K-XMSS and K-SPHINCS$^+$: Hash based Signatures with Korean Cryptography Algorithms

Minjoo Sim[1], Siwoo Eum[1], Gyeongju Song[1], HyeokDong Kwon[1], Kyungbae Jang[1], HyunJun Kim[1], HyunJi Kim[1], Yujin Yang[1], Wonwoong Kim[1], Wai-Kong Lee[2][0000−0003−4659−8979], and Hwajeong Seo[1][0000−0003−0069−9061]

[1]IT Department, Hansung University, Seoul (02876), South Korea,
{minjoos9797, shuraatum, thdrudwn98, korlethean, starj1023, khj930704, khj1594012, yujin.yang34, dnjsdndeee, hwajeong84}@gmail.com
[2]Department of Computer Engineering, Gachon University, Seongnam, Incheon (13120), Korea,
waikonglee@gachon.ac.kr

Abstract. Hash-Based Signature (HBS) uses a hash function to construct a digital signature scheme, where its security is guaranteed by the collision resistance of the hash function used. To provide sufficient security in the post-quantum environment, the length of hash should be satisfied. Modern HBS can be classified into stateful and stateless schemes. Two representative stateful and stateless HBS are XMSS and SPHINCS$^+$, respectively. In this paper, we propose two HBS schemes: K-XMSS and K-SPHINCS$^+$, which replace internal hash functions of XMSS and SPHINCS$^+$ with Korean cryptography algorithms. K-XMSS is a stateful signature, while K-SPHINCS$^+$ is its stateless counterpart. We also showcase the reference implementation of K-XMSS and K-SPHINCS$^+$ employing LSH and two hash functions based on block ciphers (i.e. CHAM and LEA) as the internal hash function. The reference code is developed as a proof-of-concept, which can be optimized for better performance using advanced implementation techniques (e.g. AVX2 and NEON).

Keywords: XMSS · SPHINCS$^+$ · Korean Cryptography Algorithms · Hash based Signatures · Software Implementations.

1 Introduction

Recently, interest in the security of existing cryptographic schemes against the rising threat from quantum computers is emerging. Accordingly, quantum-resistant cryptography has received attention by the "Post-Quantum Cryptography Standardization" initiated by the National Institute of Standards and Technology (NIST) [1]. Hash-Based Signature (HBS) [2] schemes guarantee the security

with collision resistance of the hash function used. HBS schemes are signature schemes that rely solely on the existence of a secure one-way function (i.e. hash function). HBS schemes were developed in the 1970s by Lamport [3] and extended by Merkle [4]. As the threat to quantum computers increases, interest in the field is also on the rise. The hash function can respond to the threat of quantum computers by increasing the output length. For this reason, HBS schemes have attracted attention, and XMSS has proven the feasibility of HBS. Recently, stateless SPHINCS [5], a variant of XMSS that does not need to maintain state, has been proposed. For the NIST Post-Quantum Cryptography standardization project, SPHINCS$^+$, an improved version of SPHINCS, has been proposed. SHA2, SHAKE, and HARAKA hash functions were used in the implementation of XMSS and SPHINCS$^+$.

In this paper, we show variants of XMSS and SPHINCS$^+$ (i.e. K XMSS and K-SPHINCS$^+$) by replacing the current hash function setting with Korean cryptography algorithms (i.e. LSH hash function and hash function based on Korean block ciphers). Main contributions of this work are summarized below:

1.1 Contributions

K-XMSS The original XMSS produce HBS through the use SHA2 and SHAKE hash functions. In response, we performed HBS using Korean hash functions (i.e. LSH, CHAM, and LEA). This is why we call K-XMSS. Finally, we evaluate the performance of XMSS and K-XMSS. As a result of the evaluation, among the Korean hash functions, LSH had the best performance.

K-SPHINCS$^+$ The original SPHINCS$^+$ produces HBS using the SHA2, SHAKE, and HARAKA hash functions. In response, we proposed to generate HBS using Korean hash functions (i.e. LSH, CHAM, and LEA). This is why we call the algorithm as K-SPHINCS$^+$. Finally, we evaluate the performance of SPHINCS$^+$ and K-SPHINCS$^+$. As a result of the evaluation, LSH showed the best performance among Korean hash functions in K-SPHINCS$^+$ as in K-XMSS.

Hash Function Based on Korean Block cipher We implemented a hash function using Korean block ciphers. *Tandem DM* scheme was applied to use the Korean block cipher as a hash function. *Tandem DM* can generate a hash value having a length of $2m$-bit by applying a block cipher algorithm using an m-bit block length and a $2m$-bit key length. In this approach, we implemented hash functions using Korean block ciphers by applying LEA and CHAM Korean block ciphers.

The summary of this paper is as follows:

- To the best of our knowledge, this is the first trial to implement Korean version of hash-based cryptography schemes (i.e. K-XMSS and K-SPHINCS$^+$) with Korean hash functions.

– We modify the Korean block ciphers (i.e., CHAM and LEA) into hash functions and employ them on the existing XMSS and SPHINCS$^+$ to construct proposed K-XMSS and K-SPHINCS$^+$ HBS schemes.
– We evaluate the performance of the original XMSS, SPHINCS$^+$, and proposed algorithms (K-XMSS, K-SPHINCS$^+$).

2 Related Works

2.1 eXtended Merkle Signature Scheme(XMSS)

XMSS [6] is a stateful Hash-Based-Signature (HBS) scheme based on the Merkle Signature Scheme (MSS) [7], and uses WOTS$^+$ [8] as the main building block. XMSS uses one key pair (i.e. private key and public key), and the tree height is H. XMSS can generate up-to 2^H signatures, which is illustrated on Figure 1. To ensure the security of XMSS, the used key pair (i.e. WOTS$^+$ key) should not be used again. Definitions of parameters used in XMSS are given in Table 1.

Table 1. Symbols of XMSS parameters.

Symbols	Descriptions
w	Winternitz parameter (w); Power of two; 16 in XMSS.
l	Length in bytes
c	Hash function chain
h	Hash function
C	Checksum
m	Length of binary message
r	Randomization elements; r = (r_1, $\cdots$, r_{w-1})

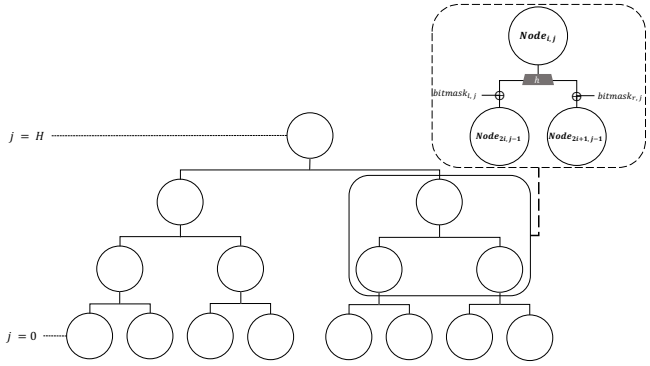

Fig. 1. Tree structure of XMSS; H is height of the tree; Bitmask is chosen uniformly at random from $(b_{l,j}||b_{r,j} \in \{0,1\}^{2n})$.

Winternitz One Time Signature (WOTS) WOTS scheme [7,9] efficiently signs the message digest. The private key is used for signing, where the key should not be used again. In other words, it is infeasible to sign more than one message using a single private key. The signature size of WOTS is smaller than Lamport OTS [3], because the message digest is signed at the same time. WOTS is based on security against collision resistance of one-way hash function.

In the signature of WOTS, w representing the number of bits to be signed, is used as a number of 2 or more. The signature key consists of a randomly selected l-bit string of length n, where l is computed as Equation 1. Equation 1 is calculated based on the selected w.

$$l = l_1 + l_2,\ l_1 = \lceil \frac{m}{\log_2(w)} \rceil,\ l_2 = \lfloor \frac{\log_2(l_1(w-1))}{\log_2(w)} \rfloor + 1 \tag{1}$$

The m-bit message M is based on w, and the checksum C for the message is calculated as Equation 2.

$$C = \sum_{i=1}^{l_1} (w - 1 - M_i) \tag{2}$$

The hash function chain of WOTS is given in Equation 3. Using the signature key l as an input to the function chain, we get the public key of WOTS as a result of Equation 3.

$$c^i(x) = h_k(c^{i-1}(x)) = h_k \circ h_k \circ \cdots \circ h_k \circ h_k,\ x \in \{0,1\}^n,\ c^0(x) = x \tag{3}$$

Winternitz One Time Signature Plus (WOTS$^+$) WOTS$^+$ [8] is a descendent of WOTS scheme. WOTS$^+$ increases the security by adding a random value, r, as shown in the process of applying the one-way function h in the Winternitz one-time signature technique. The function chain of WOTS$^+$ is expressed by the following Equation 4. Unlike WOTS, WOTS$^+$ is based on the security against secondary pre-image resistance.

$$c^i(x) = h_k(c^{i-1}(x) \oplus r_i) = (h_k \oplus r_i) \circ (h_k \oplus r_1) \circ \cdots \circ (h_k \oplus r_{i-2}) \circ (h_k \oplus r_{i-1}) \tag{4}$$

XMSS uses a Merkle hash tree of height h and a binary L-tree of height $\lceil log_2 l \rceil$ to reduce the size of the public key. Two trees are used to reduce 2^H WOTS$^+$ verification keys to one XMSS public key. The overall structure can be found in Figure 2. The public key of WOTS$^+$ obtained using WOTS$^+$ as described above constructs the leaf of the L-tree, which is an unbalanced binary tree. If there is no power of 2 leaves, a node without a right sibling is moved up until it becomes a right sibling of another node. In the case of the L-tree, the same structure as in Figure 1 is used. However, a bitmask different from that of the Merkle tree is used. The upper leaf node of the L-tree created in this way becomes the lower leaf node of the Merkle hash tree.

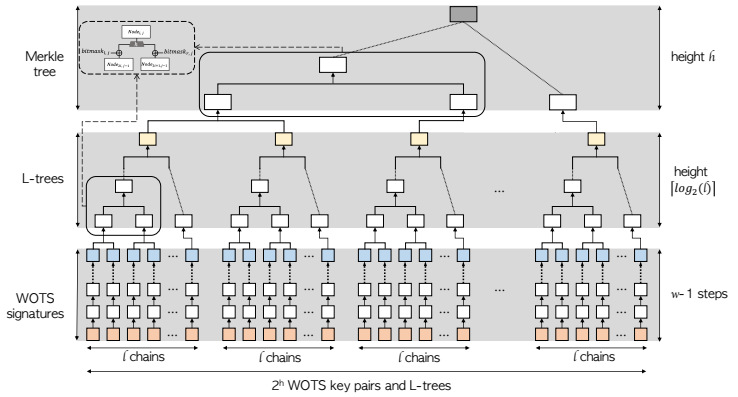

Fig. 2. Overview of L-trees and WOTS$^+$ chains; Orange block indicates WOTS$^+$ signature key, blue block indicates WOTS$^+$ public key, yellow block indicates L-tree root, and gray block indicates XMSS public key [10].

As a result, the root node of the Merkle tree becomes the final XMSS public key. The bit length of this XMSS public key is $2(H + \lceil log_2 l \rceil + 1)n$, the signature length of XMSS is $(l + H)n$, and the private key of XMSS is less than $2n$.

2.2 SPHINCS$^+$

SPHINCS$^+$ [11] is a stateless hash-based signature framework that improves the speed and signature size of SPHINCS [5]. The main contribution of SPHINCS$^+$ is the introduction of FORS (i.e. few-time signature scheme). The second contribution is the method of selecting leaf nodes. SPHINCS$^+$ uses functions with cryptographic properties and each parameter is defined as follows:

- h, d: parameters of Hyper-Tree
- b, k: parameters of FORS
- w: parameter of Winternitz

SPHINCS$^+$ with specific parameters (n=192, h=51, d=17, b=7, k=45, and w=16) showed 25% shorter signatures and 1.7× faster signature routines than those of SPHINCS$^+$. The structure of SHPINCS$^+$ is shown in Figure 3. SPHINCS$^+$ is a hyper-tree of height h and consists of d tree. The height of each tree is h/d, where d is involved in the signature time and the signature size. In a hyper-tree, layer$(d-1)$ has a single tree and layer$(d-2)$ has $2^{h/d}$ trees. The root of the layer$(d-2)$ tree is signed using the WOTS$^+$ key pair in the layer$(d-1)$ tree. Key pairs of Layer 0 WOTS$^+$ are used to sign the FORS public key. Internal values are determined through seed and bitmask, and the entire structure is not computed. For this reason, it is referred to as a "virtual structure". More information on SPHINCS$^+$ can be found in [11].

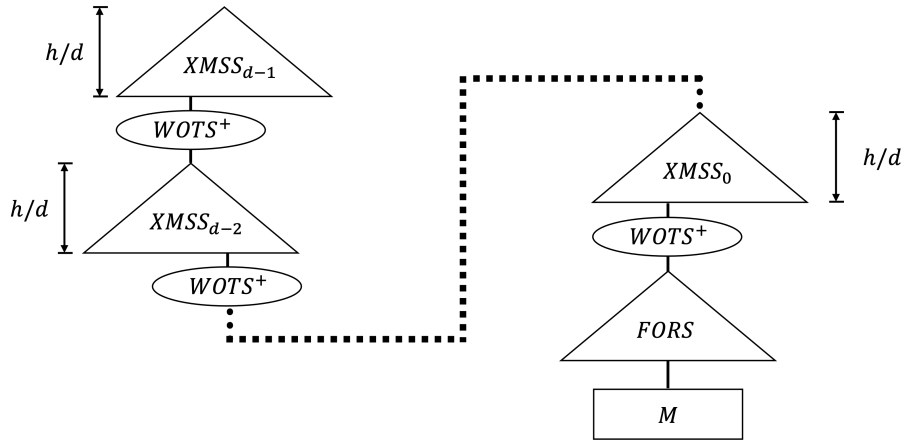

Fig. 3. Overview of SPHINCS$^+$ structure.

FORS: Forest of Random Subsets SPHINCS$^+$ defines and uses FORS, a few-time signature improved from HORST [11]. FORS is used to sign the ka bit string, which is defined as the integer (k, $t = 2^a$). The private key of FORS consists of kt random bit values. It is divided by k set of t values. Overall, it is deterministically derived from `SK.seed` using the pseudo-random function (PRF) and the key address of the hypertree. To obtain the FORS public key, k binary hash trees are constructed on the set of private key elements. Each t value is used as a leaf node and k binary hash trees with height a are created. Figure 4 shows the hash tree of FORS with $k = 6$ and $a = 3$ for message (`100 010 011 001 110 111`). FORS uses H, which is addressed using the location of the FORS key pair and the location of the function call within the tree. With WOTS$^+$, the root node compresses using the Tweakable hash function (Th_k). Th is an efficient function that maps α-bit message M to λ-bit hash value MD using a function key of public parameter P and tweak T, and is expressed as Equation 5. The FORS public key is an n-bit value. The signing process of FORS is as follows. Given a message of ka bits, the k string of a bits is extracted. This bit string k has the index of each single leaf node of FORS. The signature consists of these nodes (indexes) and authentication paths (See Figure 4). The verifier validates the public key by reconstructing the root using the certification path. Since the public key is used as a message, it is implicitly verified with the $WOTS^+$ signature.

$$\begin{aligned} &Th : P \times T \times \{0,1\}^{\alpha} \rightarrow \{0,1\}^{\lambda}, \\ &Th(P,T,M) = H(P||T||M) \end{aligned} \tag{5}$$

XMSS^{MT} XMSS^{MT} [12] is an extension of XMSS. The original XMSS scheme has a disadvantage in key generation. When the height (H) of the tree exceeds

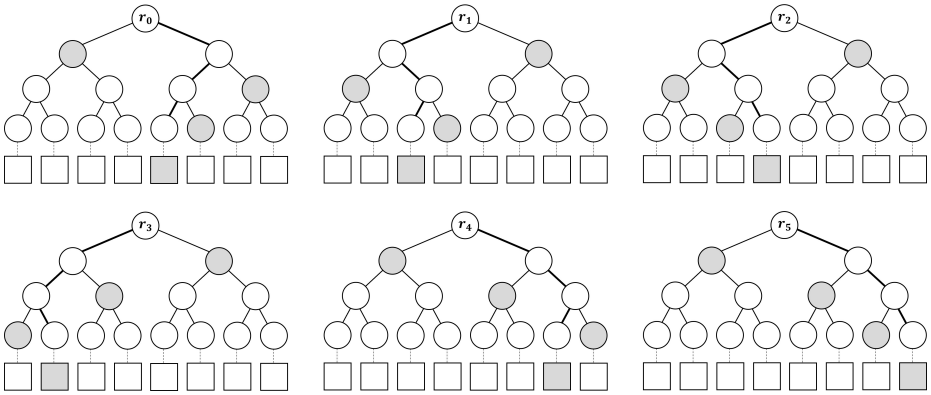

Fig. 4. Hash tree of FORS with $k = 6$ and $a = 3$ for message {100 010 011 001 110 111}.

20, the execution time could be slow. To accelerate the performance, XMSS^{MT} uses a multi-layered XMSS tree called a hyper-tree. A hyper-tree consists of 2 or more XMSS trees, and all XMSS trees have the same height. The tree of the lowest layer of the hyper-tree is used to sign the actual message. The rest of the tree is used to sign the root node of the XMSS tree in each layer.

2.3 Hash Function

The hash function is used by XMSS and SPHINCS$^+$ to construct the signature schemes. In this subsection, we briefly describe the LSH hash function and another hash function based on Korean block cipher [13].

LSH Hash Function LSH is a high-speed hash function developed in Korea that generates a hash value through initialization, compression, and completion processes. The initialization process padding the message and separating the message by the size of the block. The compression process digests the message through message expansion, addition, mixing, and word-by-word circulation functions. The completion process outputs the result of the compression process as a hash value of a specific length. Figure 5 shows the LSH hash function structure.

Hash Function Based on Block cipher An iterated hash function is determined by an easily computable function $h(\cdot,\cdot)$. The function h is called hash round function. The input message is divided into block sizes, and the hash round function calculates the next hash value using the divided message block and the previous hash value. The number of iterations of the hash round function is repeated by the number of message blocks to obtain the final hash value. Equation 6 is a modification of the iterative hash function.

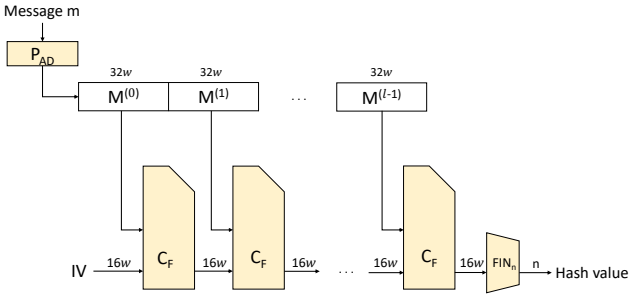

Fig. 5. Overview of LSH hash function.

$$H_i = h(H_{i-1},\ M_i) \qquad\qquad i = 1, 2, ..n. \tag{6}$$

Hash function based on block cipher uses a block cipher algorithm instead of a hash round function [14]. Several structures have been proposed to output the desired length of hash. We utilized the *Tandem DM* structure to implement hash functions using block ciphers. *Tandem DM* structure applies a block cipher algorithm using a key length of $2m$-bit when the block length is m-bit, and the output hash length is $2m$-bit. Figure 6 shows the *Tandem DM* Scheme.

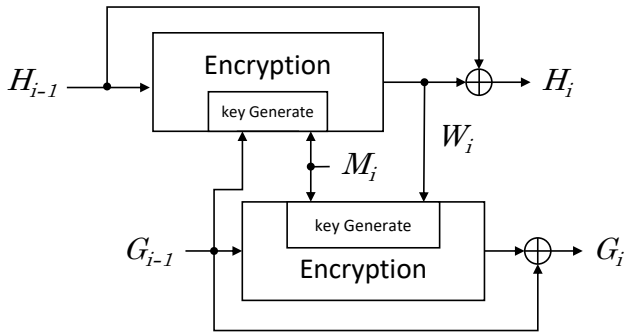

Fig. 6. $2m$-bit hash round function based on m-bit block cipher with $2m$-bit key.

In each iteration, two m-bit values (G_i and H_i) are computed from the previous values H_{i-1} and G_{i-1} and from an m-bit message block M_i as follows:

$$W_i = E_{G_{i-1}, M_i}(H_{i-1}) \tag{7}$$

$$H_i = W_i \oplus H_{i-1} \tag{8}$$

$$G_i = G_{i-1} \oplus E_{M_i, W_i}(G_{i-1}) \tag{9}$$

In this paper, *LEA* and *CHAM* block cipher algorithms were used for the hash round function.

– **LEA Block Cipher** LEA is a lightweight block cipher developed in Korea in 2013 to provide confidentiality not only in high-speed environments (e.g. big data and cloud), but also in lightweight environments, (e.g. IoT devices and mobile devices) [15]. The algorithm structure of LEA uses the ARX structure, and encryption proceeds by dividing the input block into four 32-bit. The ARX structure uses Addition, Rotation, and XOR operations. Figure 7 shows the LEA Block cipher structure. For the usage in the Tandem DM Scheme, we implemented the hash function using the LEA-128-256.

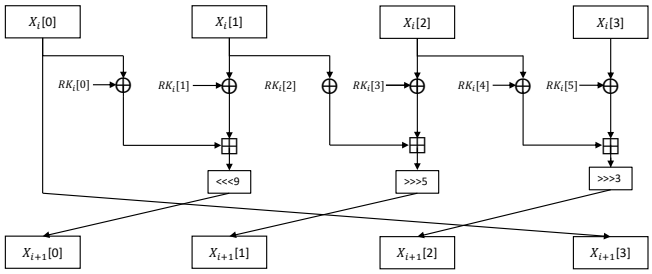

Fig. 7. Overview of LEA block cipher.

– **CHAM Block Cipher** CHAM is a lightweight block cipher announced in ICISC'17 [16]. Subsequently, the revised version of the CHAM Block cipher was announced in ICISC'19 [17]. The revised CHAM differs from the original CHAM only in the number of rounds, and the other specifications are identical. The CHAM has different operations of odd rounds and even rounds. The CHAM of the generalized 4-branch Feistel structure is based on ARX operations. Figure 8 shows the CHAM Block cipher Structure. To apply this to the Tandem DM scheme, we utilized the CHAM-128-256.

3 Proposed Method

3.1 Hash Function Based on Block cipher

In this paper, we construct hash function based on the Tandem DM scheme and utilize LEA and CHAM as the underlying block ciphers. Tandem DM scheme and block ciphers are described in Section 2.3. Algorithm 1 is a description for Figure 6.

The process of Algorithm 1 is as follows. The message received as input is divided into block size to proceed by the number of iterations. The iteration is repeated by the message length divided by the block size. In this paper, the message length is assumed to be a multiple of the block size. Lines 4 and 7

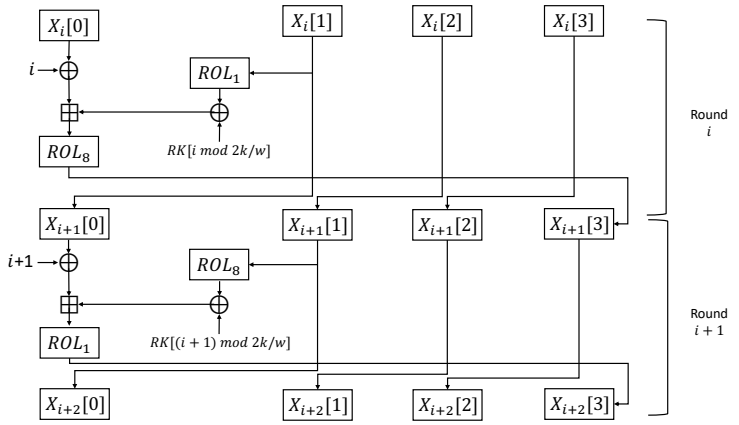

Fig. 8. Overview of CHAM block cipher.

perform key initialization. In line 4, G_i for the upper bit and $M[i]$ for the lower bit are used as a key. In line 7, $M[i]$ for the upper bit and W for the lower bit are used as a key. The initialized key generates a roundkey to be used for encryption through the *Roundkey generate* function. Then, H_i and G_i generate an encrypted value through an *Encryption* function, and finally XOR with W and G_i. If the CHAM and LEA algorithms are applied to the *Roundkey generate* function and *Encryption* function, hash values can be obtained through LEA and CHAM block ciphers.

Algorithm 1 Tandem DM scheme of hash function based on block cipher

Input: M (Message), ML (Message Length)
Output: Hash value

1: `n = Block size`
2: **for** $i = 0$ to ML/n **do**
3: $M[i]$: `Size of Block size`
4: $Key \leftarrow G_i, M[i]$ `(if` G_0`, use a initialization Vector)`
5: $RK \leftarrow RoundKey\ Generate(Key)$
6: $W \leftarrow Encrytion(H_i, RK)$ `(if` H_0`, use a initialization Vector)`

7: $Key \leftarrow M[i], W$
8: $RK \leftarrow RoundKey\ Generate(Key)$
9: $TEMP \leftarrow Encrytion(G_i, RK)$ `(if` G_0`, use a initialization Vector)`

10: $H_{i+1} \leftarrow H_i \oplus W_i$
11: $G_{i+1} \leftarrow G_i \oplus TEMP$
12: **end for**
13: **return** $Hash\ value \leftarrow H, G$

3.2 K-XMSS

In this paper, we replaced the hash functions (i.e. SHA2 and SHAKE) used in the original XMSS to Korean cryptography algorithms (i.e. K-XMSS). In particular, LSH hash function and hash function based on block cipher (CHAM and LEA) are utilized, which are given in Section 2.3. We developed the code based on the basic C reference [1] provided by [18].

Since XMSS has a tree height of h (10, 16, and 20), which determines the number of signatures with one key pair, K-XMSS adopted same parameters and structures utilized in XMSS. LSH provides the value of n only for 256 and 512. The hash function based on CHAM and LEA provides the value of n only for 256. In other words, Korean hash functions do not support the n value of 192, in K-XMSS. K-XMSS is performed on security parameters n of 256 and 512. Since the Winternitz parameter w of the original XMSS is fixed to 16, the value of w is also fixed to 16.

For the $n = 32$ setting, K-XMSS uses Equation 10, 12, and 13 for LSH-256, CHAM, and LEA, respectively. For the $n = 64$ setting, K-XMSS use Equation 11 for LSH-512.

Functions used in K-XMSS are organized as follows:

- **F**: Key encryption hash function; **F** accepts and returns byte strings of length n using keys of length n.
- **H**: Encryption hash function; **H** accepts n-byte keys and byte strings with a length of $2n$ and returns an n-byte string.
- $\mathbf{H}_{msg}$: Encryption hash function; $\mathbf{H}_{msg}$ accepts $3n$-byte keys and byte strings of arbitrary length and returns n-byte strings.
- **PRF**: Pseudo-random function; **PRF** has an n-byte key and a 32-byte index as input and generates pseudo-random value (length n).
- **toByte(x, n)**: n-byte string contains a binary representation of x (in the order of big-endian bytes);

Parameters used in K-XMSS are organized as follows:

- **KEY**: Keys with length in bytes.
- **M**: Strings with length in bytes.

K-XMSS_LSH256 For K-XMSS_LSH256 we define:

$$\begin{aligned} \mathbf{F} &= \mathbf{LSH256}(toByte(0,32)||KEY||M), \\ \mathbf{H} &= \mathbf{LSH256}(toByte(1,32)||KEY||M), \\ \mathbf{H}_{msg} &= \mathbf{LSH256}(toByte(2,32)||KEY||M), \ and \\ \mathbf{PRF} &= \mathbf{LSH256}(toByte(3,32)||KEY||M). \end{aligned} \tag{10}$$

[1] `https://github.com/XMSS/xmss-reference`

K-XMSS_LSH512 For K-XMSS-LSH512 we define:

$$\begin{array}{l} \mathbf{F} = \mathbf{LSH512}(toByte(0,64)||KEY||M), \\ \mathbf{H} = \mathbf{LSH512}(toByte(1,64)||KEY||M), \\ \mathbf{H}_{msg} = \mathbf{LSH512}(toByte(2,64)||KEY||M),\ and \\ \mathbf{PRF} = \mathbf{LSH512}(toByte(3,64)||KEY||M). \end{array} \tag{11}$$

K-XMSS_CHAM For K-XMSS_CHAM256 we define:

$$\begin{array}{l} \mathbf{F} = \mathbf{CHAM}(toByte(0,32)||KEY||M), \\ \mathbf{H} = \mathbf{CHAM}(toByte(1,32)||KEY||M), \\ \mathbf{H}_{msg} = \mathbf{CHAM}(toByte(2,32)||KEY||M),\ and \\ \mathbf{PRF} = \mathbf{CHAM}(toByte(3,32)||KEY||M). \end{array} \tag{12}$$

K-XMSS_LEA For K-XMSS_LEA256 we define:

$$\begin{array}{l} \mathbf{F} = \mathbf{LEA}(toByte(0,32)||KEY||M), \\ \mathbf{H} = \mathbf{LEA}(toByte(1,32)||KEY||M), \\ \mathbf{H}_{msg} = \mathbf{LEA}(toByte(2,32)||KEY||M),\ and \\ \mathbf{PRF} = \mathbf{LEA}(toByte(3,32)||KEY||M). \end{array} \tag{13}$$

3.3 K-SPHINCS$^+$

Similar to XMSS, we changed the hash functions (SHA2, SHAKE, and HARAKA) used in the existing SPHINCS$^+$ to Korean hash functions (LSH, CHAM, and LEA). LSH is a Korean hash function, and CHAM and LEA are utilized for the hash function. Since CHAM and LEA are block ciphers, they were used as hash functions through the method described in Section 3.1. Notations used in K-SPHINCS$^+$ are organized as follows:

Functions used in K-SPHINCS$^+$ are organized:

- $\mathbf{H}_{msg}$: Additional key hash function that can handle messages of arbitrary length.
- **PRF**: Pseudo-random function for generating pseudo-random keys.
- $\mathbf{PRF}_{msg}$: Using PRF to generate randomness for message compression.
- **F**: Second-preimage resistant, undetectable one-way function; $B^n \times B^{32} \times B^n \rightarrow B^n$
- **H**: Second-preimage resistant hash function; $B^n \times B^{32} \times B^{2n} \rightarrow B^n$
- $\mathbf{T}_l$: Weakable hash functions of the form mapping an ln-byte message M to an n-byte hash value md; $B^n \times B^{32} \times B^{ln} \rightarrow B^n$

Parameters used in K-SPHINCS$^+$ are organized:

- **R**: Random values generated based on messages and SK.prf
- **PK.seed**: Public seed which is part of the SPHINCS$^+$ public key.
- **PK.root**: Top root node which is part of the SPHINCS$^+$ public key.

- **ADRS**: 32-byte value representing an address in five defined structures.
- **SK.prf**: As one of the private key elements, the value used to deterministically generate a randomized value for a randomized message hash.
- **Optrand**: Value added when making the value of R optionally non-deterministic.

We set the hash function parameters (i.e. n, h, d, k, w) used in SPHINCS$^+$ to be the same in K-SPHINCS$^+$. LSH is applicable to 256 and 512-bit outputs, and CHAM and LEA are applicable to 256-bit outputs. For this reason, we implement it based on hash function-256.

K-SPHINCS$^+$−LSH256 For K-SPHINCS$^+$−LSH256 we define:

$$\begin{aligned}
&\mathbf{H}_{msg}(R, PK.seed, PK.root, M) = \mathbf{LSH256}(R||PK.seed||PK.root||M, 8m),\\
&\mathbf{PRF}(SEED, ADRS) = \mathbf{LSH256}(SEED||ADRS, 8n),\\
&\mathbf{PRF}_{msg}(SK.prf, Optrand, M) = \mathbf{LSH256}(SK.prf||Optrand||M, 8n),\\
&\\
&\mathbf{F}(PK.seed, ADRS, M_1) = \mathbf{LSHE256}(PK.seed||ADRS||M_1, 8n),\\
&\mathbf{H}(PK.seed, ADRS, M_1||M_2) = \mathbf{LSHE256}(PK.seed||ADRS||M_1||M_2, 8n)\\
&\mathbf{T}_l(PK.seed, ADRS, M) = \mathbf{LSHE256}(PK.seed||ADRS||M, 8n).
\end{aligned} \tag{14}$$

K-SPHINCS$^+$−CHAM256 For K-SPHINCS$^+$−CHAM we define:

$$\begin{aligned}
&\mathbf{H}_{msg}(R, PK.seed, PK.root, M) = \mathbf{CHAM}(R||PK.seed||PK.root||M, m),\\
&\mathbf{PRF}(SEED, ADRS) = \mathbf{CHAM}(SEED||ADRS, n),\\
&\mathbf{PRF}_{msg}(SK.prf, Optrand, M) = \mathbf{CHAM}(SK.prf||Optrand||M, n),\\
&\\
&\mathbf{F}(PK.seed, ADRS, M_1) = \mathbf{CHAM}(PK.seed||ADRS||M_1, 8),\\
&\mathbf{H}(PK.seed, ADRS, M_1||M_2) = \mathbf{CHAM}(PK.seed||ADRS||M_1||M_2, n)\\
&\mathbf{T}_l(PK.seed, ADRS, M) = \mathbf{CHAM}(PK.seed||ADRS||M, n).
\end{aligned} \tag{15}$$

K-SPHINCS$^+$−LEA256 For K-SPHINCS$^+$−LEA we define:

$$\begin{aligned}
&\mathbf{H}_{msg}(R, PK.seed, PK.root, M) = \mathbf{LEA}(R||PK.seed||PK.root||M, m),\\
&\mathbf{PRF}(SEED, ADRS) = \mathbf{LEA}(SEED||ADRS, n),\\
&\mathbf{PRF}_{msg}(SK.prf, Optrand, M) = \mathbf{LEA}(SK.prf||Optrand||M, n),
\end{aligned} \tag{16}$$

$$\begin{aligned}
&\mathbf{F}(PK.seed, ADRS, M_1) = \mathbf{LEA}(PK.seed||ADRS||M_1, 8),\\
&\mathbf{H}(PK.seed, ADRS, M_1||M_2) = \mathbf{LEA}(PK.seed||ADRS||M_1||M_2, n)\\
&\mathbf{T}_l(PK.seed, ADRS, M) = \mathbf{LEA}(PK.seed||ADRS||M, n).
\end{aligned}$$

4 Evaluation

4.1 K-XMSS vs XMSS

XMSS was evaluated using `test/speed.c` included in the basic C reference code provided by [18]. K-XMSS was also evaluated on the same setting by changing

existing hash functions to Korean hash functions. Since K-XMSS performed only for tree height h of 10, which determines the number of messages that can be signed with one key pair. In XMSS, the smallest of the heights is 10. As described in Section 3.2, only 256 and 512 are provided for the security parameter n of the Korean hash functions. Therefore, n of K-XMSS is 256 and 512, the comparison target XMSS was also measured only for n values of 256 and 512.

The performance evaluation of the proposed K-XMSS can be shown in Table 2.

Table 2. K-XMSS evaluation on MacBook Pro (Intel i7-9750H@2.6GHz); GK: Generating Keypair, CS: Creating Signature, VS: Verifying Signature, mid: median, avg: average, Algorithm indicates XMSS-[Hash function]_[h]_[n in bits].

Algorithm	GK		CS		VS	
	[sec]	[10^9cc]	mid [10^6cc]	avg [10^6cc]	mid [10^6cc]	avg [10^6cc]
LSH_10_256	8.28	21.45	31.64	44.78	10.91	11.22
LSH_10_512	17.17	44.52	65.86	93.00	21.69	22.37
CHAM_10_256	47.67	123.60	179.06	256.36	63.39	63.63
LEA_10_256	103.65	268.68	388.48	553.72	154.91	152.95

Among Korean Hash Functions, it was confirmed that LSH was significantly faster than other hash ciphers. The comparison was based on [Hash function]_[10]_[256].

The operation speed of the LSH Generating Keypair is 8.28sec, which is 5.75× faster than CHAM and 18.70× faster than LEA. The clock cycle of the LSH Generating Keypair is $21.45 * 10^9$cc, which is 2.07× less than CHAM and 12.53× less than LEA.

The median clock cycle of LSH signature generation is $31.64 * 10^6$ cc, which is 5.66× less than CHAM and 12.28× less than LEA. The average clock cycle of the LSH generating signature is $44.78 * 10^6$ cc, which is 5.72× less than CHAM and 12.36× less than LEA. As a result, with respect to signature generation, it was confirmed that LSH was about 6 times less than CHAM and about 12 times less than LEA.

The median clock cycle of LSH signature verification is 10.91×10^6 cc, which is 5.81× faster than CHAM and 14.20× less than LEA. The average clock cycle of an LSH verifying a signature is 11.22×10^6 cc. 5.67× faster than CHAM and 13.63× less than LEA. As a result, with respect to signature generation, it was confirmed that LSH was about 6 times less than CHAM and about 14 times less than LEA.

In the case of CHAM and LEA, it is not a separate reference code, but a code we implemented ourselves. LSH used the basic C reference code as it is. Therefore, it is judged that LSH is faster than CHAM and LEA.

Lastly, LEA256 and LEA512 are approximately 2 times as fast as LEA256 for all processes. It was confirmed that if the security parameter n is twice as large, all operation processes are also about twice as slow.

Table 3. Original XMSS evaluation on MacBook Pro (Intel i7-9750H@2.6GHz); GK: Generating Keypair, CS: Creating Signature, VS: Verifying Signature, mid: median, avg: average, Algorithm indicates XMSS-[Hash function]_[h]_[n in bits].

Algorithm	GK		CS		VS	
	[sec]	[10^9cc]	mid [10^6cc]	avg [10^6cc]	mid [10^6cc]	avg [10^6cc]
SHA2_10_256	3.53	9.17	13.54	19.13	4.62	4.63
SHA2_10_512	7.22	18.71	27.47	39.19	9.58	9.79
SHAKE_10_256	1.50	3.89	5.62	8.20	2.16	2.23
SHAKE_10_512	6.19	16.04	28.40	36.00	8.07	8.15

The performance evaluation of proposed XMSS can be shown in Table 3.

In the case of the original XMSS, the same comparison was made based on [Hash function]_[10]_[256]. As a result, it was confirmed that the performance of SHAKE was about 2 times faster than that of SHA2. However, if the value of the security parameter n is 512, the hash function that is twice as slow as the operation process is SHA2. Therefore, we compare the performance of K-XMSS-LSH, which had the best performance among XMSS-SHA2 and K-XMSS. As a result of comparing the speed and clock cycle of the Generating Keypair, it was confirmed that LSH was 2.35× sec slower than SHA2 and 2.34× cc more. In the case of median and average of the clock cycle of Creating Signature, it was confirmed that LSH was 2.34× cc and 2.34× cc more than SHA2, respectively. In the case of median and average of the clock cycle of Verifying Signature, it was confirmed that LSH was 2.36× cc and 2.42× cc more than SHA2, respectively.

As a result, it was confirmed that the LSH performance was about 2 times lower than that of the SHA2 during the entire operation process. SHA2 in original XMSS used the *OpenSSL* library, and in the case of SHAKE, the optimally implemented code for XMSS operates. In contrast, for the hash function in our implementation, K-XMSS, the hash function LSH uses a basic C reference and uses hash function implementations based on block ciphers (CHAM and LEA). In the case of CHAM and LEA, there was no reference code implemented as a hash function using the existing block function. The implementation using the [14] technique was used. The performance of K-XMSS can be further optimized by adopting the optimal implementation code of the Korean hash function.

In this paper, it is confirmed for the first time that a hash function other than the hash function used in XMSS can be applied to XMSS, and the feasibility of the Korean hash function was confirmed. The performance of K-XMSS can be further optimized by adopting the optimal implementation code of the Korean hash function.

4.2 K-SPHINCS$^+$ vs SPHINCS$^+$

We evaluated the performance by replacing hash functions (i.e. SHAKE, SHA, and HARAKA) used in the SPINCS$^+$ with the Korean hash functions (i.e. LEA, CHAM, and LSH). SPHINCS$^+$ was evaluated based on the simple code of PQ-

Clean project [2], and K-SPHINCS$^+$ was evaluated by changing the hash function to a Korean hash function (i.e. LSH, CHAM, and LEA) for the same code. Table 4 and Table 5 show the performance evaluation results of K-SPHINCS$^+$ and SPHINCS$^+$.

Table 4. K-SPHINCS$^+$ evaluation on MacBook Pro (Intel i7-9750H@2.6GHz); GK: Generating Keypair, CS: Creating Signature, VS: Verifying Signature, mid: median, avg: average, Algorithm indicates SPHINCS$^+$-[Hash function]_[n in bits].

Algorithm	GK		CS		VS	
	avg[sec]	mid[10^6cc]	avg [sec]	mid [10^9cc]	avg [sec]	mid [10^6cc]
LSH_256	0.04	108.54	0.88	2.29	0.02	60.88
CHAM_256	0.24	637.79	4.95	12.90	0.13	328.60
LEA_256	0.52	1,341.08	10.59	27.11	0.28	733.32

Among Korean hash functions, it was confirmed that LSH was significantly faster than other hash ciphers. The comparison was based on SPHINCS$^+$[Hash function]_[256].

The average speed of the LSH Generating Keypair is 0.04sec, which is 6.00× faster than CHAM and 13.00× faster than LEA. The median clock cycle of the LSH Generating Keypair is $108.54 * 10^6$cc, which is 5.88× faster than CHAM and 12.36× faster than LEA. As a result, with respect to signature generation, it was confirmed that LSH was about 6 times less than CHAM and about 12 times less than LEA.

The average speed of signature generation of LSH is 0.88sec, which is 5.63× faster than CHAM and 12.03× faster than LEA. The median clock cycle of the LSH Creating Signature is 2.29×10^9cc, which is 5.63× faster than CHAM and 11.84× faster than LEA. As a result, it was confirmed that the LSH creating the signature was about 6 times less than the CHAM and about 12 times less than the LEA.

The average speed of the LSH Verifying Signature is 0.02sec, which is 6.50× faster than CHAM and 14.00× faster than LEA. The median clock cycle of the LSH Verifying Signature is 60.88×10^6cc, which is 5.31× faster than CHAM and 12.05× faster than LEA. As a result, it was confirmed that the LSH verifying the signature was about 6 times less than the CHAM and about 13 times less than the LEA.

As mentioned in XMSS, CHAM and LEA are not separate reference codes, but directly implemented codes. LSH utilized the default C reference code. Therefore, it is judged that LSH is faster than CHAM and LEA for the same reason as XMSS.

The performance evaluation of proposed SPHINCS$^+$ can be shown in Table 5.

In the case of the original SPHINCS$^+$, the same comparison was made based on [Hash function]_[256].

[2] `https://github.com/PQClean/PQClean`

Table 5. Original SPHINCS$^+$ evaluation on MacBook Pro (Intel i7-9750H@2.6GHz); GK: Generating Keypair, CS: Creating Signature, VS: Verifying Signature, mid: median, avg: average, Algorithm indicates SPHINCS$^+$-[Hash function]-256f-simple.

Algorithm	GK		CS		VS	
	avg[sec]	mid[10^6cc]	avg [sec]	mid [10^9cc]	avg [sec]	mid [10^6cc]
SHA256	0.02	44.19	0.35	0.92	0.01	24.74
SHAKE256	0.04	94.63	0.07	1.79	0.02	49.19
HARAKA	0.03	89.10	0.76	2.01	0.02	52.70

As a result, it was confirmed that the performance of SHA2 was about 2 faster than that of SHAKE or HARAKA. Therefore, the performance of SPHINCS$^+$-SHA2 and K-SPHINCS$^+$LSH was compared. As a result of comparing the average speed and median clock cycle of the Generating Keypair, it was confirmed that LSH was 2.00× sec slower than SHA2 and 2.46× cc more than SHA2.

In the case of the average speed and median clock cycle of Creating Signature, it was confirmed that LSH was 2.51× sec slower than SHA2 and 2.49× cc more than SHA2. In the case of median and average speed and median clock cycle of Verifying Signature, it was confirmed that LSH was 2.00× sec slower than SHA2 and 2.46× cc more than SHA2. As a result, it was confirmed that the LSH performance was about 2 times lower than that of SHA2 in the entire operation process.

Original SPHINCS$^+$ used code with optimal implementations of internal hash functions (i.e. SHA2, SHAKE, and HARAKA) for SPHINCS$^+$ to work. In contrast, our implementation used the same Korean hash function as K-XMSS. The performance of K-SPHINCS$^+$ can be further optimized for the same reasons as K-XMSS.

5 Conclusion

We proposed K-XMSS, K-SHPINCS$^+$, which changed the hash functions of XMSS and SHPINCS$^+$ (i.e. SHA2, SHAKE, and HARAKA) to Korean hash functions (i.e. LSH, CHAM, and LEA). In particular, we used Korean block ciphers (i.e. CHAM and LEA) by changing them into hash functions. Finally, we evaluated the proposed K-XMSS and K-SPHINCS$^+$. Internal hash functions used in K-XMSS and K-SPHINCS$^+$ used reference codes from LSH. However, there was no code implemented for hash functions based on block ciphers CHAM and LEA. Therefore, in this paper, we used the CHAM and LEA hash function C code we implemented. As a result of the evaluation, LSH, CHAM, and LEA were not optimized for XMSS and SPHINCS$^+$. Their performance was evaluated to be lower than that of SHA2, SHAKE, and HARAKA. This paper focused on the feasibility of Korean variant of HBS. For that reason, the performance metric is not optimal. In the current version, we utilized the basic C reference version. We believe that this performance can be further optimized by adopting the optimal implementation code (e.g. AVX2 or NEON).

References

1. L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography*, vol. 12. US Department of Commerce, National Institute of Standards and Technology, 2016.
2. J. Buchmann, E. Dahmen, and M. Szydlo, *Hash-based Digital Signature Schemes*, pp. 35–93. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.
3. L. Lamport, "Constructing digital signatures from a one-way function," tech. rep., Citeseer, 1979.
4. R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*, pp. 218–238, Springer, 1989.
5. D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "SPHINCS: practical stateless hash-based signatures," in *Annual international conference on the theory and applications of cryptographic techniques*, pp. 368–397, Springer, 2015.
6. J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS-a practical forward secure signature scheme based on minimal security assumptions," in *International Workshop on Post-Quantum Cryptography*, pp. 117–129, Springer, 2011.
7. R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*, pp. 369–378, Springer, 1987.
8. A. Hülsing, "W-OTS+–shorter signatures for hash-based signature schemes," in *International Conference on Cryptology in Africa*, pp. 173–188, Springer, 2013.
9. S. Even, O. Goldreich, and S. Micali, "On-line/off-line digital signatures," *Journal of Cryptology*, vol. 9, no. 1, pp. 35–67, 1996.
10. W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen, "XMSS and embedded systems," in *International Conference on Selected Areas in Cryptography*, pp. 523–550, Springer, 2019.
11. D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 2129–2146, 2019.
12. A. Hülsing, L. Rausch, and J. Buchmann, "Optimal parameters for XMSS MT," in *International Conference on Availability, Reliability, and Security*, pp. 194–208, Springer, 2013.
13. D.-C. Kim, D. Hong, J.-K. Lee, W.-H. Kim, and D. Kwon, "LSH: a new fast secure hash function family," in *International Conference on Information Security and Cryptology*, pp. 286–313, Springer, 2014.
14. B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: A synthetic approach," in *Annual international cryptology conference*, pp. 368–378, Springer, 1993.
15. D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, "LEA: A 128-bit block cipher for fast encryption on common processors," in *International Workshop on Information Security Applications*, pp. 3–27, Springer, 2013.
16. B. Koo, D. Roh, H. Kim, Y. Jung, D.-G. Lee, and D. Kwon, "CHAM: A family of lightweight block ciphers for resource-constrained devices," in *International Conference on Information Security and Cryptology*, pp. 3–25, Springer, 2017.
17. D. Roh, B. Koo, Y. Jung, I. W. Jeong, D.-G. Lee, D. Kwon, and W.-H. Kim, "Revised version of block cipher CHAM," in *International Conference on Information Security and Cryptology*, pp. 1–19, Springer, 2019.

18. A. Hülsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle signature scheme," in *RFC 8391*, IRTF, 2018.