XMSS-SM3 and MT-XMSS-SM3: Instantiating Extended Merkle 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 XMSS and its multi-tree version described in RFC 8391 and NIST SP 800-208 with SM3, and report on the results of the preliminary performance test.

Keywords: Hash Functions, Digital Signatures, XMSS, MT-XMSS, 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 messages 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. In section 3, we show how hash functions and pseudo-random functions are used in `XMSS` and `MT-XMSS`. In section 4, we describe the address scheme for randomizing the hash function calls. Section 5 describes the one-time signature scheme `WOTS+` which is employed as a building block for the many-time signature schemes `XMSS` and `MT-XMSS` introduced in Section 6 and Section 7, respectively. In Section 8, we instantiate these hash-based signature schemes with `SM3` and report on the results of the performance test on some preliminary implementations without extensive optimization.

Remark 1. This article and [SLG+22] share a lot of similarities, and [SLG+22] provides more details, where we instantiate the hash-based stateful signature schemes `LMS` and `HSS` described in `RFC 8554` and `NIST SP 800-208` with `SM3`.

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 i into a string of n bytes, which is done by applying the conversion function $\texttt{toBytes}(i, n)$. For example, $\texttt{toBytes}(11, 1) = \texttt{0x0B}$, $\texttt{toBytes}(6214, 2) = \texttt{0xF2BC}$, and $\texttt{toBytes}(305441741, 4) = \texttt{0x1234ABCD}$.

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 $w \in \{2, 4\}$, 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 Keyed Hash Functions and Pseudo-Random Functions

The hash functions we employed in XMSS and MT-XMSS are keyed functions implemented from un-keyed ones. Since standard hash functions like SHA2-256 and SM3 do not provide a keyed mode themselves. we use the following approach illustrated with SM3 to implement the keyed hash function:

$$\begin{cases} F(\texttt{KEY}, M) = \texttt{SM3}(\texttt{toBytes}(0, 32) \| \texttt{KEY} \| M) \\ H(\texttt{KEY}, M) = \texttt{SM3}(\texttt{toBytes}(1, 32) \| \texttt{KEY} \| M) \\ H_{\texttt{msg}}(\texttt{KEY}, M) = \texttt{SM3}(\texttt{toBytes}(2, 32) \| \texttt{KEY} \| M) \\ \text{PRF}(\texttt{KEY}, M) = \texttt{SM3}(\texttt{toBytes}(3, 32) \| \texttt{KEY} \| M) \end{cases} .$$

The keyed function F is used in the WOTS+ one-time signature scheme to compute the hash chains. H is used to compute the nodes of the so-called L-trees and XMSS-trees. In XMSS and MT-XMSS, before signature generation and verification, the relevant message $M \in \mathbb{F}_2^*$ is hashed with $H_{\texttt{msg}}$ to produce an n-byte hash value. PRF is used to generate the needed secret strings, the keys (not necessarily secret) to the hash functions, the bit masks required to build the relevant virtual data structures, and other pseudo-random material (e.g., the random string r required by $H_{\texttt{msg}}$) needed.

When the pseudo-random generator PRF is used to generate the keys and masks required to compute the hash chains in WOTS+ and build the XMSS-trees and L-trees, the input to the PRF is a public seed and an address. When the pseudo-random generator PRF is used to generate some secret elements needed in the WOTS+ instances, PRF receives a secret seed and a 32-byte string encoding an integer indexing the secret element to be generated. For an XMSS (the single tree case) instance with height h, we need 2^h WOTS+ instances. Assuming each WOTS+ instance requires l secret n-byte strings, then these secret strings can be computed with the following method. For $0 \leq i < 2^h$, we can compute 2^h secret seeds for the WOTS+ instances with $SS_i = \text{PRF}(S_{\text{PRF}}, \texttt{toBytes}(i, 32))$. Then, the j-th secret n-byte string for the i-th WOTS+ instance can be computed as $\text{PRF}(SS_i, \texttt{toBytes}(j, 32))$. Following the above strategy, it is easy to devise a method for generating all the secrete strings in the MT-XMSS case. For the j-th XMSS tree in the i-th layer we can generate a seed $\text{PRF}(\text{PRF}(S_{\text{PRF}}, \texttt{toBytes}(i, 32)), \texttt{toBytes}(j, 32))$.

4 Hash Function Address Scheme

The keyed functions F, H and PRF may receive a special 32-byte string ADRS called an address as part of their input data. ADRS encodes the position (with respect to a complex virtual data structure) and purpose of the hash application and is employed to randomize each hash function call, which is called the hash function address scheme. There are three different types of addresses for different use cases, including the OTS hash address, L-tree address, and hash tree address. The structures of different types of addresses are depicted in Figure 1.

layerAddress	(32-bit)
treeAddress	(64-bit)
type = 0	(32-bit)
OTSAddress	(32-bit)
chainAddress	(32-bit)
hashAddress	(32-bit)
KeyAndMask	(32-bit)

(a) OTS address

layerAddress	(32-bit)
treeAddress	(64-bit)
type = 1	(32-bit)
ltreeAddress	(32-bit)
treeHeight	(32-bit)
treeIndex	(32-bit)
KeyAndMask	(32-bit)

(b) L-tree address

layerAddress	(32-bit)
treeAddress	(64-bit)
type = 2	(32-bit)
padding = 0	(32-bit)
treeHeight	(32-bit)
treeIndex	(32-bit)
KeyAndMask	(32-bit)

(c) Hash tree address

Fig. 1: The address scheme for XMSS and MT-XMSS

We first describe the common fields shared by all types of addresses. The layerAddress describes which layer the concerned XMSS tree resides at, starting from zero for the XMSS trees at the bottom layer. Therefore, for XMSS or MT-XMSS with a single layer, layerAddress $= 0$. The treeAddress encodes the position of the concerned XMSS tree within the layerAddress-th layer, starting with zero for the leftmost XMSS tree. addrType indicates the type of the ADRS structure, addrType $= 0$ for an OTS hash address, addrType $= 1$ for an L-tree address, and addrType $= 2$ for a hash tree address. keyAndMask distinguishes different purpose of the hash function application: keyAndMask $= 0$ for generating a key for the keyed functions, keyAndMask $= 1$ and keyAndMask $= 2$ for generating the most significant n-byte mask and the least significant n-byte mask, respectively. When only one n-byte mask is needed, we set keyAndMask $= 1$.

4.1 OTS Hash Address

The structure of an OTS hash address is illustrated in Figure 1. The first two fields layerAddress and treeAddress determine the position of the XMSS tree. OTSAddress describes the position of the leaf node associated with the WOTS+ instance within the XMSS tree. chainAddress and hashAddress determine the position where the hash function is applied within the sequence of hash chains corresponding to the WOTS+ instance.

If the XMSS tree the WOTS+ instance resides at is of height h, then OTSAddress encodes the position of the leaf corresponding to the WOTS+ instance. Therefore, OTSAddress starts from zero for the leftmost node and encodes an integer $i \in \{0, \cdots, 2^h - 1\}$.

Assume that the used WOTS+ scheme has l hash chains and each chain has depth $W - 1$. Then chainAddress describes in which chain the hash application happens, and thus chainAddress encodes an integer in $\{0, \cdots, l - 1\}$. The position within the chain is determined by hashAddress which encodes an integer in $\{0, \cdots, W - 1\}$.

4.2 *L*-Tree Address

The structure of an L-tree address is illustrated in Figure 1. The first two fields `layerAddress` and `treeAddress` determine the position of the `XMSS` tree. `ltreeAddress` identifies the position of the leaf computed with the L-tree, and thus `ltreeAddress` encodes an integer in $\{0, \cdots, 2^h - 1\}$. `treeHeight` describes the height of the two nodes the hash function call applied to. `treeIndex` encodes the index of the parent node which is one level higher than the two nodes the hash function call applied to.

4.3 Hash Tree Address

The structure of a hash tree address is illustrated in Figure 1. The first two fields `layerAddress` and `treeAddress` determine the position of the `XMSS` tree. `padding` is always set to 0 and act as a placeholder. `treeHeight` describes the height of the two nodes the hash function call applied to. `treeIndex` encodes the index of the parent node which is one level higher than the two nodes the hash function call applied to.

5 The `WOTS+` One-Time Signature Scheme

`WOTS+` is a one-time signature scheme, meaning that each private key must be used at most one time to sign any given message. If a private key is used to sign two different messages, the security of the scheme is compromised. Afterwards, an `WOTS+` private and public key pair is referred to as a `WOTS+` instance. Therefore, to generate a `WOTS+` instance is to generate a `WOTS+` key pair. `WOTS+` uses two parameters, including n the number of bytes of the output of the hash function, and the Winternitz parameter $W \in \{4, 16\}$. Note that $w \in \{2, 4\}$ is used to denote $\log_2(W)$ in this article.

5.1 `WOTS+` Key Generation

The private key and public key is generated by computing l hash chains, where $l = l_0 + l_1$,

$$\begin{cases} l_0 = \frac{8n}{\log_2(W)} \\ l_1 = \left\lfloor \frac{\log_2(l_0(W-1))}{\log_2(W)} \right\rfloor + 1 \end{cases},$$

and each node represents an n-byte value. Moreover, each hash chain contains W nodes. The ending node of a chain is computed from the starting node of the chain by iteratively applying $F()$ $W-1$ times.

The secret key contains the l starting nodes $x_{0,0}$, $x_{1,0}$, $\cdots$, $x_{l-1,0}$, and the public key contains the l ending nodes $x_{0,W-1}$, $x_{1,W-1}$, $\cdots$, $x_{l-1,W-1}$. The starting nodes can be selected randomly from the uniform distribution or generated pseudo-randomly from a n-byte secret seed. The ending nodes are computed from the starting nodes such that

$$x_{i,j+1} = F(\texttt{KEY}, x_{i,j} \oplus \texttt{BM}),$$

where $\texttt{KEY} = \mathrm{PRF}(\texttt{SEED}, \texttt{ADRS})$ with $\texttt{ADRS.chainAddress}$ set to $\texttt{toBytes}(i, 32)$, $\texttt{ADRS.hashAddress}$ set to $\texttt{toBytes}(j, 32)$ and $\texttt{ADRS.keyAndMask}$ set to 0, and $\texttt{BM} = \mathrm{PRF}(\texttt{SEED}, \texttt{ADRS})$ with $\texttt{ADRS.chainAddress}$ set to $\texttt{toBytes}(i, 32)$, $\texttt{ADRS.hashAddress}$ set to $\texttt{toBytes}(j, 32)$ and $\texttt{ADRS.keyAndMask}$ set to 1.

5.2 WOTS+ Signature Generation and Verification

A WOTS+ signature is a sequence $(z_0, \cdots, z_{l-1})$ of l n-byte strings. Given a message $M \in \mathbb{B}^n$, the signature is generated as follows. Note that here we only allow WOTS+ to sign n-byte messages. First, we convert M to an array $(M[0], \cdots, M[l_0 - 1])$ of l_0 $w = \log_2(W)$-bit values, $M[i]$ is regarded as an integer in $\{0, \cdots, W - 1\}$. Next, we compute the checksum of $(M[0], \cdots, M[l_0 - 1])$ using Algorithm 1 (see Table 1 for the rotation offsets), and let $\alpha = \texttt{Cksm}_{n,w}(M)$. Then, the signature is

$$(x_{0,M[0]}, \cdots, x_{l-1,M[l_0-1]}, x_{l,\texttt{coef}(\alpha,0,w)}, \cdots, x_{l-1,\texttt{coef}(\alpha,l_2-1,w)}), \qquad (1)$$

which can be computed from the private key $(x_{0,0}, x_{1,0}, \cdots, x_{l-1,0})$ by applying F iteratively.

Algorithm 1: $\texttt{Cksm}_{n,w}(\cdot)$: 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
    $sum \leftarrow sum + (W - 1) - \texttt{coef}(S, i, \log_2(W))$

Return $sum \ll \gamma_{n,w}$
```

Table 1: The left shift offset $\gamma_{n,w}$

n	w	$\gamma_{n,w}$
32	2	6
32	4	4

With a valid signature given in Equation (1), one can compute the correct public key. So the signature is valid if and only if the computed hypothetical public key matches the correct public key.

6 The XMSS Signature Scheme

Basically, the XMSS signature scheme provides a method for organizing a set of 2^h WOTS+ instances in a perfect binary tree with height h such that each leaf node is associated with a WOTS+ instance, and the root node is employed to authenticate the WOTS+ instances. We call this structure an XMSS tree, which corresponds to an XMSS instance. The XMSS instance can sign at most 2^h different messages in its life cycle, and each time a new signature is generated, one WOTS+ instance is consumed. Moreover, these WOTS+ instances are consumed in order from the leftmost leaf to the rightmost leaf. Figure 2 depicts an XMSS tree with height 3.

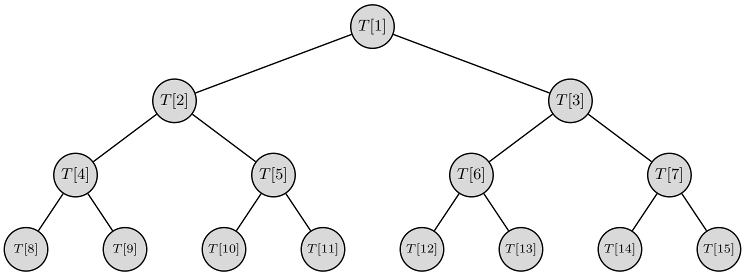

Fig. 2: An XMSS tree with height 3

6.1 The Leaf Nodes of an XMSS Tree

Each leaf node of an XMSS tree is associated with a WOTS+ instance. To be more specific, a leaf node is the root node of a so-called L-tree computed from the public key of a WOTS+ instance.

Let $(y_0, \cdots, y_{l-1}) \in \mathbb{B}^{nl}$ be the public key of a WOTS+ instance. An L-tree is a binary tree whose leaves are $y_0, \cdots, y_{l-1}$. The root of the L-tree can be computed with Algorithm 2, where the subroutine $\texttt{Parent}()$ is given in Algorithm 3. Since it is possible that l is not a power of 2, the L-tree may be unbalanced.

6.2 XMSS Private Key and Public key Generation

The private key of XMSS with height h contains the 2^h private keys of the 2^h WOTS+ instances, which can be generated on the fly from a single n-byte secret seed to save memories according to the method described in Section 3. The public key of the XMSS instance is the root of a perfect binary tree (XMSS tree) with height h, whose leaves are the roots of the 2^h L-trees constructed from the 2^h WOTS+ public keys.

The XMSS tree is constructed as follows. Let a_{2i} and a_{2i+1} be the $2i$-th and $(2i+1)$-th nodes at level k, then their parent node b_i is the i-th node at level $k+1$. We have

$$b_i = \texttt{Parent}(a_{2i}, a_{2i+1}, \texttt{SEED}, \texttt{ADRS}),$$

Algorithm 2: `LTreeRoot`(): Compute the the root node of an L-Tree

```
Input: A WOTS+ public key $y = (y_0, \cdots, y_l)$, a seed SEED, and an address ADRS
Output: An $n$-byte value representing the root of the $L$-tree whose leaves are
        the public key elements

$l' \leftarrow l$
ADRS.treeHeight $\leftarrow 0$
while $l' > 1$ do
    for $0 \leq i < \lfloor l'/2 \rfloor$ do
        ADRS.treeIndex $\leftarrow i$
        $y_i \leftarrow$ Parent($y_{2i}, y_{2i+1}$, SEED, ADRS)
    if $l' \mod 2 = 1$ then
        $y_{\lfloor l'/2 \rfloor} = y_{l'-1}$
    $l' \leftarrow \lceil l'/2 \rceil$
    ADRS.treeHeight $\leftarrow$ ADRS.treeHeight $+ 1$
Return $y_0$
```

Algorithm 3: `Parent`(): Compute the parent node

```
Input: An $n$-byte value $a$ (left node), an $n$-byte value $b$ (right node), a seed
       SEED, and an address ADRS
Output: An $n$-byte value representing the parent node of $a$ and $b$

ADRS.keyAndMask $\leftarrow 0$
KEY $\leftarrow$ PRF(SEED, ADRS)
ADRS.keyAndMask $\leftarrow 1$
BM$_0$ $\leftarrow$ PRF(SEED, ADRS)
ADRS.keyAndMask $\leftarrow 2$
BM$_1$ $\leftarrow$ PRF(SEED, ADRS)
Return $H$(KEY, $(a \oplus \mathtt{BM}_0) \parallel (b \oplus \mathtt{BM}_1)$)
```

where the fields of `ADRS` fulfills the following condition

$$\begin{cases} \mathtt{ADRS.layerAddress} = 0 \\ \mathtt{ADRS.treeAddress} = 0 \\ \mathtt{ADRS.type} = 2 \\ \mathtt{ADRS.treeHeight} = k \\ \mathtt{ADRS.treeIndex} = \mathtt{toBytes}(i, 32) \end{cases}$$

In summary, the secret key is $\mathtt{idx} \| S_{\mathrm{PRF}} \| \mathtt{Root} \| \mathtt{SEED}$, where `idx` is used to index the next `WOTS+` instance to be used to sign a message, and the public key is $\mathtt{Root} \| \mathtt{SEED}$.

6.3 XMSS Signature Generation and Verification

Given a message $M \in \mathbb{F}_2^*$ and an XMSS secret key whose idx field is j. We first compress M into an n-byte string M' such that

$$\begin{cases} r = \mathrm{PRF}(S_{\mathrm{PRF}}, \texttt{toBytes}(j, 32)) \\ M' = H_{\texttt{msg}}(r \| \texttt{Root} \| \texttt{toBytes}(j, n), M) \end{cases} . \tag{2}$$

Then, we use the j-th WOTS+ instance to sign M'. The XMSS signature contains the index j of the used WOTS+ instance, the byte string r (note that r is generated from a secret seed so it must be provided to the verifier), the WOTS+ signature, and the authentication path of the j-th node of the XMSS tree, which consists of the $h - 1$ sibling nodes appearing in the path from the j-th node to the root of the XMSS tree. Before releasing the signature, the idx field of the XMSS secret key must be increment by 1. A XMSS instance signing a message M is illustrated in Figure 3, where the gray nodes forming the authenticated path of the signature.

Given a message and its claimed signature, one can compute the hypothetical public key of the used WOTS+ instance, and then the hypothetical j-th leaf of the XMSS tree. With the help of the claimed authentication path, one can derive a hypothetical root of the XMSS tree. The signature is valid if and only if the hypothetical root matches the public key of the XMSS instance.

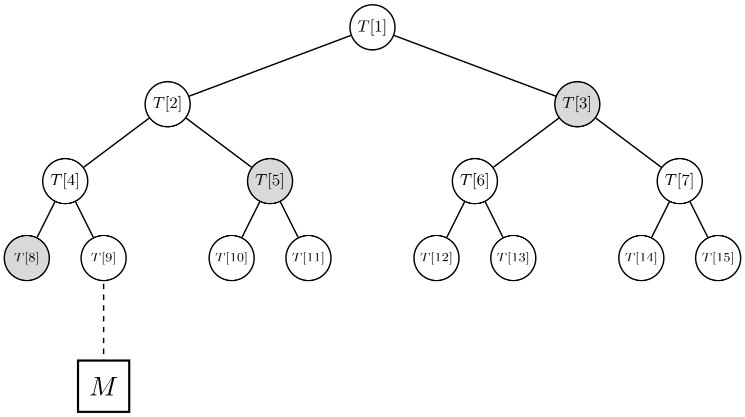

Fig. 3: An XMSS tree with height 3 signing a message M with idx = 1

7 MT-XMSS: The Multi-Tree Version of XMSS

Like XMSS, The MT-XMSS scheme is another method for organizing a large set of WOTS+ instances. In MT-XMSS, WOTS+ instances are associated with the leaves of many XMSS trees placed at different layers. The XMSS trees are "connected" in the sense that the roots of the XMSS trees are signed by the WOTS+ instances associated with the leaves of the XMSS trees in the upper layer. The leaves of the lowest layer XMSS trees are used to sign the messages. We call this structure an MT-XMSS tree or MT-XMSS instance. Note that an MT-XMSS instances with a single layer is essentially an XMSS instance. A 3-layer XMSS tree signing a message M is illustrated in Figure 4, where only the involved XMSS trees are displayed.

In MT-XMSS, we have d layers of XMSS instances, including layer 0 (the bottom layer), $\cdots$, and layer $d-1$ (the top layer). The XMSS instances in the same layer have the same height. If the total height of the MT-XMSS tree is h, then the height of the XMSS trees is h/d.

In the ($d-1$)-th layer (the top layer), there is only 1 XMSS tree. For $0 \leq i < d-1$, there are $2^{(d-1-i)\frac{h}{d}}$ XMSS trees in the i-th layer. Then, there are $2^{(d-1)\frac{h}{d}}$ XMSS trees in the 0-th layer. Therefore, the $2^{(d-1)\frac{h}{d}}$ XMSS trees in the bottom layer have 2^h leaves in total. Since the WOTS+ instances associated with these leaves are used to sign messages, the MT-XMSS instance can sign at most 2^h times, and we call 2^h is the capacity of the MT-XMSS instance. For the convenience of description, these 2^h leaves are indexed from 0 (the leftmost leaf) to $2^h - 1$ (the rightmost leaf).

7.1 MT-XMSS Key Generation

The private key of MT-XMSS contains an n-byte secret seed S_{PRF}, a $\lceil h/8 \rceil$-byte index initialized to 0 to identify the next unused WOTS+ instance on the bottom layer, a n-byte public seed SEED, and the root of the topmost XMSS tree. The public key of MT-XMSS contains the public seed SEED and the root of the topmost XMSS tree. Note that with the private key, all WOTS+ instances over the MT-XMSS tree can be generated pseudo-randomly, and thus all XMSS trees on all layers can be constructed. However, to save memories, we typically generates the involved XMSS trees on the fly during the signature generation process.

7.2 MT-XMSS Signature Generation and Verification

Let the global index of the bottom layer WOTS+ instances be idx. idx can be uniquely expressed as a tuple $(\alpha_{d-1}, \alpha_{d-2}, \cdots, \alpha_1, \alpha_0)$ with $0 \leq \alpha_i < 2^{\hbar}$ such that

$$\mathtt{idx} = \alpha_{d-1}2^{(d-1)\hbar} + \alpha_{d-2}2^{(d-2)\hbar} + \cdots + \alpha_1 2^{\hbar} + \alpha_0, \tag{3}$$

where $\hbar = h/d$. Then, the signing process only involves the single XMSS tree in layer $d-1$, the XMSS tree in layer $d-2$ whose root will be signed by the α_{d-1}-th leaf of the involved XMSS tree in layer $d-1$, the XMSS tree in layer $d-3$ whose root will be signed by the α_{d-2}-th leaf of the involved XMSS tree in layer $d-2$, $\cdots$, and the XMSS tree in layer 0 whose root will be signed by the α_1-th leaf of the involved XMSS tree in layer 1. Finally, the message will be signed by the α_0-th leaf of the involved XMSS tree in layer 0.

The XMSS signature contains the WOTS+ signature of the message and all the signatures of the roots of the XMSS tress involved. To verify a claimed signature, we can compute the hypothetical root of the top layer XMSS tree from the message

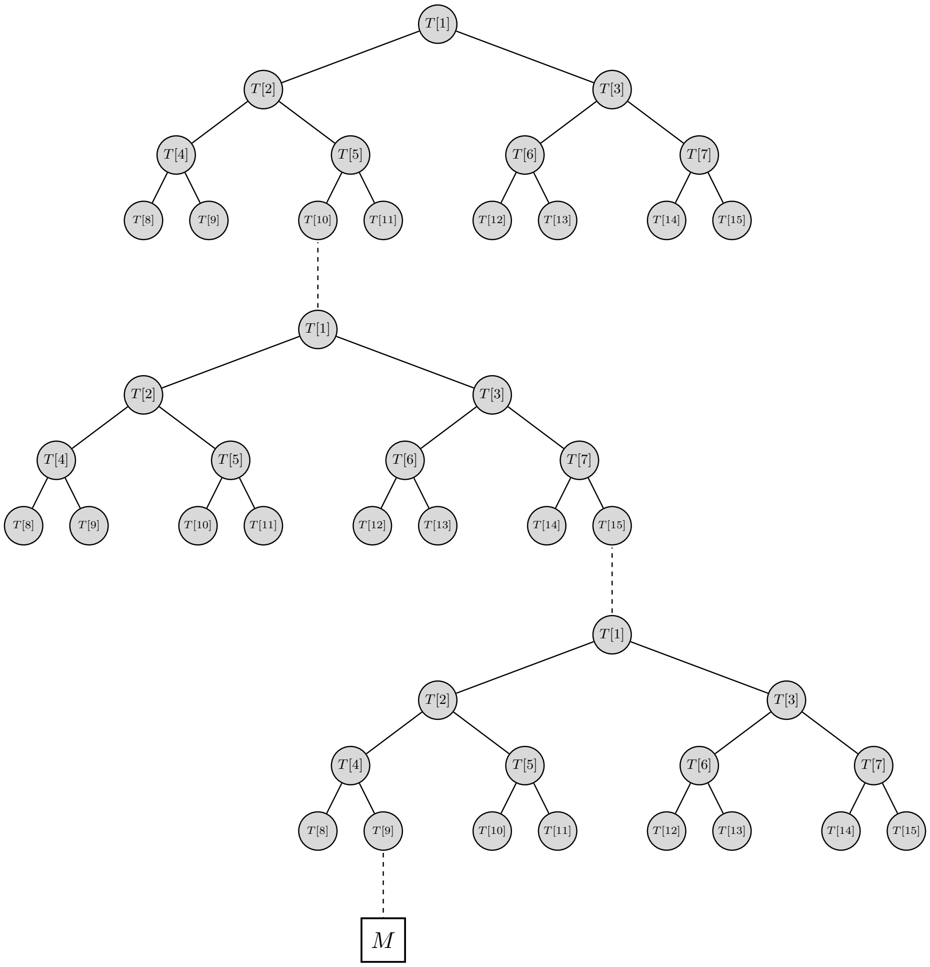

Fig. 4: A 3-layer `MT-XMSS` instance with total height 9

and the signature. The signature is valid if and only if it matches the public key of the `MT-XMSS` instance.

8 Preliminary Implementations and Performance Test

We instantiate the hash-based signature schemes described in previous sections with `SM3`, and we name them as `XMSS-SM3` and `MT-XMSS-SM3`. We implement `XMSS-SM3` and `MT-XMSS-SM3` based on the code provided at https://github.com/XMSS/xmss-reference by substituting the underlying hash function with an implementation of `SM3`. The performance of the implementation is provided in Table 2, which are obtained on a server machine with 32 cores running `Linux ubuntu 18.04` on 2.9GHz `AMD EPYC-Rome` Processor.

In Table 2, the "Height" column records the heights of the XMSS trees in the XMSS-SM3 instances. For MT-XMSS-SM3 instances, the "Height" column records the height of the full MT-XMSS trees and the number of layers. For example, h/d means there are d layers, and the height of the XMSS trees in each layer is h/d.

9 Conclusion

In this work, we instantiate the hash-based signature schemes XMSS and MT-XMSS described in RFC 8391 with SM3 and conduct some preliminary performance test. In the future, we will provide implementations of XMSS-SM3 and MT-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.

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

Table 2: Preliminary performance test of XMSS-SM3 and MT-XMSS-SM3

W	Height	KeyGenTime (s)	PubKeySize (Byte)	SignTime (s)	SigSize (Byte)	VerifyTime (s)	Capacity
16	10	9.758708	68	0.017429	2502	0.00621	2^{10}
	20/2	6.730387	68	0.019128	4965	0.01183	2^{20}
	20/4	0.210187	68	0.029314	9253	0.01949	2^{20}
	40/8	0.213399	68	0.037080	18471	0.03347	2^{40}
	60/12	0.21967	68	0.031442	27690	0.04252	2^{60}

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.

SLG$^+$22. Siwei Sun, Tianyu Liu, Zhi Guan, Yifei He, Jiwu Jing, Lei Hu, Zhenfeng Zhang, and Hailun Yan. LMS-SM3 and HSS-SM3: Instantiating Hash-based Post-Quantum Signature Schemes with SM3. Cryptology ePrint Archive, Paper 2022/1491, 2022. https://eprint.iacr.org/2022/1491.

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.