K2SN-MSS: An Efficient Post-Quantum Signature (Full Version)

Sabyasachi Karati*
School of Computer Sciences
National Institute of Science Education and Research, India
skarati@niser.ac.in

Reihaneh Safavi-Naini
Department of Computer Science
University of Calgary, Canada
rei@ucalgary.ca.

ABSTRACT

With the rapid development of quantum technologies, quantum-safe cryptography has found significant attention. Hash-based signature schemes have been in particular of interest because of (i) the importance of digital signature as the main source of trust on the Internet, (ii) the fact that the security of these signatures relies on existence of one-way functions, which is the minimal assumption for signature schemes, and (iii) they can be efficiently implemented. Basic hash-based signatures are for a single message, but have been extended for signing multiple messages. In this paper we design a Multi-message Signature Scheme (MSS) based on an existing One-Time Signature (OTS) that we refer to as KSN-OTS. KSN uses SWIFFT, an additive homomorphic lattice-based hash function family with provable one-wayness property, as the one-way-function and achieves a short signature. We prove security of our proposed signature scheme in a new strengthened security model (multi-target multi-function) of MSS, determine the system parameters for 512 bit classical (256 bit quantum) security, and compare parameter sizes of our scheme against XMSS, a widely studied hash based MSS that has been a candidate for NIST standardization of post-quantum signature scheme. We give an efficient implementation of our scheme using Intel SIMD (Single Instruction Multiple Data) instruction set. For this, we first implement SWIFFT computation using a SIMD parallelization of Number Theoretic Transform (NTT) of elements of the ring $\mathbb{Z}_p[X]/(X^{\hat{n}} + 1)$, that can support different levels of parallelization. We compare efficiency of this implementation with a comparable (security level) implementation of XMSS and show its superior performance on a number of efficiency parameters.

KEYWORDS

OTS, Merkle Tree, NTT, SWIFFT, Cover-Free Family, SIMD.

Sabyasachi Karati and Reihaneh Safavi-Naini

1 INTRODUCTION

Cryptographic primitive Digital signatures [16] form the basis of establishing trust over the Internet. Today's commonly used signatures are RSA, DSA and ECDSA [14], and use the hardness of Integer Factorization and Discrete Logarithm problems as the basis of security. Shor [37] gave efficient quantum algorithms for both the problems which make these signature schemes insecure against quantum computers. The recent call by security agencies [38] has resulted in efforts by IETF [4], NIST [34] and other organizations to move to post-quantum algorithms, and this has made development of post-quantum secure digital signature of high importance. As the digital signatures form the backbone of trust on the Internet, it is essential to employ quantum-safe signature algorithms to extend this trust into the future. To provide security against quantum computers, digital signature schemes may use computational assumptions such as the hardness of decoding of random codes [11], or finding short vectors in lattices [17], for which no efficient quantum algorithm is known today. Such schemes will remain secure as long as no efficient quantum algorithms for the underlying problems are found. However, these signatures suffer from inefficient computation, large signature length and/or public keys, or unproven security [7].

Quantum-safe digital signature schemes can be based on One-Way-Functions (OWF) which are known to be the minimum assumption for digital signatures [36], and is expected to provide long-term security assurance. Hash-based signature schemes use hash function families as OWF. OTS, introduced by Lamport in 1979 [25], uses a secret and public key-pair to sign a single message. In Lamport's construction, the secret key is a set of random binary strings, called component secret keys, and the public key is obtained by applying a OWF to each component secret key. A message is associated with a subset of component keys and signature will be the set of secret component keys associated with the subset. Security of the scheme depends on the hardness of inverting the OWF. The main advantages of OTS are: (i) it can be built using any OWF, and (ii) the signing and the verification are fast [40]. The sizes of the keys and the signature, however, can be significantly larger than today's digital signatures such as RSA, DSA or ECDSA.

To sign 2^h messages, one needs to use a OTS scheme with 2^h different keys. Using a Merkle tree [29], the public keys of the 2^h OTSs can be hashed into a single short public key. A signature now will include the signature that is generated by a OTS, together with a list of nodes in the Merkle tree which are required to compute the OTS public key to the root of the Merkle tree. A signed message will have an index/address that will indicate the position of OTS public key in the set of leaves of the Merkle tree, that will be used for authentication path verification.

In [13], a generic construction of a multi-message signature from a OTS using Merkle tree is given and the security of MSS is reduced to the security of the OTS and the second pre-image resistance of the hash function. The construction, known as SPR-MSS, is particularly attractive because it relaxes the requirement on the hash function (second pre-image instead of one-wayness and collision-freeness), and has been the basis of widely studied construction of multi-message signature scheme XMSS [9, 19]. XMSS (eXtended Merkle Signature Scheme) uses W-OTS$^+$ [30] as the one-time signature. XMSS is a prime candidate of standardization [33] of quantum-safe secure signature schemes by NIST [34] and IETF [4]. In [21], it was shown that the security model of MSS must be strengthened to realistically capture all the hash values that become available to

*The work was done while the author was a post-doctoral fellow at the Department of Computer Science, University of Calgary, Canada.

the adversary in a MSS setting and can significantly improve their success chance of a forgery. The new security notion is captured by multi-target-multi-function preimage (or second-preimage) property in [21].

In [24], Kalach and Safavi-Naini introduced a hash-based OTS scheme, called as KSN-OTS, that reduces the number of secret key components of Lamport's system [25] by using a 1-CFF (Cover-Free-Family - See Definition 3.1). KSN-OTS has a short signature because of using a family of homomorphic (additive) hash functions called SWIFFT that allows the component secret keys of a message be combined. KSN-OTS security proof reduces unforgeability of the signature to the collision resistance of SWIFFT. KSN-OTS has short signature length, and fast signing and verification (Table 1 of [24]). The main computation step in KSN-OTS is the computation of SWIFFT function that needs $\hat{m}$ multiplications over $\mathcal{R} = \mathbb{Z}_p[X]/(X^{\hat{n}} + 1)$. The fastest method of implementing ring multiplication uses Number Theoretic Transform (NTT) [26] to transform the ring elements to vectors of dimension $\hat{n}$, and allows ring multiplication and addition to become component-wise vector multiplication and addition modulo p.

1.1 Our Contribution

We design K2SN-MSS, a multi-message signature scheme, using an approach that is inspired by SPR-MSS [13], and prove its security using the notion of multi-target, multi-function pre-image and second pre-image resistance. We use SWIFFT as the hash function for KSN-OTS and also the Merkle tree. This has the benefit of using a single optimized code for the whole construction. However it requires addressing a number of challenges, including the need to introduce a new operation, called `Merge`, for constructing the Merkle tree to compensate for the mis-match between parameter sizes of SWIFFT and the SPR-MSS construction. We prove that use of this operation does not affect security of the construction.

We implement the signature scheme using an efficient implementation of SWIFFT which relies on an efficient implementation of NTT. We use SIMD parallelization to achieve low level instructional efficiency. SWIFFT and KSN-OTS are highly parallelizable algorithms that can be further parallelized at higher levels (process level).

To compare our results with XMSS, we will obtain concrete security parameters of K2SN-MSS taking into account security level and parameters of existing implementations of XMSS. Our results show that K2SN-MSS has 3 times faster key generation, signature generation and verification. It however has 5% larger public key and 4 times longer signature.

1.1.1 K2SN-MSS. Our construction uses a modified construction of SPR-MSS in two ways: (i) the Merge operation is used in each tree node to combine the hash output of the lower level, and (ii) different hash keys and random pads are used at each node and so each node is effectively using a different hash function. We prove that with these modifications, the resulting MSS is secure in the multi-function multi-target attack model, and the security of the scheme reduces to the multi-function multi-target preimage and second-preimage resistance of the underlying hash function.

1.1.2 K2SN-MSS implementation. K2SN-MSS consists of three algorithms: key generation, signing and verification algorithms. To generate component secret keys, hash keys and random pads of the Merkle tree, ChaCha20 pseudo-random function family (PRF) is used. ChaCha20 is a state-of-the-art PRF which takes 40-bytes input seed and can generate output of 128 bytes long [6]. We choose parameters of SWIFFT such that it provides 512-bit classical (256-bit quantum) security for K2SN-MSS against existential unforgeability in chosen message attack (EUF-CMA) (See Section 3.3). We implement the following code modules

- Key generation algorithm, that uses Chacha20 as a submodule, and computes the component secret keys, hash keys and the random pads. SWIFFT hash function was used to compute the component public keys, and construct the Merkle tree.
- We implemented the 1-CFF Algorithm [8] to determine the subset of component keys that are associated with a message. This module is called in signing and verification, both.
- The signing and the verification algorithms use ChaCha20, SWIFFT, and the 1-CFF algorithm above.

The time for key generation, signing and verification is dominated by the time required for calculating the hash function SWIFFT. Below we briefly describe our efficient implementation of SWIFFT using SIMD instructions.

1.1.3 Efficient implementation of SWIFFT using SIMD. Ring multiplication over the ring $\mathcal{R}$ dominates the computation cost of SWIFFT function. For the efficient implementation of multiplication in the ring $\mathcal{R}$, we use NTT. We show that the computation of NTT transformation can be parallelized using SIMD, with different levels of parallelization that is parametrized by δ, denoting simultaneous computation of 2^{δ} components of the output vector. The parallelization can be used for $\delta = 0$ (no parallelization) to $\delta = 6$ (full parallelization for the ring $\mathcal{R}$ with $\hat{n} = 64$). The parallelization in [5, 28, 31] is an instance of our general approach with $\delta = 3$. We show the result of SWIFFT function with $\delta = 4$ that leads to 25% higher speed compared to [31]. Our implementations of NTT computation can be with, or without, precomputation. The former approach results in improved speed but the memory requirement grows double-exponentially $O(2^{2^{\delta}})$ in the value of δ. Thus we consider precomputation when the required memory is acceptable.

For efficient verification of KSN-OTS (and also K2SN-MSS), we also implemented gSWIFFT (generalized input SWIFFT) that allows non-binary input strings. This extension was introduced in [24] and is used in our work.

1.1.4 Evaluation of K2SN-MSS. We compare performance of K2SN-MSS with XMSS for 2^h messages, each of length 256 bits, and security parameter n = 512-bit. We use SWIFFT function with parameters $\hat{m} = 16$, $\hat{n} = 64$, $p = 257$ (Sec Section 2.5). As we will discuss in Section 2.2, these parameters allow us to have a fair comparison in terms of functionality and security level, with XMSS. Additionally, they allow us to compare our implementation of SWIFFT with the existing SWIFFT implementation in [28].

It shows that K2SN-MSS has smaller processing time for all algorithms: key generation, signing and verification. The corresponding times for K2SN-MSS are 2.76, 2.89 and 2.65 times faster than XMSS.

On the other hand, the signature length of K2SN-MSS is approximately 4 times larger than that of XMSS. Public and secret key sizes of the two schemes are comparable (almost same). Thus the two schemes will have complementary speed/signature size properties, while enjoying similar security guarantee.

Table 1: Comparison between XMSS and K2SN-MSS scheme at 512-bit classical security level

Name	K2SN-MSS/SWIFFT-16-avx2	XMSS/SHA512/w = 16 [15]
Key Generation (ms)	526069	1452600
Signing (ms)	4.70	13.57
Verification (ms)	0.34	0.90
Signature Size	21331 Bytes	5571 Bytes
Secret Key Size	Seed	Seed
Public Key Size	152 Bytes	144 Bytes

We note that although we implemented KSN-OTS using a single hash function, but the homomorphic property of SWIFFT is only required in the verification of KSN signatures and one can use a different hash function, such as SHA2-512, for the construction of the Merkle tree. Our choice of SWIFFT for Merkle tree is further discussed in Section 7.

2 PRELIMINARIES

We use the following notations and function definitions.

- $n \in \mathbb{N}$: Security parameter where $\mathbb{N}$ denotes the set of natural numbers.
- $m \in \mathbb{N}$: Length of the messages in bits.
- $\hat{m}, \hat{n} \in \mathbb{N}$: Parameters of SWIFFT hash function family.
- p: Prime integer.
- t: The number of component key-pairs for the OTS we employ. We assume $t = 2^{\ell}$ for some positive integer ℓ and $\log_2 \binom{t}{\frac{t}{2}} \geqslant m$.
- h: Merkle tree height. The K2SN-MSS is designed to sign 2^h messages.
- $||$: concatenation of two binary strings.
- $\oplus$: bit-wise XOR of two binary strings.
- $\oplus_p$: component-wise modulo p addition of two vectors of same length.
- $\odot_p$: component-wise modulo p multiplication of two vectors of same length.
- $F_n = \left\{ f_k : \{0,1\}^{2n} \to \{0,1\}^{2n} \mid k \in \{0,1\}^n \right\}$ is pseudo-random function family and will be used in KSN-OTS and K2SN-MSS.
- $\mathcal{F}_n = \{ f_k : \{0,1\}^n \to \{0,1\}^n \mid k \in \{0,1\}^n \}$ is pseudo-random function family and will be used in W-OTS$^+$ and XMSS.
- c_m: A function that, on an input of m-bit, generates a row of the 1-CFF function family table.

2.1 Hash Function

Definition 2.1. **Hash Function Family** [21]. Let $n \in \mathbb{N}$ be the security parameter. Let $n' = n + n_\epsilon \in \text{poly}(n)$ and $k \in \text{poly}(n)$. Then we define a hash function family $\mathcal{H}_n$ as:

$$\mathcal{H}_n = \left\{ H_K : \{0,1\}^{2n} \mapsto \{0,1\}^{n'} \right\}_{K \in_R \{0,1\}^k} .$$

For example, in SHA $n_\epsilon = 0$. On the other hand, in SWIFFT with parameters $\hat{m} = 16$, $\hat{n} = 64$ and $p = 257$, $n_\epsilon = 64$.

Definition 2.2. **Preimage Resistance (PR)** [21]. This is the standard notion of preimage resistance where we define the success probability of an adversary $\mathcal{B}$ against preimage resistance of a hash function family $\mathcal{H}_n$ as:

$$\begin{aligned} \text{Succ}^{\text{pr}}_{\mathcal{H}}(\mathcal{B}) \quad = \quad & \Pr[K \xleftarrow{\$} \{0,1\}^{n'}, x \xleftarrow{\$} \{0,1\}^{2n}, \\ & y = H_K(x); \\ & x' \xleftarrow{\$} \mathcal{B}(K, y) : y = H_K(x')]. \end{aligned}$$

Definition 2.3. **Second-Preimage Resistance (PR)** [21]. This is the standard notion of second-preimage resistance where we define the success probability of an adversary $\mathcal{B}$ against preimage resistant of a hash function family $\mathcal{H}_n$ as:

$$\begin{aligned} \text{Succ}^{\text{spr}}_{\mathcal{H}}(\mathcal{B}) \quad = \quad & \Pr[K \xleftarrow{\$} \{0,1\}^{n'}, x \xleftarrow{\$} \{0,1\}^{2n}; \\ & x' \xleftarrow{\$} \mathcal{B}(K, y) : x' \neq x \wedge H_K(x) = H_K(x')]. \end{aligned}$$

Definition 2.4. **Multi-function, Multi-target Preimage Resistance (MM-PR)** [21]. We define the success probability of an adversary $\mathcal{B}$ against multi-function multi-target preimage resistant of a hash function family $\mathcal{H}_n$ which is parameterized by the number of targets as:

$$\begin{aligned} \text{Succ}^{\text{mm-pr}}_{\mathcal{H},q'}(\mathcal{B}) \quad = \quad & \Pr[K_i \xleftarrow{\$} \{0,1\}^{n'}, x_i \xleftarrow{\$} \{0,1\}^{2n}, \\ & y_i = H_{K_i}(x_i), 0 < i \leqslant q'; \\ & (j, x') \xleftarrow{\$} \mathcal{B}((K_1, y_1), \ldots, (K_{q'}, y_{q'})) : \\ & y_j = H_{K_j}(x')]. \end{aligned}$$

Definition 2.5. **Multi-function, Multi-target Second-Preimage Resistance (MM-SPR)** [21]. We define the success probability of an adversary $\mathcal{B}$ against multi-function multi-target second-preimage resistant of a hash function family $\mathcal{H}_n$ which is parameterized by the number of targets as:

$$\begin{aligned} \text{Succ}^{\text{mm-spr}}_{\mathcal{H},q'}(\mathcal{B}) \quad = \quad & \Pr[K_i \xleftarrow{\$} \{0,1\}^{n'}, x_i \xleftarrow{\$} \{0,1\}^{2n}, \\ & 0 < i \leqslant q'; \\ & (j, x') \xleftarrow{\$} \mathcal{B}((K_1, x_1), \ldots, (K_{q'}, x_{q'})) : \\ & x' \neq x_j \wedge H_{K_j}(x_j) = H_{K_j}(x')]. \end{aligned}$$

Definition 2.6. $\text{InSec}^{\text{MM-SPR}}(\mathcal{H}, q, q')$: This is defined as the maximum success probability for any attacker $\mathcal{B}$ against the multi-function multi-target second-preimage resistant of $\mathcal{H}_n$ with q' targets after q queries as

$$\text{InSec}^{\text{MM-SPR}}(\mathcal{H}_n, q, q') = \max_{\mathcal{B}} \left\{ \text{Succ}^{\text{mm-spr}}_{\mathcal{H}_n,q'}(\mathcal{B}) \right\} .$$

For more details we refer the reader to [21].

2.2 XMSS and W-OTS$^+$

XMSS [9] is a hash-based signature scheme that extends the one time signature W-OTS$^+$ to a multi-message signature scheme using Merkle tree construction. The construction was proposed in [9] and the security was reduced to the second-preimage resistance of the hash function that was used in the Merkle tree and the security of the W-OTS$^+$. XMSS^{MT} [20] extends XMSS to multiple layer tree where each layer root node is signed using W-OTS$^+$. It *allows for a virtually unlimited number of signatures, while it preserves its desirable properties* [20] of XMSS. This provides signature and key generation computation time and signature size trade-off which allow the users to choose parameters to match their requirements. XMSS is a special case of XMSS^{MT}. In [21], XMSS is proposed that provides security against multi-function-multi-target preimage (or second preimage) attacks. The new security requirement for the signature schemes is given in Appendix 2.1 and is used in our security analysis. There are several implementations for XMSS [3, 15]. We compare our implementation with the implementation in [15] because this takes the multi-function-multi-attack notion of security into consideration. Also, this is the only implementation which takes advantage of SIMD parallelization and reports the fastest result. On the other hand, choice of the parameters for SWIFFT hash function leads to 512-bit classical (256-bit quantum) security of K2SN-MSS in multi-function-multi-target model and it should be noted that there is no existing software of XMSS^{MT} at the same security level.

2.3 W-OTS$^+$

XMSS signature scheme uses W-OTS$^+$ [9] as the OTS scheme. The public key and signature of W-OTS$^+$ is computed through random walks in a function family $\mathcal{F}_n = \{f_{\mathrm{k}} : \{0,1\}^n \mapsto \{0,1\}^n \mid \mathrm{k} \in \{0,1\}^n\}$. For a $\mathrm{k}, \mathbf{x} \in \{0,1\}^n$ and $e \in \mathbb{N}$, the random walk is defined as:

$$f^0_{\mathrm{k}}(\mathbf{x}) = \mathrm{k}, \quad f^i_{\mathrm{k}}(\mathbf{x}) = f_{\mathrm{k'}}(\mathbf{x}), \quad \text{where } \mathrm{k'} = f^{i-1}_{\mathrm{k}}(\mathbf{x}).$$

For an m-bit message, and *Winternitz parameter* w, define following parameters

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

The secret key of W-OTS$^+$ is $\mathcal{SK} = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}\}$ where each $\mathbf{x}_i$ is a binary string of length n (that is generated by using a pseudo-random function on a random seed), and the corresponding public key is $\mathcal{PK} = \{\mathbf{x}, f^{w-1}_{\mathbf{x}_0}(\mathbf{x}), f^{w-1}_{\mathbf{x}_1}(\mathbf{x}), \ldots, f^{w-1}_{\mathbf{x}_{l-1}}(\mathbf{x})\}$, where $\mathbf{x}$ is a randomly generated element from $\{0,1\}^n$.

To sign a message mes, first the message is written in base w as, mes $= \{b_0, b_1, \ldots, b_{l_1-1}\}$. Next a checksum $C = \sum_{i=0}^{l_1-1}(w-1-b_i)$ in base w is calculated, $C = \{b_{l_1}, b_{l_1+1}, \ldots, b_{l-1}\}$, and appended to the message to obtain $b = \{b_0, b_1, \ldots, b_{l-1}\}$.

The signature of b is $\sigma_{\text{mes}} = \{\sigma_0, \sigma_1, \ldots, \sigma_{l-1}\} = \{f^{b_0}_{\mathbf{x}_0}(\mathbf{x}), f^{b_1}_{\mathbf{x}_1}(\mathbf{x}), \ldots, f^{b_{l-1}}_{\mathbf{x}_{l-1}}(\mathbf{x})\}$, resulting in an $l \times n$-bit signature. Verification is done by checking

$$\mathcal{PK} \stackrel{?}{=} \{f^{w-1-b_0}_{\sigma_0}(\mathbf{x}), f^{w-1-b_1}_{\sigma_1}(\mathbf{x}), \ldots, f^{w-1-b_{l-1}}_{\sigma_{l-1}}(\mathbf{x})\}.$$

2.4 XMSS: A Secure Merkle Tree MSS Construction [21]

In this section we describe XMSS briefly which is an instantiation of SPR-MSS with W-OTS$^+$ and also takes the multi-function-multi-target security model of hash functions into consideration. Let W-OTS$^+$ be a secure one-time signature scheme with 2^ℓ component secret keys (and the corresponding component public keys). To design an MSS for 2^h messages, 2^h instances of the W-OTS$^+$ will be used as follows. Let the secret key $\mathcal{SK}_i$ and public key $\mathcal{PK}_i$ of the i-th instance of the W-OTS$^+$ be,

$$\begin{aligned} \mathcal{SK}_i &= \{\mathbf{x}_{i_0}, \mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_{2^\ell-1}}\};\ \mathbf{x}_{i_j} \in \{0,1\}^n, \text{ and} \\ \mathcal{PK}_i &= \{\mathbf{x}_i, \mathbf{y}_{i_0}, \mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_{2^\ell-1}}\}; \mathbf{x}_i, \mathbf{y}_{i_j} \in \{0,1\}^n. \end{aligned}$$

To sign the i-th message, the i-th instance of the W-OTS$^+$ is used. The construction uses a Merkle tree of height $h + \ell$, and attaches a random binary string of length $2n$ for each node of the tree, denoted by $\mathbf{v}_{i,j}$ where (i,j) denotes the j-th node from left at height i as $0 \leqslant i \leqslant h + \ell$ and $0 \leqslant j \leqslant 2^i - 1$. In our construction, we consider that leaf nodes are at height $h + \ell$ and the root node is at the height 0. Similarly each node computation uses a different hash key $k_{i,j}$ and random pad $\mathbf{v}_{i,j}$. The construction of the Merkle tree consists of two types of trees: (i) $\mathcal{L}_i$ trees and (ii) $\mathcal{MSS}$ tree.

(1) Construction of $\mathcal{L}_i$ trees: $\mathcal{L}_i$ is a Merkle tree of height ℓ whose leaves are the component public keys of the i-th instance of W-OTS$^+$, that is $\{\mathbf{x}_i, \mathbf{y}_{i_0}, \mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_{2^\ell-1}}\}$. Let $\mathbf{y}_{i_{l,j}}$ be a node of the tree where it is the j-th $(0 \leqslant j \leqslant 2^l - 1)$ node from the left at height l $(0 \leqslant l \leqslant \ell)$ of the $\mathcal{L}_i$. Now we assign $\mathbf{y}_{i_{\ell,j}} = \mathbf{y}_{i_j}$ for $0 \leqslant j < 2^\ell$. The nodes $\mathbf{y}_{i_{l,j}}$ for $0 \leqslant l < \ell$ are computed as

$$\mathbf{y}_{i_{l,j}} = H_{k_{i_{l,j}}}((\mathbf{y}_{i_{l+1,2j}} \| \mathbf{y}_{i_{l+1,2j+1}}) \oplus \mathbf{v}_{i_{l,j}}), \quad (2)$$

where $H_k \in \mathcal{H}_n$, $\|$ and $\oplus$ denote the concatenation and bitwise XOR of two binary strings, respectively. Let the root of the Merkle tree $\mathcal{L}_i$ be $\mathbf{y}_{h,i}$ and then $\mathbf{y}_{h,i} = \mathbf{y}_{i_{0,0}}$.

(2) Construction of $\mathcal{MSS}$ tree: $\mathcal{MSS}$ has height h and the leaf nodes are the root nodes of the $\mathcal{L}_i$ trees, that is $\{\mathbf{y}_{h,0}, \mathbf{y}_{h,1}, \ldots, \mathbf{y}_{h,2^h-1}\}$. The intermediate nodes and the root node of the $\mathcal{MSS}$ are constructed in the similar way shown in equation (2). Let the root of the $\mathcal{MSS}$ tree be $\mathbf{y}_{0,0}$.

The pictorial view of the Merkle tree is given in the Figure 1 and Figure 2.

All the random pads and all the hash keys can be generated using a seed and pseudo-random number generator (PRG). From the seed and the PRG, we can compute the random pad and the hash key for a particular node using the index of the node in the tree. Let the seed for hash key be `hkseed` and the seed of random pads be `rpseed`. Also all the secret keys can be generated using the PRG using another seed `sk`. The multi-message signature scheme has secret and public keys given by, $\mathcal{SK} = \{\mathtt{sk}\}$ and $\mathcal{PK} = \{\mathbf{y}_{0,0}, \mathtt{hkseed}, \mathtt{rpseed}\}$, respectively. A signature consists of, (i) the index of the message, i, (ii) the signature of the i-th W-OTS$^+$ on the message, (iii) the public key $\mathcal{PK}_i$ of the i-th W-OTS$^+$, and (iv) a list of nodes of $\mathcal{MSS}$ called *authentication path* which authenticates the root of the $\mathcal{L}$ tree that corresponds to $\mathcal{PK}_i$, against the root of the $\mathcal{MSS}$ tree, $\mathbf{y}_{0,0}$. As the public key of W-OTS$^+$ can be computed using the random walks in the function family $\mathcal{F}_n$ from the W-OTS$^+$ signature, the XMSS signature does not include the aforementioned third component $\mathcal{PK}_i$ of i-th instance of W-OTS$^+$. K2SN-MSS uses this structure of Merkle, and employ a pseudo-random generator together with a random seed to generate the secret key components. Hash keys and random pads are also generated in a similar way.

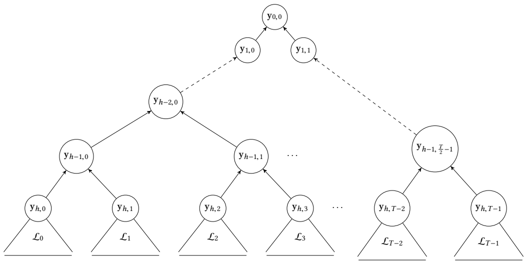

Figure 1: Merkle Tree

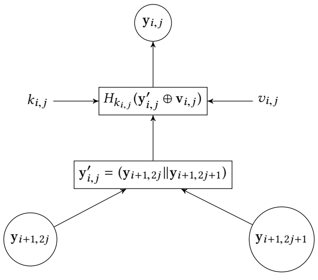
$y_{i,j}$
$k_{i,j}$
$H_{k_{i,j}}(y'_{i,j} \oplus v_{i,j})$
$v_{i,j}$
$y'_{i,j} = (y_{i+1,2j} \| y_{i+1,2j+1})$
$y_{i+1,2j}$
$y_{i+1,2j+1}$

Figure 2: Intermediate node computation

2.5 SWIFFT: An Efficient Lattice-Based Hash-Function

The SWIFFT family of hash functions [28] is defined by three parameters: (i) $\hat{n}$: a power of 2, (ii) $\hat{m}$: a small integer, and (iii) p: a prime (the construction works for non-prime also).

Let $\mathcal{R}$ be a ring as defined in the equation (3):

$$\mathcal{R} = \mathbb{Z}_p[X]/(X^{\hat{n}} + 1). \tag{3}$$

A SWIFFT function of a given SWIFFT family is defined by $\hat{m}$ elements, $\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{\hat{m}-1} \in \mathcal{R}$ called *multipliers*. The input of a SWIFFT function is a binary string of length $\hat{m}\hat{n}$: the input is divided into $\hat{m}$ $\hat{n}$-bit string, and each string is represented as an element of $\mathcal{R}$ with binary coefficients. Let $\mathbf{z} = (\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_{\hat{m}-1})$ be a SWIFFT input, where $\mathbf{z}_i$ is a ring element with binary coefficients. Then hash of $\mathbf{z}$ is computed as,

$$\text{SWIFFT}(\mathbf{z}) = \sum_{i=0}^{\hat{m}-1} (\mathbf{a}_i \cdot \mathbf{z}_i), \tag{4}$$

where the multiplications and the additions are in the ring $\mathcal{R}$. For computational efficiency, the ring multiplications are performed using NTT and so SWIFFT is written as equation (5):

$$\text{SWIFFT}(\mathbf{z}) = \sum_{i=0}^{\hat{m}-1} \Big(\text{NTT}(\mathbf{a}_i) \odot_p \text{NTT}(\mathbf{z}_i)\Big). \tag{5}$$

Notice that we do not apply iNTT on the output of equation 5 to get back the same result as of equation 4. As FFT is linear and bijective, there is no need to apply iNTT on the output of equation 5 as suggested in [28].

Security. SWIFFT is an efficient instantiation of the generalized compact knapsack [32] proposed in [28]. It has been proved [32] that the generalized compact knapsack is asymptotically one-way, and this implies that the SWIFFT family also enjoys the same property. It has been also proved [27, 35], that SWIFFT is asymptotically collision resistant. In [28], it is proved that the security of a SWIFFT function reduces to the security of the subset-sum problem.

2.5.1 gSWIFFT. We use a generalized version of SWIFFT function that takes input from $\mathbb{Z}_p^{\hat{m}\hat{n}}$ (instead of $\{0,1\}^{\hat{m}\hat{n}}$). Let the input to gSWIFFT be $\mathbf{z}' = (\mathbf{z}'_0, \mathbf{z}'_1, \ldots, \mathbf{z}'_{\hat{m}-1})$ where $\mathbf{z}'_i \in \mathbb{Z}_p^{\hat{n}}$. The input length of gSWIFFT is $\hat{m}\hat{n}\lceil\log_2(p)\rceil$ bits. Similar to SWIFFT, we define gSWIFFT function as:

$$g\text{SWIFFT}(\mathbf{z}') = \sum_{i=0}^{\hat{m}-1} (\text{NTT}(\mathbf{a}_i) \odot_p \text{NTT}(\mathbf{z}'_i)), \tag{6}$$

gSWIFFT is used for efficient verification of KSN-OTS scheme (Section 3.1).

2.5.2 Parallelizing SWIFFT Implementation using SIMD . Efficient implementation of K2SN-MSS requires efficient implementation of SWIFFT computation that consists of $\hat{m}$ ring multiplications, followed by ring additions. Thus efficient implementation of ring multiplication is essential for the efficient implementation of K2SN-MSS. We show how to parallelize each ring multiplication using SIMD instructions of Intel Intrinsics. The level of parallelization is determined by the parameter δ, which indicates 2^δ output components will be computed simultaneously. In our setting, the parallelization level δ, can be chosen in the range $\delta = 0$ (no parallelization at all) to $\delta = 6$ corresponding to the full parallelization for ring $\mathcal{R}$ with $\hat{n} = 64$.

Intel Intrinsics for SIMD `Intel` introduced SIMD instruction set MultiMedia eXtension (MMX) in their processors in 1997. MMX has been later extended to Streaming SIMD Extensions (`sse`) like `sse2`, `sse3` upto `sse4.2`. We use Advanced Vector Extensions (`avx`) and in particular `avx2` intrinsic. For more details see [18]. The achievable level of parallelization will depend on the specific set of instructions that will be chosen for the implementation. For 16 fold parallelization ($\delta = 4$), we use 16 16-bit multiplication SIMD instruction together with 16-bit addition, subtraction and other operations such as shift. More details are in Section 5.

3 K2SN-MSS CONSTRUCTION

We first give an overview of KSN-OTS [24] and then describe K2SN-MSS, and prove its security in using the stronger notion of multi-function multi-target security.

3.1 KSN: An Efficient Post-Quantum OTS

We propose an MSS based on KSN [24], an OTS scheme with two building blocks: a cover-free-family and SWIFFT hash function family.

A *w-Cover-Free Family (w-CFF)* is a set system defined as follows.

Definition 3.1. Let $(\mathcal{X}, \mathcal{B})$ be a set system, where $\mathcal{X} = \{x_1, x_2, \ldots, x_t\}$ is a set of t elements and $\mathcal{B} = \{B_1, B_2, \ldots, B_e\}$ be a set of e subsets of $\mathcal{X}$ and the size of each B_i is k. If for all $\Delta \subset \{1, 2, \ldots, e\}$ with $|\Delta| = w$ and for all $i \notin \Delta$, we have $\left|B_i \setminus \bigcup_{j\in\Delta} B_j\right| \geqslant 1$ then we call that the set system is a k-uniform w-Cover-free family with t elements and e subsets, in short w-CFF(t, e, k).

The SWIFFT hash function family is additive homomorphic and for two binary inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ of length $\hat{m}\hat{n}$, satisfying

$$\text{SWIFFT}(\mathbf{x}_1) + \text{SWIFFT}(\mathbf{x}_2) = g\text{SWIFFT}(\mathbf{x}_1 \oplus_p \mathbf{x}_2),$$

where addition $\text{SWIFFT}(\mathbf{x}_1)+\text{SWIFFT}(\mathbf{x}_2)$ is in ring $\mathcal{R}$, and addition $\mathbf{x}_1 \oplus_p \mathbf{x}_2$ is a component-wise modulo p addition of two vectors of length $\hat{m}\hat{n}$. This property is used in KSN signature scheme [24] to achieve fast signing and verification, and short signature. An overview of the scheme is given below. We use a pseudo-random function family $F_n : \{f_k : \{0,1\}^{2n} \mapsto \{0,1\}^{2n} \mid k \in \{0,1\}^n\}$ to generate component secret keys (instead of uniform selection from $\{0,1\}^{2n}$).

Consider a message space consisting of m-bit strings. The secret key and the public key of KSN-OTS scheme consists of t *component keys*, where t is an even integer satisfying $\log_2 \binom{t}{\frac{t}{2}} \geqslant m$. A component secret is a $2n$ bit random string and so the total secret key length is $2tn$ bits. The component secret keys in practice (and in our implementation) are generated by applying $f \in F_n$ on an initial seed. Thus the secret key can be considered as the seed of the PRG. The hash key k of SWIFFT is $\hat{m}$ multipliers which are ring elements and are chosen uniformly at random from $\{0,1\}^{\hat{n}\lceil \log_2(p) \rceil}$.

The signature scheme consists of three algorithms: `KeyGen`, `Sign` and `Verify`. The parameters of SWIFFT function ($\hat{m}$, $\hat{n}$ and p) and the pseudo-random function family F_n will be chosen to achieve the required security level. A function $f_{\text{sk}}()$ is chosen from F_n, and sk is kept secret (this can be seen as the seed of the pseudo-random function).

Key Generation. The `KeyGen` generates the secret key $\mathcal{SK}$ and the public key $\mathcal{PK}$ as given in Algorithm 1. The i-th component secret key is computed as $f_{\text{sk}}(i+1)$. This is the same as the secret key generation in XMSS [9].

Algorithm 1 Key Generation (`KeyGen`(1^n))

```
(1) Choose sk as a seed uniformly at random from $\{0,1\}^n$.
(2) For $i = 0, 1, \ldots, t-1$, compute $\mathbf{x}_i$ as $\mathbf{x}_i = f_{\text{sk}}(i+1)$.
(3) For $i = 0, 1, \ldots, t-1$, compute $\mathbf{y}_i$ as $\mathbf{y}_i = \text{SWIFFT}_{\text{k}}(\mathbf{x}_i)$.
(4) Return Secret Key $\mathcal{SK}$ = sk and Public Key $\mathcal{PK} = \{\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{t-1}\}$.
```

Signature Generation. The `Sign` takes an m-bit message mes and computes the signature σ_{mes} using Algorithm 2.

Algorithm 2 Signature Generation (`Sign(mes,`$\mathcal{SK}$`)`)

```
(1) For $i = 0, 1, \ldots, t-1$, compute $\mathbf{x}_i$ as $\mathbf{x}_i = f_{\mathcal{SK}}(i+1)$.
(2) Compute $B_{\text{mes}} = c_m(\text{mes})$, where $B_{\text{mes}} = \{i_0, i_1, \ldots, i_{\frac{t}{2}-1}\}$,
    and $i_j < i_{j+1}$ for all $0 \leqslant j \leqslant \frac{t}{2} - 2$.
(3) Compute $\sigma_{\text{mes}} = \mathbf{x}_{i_0} \oplus_p \mathbf{x}_{i_1} \oplus_p \cdots \oplus_p \mathbf{x}_{i_{\frac{t}{2}-1}}$;   Return $\sigma_{\text{mes}}$.
```

Signature Verification. The verification algorithm given by Algorithm 3, takes a message and signature pair $(\text{mes}, \sigma_{\text{mes}})$, and outputs Accept or Reject.

In [24], it has been proved that the KSN scheme is secure under the collision-resistance of the hash function. The security theorem is as given below:

Theorem 3.2. *[24] If $\mathcal{A}$ is a quantum adversary that (ϵ, τ)-wins the unforgeability security game for KSN scheme, then $\mathcal{A}$ can be used to $(\epsilon c', \tau + c)$-find function collisions where c, c' are constants.*

Algorithm 3 Signature Verification (`Verify`(mes,σ_{mes},$\mathcal{PK}$))

```
(1) Compute $B'_{\text{mes}} = c_m(\text{mes})$.
(2) If $\sigma_{\text{mes}}[i] \leqslant \frac{t}{2}, \forall 0 \leqslant i < \hat{m}\hat{n}$ ($\sigma_{\text{mes}}[i]$ denotes the $i$-th component of the vector $\sigma_{\text{mes}}$), then continue else Reject.
(3) Compute $\sigma'_{\text{mes}} = \mathbf{y}_{i_0} + \mathbf{y}_{i_1} + \cdots + \mathbf{y}_{i_{\frac{t}{2}-1}}$ (addition in $\mathcal{R}$), and
    $i_j \in B'_{\text{mes}}$.
(4) If $\sigma'_{\text{mes}} = g\text{SWIFFT}_{\text{k}}\,(\sigma_{\text{mes}})$, then Accept, else Reject.
```

Definition 3.3. **Strong Unforgeability.** Let $\mathcal{A}$ has queried on a message M during the query phase of the security game and σ be the signature of the message. In the strong unforgeability game, $\mathcal{A}$ is considered to be successful even if it can return a signature σ^* of the message M where $\sigma^* \neq \sigma$.

SWIFFT hash function used in KSN-OTS is itself highly parallelizable and this leads to higher speed of K2SN-MSS compared to XMSS that is based on W-OTS$^+$ that uses l random walks in the function family $\mathcal{F}_n$ which are inherently sequential. Comparison of the two signature schemes is given in Section 6.2.

3.2 K2SN-MSS construction

K2SN-MSS extends the KSN-OTS to multi-message signature scheme and uses SWIFFT as the underlying hash function. However, SWIFFT compresses $2n$ bits to $n + n_\epsilon$ bits and so concatenation of two outputs (in a node of the Merkle tree) results in $2n + 2n_\epsilon$ bits. We introduce a function `Merge` that effectively slides one string over the other and XORs the overlapping part, to generate a $2n$-bit string. The overlapping part of the two strings is XOR-ed. For our chosen parameters $n = 512$ and $n_\epsilon = 64$.

$$\begin{aligned}\texttt{Merge}(\mathbf{y}_1, \mathbf{y}_2) \quad = \quad & \texttt{hi}_{n-n_\epsilon}(\mathbf{y}_1) \| (\texttt{low}_{2n_\epsilon}(\mathbf{y}_1) \oplus \texttt{hi}_{2n_\epsilon}(\mathbf{y}_2)) \\ & \| \texttt{low}_{n-n_\epsilon}(\mathbf{y}_2), \end{aligned} \tag{7}$$

where (i) $\mathbf{y}_1$ and $\mathbf{y}_2$ are $(n + n_\epsilon)$-bit strings, (ii) $\texttt{hi}_l(\mathbf{y})$ is the most significant l bits of $\mathbf{y}$ and (iii) $\texttt{low}_l(\mathbf{y})$ is the least significant l bits of $\mathbf{y}$. The function `Merge` is used to calculate $y'_{j,i}$ Figure 2 instead of simple concatenation.

Lemma 3.4. *Let f be a random oracle which outputs $(n + n_\epsilon)$-bit long random strings. Let O_c and O_m be two oracles defined as follows: On a query, O_c outputs a $2n$-bit long string y which is computed as $y = \texttt{low}_n(x_1) \| \texttt{low}_n(x_2) : x_1, x_2 \overset{\$}{\leftarrow} f$. On the other hand, O_m outputs a $2n$-bit long string y' which is computed as $y' = \texttt{Merge}(x_1, x_2) : x_1, x_2 \overset{\$}{\leftarrow} f$. Then O_c and O_m are indistinguishable.*

Proof. The proof depends on the following facts: (i) a substring of a random string is a random string, and (ii) concatenation and XOR of two random strings are also random. The job of a attacker is to distinguish the oracles from a given string from one of the oracles. As f and g are two indistinguishable random functions, x_1, x_2, x'_1 and x'_2 are all random strings with indistinguishable distributions. Therefore, y and y' will also have indistinguishable distributions. If the attacker can distinguish the oracle from the output string with non-negligible probability, then the attacker can distinguish between f and g which contradicts the hypothesis. □

3.2.1 Algorithms. K2SN-MSS consists of three algorithms: `K2SN.KeyGen`, `K2SN.Sign` and `K2SN.Verify`.

Key Generation: `K2SN.KeyGen` uses two function families: F_n and SWIFFT as given in Algorithm 4. Here n is the length of the seed of the PRG that is used for secret key generation and is the security parameter of K2SN-MSS. During computation of each node in the tree a different hash key has been used. The public key length is $n + n_\epsilon$ + $\mathtt{hkseed}$ + $\mathtt{rpseed}$ bits.

Algorithm 4 Key Generation of K2SN-MSS ($\mathtt{K2SN.KeyGen}(1^n)$)

```
(1) Choose randomly, (i) $\mathsf{sk} \in \{0,1\}^n$, (ii) $\mathtt{hkseed} \in \{0,1\}^n$ and
    (iii) $\mathtt{rpseed} \in \{0,1\}^n$.
(2) For $i = 0, 1, \ldots, T-1$ (where $T = 2^h$) /* Construction of $\mathcal{L}_i$
    trees */
    (a) For each $\mathcal{L}_i$ tree, compute $\mathsf{sk}_i = \mathtt{low}_n(f_{\mathsf{sk}}(i+1))$.
    (b) For $j = 0, 1, \ldots, t-1$, ( $t = 2^\ell$), compute secret key com-
        ponents of the $i$-th signature as, $\mathbf{x}_{i_j} = f_{\mathsf{sk}_i}(j+1)$.
    (c) For $j = 0, 1, \ldots, t-1$, compute the public key component
        of $i$-th signature as, $\mathbf{y}_{i_j} = \mathrm{SWIFFT}_{\mathsf{k}_i}(\mathbf{x}_{i_j})$.
    (d) Compute $\mathbf{y}_{h,i}$, the root of the Merkle tree $\mathcal{L}_i$ from leaves
        $\mathbf{y}_{i_0}, \mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_{t-1}}$. All the remaining nodes of the $\mathcal{L}_i$ tree
        are computed with a new random pad and hash key.
(3) The roots of the $\mathcal{L}_i$ trees are the leaf nodes of the
    Merkle tree $\mathcal{MSS}$ at height $h$. Denote the nodes by
    $\mathbf{y}_{h,0}, \mathbf{y}_{h,1}, \ldots, \mathbf{y}_{h,T-1}$, and construct the Merkle tree $\mathcal{MSS}$
    using different hash keys and random pads. The root of the
    tree is denoted by $\mathbf{y}_{0,0}$.
(4) Return, (i) Secret key $\mathcal{SK}$ = sk, and (ii) Public key $\mathcal{PK}$ =
    $\{\mathbf{y}_{0,0}, \mathtt{hkseed}, \mathtt{rpseed}\}$.
```

Signature Generation. For the i-th message, mes, the signature $\sigma_{\mathrm{mes}} = (i, \mathrm{pk}, \mathcal{PK}_i, \mathrm{Auth})$, where $i, 1 \le i \le 2^h$ is the index of the signature, pk is the sum of the secret key components associated with B_{mes}, $\mathcal{PK}_i$ is the set of component public keys of the i-th signature (i-th instance of KSN) and Auth contains the nodes of the $\mathcal{MSS}$ tree that are required for verification of $\mathcal{L}_i$ root. The computations of pk, $\mathcal{PK}_i$ and Auth are given in Algorithm 5. The signature length consists of h-bits to represent i, $(\left\lfloor \log_2(\frac{t}{2}) \right\rfloor + 1) \times 2n$ bits to represent pk, $\mathcal{PK}_i$ consisting of t outputs of SWIFFT resulting in $t(n + n_\epsilon)$ bits, and $h(n + n_\epsilon)$ bits to represent Auth which consists of the h sibling nodes. In total signature size is $|\sigma_{\mathrm{mes}}| = h + n \times \left(2\left(\left\lfloor \log_2\left(\frac{t}{2}\right)\right\rfloor + 1\right) + t + h\right) + n_\epsilon \times (t + h)$ bits.

Signature Verification. To verify $\sigma_{\mathrm{mes}} = (i, \mathrm{pk}, \mathcal{PK}_i, \mathrm{Auth})$, first pk is verified against $\mathcal{PK}_i$ using the KSN verification algorithm (Algorithm 3), and then $\mathcal{PK}_i$ is authenticated against $\mathbf{y}_{0,0}$, the root of the $\mathcal{MSS}$ Merkle tree using the Auth. The algorithm is given in Algorithm 6.

Algorithm 5 Signature Generation of K2SN-MSS ($\mathtt{K2SN.Sign}(i, \mathrm{mes}, \mathcal{SK})$)

```
(1) Compute the seed $\mathsf{sk}_i$ for the $\mathcal{L}_i$ tree as $\mathsf{sk}_i = f_{\mathsf{sk}}(i+1)$.
(2) Compute $B_{\mathrm{mes}} = c_m(\mathrm{mes})$ and let $B_{\mathrm{mes}} = \{i_0, i_1, \ldots, i_{\frac{t}{2}-1}\}$
    such that $i_j < i_{j+1}$ for all $0 \leqslant j \leqslant \frac{t}{2} - 2$.
(3) For $j = 0, 1, \ldots, t-1$, compute the component secret keys
    of the $i$-th KSN-OTS as $\mathbf{x}_{i_j} = f_{\mathsf{sk}_i}(j+1)$.
(4) Compute the component public keys of the $i$-th KSN-OTS
    as $\mathcal{PK}_i = \{\mathbf{y}_{i_j} = \mathrm{SWIFFT}_{\mathsf{k}}(\mathbf{x}_{i_j}), j = 0, 1, \ldots, t-1\}$.
(5) $\mathrm{pk} = \mathbf{x}_{i_0} \oplus_p \mathbf{x}_{i_1} \oplus_p \cdots \oplus_p \mathbf{x}_{i_{\frac{t}{2}-1}}$, where $i_j \in B_{\mathrm{mes}}$.
(6) Let $\mathbf{y}_{h,i}$ be the root of the Merkle tree $\mathcal{L}_i$. The Auth contains
    the sibling nodes of the $\mathcal{MSS}$ Merkle tree from $\mathbf{y}_{h,i}$ to the
    root $\mathbf{y}_{0,0}$.
(7) Return $\sigma_{\mathrm{mes}} = (i, \mathrm{pk}, \mathcal{PK}_i, \mathrm{Auth})$.
```

Algorithm 6 Signature Verification of K2SN-MSS ($\mathtt{K2SN.Verify}(\mathrm{mes}, \sigma_{\mathrm{mes}})$)

```
(1) Let $\sigma_{\mathrm{mes}} = (i, \mathrm{pk}, \mathcal{PK}'_i, \mathrm{Auth})$.
(2) Compute $B_{\mathrm{mes}} = c_m(\mathrm{mes})$ and and let $B_{\mathrm{mes}} =$
    $\{j_0, j_1, \ldots, j_{\frac{t}{2}-1}\}$ such that $j_k < j_{k+1}$ for all $0 \leqslant k \leqslant \frac{t}{2} - 2$.
(3) If $\mathrm{pk}[i] \leqslant \frac{t}{2}, \forall 0 \leqslant i < \hat{m}\hat{n}$, then Continue else Reject.
(4) Let $\mathcal{PK}'_i = \{\mathbf{y}'_0, \mathbf{y}'_1, \ldots, \mathbf{y}'_{t-1}\} \in \sigma_{\mathrm{mes}}$.
(5) Compute $\mathrm{pk}' = \mathbf{y}'_{j_0} + \mathbf{y}'_{j_1} + \cdots + \mathbf{y}'_{j_{\frac{t}{2}-1}}$, where the additions
    are in the ring $\mathcal{R}$.
(6) If $\mathrm{pk}' = g\mathrm{SWIFFT}_{\mathsf{k}}(\mathrm{pk})$, then Continue, else Reject.
(7) Compute the root of the Merkle tree $\mathcal{L}_i$ where the leaf nodes
    are $\mathbf{y}'_0, \mathbf{y}'_1, \ldots, \mathbf{y}'_{t-1}$ and with the hash keys and random pads
    used during signing. The hash keys and random pads can be
    easily regenerated from the published seed. Let the root of
    the Merkle tree $\mathcal{L}_i$ be $\mathbf{y}'_{h,i}$.
(8) Compute the root of the $\mathcal{MSS}$ Merkle tree from $\mathbf{y}'_{h,i}$ using
    Auth as
      $\mathbf{y}'_{l,\frac{i}{2}} = \mathrm{SWIFFT}_{\mathsf{k}_{l,\frac{i}{2}}}(\mathtt{Merge}(\mathbf{y}'_{l+1,i}, \mathrm{Auth}[h-l]) \oplus \mathbf{v}_{l,\frac{i}{2}})$,
            where $i = 0 \pmod 2$,
      $\mathbf{y}'_{l,\frac{i-1}{2}} = \mathrm{SWIFFT}_{\mathsf{k}_{l,\frac{i-1}{2}}}(\mathtt{Merge}(\mathrm{Auth}[h-l], \mathbf{y}'_{l+1,i}) \oplus \mathbf{v}_{l,\frac{i-1}{2}})$,
            where $i = 1 \pmod 2$,
    where $\mathrm{Auth}[h-l]$ denotes the $(h-l)$-th node of the Auth.
(9) If $\mathbf{y}'_{0,0} = \mathbf{y}_{0,0}$ (where $\mathbf{y}_{0,0}$ is the published root of the $\mathcal{MSS}$
    Merkle tree as a part of $\mathcal{PK}$), then Accept, else Reject.
```

3.3 K2SN-MSS Security

Theorem 3.5. *If $\mathcal{H}$ is a $2^h(2^\ell - 1)$ multi-function multi-target second-preimage resistant hash function then K2SN-MSS is $(\epsilon, 2^h, \tau)$ secure against Strong Existential Unforgeable under Chosen Message attack (SUF-CMA). This means that for any forger $\mathcal{A}$ with success probability ϵ against K2SN-MSS after 2^h query in time τ. The upper bound of ϵ is*

$$\begin{aligned} \epsilon &\leqslant \frac{\mathit{InSec}^{\mathit{MM\text{-}SPR}}(\mathcal{H}, 2^h, 2^h(1 + 2^\ell))}{\epsilon'}, \ \textit{where} \\ \epsilon' &= (1 - 2^{-2n} - 2^{-m})(1 - 2^{-2n}) \cdot \mathit{max}\left\{2^{-2h}2^{-2(h+\ell)}, (1 - 2^{-2h})2^{-2(h+\ell)}\right\}. \end{aligned}$$

where $q_1, q \le 2^h$, and $q_2 \le 2^{h+\ell} - 1$ respectively.

Proof. Our security derivation is game-based derivation. Let $\mathcal{A}$ is a PPT adversary who has the success probability of ϵ after making q queries in time τ. Let $\mathcal{B}$ be another PPT algorithm which tries to find a second preimage from a target list of tuples (x_i, y_i, k_i) with $y_i = H_{k_i}(x_i)$. $\mathcal{B}$ divides this list in two lists of lengths q_1 and q_2 where $q_1 \leqslant 2^h$ and $q_2 < 2^{h+\ell}$. $\mathcal{B}$ uses $\mathcal{A}$ as a subroutine. Let,

(1) the tuples of the list of length q_1 be (x_i^p, y_i^p, k_i^p), and
(2) other q_2 tuples be (x_i^s, y_i^s, k_i^s).

Let $T = 2^h$ and $t = 2^\ell$. The security game has three phases: (i) Setup Phase, (ii) Query Phase, and (iii) Answer Phase. The description of the phases are given below.

(1) **Setup.**
  (a) $\mathcal{B}$ chooses q_1 distinct values uniformly at random from $\{0, 1, \ldots, T-1\}$ and let this list be $\Gamma = \{c_0, c_1, \ldots, c_{q_1-1}\}$. Each c_i of Γ denotes the c_i-th instance of KSN-OTS.
  (b) $\mathcal{B}$ sets each k_i^p as the hash key of the c_i-th instance of KSN-OTS. For the remaining KSN-OTS, it chooses hash keys uniformly at random.
  (c) Let $(\mathcal{SK}_i, \mathcal{PK}_i)$ be the secret and public key pair of i-th instance of KSN-OTS. Now $\mathcal{B}$ chooses the $\mathcal{SK}_i$ uniformly at random and computes $\mathcal{PK}_i$.
  (d) $\mathcal{B}$ chooses q_1 values uniformly at random from $\{0, 1, \ldots, t-1\}$ and and let this list be $\Psi = \{d_0, d_1, \ldots, d_{q_1-1}\}$. Each element d_i of Ψ denotes the d_i-th component of the secret key and public key of c_i-th instance of KSN-OTS.
  (e) $\mathcal{B}$ then replaces the d_i-th component secret key and public key of c_i-th instance of KSN-OTS by x_i^p and y_i^p respectively.
  (f) $\mathcal{B}$ chooses q_2 nodes from authentication Merkle tree denoted by their positions as (a_i, b_i) where a_i denotes the height in the tree and b_i denotes the b_i-th node at height a_i from left. $a_i \in_R \{0, 1, \ldots, h+\ell-1\}$ and $b_i \in_R \{0, 1, \ldots, 2^{a_i}-1\}$ such that no two (a_i, b_i) pairs are the same. Let the list be $\Upsilon = \{(a_0, b_0), (a_1, b_1), \ldots, (a_{q_2-1}, b_{q_2-1})\}$.
  (g) $\mathcal{B}$ sorts the (a_i, b_i) pairs by the increasing values of a_i and let the sorted list be (a'_i, b'_i). Now if there are multiple nodes with same height, then sort them by b_i. Let the sorted list be $\Upsilon' = \{(a'_0, b'_0), (a'_1, b'_1), \ldots, (a'_{q_2-1}, b'_{q_2-1})\}$.
  (h) $\mathcal{B}$ then constructs the authentication Merkle tree in the following fashion:
    (i) The leaf nodes of the Merkle tree are the components of the $\mathcal{PK}_i$ for $0 \leqslant i \leqslant 2^h - 1$. All the leaves are at height $h + \ell$ where the height of the root node is 0.
    (ii) $\mathcal{B}$ chooses the hash keys and the random pads uniformly at random for all the nodes except the node (a'_i, b'_i)s of the Merkle tree.
    (iii) From the leaf nodes, $\mathcal{B}$ starts constructing the tree upto height $a'_0 - 1$. Except the nodes $(a'_i, b'_i) \in \Upsilon'$ who has the same height as a'_0, $\mathcal{B}$ computes all the nodes of the layer a'_0 normally. For the nodes $(a'_i, b'_i) \in \Upsilon'$ with $a'_i = a'_0$, it sets the value of $y_{a'_i, b'_i}$ as y_j^s from the target list of the second-preimage, hash key as k_j^s and computes the random pad as

$$\mathbf{v}_{a'_i, b'_i} = x_j^s \oplus (\texttt{Merge}(y_{a'_i+1, 2b'_i}, y_{a'_i+1, 2b'_i+1})),$$

because we know that $y_j^s = \text{SWIFFT}_{k_j^s}(x_j^s)$.
    (iv) After completing the layer a'_0, $\mathcal{B}$ continues to compute the authentication Merkle tree faithfully until the next height a'_i in the list Υ' and repeats the step (iii) for new a'_i.
  (i) After completing the construction of the $\mathcal{MSS}$ tree, $\mathcal{B}$ publishes all the hash-key and the random pads along with positions of the nodes for which they have been used and the root of the $\mathcal{MSS}$ tree.

(2) **Query.**
  (a) $\mathcal{B}$ answers q queries of $\mathcal{A}$.
  (b) Let the set of messages be $\{M_0, M_1, \ldots, M_{q-1}\}$, where M_i is the message of i-th query.
  (c) $\mathcal{B}$ returns the signature $s_i = (i, \sigma_i, \mathcal{PK}_i, \texttt{Auth}_i)$ corresponding to the message M_i under the i-th instance of KSN-OTS.

(3) **Answer.**
  (a) In this phase $\mathcal{A}$ returns (M^*, s^*) where $(M^*, s^*) \neq (M_i, s_i)$ for all $0 \leqslant i \leqslant q-1$.
  (b) Let $s^* = (i^*, \sigma^*, \mathcal{PK}^*, \texttt{Auth}^*)$.
  (c) If $\texttt{K2SN.Verify(M}^*, \texttt{s}^*) = 1$, then Continue, else **failure**.
  (d) If $i^* \in \Gamma$, there are three possibilities:
    (i) $M^* = M_{i^*}$, which implies $s^* \neq s_{i^*}$. This case has also two different situations.
      (A) $(\mathcal{PK}^*, \texttt{Auth}^*) = (\mathcal{PK}_{i^*}, \texttt{Auth}_{i^*})$: This implies forgery of the i^*-th instance of KSN-OTS.
        - If $d_{i^*} \in B_{M^*}$, then $\mathcal{B}$ computes the second preimage of $(x_{i^*}^p, y_{i^*}^p, k_{i^*}^p)$ and returns it.
        - elseif $d_{i^*} \notin B_{M^*}$, then $\mathcal{B}$ returns **failure**.
      (B) $(\mathcal{PK}^*, \texttt{Auth}^*) \neq (\mathcal{PK}_{i^*}, \texttt{Auth}_{i^*})$: This implies that $\mathcal{B}$ found at least one collision during the verification of $\mathcal{PK}^*$ under $\texttt{Auth}^*$.
        - If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and the $\mathcal{B}$ found a new image for the node which is different from the value used to compute the tree, then $\mathcal{B}$ found a second-preimage for the node y_{a_i, b_i}.
        - If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and its image is the same as the value used to compute the tree, then $\mathcal{B}$ returns **failure**.
    (ii) $M^* \neq M_{i^*}$, but $s^* = s_{i^*}$. This implies forgery of the i^* instance of KSN-OTS.
      (A) If $d_{i^*} \in B_{M^*}$, then $\mathcal{B}$ computes the second preimage of $(x_{i^*}^p, y_{i^*}^p, k_{i^*}^p)$ and returns it.
      (B) As $d_{i^*} \notin B_{M^*}$, then $\mathcal{B}$ returns **failure**.
    (iii) $M^* \neq M_{i^*}$, but $s^* \neq s_{i^*}$. This case has also two different situations.
      (A) $(\mathcal{PK}^*, \texttt{Auth}^*) = (\mathcal{PK}_{i^*}, \texttt{Auth}_{i^*})$: This implies forgery of the i^* instance of KSN-OTS.
        - If $d_{i^*} \in B_{M^*}$, then $\mathcal{B}$ computes the second preimage of $(x_{i^*}^p, y_{i^*}^p, k_{i^*}^p)$ and returns it.
        - else if $d_{i^*} \notin B_{M^*}$, then $\mathcal{B}$ returns **failure**.
      (B) $(\mathcal{PK}^*, \texttt{Auth}^*) \neq (\mathcal{PK}_{i^*}, \texttt{Auth}_{i^*})$: This implies that $\mathcal{B}$ found at least one collision during the verification of $\mathcal{PK}^*$ under $\texttt{Auth}^*$.
        - If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and then $\mathcal{B}$ found a new image for the node which is different from the value used to

compute the tree, then $\mathcal{B}$ found a second-preimage for the node y_{a_i,b_i}.
  - If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and its image is the same as the value used to compute the tree, then $\mathcal{B}$ returns **failure**.

(e) If $i^* \notin \Gamma$, there are three possibilities:

(i) $M^* = M_{i^*}$ which implies $s^* \neq s_{i^*}$. This case has also two different situations.

(A) $(\mathcal{PK}^*, \mathtt{Auth}^*) = (\mathcal{PK}_{i^*}, \mathtt{Auth}_{i^*})$: This implies forgery of the i^* instance of KSN-OTS. Then $\mathcal{B}$ returns **failure**.

(B) $(\mathcal{PK}^*, \mathtt{Auth}^*) \neq (\mathcal{PK}_{i^*}, \mathtt{Auth}_{i^*})$: This implies that $\mathcal{B}$ found at least one collision during the verification of $\mathcal{PK}^*$ under $\mathtt{Auth}^*$.

- If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and then $\mathcal{B}$ found a new image for the node which is different from the value used to compute the tree, then $\mathcal{B}$ found an second-preimage for the node y_{a_i,b_i}.
- If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and its image is the same as the value used to compute the tree, then $\mathcal{B}$ returns **failure**.

(ii) $M^* \neq M_{i^*}$, but $s^* = s_{i^*}$. This implies forgery of the i^* instance of KSN-OTS. Then $\mathcal{B}$ returns **failure**.

(iii) $M^* \neq M_{i^*}$, but $s^* \neq s_{i^*}$. This case has also two different situations.

(A) $(\mathcal{PK}^*, \mathtt{Auth}^*) = (\mathcal{PK}_{i^*}, \mathtt{Auth}_{i^*})$: This implies forgery of the i^* instance of KSN-OTS. Then $\mathcal{B}$ returns **failure**.

(B) $(\mathcal{PK}^*, \mathtt{Auth}^*) \neq (\mathcal{PK}_{i^*}, \mathtt{Auth}_{i^*})$: This implies that $\mathcal{B}$ found at least one collision during the verification of $\mathcal{PK}^*$ under $\mathtt{Auth}^*$.

- If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and then $\mathcal{B}$ found a new image for the node which is different from the value used to compute the tree, then $\mathcal{B}$ found a second-preimage for the node y_{a_i,b_i}.
- If a node for any $(a_i, b_i) \in \Upsilon$ got calculated during verification and its image is the same as the value used to compute the tree, then $\mathcal{B}$ returns **failure**.

Now we compute the success probability of $\mathcal{B}$.

- **Case 3.d:** As the elements of the list Γ is chosen at random, then the probability of that $i^* \in \Gamma$ is $q_1/2^h$. Now we analyze the three different cases under this situation.
  - **Case 3.d.i:** There are 2^m (m denotes the length of the message space in bits) possible combinations of the messages and messages space is uniformly random. Therefore, the probability of $M^* = M_{i^*}$ is 2^{-m}. Case A happens with probability smaller than 2^{-2n}. Because the min-entropy of the distribution of σ is at least 2^{-2n} [24, 40] and therefore, the probability that the $\mathcal{A}$ will get another σ^* is at most 2^{-2n}. If case A happens, then $d_{i^*} \in 1CFF(M^*)$ has probability $1/2$ and then $\mathcal{B}$ found a second-preimage of the $(x_{d_{i^*}}, y_{d_{i^*}}, k_{d_{i^*}})$ of the q_1 targets. Notice that the case A and B are complementary to each other and then B happens with at least probability $(1 - 2^{-2n})$ [13]. In case of B, there must be one call to SWIFFT between the KSN-OTS public key and the root node where the output for the forgery by the pigeonhole principle and the correct signature agree for the first time and let the node be $(a^*, b^*) \in \Upsilon$. As the nodes of Υ are distributed uniformly at random over the tree, the probability that collision happens exactly at the node (a^*, b^*) is $q_2/2^{h+\ell} - 1$. As the input data depends on previously computed outputs of SWIFFT, it must differ. Hence, for challenge pair $(y_{a^*,b^*}, k_{a^*,b^*})$, the input to this call to SWIFFT is a second preimage for x_{a^*,b^*}, that $\mathcal{B}$ returns breaking MM-SPR of SWIFFT. Notice that all the events under Case 3.d.i are mutually exclusive. Hence maximum success probability of $\mathcal{B}$ under case 3.d.i is
$$2^{-m} \cdot \max\left\{2^{-2n} \times 2^{-1}, (1 - 2^{-2n}) \times q_2 2^{-(h+\ell)}\right\}.$$
As $2n >> h$, $2n >> \ell$, the maximum success probability here is
$$q_2(1 - 2^{-2n})\left\{2^{-(h+\ell+l)}\right\}.$$
  - **Case 3.d.ii:** This case essentially reduces to the problem of finding a different $t/2$-subset of elements in the $\mathcal{SK}_{i^*}$ that sums to σ_{i^*}. Again, this requires at least one element in $\{0,1\}^{2n}$. The probability of this even is upper bounded by one over the number of possible distinct $t/2$- subset sums, meaning in the order of 2^{-2n}.
  - **Case 3.d.iii:** As the cases 3.d.1, 3.d.ii and 3.d.iii are mutually exclusive, then then the probability of this case approximately $(1 - 2^{-m} - 2^{-2n})$. Suppose that case A has occurred. This happens with probability smaller than 2^{-2n} [24, 40]. Indeed, σ_{i^*} is not known to the adversary, and it requires at least one uniformly distributed secret value in $\{0,1\}^{2n}$ since $\mid B_{M^*} \setminus B_{M_{i^*}} \mid \geqslant 1$. The only thing A knows about σ_{i^*} is that it is the sum of at least one uniformly secret $2n$-dimensional binary vector and other values in $[t/2 - 1]^{2n}$, which may be part of σ_{i^*}. Now $d_{i^*} \in B_{M^*}$ has probability $1/2$. Notice that the case A and B are complementary to each other and then B happens with at least probability $(1 - 2^{-2n})$ [13]. Now if A occurs then $d_{i^*} \in B_{M^*}$ happens with probability $1/2$ as there are exactly half of the messages of the total message space whose 1CFF includes d_{i^*}. Now if B occurs, then $\mathcal{B}$ returns breaking MM-SPR of SWIFFT with probability $q_2/2^{h+\ell}$. Therefore, total probability of $\mathcal{B}$ to achieve success under case 3.d.iii is
$$(1 - 2^{-m} - 2^{-2n}) \cdot \max\left\{2^{-2n} \times 1/2, (1 - 2^{-2n})q_2/2^{h+\ell}\right\}.$$
As $2^{-2n} \approx 0$, the maximum success probability here reduces to
$$(1 - 2^{-m} - 2^{-2n}) \cdot (1 - 2^{-2n})q_2 2^{-(h+\ell)}.$$

  Therefore, the maximum success probability of $\mathcal{B}$ under case 3.d will be
$$\begin{aligned} & q_1 2^{-h} \max\left\{q_2(1 - 2^{-2n})2^{-(h+\ell+l)}, 2^{-2n}, (1 - 2^{-m} - 2^{-2n})\cdot\right. \\ & \left.(1 - 2^{-2n})q_2 2^{-(h+\ell)}\right\} \\ = \quad & q_1 2^{-h}(1 - 2^{-m} - 2^{-2n}) \cdot (1 - 2^{-2n})q_2 2^{-(h+\ell)}. \end{aligned}$$

- **Case 3.e:**As the elements of the list Γ is chosen at random, then the probability of that $i^* \notin \Gamma$ is $1 - q_1/2^h$. Now we analyze the three different cases under this situation.
  - **Case 3.e.i:** This happens with probability with 2^{-m}. As case A here is exactly the same as the case 3.d.A, the case A will happen with probability 2^{-2n}. Then case B happens with probability $(1 - 2^{-2n})$ given that case case 3.d.a has occurred. Then $\mathcal{B}$ returns breaking MM-SPR of SWIFFT with probability $q_2 2^{-(h+\ell)}$. Therefore, success probability of $\mathcal{B}$ here is
$$2^{-m} \times (1 - 2^{-2n}) \times q_2 2^{-(h+\ell)}.$$
  - **Case 3.e.ii:** This case essentially as the same as 3.d.ii, meaning the probability of this event is 2^{-2n}.
  - **Case 3.e.iii:** This happens with probability with $(1 - 2^{-2n} - 2^{-m})$. Case B happens with probability with atleast $1 - 2^{-2n}$ given that case case 3.e has occurred. Then $\mathcal{B}$ returns breaking MM-SPR of SWIFFT with probability $q_2/2^{h+\ell}$. Therefore, maximum success probability of $\mathcal{B}$ here is
$$(1 - 2^{-2n} - 2^{-m}) \times (1 - 2^{-2n}) \times q_2 2^{-(h+\ell)}.$$

  Therefore, the probability of success under case 3.e is
$$(1 - q_1 2^{-h})(1 - 2^{-2n} - 2^{-m})(1 - 2^{-2n}) q_2 2^{-(h+\ell)}.$$

If we assume the success probability of $\mathcal{A}$ is ϵ, then the success probability of $\mathcal{B}$ is $\epsilon \cdot \epsilon'$ where

$$\begin{aligned} \epsilon' &= \max\{\text{max. success probability under Case 3.e}, \\ &\quad \text{max. success probability under Case 3.e}\} \\ &= (1 - 2^{-2n} - 2^{-m})(1 - 2^{-2n}) \cdot \max\left\{q_1 2^{-h} q_2 2^{-(h+\ell)}, \right. \\ &\quad \left. (1 - q_1 2^{-h}) q_2 2^{-(h+\ell)}\right\} \end{aligned}$$

Putting $q_1 = 2^h$ and $q_2 = 2^{h+\ell}$, we get that

$$\epsilon' = (1-2^{-2n}-2^{-m})(1-2^{-2n}) \cdot \max\left\{2^{-2h} 2^{-2(h+\ell)}, (1 - 2^{-2h}) 2^{-2(h+\ell)}\right\}.$$

For a target list of length q' after q queries, the success probability of $\mathcal{B}$ is upper-bounded by $\mathrm{InSec}^{\text{MM-SPR}}(\mathcal{H}, q, q')$ and that implies

$$\epsilon \leqslant \mathrm{InSec}^{\text{MM-SPR}}(\mathcal{H}, 2^h, 2^h(1 + 2^\ell))/\epsilon'. \quad \square$$

□

Estimating security level of K2SN-MSS. In [21] it was argued that (See Equation (16) of [21]), if a hash function $\mathcal{H}$ is considered as random oracle, then

$$\mathrm{InSec}^{\text{MM-SPR}}(\mathcal{H}, q, q') = \frac{(q + 1)q'}{2^{n'}},$$

where $\mathcal{H}_K : \{0, 1\}^{2n} \mapsto \{0, 1\}^{n'}$.

In K2SN-MSS, we instantiate $\mathcal{H}$ by SWIFFT with parameters $\hat{m} = 16$, $\hat{n} = 64$ and $p = 257$. Therefore, n', the size of the hash output is 576 bits.

For $h = 20$, $\ell = 9$, $m = 256$, $n = 512$ and $n' = 576$, we have $(1 - 2^{-2n} - 2^{-m})(1 - 2^{-2n}) \approx 1$, and so,

$$\epsilon \quad \leqslant \quad \frac{2^{40}(1 + 2^9)2^{-576}}{\max\{2^{-40}2^{-58}, (1 - 2^{-40})2^{-58}\}} \approx 2^{-469}.$$

The bit-security of the signature scheme is computed as $\log_2(\epsilon/\tau)$ [9], where τ is the time (measured in the number of hash evaluation) required for key generation, q signature generations, q verifications and the time required for $\mathcal{A}$ to forge a signature [40]. Key generation requires $2^{h+\ell+1} - 1$ computations of SWIFFT, signing 2^h signatures requires $2^h \times (h + 1) \times 2^{\ell+1}$ hash computations and verification of 2^h signatures requires $2^h \times (h + 2^{\ell+1})$ hash computations [13]. Therefore, the lower bound on τ is

$$\tau \geqslant 2^{h+\ell+1} + 2^h \times (h + 1) \times 2^{\ell+1} + 2^h \times (h + 2^{\ell+1}) \approx 2^{35}$$

and the bit-security of KSN-MMS for the given parameters will be 504.

4 SIMD PARALLELIZATION OF NTT

Our implementation of K2SN-MMS uses SIMD parallelization of SWIFFT. In the following we show how ring multiplication, which is the most costly operation in SWIFFT, can be parallelized using SIMD instructions of avx2. We use the level of parallelization as a parameter that can be chosen based on the setting.

Notations. We will denote a vector by bold lowercase alphabet (e.g. $\mathbf{z}$, $\boldsymbol{\beta}$). By a single subscript, we denote a vector of dimension 64 (e.g. $\mathbf{z}_i$ is a vector of length 64). If we use two subscript separated by a ",", then it denotes a vector of length 2^δ. We denote the k^{th} component of a vector $\mathbf{z}_i$ (or $\mathbf{z}_{i,j}$) as $\mathbf{z}_i[k]$ (or $\mathbf{z}_{i,j}[k]$). We use $\oplus_p$ and $\odot_p$ as defined in Section 2. The main computation of K2SN-MSS is the computation of SWIFFT. We describe our SIMD parallelization of this computation for the following parameters:

$$\hat{n} = 64, \hat{m} = 16, p = 257 \text{ and } \omega = 42 \pmod{257},$$

These are the parameters given in [5, 28]. For these parameters, the output of SWIFFT and gSWIFFT are elements of $\mathbb{Z}_{257}^{64}$. The input to SWIFFT is from $\mathbb{Z}_2^{1024}$ and to gSWIFFT is from $\mathbb{Z}_{257}^{1024}$. An input vector $\mathbf{z}$ is mapped into elements of $\mathcal{R}$ by partitioning the vector $\mathbf{z}$ into 16 sub-vectors of dimension 64 each, such that $\mathbf{z} = (\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_{15})$, where each $\mathbf{z}_i$ can be represented as a ring element. Further assume that each $\mathbf{z}_i = (z_{i,0}, z_{i,1}, \ldots, z_{i,63})$, with $z_{i,j} \in \mathbb{Z}_2$ (or $z_{i,j} \in \mathbb{Z}_{257}$). Let $\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{\hat{m}-1}$ be the multipliers of a SWIFFT function. Then the hash of $\mathbf{z}$ is computed by equation (4) as $\sum_{i=0}^{15} \mathbf{a}_i \cdot \mathbf{z}_i$. This computation in practice is by equation (5) as $\sum_{i=0}^{15} \mathrm{NTT}(\mathbf{a}_i) \odot_p \mathrm{NTT}(\mathbf{z}_i)$ and so mainly computation of NTT.

Let $\mathbf{y}_i$ be the NTT of $\mathbf{z}_i$ given by $\mathbf{y}_i = \mathrm{NTT}(\mathbf{z}_i) = (y_{i,0}, y_{i,1}, \ldots, y_{i,63})$. Our objective is to parallelize the computation of $\mathbf{y}_i = \mathrm{NTT}(\mathbf{z}_i)$ for $0 \leqslant i \leqslant 15$. We introduce a parameter δ that determines the level of parallelization: that is a δ-parallelized computation of NTT, performs operation on 2^δ components of the vector $\mathbf{y}_i$. We divide $\mathbf{y}_i$ into $\frac{64}{2^\delta}$ vectors of dimension 2^δ such that $\mathbf{y}_i = \left(\mathbf{y}_{i,0}, \mathbf{y}_{i,1}, \ldots, \mathbf{y}_{i,\frac{64}{2^\delta}-1}\right)$, where $\mathbf{y}_{i,\ell} = \left(y_{i,2^\delta \cdot \ell+0}, y_{i,2^\delta \cdot \ell+1}, \ldots, y_{i,2^\delta \cdot \ell+2^\delta-1}\right)$ for $0 \leqslant \ell \leqslant \frac{64}{2^\delta} - 1$. Therefore, each $y_{i,j}$ is j_0^{th} component of the vector $\mathbf{y}_{i,j_1}$ for some $0 \leqslant j_0 \leqslant 2^\delta - 1$ and $0 \leqslant j_1 \leqslant \frac{64}{2^\delta} - 1$. We rewrite the j of equation (8) as $j = j_0 + 2^\delta j_1$, where $0 \leqslant j_0 \leqslant 2^\delta - 1$ and $0 \leqslant j_1 \leqslant \frac{64}{2^\delta} - 1$. As a consequence, k of the equation (8) has to broken down into k_0 and k_1 as $k = k_0 + \frac{64}{2^\delta} k_1$, where $0 \leqslant$

$k_0 \leqslant \frac{64}{2^\delta} - 1$ and $0 \leqslant k_1 \leqslant 2^\delta - 1$. Now we compute each $y_{i,j}$ as,

$$y_{i,j} = \sum_{k=0}^{63} (z_{i,k}\omega^k)\omega^{2jk}, \forall j, 0 \leqslant j \leqslant 63. \tag{8}$$

The equation (8) can be restated as:

$$\begin{aligned} y_{i,j_0+2^\delta j_1} &= \sum_{k=0}^{63} (z_{i,k}\omega^k)\omega^{2jk} &(9)\\ &= \sum_{k_0=0}^{\frac{64}{2^\delta}-1} \left(\omega^{2^{\delta+1}}\right)^{j_1 k_0} \cdot y'_{j_0,k_0}, &(10) \end{aligned}$$

where

$$\begin{aligned} y'_{j_0,k_0} &= \alpha_{j_0,k_0} \cdot \beta_{j_0,k_0}, &(11)\\ \alpha_{j_0,k_0} &= \omega^{(2j_0+1)k_0}, &(12)\\ \beta_{j_0,k_0} &= \sum_{k_1=0}^{2^\delta-1} \left(\omega^{\frac{64}{2^\delta}k_1(1+2j_0)} z_{i,k_0+\frac{64}{2^\delta}k_1}\right). &(13) \end{aligned}$$

For δ-parallelization, let $\mathbf{y}_{i,j_1} = (y_{i,2^\delta \cdot j_1+0}, y_{i,2^\delta \cdot j_1+1}, \ldots, y_{i,2^\delta \cdot j_1+2^\delta-1})$. We compute the components of the vectors $\mathbf{y}_{i,j_1}$ in parallel for $0 \leqslant j_1 \leqslant \frac{64}{2^\delta} - 1$. From equation (10), we know $y_{i,2^\delta \cdot j_1+j_0}$ (for $0 \leqslant j_0 \leqslant 2^\delta - 1$) can be expressed as $\sum_{k_0=0}^{\frac{64}{2^\delta}-1} \left(\omega^{2^{\delta+1}}\right)^{j_1 k_0} \cdot y'_{j_0,k_0}$. Therefore, the vector $\mathbf{y}_{i,j_1}$ can be written as

$$\mathbf{y}_{i,j_1} = \left(\sum_{k_0=0}^{\frac{64}{2^\delta}-1} \left(\omega^{2^{\delta+1}}\right)^{j_1 k_0} \cdot y'_{0,k_0}, \ldots, \sum_{k_0=0}^{\frac{64}{2^\delta}-1} \left(\omega^{2^{\delta+1}}\right)^{j_1 k_0} \cdot y'_{2^\delta-1,k_0} \right). \tag{14}$$

Thus the vector $\mathbf{y}_{i,j_1}$ can be computed from the vector $\mathbf{y}'_{j_0,k_0} = (y'_{0,k_0}, y'_{1,k_0}, \ldots, y'_{2^\delta-1,k_0})$ by multiplications by scalars $(\omega^{2^{\delta+1}})^{j_1 k_0}$, $0 \leqslant k_0 \leqslant \frac{64}{2^\delta} - 1$.

On the other hand, each component y'_{j_0,k_0} of the vector $\mathbf{y}'_{j_0,k_0}$ depends on α_{j_0,k_0} and β_{j_0,k_0} through the equation (11). Thus, we can express the vector $\mathbf{y}'_{j_0,k_0}$ as $\mathbf{y}'_{j_0,k_0} = (\alpha_{0,k_0} \cdot \beta_{0,k_0}, \alpha_{1,k_0} \cdot \beta_{1,k_0}, \ldots, \alpha_{2^\delta-1,k_0} \cdot \beta_{2^\delta-1,k_0})$, which can be viewed as the component-wise multiplication of the vectors $\boldsymbol{\alpha}_{j_0,k_0} = (\alpha_{0,k_0}, \alpha_{1,k_0}, \ldots, \alpha_{2^\delta-1,k_0})$, and $\boldsymbol{\beta}_{j_0,k_0} = (\beta_{0,k_0}, \beta_{1,k_0}, \ldots, \beta_{2^\delta-1,k_0})$.

The vectors $\boldsymbol{\alpha}_{j_0,k_0}$ can be precomputed as they do not depend on the input string. If $\mathbf{z} \in \mathbb{Z}_2^{1024}$, the vectors $\boldsymbol{\beta}_{j_0,k_0}$ can be precomputed or computed in real-time during the execution and this depends on the level of parallelization. To compute vectors $\mathbf{y}_i$ in parallel, we first compute vectors $\mathbf{y}'_{j_0,k_0}$ from $\boldsymbol{\alpha}_{j_0,k_0}$, and $\boldsymbol{\beta}_{j_0,k_0}$, and then using equation (14), compute the vectors $\mathbf{y}_{i,j_1}$ from the vectors $\mathbf{y}'_{j_0,k_0}$. Details are in Section 4.1.

4.1 Parallelization of NTT when $\mathbf{z}_i \in \mathbb{Z}_2^{64}$ (bNTT$2^\delta(\cdot)$)

For $\mathbf{z}_i \in \mathbb{Z}_2^{64}$, the computation is called bNTT $- 2^\delta(\cdot)$, and given below.

(1) **Precomputation of vectors $\boldsymbol{\alpha}_{j_0,k_0}$:** Define vectors $\boldsymbol{\alpha}_{j_0,k_0} = (\alpha_{0,k_0}, \alpha_{1,k_0}, \ldots, \alpha_{2^\delta-1,k_0}) \in \mathbb{Z}_{257}^{2^\delta}$, where α_{j_0,k_0} is defined by equation (12) for $0 \leqslant k_0 \leqslant \frac{64}{2^\delta} - 1$. The vectors $\boldsymbol{\alpha}_{j_0,k_0}$ can have $\frac{64}{2^\delta}$ possible values depending on the value of k_0. Let Λ_1 be the precomputation table where rows of the table are the vectors $\boldsymbol{\alpha}_{j_0,k_0}$. There are k_0 rows of the table Λ_1 which are indexed by k_0.

(2) **Precomputation of vectors $\boldsymbol{\beta}_{j_0,k_0}$:** Define vectors $\boldsymbol{\beta}_{j_0,k_0} = (\beta_{0,k_0}, \beta_{1,k_0}, \ldots, \beta_{2^\delta-1,k_0}) \in \mathbb{Z}_{257}^{2^\delta}$, where β_{j_0,k_0} is defined by equation (13). For a given k_0, $\boldsymbol{\beta}_{j_0,k_0}$ can have 2^{2^δ} possible values depending on the combination of the bits

$$\mathbf{z}_{i,k_0} = \left(z_{i,k_0+\frac{64}{2^\delta}\cdot 0}, z_{i,k_0+\frac{64}{2^\delta}\cdot 1}, \ldots, z_{i,k_0+\frac{64}{2^\delta}\cdot(2^\delta-1)}\right), \tag{15}$$

that can be easily precomputed and stored. Let the precomputation table Λ_2 with these entries be indexed from 0 to $2^{2^\delta} - 1$. We can choose an entry from the table by the decimal value of $\mathbf{z}_{i,k_0}$ computed from the expression,

$$\mathrm{dec}_{2^\delta}(\mathbf{z}_{i,k_0}) = z_{i,k_0+\frac{64}{2^\delta}\cdot 0} 2^0 + \cdots + z_{i,k_0+\frac{64}{2^\delta}\cdot(2^\delta-1)} 2^{2^\delta-1}.$$

(3) **Computation of vectors $\mathbf{y'}_{j_0,k_0}$:** For $0 \leqslant j_0 \leqslant 2^\delta - 1$, the j_0^{th} component of $\mathbf{y}_{j_0,k_0}$ only depends on the j_0^{th}-components of the elements $\Lambda_1(k_0)$s and $\Lambda_2(\mathrm{dec}_{2^\delta}(\mathbf{z}_{i,k_0}))$s of $0 \leqslant k_0 \leqslant \frac{64}{2^\delta} - 1$. Following the observation, we can compute, for $0 \leqslant k_0 \leqslant \frac{64}{2^\delta} - 1$,

$$\mathbf{y'}_{j_0,k_0} = \Lambda_1(k_0) \odot_{257} \Lambda_2(\mathrm{dec}(\mathbf{z}_{i,k_0})). \tag{16}$$

(4) **Computation of the output vectors $\mathbf{y}_{i,j_1}$:** Using the vectors $\mathbf{y'}_{j_0,k_0}$, $0 \leqslant k_0 \leqslant \frac{64}{2^\delta} - 1$, the output vectors $\mathbf{y}_{i,j_1}$ can be computed as:

$$\mathbf{y}_{i,j_1} = \sum_{k_0=0}^{\frac{64}{2^\delta}-1} (\omega^{16})^{j_1 k_0} \mathbf{y'}_{j_0,k_0}, \forall 0 \leqslant j_1 \leqslant \frac{64}{2^\delta} - 1. \tag{17}$$

In order to compute the vectors $\mathbf{y}_{i,j_1}$ of the equation (17), 2^δ vector multiplications are required and there are $\frac{64}{2^\delta}$ such vectors. Therefore, the computation is 64 vector multiplications.

Memory requirement of the precomputation: Required memory for the table Λ_1 of vectors $\boldsymbol{\alpha}_{j_0,k_0}$ is of $O\left(\frac{64}{2^\delta}\right)$. The size of the table Λ_2 of vectors $\boldsymbol{\beta}_{j_0,k_0}$ is of order $O\left(2^{2^\delta}\right)$. The memory requirement is thus dominated by the size of the Λ_2 and it increases exponentially as δ increases.

4.2 Parallelization of NTT when $\mathbf{z}_i \in \mathbb{Z}_{257}^{64}$ (gNTT$2^\delta(\cdot)$)

The input to the function gNTT is a vector in $\mathbb{Z}_{257}^{64}$. The computation of $\mathbf{y_i} = g\mathrm{NTT}(\mathbf{z}_i)$ is the same as the computation of the function bNTT except for Step 2, for which we provide details below.

The components of $\mathbf{z}_{i,k_0}$ in equation (15) are now elements from $\mathbb{Z}_{257}$ and so vector $\mathbf{z}_{i,k_0} = \left(z_{i,k_0+\frac{64}{2^\delta}\cdot 0}, z_{i,k_0+\frac{64}{2^\delta}\cdot 1}, \ldots, z_{i,k_0+\frac{64}{2^\delta}\cdot(2^\delta-1)}\right)$ has 257^{2^δ} possible values. If we processed the same as the computation of bNTT, the size of the table Λ_2 will be $2^{2^{3\delta}}$ which is not acceptable. Therefore, we compute the required $\boldsymbol{\beta}_{j_0,k_0}$ vectors during the computation of gNTT$(\cdot)$ as given in Algorithm 7.

Define function $\mathtt{Select}2^\delta(\mathbf{z}_{i,k_0}, j) = (\mathbf{z}_{i,k_0}[j], \mathbf{z}_{i,k_0}[j], \ldots, \mathbf{z}_{i,k_0}[j])$. To compute vectors $\boldsymbol{\beta}_{j_0,k_0}$ as in Algorithm 7, we also require the following precomputed vectors

$$\boldsymbol{\omega}_{k_1,j_0} = \left(\omega^{\frac{64}{2^\delta}k_1(1+2\cdot 0)}, \omega^{\frac{64}{2^\delta}k_1(1+2\cdot 1)}, \ldots, \omega^{\frac{64}{2^\delta}k_1(1+2\cdot(2^\delta-1))}\right)$$
$$\text{for } 0 \leqslant k_1 \leqslant 2^\delta - 1.$$

```
Algorithm 7 Parallelized computation of $\boldsymbol{\beta}_{j_0,k_0}$
    Input: $(\mathbf{z}_{i,0},\ldots,\mathbf{z}_{i,\frac{64}{2^\delta}-1})$ and $(\boldsymbol{\omega}_{0,j_0},\ldots,\boldsymbol{\omega}_{2^\delta-1,j_0})$;
    Output: $\boldsymbol{\beta}_{j_0,0},\boldsymbol{\beta}_{j_0,1},\ldots,\boldsymbol{\beta}_{j_0,\frac{64}{2^\delta}-1}$;
    $\boldsymbol{\beta}_{j_0,k_0} = \boldsymbol{\beta}_{j_0,k_0} \oplus_{257} \left(\boldsymbol{\omega}_{k_1,j_0} \odot_{257} \texttt{Select2}^\delta(\mathbf{z}_{i,k_0},k_1)\right)$, $\forall 0 \leqslant k_0 \leqslant \frac{64}{2^\delta}-1, \forall 0 \leqslant k_1 \leqslant 2^\delta - 1$
```

4.3 Optimization of the Step-4 of $b\mathrm{NTT}2^\delta$ and $g\mathrm{NTT}2^\delta$ for $\delta = 3$

To compute vectors $\mathbf{y}_{i,j_1}$ of the equation (17), 64 vector multiplications are required. However, the particular choice of $\omega \equiv 42 \pmod{257}$, we have $\omega^{16} \equiv 2^2 \pmod{257}$ and $(\omega^{16})^4 \equiv -1 \pmod{257}$. This will remove the need for multiplication and simplify the computations of $\mathbf{y}_{i,j_1}$s of equation (17) using left shifts by 2 and addition operations only. Table 2 includes the computations of the $\mathbf{y}_{i,j_1}$ and the expressions that involve only component-wise addition or subtraction. In Table 3, we list all the sub-expressions which we have to compute to obtain the final expressions of Table 2. The expressions of the Table 3 include component-wise addition or subtraction, left shifts by 2, 4 and 6 only.

4.4 Optimization of the Step-4 of $b\mathrm{NTT}2^\delta$ and $g\mathrm{NTT}2^\delta$ for $\delta = 4$

For $\omega \equiv 42 \pmod{257}$, we have $\omega^{32} \equiv 2^4 \pmod{257}$ and $(\omega^{32})^2 \equiv -1 \pmod{257}$. We can simplify the computations of $\mathbf{y}_{i,j_1}$s of equation (17) using left shifts by 4 and addition/subtraction operations only, without any multiplication. Table 4 includes all the computations of the $\mathbf{y}_{i,j_1}$ and all its the expressions involve only component-wise addition or subtraction. On the other hand, Table 5 includes all the sub-expressions which are required to compute the final expressions of Table 4. Observe that the expressions of the Table 5 only need component-wise addition and subtraction and left shift by 4.

Table 2: Final computations of $\mathbf{y}_{i,j_1}$

j_1	$\mathbf{y}_{i,j_1} \in \mathbb{Z}_{257}^8$
0	$\mathbf{y}''_{i,0} \oplus_{257} \mathbf{y}''_{i,1}$
1	$\mathbf{y}''_{i,8} \oplus_{257} \mathbf{y}''_{i,9}$
2	$\mathbf{y}''_{i,6} \oplus_{257} \mathbf{y}''_{i,7}$
3	$\mathbf{y}''_{i,10} \oplus_{257} \mathbf{y}''_{i,11}$
4	$\mathbf{y}''_{i,0} \ominus_{257} \mathbf{y}''_{i,1}$
5	$\mathbf{y}''_{i,8} \ominus_{257} \mathbf{y}''_{i,9}$
6	$\mathbf{y}''_{i,6} \ominus_{257} \mathbf{y}''_{i,7}$
7	$\mathbf{y}''_{i,10} \ominus_{257} \mathbf{y}''_{i,11}$

Table 3: Sub-computations of $\mathbf{y}_{i,j_1}$ of Table 2, where each $\mathbf{y}''_{i,\cdot} \in \mathbb{Z}_{257}^8$. Component-wise left Shift is denoted by $\ll$.

$\mathbf{y}''_{i,0} = \mathbf{y}'_{i,0} \oplus_{257} \mathbf{y}'_{i,2} \oplus_{257} \mathbf{y}'_{i,4} \oplus_{257} \mathbf{y}'_{i,6}$
$\mathbf{y}''_{i,1} = \mathbf{y}'_{i,1} \oplus_{257} \mathbf{y}'_{i,3} \oplus_{257} \mathbf{y}'_{i,5} \oplus_{257} \mathbf{y}'_{i,7}$
$\mathbf{y}''_{i,2} = \mathbf{y}'_{i,0} \ominus_{257} \mathbf{y}'_{i,4}$
$\mathbf{y}''_{i,3} = \left(\mathbf{y}'_{i,1} \ominus_{257} \mathbf{y}'_{i,5}\right)$
$\mathbf{y}''_{i,4} = \left(\mathbf{y}'_{i,2} \ominus_{257} \mathbf{y}'_{i,6}\right) \ll 4$
$\mathbf{y}''_{i,5} = \left(\mathbf{y}'_{i,3} \ominus_{257} \mathbf{y}'_{i,7}\right)$
$\mathbf{y}''_{i,6} = \mathbf{y}'_{i,0} \oplus_{257} \mathbf{y}'_{i,4} \ominus_{257} \mathbf{y}'_{i,2} \ominus_{257} \mathbf{y}'_{i,6}$
$\mathbf{y}''_{i,7} = \left(\mathbf{y}'_{i,1} \oplus_{257} \mathbf{y}'_{i,5} \ominus_{257} \mathbf{y}'_{i,3} \ominus_{257} \mathbf{y}'_{i,7}\right) \ll 4$
$\mathbf{y}''_{i,8} = \mathbf{y}''_{i,2} \oplus_{257} \mathbf{y}''_{i,4}$
$\mathbf{y}''_{i,9} = (\mathbf{y}''_{i,3} \ll 2) \oplus_{257} (\mathbf{y}''_{i,5} \ll 6)$
$\mathbf{y}''_{i,10} = \mathbf{y}''_{i,2} \ominus_{257} \mathbf{y}''_{i,4}$
$\mathbf{y}''_{i,11} = (\mathbf{y}''_{i,3} \ll 6) \oplus_{257} (\mathbf{y}''_{i,5} \ll 2)$

Table 4: Final computations of $\mathbf{y}_{i,j_1}$

j_1	$\mathbf{y}_{i,j_1} \in \mathbb{Z}_{257}^{16}$
0	$\mathbf{y}''_{i,0} \oplus_{257} \mathbf{y}''_{i,1}$
1	$\mathbf{y}''_{i,2} \oplus_{257} \mathbf{y}''_{i,3}$
2	$\mathbf{y}''_{i,0} \ominus_{257} \mathbf{y}''_{i,1}$
3	$\mathbf{y}''_{i,2} \ominus_{257} \mathbf{y}''_{i,3}$

Table 5: Sub-computations of $\mathbf{y}_{i,j_1}$ of Table 4, where each $\mathbf{y}''_{i,\cdot} \in \mathbb{Z}_{257}^8$. Component-wise left Shift is denoted by $\ll$.

$\mathbf{y}''_{i,0} = \mathbf{y}'_{i,0} \oplus_{257} \mathbf{y}'_{i,2}$
$\mathbf{y}''_{i,1} = \mathbf{y}'_{i,1} \oplus_{257} \mathbf{y}'_{i,3}$
$\mathbf{y}''_{i,2} = \mathbf{y}'_{i,0} \ominus_{257} \mathbf{y}'_{i,2}$
$\mathbf{y}''_{i,3} = (\mathbf{y}'_{i,1} \ominus_{257} \mathbf{y}'_{i,3}) \ll 4$

4.5 Parallelizing SWIFFT with parameter δ

Let $(\mathbf{z}_0,\ldots,\mathbf{z}_{15})$ and $\mathbf{x}_S = (\mathbf{x}_{S,0},\ldots,\mathbf{x}_{S,\frac{64}{2^\delta}-1}), \mathbf{x}_{S,i} \in \mathbb{Z}_{257}^{2^\delta}, \forall i$, be the input and output of SWIFFT, respectively, and $(\mathbf{a}_0,\mathbf{a}_1,\ldots,\mathbf{a}_{15})$ denote the vector of multipliers. The pre-computation for SWIFFT and gSWIFFT, both, will be for $g\mathrm{NTT}2^\delta(\mathbf{a}_i) = (\mathbf{A}_{i,0},\mathbf{A}_{i,1},\ldots,\mathbf{A}_{i,\frac{64}{2^\delta}})$. The computation of $\mathbf{x}_S$ is given in the Algorithm 8.

In gSWIFFT-2^δ, the input vector $\mathbf{z}_i$s are in $\mathbb{Z}_{257}^{64}$, and the computation is given by Algorithm 9, where the inputs are processed by the function $g\mathrm{NTT}2^\delta$ while the rest of the computation is the same as in Algorithm 8. Note that NTT computations and SWIFFT function evaluation only use vector operations $\oplus_{257}$ and $\odot_{257}$ that are performed on the vectors component-wise. The output vector of these vector operations can be directly used for the next vector operation. Thus the above 2^δ-way parallelization of NTT implementation of SWIFFT function is a suitable candidate for SIMD-based parallelization. The choice of δ depends on the available intrinsic operations that are required for integer SIMD operations.

We provide two new implementations of SWIFFT for $\delta = 3$ and $\delta = 4$ using avx2 intrinsic [22], supporting 256-bit registers for integer operations. We refer to the two implementations as SWIFFT-8-avx2 and SWIFFT-16-avx2, achieving 8-way and 16-way

Algorithm 8 SWIFFT-2^δ

```
Input: $(\mathbf{z}_0, \ldots, \mathbf{z}_{\hat{m}-1}), \mathbf{z}_i \in \mathbb{Z}_2^{64}$; $(\mathbf{A}_{i,0}, \ldots, \mathbf{A}_{i,\frac{64}{2^\delta}}), 0 \leqslant i \leqslant \hat{m}-1$;
Output: $\mathbf{x}_\mathrm{S} = (\mathbf{x}_{\mathrm{S},0}, \ldots, \mathbf{x}_{\mathrm{S},\frac{64}{2^\delta}-1}) \in \mathbb{Z}_{257}^{64}$;
    (1) Compute $\mathbf{y}_i = b\mathrm{NTT}2^\delta(\mathbf{z}_i) = (\mathbf{y}_{i,0}, \ldots, \mathbf{y}_{i,\frac{64}{2^\delta}-1}), \forall 0 \leqslant i \leqslant \hat{m}-1$;
    (2) Compute $\mathbf{t}_{i,j} = \mathbf{y}_{i,j} \odot_{257} \mathbf{A}_{i,1}, \forall 0 \leqslant i \leqslant \hat{m}-1, \forall 0 \leqslant j \leqslant \frac{64}{2^\delta}$.
    (3) $\mathbf{x}_{\mathrm{S},i} = \mathbf{x}_{\mathrm{S},i} \oplus_{257} \mathbf{t}_{j,i}, \forall 0 \leqslant i \leqslant \frac{64}{2^\delta}, \forall 0 \leqslant j \leqslant \hat{m}-1$.
```

Algorithm 9 generalized SWIFFT-2^δ, in short gSWIFFT-2^δ

```
Input: $(\mathbf{z}_0, \ldots, \mathbf{z}_{15})$ where $\mathbf{z}_i \in \mathbb{Z}_2^{64}$ and $\left(\mathbf{A}_{i,0}, \ldots, \mathbf{A}_{i,\frac{64}{2^\delta}}\right)$ for $0 \leqslant i \leqslant 15$;
Output: $\mathbf{x}_\mathrm{S} = \left(\mathbf{x}_{\mathrm{S},0}, \ldots, \mathbf{x}_{\mathrm{S},\frac{64}{2^\delta}-1}\right) \in \mathbb{Z}_{257}^{64}$;
    (1) Compute $\mathbf{y}_i = g\mathrm{NTT}2^\delta(\mathbf{z}_i) = \left(\mathbf{y}_{i,0}, \ldots, \mathbf{y}_{i,\frac{64}{2^\delta}-1}\right), \forall 0 \leqslant i \leqslant 15$.
    (2) Compute $\mathbf{t}_{i,j} = \mathbf{y}_{i,j} \odot_{257} \mathbf{A}_{i,1}, \forall 0 \leqslant i \leqslant 15, \forall 0 \leqslant j \leqslant \frac{64}{2^\delta} - 1$.
    (3) Compute $\mathbf{x}_{\mathrm{S},i} = \mathbf{x}_{\mathrm{S},i} \oplus_{257} \mathbf{t}_{j,i}, \forall 0 \leqslant i \leqslant \frac{64}{2^\delta} - 1, \forall 0 \leqslant j \leqslant 15$.
```

parallelizations, respectively. SWIFFT-16-avx2 has the highest parallelization level to the date which gives the fastest implementation. We obtained better implementation results for K2SN-MSS using SWIFFT-16-avx2 than SWIFFT-8-avx2 and so in the following we give implementation details of SWIFFT-16-avx2 only. We omit the implementation details of SWIFFT-8-avx2 here due to page limits. All implemented codes are publicly available at GitHub [23].

The existing benchmark implementation of SWIFFT in [31] corresponds to [5, 28]. The implementation uses $\delta = 3$ and parallelizes the code using `sse2` [22] instructions of `Intel` processors. We refer to this implementation as SWIFFT-8-`sse2`.

5 K2SN-MSS SOFTWARE IMPLEMENTATION

In this section, we provide the details of the implementation of the software K2SN-MSS. Table 6 lists the values of the parameters used for the software K2SN-MSS and also the functions are used.

The performance of K2SN-MSS is dominated by the computation of SWIFFT. We describe our SIMD parallelization of this computation for the following parameters:

$$\hat{n} = 64, \hat{m} = 16, p = 257 \text{ and } \omega = 42 \pmod{257},$$

that were given in [5, 28].

Our reasoning for these choices are below. Lyubashevsky et al. reduced the SWIFFT function to the subset sum problem and used k-list attack to compute the preimage of a SWIFFT function [28] in practice.

This choice of parameters also allows us to compare our implementation of SWIFFT against existing software of SWIFFT [31], both for correctness and efficiency. Note that in $\mathbb{Z}_p$, 42 is a $2\hat{n} = 128$-th root of unity. Therefore, for NTT, we use $\omega = 42 \pmod{257}$ and show that with this choice of ω, multiplications reduce to bit-wise left-shift operations and results efficient implementation.

Parameters		
Symbol	Value	Meaning
n	512	Security parameter
m	256	Message Space
$\hat{n}$	64	Dimension of the ring $\mathcal{R}$ of SWIFFT
$\hat{m}$	16	Number of multipliers of SWIFFT
p	257	Prime of SWIFFT
t	262	optimized CFF for 256-bit messages
Functions		
Name	Description	
Hash Function: SWIFFT	$\{0,1\}^{\hat{n}\hat{m}} \mapsto \{0,1\}^{\hat{n}\lceil\log_2(p)\rceil}$	
Hash Function: gSWIFFT	$\{0,1\}^{\hat{n}\hat{m}\lceil\log_2(p)\rceil} \mapsto \{0,1\}^{\hat{n}\lceil\log_2(p)\rceil}$	
PRF ChaCha20	$\{0,1\}^{\hat{m}\hat{n}} \mapsto \{0,1\}^{\hat{m}\hat{n}}$	
1CFF: c_m	$\{0,1\}^m \mapsto \{i_0, i_1, \ldots, i_{\frac{t}{2}-1}\}$, where each $0 \leqslant i_j < t$ and $i_{j_1} \neq i_{j_2}$ for any $0 \leqslant j_1 < j_2 < \frac{t}{2}$	
Sizes		
Name	Lengths	
Signature Size	21331 Bytes	
Secret Key Size	40 Bytes	
Public Key Size	152 Bytes	

Table 6: K2SN-MSS parameters, functions and resulting sizes

For $\hat{m} = 16$ and $\hat{n} = 64$, the input and output of SWIFFT are, 1024 and 576 bits, respectively. Therefore, SWIFFT is a hash function which compresses $2n$ bits to $n + n_\epsilon$ with $n = 512$ and $n_\epsilon = 64$.

Our implemented software is for 256-bit messages. Relaxing the condition $\frac{t}{2} < \frac{p}{2}$ by 3, we choose $\frac{t}{2} = 131$, such that t becomes the smallest even positive integer 262 where $\log_2 \binom{262}{131} \geqslant 256$. Therefore we use $t = 262$ for our software. We start with by providing the implementation details using `intel` `avx2` intrinsic.

On the other hand, choice of these parameters for SWIFFT hash function leads to 512-bit classical (256-bit quantum) security of K2SN-MSS in multi-function-multi-target model. Therefore, we can only compare our implemented signature scheme against hash-based signature schemes which provide same level of security in the same security model. The most optimized implementation of XMSS at the same security level is given in [15]. It should be noted that there is no existing software of XMSS^{MT} and SPHINCS at the same security level.

5.1 SWIFFT-16-avx2

In the implementation of 16-way parallelization, each 256-bit register `__m256i` is partitioned into 16 blocks of 16-bit each. The input strings are over $\mathbb{Z}_{257}$, where each element is 9-bit. The resulting vector of 16 $\mathbb{Z}_{257}$-elements is stored in a register. This is called *packing*. After completing the computation, the register content is moved to an array of integers. This is called *unpacking*.

We will denote packed integers as vectors of $(\mathtt{int}_0, \mathtt{int}_1, \ldots, \mathtt{int}_{15})$, each $\mathtt{int}_i$ up to 16-bit. Here, unless otherwise stated, vectors are of dimension 16. Initially, each of the 16-bit blocks contains an element of $\mathbb{Z}_{257}$ which is represented by 9-bit. The result of vector addition and multiplication must be reduced modulo $p = 257$. The Details of the avx2 implementations of modular reduction, and the three main vector operations addition, subtraction and multiplication are given in Appendix **??** and the code is publicly available at [23].

5.2 Modular Reduction

We use two types of modular reductions referred to as LazyReduced16 and Reduced16. In LazyReduced16 the output vector can have components that are negative integers, while the components of Reduced16 output vector are integers in the range $[0, p-1]$. LazyReduced16 is a faster operation but the negative output may not be accepted as the input in the next operation such as shifting and so we use LazyReduced16 when negative output is acceptable by the next operation.

(1) LazyReduced16: Let $\mathbf{c} = (c_0, c_1, \ldots, c_{15})$. Each component c_i of $\mathbf{c}$ can be written as $c_i = c_{i,0} + c_{i,1} \cdot 2^8 = (c_{i,0} - c_{i,1}) + c_{i,1} \cdot (2^8 + 1)$. Therefore $c_i = c_{i,0} - c_{i,1} \pmod{p}$. Note that the i-th component, depending on the values of $c_{i,0}$ and $c_{i,1}$, can be negative or positive. The _mm256_and_si256 instruction computes the bit-wise AND of two __m256i registers. Let *mask*255 be a vector of length 16, and each component being 255. Applying _mm256_and_si256 on $\mathbf{c}$ and *mask*255, results in the vector $\mathbf{c}_0 = (c_{0,0}, c_{1,0}, \ldots, c_{15,0})$. The _mm256_srli_epi16 performs right-shift on packed 16-bit integers of __m256i. We compute $\mathbf{c}_1$ = _mm256_srli_epi16($\mathbf{c}$, 8). Let $\mathbf{c}_1 = (c_{0,1}, c_{1,1}, \ldots, c_{15,1})$. We obtain the lazy reduced vector $\hat{\mathbf{c}}$ as _mm256_sub_epi16($\mathbf{c}_0, \mathbf{c}_1$), where _mm256_sub_epi16 performs subtraction on packed 16-bit integers of two __m256i registers.

(2) Reduced16: Let $\hat{\mathbf{c}} = \{\hat{\mathbf{c}}_0, \hat{\mathbf{c}}_1, \ldots, \hat{\mathbf{c}}_{15}\}$ = LazyReduced16($\mathbf{c}$). To determine if the i-th component of the reduced vector $\hat{\mathbf{c}}$ is negative, we use $\mathbf{c}'$ = _mm256_cmpgt_epi16($\hat{\mathbf{c}}$, allone), where allone is a vector of all -1 component. If $\hat{\mathbf{c}}_i \geqslant 0$, then $\mathbf{c}'_i = -1$, else it is 0. Thus we obtain $\mathbf{c}^*$ with 0 and -1 components, and 0 components corresponding to negative components of $\mathbf{c}'$. Let $\mathbf{p}$ be a vector of length 16, each component being $p = 257$. By applying _mm256_and_si256 on $\mathbf{p}$ and $\mathbf{c}'$, we obtain a vector whose i-th component is p modulo p if $\hat{\mathbf{c}}_i < 0$, else 0. To obtain $\hat{\mathbf{c}}$ modulo p, we use _mm256_add_epi16($\mathbf{p}'$ and $\hat{\mathbf{c}}$), to add $\mathbf{p}'$ and $\hat{\mathbf{c}}$ where _mm256_add_epi16 performs additions on packed 16-bit integers of the two __m256i registers.

For two vectors $\mathbf{a} = (a_0, a_1, \ldots, a_{15})$ and $\mathbf{b} = (b_0, b_1, \ldots, b_{15})$, we perform the following vector operations.

5.3 Modular Vector Addition ($\oplus_{257}$)

The addition of the vectors is performed by _mm256_add_epi16 and then performing the modular reduction on the result.

5.4 Modular Vector Subtraction ($\ominus_{257}$)

First add the vector $k\mathbf{p}$ (each component kp, for an appropriate choice of integer k), to the vector $\mathbf{a}$ such that all components of the vector $(\mathbf{a} + k\mathbf{p}) - \mathbf{b}$ are positive. The vector subtraction operation is done by the instruction _mm256_sub_epi16. Because of adding $k\mathbf{p}$, the positive components of $(\mathbf{a} - \mathbf{b})$ will become large and so we perform a modular reduction at the end to reduce the components to 9-bit representation.

5.5 Modular Vector Multiplication ($\odot_{257}$)

Each a_i and b_i is 9-bit long and so $c_i = a_i \cdot b_i$ may need more than 16-bits to represent, and this will result in an overflow. To overcome this problem, c_i is divided into two parts, $c_i = c_{i,0} + c_{i,1} \cdot 2^{16}$, and the reduction is computed as:

$$\begin{aligned} c_i &= c_{i,0} + c_{i,1} \cdot 2^8(2^8+1) - c_{i,1} \cdot 2^8 \\ &= c_{i,0} - c_{i,1}(2^8+1) + c_{i,1} = c_{i,0} + c_{i,1} \pmod{257}. \end{aligned}$$

The maximum value of a_i and b_i is 256. If both a_i and b_i are 256 then only $c_{i,1}$ becomes 1 and $c_{i,0}$ is zero. For all other combinations of a_i and b_i, $c_{i,1}$ is 0.

Let $\mathbf{c}_0 = \{c_{0,0}, c_{1,0}, \ldots, c_{15,0}\}$ and $\mathbf{c}_1 = \{c_{0,1}, c_{1,1}, \ldots, c_{15,1}\}$. We compute the vector $\mathbf{c}_0$ using the instruction _mm256_mullo_epi16, and $\mathbf{c}_1$ using the instruction _mm256_mulhi_epu16 from the $\mathbf{a}$ and $\mathbf{b}$. Then we perform modular reduction Reduced16 on the vector $\mathbf{c}_0$. The final vector is $\mathbf{c}^*$. Adding $\mathbf{c}^*$ and $\mathbf{c}_1$ gives the desired reduced vector.

5.6 Implementation of the function Select16

The Select function takes an input vector of length 16 and outputs a vector of length 16 whose each component is the same as the one of the selected component of the input vector by an index as described in Section 4.2. This has been implemented using _mm256_permutevar8x32_epi32 as avx2 does not provide a permutation operation on 16-bit __m256i register data, and so, implementations of Select16() and Select8() are different. For Select16(), we use the set of vectors $\{\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_7\}$ where each $\mathbf{s}_i, i = 0, 1, \ldots, 7$ is an 8-dimensional vector with each component i as $\mathbf{s}_i = (i, i, \ldots, i)$. Let $\mathbf{a} = (a_{2\cdot 0}, a_{2\cdot 0+1}, a_{2\cdot 1}, a_{2\cdot 1+1}, \ldots, a_{2\cdot 7}, a_{2\cdot 7+1})$ be the input to Select() function. First we select the i-th pair $(a_{2\cdot i}, a_{2\cdot i+1})$ of $\mathbf{a}$ using the instruction _mm256_permutevar8x32_epi32, and construct the vector $\mathbf{a}_i$ as

$$\begin{aligned} \mathbf{a}_i &= \texttt{\_mm256\_permutevar8x32\_epi32}(\mathbf{a}_i, \mathbf{s}_i) \\ &= (a_{2\cdot i}, a_{2\cdot i+1}, \ldots, a_{2\cdot i}, a_{2\cdot i+1}). \end{aligned}$$

To select the $2i^{th}$ component of $\mathbf{a}$, we first compute the vector $\mathbf{a}_{i,0} = (a_{2\cdot i}, 0, \ldots, a_{2\cdot i}, 0)$ using the instruction _mm256_and_si256 on the vectors $\mathbf{a}_i$ and the vector $(-1, 0, \ldots, -1, 0)$. Applying _mm256_srli_epi32 on $\mathbf{a}_{i,0}$ for 16-bits, we get $\mathbf{a}'_{i,0} = (0, a_{2\cdot i}, \ldots, 0, a_{2\cdot i})$. If we perform _mm256_or_si256 on $\mathbf{a}_{i,0}$ and $\mathbf{a}'_{i,0}$, the resulting vector will be the output of the select function for $2i$. Similarly, we perform the select operation for $(2i+1)^{th}$ components.

5.7 Further Details

Reduction is a costly operation. In the following, we show how the number of reductions can be reduced. In bNTT16 computation of Algorithm 8, multiplications in step 3 are accompanied by a LazyReduced16 and so the components of the vectors $\mathbf{y'}_{i,j}$ are in the range $[-256, +255]$. To compute the vectors $\mathbf{y}_{i,j}$ in step 4, we do not perform reduction for vector addition and subtraction, and subtraction uses $k\mathbf{p}$ for $k = 2^2$ and 2^5. This results in the components of the output vector to be at most 14-bit long. The multiplications $\mathbf{t'}_{i,j} = \mathbf{y}_{i,j} \odot_{257} \mathbf{A'}_{i,j}$ in Algorithm 8 are implemented using the above vector multiplication followed by a Reduced16 operation, resulting in each component of the output vector to be 9-bits and so no further reduction for the additions $\mathbf{x'}_{\mathrm{S},i} = \mathbf{x'}_{\mathrm{S},i} \oplus_{257} \mathbf{t'}_{i,j}$ is needed. This results in vectors whose components are at most 13-bit long. Finally, we use Reduced16 on the vector $\mathbf{x'}_{S,i}$ to produce the final result where components of $\mathbf{x'}_{S,i}$ are all at most 9 bits.

These optimizations reduce the number of reduction operations. Figures 2, 3, 4 and 5 show that the total number of vector operations that are required for steps 3 and 4 of 16-way parallelization of NTT is significantly smaller than the number of operations required for 8-way parallelization, and this leads to a substantial speedup of the implementation of SWIFFT-16-avx2.

5.8 Implementation of Cover-Free-Family

KSN uses a 1-CFF which is obtained by taking all subsets of size $t/2$ from a set of size t and so one can use a $\log_2 \binom{t}{t/2}$ bit message space. Each subset corresponds to a particular message. For efficient encoding of messages however, we use the algorithm proposed in [8, 12, 40].

5.9 Implementation of Pseudo-Random Function

We use ChaCha20 [6] as the pseudo-random function family F_n. ChaCha20 is an state-of-the-art stream-cipher which we use to generate the seed of each KSN-OTS from secret-key of K2SN-MSS, and all the component secret keys of the OTS instances. We use the avx2-based implementation of ChaCha20 from supercop [2][1].

6 EXPERIMENTS

We used the following platform for our implementations and experiments:

Skylake: Intel®Core™i7-6700 4-core CPU @ 3.40GHz running. The timing experiments are performed on a single core. The OS is 64-bit Ubuntu-18.04 LTS and C codes were compiled by GCC version 7.3.0. The code of the software is available at [23]. During the experiments, the turbo boost and hyper-threading were turned off. The cache warm-up was done by 25000 iterations and the measurements are taken as the median over 100000 iterations.

6.1 Performance Comparison of SWIFFT Implementations

We have implemented all the parallelized version of the SWIFFT function using 16-bit avx2 intrinsic instructions. Each element of $\mathbb{Z}_{257}$ is represented by 9-bit. The SWIFFT evaluations were computed over 1024-bit data blocks. The generalized SWIFFT function uses 9×1024 bit data blocks as input. The output in both cases is a $9 \times 64 = 576$ bits string. For binary versions of SWIFFT-8-avx2 and SWIFFT-16-avx2, we require approximately 8KB and 2MB memory for the precomputation tables, respectively. The Time Stamp Counter (TSC) was read from the CPU to RAX and RDX registers using RDTSC instruction. All the experimental results are listed in Table 7. The results show that SWIFFT-8-avx2 and SWIFFT-16-avx2 are approximately 8% and 25% faster than the previous implementation in [5, 28]. Based on these results, we use SWIFFT-16-avx2 for K2SN-MSS.

Function Name	Intrinsic Flag	Binary Version		Generalized Version	
		Total clk	clk/byte	Total Clk	clk/byte
SWIFFT-8-sse2 [31]	mavx2	1150	8.98	-	-
SWIFFT-8-avx2 [this paper]	mavx2	1047	8.17	11435	9.93
SWIFFT-16-avx2 [this paper]	mavx2	866	6.77	9535	8.27

Table 7: Required clock cycles (clk) for various SWIFFT implementation.

6.2 Performance Comparison of KSN and W-OTS$^+$

KSN-OTS and W-OTS$^+$ are the OTS that are used in K2SN-MSS and XMSS respectively. We compare both of them at $n = 512$. For the W-OTS$^+$, we used the code for XMSS [19], available at [1, 3]. The Time Stamp Counter reading did not work with the XMSS code of [1, 3]. We instead used clock() function of "time.h" header file. Our measurement is the average over 1,000,000 iterations. The results of performance comparison of KSN-OTS and W-OTS$^+$ are in Table 8 and it shows that, the key generation of KSN is approximately 22 times faster than that of W-OTS$^+$, while signing and verification are approximately 23 and 167 times faster than those of W-OTS$^+$. This performance is due to the simple signing operation (generation of component secret keys and component-wise vector addition) and efficient implementation of gSWIFFT that is used in the verification operation of KSN-OTS.

	KSN/SWIFFT-16-avx2	W-OTS$^+$/SHA512/$w = 16$
Key Generation (μs)	164	3575
Signing (μs)	83	1872
Verification (μs)	10	1674
Secret Key/ (Bytes)	40	64
Public Key (Bytes)	4608	4352
Signature Sizes (Bytes)	1024	4288

Table 8: Performance Comparison of KSN and W-OTS$^+$ in μs

Remark: We compare our implemented K2SN-MSS software against the results of [15], but the code of [15] is not public.[2] [15] reports the fastest result for XMSS.

6.3 Performance Comparison between K2SN-MSS and XMSS

The timing measurements are done the same as in Section 6.2. We compare our implemented software against the XMSS software results available at [15]. We used the SWIFFT-16-avx2 implementation of K2SN-MSS. We compute the authentication path using the TREEHASH algorithm of [10]. The results of the performance comparisons are in Table 1 and it shows that the key generation, signing and verification of K2SN-MSS are 2.76 times, 2.89 times and 2.65 times faster than the corresponding operations in XMSS [15], respectively.

Remarks:

(1) The signature size of K2SN-MSS is comparable with the signature sizes of the XMSS^{MT} [20] and the SPHINCS [7], but we can not compare K2SN against XMSS^{MT} and SPHINCS

[1] The software is implemented by D. J. Bernstein and R. Dolbeau and available in the directory "supercop-20171218/crypto_stream/chacha20/dolbeau/amd64-avx2"

[2] We tried to communicate the authors, but did not receive any reply yet.

because the use of the Merkle tree is different in the two cases. In K2SN-MSS and XMSS (single tree version of XMSS^{MT}) use only one layer of Merkle tree where XMSS^{MT} and SPHINCS uses multiple layers of Merkle tree.

(2) Even so, if we want to compare K2SN-MSS against the XMSS^{MT} and SPHINCS, we need optimized software of them for $n = 512$. But in the literature, we could not find a single implementation of XMSS^{MT} and SPHINCS which provides 512-bit classical (256-bit quantum) security. Thus we are unable to compare them.

(3) Our comparison is against the multi-buffer entries of the third row of TABLE IV of [15]. It is due to the following reasons:

   (a) this is the only reported results for SHA512, where SWIFFT output is 576 bits. This comparison is fair because of the comparative sizes of the used hash functions.

   (b) The experimental platform of [15] was
   Skylake: Intel®Core™i7-6700 4-core CPU @ 4.0GHz
   which is a faster machine than the machine used in our experiments. Therefore, we believe that the comparisons are made in Table 1 are valid and fair.

7 CONCLUDING REMARKS

Hash-based signatures are a strong alternative for post-quantum signatures. We extended KSN-OTS for signing multiple messages which is secure under multi-function multi-target attacks and gave an efficient implementation using parallelization at the instruction level, using a widely accessible technology of `Intel`. Our implementation also improves the state-of-the-art implementation of SWIFFT and provides parallelization of NTT computation, both of independent interest. Although our results are for concrete levels of parallelization, but by providing implementation details we provide a template for other parameters, for binary and non-binary input vectors, and with or without precomputation. Our implementation shows that K2SN-MSS is significantly faster than XMSS, which is recently proposed as a candidate for standardization.

In our implementation we used SWIFFT both for KSN-OTS, and the Merkle hash tree. Although SWIFFT is essential in KSN-OTS, the Merkle tree is used to authenticate the public key $\mathcal{PK}_i$, against the root of the $\mathcal{MSS}$ tree and we are not restricted to SWIFFT for the hash function. Thus, for this tree one can use traditional hash functions such as SHA512. Our choice of SWIFFT hash function family for the construction of $\mathcal{L}$ trees and $\mathcal{MSS}$ has the following advantages. Firstly, it reduces the complexity of the code: using two different hash function families will increase the code complexity and size. Secondly, SWIFFT has (asymptotic) provable security and this improves confidence in the security of the design. Finally, SHA512 hash function family follows Merkle-Damgård construction [39] which is inherently a sequential construction and so not easily amenable to parallelization. SWIFFT function family, however, is highly parallelizable, and can benefit from processor architecture with longer registers to achieve higher speed.

REFERENCES

[1] [n. d.]. Hülsing. https://huelsing.wordpress.com/code/.

[2] [n. d.]. Supercop: Version 2017.12.18. https://bench.cr.yp.to/supercop.html.

[3] [n. d.]. xmss-reference. https://github.com/joostrijneveld/xmss-reference.

[4] 2018. The Internet Engineering Task Force. https://tools.ietf.org/html/rfc8391.

[5] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. 2008. SWIFFTX: A Proposal for the SHA-3 Standard. https://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifftx.pdfl.

[6] D. J. Bernstein. 2008. ChaCha, a variant of Salsa20.

[7] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, P. Schwabe, and Z. W. O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology – EUROCRYPT (Lecture Notes in Computer Science)*, Vol. 9056. Springer, 368–397.

[8] K. Bicakci, G. Tsudik, and B. Tung. 2003. How to construct optimal one-time signatures. *Journal of Computer Networks* 43, 3 (2003), 339–349.

[9] J. Buchmann, E. Dahmen, and A. Hülsing. 2011. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In *Post-Quantum Cryptography – PQCrypto (Lecture Notes in Computer Science)*, Vol. 7071. Springer, 117–129.

[10] J. Buchmann, E. Dahmen, and M. Schneider. 2008. Merkle Tree Traversal Revisited. In *Post-Quantum Cryptography - PQCrypto (Lecture Notes in Computer Science)*, Vol. 5299. Springer, 63–78.

[11] N. Courtois, M. Finiasz, and N. Sendrier. 2001. How to Achieve a McEliece-based Digital Signature Scheme. In *Advances in Cryptology – ASIACRYPT (Lecture Notes in Computer Science)*, Vol. 2248. Springer, 157–174.

[12] T. Cover. 1973. Enumerative Source Encoding. *IEEE Transactions on Information Theory* 19, 1 (1973), 73–77.

[13] E. Dahmen, K. Okeya, T. Takagi, and C. Vuillaume. 2008. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In *Post-Quantum Cryptography – PQCrypto (Lecture Notes in Computer Science)*, Vol. 5299. Springer, 109–123.

[14] A. Das and C. E. Veni Madhavan. 2009. *Public-Key Cryptography: Theory and Practice.* Pearson.

[15] A. K. D. S. de Oliveira, J. LÂťopez, and R. Cabral. 2017. High Performance of Hash-based Signature Schemes. *International Journal of Advanced Computer Science and Applications* 8, 3 (2017).

[16] W. Diffie and M. E. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654.

[17] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *Advances in Cryptology – CRYPTO (Lecture Notes in Computer Science)*, Vol. 8042. Springer, 40–56.

[18] A. Fog. 2016. Software optimization resources. http://agner.org/optimize/.

[19] A. Hülsing, D. Butin, S.-L. Gazdag, and A. Mohaisen. 2017. XMSS: Extended Hash-Based Signatures. draft-irtf-cfrg-xmss-hash-based-signatures, work in progress.

[20] A. Hülsing, L. Rausch, and J. Buchmann. 2013. Optimal Parameters for XMSS^{MT}. In *Security Engineering and Intelligence Informatics: CD-ARES (Lecture Notes in Computer Science)*, Vol. 8128. Springer, 194–208.

[21] A. Hülsing, J. Rijneveld, and F. Song. 2016. Mitigating Multi-target Attacks in Hash-Based Signatures. In *Public Key Cryptography – PKC (Lecture Notes in Computer Science)*, Vol. 9614. Springer, 387–416.

[22] Intel. [n. d.]. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#.

[23] K2SN-MSS. [n. d.]. https://github.com/anon1985/K2SN-MSS.

[24] K. Kalach and R. Safavi-Naini. 2016. An Efficient Post-Quantum One-Time Signature Scheme. In *Selected Areas in Cryptography – SAC (Lecture Notes in Computer Science)*, Vol. 9566. Springer, 331–351.

[25] L. Lamport. 1979. Constructing Digital Signatures from a One Way Function. technical report of SRI International.

[26] P. Longa and M. Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security – CANS (Lecture Notes in Computer Science)*, Vol. 10052. Springer, 124–139.

[27] V. Lyubashevsky and D. Micciancio. 2006. Generalized Compact Knapsacks Are Collision Resistant. In *International Colloquium on Automata, Languages, and Programming – ICALP (Lecture Notes in Computer Science)*, Vol. 4052. Springer, 144–155.

[28] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. 2008. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption – FSE (Lecture Notes in Computer Science)*, Vol. 5086. Springer, 54–72.

[29] R. C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology – CRYPTO (Lecture Notes in Computer Science)*, Vol. 293. Springer, 369–378.

[30] R. C. Merkle. 1989. A Certified Digital Signature. In *Advances in Cryptology – CRYPTO (Lecture Notes in Computer Science)*, Vol. 435. Springer, 218–238.

[31] D. Micciancio. [n. d.]. https://github.com/micciancio/SWIFFT.

[32] D. Micciancio. 2007. Generalized Compact Knapsacks, Cyclic Lattices, and Efficient One-Way Functions. *Computational Complexity* 16, 4 (2007), 365–411.

[33] D. Moody. [n. d.]. Stateful hash-based signatures.

[34] NIST. 2017. Post-Quantum Cryptography - Call for Proposals. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals.

[35] C. Peikert and A. Rosen. 2006. Efficient Collision-Resistant Hashing from Worst-Case Assumptions on Cyclic Lattices. In *Theory of Cryptography Conference –*

TCC (Lecture Notes in Computer Science), Vol. 3876. Springer, 145–166.

[36] J. Rompel. 1990. One-way functions are necessary and sufficient for secure signatures. In *ACM symposium on Theory of computing – STOC*. ACM, 387–394.

[37] P. W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134.

[38] T. Simonite. [n. d.]. MIT Technology Review.

[39] W. Sun, H. Guo, H. He, and Z. Dai. 2007. Design and optimized implementation of the SHA-2(256, 384, 512) hash algorithms. In *International Conference on ASIC*. IEEE, 858–861.

[40] G. M. Zaverucha and D. R. Stinson. 2011. Short one-time signatures. *Advances in Mathematics of Communications–AMC* 5, 3 (2011), 473–488.