_W16_H40_D2		
PK Gen	720 s	3340 s	4.64
Sign	1.91 ms	7.70 ms	4.03
Verify	0.350 ms	1.75 ms	5.00

Table 6: Measured time per operation for LMS and XMSS

3 Discussion

The LMS and XMSS standards are similar. They address the same issues and provide post-quantum secure digital signatures that could find different applications in practice. Thus, various protocols and implementers might find it hard to decide between the two in order to pick the more suitable scheme for their usecase. When phased with that question a security practitioner should take into consideration the analysis presented above.

Both XMSS and LMS make similar assumptions on their hash function; for both LMS and XMSS, a second preimage attack on the hash function would allow a single forgery, and a preimage attack that allowed the attacker to specify all but n bits would allow the attacker to generate his own Merkle tree based on a public key, allowing him to sign as many messages as he wished.

Where they differ is that XMSS strives to provide the hash function with random independent inputs for every hash evaluation; while LMS has inputs with predictable changes. This difference allows a tighter proof model for XMSS' tree hierarchy (because the attacker has to find a preimage of a hash of random inputs). On the other hand, both systems achieve the same security level during the initial message hashing (with both LMS and XMSS providing an unpredictable prefix); as this requires a stronger assumption of the hash function (second preimage resistance), it's debatable whether XMSS' tighter proof model for the tree hierarchy is important.

public key generations per signature, so we selected trade-offs slightly in favor of XMSS.

Other factors to consider when making a decision between XMSS and LMS are the signature and public key sizes, and the computation time. As we have seen above, only XMSS^{MT} has slightly smaller signature sizes than HSS, while LMS performs significantly faster. In addition, while we have studied them in isolation, they need to be considered together. There are parameter sets that reduce the signature size at the cost of computation; LMS (with its cheaper computation) may make such a trade-off more acceptable, and such a reduction in signature size might more than make up the slightly larger LMS signature encoding. To give a concrete example, we use two similar parameter sets. For XMSS we have XMSSMT_SHA2-256_W16_H20_D2 (which we benchmarked above). For LMS, we select a level 2 HSS, with each level being LMS_SHA256_N32_H10 and LMOTS_SHA256_N32_W8, that is, using $w = 256$ in the XMSS context. The results of this comparison are in Table 7. This table shows that LMS (with these settings) is measured to perform moderately slower than XMSS; however the LMS signature size is almost half of the XMSS signature. One could define an equivalent XMSS parameter set with $w = 256$; however that would drastically increase the amount of computation required.

Operation	HSS	XMSS^{MT}
PK Gen	5.44 sec	3.26 sec
Sign Gen	6.49 msec	4.72 msec
Sig Ver	2.66 msec	1.76 msec
Size Public Key	60 byte	68 byte
Size Signature	2964 byte	5605 byte

Table 7: Comparison of LMS ($w = 256$) and XMSS ($w = 16$).

In summary, XMSS with equivalent multilevel parameter sets has slightly smaller signature sizes than LMS. However, LMS performs significantly better, which allows us more options when selecting parameter sets that fit within the application constraints.

References

1. Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. on Computing*, 26(5):1484–1509, 1997.
2. John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Info. Comput.*, 3(4):317–344, July 2003.
3. C. Dods, Nigel P. Smart, and Martijn Stam. Hash Based Digital Signature Schemes. In Nigel P. Smart, editor, *Cryptography and Coding, 10th IMA International Conference*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer, 2005.

4. T. Leighton and S. Micali. Large provably fast and secure digital signature schemes from secure hash functions. U.S. Patent 5,432,852, 1995.
5. Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle Signatures with Virtually Unlimited Signature Capacity. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.
6. Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS — An Improved Merkle Signature Scheme. In Rana Barua and Tanja Lange, editors, *Progress in Cryptology — INDOCRYPT 2006, 7th International Conference on Cryptology in India*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2006.
7. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS — A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography — 4th International Workshop, PQCrypto 2011*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.
8. Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology — EUROCRYPT 2015 — 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.
9. Andreas Hülsing, Joost Rijneveld, and Fang Song. *Mitigating Multi-target Attacks in Hash-Based Signatures*, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
10. Ralph C. Merkle. A Certified Digital Signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO '89, 9th Annual International Cryptology Conference*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.
11. David McGrew and Michael Curcio. Hash-Based Signatures. `https://datatracker.ietf.org/doc/draft-mcgrew-hash-sigs/`, 2016. Internet-Draft. Accessed 2016-06-06.
12. Andreas Hülsing, Denis Butin, Stefan Gazdag, and Aziz Mohaisen. XMSS: Extended Hash-Based Signatures. `https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures/`, 2016. Internet-Draft. Accessed 2016-06-06.
13. Andreas Hülsing. W-OTS$^+$ — Shorter Signatures for Hash-Based Signature Schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology — AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013.
14. Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. *Digital Signatures Out of Second-Preimage Resistant Hash Functions*, pages 109–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
15. Jonathan Katz. Analysis of a Proposed Hash-Based Signature Standard. `https://www.cs.umd.edu/~jkatz/papers/HashBasedSigs-04.pdf`, 2016. Accessed 2016-11-14.
16. Scott Fluhrer. Further analysis of a proposed hash-based signature standard. Cryptology ePrint Archive, Report 2017/553, 2017. `http://eprint.iacr.org/2017/553`.

17. Edward Eaton. Leighton-micali hash-based signatures in the quantum random-oracle model. Cryptology ePrint Archive, Report 2017/607, 2017. `http://eprint.iacr.org/2017/607`.
18. Tal Malkin, Daniele Micciancio, and Sara Miner. *Efficient Generic Forward-Secure Signatures with an Unbounded Number of Time Periods*, pages 400–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
19. Scott Fluhrer. LMS implementation. `https://github.com/cisco/hash-sigs`, 2017. Accessed 2017-13-07.