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

Siwei Sun¹, Tianyu Liu¹, Zhi Guan², Yifei He², Jiwu Jing¹, Lei Hu¹, Zhenfeng Zhang³, Hailun Yan¹

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 $F_2 = \{0, 1\}$ be the binary field and $B = F_2^8$ 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 $(0001\ 1111\ 0001\ 0010)$.

Sometimes, we need to convert a unsigned integer into a string of bytes, which is done by applying the conversion functions $u8str(\cdot)$, $u16str(\cdot)$, and $u32str(\cdot)$. For example, $u8str(11) = 0x0B$, $u16str(6214) = 0xF2BC$, and $u32str(305441741) = 0x1234ABCD$. Conversely, we use $int32(\cdot)$ to convert an integer $i \in \{0, 1, \cdots, 2^{32} - 1\}$ to a 4-byte string. For example, $int32(0x1234ABCD) = 305441741$. Also, $u32str(int32(S)) = S$ holds for any 4-byte string $S$.

Let $S$ be a byte string. Then, $byte(S, i)$ denotes the $i$-th byte of $S$, and $byte(S, i, j)$ denotes the range of bytes from the $i$-th to the $j$-th byte, inclusive. For example, if $S = 0xABCDF01$, then $byte(S, 0) = 0xAB$, $byte(S, 3) = 0x01$, and $byte(S, 1, 3) = 0xCDEF01$. In addition, for $w \in \{1, 2, 4, 8\}$, we use $coef(S, i, w)$ to denote the unsigned integer represented by the $i$-th $w$-bit string of $S$. For example, $S = 0xABCDF01$, then $coef(S, 0, 4) = 0xA = 10$, $coef(S, 6, 4) = 0x0 = 0$, and $coef(S, 3, 2) = 3$. Also, we always have $coef(S, i, 8) = byte(S, i)$.

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

LM-OTS is a one-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 \( \text{otstype} \) 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 \( \text{otstype} \) 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

\[
\begin{align*}
    u &= \frac{8n}{w} \\
    v &= \left\lceil \log_2(u(2^w-1))^w \right\rceil + 1
\end{align*}
\]

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 \( \text{SEED} \in \mathbb{B}^{n} \) such that

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

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

In the following, we will use \( \text{sk.FieldName} \) to access a particular field of the given private key \( \text{sk} \). For example, if

\[
\text{sk} = (0x000000003, 0x00000000000000000000000000000000000000000, q, x),
\]

Then \( \text{sk.\text{otstype}} = 0x00000003 \).

### 3.2 LM-OTS Public Key Generation

From the LM-OTS secret key \( (\text{otstype}, I, q, x) \) with \( x = (x[0], \cdots, x[p]) \) we can compute the LM-OTS public key \( \text{pk} = (\text{otstype}, I, q, K) = \text{GenPubKey(otstype, I, q, x)} \) with Algorithm 1, where the values of \( \gamma_{n,w} \) is given in Table 4 for different \( n \) and \( w \). Similarly, we will use \( \text{pk.FieldName} \) to access a particular field of the given public key \( \text{pk} \).
### Table 1: Values for `otstype` from RFC 8554

<table>
<thead>
<tr>
<th><code>otstype</code> name</th>
<th><code>H</code></th>
<th><code>n</code></th>
<th><code>w</code></th>
<th><code>otstype value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td>0x0000000000</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W1</td>
<td>SHA256</td>
<td>32</td>
<td>1</td>
<td>0x00000001</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W2</td>
<td>SHA256</td>
<td>32</td>
<td>2</td>
<td>0x00000002</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W4</td>
<td>SHA256</td>
<td>32</td>
<td>4</td>
<td>0x00000003</td>
</tr>
<tr>
<td>LMOTS_SHA256_N32_W8</td>
<td>SHA256</td>
<td>32</td>
<td>8</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Unassigned</td>
<td></td>
<td></td>
<td></td>
<td>0x00000005 - 0xDDDDDDDC</td>
</tr>
<tr>
<td>Reserved for private use</td>
<td></td>
<td></td>
<td></td>
<td>0xDDDDDDDD - 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><code>otstype</code> name</th>
<th><code>H</code></th>
<th><code>n</code></th>
<th><code>w</code></th>
<th><code>otstype value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>LMOTS_SHA256_N24_W1</td>
<td>SHA256-192</td>
<td>24</td>
<td>1</td>
<td>0x00000005</td>
</tr>
<tr>
<td>LMOTS_SHA256_N24_W2</td>
<td>SHA256-192</td>
<td>24</td>
<td>2</td>
<td>0x00000006</td>
</tr>
<tr>
<td>LMOTS_SHA256_N24_W4</td>
<td>SHA256-192</td>
<td>24</td>
<td>4</td>
<td>0x00000007</td>
</tr>
<tr>
<td>LMOTS_SHA256_N24_W8</td>
<td>SHA256-192</td>
<td>24</td>
<td>8</td>
<td>0x00000008</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N32_W1</td>
<td>SHAKE256-256</td>
<td>32</td>
<td>1</td>
<td>0x00000009</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N32_W2</td>
<td>SHAKE256-256</td>
<td>32</td>
<td>2</td>
<td>0x0000000A</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N32_W4</td>
<td>SHAKE256-256</td>
<td>32</td>
<td>4</td>
<td>0x0000000B</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N32_W8</td>
<td>SHAKE256-256</td>
<td>32</td>
<td>8</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N24_W1</td>
<td>SHAKE256-192</td>
<td>24</td>
<td>1</td>
<td>0x0000000D</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N24_W2</td>
<td>SHAKE256-192</td>
<td>24</td>
<td>2</td>
<td>0x0000000E</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N24_W4</td>
<td>SHAKE256-192</td>
<td>24</td>
<td>4</td>
<td>0x0000000F</td>
</tr>
<tr>
<td>LMOTS_SHAKE_N24_W8</td>
<td>SHAKE256-192</td>
<td>24</td>
<td>8</td>
<td>0x00000010</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><code>otstype</code> name</th>
<th><code>H</code></th>
<th><code>n</code></th>
<th><code>w</code></th>
<th><code>otstype value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>LMOTS_SM3_N32_W1</td>
<td>SM3</td>
<td>24</td>
<td>1</td>
<td>0x00000011</td>
</tr>
<tr>
<td>LMOTS_SM3_N32_W2</td>
<td>SM3</td>
<td>24</td>
<td>2</td>
<td>0x00000012</td>
</tr>
<tr>
<td>LMOTS_SM3_N32_W4</td>
<td>SM3</td>
<td>24</td>
<td>4</td>
<td>0x00000013</td>
</tr>
<tr>
<td>LMOTS_SM3_N32_W8</td>
<td>SM3</td>
<td>24</td>
<td>8</td>
<td>0x00000014</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>( w )</th>
<th>$\gamma_{n,w}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm 1: GenPubKey\((\cdot)\): Compute the LM-OTS public key from the secret key

**Input:** The LM-OTS private key \((\text{otype}, I, q, x) \in \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{np}\)

**Output:** The LM-OTS public key \((\text{otype}, I, q, K) \in \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{n}\)

1. for \(0 \leq i < p\) do
2. \(\text{tmp} \leftarrow x[i]\)
3. for \(0 \leq j < 2^w - 1\) do
4. \(\text{tmp} \leftarrow H(I || \text{u32str}(q) || \text{u16str}(i) || \text{u8str}(j) || \text{tmp})\)
5. \(y[i] \leftarrow \text{tmp}\)
6. \(K \leftarrow H(I || \text{u32str}(q) || 0x8080 || y[0] || \cdots || y[p-1])\)
7. Return \((\text{otype}, I, q, K)\)

3.3 LM-OTS Signature Generation

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

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

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

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

**Input:** An \(n\)-byte string \(S\)

**Output:** A 16-bit unsigned integer

1. \(\text{sum} \leftarrow 0\)
2. for \(0 \leq i < \frac{nw}{w}\) do
3. \([\text{sum} \leftarrow \text{sum} + (2^w - 1) - \text{coef}(S, i, w)\]
4. Return \(\text{sum} \ll \gamma_{n,w}\)

3.4 LM-OTS Signature Verification

Given a message \(M \in \mathbb{F}_2^*\) and its LM-OTS signature \(\sigma\), \(\sigma\) 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(·): Compute the LM-OTS signature

**Input:** The message $M \in \mathbb{F}_p^*$ and the private key $sk = (\text{otype}, I, q, x)$

**Output:** The LM-OTS signature of $M$

1. $C \leftarrow$ A random $n$-byte string
2. $Q \leftarrow H(I||u32str(q)||0x8181||C||M)$
3. $Q \leftarrow Q||c_{x^{n-1}}(Q)$
4. for $0 \leq i < p$ do
   
   a. $a \leftarrow \text{coef}(Q, i, w)$
   
   b. $tmp \leftarrow x[i]
   
   c. for $0 \leq j < a$ do
      
      i. $tmp \leftarrow H(I||u32str(q)||u16str(i)||u8str(j)||tmp)
   
   d. $y[i] \leftarrow tmp$

5. Return $\text{otype}||C||y[0]\cdots||y[p-1]$

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 instance. 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 otype 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 otype 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 $x_i = (x_i[0], \cdots, x_i[p-1]) \in \mathbb{B}^m$. 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 $\text{SEED} \in \mathbb{B}^m$ such that $x_i[j] = H(I||u32str(i)||u16str(j)||0x8181||\text{SEED})$ for $j \in \{0, \cdots, p-1\}$. Let $x = (x_0, \cdots, x_{2^h-1})$, the LMS secret key is

$$(\text{lmstype}, \text{otype}, I, q, x) \in \mathbb{B}^4 \times \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^4 \times \mathbb{B}^{(2^h-1)mp}.$$
Algorithm 4: LMOTS\_VerifySig(\cdot): Verify the LM-OTS signature

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

**Output:** True if the signature is valid

1. Determine $n$, $I$, $q$, $p$ and $K$ from the public key
2. if byte($\sigma$, 0, 3) \neq pk.\text{otstype} then
   3. Return INVALID
3. if $\sigma$ is not $4 + n(p + 1)$-byte long then
   4. Return INVALID
4. $C \leftarrow$ byte($\sigma$, 4, 4 + $n - 1$)
5. $y[0] \leftarrow$ byte($\sigma$, 4 + $n$, 4 + 2$n - 1$)
6. \ldots \ldots
7. $y[p - 1] \leftarrow$ byte($\sigma$, 4 + $pn$, 4 + ($p + 1)n - 1$)
8. /* Compute the hypothetical public key $K'$ */
9. $Q \leftarrow H(I\|u32str(q)\|0x8181\|C\|M)$
10. $Q \leftarrow Q\|\text{Cksm}_{n,w}(Q)$
11. for 0 \leq i < p do
12.  a \leftarrow \text{coef}(Q, i, w)
13.  tmp \leftarrow y[i]
14.  for a \leq j < 2^w - 1 do
15.   tmp \leftarrow H(I\|u32str(q)\|u16str(i)\|u8str(j)\|tmp)
16.   z[i] \leftarrow tmp
17. $K' \leftarrow H(I \|| u32str(q) \| 0x8080 \| z[0] \| \cdots \| z[p - 1])$
18. if $pk.K = K'$ then
19.   Return TRUE
20. else
21.   Return INVALID
Table 5: The values for lmstype from RFC 8554

<table>
<thead>
<tr>
<th>lmstype name</th>
<th>H</th>
<th>m</th>
<th>h</th>
<th>lmstype value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0x00000000 - 0x00000004</td>
</tr>
<tr>
<td>LMS_SHA256_M32_H5</td>
<td>SHA256</td>
<td>32</td>
<td>1</td>
<td>0x00000005</td>
</tr>
<tr>
<td>LMS_SHA256_M32_H10</td>
<td>SHA256</td>
<td>32</td>
<td>2</td>
<td>0x00000006</td>
</tr>
<tr>
<td>LMS_SHA256_M32_H15</td>
<td>SHA256</td>
<td>32</td>
<td>4</td>
<td>0x00000007</td>
</tr>
<tr>
<td>LMS_SHA256_M32_H20</td>
<td>SHA256</td>
<td>32</td>
<td>8</td>
<td>0x00000008</td>
</tr>
<tr>
<td>LMS_SHA256_M32_H25</td>
<td>SHA256</td>
<td>32</td>
<td>8</td>
<td>0x00000009</td>
</tr>
<tr>
<td>Unassigned</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0x0000000A - 0xDDDDDDDC</td>
</tr>
<tr>
<td>Reserved for private use</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0xDDDDDDDD - 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>lmstype name</th>
<th>H</th>
<th>m</th>
<th>h</th>
<th>lmstype value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS_SHA256_M24_H5</td>
<td>SHA256</td>
<td>24</td>
<td>5</td>
<td>0x00000000A</td>
</tr>
<tr>
<td>LMS_SHA256_M24_H10</td>
<td>SHA256</td>
<td>24</td>
<td>10</td>
<td>0x00000000B</td>
</tr>
<tr>
<td>LMS_SHA256_M24_H15</td>
<td>SHA256</td>
<td>24</td>
<td>15</td>
<td>0x00000000C</td>
</tr>
<tr>
<td>LMS_SHA256_M24_H20</td>
<td>SHA256</td>
<td>24</td>
<td>20</td>
<td>0x00000000D</td>
</tr>
<tr>
<td>LMS_SHA256_M24_H25</td>
<td>SHA256</td>
<td>24</td>
<td>25</td>
<td>0x00000000E</td>
</tr>
<tr>
<td>LMS_SHAKE_M32_H5</td>
<td>SHAKE256</td>
<td>32</td>
<td>5</td>
<td>0x00000000F</td>
</tr>
<tr>
<td>LMS_SHAKE_M32_H10</td>
<td>SHAKE256</td>
<td>32</td>
<td>10</td>
<td>0x000000010</td>
</tr>
<tr>
<td>LMS_SHAKE_M32_H15</td>
<td>SHAKE256</td>
<td>32</td>
<td>15</td>
<td>0x000000011</td>
</tr>
<tr>
<td>LMS_SHAKE_M32_H20</td>
<td>SHAKE256</td>
<td>32</td>
<td>20</td>
<td>0x000000012</td>
</tr>
<tr>
<td>LMS_SHAKE_M32_H25</td>
<td>SHAKE256</td>
<td>32</td>
<td>25</td>
<td>0x000000013</td>
</tr>
<tr>
<td>LMS_SHAKE_M24_H5</td>
<td>SHAKE256</td>
<td>24</td>
<td>5</td>
<td>0x000000014</td>
</tr>
<tr>
<td>LMS_SHAKE_M24_H10</td>
<td>SHAKE256</td>
<td>24</td>
<td>10</td>
<td>0x000000015</td>
</tr>
<tr>
<td>LMS_SHAKE_M24_H15</td>
<td>SHAKE256</td>
<td>24</td>
<td>15</td>
<td>0x000000016</td>
</tr>
<tr>
<td>LMS_SHAKE_M24_H20</td>
<td>SHAKE256</td>
<td>24</td>
<td>20</td>
<td>0x000000017</td>
</tr>
<tr>
<td>LMS_SHAKE_M24_H25</td>
<td>SHAKE256</td>
<td>24</td>
<td>25</td>
<td>0x000000018</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>lmstype name</th>
<th>H</th>
<th>m</th>
<th>h</th>
<th>lmstype value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS_SM3_M32_H5</td>
<td>SM3</td>
<td>32</td>
<td>5</td>
<td>0x000000019</td>
</tr>
<tr>
<td>LMS_SM3_M32_H10</td>
<td>SM3</td>
<td>32</td>
<td>10</td>
<td>0x00000001A</td>
</tr>
<tr>
<td>LMS_SM3_M32_H15</td>
<td>SM3</td>
<td>32</td>
<td>15</td>
<td>0x00000001B</td>
</tr>
<tr>
<td>LMS_SM3_M32_H20</td>
<td>SM3</td>
<td>32</td>
<td>20</td>
<td>0x00000001C</td>
</tr>
<tr>
<td>LMS_SM3_M32_H25</td>
<td>SM3</td>
<td>32</td>
<td>25</td>
<td>0x00000001D</td>
</tr>
</tbody>
</table>

4.2 LMS Public Key Generation

Given the secret key (lmstype, otstype, I, q, x), one can derive \(2^h\) LMS-OTS secret key \(sk_q = (otstype, I, q, x_q)\) for \(0 \leq q < 2^h\). Then, for each \(q \in \{0, \ldots, 2^h - 1\}\),
we can derive its corresponding public key \( pk_q \) with Algorithm 1, from which we can derive \( pk_q, K \).

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

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

The LMS public key is \((\text{lmstype, otstype}, I, T[1]) \in \mathbb{B}^4 \times \mathbb{B}^4 \times \mathbb{B}^{16} \times \mathbb{B}^m\), 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 \((\text{lmstype, otstype}, I, q, x)\), the signature of \( M \) is

\[(u32str(q), \sigma_q(M), \text{lmstype}, \text{path}[0], \text{path}[1], \ldots, \text{path}[h-1]),\]

where

\[\sigma_q(M) = \text{LMOTS\_GenSig}(M, (\text{lmstype, otstype}, I, q, x))\]

and

\[(\text{path}[0], \ldots, \text{path}[h-1])\]

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

4.4 LMS Signature Verification

Let \((u32str(q), \sigma_q(M), \text{lmstype}, \text{path}[0], \text{path}[1], \ldots, \text{path}[h-1])\) 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

\[(\text{path}[0], \text{path}[1], \ldots, \text{path}[h-1]),\]
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, · · · , 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 $2^{L−1−i}$ 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 $2^h$ 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 $\text{prv}[0]$ and store the public key into $\text{pub}[0]$.

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

\[
\begin{align*}
\text{prv} &= (\text{prv}[0], \text{prv}[1], \cdots, \text{prv}[L - 2], \text{prv}[L - 1]) \\
\text{pub} &= (\text{pub}[0], \text{pub}[1], \cdots, \text{pub}[L - 2], \text{pub}[L - 1]) \\
\text{sig} &= (\text{sig}[0], \text{sig}[1], \cdots, \text{sig}[L - 2])
\end{align*}
\]
The HSS private key is \((\text{prv}, \text{pub}, \text{sig})\), and the HSS public key is \(\text{u32str}(L) \parallel \text{pub}[0]\).

5.2 HSS Signature Generation

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

\(\text{u32str}(\text{Nspk}) \parallel \text{signed_pub_key}[0] \parallel \cdots \parallel \text{signed_pub_key}[\text{Nspk}-1] \parallel \text{sig}[\text{Nspk}]\),

where \(\text{Nspk} = L - 1\) (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 \(M\)

1. \(d \leftarrow L\)
2. while \(\text{prv}[d-1].q = 2^{\text{prv}[d-1].h}\) do
   3. \(d \leftarrow d - 1\)
   4. if \(d = 0\) then
   5. return FAILURE
3. while \(d < L\) do
   6. Randomly generate \(\text{pub}[d]\) and \(\text{prv}[d]\)
   7. \(\text{sig}[d-1] \leftarrow \text{The LMS signature of pub}[d] \text{ signed with prv}[d-1]\)
   8. \(d \leftarrow d + 1\)
9. \(\text{sig}[L-1] \leftarrow \text{The LMS signature of } M \text{ signed with prv}[L-1]\)
10. for \(0 \leq i < L - 1\) do
11. \(\text{signed_pub_key}[i] \leftarrow \text{sig}[i] \parallel \text{pub}[i+1]\)
13. return \(\text{u32str}(L-1) \parallel \text{signed_pub_key}[0] \parallel \cdots \parallel \text{signed_pub_key}[L-2] \parallel \text{sig}[L-1]\)

5.3 Signature Verification

The signature is valid if and only if all of

\(\text{signed_pub_key}[0], \cdots, \text{signed_pub_key}[L-2]\)

and \(\text{sig}[L-1]\) 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


<table>
<thead>
<tr>
<th>Capacity</th>
<th>Height</th>
<th>KeyGenTime (s)</th>
<th>PubKeySize (Byte)</th>
<th>PrvKeySize (Byte)</th>
<th>SignTime (s)</th>
<th>SigSize (Byte)</th>
<th>VerifyTime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>0.011</td>
<td>60</td>
<td>64</td>
<td>0.0056</td>
<td>2,352</td>
<td>0.0015</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>0.098</td>
<td>60</td>
<td>64</td>
<td>0.0079</td>
<td>2,512</td>
<td>0.0014</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>2.91</td>
<td>60</td>
<td>64</td>
<td>0.019</td>
<td>2,672</td>
<td>0.0018</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>100.02</td>
<td>60</td>
<td>64</td>
<td>0.019</td>
<td>2,832</td>
<td>0.0020</td>
</tr>
<tr>
<td>15/10</td>
<td>2.99</td>
<td>60</td>
<td>64</td>
<td>0.01</td>
<td>5,236</td>
<td>0.0034</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>15/15</td>
<td>3.18</td>
<td>60</td>
<td>64</td>
<td>0.01</td>
<td>5,396</td>
<td>0.0048</td>
</tr>
<tr>
<td>25</td>
<td>15/20</td>
<td>107.35</td>
<td>60</td>
<td>64</td>
<td>0.013</td>
<td>5,396</td>
<td>0.0038</td>
</tr>
<tr>
<td>30</td>
<td>20/15</td>
<td>118.06</td>
<td>60</td>
<td>64</td>
<td>0.011</td>
<td>5,536</td>
<td>0.0030</td>
</tr>
<tr>
<td>30</td>
<td>20/25</td>
<td>928.21</td>
<td>60</td>
<td>64</td>
<td>0.01</td>
<td>5,536</td>
<td>0.0050</td>
</tr>
<tr>
<td>35</td>
<td>20/30</td>
<td>23.95</td>
<td>60</td>
<td>64</td>
<td>0.056</td>
<td>3,124</td>
<td>0.010</td>
</tr>
<tr>
<td>30</td>
<td>20/35</td>
<td>25.62</td>
<td>60</td>
<td>64</td>
<td>0.068</td>
<td>3,284</td>
<td>0.013</td>
</tr>
<tr>
<td>30</td>
<td>20/40</td>
<td>911.42</td>
<td>60</td>
<td>64</td>
<td>0.057</td>
<td>3,284</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Table 8: Preliminary performance test of HSS-SN3


