LMS-SM3 and HSS-SM3: Instantiating Hash-based Post-Quantum Signature Schemes with SM3

Siwei Sun[1], Tianyu Liu[1], Zhi Guan[2], Yifei He[2], Jiwu Jing[1], Lei Hu[1], Zhenfeng Zhang[3], Hailun Yan[1]

[1] School of Cryptology, University of Chinese Academy of Sciences, China
siweisun.isaac@gmail.com
[2] Peking University, China
heyifei@pku.edu.cn, guan@pku.edu.cn
[3] Trusted Computing and Information Assurance Laboratory, Institute of Software, China

Abstract. We instantiate the hash-based post-quantum *stateful* signature schemes LMS and HSS described in RFC 8554 and NIST SP 800-208 with SM3, and report on the results of the preliminary performance test.

Keywords: Hash Functions, Digital Signatures, Leighton-Micali Signature, Hierarchical Signature System, SM3

1 Introduction

It is well known that the security of many widely deployed digital signature schemes (e.g., RSA, DSA, and ECDSA) will be compromised if large-scale quantum computers are ever built [Sho99]. Hash-based signature scheme is one important type of quantum-resistant cryptographic algorithms. Generally speaking, there are two approaches for constructing signature schemes based on hash functions. The first one signs messages by exposing the pre-images of certain one-way functions [Lam79, Mer89, BDH11], the second one signs message by proving the possession of the pre-images with zero-knowledge techniques [CDG+17]. In this work, we restrict our attention to the former approach. Also, we only consider hash-based stateful signature schemes, since these type of hash-based signature schemes are actively standardized [MCF19, HBG+18, NIS20] and are more appealing with regards to performance and resource consumption. For the construction of stateless hash-based signature schemes, we refer the reader to [BHH+15, BHK+19] for more information.

The study of hash-based signature schemes has a long history. The first such scheme can be traced back to 1976 [DH76], where Diffie and Hellman proposed a one-time signature scheme for signing a single bit. After more than 40 years of development, the construction and implementation of hash-based signatures are well studied and have benefited from renewed attention in the last decade due to the concern of quantum attacks.

Hash-based stateful signatures have the following advantages. Firstly, hash-based signatures are arguably the most conservative designs with respect to

security. They enjoy provable security which relies so solely on the (second) pre-image resistance of the underlying hash functions. secondly, They are *hash-function-agnostic*, meaning that any compatible hash function can be used to instantiate the schemes. Therefore, when one hash function is insecure, we can simply replace it with a secure one. Thirdly, it features small private and public keys, and fast signature generation and verification, making it suitable for compact verifier implementations. Finally, hash-based signatures have a rich set of tunable parameters, and thus it is easy to tailor the designs for specific application scenarios [WJW+19, HRB17, vdLPR+18, ZCY22]. The disadvantage of stateful hash based signatures including large signature sizes, and the issue of state management.

Outline. In section 2 we give some preliminaries and notations used later. Section 3 describes the one-time signature scheme `LM-OTS` which is employed as a building block for the many-time signature schemes `LMS` and `HSS` introduced in Section 4 and Section 5, respectively. In Section 6, we instantiate these hash-based signature schemes with `SM3` and report on the results of the performance test on some preliminary implementations without optimization.

2 Notations and Preliminaries

Let $\mathbb{F}_2 = \{0, 1\}$ be the binary field and $\mathbb{B} = \mathbb{F}_2^8$ be the set of all 8-bit binary strings. Concrete values of byte strings are specified in binary or hexadecimal notations. For example, we use `0x1F12` to denote the 2-byte string $(\texttt{0001 1111 0001 0010})_2$. Sometimes, we need to convert a unsigned integer into a string of bytes, which is done by applying the conversion functions $\texttt{u8str}(\cdot)$, $\texttt{u16str}(\cdot)$, and $\texttt{u32str}(\cdot)$. For example, $\texttt{u8str}(11) = \texttt{0x0B}$, $\texttt{u16str}(6214) = \texttt{0xF2BC}$, and $\texttt{u32str}(305441741) = \texttt{0x1234ABCD}$. Conversely, we use $\texttt{int32}(\cdot)$ to convert an integer $i \in \{0, 1, \cdots, 2^{32} - 1\}$ to a 4-byte string. For example, $\texttt{int32}(\texttt{0x1234ABCD}) = 305441741$. Also, $\texttt{u32str}(\texttt{int32}(S)) = S$ holds for any 4-byte string S.

Let S be a byte string. Then, $\texttt{byte}(S, i)$ denotes the i-th byte of S, and $\texttt{byte}(S, i, j)$ denotes the range of bytes from the i-th to the j-th byte, inclusive. For example, if $S = \texttt{0xABCDFF01}$, then $\texttt{byte}(S, 0) = \texttt{0xAB}$, $\texttt{byte}(S, 3) = \texttt{0x01}$, and $\texttt{byte}(S, 1, 3) = \texttt{0xCDEF01}$. In addition, for $w \in \{1, 2, 4, 8\}$, we use $\texttt{coef}(S, i, w)$ to denote the unsigned integer represented by the i-th w-bit string of S. For example, $S = \texttt{0xABCDFF01}$, then $\texttt{coef}(S, 0, 4) = \texttt{0xA} = 10$, $\texttt{coef}(S, 6, 4) = \texttt{0x0} = 0$, and $\texttt{coef}(S, 3, 2) = 3$. Also, we always have $\texttt{coef}(S, i, 8) = \texttt{byte}(S, i)$.

3 The Leighton-Micali One-Time Signature (`LM-OTS`)

`LM-OTS` is a ote-time signature scheme, meaning that each private key must be used at most one time to sign any given message. Otherwise, the security of the signature scheme is not guaranteed. In the sequel, an `LM-OTS` private and public

key pair is referred to as an LM-OTS instance. Therefore, to generate an LM-OTS instance is to generate an LM-OTS key pair.

Before key generation, three system parameters must be determined, including the employed hash function $H(\cdot)$, the number of bytes of the output of the hash function, denoted by n, and the Winternitz parameter $w \in \{1, 2, 4, 8\}$.

3.1 LM-OTS Private Key Generation

First, set the 4-byte value otstype according to Table 1, which is compatible with RFC 8554 [MCF19] and the Internet draft [FD22] work in progress. If we instantiate the LM-OTS scheme with SM3, the value of otstype is set according to Table 3 in this article. Then, set I to be a randomly chosen 16-byte string. Next, we set $q = 0$.

Then, we need to generate $x = (x[0], \cdots, x[p-1]) \in \mathbb{B}^{np}$, where $x[j]$ is an n-byte string for $0 \leq j < p$. We call x the array of secret pre-images of the LM-OTS secret key. The number p of pre-images in the array is determined by n and $w \in \{1, 2, 4, 8\}$, and can be computed as $p = u + v$ with

$$\begin{cases} u = \frac{8n}{w} \\ v = \left\lceil \frac{\lfloor \log_2(u(2^w - 1)) \rfloor + 1}{w} \right\rceil \end{cases}, \tag{1}$$

where u is the number of w-bit fields needed to hold the n-byte output of the hash function and v is the number of w-bit fields needed to hold the checksum described in Section 3.3.

To generate the array x, there are two methods. In the first method, we can just independently set $x[j]$ to be a uniformly random n-byte string for each $j \in \{0, \cdots, p-1\}$. In the second method, we can generate $x[j]$ pseudorandomly from a secret seed SEED $\in \mathbb{B}^n$ such that

$$x[j] = H(I \| \texttt{u32str}(q) \| \texttt{u16str}(j) \| \texttt{0xFF} \| \texttt{SEED})$$

for $j \in \{0, \cdots, p-1\}$. The private key of the generated LM-OTS instance is $\mathsf{sk} = (\texttt{otstype}, I, q, x) \in \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{np}$, where $(\texttt{otstype}, I, q)$ can be published and x must be kept secret.

In the following, we will use sk.FieldName to access a particular field of the given private key sk. For example, if

$$\mathsf{sk} = (\texttt{0x00000003}, \texttt{0x00000000000000000000000000000000}, q, x),$$

Then sk.otstype = 0x00000003.

3.2 LM-OTS Public Key Generation

From the LM-OTS secret key $(\texttt{otstype}, I, q, x)$ with $x = (x[0], \cdots, x[p])$ we can compute the LM-OTS public key $\mathsf{pk} = (\texttt{otstype}, I, q, K) = \texttt{GenPubKey}(\texttt{otstype}, I, q, x)$ with Algorithm 1, where the values of $\gamma_{n,w}$ is given in Table 4 for different n and w. Similarly, we will use pk.FieldName to access a particular field of the given public key pk.

Table 1: Values for `otstype` from `RFC 8554`

`otstype` name	H	n	w	`otstype` value
Reserved	–	–	–	`0x00000000`
`LMOTS_SHA256_N32_W1`	`SHA256`	32	1	`0x00000001`
`LMOTS_SHA256_N32_W2`	`SHA256`	32	2	`0x00000002`
`LMOTS_SHA256_N32_W4`	`SHA256`	32	4	`0x00000003`
`LMOTS_SHA256_N32_W8`	`SHA256`	32	8	`0x00000004`
Unassigned	–	–	–	`0x00000005` – `0xDDDDDDDC`
Reserved for private use	–	–	–	`0xDDDDDDDD` – `0xFFFFFFFF`

Table 2: Additional values for `otstype` from the Internet draft [FD22]

`otstype` name	H	n	w	`otstype` value
`LMOTS_SHA256_N24_W1`	`SHA256-192`	24	1	`0x00000005`
`LMOTS_SHA256_N24_W2`	`SHA256-192`	24	2	`0x00000006`
`LMOTS_SHA256_N24_W4`	`SHA256-192`	24	4	`0x00000007`
`LMOTS_SHA256_N24_W8`	`SHA256-192`	24	8	`0x00000008`
`LMOTS_SHAKE_N32_W1`	`SHAKE256-256`	32	1	`0x00000009`
`LMOTS_SHAKE_N32_W2`	`SHAKE256-256`	32	2	`0x0000000A`
`LMOTS_SHAKE_N32_W4`	`SHAKE256-256`	32	4	`0x0000000B`
`LMOTS_SHAKE_N32_W8`	`SHAKE256-256`	32	8	`0x0000000C`
`LMOTS_SHAKE_N24_W1`	`SHAKE256-192`	24	1	`0x0000000D`
`LMOTS_SHAKE_N24_W2`	`SHAKE256-192`	24	2	`0x0000000E`
`LMOTS_SHAKE_N24_W4`	`SHAKE256-192`	24	4	`0x0000000F`
`LMOTS_SHAKE_N24_W8`	`SHAKE256-192`	24	8	`0x00000010`

Table 3: Values of `otstype` for `LM-OTS` instantiated with `SM3`

`otstype` name	H	n	w	`otstype` value
`LMOTS_SM3_N32_W1`	`SM3`	24	1	`0x00000011`
`LMOTS_SM3_N32_W2`	`SM3`	24	2	`0x00000012`
`LMOTS_SM3_N32_W4`	`SM3`	24	4	`0x00000013`
`LMOTS_SM3_N32_W8`	`SM3`	24	8	`0x00000014`

Table 4: The left shift offset $\gamma_{n,w}$

n	w	$\gamma_{n,w}$
32	1	7
32	2	6
32	4	4
32	8	0

Algorithm 1: $\texttt{GenPubKey}(\cdot)$: Compute the LM-OTS public key from the secret key

Input: The LM-OTS private key $(\texttt{otstype}, I, q, x) \in \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{np}$
Output: The LM-OTS public key $(\texttt{otstype}, I, q, K) \in \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{n}$

for $0 \leq i < p$ **do**
  $tmp \leftarrow x[i]$
  for $0 \leq j < 2^w - 1$ **do**
    $tmp \leftarrow H(I \parallel \texttt{u32str}(q) \parallel \texttt{u16str}(i) \parallel \texttt{u8str}(j) \parallel tmp)$
  $y[i] \leftarrow tmp$
$K \leftarrow H(I \parallel \texttt{u32str}(q) \parallel \texttt{0x8080} \parallel y[0] \parallel \cdots \parallel y[p-1])$
Return $(\texttt{otstype}, I, q, K)$

3.3 LM-OTS Signature Generation

In the signature generation process, we need to employ the following $\texttt{Cksm}(\cdot)$ function described in Algorithm 2 as a subroutine. The signature

$$\sigma = \texttt{otstype} \| C \| y[0] \| \cdots \| y[p-1] \in \mathbb{B}^4 \times \mathbb{B}^n \times \mathbb{B}^{np}$$

of a message $M \in \mathbb{F}_2^*$ can be computed by the Algorithm 3 with the secret key $(\texttt{otstype}, I, q, x)$.

Algorithm 2: $\texttt{Cksm}_{n,w}(\cdot)$: Compute the checksum of an n-byte string

Input: An n-byte string S
Output: A 16-bit unsigned integer

$sum \leftarrow 0$
for $0 \leq i < \frac{8n}{w}$ **do**
  $sum \leftarrow sum + (2^w - 1) - \texttt{coef}(S, i, w)$
Return $sum \ll \gamma_{n,w}$

3.4 LM-OTS Signature Verification

Given a message $M \in \mathbb{F}_2^*$ and its LM-OTS signature σ, σ can be verified with Algorithm 4. Note that according to Algorithm 4, a *hypothetical* public key can be computed from the LM-OTS signature.

4 The LMS Signature Scheme

Basically, the LMS signature scheme provides a method for organizing a set of 2^h LM-OTS instances in a perfect binary tree with height h such that each leaf

Algorithm 3: `LMOTS_GenSig`$(\cdot)$: Compute the `LM-OTS` signature

Input: The message $M \in \mathbb{F}_2^*$ and the private key $\mathtt{sk} = (\mathtt{otstype}, I, q, x)$
Output: The `LM-OTS` signature of M

$C \leftarrow$ A random n-byte string
$Q \leftarrow H(I \| \mathtt{u32str}(q) \| \mathtt{0x8181} \| C \| M)$
$Q \leftarrow Q \| \mathtt{Cksm}_{n,w}(Q)$
for $0 \leq i < p$ **do**
  $a \leftarrow \mathtt{coef}(Q, i, w)$
  $tmp \leftarrow x[i]$
  for $0 \leq j < a$ **do**
    $tmp \leftarrow H(I \| \mathtt{u32str}(q) \| \mathtt{u16str}(i) \| \mathtt{u8str}(j) \| tmp)$
  $y[i] \leftarrow tmp$
Return $\mathtt{otstype} \| C \| y[0] \| \cdots \| y[p-1]$

node is associated with an `LM-OTS` instance, and the root node is employed to authenticate the `LM-OTS` instances. We call this structure a Merkle tree or `LMS` tree, which corresponds to an `LMS` instances. The `LMS` instance can sign at most 2^h times in its life cycle, and each time a new signature is generated, one `LM-OTS` instance is consumed. Moreover, these `LM-OTS` instances are consumed in order. Figure 1 depicts an `LMS` tree with height 3, and the `LM-OTS` instances are consumed in the order: $T[8]$, $T[9]$, $\cdots$, $T[15]$.

Before key generation, four system parameters must be determined, including the employed hash function $H(\cdot)$, the number m of bytes associated with each node of the underlying `LMS` tree, the height of the tree, and the `otstype` of the underlying `LM-OTS` scheme employed.

4.1 `LMS` Private Key Generation

First, set the 4-byte value `lmstype` according to Table 5 and Table 6. If we instantiate the `LMS` scheme with `SM3`, the value of `lmstype` can be set according to Table 7. Then, set the 4-byte value `otstype` as described in Section 3. Then, set I to be a randomly chosen 16-byte string. Then, we set $q = 0$.

Next, we need to generate 2^h `LM-OTS` arrays $x_0, \cdots, x_{2^h-1}$ of secret pre-images with $x_i = (x_i[0], \cdots, x_i[p-1]) \in \mathbb{B}^{mp}$. There are two methods for generating these arrays. In the first method, we can just independently set x_j to be a uniformly random mp-byte string for each $i \in \{0, \cdots, 2^h - 1\}$. In the second method, we can generate $x_i[j]$ pseudorandomly from a secret seed $\mathtt{SEED} \in \mathbb{B}^m$ such that $x_i[j] = H(I \| \mathtt{u32str}(i) \| \mathtt{u16str}(j) \| \mathtt{0xFF} \| \mathtt{SEED})$ for $j \in \{0, \cdots, p-1\}$. Let $\mathbf{x} = (x_0, \cdots, x_{2^h-1})$, the `LMS` secret key is

$$(\mathtt{lmstype}, \mathtt{otstype}, I, q, \mathbf{x}) \in \mathbb{B}^4 \times \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{(2^h-1)mp}.$$

Algorithm 4: `LMOTS_VerifySig`$(\cdot)$: Verify the `LM-OTS` signature

Input: The message $M \in \mathbb{F}_2^*$, signature σ, and public key $\mathtt{pk} = (\mathtt{otstype}, I, q, K)$

Output: `True` is the signature is valid

Determine n, I, q, p and K from the public key

if $\mathtt{byte}(\sigma, 0, 3) \neq \mathtt{pk.otstype}$ **then**
    Return `INVALID`

if *σ is not $4 + n(p+1)$-byte long* **then**
    Return `INVALID`

$C \leftarrow \mathtt{byte}(\sigma, 4, 4 + n - 1)$
$y[0] \leftarrow \mathtt{byte}(\sigma, 4 + n, 4 + 2n - 1)$
$\cdots \quad \cdots$
$y[p-1] \leftarrow \mathtt{byte}(\sigma, 4 + pn, 4 + (p+1)n - 1)$

`/* Compute the hypothetical public key` K' `*/`
$Q \leftarrow H(I \| \mathtt{u32str}(q) \| \mathtt{0x8181} \| C \| M)$
$Q \leftarrow Q \| \mathtt{Cksm}_{n,w}(Q)$

for $0 \leq i < p$ **do**
    $a \leftarrow \mathtt{coef}(Q, i, w)$
    $tmp \leftarrow y[i]$
    for $a \leq j < 2^w - 1$ **do**
        $tmp \leftarrow H(I \| \mathtt{u32str}(q) \| \mathtt{u16str}(i) \| \mathtt{u8str}(j) \| tmp)$
    $z[i] \leftarrow tmp$

$K' = \leftarrow H(I \parallel \mathtt{u32str}(q) \parallel \mathtt{0x8080} \parallel z[0] \parallel \cdots \parallel z[p-1])$

if $\mathtt{pk}.K = K'$ **then**
    Return `TRUE`
else
    Return `INVALID`

Table 5: The values for lmstype from RFC 8554

lmstype name	H	m	h	lmstype value
Reserved	–	–	–	0x00000000 – 0x00000004
LMS_SHA256_M32_H5	SHA256	32	1	0x00000005
LMS_SHA256_M32_H10	SHA256	32	2	0x00000006
LMS_SHA256_M32_H15	SHA256	32	4	0x00000007
LMS_SHA256_M32_H20	SHA256	32	8	0x00000008
LMS_SHA256_M32_H25	SHA256	32	8	0x00000009
Unassigned	–	–	–	0x0000000A – 0xDDDDDDDC
Reserved for private use	–	–	–	0xDDDDDDDD – 0xFFFFFFFF

Table 6: The values for lmstype from the Internet draft [FD22]

lmstype name	H	m	h	lmstype value
LMS_SHA256_M24_H5	SHA256-192	24	5	0x0000000A
LMS_SHA256_M24_H10	SHA256-192	24	10	0x0000000B
LMS_SHA256_M24_H15	SHA256-192	24	15	0x0000000C
LMS_SHA256_M24_H20	SHA256-192	24	20	0x0000000D
LMS_SHA256_M24_H25	SHA256-192	24	25	0x0000000E
LMS_SHAKE_M32_H5	SHAKE256-256	32	5	0x0000000F
LMS_SHAKE_M32_H10	SHAKE256-256	32	10	0x00000010
LMS_SHAKE_M32_H15	SHAKE256-256	32	15	0x00000011
LMS_SHAKE_M32_H20	SHAKE256-256	32	20	0x00000012
LMS_SHAKE_M32_H25	SHAKE256-256	32	25	0x00000013
LMS_SHAKE_M24_H5	SHAKE256-192	24	5	0x00000014
LMS_SHAKE_M24_H10	SHAKE256-192	24	10	0x00000015
LMS_SHAKE_M24_H15	SHAKE256-192	24	15	0x00000016
LMS_SHAKE_M24_H20	SHAKE256-192	24	20	0x00000017
LMS_SHAKE_M24_H25	SHAKE256-192	24	25	0x00000018

Table 7: The values of lmstype for LMS instantiated with SM3

lmstype name	H	m	h	lmstype value
LMS_SM3_M32_H5	SM3	32	5	0x00000019
LMS_SM3_M32_H10	SM3	32	10	0x0000001A
LMS_SM3_M32_H15	SM3	32	15	0x0000001B
LMS_SM3_M32_H20	SM3	32	20	0x0000001C
LMS_SM3_M32_H25	SM3	32	25	0x0000001D

4.2 LMS Public Key Generation

Given the secret key $(\texttt{lmstype}, \texttt{otstype}, I, q, \mathbf{x})$, one can derive 2^h LM-OTS secret key $\mathsf{sk}_q = (\texttt{otstype}, I, q, x_q)$ for $0 \leq q < 2^h$. Then, for each $q \in \{0, \cdots, 2^h - 1\}$,

we can derive its corresponding public key pk_q with Algorithm 1, from which we can derive $\mathsf{pk}_q.K$.

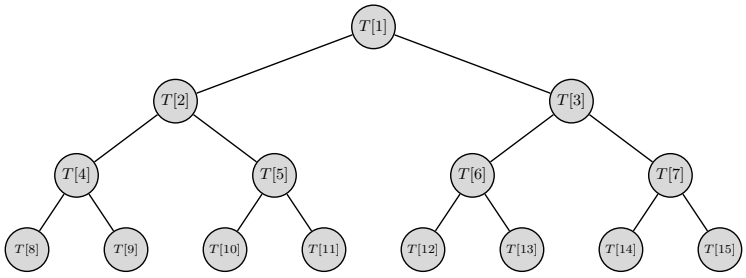

Fig. 1: An LMS tree with height 3

Let $K_q = \mathsf{pk}_q.K$. With these 2^h K_i's, we can construct a height-h perfect binary tree with $2^{h+1}-1$ nodes labeled by $T[1], T[2], \cdots, T[2^{h+1}-1]$. Figure 1 depicts a LMS tree with height 3. The values of $T[j]$'s are computed according to the following formula:

$$T[r] = \begin{cases} H(I \| \mathtt{u32str}(r) \| \mathtt{0x8282} \| K_{r-2^h}), & r \geq 2^h \\ H(I \| \mathtt{u32str}(r) \| \mathtt{0x8383} \| T[2r] \| T[2r+1]), & 0 \leq r < 2^h \end{cases} . \quad (2)$$

The LMS public key is $(\mathtt{lmstype}, \mathtt{otstype}, I, T[1]) \in \mathbb{B}^4 \times \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^m$, where $T[1]$ is the root of the associated LMS tree.

4.3 LMS Signature Generation

Given a message $M \in \mathbb{F}_2^*$ and an LMS secret key $(\mathtt{lmstype}, \mathtt{otstype}, I, q, \mathbf{x})$, the signature of M is

$$(\mathtt{u32str}(q), \sigma_q(M), \mathtt{lmstype}, (\mathtt{path}[0], \mathtt{path}[1], \cdots, \mathtt{path}[h-1])),$$

where
$sigma_q(M) = \mathtt{LMOTS\_GenSig}(M, (\mathtt{lmstype}, \mathtt{otstype}, I, q, \mathbf{x}))$, and

$$(\mathtt{path}[0], \cdots, \mathtt{path}[h-1])$$

is the authentication path of $T[2^h+q]$. For example, if $h=3$ and $q=1$, then $(\mathtt{path}[0], \mathtt{path}[1], \mathtt{path}[2]) = (T[8], T[5], T[3])$ as shown in Figure 2.

4.4 LMS Signature Verification

Let $(\mathtt{u32str}(q), \sigma_q(M), \mathtt{lms\_type}, (\mathtt{path}[0], \mathtt{path}[1], \cdots, \mathtt{path}[h-1]))$ be the LMS signature of a message M. Then, we can compute the *hypothetical* value of the q-th leaf node $T'[2^h+q]$ from $\sigma_q(M)$. Then, with the information provided in

$$(\mathtt{path}[0], \mathtt{path}[1], \cdots, \mathtt{path}[h-1]),$$

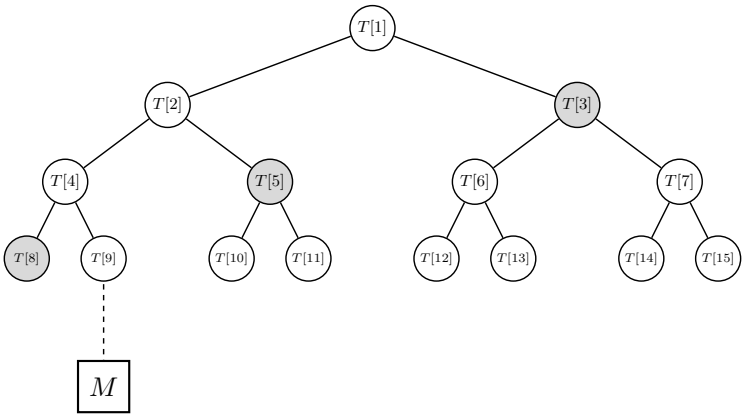

Fig. 2: An LMS tree with height 3

we can compute the *hypothetical* value $T'[1]$ of the root. The signature is valid if and only if $T[1] = T'[1]$. After generating the signature, the signing process increments the q value in the private key by 1. This is why we say that LMS is a stateful signature scheme.

5 The Hierarchical Signature Scheme HSS

Like LMS, The HSS scheme is another method for organizing a large set of LM-OTS instances. The time complexity for generating an LMS key pair can be very high if the height of the corresponding LMS tree is large, since all of the tree nodes have to be computed to obtain the root.

The HSS signature scheme can be employed if we want to reduce the time taken by the key generation process. In HSS, LM-OTS instances are associated with the leaves of many LMS trees placed at different levels. The LMS trees are "connected" in the sense that the roots of the LMS trees are signed by the LM-OTS instances associated with the leaves of the LMS trees in the upper level. The leaves of the lowest level LMS trees are used to sign the messages. We call this structure an HSS tree or HSS instance. A 3-level HSS tree signing a message M is illustrated in Figure 3, where only the involved LMS trees are displayed.

In HSS, we have L layers of LMS instances, including layer 0, $\cdots$, layer $L-1$. The LMS instances in the same layer have the same height. Let h_i be the height of the LMS trees in layer i. In the 0-th layer, there is only 1 LMS tree. For $1 \leq i < L$, there are $2^{\sum_{j=0}^{i-1} h_j}$ LMS trees in the i-th layer. Let $H = \sum_{j=0}^{L-1} h_j$, then there are 2^H leaves in the $(L-1)$-th layer, which correspond to 2^H LM-OTS keys. Such an HSS instance can sign at most 2^H times, and we call 2^h is the capacity. Note that a 1-level HSS instances is essentially an LMS instance.

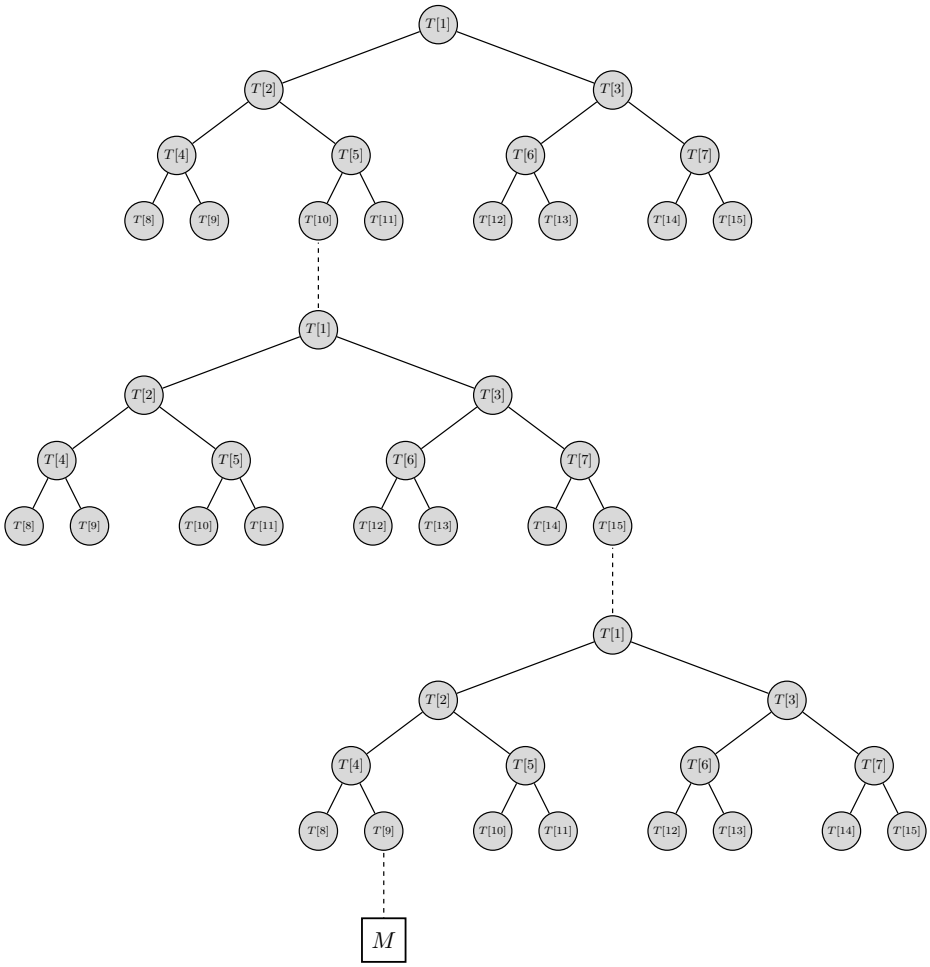

Fig. 3: A 3-level HSS tree

5.1 HSS Key Generation

First, using the method described in Section 4, we generate one LMS key pair. We store the private key into $\texttt{prv}[0]$ and store the public key into $\texttt{pub}[0]$.

For each $i \in \{1, 2, \cdots, L-1\}$, we independently generate an LMS key pair. We store the private key into $\texttt{prv}[i]$ and store the public key into $\texttt{pub}[i]$. Also, we sign $\texttt{pub}[i]$ with $\texttt{prv}[i-1]$, and store the signature into $\texttt{sig}[i-1]$. Now, we have 3 arrays.

$$\begin{cases} \texttt{prv} = (\texttt{prv}[0], \texttt{prv}[1], \cdots, \texttt{prv}[L-2], \texttt{prv}[L-1]) \\ \texttt{pub} = (\texttt{pub}[0], \texttt{pub}[1], \cdots, \texttt{pub}[L-2], \texttt{pub}[L-1]) \\ \texttt{sig} = (\texttt{sig}[0], \texttt{sig}[1], \cdots, \texttt{sig}[L-2]) \end{cases} . \tag{3}$$

The HSS private key is $(\texttt{prv}, \texttt{pub}, \texttt{sig})$, and the HSS public key is $\texttt{u32str}(L) \parallel \texttt{pub}[0]$.

5.2 HSS Signature Generation

There are 3 cases for different values of L. When $L = 0$, the signature is $\texttt{u32str}(0) \parallel \texttt{sig}[0]$. When $L > 0$, the signature is

$$\texttt{u32str}(\texttt{Nspk}) \parallel \texttt{signed\_pub\_key}[0] \parallel \cdots \parallel \texttt{signed\_pub\_key}[\texttt{Nspk}-1] \parallel \texttt{sig}[\texttt{Nspk}],$$

where $\texttt{Nspk} = L - 1$ (Number of Signed Public Keys) denotes the number of signed LMS public keys. The above signature can be obtained by Algorithm 5.

Algorithm 5: Compute the HSS signature of a message M

$d \leftarrow L$
while $\texttt{prv}[d-1].q = 2^{\texttt{prv}[d-1].h}$ **do**
  $d \leftarrow d - 1$
  if $d = 0$ **then**
    return FAILURE
while $d < L$ **do**
  Randomly generate $\texttt{pub}[d]$ and $\texttt{prv}[d]$
  $\texttt{sig}[d-1] \leftarrow$ The LMS signature of $\texttt{pub}[d]$ signed with $\texttt{prv}[d-1]$
  $d \leftarrow d + 1$
$\texttt{sig}[L-1] \leftarrow$ The LMS signature of M signed with $\texttt{prv}[L-1]$
for $0 \le i < L - 1$ **do**
  $\texttt{signed\_pub\_key}[i] \leftarrow \texttt{sig}[i] \| \texttt{pub}[i+1]$
return $\texttt{u32str}(L-1) \parallel \texttt{signed\_pub\_key}[0] \parallel \cdots \parallel \texttt{signed\_pub\_key}[L-2] \parallel \texttt{sig}[L-1]$

5.3 Signature Verification

The signature is valid if and only if all of

$$\texttt{signed\_pub\_key}[0], \cdots, \texttt{signed\_pub\_key}[L-2]$$

and $\texttt{sig}[L-1]$ are valid.

6 Preliminary Implementations and Performance Test

We instantiate the hash-based signature schemes described in previous sections with SM3, and we name them as LMS-SM3 and HSS-SM3. We implement LMS-SM3 and HSS-SM3 based on the code provided at https://github.com/cisco/hash-sigs by substituting the underlying hash function with an implementation of SM3. The

performance of the implementation is provided in Table 8, which are obtained on a server machine with 32 cores running `Linux ubuntu 18.04` on 2.9GHz `AMD EPYC-Rome` Processor.

In Table 8, the "Height" column records the heights of the `LMS` trees in `LMS-SM3` or `HSS-SM3`, where a single number h means a `LMS-SM3` instance with height h, and h_0/h_1 means a two level `HSS-SM3` instance with height h_0 in the 0-th level and height h_1 in the 1st level.

7 Conclusion

In this work, we instantiate the hash-based signature schemes `LMS` and `HSS` described in `RFC 8554` with `SM3` and conduct some preliminary performance test. In the future, we will provide implementations of `LMS-SM3`, `HSS-SM3`, and `XMSS-SM3` on various platforms and deploy them in real application scenarios to test the applicability of hash-based signature schemes.

References

BDH11. 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, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.

BHH+15. 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, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.

BHK+19. Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2129–2146. ACM, 2019.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1825–1842. ACM, 2017.

DH76. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

Table 8: Preliminary performance test of HSS-SM3

w	Height	`KeyGenTime` (s)	PubKeySize (Byte)	`PrvKeySize` (Byte)	`SignTime` (s)	`SigSize` (Byte)	`VerifyTime` (ms)	Capacity
4	5	0.011	60	64	0.0056	2,352	0.0015	2^5
	10	0.098	60	64	0.0079	2,512	0.0014	2^{10}
	15	2.91	60	64	0.019	2,672	0.0018	2^{15}
	20	100.02	60	64	0.019	2,832	0.002	2^{20}
	15/10	2.99	60	64	0.01	5,236	0.0034	2^{25}
	15/15	3.18	60	64	0.01	5,396	0.0048	2^{30}
	20/10	107.35	60	64	0.013	5,396	0.0038	2^{30}
	20/15	118.06	60	64	0.011	5,536	0.003	2^{35}
8	5	0.033	60	64	0.024	1,296	0.0033	2^5
	10	0.697	60	64	0.042	1,456	0.0036	2^{10}
	15	21.35	60	64	0.01	1,616	0.0046	2^{15}
	20	928.21	60	64	0.09	1,776	0.005	2^{20}
	15/10	23.95	60	64	0.056	3,124	0.01	2^{25}
	15/15	25.62	60	64	0.068	3,284	0.01	2^{30}
	20/10	911.42	60	64	0.057	3,284	0.01	2^{30}
	20/15	960.46	60	64	0.051	3,444	0.0087	2^{35}

FD22. Scott Fluhrer and Quynh Dang. Additional Parameter sets for LMS Hash-Based Signatures. Internet-Draft draft-fluhrer-lms-more-parm-sets-08, Internet Engineering Task Force, 2022. Work in Progress.

HBG+18. Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, 2018.

HRB17. Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for xmssˆmt. *IACR Cryptol. ePrint Arch.*, page 966, 2017.

Lam79. Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979.

MCF19. David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, 2019.

Mer89. Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

NIS20. NIST. Recommendation for stateful hash-based signature schemes. NIST SP 800-208, 2020.

Sho99. Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, 41(2):303–332, 1999.

vdLPR+18. Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. Is java card ready for hash-based signatures? In Atsuo Inomata and Kan Yasuda, editors, *Advances in Information and Computer Security - 13th International Workshop on Security, IWSEC 2018, Sendai, Japan, September 3-5, 2018, Proceedings*, volume 11049 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2018.

WJW+19. Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 523–550. Springer, 2019.

ZCY22. Kaiyi Zhang, Hongrui Cui, and Yu Yu. Sphincs-α: A compact stateless hash-based signature scheme. *IACR Cryptol. ePrint Arch.*, page 59, 2022.