An Overview of Hash Based Signatures

Vikas Srivastava[1*], Anubhab Baksi[2], and Sumit Kumar Debnath[1]

[1] National Institute of Technology Jamshedpur, India

[2] Nanyang Technological University, Singapore

2020rsma011@nitjsr.ac.in, anubhab001@e.ntu.edu.sg, sdebnath.math@nitjsr.ac.in

Abstract. Digital signatures are one of the most basic cryptographic building blocks which are utilized to provide attractive security features like authenticity, unforgeability, and undeniability. The security of existing state of the art digital signatures is based on hardness of number theoretic hardness assumptions like discrete logarithm and integer factorization. However, these hard problems are insecure and face a threat in the quantum world. In particular, quantum algorithms like Shor's algorithm can be used to solve the above mentioned hardness problem in polynomial time. As an alternative, a new direction of research called post-quantum cryptography (PQC) is supposed to provide a new generation of quantum-resistant digital signatures. Hash based signature is one such candidate to provide post-quantum secure digital signatures. Hash based signature schemes are a type of digital signature scheme that use hash functions as their central building block. They are efficient, flexible, and can be used in a variety of applications. In this document, we provide an overview of the hash based signatures. Our presentation of the topic covers a wide range of aspects that are not only comprehensible for readers without expertise in the subject matter, but also serve as a valuable resource for experts seeking reference material.

Keywords: Hash Based Cryptography · Digital Signature · Post-quantum Cryptography

1 Introduction

Post-quantum signatures are cryptographic signatures that are designed to be secure against attacks from quantum computers. Quantum computers have the potential to break many of the cryptographic schemes that are currently in use, including the signature schemes that are used to verify the authenticity of digital documents and transactions. Quantum computers can use a technique called Shor's algorithm to quickly factor large numbers, which is the basis of many cryptographic systems. This means that the security of many current signature schemes will be compromised once practical quantum computers are built. Post-quantum signature schemes are designed to be secure even against attacks from quantum computers. They use mathematical problems that are believed to be hard even for quantum computers to solve, such as the problem of finding short vectors in lattices or the problem of solving multivariate polynomials.

As the development of quantum computers continues to progress, the need for post-quantum signatures becomes increasingly important to ensure the security of digital communications and transactions. There are several categories of post-quantum signature schemes, each based on a different mathematical problem or approach:

1. **Hash based signatures:** These schemes are based on the use of one-way hash functions, which are believed to be secure even against quantum computers. Examples include the Merkle signature scheme [17], XMSS^{MT} [12], and Sphincs [2].
2. **Lattice based signatures:** These schemes are based on the hardness of certain lattice problems, which are believed to be hard for both classical and quantum computers. Examples include the FALCON [8] and Dilithium[7] signature schemes.

* Vikas Srivastava would like to acknowledge the support from International Mathematical Union (IMU) and the Graduate Assistantships in Developing Countries (GRAID) Program.

3. **Code based signatures:** These schemes are based on the hardness of certain coding problems, which are believed to be hard for quantum computers. Example include the Niederreiter signature scheme [18].
4. **Multivariate polynomial based signatures:** These schemes are based on the difficulty of solving systems of multivariate polynomials, which are believed to be hard for both classical and quantum computers. Examples include the Rainbow[6] and HFE signature scheme [19].
5. **Isogeny based signatures:** These schemes are based on the use of isogenies, which are maps between elliptic curves that are believed to be hard to compute for quantum computers. The CSI-FiSh signature scheme [4] is one such example of isogeny-based signature.

Each of these categories has its own strengths and weaknesses, and the choice of which post-quantum signature scheme to use will depend on the specific requirements and constraints of the application. Among the post-quantum signature candidates, Hash based signature is a promising candidate to provide secure digital signatures. They rely on the properties of one-way hash functions, which are believed to be secure against both classical and quantum computers. Hash based signatures are also attractive because they are simple, fast, and efficient. They can be implemented with relatively small key sizes and require minimal computation for both signing and verification.

The study of hash functions is already a crucial aspect of cryptography, and there have been various methods and studies aimed at achieving different security properties. Unlike digital signatures based on difficult number theory problems, if a hash function is attacked, it won't compromise the overall security of a hash based signature. The attacked hash function can be replaced by a secure one easily. Hash functions have been refined and efficiently implemented over several decades, making hash based signatures highly efficient. Furthermore, by selecting different underlying hash functions and their parameters, one can balance signature size, time, and storage to cater to diverse application needs. Particularly, given the symmetric key primitives are considered secure against a quantum adversary (due to infeasible resource requirement; see, e.g., [13]), it makes intuitive sense to use those primitives to design asymmetric key alternatives. A chronology for the hash based signatures is given in Figure 1.

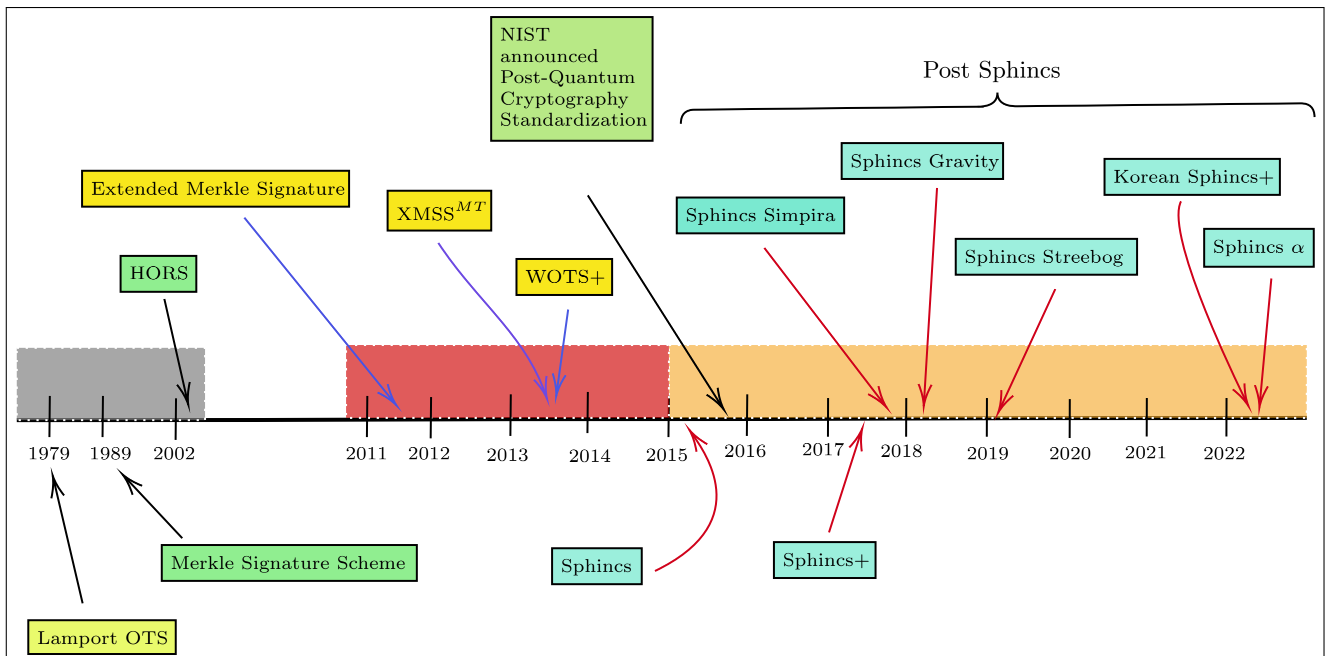

Fig. 1. Timeline of hash based signatures

2 Background and Motivation

In simple terms, a digital signature scheme provides a cryptographic version of a handwritten signature that offers much stronger security. These signatures are widely accepted as legally binding in many countries and can be used to certify contracts, notarize documents, authenticate individuals or corporations. Digital signatures are also essential for the secure distribution and transmission of public keys, which is the foundation of public-key cryptography. A digital signature scheme is a type of public key cryptography where a user, also known as the signer, creates a pair of keys: a private key and a public key. The user keeps the private key secret and can use it to generate a digital signature for any message. The process of creating a digital signature involves signer signing the message using his private key. Only signer can sign a message on his behalf assuming he keeps his private key secure. The signature algorithm takes both the private key and the message as input. Once signer has signed the message, he appends the resulting signature σ to m and sends the pair (σ, m) to the verifier. It is important to note that a digital signature is meaningless without the accompanying message. Anyone who has access to the public key can verify the authenticity of the message and the associated digital signature.

A signature scheme is a tuple of algorithms:$(\mathsf{Kg}, \mathsf{Sign}, \mathsf{Ver})$ along with an associated message space.

$(pk, sk) \leftarrow \mathsf{Kg}(\kappa)$: Given as input the security parameter κ, the key generation algorithm outputs a pair of keys (pk, sk). Here, pk is called the public key or the verification key, and sk is known as the private key, the secret key, or the signing key.

$\sigma \leftarrow \mathsf{Sign}\ (m, sk)$: Given a message m and a secret key sk as input, the algorithm Sign outputs a signature σ associated with the message m.

$0/1 \leftarrow \mathsf{Ver}(m, \sigma, pk)$: Given as input pk, and a message signature pair $(m,\ \sigma)$, Ver outputs a single bit b. Here $b = 1$ signifies 'accept' and $b = 0$ signifies 'reject'.

The digital signatures provides security features like message integrity, authenticity and non-reputation. As an example of a digital signature scheme, we present the the RSA signature scheme [22]. It is founded on the concept of number theoretic hardness, specifically on the challenge of factoring a product of two large prime numbers (which is called the integer factorization problem) in order to maintain its security. It has been around since 1978 and since then, it has become the most frequently used digital signature scheme in practical applications. The RSA signature scheme comprises three algorithms: $\mathsf{RSA.Kg}$, $\mathsf{RSA.Sign}$, and $\mathsf{RSA.Ver}$. Refer to Algorithms 1, 2, and 3 for the exact description.

Algorithm 1 Key generation algorithm of RSA signature, $\mathsf{RSA.Kg}$

Output: public key: (n, e) and private key: $sk = (d)$

1: Choose two large primes p and q.
2: Compute $n = p \times q$.
3: Compute $\phi(n) = (p-1)(q-1)$.
4: Select the public exponent $e \in \{1, 2, ..., \phi(n) - 1\}$ such that $gcd(e, \phi(n)) = 1$.
5: Compute the private key d such that $de \equiv \mod \phi(n)$

Algorithm 2 Signature algorithm $\mathsf{RSA.Sign}$

Input: sk, message m

Output: : signature σ on the message m

1: Computes $\sigma = m^d \mod n$
2: Return σ

Algorithm 3 Verification algorithm RSA.Ver

Input: pk, m, σ

Output: : 0 or 1

1: Computes $m' = \sigma^e \mod n$

2: If $m' = m$, then declare the signature message pair (σ, m) as valid and outputs 1; otherwise, reject the signature and output 0.

A toy example is given here for better clarity:

RSA.Kg : We take $p = 3$ and $q = 11$. Compute $n = p.q = 33$. In the following, computes $\phi(n) = (3-1).(11-1) = 20$. Choose $e = 3$. Computes $d \equiv e^{-1} \mod 20 \equiv 7$. We set $pk = (33, 3)$

RSA.Sign : Let $m = 4$ be the message we want to sign. Signer computes $\sigma = m^d = 4^7 = 16 \mod 33$. The signer produces $(4, 16)$ as the message signature pair.

RSA.Ver : Verifier given m, σ and pk computes $m' = \sigma^e \mod n = 16^3 \mod n = 4$. Note that $m' = m$, therefore the signature-message pair (σ, m) is valid.

The present-day techniques of digital signature, such as RSA, DSA, and ECDSA, count on the safety of specific one-way functions with trapdoors. These functions are linked to the difficulty of factorizing integers and computing discrete logarithms. However, it is uncertain whether these problems will continue to be hard in the future as it has been demonstrated that quantum computers can solve them quickly. Therefore, it is crucial to develop alternative digital signature schemes that provide maximum security and are resistant to quantum computers. These new schemes are referred to as post-quantum signature schemes and must be designed to prevent quantum computers from being able to break them to ensure the importance of digital signatures. Some candidate for post-quantum signatures are: lattice-based signatures, multivariate-based signatures, isogeny-based signatures, code-based signatures, and hash based signatures. Hash based signature is one such post-quantum cryptography candidate. Hash based signature schemes are a type of digital signature scheme that use hash functions as their central building block. They are efficient and flexible, and can be used in a variety of applications. Hash functions are a crucial component in modern cryptography, providing a range of security properties that are essential for protecting sensitive data and ensuring the security of cryptographic applications. Their efficiency and flexibility make them an ideal choice for many different applications, and ongoing research and development will continue to improve their security and performance in the years to come. Hash based signature is a very old area of research. In the late 1970s, Leslie Lamport invented hash based signature schemes, which were then improved upon by Ralph Merkle and others. For a long time, these schemes were not given much attention by the cryptographic community, partly because they generate relatively large signatures and have other complexities. However, in recent years, there has been renewed interest in these schemes, largely because they are believed to be resistant to significant quantum attacks like Shor's algorithm. This is unlike signature schemes based on RSA or the discrete logarithm assumption, which are vulnerable to such attacks.

Hash functions are mathematical functions that take input data of arbitrary size and produce a fixed-length output, known as a hash value or digest. They are widely used in various applications, including data integrity verification, message authentication, and digital signature schemes. One of the key features of hash functions is their one-wayness property, which means that it is computationally infeasible to recover the input data from the hash value. This property makes hash functions suitable for protecting the integrity of data and detecting any changes or tampering. Another important property of hash functions is collision resistance, which means that it is computationally infeasible to find two different inputs that produce the same hash value. This property is essential for ensuring the security of many cryptographic applications, including digital signatures. In recent years, there have been many attacks on various hash functions, leading to the development of new hash functions and

the retirement of old ones. Some popular hash functions include SHA-256, SHA-3, and BLAKE-2. We now introduce the first hash based signature scheme.

3 Lamport OTS

Lamport in 1979 designed the first signature scheme [16] solely based on hash functions. Let us assume that we possess a hash function that can handle 256-bit inputs and generate 256-bit outputs, such as SHA-256. Suppose we aim to sign messages of 256 bits. To create our private key, our initial step is to produce 512 individual random bit sequences, each 256 bits long. We will organize these bit-strings into two separate lists and label each with an index.

$$sk_0 = sk_1^0, sk_2^0, \ldots, sk_{256}^0$$
$$sk_1 = sk_1^1, sk_2^1, \ldots, sk_{256}^1$$

The lists (sk_0, sk_1) represent the *secret* key. The public key pk is generated by *hash* of every one of those random strings. The hash function H is used for this purpose. The public key is given by:

$$pk_0 = H(sk_1^0), H(sk_2^0), \ldots, H(sk_{256}^0)$$
$$pk_1 = H(sk_1^1), H(sk_2^1), \ldots, H(sk_{256}^1)$$

To sign a 256-bit message with sk, we proceed as follows.

1. Represent m as a sequence of 256 individual bits:
$$m = m_1, \ldots, m_{256}, \quad m_i \in \{0, 1\}$$
2. For $i = 1$ to 256: If the ith message bit $M_i = 0$, take the ith private string (sk_i^0) from the sk_0, and output that string as part of our signature.
3. If the message bit $M_i = 1$, we take the appropriate string (sk_i^1) from the sk_1 list.
4. Concatenate all the strings together to output the signature

A toy example for the signature generation is given in Figure 2. We now discuss the verification algorithm for L-OTS. Refer to Figure 3) for an illustrated example. Given a message signature pair (m, σ) and the public key $pk = (pk_0, pk_1)$, a verifier proceeds in the following way:

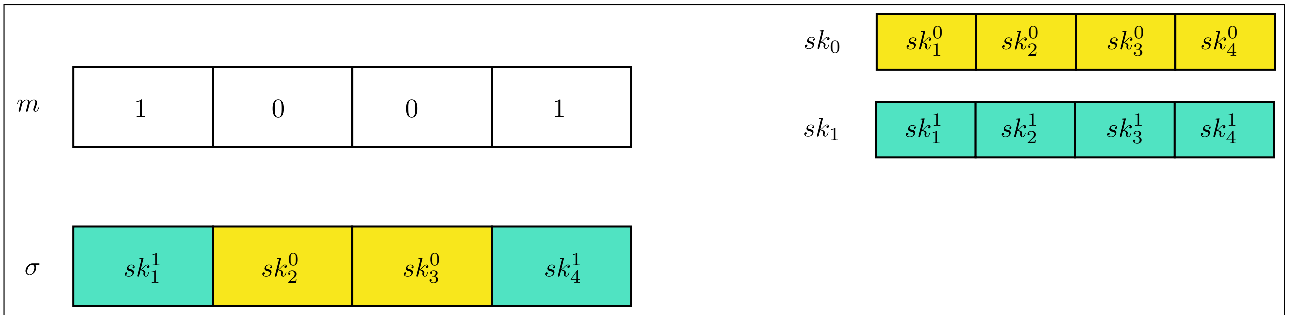

Fig. 2. Signature Generation of L-OTS for the message $m = 1001$ with secret key $sk = (sk_0, sk_1)$

1. Let σ_i denotes the ith component of σ.
2. For each $i \in \{1, 256\}$, the verifier considers the message-bit m_i, and calculate $H(\sigma_i)$.
3. If $M_i = 0$, the $H(\sigma_i)$ should be equal to the corresponding element from pk_0. If $M_i = 1$, $H(\sigma_i)$ should be equal to the corresponding element in pk_1.
4. Signature is declared valid if every component of the signature, when hashed, matches the correct portion of the pk

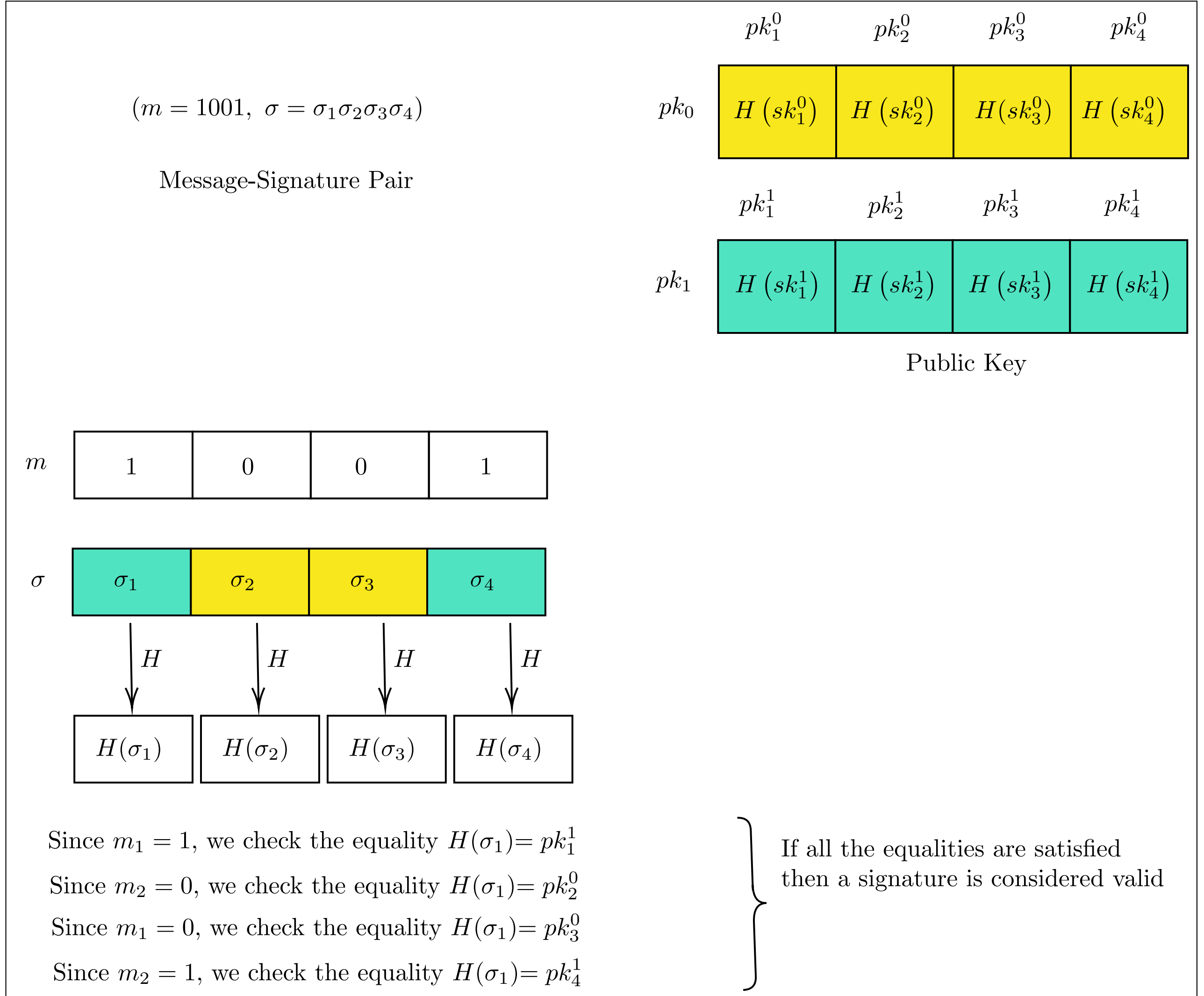

Fig. 3. Signature Verification of L-OTS for the message $m = 1001$ with public key $pk = (pk_0, pk_1)$

4 WOTS+

4.1 Preliminaries

In contrast to L-OTS, which generates private and public key pairs for each bit of a message, Winternitz OTS (W-OTS+) [11,2] divides a hashed message into segments. The W-OTS+ approach aims to reduce the size of signatures and key pairs, albeit at the cost of additional hash evaluations. Specifically, W-OTS+ first converts the message m into a new form using a base w representation, and then breaks it down into blocks of length $\log w$. For each block, it applies a 'function' up to a maximum of $w-1$ times, and the output of the 'function' becomes the signature for that block. The resulting signatures for each block are concatenated in sequence to form the entire signature for m.

Parameters and Functions: A cryptographic hash function F defined as

$$F : \{0,1\}^n \rightarrow \{0,1\}^n.$$

We also need a a family of pseudo-random generators G_λ defined by

$$G_\lambda : \{0,1\}^n \rightarrow \{0,1\}^{\lambda n}$$

for different values of λ. We fix the message length to be n. Given n and w, we define $l_1 = \lceil \frac{n}{log(w)} \rceil$ and $l_2 = \lfloor \frac{\log(l_1(w-1))}{\log w} \rfloor + 1$. We further define $l = l_1 + l_2$.

Chaining function $c^i(x,r)$: Given a input value $x \in \{0,1\}^n$, a iterative counter $i \in \mathbb{N}$, and bitmask $r = (r_1, \ldots, r_j) \in \{0,1\}^{n \times j}$ ($j \geq i$), the chain function works as follows:

1. If $i = 0$, $c^0(x,r) = x$.
2. If $i \geq 0$, $c^i(x,r) = F(c^{i-1}(x,r) \oplus r_i)$.

Notation: $r_{a,b}$ denotes the substring $(r_a, \ldots, r_b)$ of r. If b is less than a, $r_{a,b}$ is set to be empty string. Figure 4 summarizes the chain function used in WOTS+.

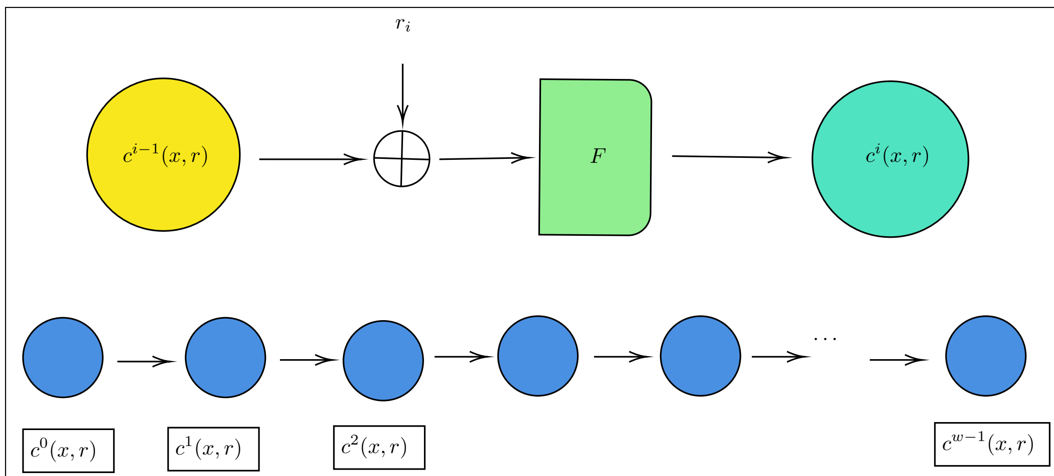

Fig. 4. Chain function used in WOTS+

4.2 Protocol Description

$(sk, pk) \leftarrow$WOTS+.Kg(S, r)**:** Given on input a seed $S \in \{0,1\}^n$ and bitmasks $r = (r_1, \ldots, r_{w-1}) \in \{0,1\}^{n\times(w-1)}$, WOTS+.Kg outputs the secret key and public key by going through following steps:

1. The secret key is computed using the seed S by employing the pseudo random generator G_λ

$$sk = (sk_1, \ldots, sk_l) \leftarrow G_l(S)$$

2. The public key pk is computed by using the bitmask r and the chain function defined above:

$$pk = (pk_1, \ldots, pk_l) = (c^{w-1}(sk_1, r), \ldots, c^{w-1}(sk_l, r)).$$

WOTS+.Sig**:** On input of an n-bit message m, seed S and the bitmasks r, the algorithm WOTS+.Sig first computes a base-w representation of m. In other words, we consider $m = (m_1, \ldots, m_{l_1})$, $m_i \in \{0, \ldots, w-1\}$. In the following, the checksum $C = \sum_{i=1}^{l-1}(w-1-M_i)$ is also represented in base w representation. Let $C = (C_1, \ldots, C_{l_2})$. It is of the length atmost l_2. We append both m and C to get $b = m||C$, i.e., $b = (b_1, \ldots, b_l)$. The signature of m is

$$(\sigma_1, \ldots, \sigma_l) = (c^{b_1}(sk_1, r), \ldots, c^{b_l}(sk_l, r))$$

. Refer to Figure 6 for an illustrated workflow of the signature generation in W-OTS+.

WOTS+.Ver**:** Signature is verified by constructing $(b_1, \ldots, b_l)$ and checking

$$(pk'_1, \ldots, pk'_l) \stackrel{?}{=} (c^{w-1-b_1}(\sigma_1, r_{b_1+1,w-1}), \ldots, c^{w-1-b_l}(\sigma_1, r_{b_l+1,w-1}))$$

The public key generation in W-OTS+ is summarized in Figure 5. In addition, we have given a toy example for W-OTS+ in Figure 7.

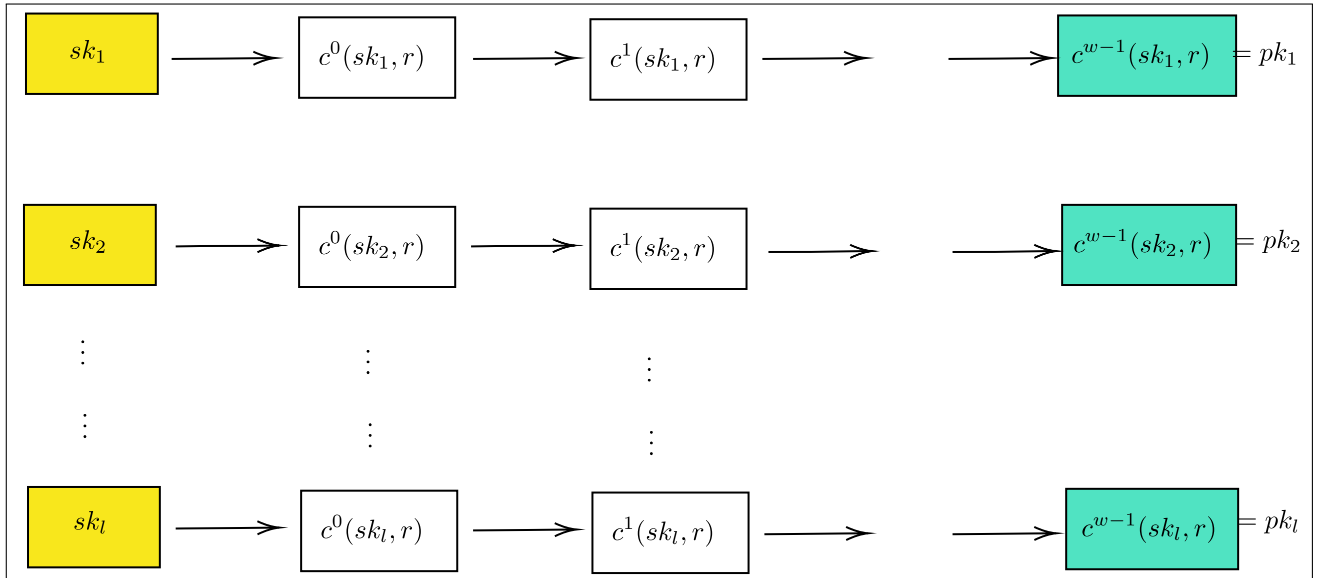

Fig. 5. Key generation in W-OTS+

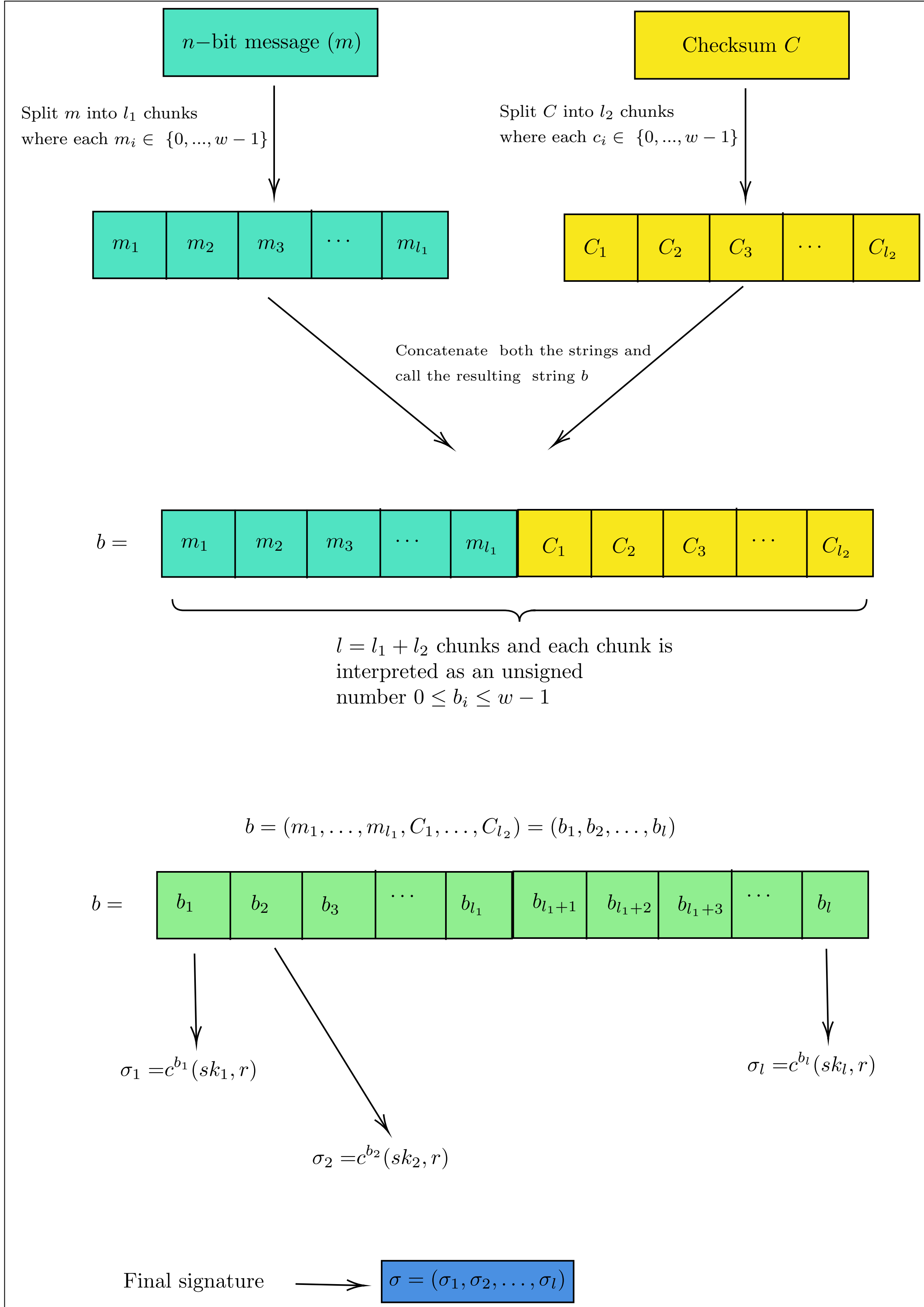

Fig. 6. Signature generation in W-OTS+

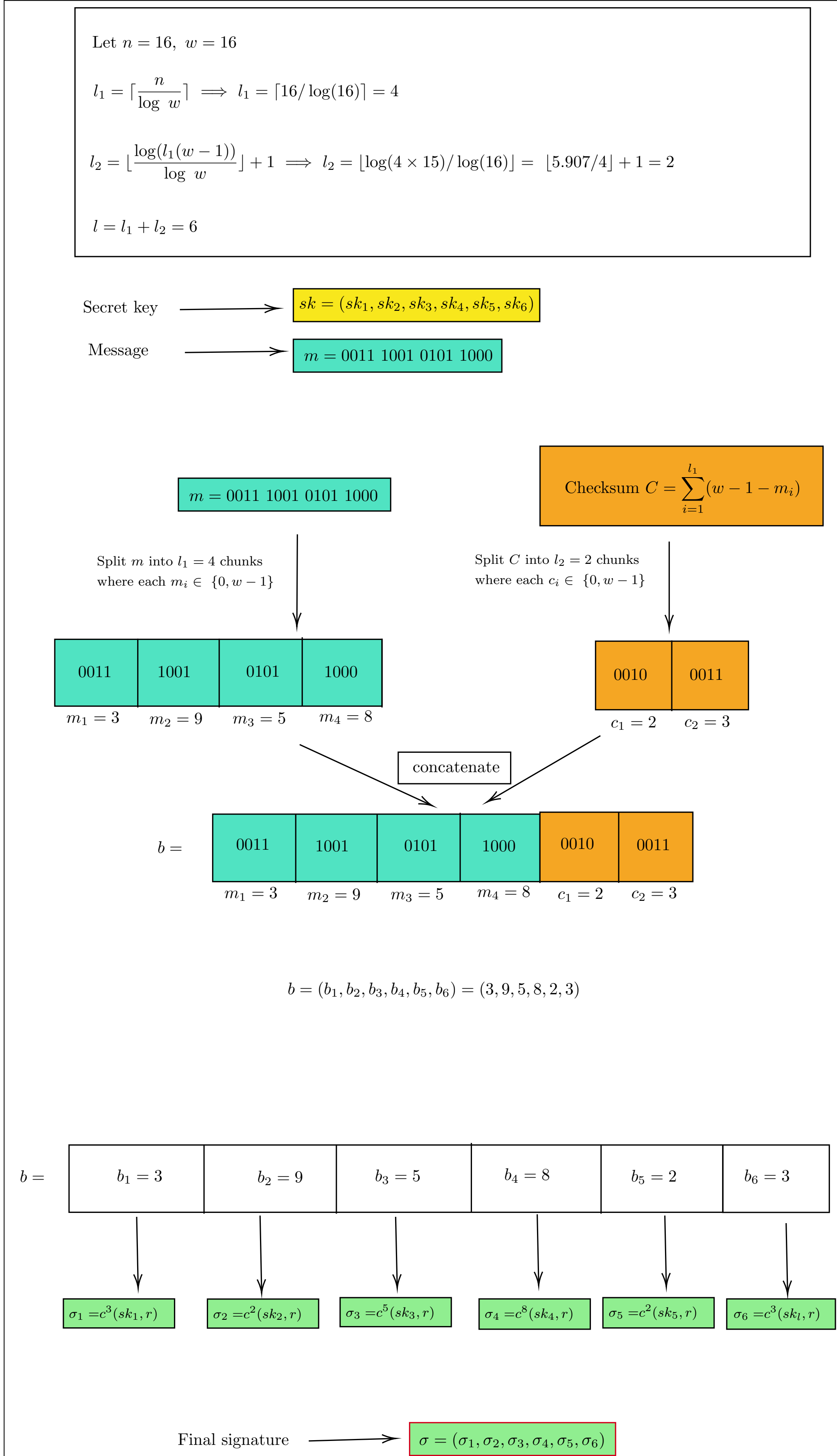

Fig. 7. A toy example for W-OTS+ signature generation with $n = 16, w = 16$

5 Merkle Signature Scheme (MSS)

Motivation and Overview: Because L-OTS and W-OTS+ can only sign one message at a time, if multiple messages need to be signed, there must be many keys. The Merkle Tree Signature Scheme (commonly called "MSS") [17] was invented by Ralph Merkle to manage OTS keys.

The basic idea is to use Merkle tree leaves to store OTS keys. Because the Merkle tress are binary trees, a Merkle tree of height h has 2^h leaves. The leaves are used to manage digests of OTS keys, and a Merkle tree can manage 2^h OTS key pairs. When signing a message, one OTS key pair needs to be picked up from the tree that has not been used before. The signature consists of the index of the leaf, the OTS public key, the digest of the OTS public key (the leaf), and the authentication path of that leaf.

5.1 Protocol Description

MSS.Kg: We now describe the key generation algorithm of the MSS. Refer to Figure 8 for an illustrated diagram.

1. Generate $N = 2^n$ public key-private key pairs

$$(OTS_{PK_0}, OTS_{SK_0}), \ldots (OTS_{PK_{N-1}}, OTS_{SK_{N-1}})$$

of some OTS scheme.

2. For each $i \in \{0, 2^n - 1\}$, compute $h_i = H(OTS_{PK_i})$
3. The hash values h_i are placed as leaves and hashed recursively to form a binary tree. The details are presented below.
   (a) Let $a_{i,j}$ denote the node in the tree with height i, and left-right position j.
   (b) The hash values $h_i = a_{0,i}$ are the leaves.
   (c) The inner nodes are calculated by

$$a_{i,j} = H(a_{i-1,2j} || a_{i-1,2j+1})$$

   where $i = 1, \ldots, h$ and $j = 0, \ldots, 2^{i-1}$.
   (d) The public key pub is the root of the tree $a_{n,0}$
4. The private key of the Merkle signature scheme is the entire set of (OTS_{PK_i}, OTS_{SK_i}) pairs

MSS.Sig: 1. The signer selects the ith public key say OTS_{PK_i} from the tree, and signs the message using the corresponding OTS secret key OTS_{SK_i} resulting in a signature σ_{OTS_i}

2. In addition, the final signature consists of the index of the leaf, the OTS public key, and the authentication path of that leaf.

$$\sigma = (i, OTS_{PK_i}, \sigma_{OTS_i}, Auth_i)$$

MSS.Ver: Given σ and m, verifier proceeds to verify the message-signature pair by going through following steps. Note that verifier already knows the master public key pub.

1. First, the receiver verifies the one-time signature σ_{OTS_i} of the message m using the one-time signature public key OTS_{PK_i}. If σ_{OTS_i} is a valid signature on the message m, then proceeds further, otherwise aborts the process and rejects the signature.
2. Verifier computes $a_{0,i} = H(OTS_{PK_i})$ by hashing the public key of the OTS.
3. Using the authentication path $Auth_i$ computes the root of the Merkle tree (say pub'. If pub'=pub, then the verifier declares the signature as valid otherwise rejects.

Refer to Figure 9 for an illustrative toy example depicting signature generation of MSS.

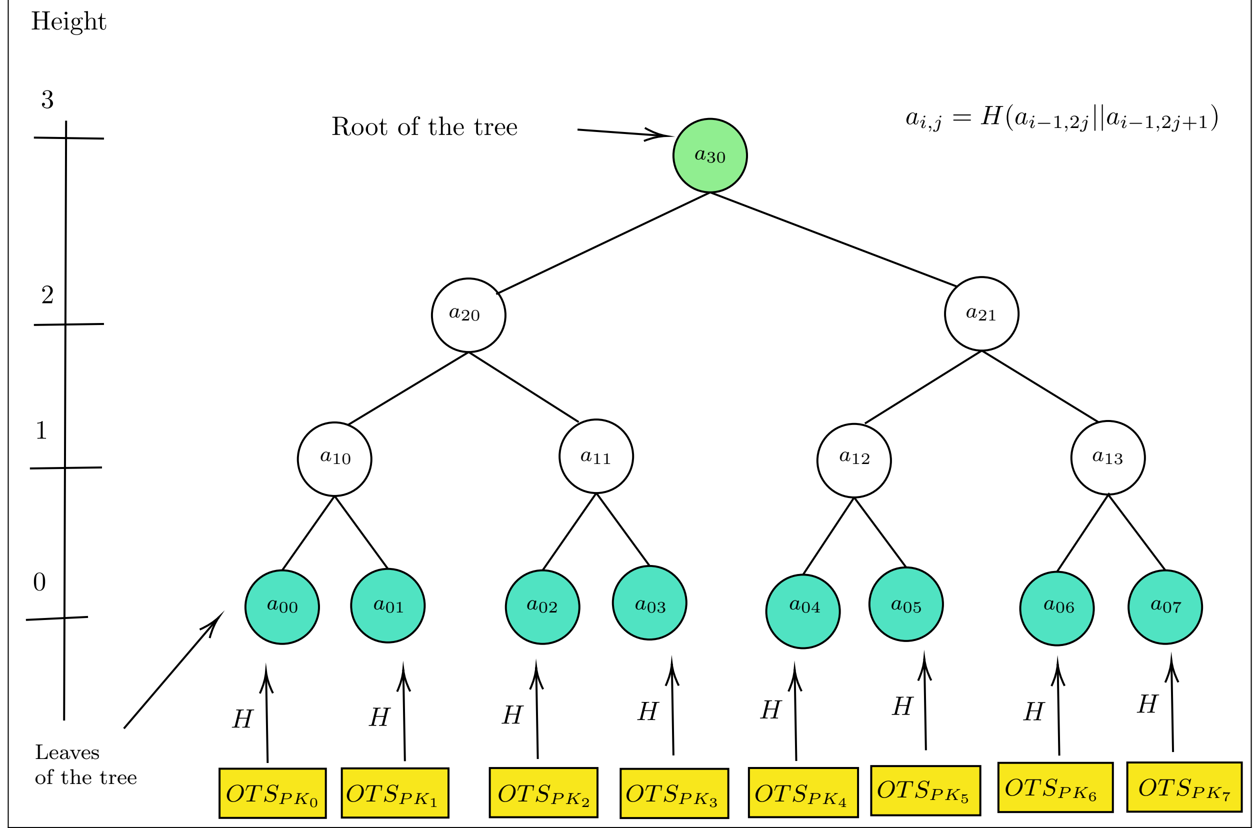

Fig. 8. Key generation of Merkle signature scheme with Merkle tree having height $h = 3$ and $2^h = 8$ leaves.

6 HORS

In this section, we will briefly have an overlook over HORS [20] and HORS (HORS with Trees) [2]. Hash to Obtain Random Subset (HORS) is an FTS (few-times signature) signature algorithm. Unlike OTS, an FTS algorithm can be used to sign messages for a few times, each time it is used, some information is exposed, reducing the key's security.

6.1 Protocol Description of HORS

$(SK, PK) \leftarrow \mathsf{KeyGen}(1^n)$: Given the security parameter 1^n, the algorithm generate t random n-bit strings to produce the secret key: $SK = (s_1, \ldots, s_t)$. In the following, public key is computed as $PK = (k, v_1, \ldots, v_t)$ where $v_i = F(sk_i)$.

$\sigma \leftarrow \mathsf{Sign}(m, SK)$: Given the message m and the secret key SK, the signature σ over m is computed as:

1. Computes $a = Hash(m)$
2. Split a into k substrings $a_1, \ldots a_k$, of length $\log_2 t$ bits each
3. Interpret each a_j as an integer i_j for $1 \leq j \leq k$
4. Outputs the signature $\sigma = (sk_{i_1}, \ldots, sk_{i_k})$

$1/0 \leftarrow \mathsf{Ver}(m, \sigma, PK)$: Given a message - signature pair $(m, \sigma = (s'_1, \ldots, s'_k)$, a verifier asserts the validity of the signature by going through the following steps:

1. Compute $a = Hash(m)$
2. Split a into k substrings $a_1, \ldots a_k$, of length $\log_2 t$ bits each
3. Interpret each a_j as an integer i_j for $1 \leq j \leq k$

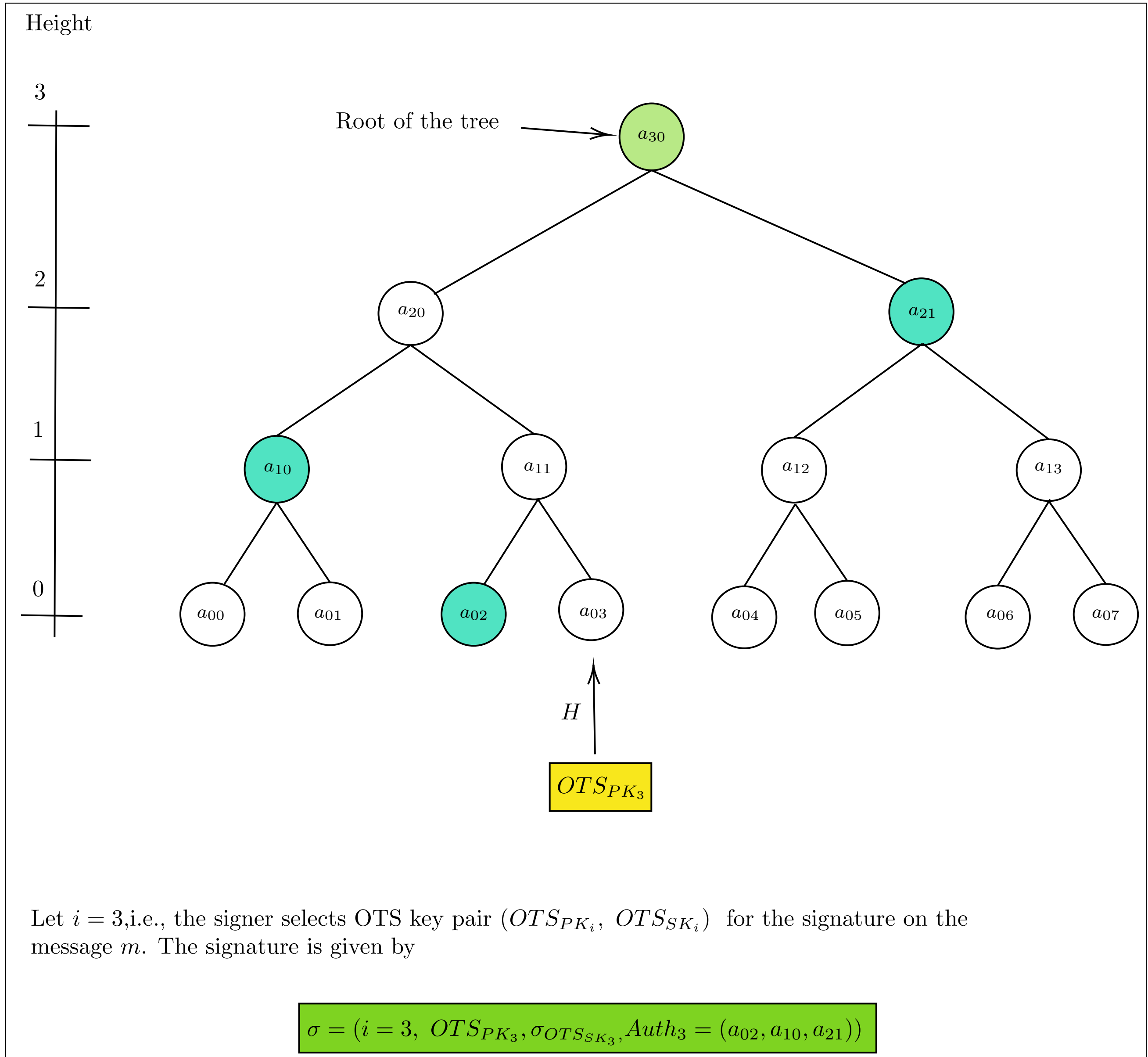

Fig. 9. A toy example illustrating signature generation of MSS

4. If for each j (where $1 \leq j \leq k$), $v_{i_j} = F(sk_j)$, then accepts the signature and outputs 1. Otherwise rejects and outputs 0.

Remark: F is a one way function and $Hash$ denotes a hash function. Both F and $Hash$ can be realized using a standard hash function like SHA3-256. A toy example of HORS is given in Figure 10.

7 HORST

HORST is an improvement over HORS because it uses a binary hash-tree structure to reduce the size of both the public key and signature. In SPHINCS, HORS with trees (HORST) replaces the t-value public key with a single value that represents the root of the Merkle tree. The leaves of this tree are the pk_i's. A HORST signature includes k sk_i's and their respective authentication paths, which are lists of sibling nodes required to connect each pk_i to the root. To optimize the process, SPHINCS includes all nodes at a particular level in the signature because the k authentication paths will often share high-level authentication nodes.

7.1 Protocol Description

Parameter Description: HORST signs uses parameters k and $t = 2^\tau$ with $k\tau = m$. The value x is determined based on t and k such that $k(\tau - x + 1) + 2^x$ is minimal.

$(sk, pk) \leftarrow \mathsf{HORST.kg}(S, Q)$: On input of seed $S \in \{0,1\}^n$ and bitmasks $Q \in \{0,1\}^{2n \times \log t}$, it generates the secret key-public key pair sk and pk:

1. The internal secret key is computed as $sk = (sk_1, \ldots, sk_t) \leftarrow G_t(S)$.
2. Compute the leaves of the tree by using F: $L_i = F(sk_i)$ for $i = 1, \ldots, t$. A tree is constructed by utilizing the bitmask Q.
3. The public key pk is computed as the root node of a binary tree of height $\log t$.
4. Outputs $sk = sk = (sk_1, \ldots, sk_t)$ and pk

The HORST key generation using binary tree is summarized in Figure 11.

$(\sigma, pk) \leftarrow \mathsf{HORST.sign}(M, S, Q)$: The signature on the message M is generated in the following manner:

1. Given S and Q, internal secret key sk is computed by employing $\mathsf{HORST.Kg}$.
2. M is divided into k chunks each of length $\log t$, i.e., $M = (M_1, \ldots, M_k)$. In the following, each M_i is interpreted as an integer.
3. Each M_i is now utilized as an index value to address a piece of the sk, sk_{M_i}.
4. The signature $\sigma = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k)$ consists of k blocks $\sigma_i = (sk_{M_i}, Auth_{M_i})$ for $i = 1, \ldots, k$ containing the M_ith secret key element and the authentication path connecting pk_i to the root.

The signature generation is explined through an illustrated toy example in Figure 12.

$0/1 \leftarrow \mathsf{HORST.vf}(M, \sigma, Q)$: The verification algorithm proceeds as:

1. M is divided into k chunks each of length $\log t$, i.e., $M = (M_1, \ldots, M_k)$. In the following, each M_i is interpreted as an integer.
2. Initially, the disclosed segments of sk are hashed and the message is divided into k sections. These sections are then considered as integers and used to locate the appropriate segments of sk on the corresponding leaves. With the nodes provided in the signature, the path to the root node can be determined. This process is repeated for all nodes and authentication paths, ensuring that they all converge on the same root. If they don't converge on the same root, the verification fails.

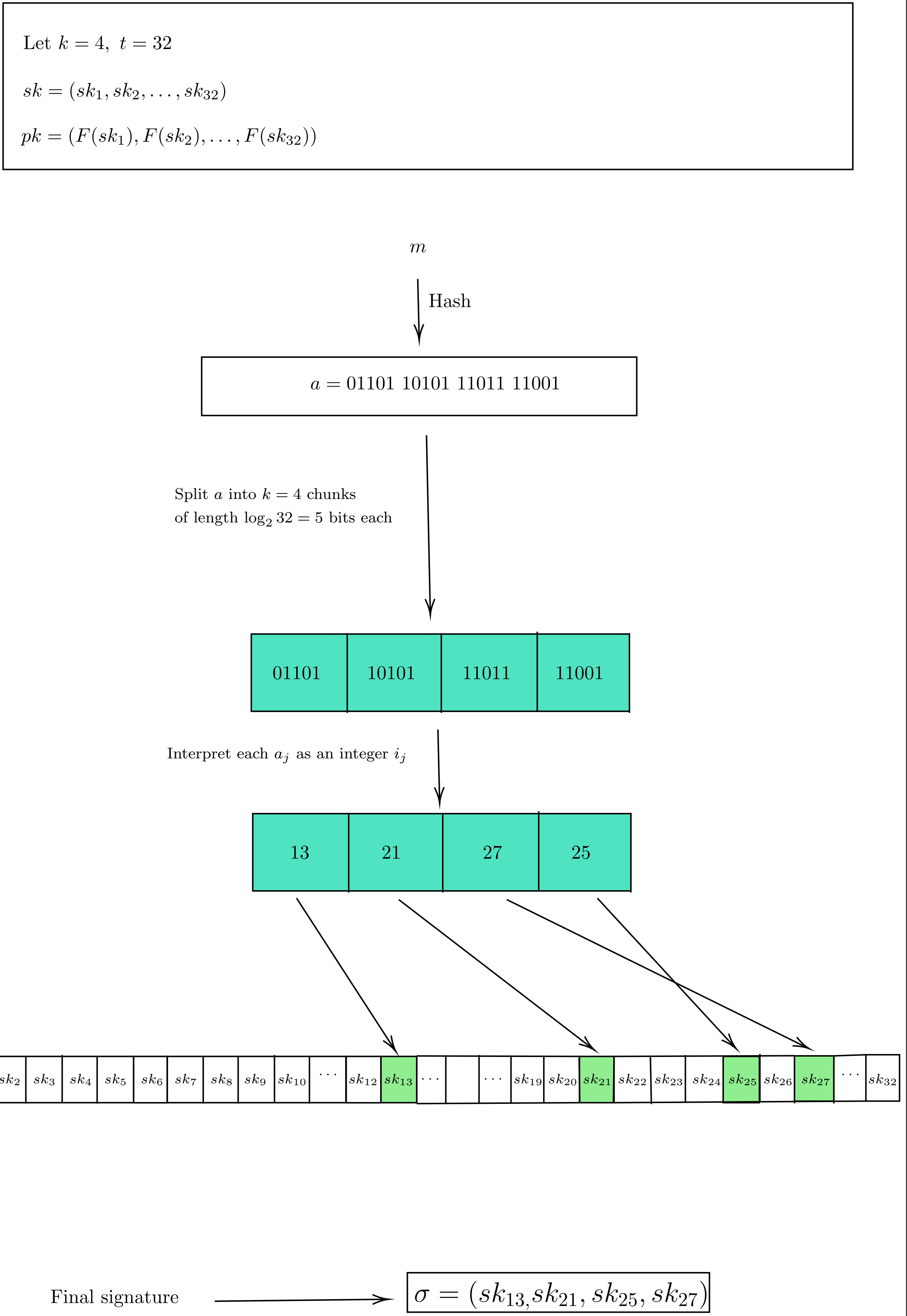

Fig. 10. HORS signature toy example

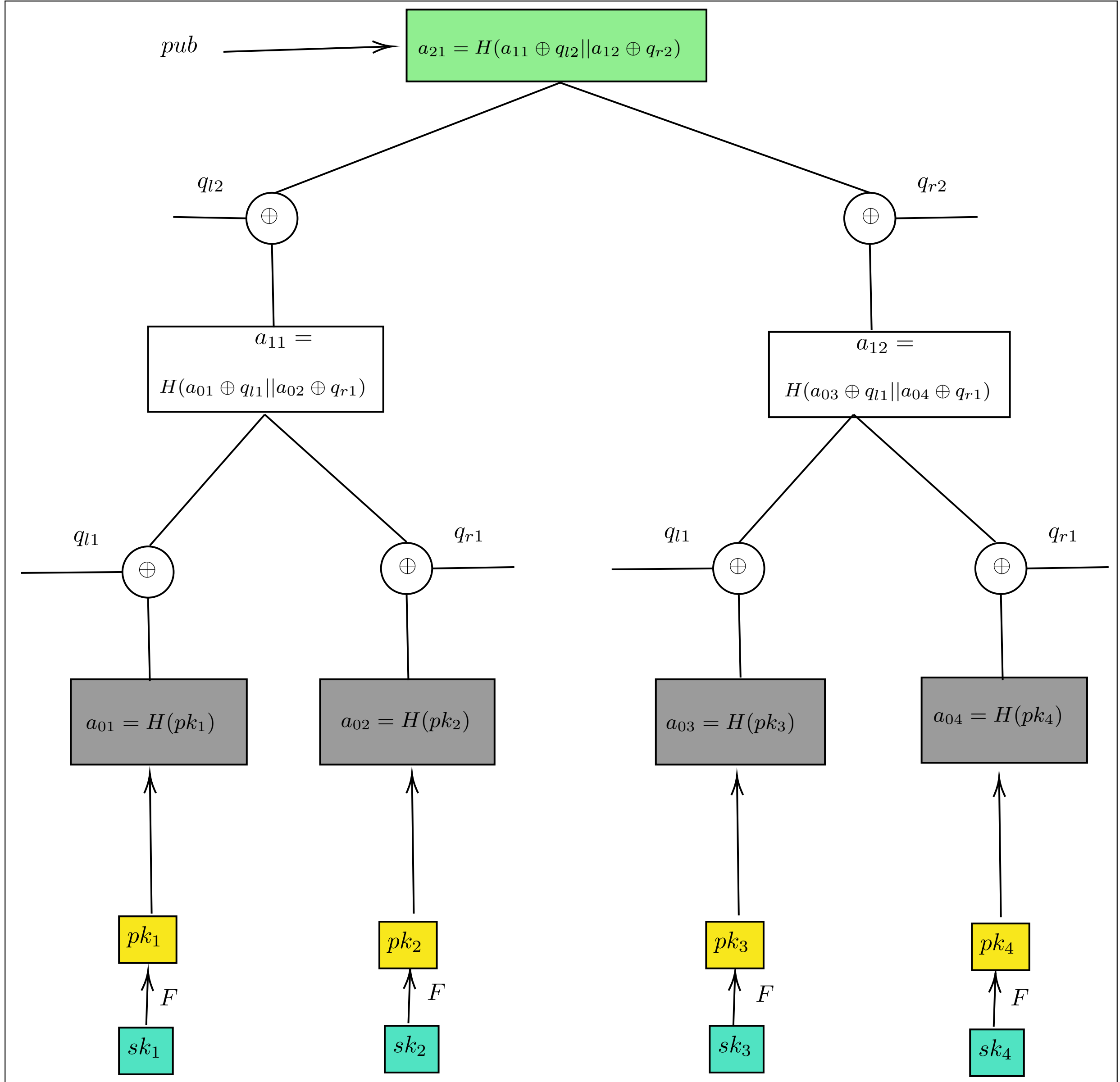

Fig. 11. HORST key generation

Algorithm 4 Root computation

Input: Leaf index i, leaf L_i, authentication path $Auth_i = (A_0, \ldots, A_{h-1})$ for L_i.
Output: Root node Root of the tree that contains L_i.

1: Set $P_0 \leftarrow L_i$;
2: **for** $j \leftarrow 1$ up to h **do**
3: $$P_j = \begin{cases} H((P_{j-1}||A_{j-1}) \oplus Q_j), & \text{if } \lfloor i/2^{j-1} \rfloor \equiv 0 \mod 2 \\ H((A_{j-1}||P_{j-1}) \oplus Q_j), & \text{if } \lfloor i/2^{j-1} \rfloor \equiv 1 \mod 2 \end{cases}$$
4: **end**
5: **return** P_h

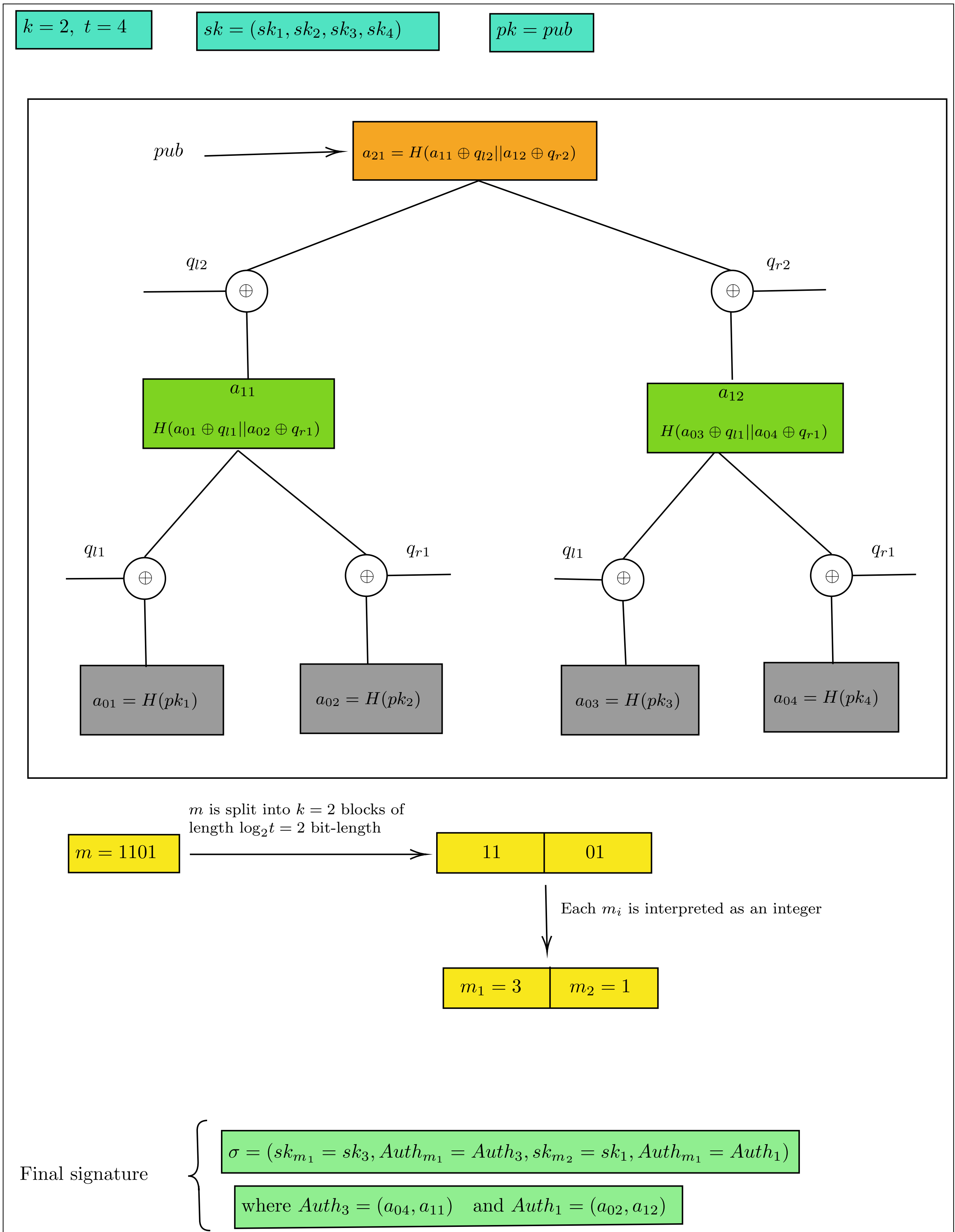

Fig. 12. A toy example illustrating HORST signature generation

8 FORS

In this section, we will described FORS used in Sphincs+. FORS is an improvement on HORS, designed by the Sphincs+ team. Both the HORS and FORS schemes are defined in terms of integers k and $t = 2^a$ and can be used to sign a string of ka bits. In both schemes we split the message digest into k bitstrings of length a each and interpret each a-bit substring as an integer written in binary. We use this integer as an index into the key structure.

FORS is similar to HORS except we have kt sets of private keys and k binary hash trees. Like HORS, FORS is defined in terms of integers k and $t = 2^a$ and can be used to sign a string of ka bits. A FORS private key consists of kt random n-bit values grouped together into k sets of t values each. In Sphincs+ these are deterministically derived from $SK.seed$ and the address $ADRS$ of the key in the hypertree.

Key Generation: To derive a FORS public key, construct k binary hash trees on top of the k sets of t private key elements. Each of the $t = 2^a$ values is used as a leaf node, resulting in k trees of height a. In Sphincs+, compress the root nodes using a call to a tweakable hash function. The resulting n-bit value is the FORS public key.

Signature Generation: Given a message of ka bits, extract k strings of a bits. Each of these bit strings is interpreted as the index of a single leaf node $(0, \ldots, t-1)$ in each of the k FORS trees. Use the 1st key set for the first signature value, use the 2nd key set for the second, and so forth. The signature consists of these k nodes and their respective authentication paths.

Signature Verification: The verifier recomputes the authentication path for each σ_i and checks the computed root equals the published public key. Unlike HORS, which selects all its signature values from the same set of t keys, FORS generates kt secret keys and dedicates t secret keys for each index out of the k indices. FORS mitigates against weak message attacks because, even if two bitstring segments produce an integer of the same value, their index values form different secret paths.

Refer to Figure 13 for an illustrated example of FORS.

9 Tree Based Authentication: Two Different Approaches

One of the major drawbacks of both OTS and FTS is that a given key pair can only be employed for signing one or a few number of messages. The technique of *tree based public key authentication* is to solve the problem of authenticating public key on a large scale in the context of hash based cryptography. To effectively authenticate public keys, a tree-based authentication system that uses the hash tree is widely used to address the key management issue.

9.1 Down-Top Approach

While using the down-top authentication approach, the root node is constructed form the leaf nodes (bottom layer). Note that root node is the master public key and it can be used to verify all of the individual OTS public keys (which appear as a leaf node on the hash tree). We now explain this approach in more details. Merkle hash based signature scheme described in Section 5 is based on Down-Top tree based authentication.

9.2 Top-Down Approach

Similar to down-top approach, firstly, one generates OTS key pairs. We then, construct the authentication path in the reverse direction compared with approach discussed above. In other words, we go from the direction of root node to the leaves. Private key associated to the parent node is employed to authenticate the child node in top to down order. Refer to Section 11.

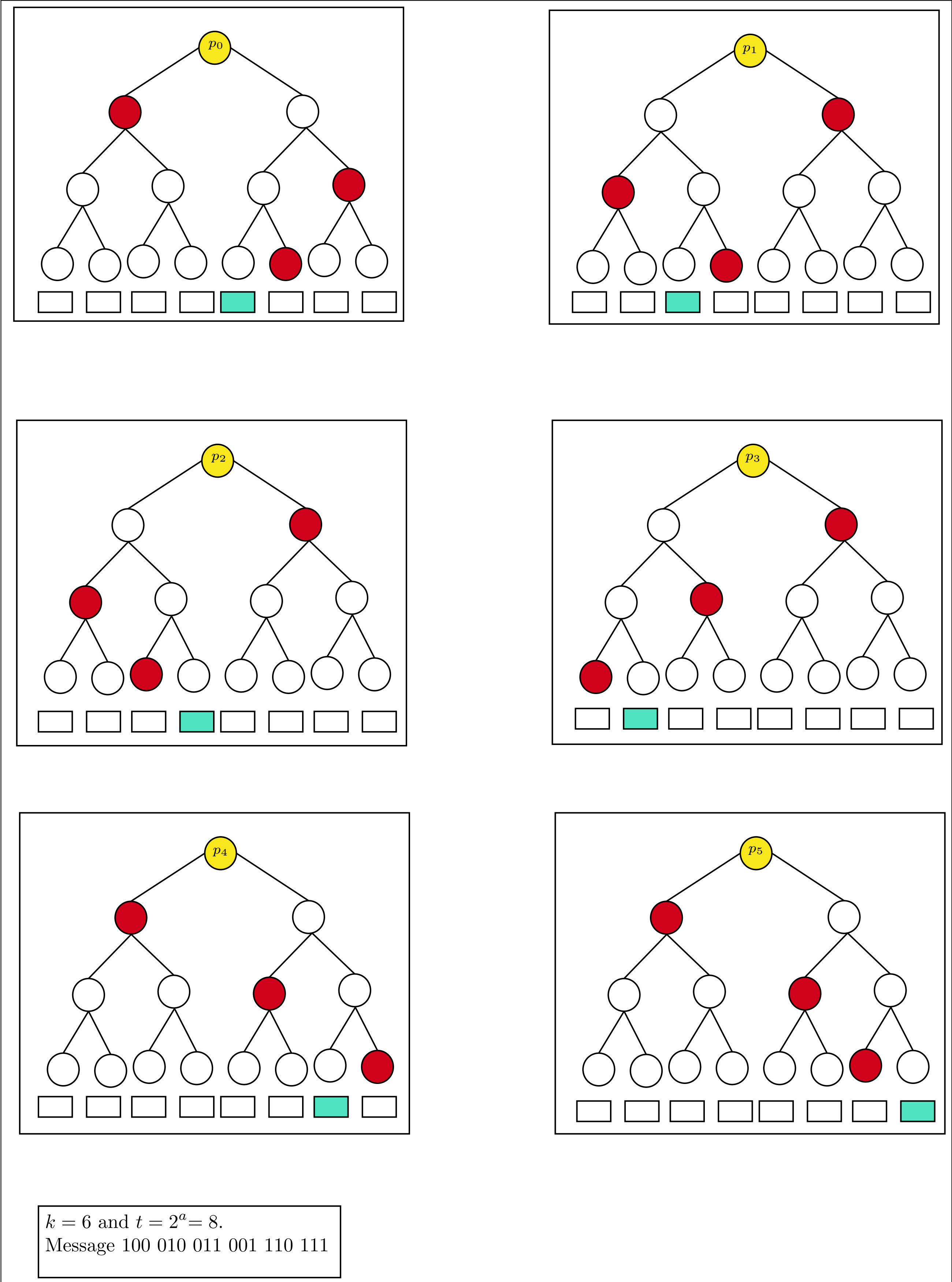

Fig. 13. FORS signature (schematic)

9.3 Hypertree

Hyper-tree is a form of a generalized hash tree which combines both of the two approaches given in Section 9.1 and Section 9.2. SPHINCS and almost all of its variants such as SPHINCS+, Gravity SPHINCS make use of hypertree structure.

10 XMSS

The Extended Merkle Tree Signature Scheme (commonly called XMSS) [5] uses Merkle tree to manage WOTS+ keys in a similar way that MSS uses a Merkle tree to manage Lamport keys. XMSS is a stateful signature based on MSS and WOTS+.

10.1 XMSS-XOR Tree

The XMSS-XOR tree [2] utilized in XMSS is an improved variant of the Merkle tree (refer to Figure 5 for detailed description of hash tree used in MSS). The nodes on level j, $0 \leq j \leq h$, are denoted by $a_{i,j}, 0 \leq i < 2^{h-j}$ (where h is the height of the tree). Level j, $0 < j \leq h$, is constructed using a bit-mask $(q_{l,j}||q_{r,j}) \in_R \{0,1\}^{2n}$. The nodes are computed as

$$a_{i,j} = Hash((a_{2i,j-1} \oplus q_{l,j})||(a_{2i+1,j-1} \oplus q_{r,j}))$$

A pictorial representation of XMSS tree (using a toy example) is given in Figure 14.

10.2 L Tree

The XMSS tree maybe employed to authenticate 2^h number of W-OTS (W-OTS+) public keys. Each individual WOTS+ public keys is utilized for generating one leaf of the XMSS tree. For the construction of leaves, we make use of a slight variant of the XMSS-XOR tree described above. The tree which is used to compress the public keys of each WOTS+ is also known as L-tree. The first l leaves of an L-tree are the l bit strings $(pk_1, ..., pk_l)$ from the corresponding public key of WOTS+. Since, it might be possible that l is not a power of 2, hence, "*a node with no right sibling is pushed to a higher level of the L-tree until it becomes the right sibling of another node*". Refer to Figure 15 for an illustrated toy example of L-tree.

To summarize, unlike MSS, leaves of XMSS-tree is not simply a hash of OTS public key. Root of another tree (also known as L-tree) is used as the leaves of the XMSS tree. A detailed pictorial representation of XMSS is given in Figure 16.

10.3 Protocol Description

XMSS.Kg: The key generation algorithm generates 2^h WOTS+ public key - secret key pair by using the algorithms key generation algorithm described in Section 4.

XMSS.Sig : To sign the ith message, the ith W-OTS key pair is used. The signature $SIG = (i, \sigma, Auth)$ contains the index i, the W-OTS signature σ, and the authentication path for the leaf $Node_{0,i}$. It is the sequence $Auth = (Auth_0, \ldots, Auth_{H-1})$ of the siblings of all nodes on the path from $Node_{0,i}$ to the root.

XMSS.Ver : To verify the signature $SIG = (i, \sigma, Auth)$, the string $(b_1, \ldots, b_l)$ is computed as described in the WOTS+ signature generation (refer to Section 4). The ith verification key $(pk_1, \ldots, pk_l)$ is computed similar to verification algorithm of WOTS+. The corresponding leaf of the XMSS tree is constructed using the L-tree. This leaf and the authentication path are used to compute the root. If it matches with the root of the XMSS tree in the public key, the signature is accepted. Otherwise, it is rejected.

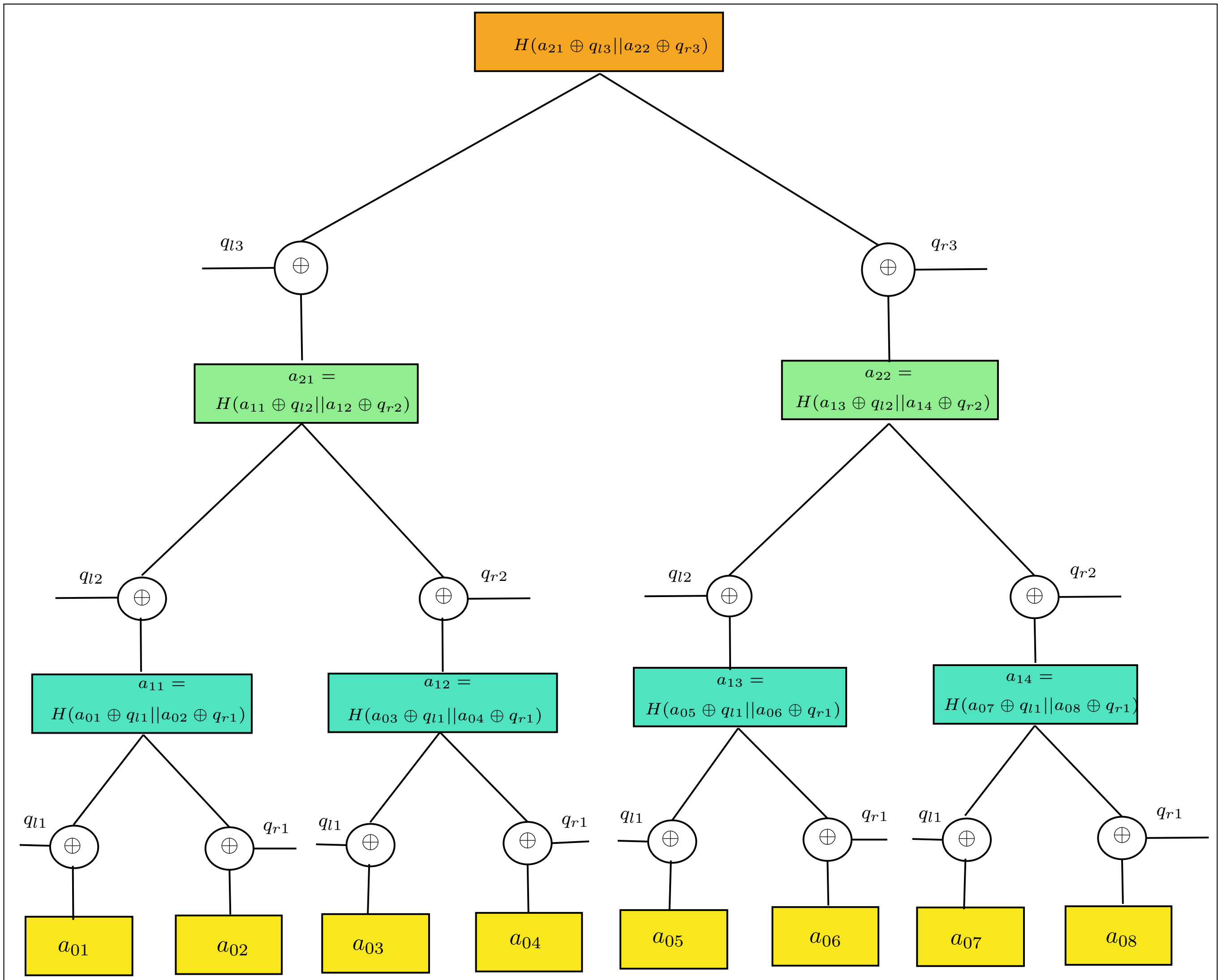

Fig. 14. Tree structure of XMSS; $H = 3$ is height of the tree (a tree of height h requires $2h$ bitmasks).

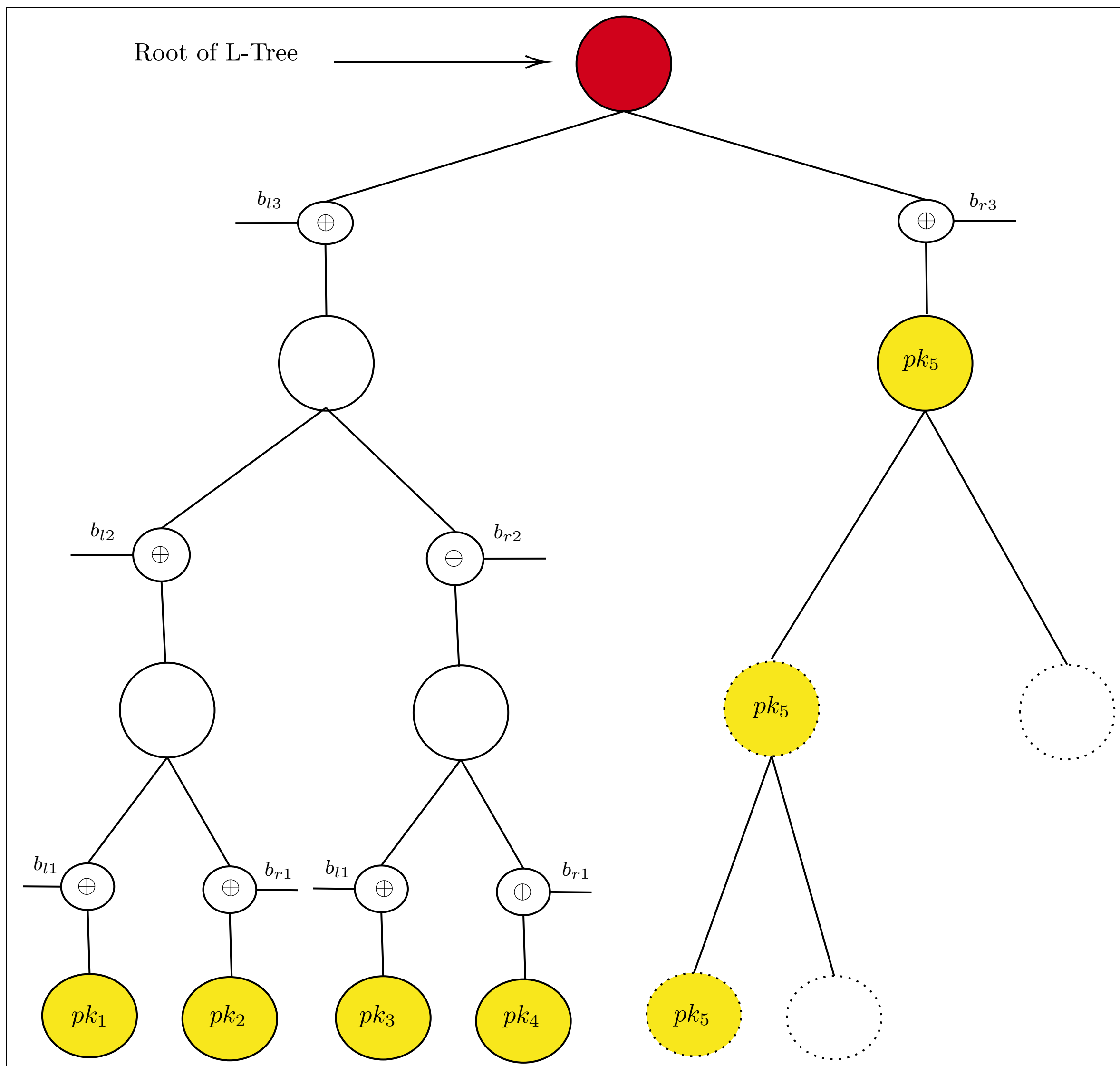

Fig. 15. Toy example illustrating the L-tree construction of a WOTS+ public key $pk = (pk_1, \ldots, pk_5)$.

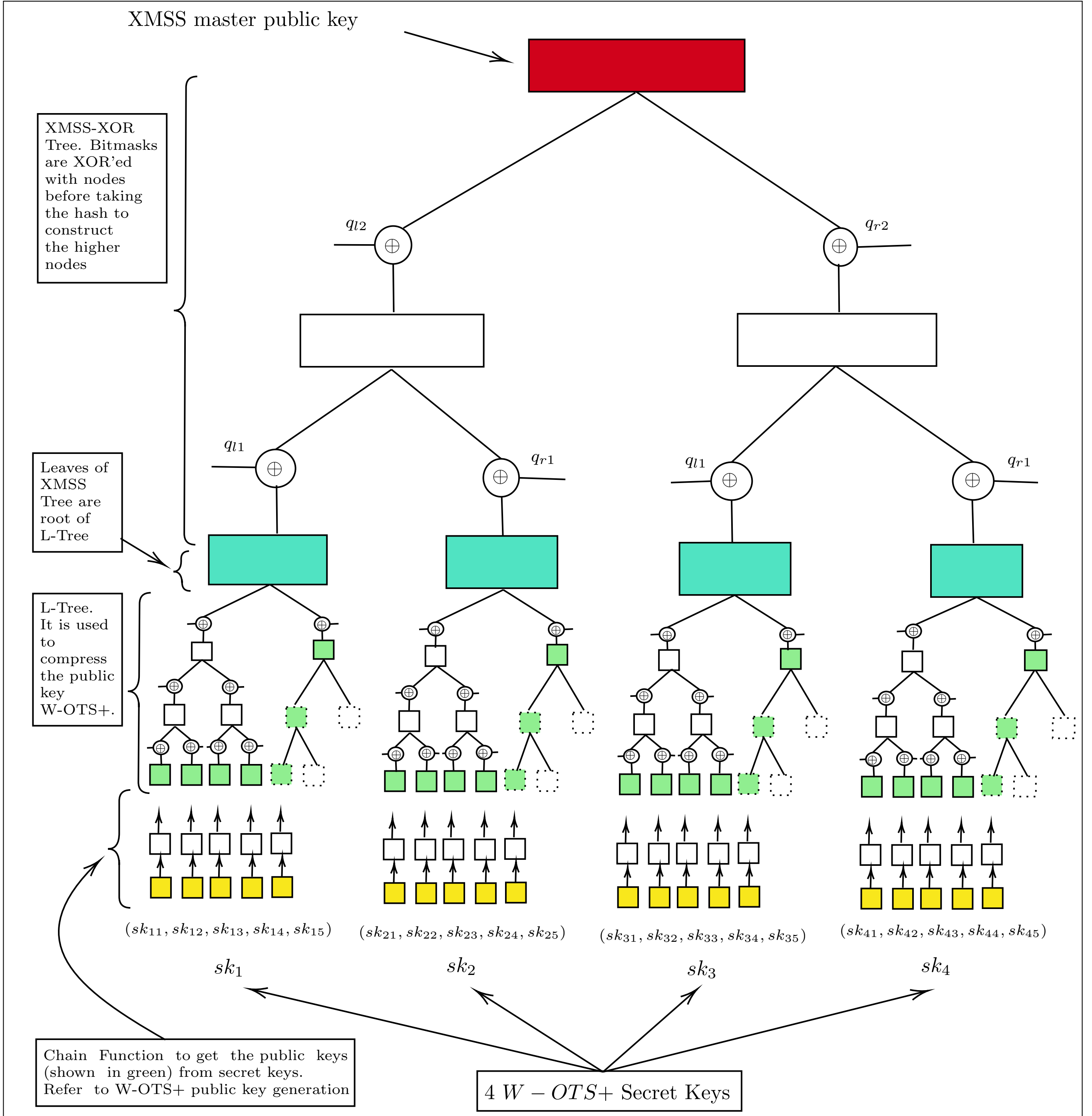

Fig. 16. Detailed pictorial representation of XMSS with L-Tree

11 XMSS^{MT}: A Hypertree variant of XMSS

XMSS^{MT} [12] is the hypertree variant of XMSS which enables an unlimited number of messages to be signed cryptographically. It uses XMSS to build the interior authentication path in a subtree (using down-top approach, refer to Section 9.1). It further utilizes WOTS+ to sign the root of the subtree by the signature key corresponding to the leaf node on the one layer higher (top-down authentication approach, refer to Section 9.2). Refer to Figure 17 for pictoirial summary of the construction of XMSS^{MT}. Each inner subtree in XMSS^{MT} is an XMSS tree. When signing a message using XMSS^{MT}, the authentication path within a sub-tree follows a bottom-up approach, and the root node of the sub-tree is computed from the bottom node to the top. On the other hand, authentication between sub-trees follows a top-down approach, where the leaf nodes of the higher sub-tree are utilized to authenticate the root of the sub-tree that is one layer lower.

The trees at the lowest level are utilized for message signing, while the trees at higher levels are utilized for signing the roots of the trees located on the layer below. To create a signature, all these WOTS+ signatures along the way to the highest tree are combined. An XMSS^{MT} signature, denoted as

$$\sigma = (i, \sigma_0, Auth_0, \sigma_1, Auth_1, \ldots, \sigma_d, Auth_d),$$

comprises the following components: the index i, the W-OTS signature σ_0 on the message M, and the corresponding authentication path. Additionally, the signature includes the W-OTS signatures on the roots of the trees that are currently in use, along with their corresponding authentication paths.

12 Sphincs

Sphincs [2] is composed of WOTS+ [11], XMSS [5], and HORST[20] building blocks. Refer to Figure 19. Based on HORST and WOTS+, a new stateless hash based signature algorithm called Sphincs can be constructed. It is important to observe that while past key management systems such as WOTS+ and HORST can handle a substantial quantity of keys, it is necessary to generate the keys beforehand to calculate the Merkle root. This acts as a constraint on the number of keys that can be managed. In contrast, Sphincs can manage a much larger quantity of keys without the need to pre-compute all the leaves by utilizing two methods:

- Hyper-tree
- Random key path addressing scheme

Table 1. Functions used in Sphincs

$F : \{0,1\}^n \rightarrow \{0,1\}^n$	Cryptographic hash function
$H : \{0,1\}^{2n} \rightarrow \{0,1\}^n$	Cryptographic hash function
$\mathcal{H} : \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^m$	Arbitrary-input randomized hash function
$G_\lambda : \{0,1\}^n \rightarrow \{0,1\}^{\lambda n}$	Family of pseudo-random generator
$\mathcal{F}_\lambda : \{0,1\}^\lambda \{0,1\}^n \rightarrow \{0,1\}^n$	Ensemble of pseudo-random function families
$F : \{0,1\}^* \{0,1\}^n \rightarrow \{0,1\}^{2n}$	Pseudo-random function family

The hyper-tree structure in Sphincs is a tree of trees, where the height of the hyper-tree is denoted by h. This hyper-tree is composed of trees with a height of h/d. It is important to note that the hyper-tree is a virtual structure, meaning that all sub-trees do not need to be constructed at once. Instead, only trees along a specific path on the hyper-tree need to be generated when signing a message. At the bottom level of the Sphincs hyper-tree, there is a level of HORS trees that contain

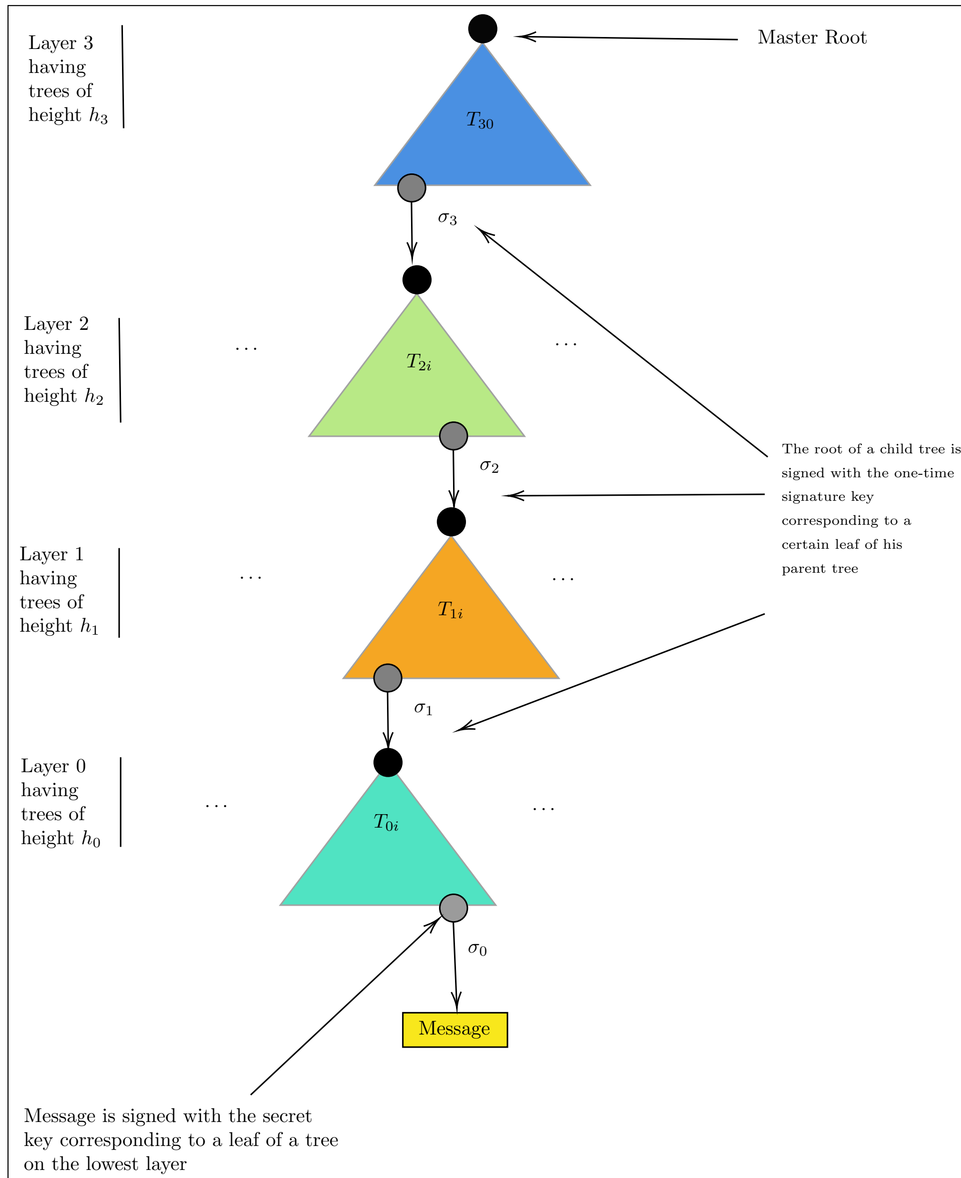

Fig. 17. Pictorial description of XMSS^{MT} with 4 layers.

private keys used for signing messages. When a message needs to be signed, Sphincs selects a HORS tree to sign the message and generates a signature σ_H. Above the HORS level, which is level 0, there are L-trees consisting of WOTS+ key pairs. Each leaf of these trees contains the public key strings of WOTS+, and their corresponding private keys are used for signing the root of the trees on the level below.

- There is only one tree on level $d-1$ which is the top tree.
- There are $2^{(d-i-1)*(h/d)}$ trees on level $i, i \in [0, d-2]$, and the root of the tree in level i will be signed by the WOTS+ private key of the tree on level $i+1$.

This implies that the WOTS+ trees and HORS trees are not dependent on each other, despite being chosen on the same path for signature authentication. This feature is useful in avoiding the need to pre-compute all the trees during key generation, thereby enabling the management of a vast number of keys under a single Sphincs public key. This large key space also makes Sphincs a practically stateless signature scheme. As previously mentioned, Sphincs only identifies specific paths

in the hyper-tree when signing a message. Sphincs generates this path by employing an addressing scheme to locate the WOTS+ public keys in the hyper-tree. The addressing scheme consists of the level of the hyper-tree, the tree on that level, and the leaf within that tree. Using this format, we can uniquely identify the location of each WOTS+ public key at every level of the hyper-tree.

The private key SK of Sphincs consists of:

- An n-bit key SK_1 generated using a PRG. It is used to generate random seeds for HORST and WOTS+ private key generation.
- An n-bit key SK_2 is also generated using a PRG. This key is used for generating an unpredictable index and message hash.
- Bitmasks $Q = (Q_0, Q_1, \ldots, Q_{p-1})$: Bitmasks are used in HORST, WOTS+, L-tree, and hyper-tree. WOTS+ needs $w-1$ bitmasks, HORST needs $2\log(t)$ bitmasks, and L-tree needs $2\lceil\log(l)\rceil$ bitmasks. In total, the complete Sphincs structure needs p bitmasks where $p = \max(w-1, 2(h + \lceil\log(l)\rceil), 2\log(t))$.

The address of the leaves of the tress at the highest layer, i.e., layer $d-1$ is given by

$$A = (d-1||0||i)(i \in [2^{\frac{h}{d}} - 1)]$$

We generate the seed $S_A \longleftarrow F(A, SK_1)$ using the n-bit secret key SK_1. In the following, we use S_A as the seed for the generation of the private keys of WOTS+. Later, we compute the root of this top level tree (let's denote the root by PK_{root}). The final private key and public key of Sphincs is given by

$$SK = (SK_1, SK_2, Q)$$
$$PK = (PK_{\text{root}}, Q)$$

We use F to denote random seed generation functions across this article. We first give a bird eye overview of signature generation in Sphincs. To create a signature for a given message, first a HORST is generated. In the second step, a specific hypertree path of Sphincs is produced. To produce the HORST key pair, first, the address of the HORST key pair is determined. Once we have the address, the corresponding random seed can be generated, and in the following, by utilizing the random seed, HORST key pair is generated. These keys are used to sign the message. All other WOTS+ signatures is generated in the similar fashion:

determination of address $\rightarrow$ generation of random seed $\rightarrow$ key pair generation $\rightarrow$ signature generation

The details are presented below.

- Generate two random n-bits R_1 and R_2 by $F(M, SK_2)$.
- Compute the message digest $D \longleftarrow H(R_1, M)$.
- Compute HORST address $i \longleftarrow Chop(R_2, h)$. and $\text{Address}_H = (d||i(0,(d-1)h/d)||i((d-1)h/d, h/d))$.
- Generate HORST key pair and HOTST signature
  - Generate HORST key pair seed by $\text{Seed}_H \longleftarrow F(\text{Address}_H, SK_1)$
  - Generate HORST signature and public key by (σ_H, pk_H) by executing the signature generation algorithm of HORST with (D, Seed_H, Q_H) as inputs.
- Generate all WOTS+ signatures along the Sphincs path
  - Compute all addresses of WOTS+ in the path $\text{Address}_{w,j} = (j||i(0,(d-1-j)h/d||i((d-1-j)h/d), h/d))$ where j is the level and $j \in [0, d-1]$.
  - Compute all the seeds $\text{Seed}_{w,j} = F(\text{Address}_{w,j}, SK_1)$

- Generate WOTS+ signature $\sigma_{w,j}$ by running the signature generation algorithm of WOTS+ with $(pk_{w,j-1}, \text{Seed}_{w,j}, Q_{WOTS+})$ as inputs. Here, $pk_{w,j-1}$ is the root of the tree of $j-1$ level. Also we need to generate the authentication path auth_{A_j} of corresponding WOTS+ public key.

The Sphincs signature is

$$\sigma_{\text{Sphincs}} = (i, R_1, \sigma_H, \sigma_{w,0}, \text{auth}_{A_0}, \sigma_{w,1}, \text{auth}_{A_1}, \ldots, \sigma_{w,d-1}, \text{auth}_{A_{d-1}})$$

The verification algorithm in Sphincs involves the following steps:

- The first step involves checking the HORST signature. The verification algorithm computes the digest D by computing $H(R_1, M)$. In the following, the verifier runs verification algorithm of HORST with (D, Q_{HORST}, σ_H) as inputs to check the validity of the HORST signature σ_H.
- The second step involves checking all WOTS+ signatures. The verifier first verifies $\sigma_{w,0}$ by executing verification algorithm of WOTS+ with $(pk_H, \sigma_{w,0}, Q_{HORST})$ as inputs. In the following, the verifier verifies $\sigma_{w,i}$ by running the verification algorithm of WOTS+ with $(pk_{w,i}, \sigma_{w,i}, Q_{HORST+})$ as inputs. Here $i \in [1, d-1]$
- Reject if any one of the WOTS+ signatures cannot be validated.
- On hyper-tree level $d-1$, the verifier gets the root of the hyper-tree. If the $root == PK_{\text{root}}$, the $\sigma_{Sphincs}$ is validated, otherwise reject.

Although the whole signature scheme looks complicated, the idea is simple: hyper-tree allows a much bigger key space without the need to pre-computing all keys and intermediary nodes.

How Sphincs Achieves Stateless Property

It's worth noting that both XMSS^{MT} and Sphincs use a hyper tree structure, which is illustrated in Figure 17 and Figure 19, respectively. These protocols have the ability to sign an unlimited number of messages cryptographically. The fundamental difference between these two protocols is that Sphincs is stateless, while XMSS^{MT} is stateful.

To sign a message using Sphincs, the initial step involves generating a hash of M that is randomized along with a random index computed deterministically using a PRF. This index is then used to determine which HORST to choose for signing the randomized hash of M. Therefore, by using a deterministic index based on the message, the design of Sphincs was able to eliminate the need for maintaining state information.

13 Sphincs+

With an understanding of Sphincs, we now introduce the state-of-art hash based signature scheme — Sphincs+ [3]. Among the latest hash based signature schemes, Sphincs+ has great advantages in speed, security and signature size. In general, the ideas of Sphincs+ and Sphincs are quite similar, but there are some differences:

- FORS (a FTS) is used for signing message in Sphincs+ in place of HORS.
- A publicly verifiable method for selection of leaf index is used.
- The concept of tweakable hash functions is introduced in Sphincs+, which enables the unification of the security analysis of hash based signature schemes.

We now introduce the key generation of Sphincs+. The public key PK of Sphincs+ comprises the following:

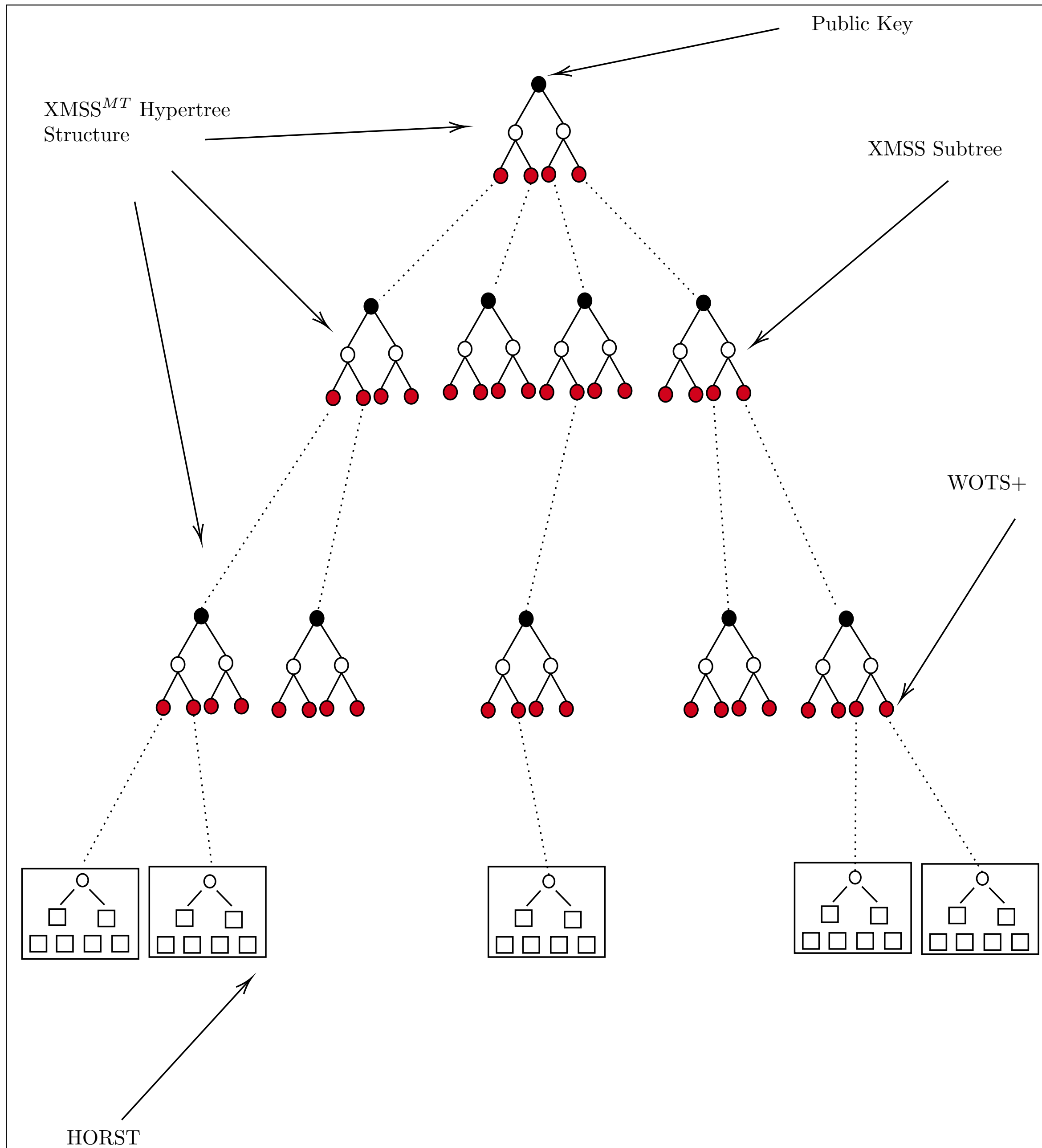

Fig. 18. General construction of Sphincs (schematic)

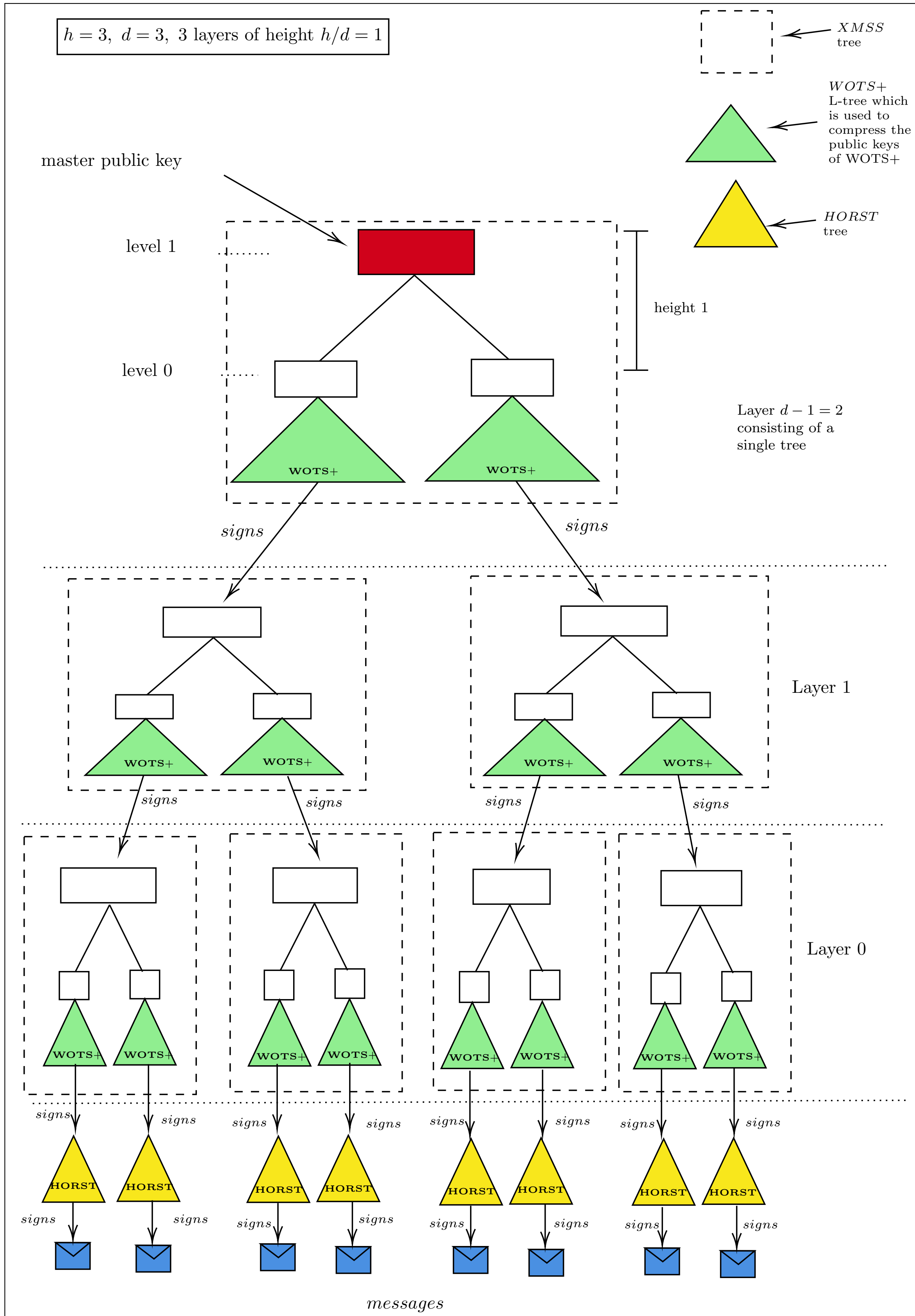

Fig. 19. Tree structure of Sphincs (schematic)

- root node of the subtree on the highest layer (i.e., root of the hyper tree structure) PK_{root},
- a random seed $PK.seed$

The private key SK comprises:

- $SK.seed$. It is an n-bit seed which is utilized for secret key generation of WOTS+ and FORS.
- $SK.PRF$. It is also an n-bit seed utilized for generation of random message digest.

Sphincs+ varies from the original Sphincs in the methods used to compute the message digest and leave selection.

- Sphincs+ pseudo-randomly generates a value R (which acts as randomizer). This value R depends on the message to be signed and $SK.PRF$. We can make the computation of R non deterministic by introducing another parameter of randomness *OptRand*. It might be helpful in circumventing the side-channel attacks. Here, $R = PRF(SK.prf, OptRand, M)$ which is part of the signature.
- Now by making use of R, we derive the index of the leaf node that is to be used, as well as the message digest $(MD||idx) = H_{msg}(R, PK.seed, PK.root, M)$ where MD means message digest and idx means leaf index. H_{msg} is an additional keyed hash function to compress the message.

Note that in Sphincs+ we use a publicly verifiable method for the selection of the index. It prevents a malicious party from choosing a random looking index, and combining it with a message of their choice. Since, the index is publicly verifiable and can be easily computed by the verifier, it is not a part of the signature in Sphincs+. The Sphincs+ signature is

$$\sigma_{Sphincs+} = (MD, R, \sigma_F, \sigma_{w,0}, \text{auth}_{A_0}, \sigma_{w,1}, \text{auth}_{A_1}, \ldots, \sigma_{w,d-1}, \text{auth}_{A_d-1})$$

where σ_F is the signature of FORS. To verify a signature $\sigma_{Sphincs+}$, a verifier proceeds in the following manner:

- Check the FORS signature by first generating MD and idx. Note that the process of generating MD and idx is same as described in signature generation of Sphincs+. In the following, use MD, idx and σ_F as input to the signatures verification algorithm of FORS to verify σ_F. To verify the signer's idx, verifier can also compute $H_{msg}(R, PK.seed, PK.root, M)$ and compare it.
- Rest of the process is same as in Sphincs. Basically the verifier checks the WOTS+ signatures of each level.

14 Improvements/Variants/Follow-ups of Sphincs and Sphincs+

14.1 Korean Sphincs+

The original SPHINCS+ algorithm utilized SHA2, SHAKE, and HARAKA hash functions to produce hash based signature. The authors of [21] suggested using Korean hash functions such as LSH, CHAM, and LEA to generate hash based signature. The resulting algorithm is called K-SPHINCS+.

14.2 Work by Kölbl

1. In [15], performance analysis of SPHINCS algorithm implementation using various cryptographic hash functions on modern Intel and AMD computer platforms as well as the ARMv8-A platform commonly found in mobile phones was done.
2. The efficiency of SPHINCS is highly dependent on the effectiveness of two functions, namely **F** and **H**, which must satisfy specific security criteria.

$$\mathbf{F} : \{0,1\}^{256} \to \{0,1\}^{256}$$

$$\mathbf{H} : \{0,1\}^{512} \to \{0,1\}^{256}$$

For **F** we require *preimage resistance, second-preimage resistance and undetectability*, while **H** has to be *second-preimage resistant.*

3. The analysis encompasses the implementation of several cryptographic hash functions, such as SHA256, Keccak, Simpira, Haraka, and ChaCha, optimized to hash small inputs concurrently by using vector instructions and cryptographic extensions present in these microprocessors.

14.3 Sphincs Simpira

1. The integration of AES instructions in modern processors has brought significant changes to the field of cryptography. Gueron et al.[9] introduced Simpira, a set of cryptographic permutations designed to achieve high throughput on processors equipped with AES instructions. The authors in [10] explored the integration of Simpira in SPHINCS-256. In particular, they used Simpira to instantiate the **F** and **H** functions.
2. The post-quantum security guarantees of SPHINCS-Simpira are equivalent to those of the original SPHINCS algorithm. However, the performance tests in [10] indicate that SPHINCS-Simpira provides faster key pair generation by 1.5 times, signing of 59-byte messages by 1.4 times, and signature verification by 2.0 times.

14.4 Sphincs Streebog

The authors of [14] discussed the use of Russian hash function Streebog in Sphincs and Sphincs+.

14.5 Sphincs α

The design presented in [23] has the following differences compared to Sphincs

1. Use of an improved Winternitz one-time signature.
2. Use of a variant of a few-time signature scheme FORS, called FORC.
3. By combining the above mentioned improvements with SPHINCS+ framework, the authors achieve a certain level of improvement in performance.

14.6 Sphincs Gravity

The following improvements are made by the authors in [1]:

1. Development of PORS, which offers more security when compared to HORS.
2. Secret key caching. The technique helps in speeding up the process of signature generation. In addition, it also helps in reducing the size of the signature.

15 Conclusion and Outlook

Hash based signatures have been proposed as an alternative to traditional digital signatures based on public-key cryptography. Hash based signatures are a promising candidate to provide post-quantum secure digital signatures. In this document, an effort has been put to gather various ideas and concepts related to hash based signature together. This document may serve as a starting point for readers who wants to start working in the field of hash based signatures. A basic classification of hash based signatures is given in Figure 20.

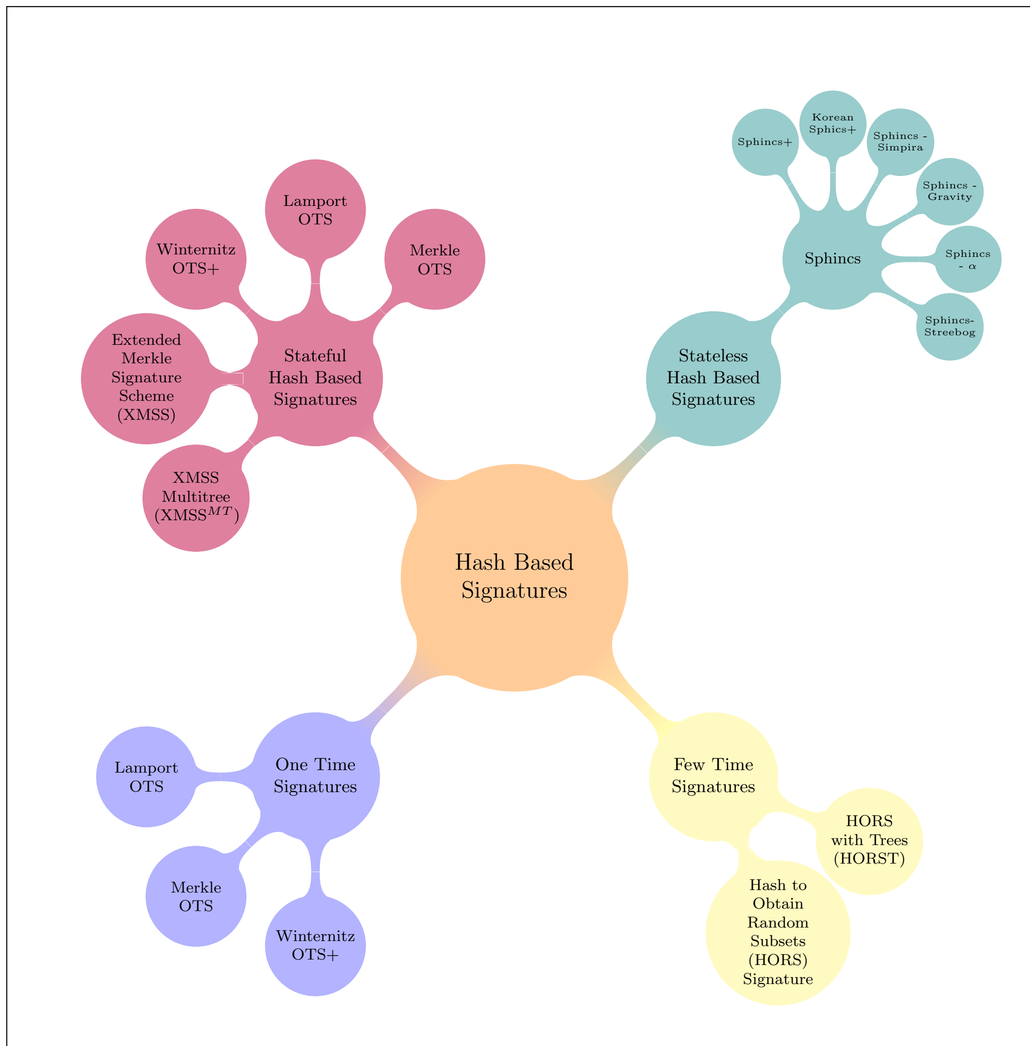

Fig. 20. Classification of hash based signatures

References

[1] Aumasson, J.P., Endignoux, G.: Improving stateless hash-based signatures. In: Topics in Cryptology–CT-RSA 2018: The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings. pp. 219–242. Springer (2018) 31

[2] Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: Sphincs: practical stateless hash-based signatures. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 368–397. Springer (2015) 1, 7, 12, 20, 24

[3] Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp. 2129–2146 (2019) 27

[4] Beullens, W., Kleinjung, T., Vercauteren, F.: Csi-fish: efficient isogeny based signatures through class group computations. In: Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part I. pp. 227–247. Springer (2019) 2

[5] Buchmann, J., Dahmen, E., Hülsing, A.: Xmss-a practical forward secure signature scheme based on minimal security assumptions. In: International Workshop on Post-Quantum Cryptography. pp. 117–129. Springer (2011) 20, 24

[6] Ding, J., Schmidt, D.: Rainbow, a new multivariable polynomial signature scheme. In: ACNS. vol. 5, pp. 164–175. Springer (2005) 2

[7] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 238–268 (2018) 1

[8] Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al.: Falcon: Fast-fourier lattice-based compact signatures over ntru. Submission to the NIST's post-quantum cryptography standardization process **36**(5) (2018) 1

[9] Gueron, S., Mouha, N.: Simpira v2: A family of efficient permutations using the aes round function. In: Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I. pp. 95–125. Springer (2016) 31

[10] Gueron, S., Mouha, N.: Sphincs-simpira: Fast stateless hash-based signatures with post-quantum security. Cryptology ePrint Archive (2017) 31

[11] Hülsing, A.: W-ots+–shorter signatures for hash-based signature schemes. In: International Conference on Cryptology in Africa. pp. 173–188. Springer (2013) 7, 24

[12] Hülsing, A., Rausch, L., Buchmann, J.: Optimal parameters for xmss mt. In: Security Engineering and Intelligence Informatics: CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings 8. pp. 194–208. Springer (2013) 1, 24

[13] Jang, K., Baksi, A., Song, G., Kim, H., Seo, H., Chattopadhyay, A.: Quantum analysis of AES. IACR Cryptol. ePrint Arch. p. 683 (2022), https://eprint.iacr.org/2022/683 2

[14] Kiktenko, E., Bulychev, A., Karagodin, P., Pozhar, N., Anufriev, M., Fedorov, A.: Sphincs+ post-quantum digital signature scheme with streebog hash function. In: AIP Conference Proceedings. vol. 2241, p. 020014. AIP Publishing LLC (2020) 31

[15] Kölbl, S.: Putting wings on sphincs. In: Post-Quantum Cryptography: 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings 9. pp. 205–226. Springer (2018) 30

[16] Lamport, L.: Constructing digital signatures from a one way function (1979) 5

[17] Merkle, R.C.: A certified digital signature. In: Advances in cryptology—CRYPTO'89 proceedings. pp. 218–238. Springer (2001) 1, 11

[18] Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Contr. Inform. Theory **15**(2), 157–166 (1986) 2

[19] Patarin, J.: Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In: Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15. pp. 33–48. Springer (1996) 2

[20] Reyzin, L., Reyzin, N.: Better than biba: Short one-time signatures with fast signing and verifying. In: Batten, L., Seberry, J. (eds.) Information Security and Privacy. pp. 144–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2002) 12, 24

[21] Sim, M., Eum, S., Song, G., Kwon, H., Jang, K., Kim, H., Kim, H., Yang, Y., Kim, W., Lee, W.K., et al.: K-xmss and k-sphincs+ : Hash based signatures with korean cryptography algorithms. Cryptology ePrint Archive (2022) 30

[22] Stinson, D.R., Paterson, M.: Cryptography: theory and practice. CRC press (2018) 3

[23] Zhang, K., Cui, H., Yu, Y.: Sphincs-α: A compact stateless hash-based signature scheme. Cryptology ePrint Archive (2022) 31