Hash-based Signature Schemes for Bitcoin

Mikhail Kudinov*, Jonas Nick†
Blockstream Research

Revision 2025-12-05

Abstract

Hash-based signature schemes offer a promising post-quantum alternative for Bitcoin, as their security relies solely on hash function assumptions similar to those already underpinning Bitcoin's design. We provide a comprehensive overview of these schemes, from basic primitives to SPHINCS$^+$ and its variants, and investigate parameter selection tailored to Bitcoin's specific requirements. By applying recent optimizations such as SPHINCS+C, TL-WOTS-TW, and PORS+FP, and by reducing the allowed number of signatures per public key, we achieve significant size improvements over the standardized SPHINCS$^+$ (SLH-DSA). We provide public scripts for reproducibility and discuss limitations regarding key derivation, multi-signatures, and threshold signatures.

Contents

*mishel.kudinov@gmail.com
†jonas@n-ck.net

1 Introduction

Public-key cryptography plays a critical role in Bitcoin: digital signatures allow users to prove ownership of funds and authorize transactions in a verifiable manner. Currently, Bitcoin employs ECDSA and Schnorr signatures over the secp256k1 curve, both of which are highly efficient in terms of signature size, verification time, and signing time.

However, the emergence of quantum computing poses a significant threat to Bitcoin's long-term security. While today's cryptographic schemes are considered secure against classical adversaries, Shor's algorithm [43] demonstrates that sufficiently powerful quantum computers would be able to efficiently solve the elliptic curve discrete logarithm problem, thereby breaking both ECDSA and Schnorr signatures. This would allow an adversary with quantum capabilities to forge signatures, seize control over existing funds, and compromise the integrity of the Bitcoin network. Although practical large-scale quantum computers remain speculative, the pace of quantum research and development underscores the urgency of preparing the Bitcoin ecosystem for a potential "post-quantum" era.

In this work, we investigate hash-based schemes as a post-quantum alternative to ECDSA and Schnorr signatures. These schemes offer a promising path for securing Bitcoin in a post-quantum world. Unlike number-theoretic systems, their security relies only on the hardness of inverting cryptographic hash functions, which are believed to resist both classical and quantum adversaries apart from Grover's quadratic speedup [16] (see Section 12 for further discussion). Notably, Bitcoin's security already relies on the collision resistance of SHA-256, as blocks and transactions are identified by their hash. Moreover, hash-based constructions are already standardized by NIST, with schemes such as LMS [33], XMSS [33], and SPHINCS$^+$ [34] offering varying tradeoffs in terms of efficiency, state management, and signature size. Adopting SPHINCS$^+$ as a post-quantum alternative for Bitcoin has already been discussed in the bitcoin-dev mailing list.[1]

Adopting hash-based signatures would require selecting appropriate parameters from the standardized schemes. A key focus of this work is investigating alternatives to the standardized schemes to better match Bitcoin's specific requirements, for example by using recent optimizations such as SPHINCS+C [22], TL-WOTS-TW [26, 14], and PORS+FP [1], as well as limiting the allowed number of signatures under a single public key [28].

Our presentation starts from basic primitives and builds up to SPHINCS$^+$ and its variants. We then discuss security requirements in the presence of a quantum adversary, followed by parameter selection. We present estimates of the required effort for signing and verifying, as well as signature sizes for different parameters and modifications; our analysis is reproducible via public scripts [6]. Finally, we discuss public key derivation, multi-signatures, and threshold signatures using hash-based schemes.

2 Hash Functions

Cryptographic hash functions are among the most fundamental primitives in modern cryptography. They provide a way to map arbitrary data to fixed-length outputs in a manner that appears random and unpredictable, yet is efficiently computable. In particular, hash functions are widely used for data integrity, authentication, and as building blocks in more complex protocols such as digital signatures, message authentication codes, and key derivation functions. In the context of post-quantum cryptog-

[1] See Proposing a P2QRH BIP towards a quantum resistant soft fork and Trivial QC signatures with clean upgrade path.

raphy, they play an even more crucial role, since hash-based schemes are built directly on the hardness of different hash functions properties, such as preimage resistance or target collision resistance.

We start by recalling the basic definition of an unkeyed hash function.

Definition 2.1 (Unkeyed hash function)**.** *Let $\mathcal{M}$ be the message space and $\mathcal{N}$ the output space. An unkeyed hash function is an efficient deterministic function*

$$F : \mathcal{M} \to \mathcal{N},\ y \leftarrow F(m)$$

generating a digest y out of a message m.

Typically, $\mathcal{M}$ consists of all bit strings of arbitrary length, while $\mathcal{N}$ is the set of bit strings of some fixed length n. It is common for n to be a power of two (for example, $n = 256$ in SHA-256), but this is not strictly necessary. The fixed-length digest serves as a compact representation of the message, suitable for efficient verification and use in higher-level protocols.

In many applications, it is desirable to incorporate a secret key into the hashing process. This leads to the notion of keyed hash functions, which generalize the unkeyed case.

Definition 2.2 (Keyed hash functions)**.** *Let $\mathcal{M}$ be the message space, $\mathcal{K}$ be the key space, and $\mathcal{N}$ the output space. A keyed hash function is an efficient function*

$$F : \mathcal{K} \times \mathcal{M} \to \mathcal{N},\ y \leftarrow F(k, m)$$

generating a digest y out of a key k and a message m.

We often write $F_k(m)$ to denote the keyed function $F(k, m)$. Conceptually, keyed hash functions can be viewed as families of unkeyed hash functions, each indexed by a particular key. When the key is secret, such functions can provide authentication or message integrity guarantees, as in the case of HMAC. When the key is public, keyed hash functions serve as versatile building blocks that allow protocols to introduce controlled variability into hashing operations.

A further refinement, motivated by the design of advanced hash-based signature schemes, is the notion of a *tweakable* hash function [5]. The idea of tweakability is to provide an additional input, called a *tweak*, that modifies the behavior of the hash function without requiring a new key.[2] This mechanism enables domain separation, i.e., the ability to ensure that the same message hashed in different contexts produces unrelated outputs. Tweakable hash functions were introduced in the context of the SPHINCS$^+$ signature scheme, where domain separation is critical for structuring the large number of hashes required for key and signature generation.

Definition 2.3 (Tweakable hash function [5])**.** *Let $\mathcal{P}$ be the public parameters space, $\mathcal{T}$ the tweak space, $\mathcal{M}$ the message space, and $\mathcal{N}$ the output space. A tweakable hash function is an efficient function*

$$\mathsf{Th} : \mathcal{P} \times \mathcal{T} \times \mathcal{M} \to \mathcal{N},\ y \leftarrow \mathsf{Th}(P, T, m)$$

[2] A similar approach is usually referred to as a tagged hash function in the Bitcoin space.

mapping a message m to a hash value y using a function key called public parameter $P \in \mathcal{P}$ and a tweak $T \in \mathcal{T}$.

Here, the public parameter P may be viewed as fixing a particular instantiation of the hash function, while the tweak T provides lightweight variability akin to a nonce or a label. By leveraging tweaks, protocol designers can enforce domain separation without having to maintain multiple independent hash functions or keys, which simplifies both analysis and implementation.

3 Lamport OTS

Lamport's one-time signature scheme [30] is the prototypical example of a hash-based signature scheme. It illustrates how the one-wayness of a cryptographic hash function alone can be leveraged to provide digital authenticity. Although Lamport signatures are rarely used in practice in their raw form due to their large key and signature sizes, they form the conceptual foundation for more advanced hash-based constructions.

Lamport OTS with an unkeyed hash function. Let $F : \mathcal{N} \to \mathcal{N}$ be an unkeyed cryptographic hash function with output and input length $2^n = N = |\mathcal{N}|$. To sign n-bit messages, the Lamport scheme proceeds as follows.

- **Key generation:** The signer samples $2n$ random secret values

$$sk_{i,0}, sk_{i,1} \xleftarrow{\$} \mathcal{N} \quad \text{for } i = 1, \dots, n.$$

  The corresponding public key is the collection of their hashes:

$$pk_{i,0} = F(sk_{i,0}), \quad pk_{i,1} = F(sk_{i,1}).$$

- **Signing:** To sign a message $m \in \{0,1\}^n$, the signer outputs

$$\sigma = (sk_{1,m_1}, sk_{2,m_2}, \dots, sk_{n,m_n}),$$

  i.e., for each bit m_i of the message, reveal the corresponding secret key element.

- **Verification:** Given (m, σ) and the public key, the verifier checks for each i that

$$F(\sigma_i) = pk_{i,m_i}.$$

  The signature is valid if and only if all checks pass.

The scheme is secure against forgery under the assumption that F is one-way: to forge a signature on a new message, an adversary would need to invert the hash on at least one unrevealed preimage. An example of a Lamport signature instance is given in Fig. 1.

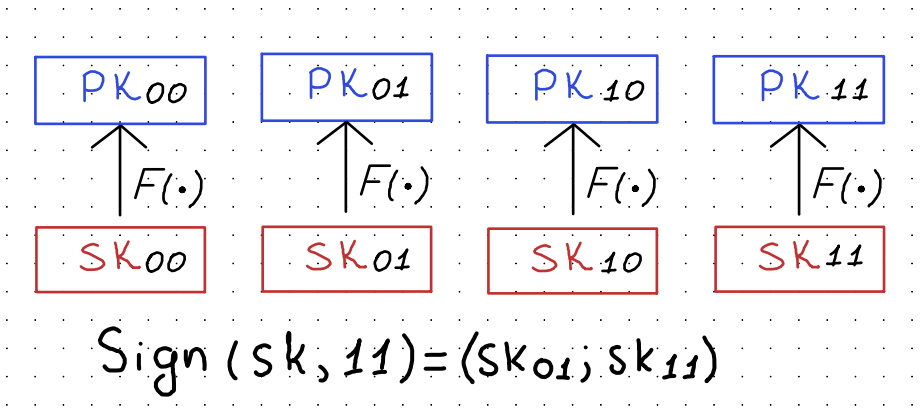


Figure 1: An example of a Lamport signature scheme, used to sign $m = 11$.

Lamport OTS with a tweakable hash function. One drawback of the classical Lamport scheme is that the same hash function is reused across all positions and contexts, which reduces the security of the scheme, as the adversary can search for any of the unrevealed secret key elements at the same time. This motivates the use of tweakable hash functions $\mathsf{Th} : \mathcal{P} \times \mathcal{T} \times \mathcal{N} \to \mathcal{N}$ (see Definition 2.3), where the tweak allows us to ensure that each hash invocation is uniquely bound to its position in the key pair, and the public parameter can help with multi-user security. This can be viewed as domain separation for the hash function calls.

- **Key generation:** As before, sample $sk_{i,0}, sk_{i,1} \xleftarrow{\$} \mathcal{N}$, but also sample a public parameter $P \leftarrow_{\$} \mathcal{P}$. The public key elements are now computed with explicit tweaks:
$$pk_{i,b} = \mathsf{Th}(P, (i, b), sk_{i,b}), \quad i = 1, \ldots, n, \ b \in \{0, 1\}.$$
Here $(i, b) \in \mathcal{T}$ denotes a tweak encoding of the index i and bit value b. The public parameter P becomes a part of both the public key and the secret key.
- **Signing:** Identical to the unkeyed version: for message $m \in \{0, 1\}^n$, output $\sigma = (sk_{1,m_1}, \ldots, sk_{n,m_n})$.
- **Verification:** For each i, the verifier computes
$$\mathsf{Th}(P, (i, m_i), \sigma_i) \stackrel{?}{=} pk_{i,m_i}.$$

This variant offers a cleaner abstraction: each element of the public key is bound to its position by the tweak. Previously the brute-force over the secret values could land in any of the public key elements and hence constitute a forgery. With each try there was an $n/|\mathcal{N}|$ success probability. With the introduction of tweaks, the guess must include the tweak for which the guess is performed. Hence, the success probability of a single try is now $1/|\mathcal{N}|$. In larger constructions, such as when Lamport OTS is used as a component inside a Merkle tree or hypertree, tweakable hashing simplifies the security analysis and ensures proper domain separation across all invocations.

Seed-based generation of Lamport secret keys. A practical drawback of the Lamport signature scheme in its basic form is the large size of the secret key: for n-bit messages, the signer must store $2n$ independent secret values. For typical parameters ($n = 256$), this already requires storing 512 random strings.

This problem can be solved by generating all secret values deterministically from a single short seed using a pseudorandom function (PRF). Concretely, let $\mathbf{PRF} : \mathcal{K} \times \mathcal{M} \to \mathcal{N}$ be a keyed hash function that behaves as a PRF. The signer samples a uniformly random seed $\mathsf{SK.seed} \leftarrow_{\$} \mathcal{K}$, which serves as the master secret key. The $2n$ secret values for the Lamport scheme are then derived as

$$sk_{i,b} = \mathbf{PRF}(\mathsf{SK.seed}, (i, b)) \quad \text{for } i = 1, \ldots, n, \ b \in \{0, 1\},$$

where (i, b) is an encoding of the position i and the bit value b.

The corresponding public key elements are computed in the usual way:

$$pk_{i,b} = \mathsf{Th}(P, (i, b), sk_{i,b}),$$

During signing and verification, the scheme remains identical to the basic Lamport construction, with the exception that the secret key elements are computed from the seed. Now the signer only needs to store the short seed s rather than the full set of $2n$ random values.

As long as **PRF** behaves as a secure PRF, the derived values $sk_{i,b}$ are computationally indistinguishable from uniformly random strings. Thus, replacing the explicit storage of $2n$ random values with PRF-derived ones does not affect the security of the Lamport scheme. This seed-based optimization is standard in practice and is crucial in more advanced schemes such as XMSS and SPHINCS$^+$, where the number of secret elements is much bigger and these values must be managed efficiently. However, this does not solve the problem of big public keys and big signatures.

Public key compression. We can do a further optimization concerning the size of the public key. In the standard Lamport scheme, the public key consists of $2n$ hash values, which can be substantial for large n. To reduce this overhead, one can apply a *public key compression* step. The idea is to compute a single digest that represents the entire set of public key elements. Using a tweakable hash function Th, the signer can define the compressed public key as

$$pk = \mathsf{Th}(P, T_{\text{pk}}, (pk_{1,0}, pk_{1,1}, \ldots, pk_{n,0}, pk_{n,1})),$$

where P is the public parameter and T_{pk} is a dedicated tweak for public key compression. This approach ensures that all individual public key components are bound together into one short value, while the tweak guarantees proper domain separation from other uses of Th within the scheme. The compressed public key significantly reduces storage and transmission costs, and it allows the verifier to recompute the same digest from the revealed secret values and the message during signature verification, without requiring access to the full uncompressed public key. When we are using tweakable hash functions, we also need to include the public parameter into the public key.

The seed-based generation and public key compression techniques will be useful for any one-time signature scheme that will be presented in this work.

4 Winternitz OTS With Tweaks (WOTS-TW)

The Winternitz one-time signature (WOTS) [32, 12] scheme is a classic hash-based signature construction that improves upon Lamport signatures by reducing the signature size through the use of hash chains. SPHINCS$^+$ introduced its own variant of WOTS; we give the description of the scheme below. Given that the defining characteristic of this variant is its reliance on tweakable hash functions, it is referred to as WOTS-TW [5, 21].

Parameters. WOTS-TW uses several parameters. The main security parameter is $n \in \mathbb{N}$. The length of messages to be signed is denoted by $m \in \mathbb{N}$, which, in the case of SPHINCS$^+$, is set to n. The message is then viewed as an integer. The Winternitz parameter $w \in \mathbb{N}$ determines the base of the representation of this value. The Winternitz parameter also determines the parameter len:

$$\mathsf{len}_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \mathsf{len}_2 = \left\lfloor \frac{\log(\mathsf{len}_1(w-1))}{\log(w)} \right\rfloor, \quad \mathsf{len} = \mathsf{len}_1 + \mathsf{len}_2.$$

The tweak space $\mathcal{T}$ must have a size of at least $\mathsf{len} \cdot w$. However, in larger constructions such as SPHINCS$^+$, where multiple instances of WOTS-TW are used, the tweak space should be even larger to ensure that each hash function call in the global structure uses a distinct tweak.

For WOTS-TW we use the following functions:

- A pseudorandom function $\mathbf{PRF} : \{0,1\}^n \times \mathcal{T} \to \{0,1\}^n$,
- A tweakable hash function $\mathsf{Th} : \{0,1\}^n \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$.

WOTS-TW scheme. In WOTS-like schemes, a sequence of hash chains is computed. The message is initially encoded and subsequently mapped to the corresponding intermediate values within these hash chains. The computation of the hash chains is performed using a chaining function.

Chaining function $c^{j,k}(x, i, P)$: The chaining function takes as inputs a bitstring $x \in \{0,1\}^n$, an iteration counter $k \in \mathbb{N}$, a start index $j \in \mathbb{N}$, a chain index i, and a public parameter P. The chaining function selects a unique tweak $T_{a,b} \in \mathcal{T}$ for each hash function call. The chaining function works as follows:

- If $k \leq 0$, return the input unchanged:

$$c^{j,0}(x, i, P) = x.$$

- For $k > 0$, define the function recursively as

$$c^{j,k}(x, i, P) = \mathsf{Th}(P, T_{i,j+k-1}, c^{j,k-1}(x, i, P)).$$

As we will see later, the signature will reveal some intermediate blocks in the chains. Hence, these blocks will be computed as $\sigma_i = c^{0,b_i}(\mathsf{sk}_i, i, P)$, for some index b_i. Here the

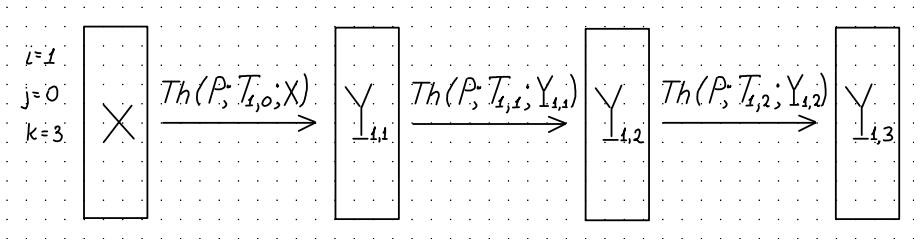


Figure 2: An example of chaining function computation: $c^{0,3}(X, 1, P) = Y_{1,3}$.

start index is 0 and the chaining function is called recursively b_i times. The verifier will need to finish the hash chain to validate the signature by computing $c^{b_i, w-1-b_i}(\sigma_i, i, P)$. Now the hash chain starts from index b_i and the chaining function is called recursively $w - 1 - b_i$ times.

When multiple instances of WOTS-TW are considered, we will add a subscript and use $c^{j,k}_{\mathbf{ADRS}}(x, i, P)$ to denote that the tweaks used to construct the chain have **ADRS** as a prefix. For example, if we have two instances of WOTS-TW, we have to distinguish between the tweaks used in the first one and in the second one. Then, we could define the **ADRS** of the first one to be 0 and the second one to be 1. Hence, the tweaks of the first instance would be $T_{\mathbf{ADRS},i,j} = 0 \| T_{i,j}$. And the hash invocation will look like $\mathsf{Th}(P, T_{\mathbf{ADRS},i,j}, x)$. An example of chaining function computation is shown in Fig. 2.

With this chaining function, we now describe the algorithms of WOTS-TW. An example of a WOTS-TW instance is shown in Fig. 3.

Key generation algorithm $(\mathsf{SK}, \mathsf{PK}) \leftarrow$ **WOTS-TW**.kg$(\mathcal{C}; \mathsf{SK.seed})$**:**
The key generation algorithm optionally takes context information $\mathcal{C} = (P, \mathbf{ADRS})$ as input, where $P \in \{0,1\}^n$ is a public parameter and **ADRS** is a tweak prefix that is used for this instance of WOTS-TW. We will omit the **ADRS** in the presentation below, as it only affect the tweak encodings. Additionally, it takes randomness $\mathsf{SK.seed} \in \{0,1\}^n$, which we call the secret seed. These inputs are intended for use in more complex protocols, such as SPHINCS$^+$. If not provided, the algorithm samples them at random and sets **ADRS** to 0.

The internal secret key $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{len}})$ is derived as

$$\mathsf{sk}_i \leftarrow \mathbf{PRF}(\mathsf{SK.seed}, T_{i,0}),$$

meaning the $\mathsf{len} \cdot n$ bit secret key elements are deterministically derived from the seed using addresses.

The public key elements are then computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}}) = \left(c^{0,w-1}(\mathsf{sk}_1, 1, P), \ldots, c^{0,w-1}(\mathsf{sk}_{\mathsf{len}}, \mathsf{len}, P)\right).$$

The key generation algorithm outputs $\mathsf{SK} = (\mathsf{SK.seed}, \mathcal{C})$ and $\mathsf{PK} = (\mathsf{pk}, \mathcal{C})$. Note that both sk and pk can be recomputed from SK.

Signature algorithm $\sigma \leftarrow$ **WOTS-TW**.sign(m, SK)**:** Given an m-bit message m, and the secret key $\mathsf{SK} = (\mathsf{SK.seed}, \mathcal{C})$, the signing algorithm proceeds as follows. First

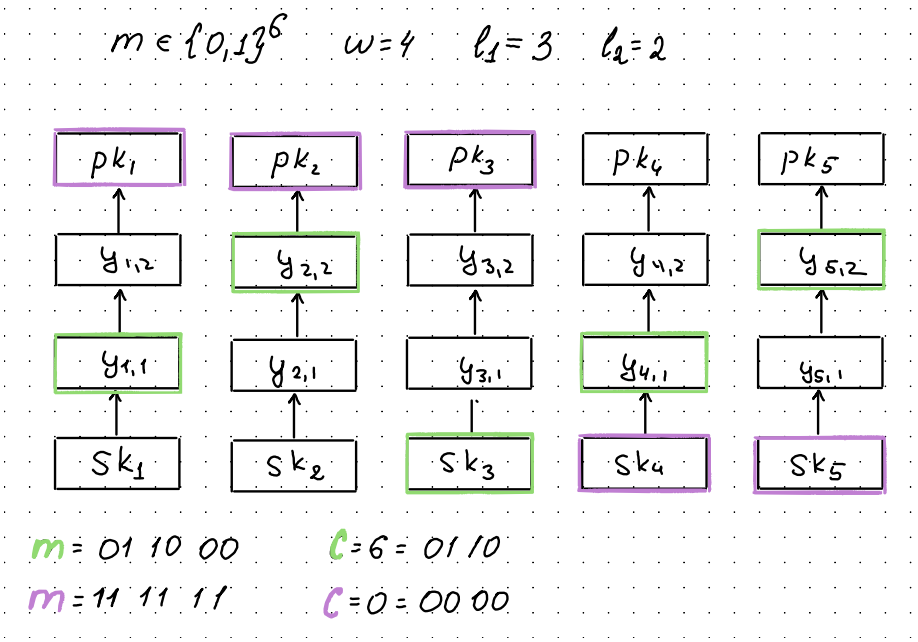


Figure 3: An example of a WOTS-TW instance. In this example $w = 4$, $\mathsf{len}_1 = 3$; $\mathsf{len}_2 = 2$.

compute a base-w representation of m:

$$m = (a_1, \dots, a_{\mathsf{len}_1}), \quad a_i \in \{0, \dots, w-1\}.$$

That is, treat m as the binary representation of a natural number x and compute its w-ary representation.

Next compute the checksum

$$C = \sum_{i=1}^{\mathsf{len}_1} (w - 1 - m_i),$$

and its base-w representation

$$C = (C_1, \dots, C_{\mathsf{len}_2}), \quad C_i \in \{0, \dots, w-1\}.$$

Define

$$B = (b_1, \dots, b_{\mathsf{len}}) = m \,\|\, C,$$

the concatenation of the message and checksum digits.

Then compute the internal secret key as in key generation, and output the signature

$$\sigma = (\sigma_1, \dots, \sigma_{\mathsf{len}}) = \left(c^{0,b_1}(\mathsf{sk}_1, 1, P), \dots, c^{0,b_{\mathsf{len}}}(\mathsf{sk}_{\mathsf{len}}, \mathsf{len}, P)\right).$$

Verification Algorithm ($\{0,1\} \leftarrow$ **WOTS-TW**.vf(m, σ, PK))**:** Given a message m, a signature σ, and public key $\mathsf{PK} = (\mathsf{pk}, C)$, the verifier recomputes the digits b_i as described above. It then checks whether

$$\begin{aligned}\mathsf{pk} \stackrel{?}{=} \mathsf{pk}' &= (\mathsf{pk}'_1, \dots, \mathsf{pk}'_{\mathsf{len}}) \\ &= \left(c^{b_1, w-1-b_1}(\sigma_1, 1, P), \dots, c^{b_l, w-1-b_{\mathsf{len}}}(\sigma_{\mathsf{len}}, \mathsf{len}, P)\right).\end{aligned}$$

If equality holds, the signature is accepted; otherwise, it is rejected.

Security. To give a high-level intuition on the security of WOTS-TW, let us assume an adversary observes a valid message–signature pair (m, σ). The adversary wants to forge a different pair (m', σ'), where $m' \neq m$. Let us denote the encoding of m as $(b_1, \dots, b_{\mathsf{len}})$ and the encoding of m' as $(b'_1, \dots, b'_{\mathsf{len}})$. If we compare the encodings of m and m', then there will be at least one position i where b'_i appears *earlier* in the hash chain than b_i (i.e., $b'_i < b_i$). This follows from the checksum computation. First note that since $m \neq m'$, the encodings must be different. Suppose $b'_i \geq b_i$ for all $i \in [1, \mathsf{len}_1]$ (the encoding without the checksum); then the derived checksum C' will necessarily be smaller than the original checksum C. Consequently, the base-w representation of C' must contain at least one digit that is strictly smaller than the corresponding digit in C. This ensures that any forgery attempt requires the adversary to produce at least one signature block $\sigma'_i = c^{0, b'_i}(\mathsf{sk}_i, i, P)$ that is closer to the secret key sk_i than the observed block $\sigma_i = c^{0, b_i}(\mathsf{sk}_i, i, P)$. This ensures that any forgery attempt requires finding at least one hash chain with an earlier block, which is infeasible if Th behaves as a secure tweakable hash function.

5 WOTS+C

In this section, we present a modification of the WOTS construction that reduces signature size *without* increasing chain lengths and while also decreasing the verification time, called WOTS+C [22]. This scheme introduces two additional parameters $S_{w,n}, z \in \mathbb{N}$. Instead of signing the message m directly, we sign the digest

$$d = \mathsf{Th}(P, T^*, \mathsf{m} \| \mathsf{count})$$

where count is a nonce. We iterate the counter count until the resulting d satisfies the following two properties when mapped to len_1 chain locations $a_1, \dots, a_{\mathsf{len}_1} \in [0, w-1]$:

1. **Fixed sum:** $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$.
2. **Additional zero-chains:** $\forall i \in [w-1-z+1,\ w-1]\ :\ a_i = 0$.

Rationale and benefits. The first condition fixes the sum of all base-w digits to $S_{w,n}$ (parameter of the scheme), which the verifier can check; hence, there is no need to authenticate this sum via extra checksum chains as in WOTS-TW, yielding smaller signatures and faster verification.

The second condition allows us to *omit* the last z chains from the signature entirely; again, the verifier can check this. Overall, this scheme can be viewed as a variant of WOTS/WOTS-TW (and applies to other variants as well) with fewer chains to sign and verify: instead of $\mathsf{len} = \mathsf{len}_1 + \mathsf{len}_2$ chains, we sign/verify only

$$l \;=\; \mathsf{len}_1 - z$$

chains (Fig. 4).

Instantiation interface. We can now describe the WOTS+C key generation, signing, and verification procedures.

$\mathsf{KeyGen}(1^n)$:

1. The secret key is random strings $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_l)$.
2. The public key is $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_l)$, where $\mathsf{pk}_i = c^{0,w-1}_{\mathrm{ADRS}}(\mathsf{sk}_i, i, P)$.

$\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$:

1. Increment a $\mathsf{count} \in \{0,1\}^r$, until $d = \mathsf{Th}(P, T^*_{\mathrm{ADRS}}, \mathsf{m}\|\mathsf{count})$ satisfies the two conditions[3].
2. Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [0, w-1]$.
3. For $i \in [l]$ compute $\sigma_i = c^{0,a_i}_{\mathrm{ADRS}}(\mathsf{sk}_i, i, P)$.
4. Output $\sigma = (\sigma_1, \ldots, \sigma_l, \mathsf{count})$.

$\mathsf{Verify}(\mathsf{m}, \sigma, \mathsf{pk})$:

1. Parse σ as $(\sigma_1, \ldots, \sigma_l, \mathsf{count})$.
2. Parse $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_l)$.
3. Compute $d = \mathsf{Th}(P, T^*_{\mathrm{ADRS}}, \mathsf{m}\|\mathsf{count})$.
4. Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [0, w-1]$.
5. Verify that $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$ and that $\forall i \in [w-1-z+1,\ w-1] : a_i = 0$.
6. Verify that for all $i \in [l]$, it holds that $\mathsf{pk}_i = c^{a_i, w-1-a_i}_{\mathrm{ADRS}}(\sigma_i, i, P)$.

The intuition behind the security of this scheme is similar to the security of the original construction. One needs to search either for a second preimage for a message hash, or one would have to invert one of the chains.

By eliminating the checksum chains, verification avoids hashing along those chains and thus becomes faster. Signing includes an additional step of searching for a nonce value that yields a digest meeting the constraints, but this cost is offset by hashing fewer chains overall, so the expected signing time remains comparable. Note that the $S_{w,n}$ value that requires the fewest counter iterations is the average value: $(w-1)\cdot \mathsf{len}_1/2$. On the other hand, we can choose a higher value of the $S_{w,n}$. This will result in more work for the signer, but the verifier will have to perform less hashing to complete the chains. Choosing $z > 0$ yields further signature compression at the cost of (probabilistically) increasing the expected number of trials for count.

[3] Recall the conditions: **Fixed sum:** $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$. **Additional zero-chains:** $\forall i \in [w-1-z+1,\ w-1] : a_i = 0$.

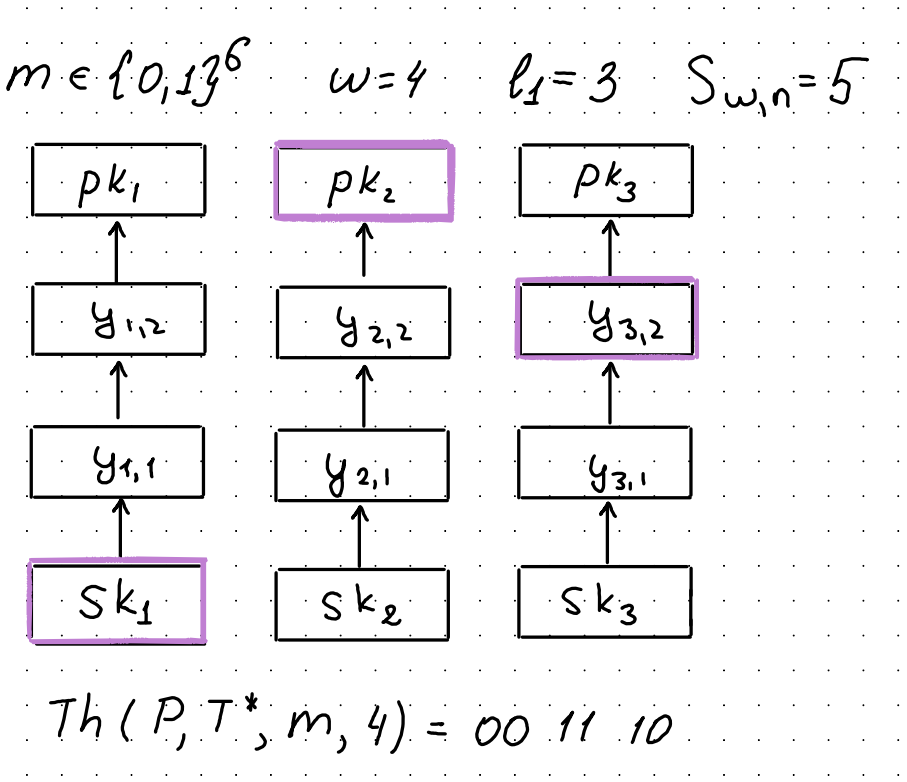


Figure 4: A WOTS+C example with $z = 0$.

Security overview. Intuitively, the security of WOTS+C relies on two hardness assumptions: (i) unforgeability of the underlying WOTS-like scheme (e.g., WOTS+ or WOTS-TW), and (ii) the difficulty of finding a colliding message that lands in the restricted subset of the hash image defined by the constraints above. A forgery on a new message either yields a standard WOTS-type forgery (if $\mathsf{Th}(P, T^*_{\mathrm{ADRS}}, \mathsf{m}\|\mathsf{count}) \neq \mathsf{Th}(P, T^*_{\mathrm{ADRS}}, \mathsf{m}'\|\mathsf{count}')$) or produces a second preimage for Th.

5.1 Complexity Analysis of WOTS+C

We count time in hash calls. As in WOTS(-TW), chain computation costs scale with (chain length) × (number of chains). The remaining cost is the search for a nonce count such that $d = \mathsf{Th}(P, T^*, \mathsf{m}\|\mathsf{count})$ meets the two constraints. Assuming H behaves as a random function, outputs are uniform, and each trial is an independent Bernoulli with success probability p_v.

We optionally replace the second condition by requiring that the last z_b bits of d are zero, which is convenient when $\log w \nmid n$. This scales the success probability by 2^{-z_b}.

For the *fixed-sum* constraint, the number of valid strings is the number of ordered

l-tuples from $[0, w-1]$ summing to $S_{w,n}$ can be counted as:

$$v = \sum_{j=0}^{l} (-1)^j \binom{l}{j} \binom{(S_{w,n}+l) - jw - 1}{l-1}, \tag{1}$$

One can also compute the value of v as a coefficient with the variable $x^{S_{w,n}}$ in the expanded form of $(1 + x + ... + x^{w-1})^l$. Thus,

$$p_v = \frac{v}{w^l \cdot 2^{z_b}},$$

the expected number of trials is $1/p_v$, and the tail probability of needing more than k trials is $(1 - p_v)^k$.

6 Top-Layer WOTS-TW (TL-WOTS-TW)

In the recent work *At the Top of the Hypercube – Better Size-Time Tradeoffs for Hash-Based Signatures* [26], a new observation was made regarding the design of hash-based signature schemes. Suppose we fix the signature size, i.e., the number of chains in the construction. Let $\mathcal{M}$ denote the message space[4]. Traditionally, the message space is mapped bijectively onto the chain indices, i.e., $\mathcal{M} = [0, w-1]^l$. The key idea is to relax this mapping. Instead of enforcing equality, we may choose w larger than required so that $\mathcal{M} \subset [0, w-1]^l$. This extra flexibility allows us to redesign the mapping between messages and chains. In particular, by selecting a larger Winternitz parameter w, we can map messages closer to the *top* of the chains. Such a design shift directly impacts performance. For the verifier, fewer hashing steps are needed since the signature reveals values already higher in the chains. This improves verification time. On the other hand, the signer must now perform additional work to compute those higher chain values when producing a signature. The resulting tradeoff, lighter verification at the cost of heavier signing, opens a new dimension for optimizing hash-based signature schemes. As we will see later, the verification complexity is not our priority, hence we don't think this modification is crucial for the Bitcoin case. However, an interested reader can find a more detailed presentation in Appendix A.

7 The XMSS Structure

XMSS (eXtended Merkle Signature Scheme) [8, 20] is a hash-based digital signature scheme that builds on the idea of combining one-time signatures (OTS) with a Merkle tree. The central goal is to achieve many-time signing capability, while keeping both the public key and signature sizes practical.

[4] If a preprocessing hash function is applied to messages, then its output space serves as the message space for the underlying WOTS-TW construction.

Basic structure. At its core, XMSS relies on a one-time signature scheme, such as WOTS-TW or WOTS+C or TL-WOTS-TW, to sign individual messages. Each OTS key pair is used at most once. To authenticate the many OTS public keys that are needed, XMSS organizes them as the leaves of a binary Merkle tree of height h. Each internal node of the tree is a hash of its two children, and the root of the tree serves as the global public key. An example of a XMSS structure is given in Fig. 5.

To sign a message, it is first hashed with a salt. The salt is used to avoid relying on collision resistance. Avoiding the need for collision resistance allows us to use hash functions with smaller output sizes, which in turn reduces the size of the OTS signature. In addition, collision resistance has historically been one of the first properties to fail in practical hash functions. Designing the scheme so that it does not rely on this property therefore provides a security advantage.

Signing. To sign a message:

1. The signer chooses the next unused OTS key pair (at leaf index i).
2. The randomized hash of the message is signed using the OTS private key, producing an OTS signature σ_{OTS}.
3. To allow the verifier to authenticate the corresponding OTS public key, the signer also provides the authentication path of the leaf in the Merkle tree (i.e., the sequence of sibling nodes from the leaf up to the root).
4. The final signature is the tuple $(i, R, \sigma_{\text{OTS}}, \text{AuthPath}_i)$, where R is the randomness used to compute the hash of the message.

Verification. The verifier proceeds as follows:

1. Compute the message digest.
2. Reconstruct the OTS public key from σ_{OTS} and the digest.
3. Using the OTS public key as the leaf value at position i, recursively hash it together with the provided authentication path up to the root.
4. Compare the result with the global public key root; accept if they match.

Statefulness. XMSS is a *stateful* scheme: each OTS key pair can only be used once, and the signer must ensure that indices are never reused. This requires the signer to maintain persistent state between signatures, keeping track of the next available index. If a signer accidentally reuses an OTS key pair, the security of the entire scheme can be compromised, as it could allow an attacker to forge signatures.

XMSS achieves post-quantum security under minimal assumptions, relying only on the hardness of several properties of cryptographic hash functions. The public key size is very small, consisting only of the Merkle root and a public parameter. The signature sizes are also practical, as they include just one OTS signature and

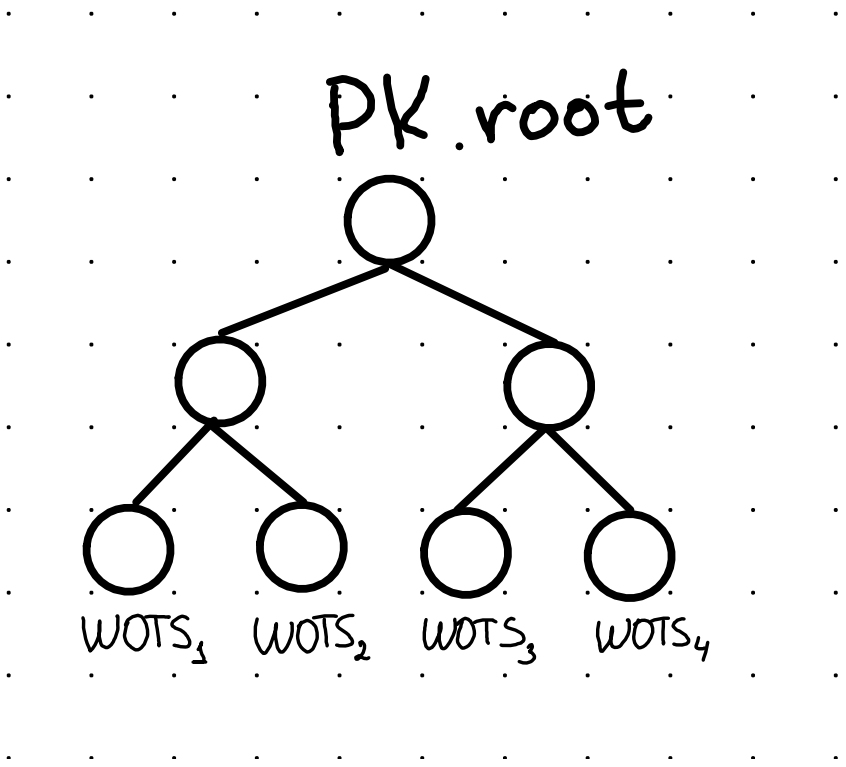


Figure 5: XMSS tree example.

one authentication path. Furthermore, XMSS is standardized (RFC 8391 [20]) and recommended by NIST for post-quantum cryptography.

We want to highlight again that the core disadvantage of XMSS is that it is stateful, which makes deployment in practical systems more complex and potentially risky. Another limitation is that the number of signatures that can be produced is fixed in advance by the tree height h, allowing at most 2^h signatures per key pair. Note that for all of the hash-based signatures the total number of signatures is limited. But for schemes like XMSS^{MT} and SPHINCS$^+$ we can make this number as big as 2^{64} or even bigger. For XMSS the key generation algorithm would have to generate all 2^h OTS public keys, hence bounding the total number of signatures by a much smaller value than 2^{64}. This also affects signing, which can be computationally costly, since computing the authentication path requires knowledge of the entire tree. This means that one must either cache the tree or be able to recompute it for each signature. The large values of h make both key generation and signing impractical.

8 XMSS^{MT}: Hypertree XMSS

While XMSS provides compact and secure hash-based signatures, scaling to large numbers of signatures with a single tree is impractical. A tree of height h supports 2^h signatures but signing requires a costly computation of the authentication path, since it involves updating or recomputing many intermediate nodes (in the worst case, all of them including the regeneration of 2^h OTS instances). To address this, the hypertree variant of XMSS, called XMSS^{MT}[5] [23], was proposed. XMSS^{MT} is specified in RFC 8391 [20], which supports up to 2^{60} signatures.

Structure. XMSS^{MT} organizes the signing capability into a hypertree of d layers of Merkle trees. Instead of building one very tall Merkle tree of height h, the scheme constructs d smaller trees, each of height $h' = h/d$. The root of each lower-level tree is signed by a one-time signature scheme using a key from the tree above it. The topmost tree's root becomes the global public key. Thus, signing a message involves traversing multiple layers, but each individual Merkle tree is smaller and therefore more efficient to handle.

Signing. To sign a message:

1. The signer begins at the top tree. A WOTS key pair from this tree is used to authenticate the root of the next tree.
2. This process continues recursively until the signer reaches the bottom tree, where a WOTS key pair is used to sign the randomized hash of the actual message.
3. The full signature contains the WOTS signatures at each layer along with the authentication paths proving the correctness of each Merkle root down to the message signature, as well as the randomness used to compute the hash.

Verification. The verifier proceeds as follows:

1. Compute the randomized hash of the message.
2. Reconstruct the OTS public key from σ_{OTS} of the bottom layer and the hash of the message.
3. Using the OTS public key as the leaf value at position i, recursively hash it together with the provided authentication path up to the root of the bottom layer Merkle tree.
4. Use this root and σ_{OTS} of the next layer to reconstruct the OTS public key of the second layer from the bottom.
5. With the help of the provided authentication path recompute the root of the Merkle tree at this level.

[5] Originally it was called Multi Tree XMSS.

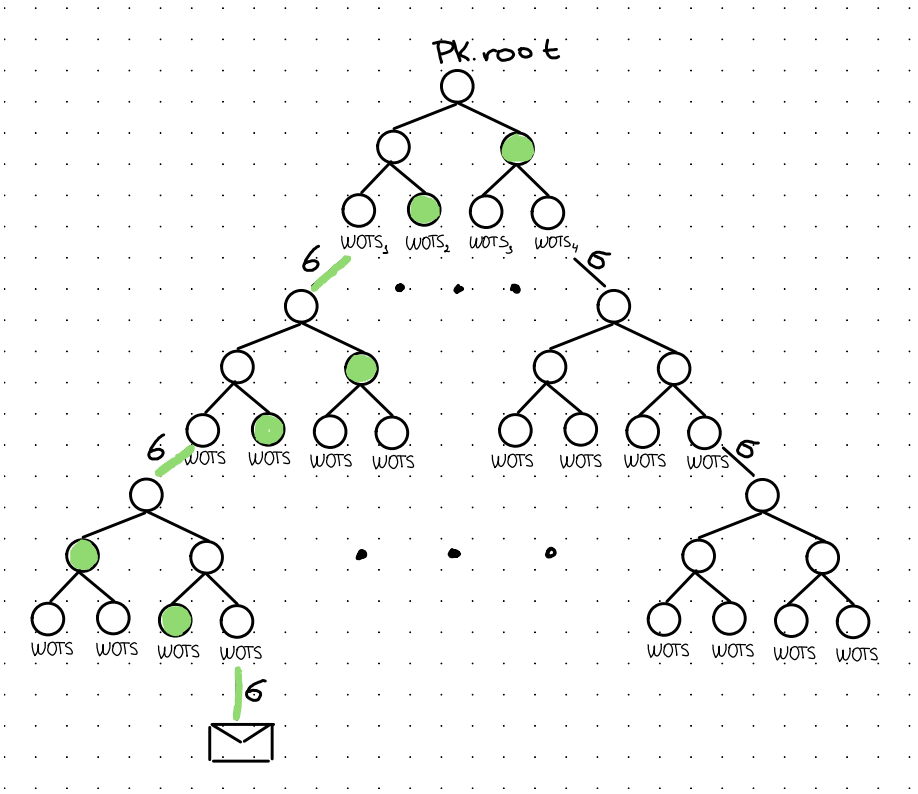


Figure 6: XMSS^{MT} hypertree example.

6. Continue this process, until you reach the root of the top-layer Merkle tree.

7. Compare the result with the global public key root; accept if they match.

Performance. By replacing one tall tree with multiple smaller ones, the cost of computing and updating authentication paths is reduced. During key generation, only the top-layer tree of size $2^{h/d}$ needs to be computed. During signing, only the d trees on the path from the root to the used index need to be generated, each of size $2^{h/d}$, along with their WOTS instances. This makes XMSS^{MT} practical even for very large numbers of signatures.

The number of layers d provides a tradeoff: larger d means faster signing and key generation, but signatures grow linearly with d since each layer contributes one WOTS signature and one authentication path. Verification time also increases with d.

Like XMSS, XMSS^{MT} is stateful. Each OTS key pair must be used exactly once, making state management a main practical challenge.

An example of an XMSS^{MT} signature can be found in Fig. 6.

9 FORS: Forest of Random Subsets

The *Forest of Random Subsets* (FORS) signature scheme is a hash-based signature primitive designed to help in the construction of stateless, many-time signature schemes. It was introduced as part of the SPHINCS$^+$ design, where it is used after the OTS of the bottom layer of XMSS^{MT} as the building block that signs the message and connects with a large hypertree structure.

Basic idea. FORS can be viewed as a collection (or "forest") of k small Merkle trees, each of height a. A message to be signed is first mapped onto k indices, each a bits long. For each index, the signer reveals the corresponding secret key leaf value together with its authentication path in the small Merkle tree. By going through each tree "in the forest", the verifier can reconstruct the relevant leaves and authenticate them against the published FORS public key (the hashed roots of all trees). An example of a FORS signature is given in Fig. 7.

Signing.

1. Given a message m, compute its randomized digest using a hash function and randomness R, then split it into k indices $(i_1, \dots, i_k)$, each a bits long.
2. For each index i_j, the signer reveals the corresponding secret key element from the j-th tree, together with the authentication path proving its membership up to the root of that tree.
3. The FORS signature is the collection of these secret key elements and authentication paths, as well as the randomness R.

Verification.

1. Compute the message digest using the randomness R from the signature.
2. For each revealed secret key element and authentication path, recompute the corresponding tree root.
3. Hash all t roots together to reconstruct the FORS public key value.
4. Compare the computed FORS value with the published public key; accept if they match.

9.1 FORS as a Few-Time Signature Scheme

An important property of FORS is that it is not strictly a one-time signature scheme (like WOTS), nor a full many-time scheme (like XMSS or SPHINCS$^+$). Instead, FORS is best described as a *few-time signature scheme*.

A few-time signature scheme can be securely used more than once, but not too many times. In FORS, this follows from the way signatures reveal secret key material: each signature reveals a small number of leaves from the underlying Merkle trees

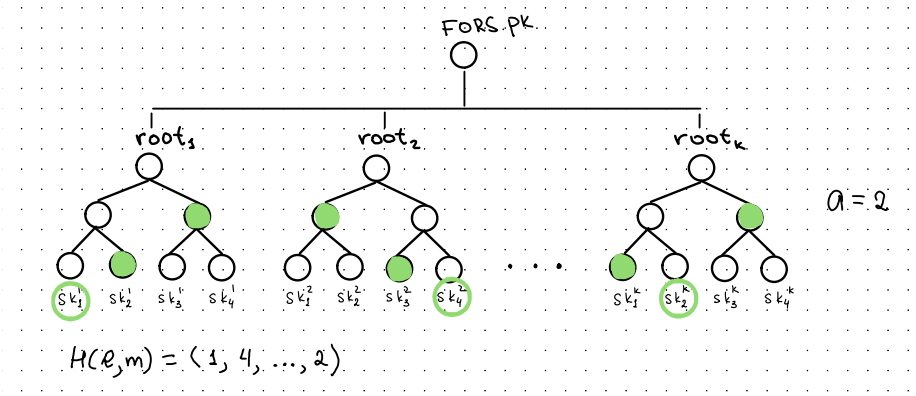


Figure 7: FORS example.

(along with their authentication paths). If only one signature is produced, the adversary learns little. However, with each additional signature, more leaves are exposed, and the probability that an adversary can reconstruct or forge a valid signature increases. The trivial attack requires from the adversary to find a message that will be hashed to already revealed indices of the FORS instance.

Security degradation. The security of FORS therefore, degrades with the number of signatures:

- After one signature, only t leaves are revealed (one from each small tree), leaving the majority of the secret key hidden.
- After multiple signatures, the set of revealed leaves grows, and eventually the adversary may collect enough information to forge a signature on a new message.

Formally, if FORS is instantiated with k trees of height a, then each signature reveals k leaves. The total output space range size is $(2^a)^k$. After seeing a single signature the adversary has a single option for the first tree, a single option for the second tree and so on. Hence, the number of possible hash values that the adversary would be able to sign with the revealed secret elements is $1 \cdot 1 \cdot \ldots \cdot 1 = 1$. After seeing a second signature for the same FORS instance, more secret values will be revealed. There is a chance that for every FORS tree a new secret element will be revealed, hence the adversary now has $2 \cdot 2 \cdot \ldots \cdot 2 = 2^k$ satisfying message digests. With q signatures, an adversary sees $\leq q \cdot k$ leaves, which gives $\leq q^k$ targets, so the effective security decreases roughly in proportion to the number of exposed leaves. In the extreme case, if enough signatures are observed to cover a significant fraction of the leaves, security is no longer guaranteed.

Implications. In practice, this property means that FORS alone is only secure for a limited number of signatures (e.g., 7 signatures for SPHINCS$^+$ parameters given in

Table 1). To build a full stateless scheme like SPHINCS$^+$, FORS is combined with a large hypertree structure. This way, the few-time nature of FORS is combined into a design that safely allows a very big number of signatures overall.

9.2 FORS+C

The core improvement of FORS+C over FORS is to reduce the number of authentication paths while maintaining the same level of security (and the same running time of all algorithms). The idea is to force the hash for the last tree to always open the first leaf. This is equivalent to requiring that the last a bits of the digest of the message that is signed by FORS are all zeros. This can be achieved by grinding, i.e., repeatedly trying different inputs until the hash output ends with a zero bits.

Consequently, only signatures that reveal the first leaf in the last tree are valid, which the verifier can check directly from the digest value. Thus, the signature itself *does not require the actual authentication path.* This means that we now only need to store the grinding values (e.g., a counter or just randomness) in the signature instead of the whole authentication path of the last tree. Moreover, there is no need for the signer to compute the last tree. See an illustration in Fig. 8.

On average, this will require trying 2^a hash function evaluations before finding a suitable counter and digest. However, this cost is offset by not needing to generate the last tree (which also requires 2^a hashes), thus keeping the running time of the signer approximately the same. Moreover, as a result, verification is also (slightly) faster, as it has one less tree to verify. This results in signatures that are strictly better, allowing us to generate smaller signatures that are faster to verify with the same signature time.

As final remarks, let us discuss the message hashing in more details. The message hash in SPHINCS$^+$ takes PK.seed, and PK.root, randomness R and the message m. PK.seed, and PK.root are mainly used for domain separation. The randomness is needed to avoid collision attacks. In practice one can generate this randomness by using a PRF function, that takes a secret seed and a message. To add the possibility of grinding there are two main options: we can add a counter i to the message hashing, or add a counter i to the PRF function that samples randomness for the message hash.

In the case of adding the counter directly to the message hash, the signer first computes R and then iterates over values of i until it finds one where the digest of the message with the counter ends with a bits of zero and signs it. This option was suggested in the original SPHINCS+C paper, and it requires fewer hash function calls, but needs to store the resulting value of the counter in the signature.

In the case of adding the counter to the PRF function, the signer recomputes a new value for R for each attempt, and checks if the message hash with the new value of R gives a satisfying digest. This option allows for avoiding the need to add a counter to the signature (because the randomness is already part of the signature), but adds an invocation of a PRF function for each search attempt.

If we consider the security analysis for the required property of the hash function in the Quantum Random Oracle model, then the first approach is typically analyzed using generic attacks, while the second approach is amenable to a provable security analysis (though a formal security reduction for FORS+C has not yet been established).

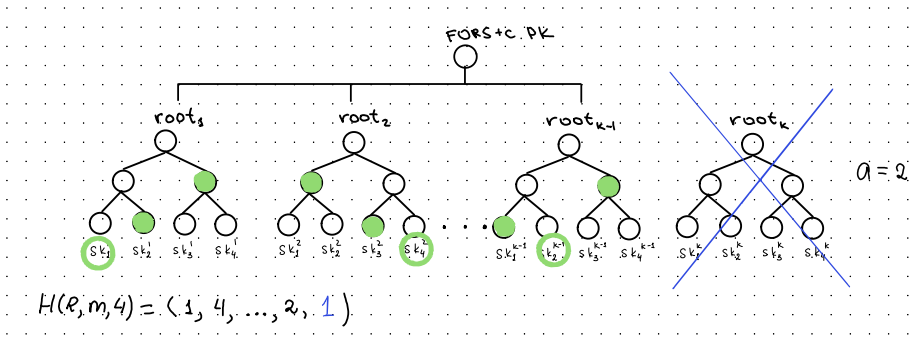


Figure 8: FORS+C example.

10 PORS+FP

A recent work [1] by Abri and Katz proposes another modification for SPHINCS$^+$-like schemes, substituting the FORS (+C) scheme with a different few-time signature scheme. They introduce *PRNG to Obtain a Random Subset with Forced Pruning* (PORS+FP), which builds on the PORS scheme [3].

The key idea of PORS+FP is to use a single, larger Merkle tree built from secret values, combined with a grinding search process. The message is hashed into k different leaves, determining which secret values must be revealed. The search ensures that the revealed leaves' authentication paths can be combined to produce a single compressed list of authentication nodes, which helps to reduce the size of the signature. As detailed below, this can yield even shorter signatures than FORS+C.

Let us start with a simplified version of PORS. For the scheme, we need a special function $\mathsf{H}_{\binom{t}{k}} : \mathcal{R} \times \mathcal{M} \to \mathcal{B}_{t,k}$, where $\mathcal{B}_{t,k}$ denotes a space of all possible subsets of k distinct values from a set $[0; t-1]$. Hence, $|\mathcal{B}_{t,k}| = \binom{t}{k}$. In PORS, a (possibly unbalanced) Merkle tree with t leaves is generated. The root of this tree serves as the public key of the scheme. An example of the simplified PORS signature scheme is given in Fig. 9.

PORS Signing.

1. Given a message m, compute its randomized digest, using $\mathsf{H}_{\binom{t}{k}}$ and randomness R. We denote the digest as $(i_1, \ldots, i_k)$, which represents k different indices from $[0; t-1]$.

2. For each index i_j, the signer reveals the corresponding secret key element from the Merkle tree, together with the authentication path proving its membership up to the root of the tree.

3. The PORS signature is the collection of these secret key elements and authentication paths, as well as the randomness R.

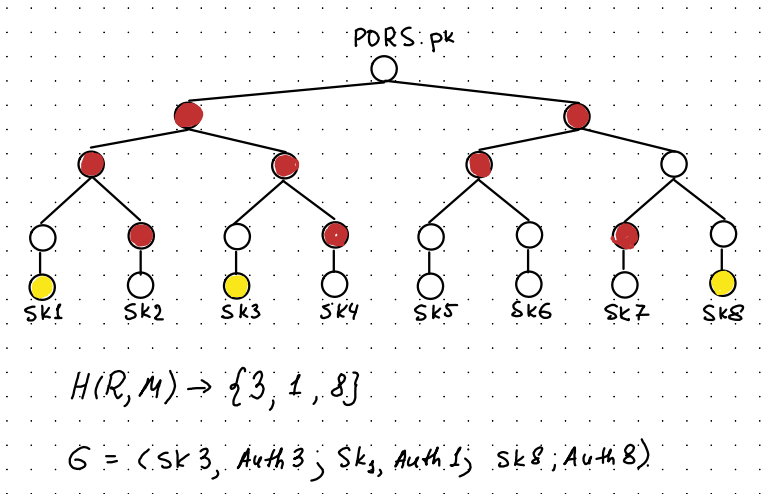


Figure 9: Simplified PORS example.

PORS Verification.

1. Compute the message digest using the randomness R from the signature.
2. For each revealed secret key element and authentication path, recompute the tree root.
3. Compare each tree root with the root of the Merkle tree; accept if all the computed roots match.

One can use the fact that all these authentication paths are in the same Merkle tree. To reduce the signature size it is possible to combine them into a single authentication set using an algorithm called Octopus [2, 3]. The Octopus algorithm produces an optimal set of nodes required for authentication; we provide a description of the algorithm in Appendix C. Subsequently, one can search for a message hash (by integrating a counter into the message hash or sampling new randomness) that yields a small Octopus set size. A boundary on the size of the authentication set will determine the signature size. The lower this boundary is set, the harder it is to find a valid hash, thereby requiring more computational effort from the signer. The computations during the search process include the evaluation of $\mathsf{H}_{\binom{t}{k}}$ and the execution of the Octopus algorithm to determine the size of the authentication set. According to Abri and Katz [1], the runtime of the grinding is dominated by the hash evaluation time and thus Octopus evaluation has a negligible impact on runtime.

Given a boundary $m_{\max}$, the resulting algorithms are the following:

PORS+FP Signing.

1. Sample randomness R.

2. Given a message m, compute its randomized digest, using $\mathsf{H}_{\binom{t}{k}}$ and randomness R. We denote the digest as $(i_1, \ldots, i_k)$, which represents k different indices from $[0; t-1]$.
3. Compute the authentication set S for $(i_1, \ldots, i_k)$.
4. If $|S| > m_{\max}$: Go to step 1.
5. For each index i_j, the signer reveals the corresponding secret key element from the Merkle tree.
6. The PORS+FP signature is the collection of these secret key elements and authentication set S, as well as the randomness R.

PORS+FP Verification.

1. Reject if $|S| > m_{\max}$.
2. Compute the message digest using the randomness R from the signature.
3. Using the revealed secret key elements and authentication set S, recompute the tree root.
4. Compare the tree root with the root of the Merkle tree; accept if the computed root matches.

To make the signature constant size one can pad the authentication set S to the size $m_{\max}$.

Another important question here is how to implement the $\mathsf{H}_{\binom{t}{k}}$ function. Several previous works [39, 3] suggest possible implementations. Reyzin and Reyzin [39] propose two algorithms that do not produce a uniform distribution over the whole domain $\binom{t}{k}$. The functions take a b-bit string, such that $2^b \leq \binom{t}{k}$ and use a deterministic encoding to the subset of $\binom{t}{k}$. It is possible to improve upon this approach and hash the message to n-bit value, such that $2^{n-1} < \binom{t}{k} \leq 2^n$. Then using a rejection sampling mechanism we resample, until the digest value is in $[0; \binom{t}{k}]$. This way, we can again get a uniform sample from the required range.

Aumasson and Endignoux [3] propose another hashing algorithm. It produces both an index of a PORS+FP instance (in case it is used in a SPHINCS$^+$-like hypertree structure), and a sample from $\mathcal{B}_{t,k}$. The key idea is to use a PRF to first derive a PORS+FP index, and then to use the same PRF with different tweaks to generate candidate leaf indices, collecting them until k distinct values have been obtained. The algorithm as it is presented in [3] requires several minor modifications to obtain a uniform mapping into $\mathcal{B}_{t,k}$.

We propose an alternative hashing approach. If the output of the hash function is large enough to encode both the PORS+FP instance index and k values in $[0; t-1]$, we can hash the message together with a salt and extract all required values at once. If fewer than k distinct valid indices are obtained, we retry with a new salt. The final salt is included in the signature, so the verifier must perform only a single hash evaluation.

In the approach of Aumasson and Endignoux [3], the verifier must repeat all of the signer's PRF calls, because leaf indices are derived iteratively. Extracting all indices at once avoids this overhead, but requires a sufficiently large hash output satisfying $n > h + k \cdot \lceil \log_2(t) \rceil$. The algorithm is presented in Algorithm 1.

Algorithm 1: Hash to Obtain a Random Subset

Input: message $m \in \mathcal{M}$, hash function $H : \mathcal{R} \times \mathcal{M} \to \{0,1\}^n$; $n > h + k \cdot \lceil \log_2 t \rceil$

Output: Randomness s, hyper-tree index $\tau \in [0; 2^h - 1]$ and k distinct indices $x_i \in [0; t-1]$

1 $s \leftarrow_{\$} \mathcal{R}$
2 $y \leftarrow H(s, m)$
3 Let τ be the first h bits of y.
4 $X \leftarrow \emptyset$
5 Split last $n - h$ bits of y into $v = \left\lfloor \frac{n-h}{\lceil \log_2 t \rceil} \right\rfloor$ blocks of $\lceil \log_2 t \rceil$ bits, $y = \tau \| b_1 \| b_2 \| \cdots \| b_v$
6 **for** $i \in \{1, \ldots, v\}$ **do**
7 **if** $b_i \in [0; t-1] \wedge |X| < k$ **then**
8 $X \leftarrow X \cup \{b_i\}$
9 **if** $|X| < k$ **then**
10 GoTo step 1.
11 $(x_1, \ldots, x_k) \leftarrow \text{sorted}(X)$
12 **return** $(s, \tau, x_1, \ldots, x_k)$

11 SPHINCS$^+$

SPHINCS$^+$ is a stateless hash-based signature scheme that has been standardized by NIST as SLH-DSA (Stateless Hash-based Digital Signature Algorithm) in FIPS 205 [34]. It builds on the concepts of WOTS, XMSS, and FORS. SPHINCS$^+$ uses WOTS-TW and FORS as its building blocks. The rest of the construction remains the same.

From OTS to FTS. The core idea of SPHINCS$^+$ is to use the one-time signature (OTS) at the bottom layer of the hypertree to sign a *few-time signature* (FTS), namely the FORS scheme. The FTS is then used to sign the message hash. This allows the same index to be used several times without compromising security, as long as the number of signatures per leaf remains bounded. In contrast, XMSS^{MT} required strict one-time usage of each leaf, making it inherently stateful. By adopting FORS, SPHINCS$^+$ avoids state management entirely.

Hypertree size. To ensure security, the hypertree in SPHINCS$^+$ must be large enough that when selecting q_s leaves uniformly at random, the probability of reusing the same leaf (i.e., the same FORS instance) too often remains negligible. This ensures that the

few-time property of FORS does not degrade the overall security of the scheme significantly. Thus, the hypertree plays the role of distributing signatures across many FORS instances, mitigating the risk of overuse.

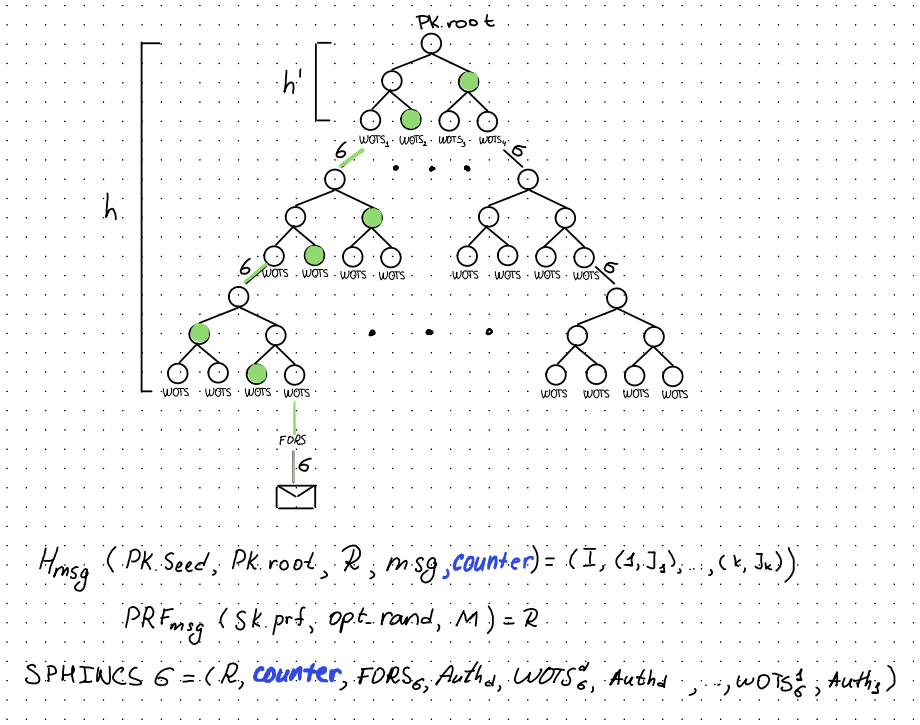


Figure 10: SPHINCS$^+$ tree example.

Leaf selection. For each signature, the actual leaf index is chosen pseudorandomly by hashing together the message, the public key, and a randomness value:

$$(\text{idx}, (1, i_1), \ldots (k, i_k)) \;=\; \mathbf{H_{msg}}(\mathsf{PK.seed}, \mathsf{PK.root}, R, m)\,.$$

Here, H is a cryptographic hash function and R is per-signature randomness. This mechanism ensures that the selected leaf (idx) is unpredictable, preventing an adversary from forcing the signer to reuse a particular FORS instance.

Randomness generation. The randomness value R itself is generated via another hash function that takes as input a secret seed and the message:

$$R \;=\; \mathbf{PRF_{msg}}(\mathsf{SK.prf}, \text{opt}, m),$$

where the optional input *opt* can be used to provide additional entropy (e.g., from an external randomness source), though this is not strictly required.

One can substitute parts of SPHINCS$^+$ to obtain different schemes, for example substituting WOTS-TW with WOTS+C and FORS with FORS+C will result in the SPHINCS+C scheme. It is also possible to substitute FORS with PORS+FP. We can also consider different combinations, such as using WOTS+C and PORS+FP together, keeping the core SPHINCS$^+$ framework. For PORS+FP and FORS+C we have to do some grinding during the message hashing.

For the grinding cases (FORS+C or PORS+FP), we can go for a counter in the message hash, i.e., $\mathbf{H}_{\mathbf{msg}}$ will be called as $\mathbf{H}_{\mathbf{msg}}(\text{PK.seed}, \text{PK.root}, R, m, \text{counter})$ or we can put the counter in the $\mathbf{PRF}_{\mathbf{msg}}$: $\mathbf{PRF}_{\mathbf{msg}}(\text{SK.prf}, \text{opt}, m, \text{counter})$, as discussed in Section 9.2. For this document we sample new randomness for each grinding attempt, i.e., we use $\mathbf{H}_{\mathbf{msg}}(\text{PK.seed}, \text{PK.root}, R, m)$ where

$$R = \mathbf{PRF}_{\mathbf{msg}}(\text{SK.prf}, \text{opt}, m, \text{counter}) .$$

In case of PORS+FP we also need to use $\mathbf{H}_{\mathbf{msg}}$ with a sufficiently large output size, to account for mapping into $\mathcal{B}_{t,k}$.

An example of a SPHINCS$^+$ structure is given in Fig. 10.

12 Security Levels and Quantum Attack Complexity

NIST's post-quantum security categories [35] are often communicated in terms of attack costs against well-studied primitives. At the low end, "Category/Level 1" is typically aligned with the effort required to break 128-bit symmetric security (e.g., recovering an AES-128 key) and, depending on the comparison being made, is sometimes informally described as a preimage search on an n-bit hash function with $n = 128$.[6] At the high end, "Category/Level 5" corresponds to the complexity of key search on a block cipher with a 256-bit key (e.g., AES-256). In this section we focus on the Level 1 security level, which matches the current 128-bit classical security of Bitcoin (due to the secp256k1 elliptic curve and SHA-256 collision resistance). We examine the complexity of the relevant attacks and argue that Level 1 provides sufficient security guarantees for Bitcoin.

Assuming oracle access to a random function $H : \{0,1\}^* \to \{0,1\}^n$ and a given output y, classically finding x with $H(x) = y$ requires a *query* complexity of $\Theta(2^n)$ in expectation. Quantumly, Grover's algorithm [17] reduces this to $\Theta(2^{n/2})$, which is asymptotically optimal [4, 47].

This section unpacks what the big-O notation hides: the constant factors, success probability management, and (crucially) the circuit resources needed to implement a reversible oracle for H, the Grover diffusion operator, and extra overheads to make the computations fault-tolerant. We also discuss classical and quantum parallelism, including why k quantum processors yield only a $\sqrt{k}$ wall-clock speedup for unstructured search[7].

[6] Formally, NIST's categories are specified relative to concrete targets such as AES-128/192/256 key search and SHA-256/384 collision targets; informal descriptions using hash preimage of n bits" are only approximations.

[7] Unstructured search - search over a domain where the only way to distinguish the target element from non-targets is by querying an oracle, and no additional structure (algebraic, combinatorial, or otherwise) can be exploited to accelerate the search.

12.1 From Query Complexity to Real Resources

Grover's algorithm performs $T \approx \frac{\pi}{4}\sqrt{N/M}$ iterations where N is the search space size and M is the number of solutions ($N/M \approx 2^n$ for preimage search). Each iteration performs:

1. One call to the *oracle* O_H marking solutions ($|x\rangle|b\rangle \mapsto |x\rangle|b \oplus [H(x) = y]\rangle$),
2. One *diffusion* (inversion-about-the-mean) operator on the n-qubit index register.

Big-O statements like $O(2^{n/2})$ count *oracle queries*. In practice, we must translate each query and each diffusion into low-level quantum gates and then into fault-tolerant space-time resources.

12.2 Best-Known Toffoli Count Implementations of SHA-256

A crucial ingredient in estimating the cost of Grover-based preimage attacks on SHA-256 is the resource estimate for a reversible implementation of the hash function. In quantum resource analyses, this cost is typically expressed in terms of Toffoli gates. A Toffoli gate is a CCNOT gate (two control qubits and one target qubit). It is convenient to use Toffoli gates as the primary cost metric because any reversible circuit can be built from them. The main complexity measures are the *Toffoli count* (how many Toffoli gates are used in total), *Toffoli depth* (critical path length of Toffoli gates), and required ancilla qubits. Here we will focus just on the Toffoli depth.

In [27, Section 6.3] the authors discuss the Toffoli depth of a preimage search for SHA-256. They calculate this value by giving an estimates on the complexity of a single SHA-256 hash call and extra costs related to the Grover's search algorithm:

$$\text{cost(Grover iteration)} = 2 \cdot \text{cost(SHA-256)} + \text{cost(C}^{256}\text{NOT)} + \text{cost(C}^{266}\text{NOT)},$$

where C^nNOT is a n-fold controlled-NOT gate. The smallest cost of SHA-256 implementation they suggest is 10112 Toffoli depth. They also claim that Toffoli-depth and the work qubits required for lower-depth (less-qubit) C^{256}NOT gate are 509 (2024) and 254 (1), respectively (we will take this estimate for the C^{266}NOT as well). Hence, a single Grover iteration requires Toffoli depth of size:

$$2 \cdot 10112 + 509 + 509 = 21242 \approx 2^{14.4}.$$

The case of a hash function with 128 bits output can be made from SHA-256 hash function by just truncating 128 bits. Hence to find a preimage of such 128-bit hash, the Grover iteration cost will be the same as for the SHA-256, which we estimate $\approx 2^{14} \cdot 2^{64} = 2^{78}$ Toffoli depth.

To give a comparison, Shor's algorithm [42] for computing discrete logarithms on the secp256k1 elliptic curve would roughly require a circuit of 2^{37} Toffoli depth [40, Table 2].

Implications. These numbers highlight two points. First, the cost of a "single Grover iteration" is not just one black-box oracle call: it involves tens of thousands of Toffoli gates once the SHA-256 oracle is implemented reversibly. Second, there is a tradeoff between *depth* and *width*: more qubits can be used to reduce depth, but this increases the requirement on the number of achievable logical qubits. Thus, resource estimates for Grover's algorithm on SHA-256 must specify both depth and width, not merely the asymptotic $O(2^{n/2})$ iteration count.

The presented reversible SHA-256 implementations achieve Toffoli depth around 10^5 with width ≈ 938 qubits. As a result we get $\approx 2^{78}$ Grover's search cost, instead of 2^{64} for a SHA-256/128 hash function. While a better implementation of a SHA-256 can appear in the future, the constant in the Big-O notation will probably remain high and play an important role in the estimations of the efficiency of a quantum adversary.

A broader discussion was given in the talk by Sam Jaques: Invited talk II by Sam Jaques (CHES 2024), where he argues that AES-128 probably won't be broken by quantum computers in our lifetime and possibly never at all.

12.3 Parallelism: Classical vs. Quantum

For unstructured search over $N = 2^n$ items with number of solutions $M = 1$:

Classical. Partitioning across k processors gives an *almost linear* speedup: each processor searches N/k candidates in $\Theta(N/k)$ time; wall-clock time decreases by $\approx k$ (ignoring overhead).

Quantum. Partitioning the domain into k disjoint subsets and running Grover on each in parallel yields per-processor iterations

$$T_k \approx \frac{\pi}{4}\sqrt{\frac{N}{k}} = \frac{1}{\sqrt{k}} \cdot \frac{\pi}{4}\sqrt{N}.$$

Wall-clock time improves only by a factor of $\sqrt{k}$, not k. This $\sqrt{k}$ barrier is fundamental for unstructured search.

If you have k identical quantum computers, each must still execute a Grover run only $\sqrt{k}$ times shorter than a single-machine run; equivalently, to halve time-to-solution you must quadruple the number of quantum processors.

A good example for this case was given in the same talk by Sam Jaques: Invited talk II by Sam Jaques (CHES 2024). Consider a 56-bit DES key. A classical adversary would need 2^{56} iterations to break it. If each iteration takes 100 clock cycles, then a modern 5 GHz CPU would break DES in 46 years. The conclusion that he makes is that all realistic attacks are parallel, which become more difficult in the quantum case.

In NIST's 2017 call for post-quantum cryptography, they introduced "MAXDEPTH", a metric to account for this issue in security analysis (see Security (Evaluation Criteria)), where they used MAXDEPTH $= 2^{64}$ as "the approximate number of gates that current classical computing architectures can perform serially in a decade". Applying the same bound for quantum computers, would lead to a requirement of $(2^{78-64})^2 = 2^{28} \approx 268,000,000$ large scale fast quantum computers running in parallel for 10 years

to break the preimage resistance of 128-bit hash function (based on the current circuits).

12.4 Summary

In this document we consider Level 1 security category to be sufficient for Bitcoin (i.e. 128-bit hash functions), given that the attack requires $O(2^{n/2})$ operations for the quantum adversary. We emphasize that the big-O notation hides *big* constants in quantum resource estimates, including the costs of oracle reversibilization, diffusion operations, amplitude amplification, and fault-tolerance overheads. Additionally, quantum parallelism for unstructured search is limited to a $\sqrt{k}$ wall-clock speedup with k processors; there is no linear-in-k quantum analog of classical parallel brute force. Taking these factors into account, the preimage search would rather require $\approx 2^{78}$ Toffoli operations rather than 2^{64}.

13 Parameters

In this section, we discuss the main parameters that must be chosen for a hash-based signature scheme. A fundamental decision is whether to employ a *stateful* signature scheme. Stateful designs, such as XMSS or XMSS^{MT}, can provide shorter signatures and enable optimizations (e.g., tree traversal algorithms). However, they come with a severe drawback: the signer must carefully manage persistent state to ensure that each one-time key is used exactly once. If two signatures are ever generated from the same instance, the security of the scheme can no longer be guaranteed. We further investigate the stateful options in Appendix B.

For Bitcoin, stateful signatures present significant practical challenges. Users need to back up their keys, but restoring from outdated backups could lead to state reuse and enable forgeries. State management also introduces new risks for bugs and user errors, as well as a new threat model where adversaries could trick users into replaying old state. While stateful schemes could be introduced in parallel or as a later addition, we therefore focus on *stateless* signature schemes in the remainder of this discussion.

Main parameters. The key parameters to determine are:

- h: hypertree height;
- d: number of layers in the hypertree;
- n: hash function output size;
- $t = 2^a$: number of leaves in each FORS tree;
- k: number of FORS trees;
- w: Winternitz parameter.

Depending on the building blocks we use we also need to set $S_{w,n}$ for WOTS+C, and determine the total number of chains l in the WOTS-like signature. For the PORS+FP we set the number of leaves in a big Merkle tree to be $k \cdot t$.

As the first step we need to make several key decisions on how we will set these parameters. We present three possible approaches: based on generic attacks, based on provable security bounds combined with generic attacks against the properties, or based on provable security bounds and analysis of the properties in the Quantum Random Oracle Model. The second choice is whether we strictly require exactly λ bits of security (for example 128), or if we can accept a slightly smaller value.

The SPHINCS$^+$ authors used the first approach. They provide a script that requires the output of the hash function in the SPHINCS$^+$ structure to be λ bits (assuming that the generic attacks against such functions will require $\approx 2^\lambda$ classical queries) and uses an approximation for a generic attack against FORS.

In [21] a provable security bound was presented. The security of SPHINCS$^+$ was related to the security of a set of hash function properties. In [22] a similar bound for SPHINCS+C was given[8]. When we use the selected parameters from the standardized SLH-DSA [34], assuming a bound of 2^{-n} on the advantage for all security properties and a corresponding generic attack bound for FORS(+C), we will get approximately 123 bits of security.

One could go one step further and instead of assuming a 2^{-n} bound for the hash function properties, use a provable security bound for these properties in the Quantum Random Oracle Model. Due to proof techniques some of these bounds have factors up to 32. By plugging these bounds into the security proof this can result in a loss of approximately 10 bits. To compensate for this loss, one would need to use a hash function with a bigger output size: for example 138 bits. Such outputs are less common, but if we increase the hash function output to 256 bits the resulting signature size will increase significantly.

In the rest of this section we will follow the SPHINCS$^+$ approach. All parameter sets we present achieve $\lambda = 128$ bits of security, as discussed in Section 12. Hence, the output length of the hash function must match the desired security level, i.e., $n \geq \lambda$. For hypertree-based structures, $n = \lambda$ is essentially sufficient to reach the target security. We set the public key seed to be 128 bits and the $\mathbf{PRF}_{\mathbf{msg}}$ output to be 256 bits. Once n is fixed, the next major choice is the Winternitz parameter w. The $\mathbf{H}_{\mathbf{msg}}$ output size is determined by the size of the hypertree, which we discuss later in this section.

13.1 WOTS-TW, WOTS+C, and TL-WOTS-TW

To sign an n-bit message using Winternitz signatures, we require at least $\frac{n}{\log w}$ chains. The exact number of chains depends on the variant:

WOTS-TW. In WOTS-TW, checksum chains are added to protect against message modifications. Let $l_1 = \frac{n}{\log w}$ and $l_2 = \left\lceil \frac{\log(l_1(w-1))}{\log w} \right\rceil$. Then the total number of chains is $l = l_1 + l_2$.

[8] Note that FORS+C security was still based on the generic attack approach.

During key generation in SPHINCS$^+$, the signer computes all chains up to their final value to obtain the public key, requiring $l \cdot w$ hash evaluations. During verification, the expected number of hash computations is $l \cdot w/2$, since the verifier only needs to complete each chain on average halfway. Increasing w yields smaller signatures at the cost of higher computational overhead: signature size scales as $O\left(\frac{n}{\log w}\right)$, while signing and verification costs scale as $O\left(\frac{n}{\log w} \cdot w\right)$.

WOTS+C. In WOTS+C, the checksum chains are removed and replaced by a counter value, reducing signature size. Here, the total number of chains is simply $l = l_1$, and an additional parameter $S_{w,n}$ specifies the target sum for the encoded message. Only message hashes whose base-w expansion sums to $S_{w,n}$ are valid.

The number of possible valid encodings is:

$$\nu = \sum_{j=0}^{l} (-1)^j \binom{l}{j} \binom{(S_{w,n} + l) - jw - 1}{l - 1}, \tag{2}$$

At the extremes, choosing $S_{w,n} = l \cdot (w-1)$ yields only one valid encoding (requiring about 2^n trials to find), while choosing $S_{w,n} = l \cdot (w-1)/2$ maximizes the number of valid encodings. Thus, $S_{w,n}$ provides a tunable tradeoff between signing effort and verification effort. Since the verifier only needs to finish the chains, the higher the value $S_{w,n}$, the less the verifier needs to do.

TL-WOTS-TW. The TL modification is more complex (see Appendix A). The idea is to remap message encodings so that signatures reveal values closer to the top of the chains, reducing verification cost. Optimal encodings can be computed using the public script, and detailed parameter studies are available in the original paper [26, Tables 1 and 2].

The TL modification is most effective for small w, which results in longer chains and larger signatures. Since signature size is a primary constraint for Bitcoin, and TL also introduces implementation complexity, we do not consider it further. For applications where larger signatures are acceptable, such as when using SNARK aggregation [13], TL may be worth exploring.

13.2 FORS (+C), PORS+FP in SPHINCS(+C)

For FORS and FORS+C we need to satisfy the security requirements. These parameters highly depend on the number of allowed signing operations (q_s) under a single public key. For a given security bound λ, we require that

$$2^{-\lambda} \leq \sum_{r=1}^{q_s} \left(1 - \left(1 - \frac{1}{t}\right)^r\right)^k \binom{q_s}{r} \left(1 - \frac{1}{2^h}\right)^{q_s - r} \frac{1}{2^{hr}}.$$

For PORS+FP the requirement is:

$$2^{-\lambda} \leq \sum_{r=1}^{q_s} \frac{\binom{\min\{t, k \cdot r\}}{k}}{\binom{t}{k}} \binom{q_s}{r} \left(1 - \frac{1}{2^h}\right)^{q_s - r} \frac{1}{2^{hr}}.$$

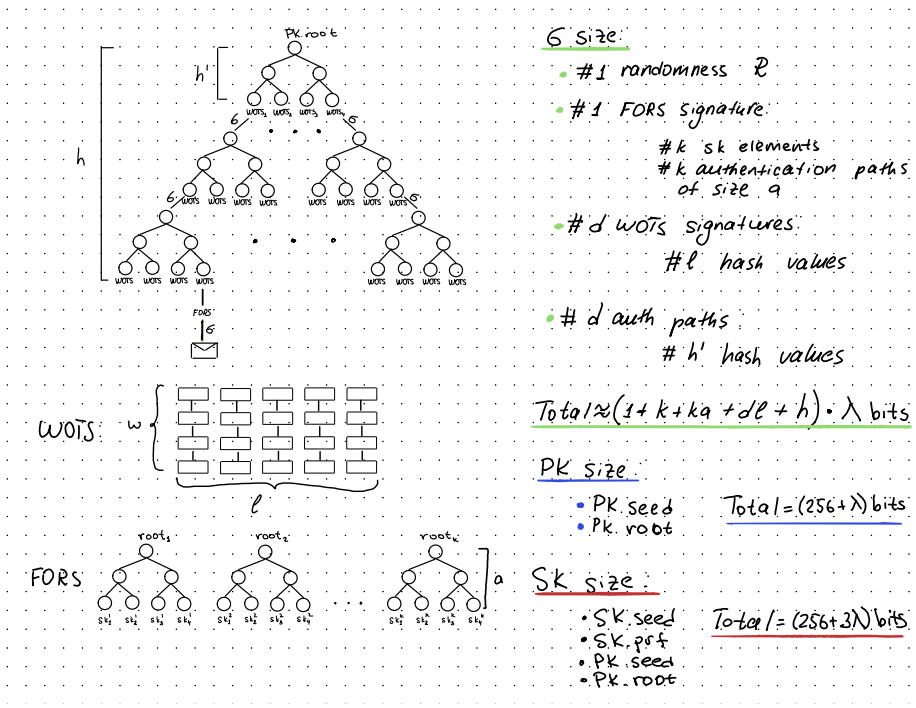


Figure 11: Size of SPHINCS$^+$ signature and keys.

The total size of the SPHINCS$^+$ (without the modifications) signature is

$$(1 + (a + 1)k + d \cdot l + h)\,\frac{\lambda}{8} \text{ bytes.}$$

This gives a rough estimate of how parameters affect signature size in SPHINCS$^+$-like constructions.

The security requirement constrains the choices of h, a, and k, while d only affects performance tradeoffs. Specifically, d controls the number of hypertree layers: increasing d reduces signing time (each layer requires fewer WOTS chains) but increases signature size and verification cost (since more WOTS signatures must be verified). The dependencies are presented in Figs. 11 to 13.

The NIST standardization process for post-quantum signatures required supporting 2^{64} signing operations. However, no one would be able to post 2^{64} SPHINCS$^+$ signatures to the Bitcoin blockchain, as this corresponds to roughly 14.6 billion terabytes of data. Without a block size increase, 2^{27} signatures of size 3KB would fill a year's worth of blocks, while 200 years of blockchain can include at most 2^{35} signatures. Since not all signatures end up on-chain, it makes sense to explore parameters supporting fewer signatures.

Regular on-chain wallets typically produce only one signature per public key, because for privacy reasons they generate a new public key for each payment. However, there are several reasons to sign multiple times with the same public key:

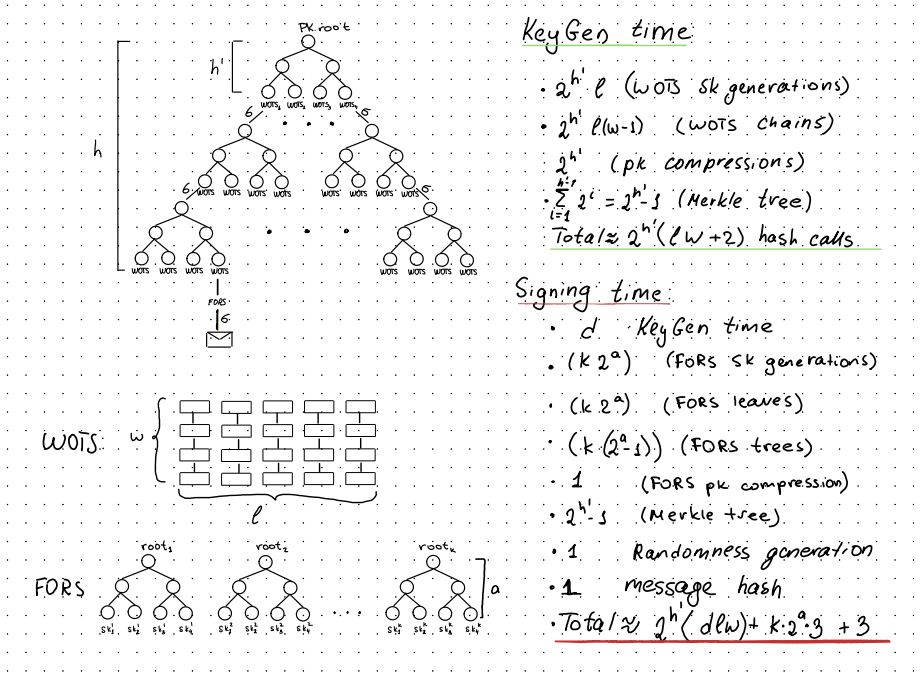


Figure 12: Number of hashes for KeyGen and Sign of SPHINCS$^+$.

- The wallet signed a transaction, but the transaction fee was insufficient to confirm in time. The wallet can then sign a replacement transaction spending the same inputs with an increased fee.
- The sender (or multiple senders) accidentally send multiple payments to the same address.
- The receiver published a single address, expecting multiple people to pay to it. This is simpler at the cost of reduced privacy, because generating a new address per sender requires some form of interaction (in the absence of Silent Payments [24]).
- In some Layer 2 protocols, transactions spending the same outputs are signed frequently, but only a single transaction is eventually written to the chain. Such protocols are not bound by the size of the blockchain.
- A static public address can be beneficial for auditability, allowing everyone to see the amounts received.
- Users may want to provide proofs of control [41] of funds by producing signatures without spending.

On the other hand, reducing the number of supported signatures too much makes the scheme effectively stateful. For example, a scheme supporting only 2^3 signatures

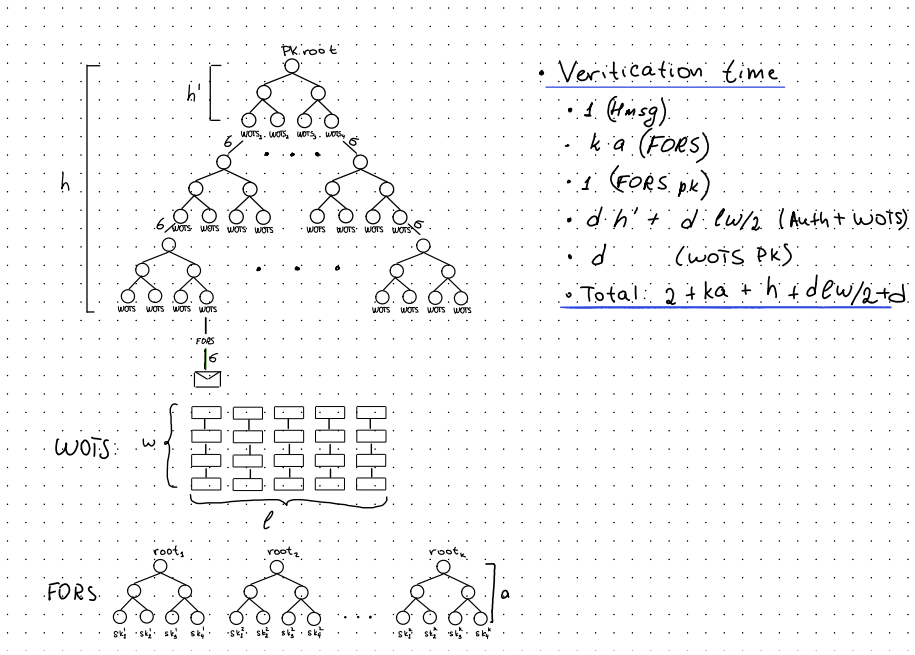


Figure 13: Number of hashes for Verify of SPHINCS$^+$.

could not be considered stateless, because practical implementations would need to remember the number of signatures already produced to prevent forgeries. An adversary could continually request signatures until enough have been produced to enable a forgery. With 2^{40} allowed signatures this is less of a concern: using a highly optimized implementation [18], producing 2^{40} signatures would take a single 4.5 GHz core roughly 55 years. Moreover, many signing devices require manual approval, making it difficult to imagine more than $2^{20} \approx 1{,}000{,}000$ approvals before such an attack would be detected. Even in automated settings, software rate-limiting can provide some protection.

Supporting 2^{30} or 2^{40} signatures, therefore, appears more realistic for blockchain applications than the full 2^{64} target. As discussed in A note on SPHINCS+ parameter sets [28], reducing the maximum number of supported signatures yields substantial efficiency gains (a quick overview [29]). For instance, compressing from 2^{64} to 2^{30} signatures reduces signature size from about 7856 bytes to 3888 bytes. The script from the SPHINCS$^+$ team or the complete parameter set notes can be used to explore such parameter sets. For 2^{40} signatures, the lowest suggested sizes are around 4144 bytes.

In summary, careful parameter selection, especially for w, h, d, and FORS parameters, enables tailoring SPHINCS$^+$-like schemes to the Bitcoin setting. As shown in Table 1, reducing the number of supported signatures and implementing WOTS+C and either FORS+C, or PORS+FP modifications significantly reduces signature size as well. Note that an important parameter is the number of compression function calls

that occur in each hash function invocation. While the calls in the hypertree structure will use at most two compression calls, the message hash might use more. If we use the signature scheme in a similar manner to how it is currently used in Bitcoin, then we will simply sign a 256-bit digest, which results in two compression function calls for $\mathbf{H}_{\mathbf{msg}}$ as well.

If 2^{20} signatures suffice, the parameters in Table 2 offer further size reductions.

13.3 Exploring Parameter Tradeoffs

In this section we explore possible parameters and different modification of the original SPHINCS$^+$ scheme. The scripts used to compute the security levels, sizes, and costs of the schemes are available in our repository [6].

Signing time is probabilistic: a brute-force search is required to find a good digest for the WOTS+C, FORS+C and PORS+FP instances. In the tables (Table 1, Table 2), we added a column that shows the amount of hashing effort consumed by the search during signing (Expected Search). That value represents the expected time, but actual search times may be shorter or longer. This value is part of the total signing time, given in the corresponding column.

Note that for the PORS+FP scheme (for the parameters in Section 13), 512-bit output hash value on expectation should be sufficient to extract k unique values and an instance pointer $i \in [0; 2^h - 1]$. Hence, we propose to use a hash function with sufficiently large output size (for example SHA-512). In practice, if one did not get enough sample in the computed digest, one can recompute with new randomness (see Algorithm 1). For simplicity, in the Tables 1 and 2 we assume that a single hash function invocation is sufficient to obtain the unique subset. For the presented parameters we approximate this by 2 compression function calls (although for SHA-512 one compression function call would be sufficient).

We mark in bold the most interesting parameter sets from our perspective that are presented in this table. We wanted to highlight, that the parameters presented here are not meant to be a proposal, but rather a high-level overview of possible tradeoffs.

To compare verification time with Schnorr signatures as specified in BIP 340 [46], which are currently deployed on Bitcoin, we look at verification time per byte of signature. Verifying a 64-byte Schnorr signature using libsecp256k1 on an Intel i7 4.5 GHz CPU takes approximately 28us, equivalent to 0.438us per byte. On the same machine, a single SHA-256 compression function invocation takes 0.168us for a naive C implementation and 0.135us when compiled with `-march=native`. Using compression function invocations as a proxy, we want at most 2.6 invocations per byte of signature. For all the parameters (except one with ratio 2.61) presented in the tables this bound is satisfied.

	q_s	h	d	a	k	w	l	$S_{w,n}$	SigSize (byte)	SigTime (hashes)	SigTime (compr.)	Exp. Search (hashes)	WC Search (hashes) $p = 2^{-30}$	SigVerify (hashes)	SigVerify (compr.)	Compr./ Byte
SPX	2^{64}	**63**	**7**	**12**	**14**	**16**	32 + 3	–	**7872**	$\mathbf{219 \cdot 10^4}$	$\mathbf{228 \cdot 10^4}$	**0**	**0**	**2088**	**2387**	**0.30**
W+C	2^{40}	44	4	16	8	16	32	240	4976	$578 \cdot 10^4$	$638 \cdot 10^4$	264	1907	1150	1357	0.27
		44	4	16	8	16	32	304	4976	$579 \cdot 10^4$	$639 \cdot 10^4$	5344	39021	894	1101	0.22
		44	4	16	8	256	16	2040	3952	$3515 \cdot 10^4$	$3571 \cdot 10^4$	2996	21854	8350	8541	2.16
		40	5	14	11	256	16	2040	4612	$579 \cdot 10^4$	$598 \cdot 10^4$	3745	23589	10417	10635	2.31
		40	5	14	11	256	16	2840	4612	$594 \cdot 10^4$	$612 \cdot 10^4$	$15 \cdot 10^4$	$95 \cdot 10^4$	6417	6635	1.44
W+C F+C	2^{40}	44	4	16	8	16	32	240	4704	$572 \cdot 10^4$	$638 \cdot 10^4$	$13 \cdot 10^4$	$273 \cdot 10^4$	1133	1324	0.28
		40	**5**	**14**	**11**	**256**	**16**	**2040**	**4372**	$\mathbf{577 \cdot 10^4}$	$\mathbf{598 \cdot 10^4}$	**36513**	$\mathbf{70 \cdot 10^4}$	**10402**	**10606**	**2.43**
W+C P+FP	2^{40}	44	4	16	8	16	32	240	4480	$603 \cdot 10^4$	$687 \cdot 10^4$	$24 \cdot 10^4$	$507 \cdot 10^4$	1118	1292	0.29
		40	**5**	**14**	**11**	**256**	**16**	**2040**	**4036**	$\mathbf{601 \cdot 10^4}$	$\mathbf{642 \cdot 10^4}$	$\mathbf{23 \cdot 10^4}$	$\mathbf{462 \cdot 10^4}$	**10380**	**10559**	**2.62**
W+C	2^{30}	36	3	14	9	16	32	240	4316	$676 \cdot 10^4$	$702 \cdot 10^4$	198	1745	899	1088	0.25
		33	3	15	9	16	32	240	4412	$404 \cdot 10^4$	$439 \cdot 10^4$	198	1745	905	1100	0.25
		33	3	15	9	16	32	304	4412	$405 \cdot 10^4$	$440 \cdot 10^4$	4008	35705	713	908	0.21
		33	3	15	9	256	16	2040	3644	$2607 \cdot 10^4$	$2639 \cdot 10^4$	2247	19997	6305	6488	1.78
		32	4	14	10	256	16	2040	3984	$469 \cdot 10^4$	$486 \cdot 10^4$	2996	21854	8352	8544	2.14
		32	4	14	10	256	16	2840	3984	$481 \cdot 10^4$	$498 \cdot 10^4$	$12 \cdot 10^4$	$88 \cdot 10^4$	5152	5344	1.34
W+C F+C	2^{30}	36	3	14	9	16	32	240	4076	$674 \cdot 10^4$	$702 \cdot 10^4$	32966	$68 \cdot 10^4$	884	1059	0.26
		33	3	15	9	16	32	240	4156	$401 \cdot 10^4$	$439 \cdot 10^4$	65734	$136 \cdot 10^4$	889	1069	0.26
		33	3	15	9	256	16	2040	3388	$2603 \cdot 10^4$	$2639 \cdot 10^4$	67783	$138 \cdot 10^4$	6289	6457	1.91
		32	**4**	**14**	**10**	**256**	**16**	**2040**	**3744**	$\mathbf{467 \cdot 10^4}$	$\mathbf{486 \cdot 10^4}$	**35764**	$\mathbf{70 \cdot 10^4}$	**8337**	**8514**	**2.27**
W+C P+FP	2^{30}	36	3	14	9	16	32	240	3788	$707 \cdot 10^4$	$764 \cdot 10^4$	$31 \cdot 10^4$	$653 \cdot 10^4$	865	1019	0.27
		33	3	15	9	16	32	240	3900	$421 \cdot 10^4$	$472 \cdot 10^4$	$16 \cdot 10^4$	$339 \cdot 10^4$	872	1033	0.26
		32	**4**	**14**	**10**	**256**	**16**	**2040**	**3440**	$\mathbf{489 \cdot 10^4}$	$\mathbf{525 \cdot 10^4}$	$\mathbf{20 \cdot 10^4}$	$\mathbf{411 \cdot 10^4}$	**8317**	**8472**	**2.46**

Table 1: Parameters for 2^{40} and 2^{30} signatures. W+C denotes WOTS+C, F+C denotes FORS+C, and P+FP denotes PORS+FP. For PORS+FP the number of leaves is $k \cdot 2^a$. Exp. Search is the expected number of hashes for grinding, included in SigTime. WC (worst-case) Search is the number of hashes that suffices except with probability 2^{-30}. Compr. refers to hash function compression function calls. Compr./Byte is verification compressions divided by signature size.

	q_s	h	d	a	k	w	l	$S_{w,n}$	SigSize (byte)	SigTime (hashes)	SigTime (compr.)	Exp. Search (hashes)	WC Search (hashes) $p = 2^{-30}$	SigVerify (hashes)	SigVerify (compr.)	Compr./ Byte
W+C	2^{20}	24	2	16	8	16	32	240	3624	$578 \cdot 10^4$	$638 \cdot 10^4$	132	1566	646	817	0.23
		24	2	16	8	256	16	2040	3112	$3515 \cdot 10^4$	$3571 \cdot 10^4$	1498	17952	4246	4409	1.42
W+C F+C	2^{20}	24	2	16	8	16	32	240	3352	$572 \cdot 10^4$	$638 \cdot 10^4$	$13 \cdot 10^4$	$273 \cdot 10^4$	629	784	0.23
W+C P+FP	2^{20}	24	2	16	8	16	32	240	3128	$603 \cdot 10^4$	$687 \cdot 10^4$	$24 \cdot 10^4$	$507 \cdot 10^4$	614	752	0.24
W+C	2^{20}	20	2	15	10	256	16	2040	3432	$938 \cdot 10^4$	$972 \cdot 10^4$	1498	17952	4266	4448	1.30
W+C F+C	2^{20}	20	2	15	10	256	16	2040	3176	$934 \cdot 10^4$	$972 \cdot 10^4$	67034	$138 \cdot 10^4$	4250	4416	1.39
W+C P+FP	2^{20}	20	2	15	10	256	16	2040	2856	$991 \cdot 10^4$	$1078 \cdot 10^4$	$53 \cdot 10^4$	$1111 \cdot 10^4$	4229	4372	1.53

Table 2: Parameters for 2^{20} signatures. W+C denotes WOTS+C, F+C denotes FORS+C, and P+FP denotes PORS+FP. For PORS+FP the number of leaves is $k \cdot 2^a$. Exp. Search is the expected number of hashes for grinding, included in SigTime. WC (worst-case) Search is the number of hashes that suffices except with probability 2^{-30}. Compr. refers to hash function compression function calls. Compr./Byte is verification compressions divided by signature size.

14 Hierarchical Deterministic Wallets

In this section, we discuss the concept of hierarchical deterministic (HD) wallets in Bitcoin and consider how (or if) such an approach can be adapted to hash-based signature schemes.

Background. Before HD wallets, Bitcoin wallets typically generated and stored random private keys independently. The users wallets could not easily generate large sets of addresses in a deterministic and recoverable way and had to back up every newly generated key, otherwise losing access to funds if a device failed.

HD wallets, introduced in BIP 32 [45] and widely deployed in combination with mnemonics as in BIP 39 [36], address these issues by deriving all keys from a single *master seed*. A seed is typically 128 to 256 bits of entropy, represented to the user as a human-readable mnemonic phrase of 12 to 24 words. From this seed, a master private key and a *chain code* are derived via HMAC-SHA512. Together, these form an *extended private key*:

- Private key: a 256-bit scalar.
- Chain code: a 256-bit value that provides domain separation for child derivation.

Public child key derivation. One of the most useful features of HD wallets is the ability to derive new public keys without exposing the private key. This enables a separation of roles. The hardware wallet stores the private key and performs signing. The software wallet stores the public key and chain code, and derives new addresses for monitoring and receiving funds.

This is achieved using *extended keys*:

$$\mathsf{xprv} = (k, c), \qquad \mathsf{xpub} = (K, c),$$

where k is the private key, $K = kG$ the corresponding public key, and c the chain code.

For a non-hardened child with index $i < 2^{31}$, derivation works as follows:

- Parent private key: k, parent public key: $K = kG$.
- Compute $I_L \| I_R = \mathsf{HMAC}(kG, c, i)$.
- Child private key: $k_i = (k + I_L) \bmod n$.
- Child public key: $K_i = K + I_L G$.
- Child chain code: $c_i = I_R$.

This allows public key derivation from xpub alone, enabling new addresses to be generated without the master private key. A schematic overview is given in Fig. 14.

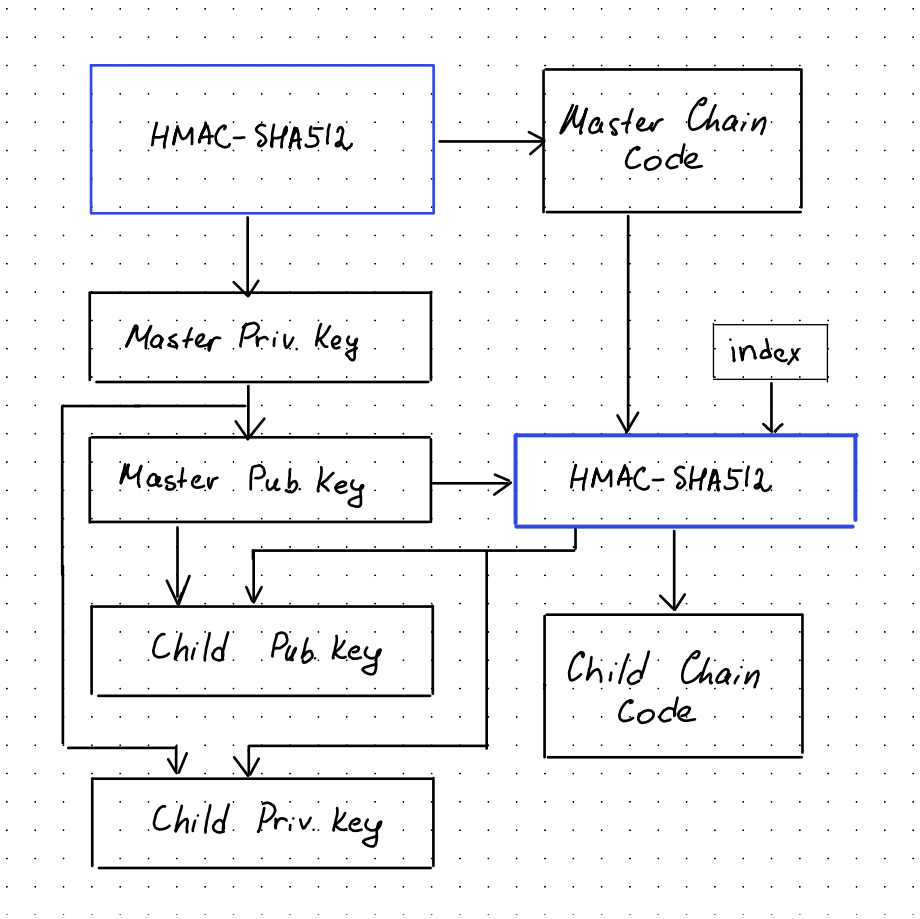


Figure 14: Public child key derivation scheme in HD wallets.

Challenges for hash-based signatures. Unfortunately, this elegant approach does not extend naturally to hash-based signature schemes. The reason lies in the absence of an algebraic group structure: for HD wallets, the linearity of elliptic curves ensures that tweaking the public key by $I_L G$ corresponds to a predictable tweak of the private key by I_L. In hash-based schemes, the public key is typically defined as a Merkle tree root or a hash output, and there seem to be no analogous operation to derive a corresponding child public key without recomputing the entire structure[9].

For now, the most straightforward solution is for the hardware wallet to precompute multiple public keys (each consisting of a public parameter and Merkle root, usually well under 512 bits total) and transmit them to the software wallet. When the available pool is exhausted, the software wallet requests additional public keys.

[9]We considered whether techniques such as chameleon hash functions could be used to "patch" Merkle roots for derivation. However, this would both enlarge public keys significantly and introduce new, non-standard security assumptions.

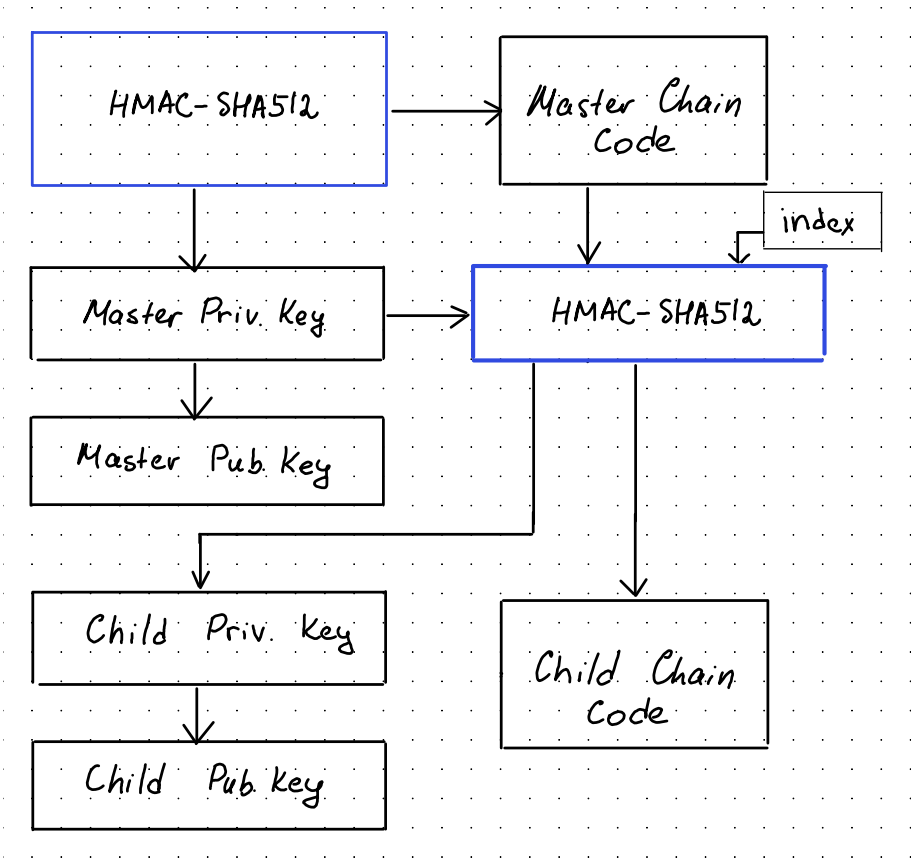


Figure 15: Hardened child key derivation scheme (seed-based).

Hardened derivation. While we did not find a way that non-hardened derivation could work, the hardened approach remains compatible. In hash-based schemes, secret elements of the (hyper-)tree structure are already generated deterministically from a secret seed. Given a parent seed, a chain code (optional), and an index, one can derive a new child secret seed. This mirrors hardened derivation in BIP 32, where only the holder of the master private key (or seed) can derive further private keys. A schematic representation is given in Fig. 15.

15 Multi-Signatures and Threshold Signatures

An important question is whether efficient multi-signature and threshold variants of hash-based signature schemes can be realized. The trivial solution is to rely on Bitcoin's existing scripting capabilities (e.g., `OP_CHECKMULTISIG`). Here we investigate whether more compact and efficient native constructions are possible.

In the *multi-signature* setting, n parties jointly produce a single signature that is

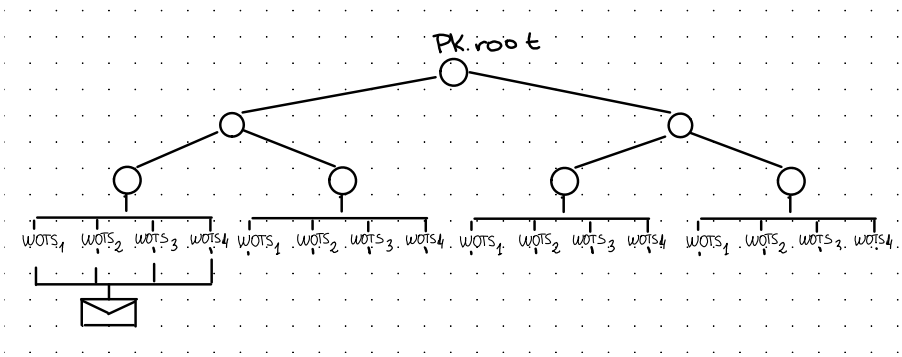


Figure 16: Multi-signature via combined WOTS instances for $n = 4$.

valid only if *all* parties authorize the transaction. In the *threshold* setting, any authorized subset of size $t \leq n$ should be able to produce a valid signature. We will also consider *distributed* signatures – where we can arbitrary specify the authorized subsets.

15.1 Multi-Signatures via Multi-OTS/FTS

We propose a construction that reuses authentication paths in the Merkle tree. The high-level idea is as follows:

- We base our construction on the XMSS scheme.
- Each user generates their own WOTS instances.
- The digests of all corresponding WOTS signatures are aggregated with hash calls.
- The resulting combined digests are used to construct a Merkle tree.

This means that *only one authentication path is required*, regardless of the number of participants, since all signatures are authenticated collectively. The public key structure remains unchanged: a public parameter and the Merkle tree root. The signers also need to come up with signing randomness, for example by first committing to a random value and then revealing it. The randomness for the message then would be the XOR of the revealed values. A toy example of this approach is illustrated in Fig. 16.

Communication overhead. The main drawback of this construction is the interactive nature of both key generation and signing:

- During key generation, all parties must exchange the public keys of their WOTS instances Merkle tree.

- During signing, each party must communicate its WOTS and FORS signatures, as well as the WOTS public keys of the corresponding Merkle trees. This approach requires approximately d interactions.

This approach does not scale to a hyper-tree construction because each signer must ensure that the intermediate tree roots remain consistent across all signatures. However, these roots can be altered by the actions of other participants.

Size comparison. If we use the trivial approach (concatenating n independent signatures), the resulting signature size is:

$$n \cdot \Big((l + h) + (\text{counter}) + \text{randomness} \Big).$$

In contrast, our multi-OTS/FTS approach yields:

$$n \cdot l + \Big(h + (\text{counter}) + \text{randomness} \Big).$$

This achieves a saving of roughly $(n-1)h \cdot 16 + (n-1) \cdot 4 + (n-1) \cdot 32$ bytes.

For example, with $n = 3$, $h = 20$, this can reduce the signature size from 1836 bytes to 1196 bytes.

15.2 Distributed and Threshold Signatures

Kelsey, Lang, and Lucks [25] present techniques for transforming stateful hash-based signature schemes, such as LMS and XMSS, into distributed and threshold signature schemes. Their work addresses the challenge of enabling multiparty control over hash-based signatures, a setting traditionally difficult due to the lack of algebraic structure in hash-based signatures. One of the core disadvantages of the approach is a necessity of a trusted dealer, that generates all the secret values for the parties. This requirement drastically limits the possible applications.

Core Idea. A trusted dealer generates a key pair for a stateful hash-based signature scheme and a *common reference value (CRV)*, a large but public string (hundreds of MBs to a few GBs in size). This CRV contains a correction value for each node in the (hyper-) tree structure.

The dealer distributes secret key shares to multiple trustees, while the aggregator holds the CRV. The secret shares are keys to a **PRF** function. Using these keys and **PRF**, one can sample pseudorandom values for any node in the hash-tree. If we have n parties they sample n sets of random nodes, that correspond to the positions in the hash tree. CRV then serves as a correction value, such that if all the random nodes that correspond to a given position are XORed and then the corresponding CRV value is XORed as well, then the resulting output will match the node in the honest key pair tree structure.

Signing proceeds in two communication rounds: first the users must agree on the randomness that will be used to compute the randomized digest of the message and then the users reveal the needed nodes of their random trees. The aggregator

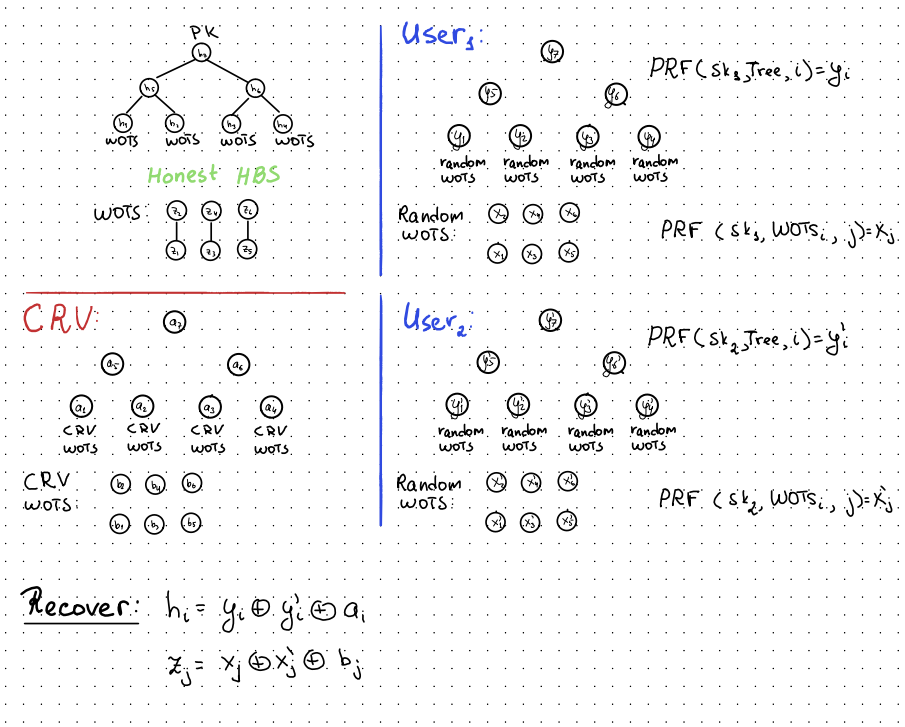


Figure 17: XMSS multi-signature through secret sharing and CRV for $n = 2$.

combines these into a valid signature. Verification remains unchanged. the output is indistinguishable from a standard XMSS signature. An example of a small XMSS tree distributed to two users is given in Fig. 17.

To scale it to arbitrary authorized sets, the users need to get a new PRF key for any each authorized set they are participating in and a CRV must include a separate set of correction values for each authorized set. For example, if we use a rule: any two out of 3, then the CRV will contain three sets of correction values and each user will get two **PRF** keys to compute their random shares.

Security. The construction prevents accidental one-time key reuse by ensuring that multiple trustees must fail simultaneously for a breach to occur. The scheme's security reduces to that of the underlying hash-based signature scheme and a pseudorandom function. A malicious aggregator or corrupted CRV can only cause denial-of-service, not forgery. Furthermore, the design allows revocation of untrusted trustees without regenerating the entire key.

Efficiency and usability. Let's start with the pros of the scheme: the extra computations are performed completely off-chain and invisible to the verifier. From the verifier perspective this is just a regular XMSS signature. Trustees perform similar

amount of work as in the regular signing algorithm. Communication is just a two round protocol, where the first round is meant to commit to the randomness.

Now we come to the cons: the CRV is big and grows with the number of authorized sets. The authors claim that for threshold $(5, 10)$ the scheme is not realistic. We must say though, while the CRV requires significant storage, it still maybe feasible for smaller number authorized parties and this CRV can be stored on the cloud, since it does not affect the security. The CRV, however, must be available, otherwise the parties will loose the ability to sign. The CRV contains the value of the whole tree, in the paper the authors use an XMSS of size 2^{15}, as the number of allowed signatures grow bigger the CRV will grow as well. Kelsey, Lang, and Lucks claim that it is feasible to use this scheme for stateless SPHINCS$^+$ like scheme with a limit of 2^{20} signatures, but this claim needs further studies. For $w = 8$, number of allowed signatures = 2^{15}, and number of authorized sets = 3, the CRV size is already 2.3 GiB. For $w = 4$, number of allowed signatures = 2^{25}, and number of authorized sets = 3, the CRV will grow to approximately 1.1 TiB.

The scheme uses a trusted setup, where the dealer knows all the secret values. We also don't see the ways to make the setup distributed, so that if there was at least one honest party, the setup would be done in a secure way. The trusted setup can be acceptable though in case a single user wants to gain extra protection by distributing its key.

15.3 General MPC Approach

Another possible way to realize multi-signatures or threshold signatures in a hash-based setting is to let each party hold a distinct secret key, and then jointly evaluate the signing algorithm by computing the required hash functions within a general-purpose Multi-Party Computation (MPC) protocol. In principle, this allows distributed signing without modifying the underlying scheme. However, the approach does not look very promising with current techniques.

In [10, Section 5.1], the authors estimate the cost of thresholdizing SPHINCS$^+$ using MPC based on garbled circuits. They report that generating a single signature would require around 85 minutes of computation, which is clearly infeasible for real-world use. Similarly, in [7], the authors apply general-purpose MPC to the evaluation of hash functions and conclude that using standard primitives such as SHA or SHAKE is highly inefficient in this setting. They suggest replacing these with more MPC-friendly alternatives, but this would deviate from standardized hash-based schemes and potentially weaken trust in the construction.

Overall, while theoretically possible, the general MPC approach introduces prohibitive computational overhead and does not currently offer a practical solution.

16 Summary and Discussion

This document provides a comprehensive overview of hash-based signature schemes as a candidate for Bitcoin's post-quantum future. We highlight that these schemes represent a conservative choice, as they rely on standard hash function assumptions

similar to those already underpinning Bitcoin, and benefit from a conceptually simple design. We argue that NIST security level 1 offers a sufficient security margin while maximizing efficiency and minimizing signature size. Furthermore, we show that by deviating from the standardized SPHINCS+ (SLH-DSA) through optimizations like +C, PORS+FP, and reduced signature limits, significant size improvements are achieved. In terms of performance, we observed that the verification cost per byte of our optimized parameter sets is competitive with current Schnorr signatures, alleviating concerns about block validation cost. We provide scripts to compute the security levels, sizes, and costs of the schemes, allowing readers to reproduce the values in our tables or explore custom parameter sets [6]. However, we also note inherent limitations, particularly regarding non-hardened key derivation and efficient multi- and threshold signatures.

Comparison with lattice-based alternatives. While hash-based signatures are often critiqued for their size, the combined cost of public key and signature is the relevant practical metric. For context, the smallest ML-DSA (NIST standardized lattice-based signature scheme) parameter set requires approximately 3732 bytes for (pk + sig) [37, 31, 15]. Although lattice-based schemes like Falcon [38] may offer smaller sizes, these schemes rely on additional hardness assumptions that warrant further study.

Open questions. To conclude, we summarize several open questions arising from this work:

Performance benchmarks. What concrete performance benchmarks are required across various hardware types, including low-power devices with and without SHA-256 hardware acceleration, to effectively guide parameter selection.

Parameter flexibility. Would the ecosystem benefit from standardizing multiple stateless schemes (e.g., with varying limits on the number of signatures) to offer distinct trade-offs? For instance, considering parameters for both 2^{20} and 2^{40} signatures could allow for both compactness in common use cases and capacity for high-demand applications.[10]

Stateful schemes. Is there value in adopting a stateful scheme alongside a stateless one to leverage their extremely compact signatures (see Section B)? Stateful schemes offer size efficiency but require strict state management, posing challenges for backup, recovery, and security.

Acknowledgements

We would like to express our gratitude to **Tim Ruffing** for his thoughtful discussions, insightful suggestions, and careful review of this document.

[10] NIST currently plans to standardize SPHINCS$^+$ with limited signature counts (e.g., 2^{24} for certificate signing) [11]. See also the discussion on the PQC forum.

We also acknowledge the assistance of **ChatGPT**, **Claude**, and **Gemini**, which were used primarily for editorial and language refinement purposes during the preparation of this document.

References

[1] Abri, M., Katz, J.: Shorter hash-based signatures using forced pruning. Cryptology ePrint Archive, Paper 2025/2069 (2025), https://eprint.iacr.org/2025/2069

[2] Aumasson, J.P., Endignoux, G.: Clarifying the subset-resilience problem. Cryptology ePrint Archive, Paper 2017/909 (2017), https://eprint.iacr.org/2017/909

[3] Aumasson, J.P., Endignoux, G.: Improving stateless hash-based signatures. In: Smart, N.P. (ed.) Topics in Cryptology – CT-RSA 2018. Lecture Notes in Computer Science, vol. 10808, pp. 219–242. Springer, Cham, Switzerland, San Francisco, CA, USA (Apr 16–20, 2018). doi:10.1007/978-3-319-76953-0_12

[4] Bennett, C.H., Bernstein, E., Brassard, G., Vazirani, U.: Strengths and weaknesses of quantum computing. SIAM Journal on Computing **26**(5), 1510–1523 (1997). doi:10.1137/S0097539796300933, https://doi.org/10.1137/S0097539796300933

[5] Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS$^+$ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019: 26th Conference on Computer and Communications Security. pp. 2129–2146. ACM Press, London, UK (Nov 11–15, 2019). doi:10.1145/3319535.3363229

[6] Blockstream Research: Supporting scripts for SPHINCS+ parameter exploration (2025), https://github.com/BlockstreamResearch/SPHINCS-Parameters

[7] Bonte, C., Smart, N.P., Tanguy, T.: Thresholdizing HashEdDSA: MPC to the rescue. Cryptology ePrint Archive, Report 2020/214 (2020), https://eprint.iacr.org/2020/214

[8] Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. Cryptology ePrint Archive, Report 2011/484 (2011), https://eprint.iacr.org/2011/484

[9] Buchmann, J.A., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: Buchmann, J., Ding, J. (eds.) Post-quantum cryptography, second international workshop, PQCRYPTO 2008. pp. 63–78. Springer Berlin Heidelberg, Germany, Cincinnati, Ohio, United States (Oct 17–19, 2008). doi:10.1007/978-3-540-88403-3_5

[10] Cozzo, D., Smart, N.P.: Sharing the LUOV: Threshold post-quantum signatures. In: Albrecht, M. (ed.) 17th IMA International Conference on Cryptography and Coding. Lecture Notes in Computer Science, vol. 11929, pp. 128–153. Springer, Cham, Switzerland, Oxford, UK (Dec 16–18, 2019). doi:10.1007/978-3-030-35199-1_7

[11] Dang, Q.: Smaller SLH-DSA: SPHINCS+ smaller parameter sets. Tech. rep., National Institute of Standards and Technology (NIST) (September 2025), https://csrc.nist.gov/csrc/media/presentations/2025/sphincs-smaller-parameter-sets/sphincs-dang_2.2.pdf, cNCS Presentation, 25 September 2025

[12] Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) 10th IMA International Conference on Cryptography and Coding. Lecture Notes in Computer Science, vol. 3796, pp. 96–115. Springer Berlin Heidelberg, Germany, Cirencester, UK (Dec 19–21, 2005). doi:10.1007/11586821_8

[13] Drake, J., Khovratovich, D., Kudinov, M., Wagner, B.: Technical note: LeanSig for post-quantum ethereum. Cryptology ePrint Archive, Paper 2025/1332 (2025), https://eprint.iacr.org/2025/1332

[14] Drake, J., Khovratovich, D., Kudinov, M.A., Wagner, B.: Hash-based multi-signatures for post-quantum ethereum. IACR Communications in Cryptology (CiC) **2**(1), 13 (2025). doi:10.62056/aey7qjp10

[15] 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 **2018**(1), 238–268 (2018). doi:10.13154/tches.v2018.i1.238-268, https://tches.iacr.org/index.php/TCHES/article/view/839

[16] Grover, L.K.: A fast quantum mechanical algorithm for database search. In: 28th Annual ACM Symposium on Theory of Computing. pp. 212–219. ACM Press, Philadephia, PA, USA (May 22–24, 1996). doi:10.1145/237814.237866

[17] Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. p. 212–219. STOC '96, Association for Computing Machinery, New York, NY, USA (1996). doi:10.1145/237814.237866, https://doi.org/10.1145/237814.237866

[18] Hanson, T., Wang, Q., Ghosh, S., Virdia, F., Reinders, A., Sastry, M.R.: Optimization for SPHINCS+ using intel secure hash algorithm extensions. Cryptology ePrint Archive, Report 2022/1726 (2022), https://eprint.iacr.org/2022/1726

[19] Hirch, S.E., Custers, F., van Vredendaal, C.: Tearing solutions for tree traversal in stateful hash-based cryptography. In: Nitaj, A., Petkova-Nikova, S., Rijmen, V. (eds.) AFRICACRYPT 25: 16th International Conference on Cryptology in Africa. Lecture Notes in Computer Science, vol. 15651, pp. 470–492. Springer, Cham, Switzerland, Rabat, Morocco (Jul 21–23, 2025). doi:10.1007/978-3-031-97260-7_21

[20] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle signature scheme. RFC 8391, Request for Comments (May 2018). doi:10.17487/RFC8391, https://www.rfc-editor.org/rfc/rfc8391, informational

[21] Hülsing, A., Kudinov, M.A.: Recovering the tight security proof of SPHINCS$^+$. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology – ASIACRYPT 2022, Part IV. Lecture Notes in Computer Science, vol. 13794, pp. 3–33. Springer, Cham, Switzerland, Taipei, Taiwan (Dec 5–9, 2022). doi:10.1007/978-3-031-22972-5_1

[22] Hülsing, A., Kudinov, M.A., Ronen, E., Yogev, E.: SPHINCS+C: Compressing SPHINCS+ with (almost) no cost. In: 2023 IEEE Symposium on Security and Privacy. pp. 1435–1453. IEEE Computer Society Press, San Francisco, CA, USA (May 21–25, 2023). doi:10.1109/SP46215.2023.10179381

[23] Hülsing, A., Rausch, L., Buchmann, J.: Optimal parameters for XMSS^{MT}. Cryptology ePrint Archive, Report 2017/966 (2017), https://eprint.iacr.org/2017/966

[24] josibake, Somsen, R.: Silent payments. Bitcoin Improvement Proposal (BIP) 352 (2023), https://github.com/bitcoin/bips/blob/master/bip-0352.mediawiki

[25] Kelsey, J., Lang, N., Lucks, S.: Turning hash-based signatures into distributed signatures and threshold signatures. IACR Communications in Cryptology **2**(2) (2025). doi:10.62056/a6ksudy6b

[26] Khovratovich, D., Kudinov, M.A., Wagner, B.: At the top of the hypercube - better size-time tradeoffs for hash-based signatures. In: Kalai, Y.T., Kamara, S.F. (eds.) Advances in Cryptology – CRYPTO 2025, Part VI. Lecture Notes in Computer Science, vol. 16005, pp. 93–123. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 17–21, 2025). doi:10.1007/978-3-032-01887-8_4

[27] Kim, P., Han, D., Jeong, K.C.: Time–space complexity of quantum search algorithms in symmetric cryptanalysis: applying to AES and SHA-2. Quantum Information Processing **17**(12), 339 (2018). doi:10.1007/s11128-018-2107-3, https://link.springer.com/article/10.1007/s11128-018-2107-3

[28] Kölbl, S.: A note on SPHINCS+ parameter sets. Cryptology ePrint Archive, Report 2022/1725 (2022), https://eprint.iacr.org/2022/1725

[29] Kölbl, S., Philipoom, J.: A note on SPHINCS+ parameter sets. Presentation (PDF), CSRC / NIST, Fifth PQC Standardization Conference, April 11, 2024 (2024), https://csrc.nist.gov/csrc/media/Presentations/2024/a-note-on-sphincs-parameter-sets/images-media/kolbl-note-on-sphincs-plus-pqc2024.pdf, accessed: December 5, 2025

[30] Lamport, L.: Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory (Oct 1979)

[31] Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022

[32] Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO'89. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer, New York, USA, Santa Barbara, CA, USA (Aug 20–24, 1990). doi:10.1007/0-387-34805-0_21

[33] National Institute of Standards and Technology: Recommendation for stateful hash-based signature schemes (sp 800-208), https://csrc.nist.gov/publications/detail/sp/800-208/final, approved October 29, 2020

[34] National Institute of Standards and Technology: Stateless hash-based digital signature standard (FIPS 205). doi:10.6028/NIST.FIPS.205, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf

[35] NIST: Call for additional digital signature schemes for the post-quantum cryptography standardization process (2022), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf

[36] Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: Mnemonic code for generating deterministic keys. Bitcoin Improvement Proposal (BIP) 39 (2013), https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[37] PQShield Research Team: Post-quantum signatures zoo. https://pqshield.github.io/nist-sigs-zoo/ (2024), last updated October 28, 2024, CC-BY-SA 4.0

[38] Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022

[39] Reyzin, L., Reyzin, N.: Better than BiBa: Short one-time signatures with fast signing and verifying. In: Batten, L.M., Seberry, J. (eds.) ACISP 02: 7th Australasian Conference on Information Security and Privacy. Lecture Notes in Computer Science, vol. 2384, pp. 144–153. Springer Berlin Heidelberg, Germany, Melbourne, Victoria, Australia (Jul 3–5, 2002). doi:10.1007/3-540-45450-0_11

[40] Roetteler, M., Naehrig, M., Svore, K.M., Lauter, K.E.: Quantum resource estimates for computing elliptic curve discrete logarithms. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, Part II. Lecture Notes in Computer Science, vol. 10625, pp. 241–270. Springer, Cham, Switzerland, Hong Kong, China (Dec 3–7, 2017). doi:10.1007/978-3-319-70697-9_9

[41] Roose, S.: Simple proof-of-reserves transactions. Bitcoin Improvement Proposal (BIP) 127 (2019), https://github.com/bitcoin/bips/blob/master/bip-0127.mediawiki

[42] Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. IEEE Computer Society Press, Santa Fe, NM, USA (Nov 20–22, 1994). doi:10.1109/SFCS.1994.365700

[43] Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (Oct 1997). doi:10.1137/S0097539795293172, https://epubs.siam.org/doi/10.1137/S0097539795293172

[44] Wiggers, T., Bashiri, K., Kölbl, S., Goodman, J., Kousidis, S.: Hash-based signatures: State and backup management. Internet-Draft draft-wiggers-hbs-state-00, IETF (Feb 2024), https://www.ietf.org/archive/id/draft-wiggers-hbs-state-00.html, expires 22 August 2024

[45] Wuille, P.: Hierarchical deterministic wallets. Bitcoin Improvement Proposal (BIP) 32 (2012), https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[46] Wuille, P., Nick, J., Ruffing, T.: Schnorr signatures for secp256k1. Bitcoin Improvement Proposal (BIP) 340 (2020), https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[47] Zalka, C.: Grover's quantum searching algorithm is optimal. Phys. Rev. A **60**, 2746–2751 (Oct 1999). doi:10.1103/PhysRevA.60.2746, https://link.aps.org/doi/10.1103/PhysRevA.60.2746

A Top-Layer WOTS-TW

Here we give the TL-WOTS-TW scheme [26]. As we discussed before, the core idea is to increase the Winternitz parameter, so we can have an encoding that is closer to the top of the hash chains. This way we can reduce the verification time. To give a better intuition we will start with an example.

Example. Assume we want to sign messages of length 5 bits using a WOTS-like construction with $l = 5$ chains. Let us compare two scenarios:

1. **Standard choice:** Suppose we use $w = 2$. Then the message space is mapped as $\mathcal{M} = [0, 1]^5$. On average, the verifier must hash through $(w - 1)/2 = 1/2$ links in each chain to reach the public key element. The expected verification cost is therefore about 2.5 hash evaluations, and in the worst case the verifier will have to do 5 hash calls.

2. **Larger w:** Now suppose we instead use $w = 3$. The message space $\mathcal{M}$ is still of the same size, but is now embedded into the larger space $[0, 2]^5$. By carefully designing the mapping, each message can be placed "closer to the top" of the chains. To do so, we can count the number of hashings required to incorporate the entire message space. If we allow zero hashings, there is only one mapping: to the top of the chains. If we allow exactly one hash function calculation, then there are five possible encodings: [1, 2, 2, 2, 2], [2, 1, 2, 2, 2], [2, 2, 1, 2, 2], [2, 2, 2, 1, 2], and [2, 2, 2, 2, 1]. If we allow exactly two hash function calculations, we can do both of them in one chain or separate them into two different chains. Hence, the number of such encodings is: $\binom{5}{1}+\binom{5}{2} = 15$. If we allow exactly three hashings, we can get 35 different encodings. Since our message space has only $2^5 = 32$ elements, we only need 32 encodings. We can choose a function that maps the message into one of these encodings that uses up to three hashings. As a result of such mapping to the top of the chains we would need three hashings in the worst case and 2.125 on average (see Fig. 18).

The tradeoff is that the signer must compute values higher in the chains before releasing them. In the example above, when $w = 3$, the signer must compute roughly 7.875 links per chain on average, compared to only 2.5 when $w = 2$. Thus, verification becomes cheaper, but signing becomes more expensive.

This example illustrates the general principle: by embedding $\mathcal{M}$ into a larger hypercube $[0, w-1]^l$ and selecting an appropriate mapping, one can shift computational work from the verifier to the signer, while keeping the signature size fixed. To achieve the requirement that any two messages have a node lower in one of the chains one can either attach a checksum, or use the target sum technique from WOTS+C.

Formal construction. Here we present a formal construction form At the Top of the Hypercube - Better Size-Time Tradeoffs for Hash-Based Signatures:

Definition A.1 (Hypercube Layers)**.** *Let* $\mathtt{w}$ *and* v *be integers and consider the hypercube* $\{0, \dots, \mathtt{w} - 1\}^v$*. The layer* $\mathcal{L}_d \subseteq \{0, \dots, \mathtt{w} - 1\}^v$ *is the set of vertices at distance* d *from the*

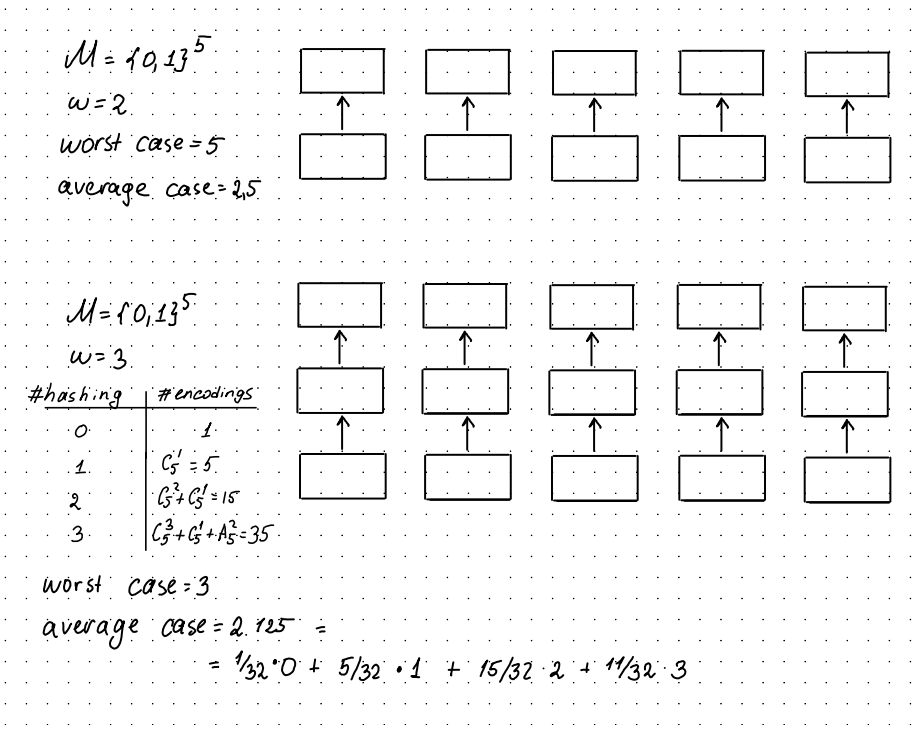


Figure 18: Embedding example for TL-WOTS-TW.

sink $(\mathsf{w}-1,\dots,\mathsf{w}-1)$. *Precisely, that is*

$$\mathcal{L}_d := \left\{ (x_1, x_2, \dots, x_v) \in \{0,\dots,\mathsf{w}-1\}^v \;\middle|\; v(\mathsf{w}-1) - \sum_{i=1}^{v} x_i = d \right\}.$$

For $x \in \mathcal{L}_d$, *we write* $\mathrm{layer}(x) = d$. *Note that* d *can range from* 0 *to* $v(\mathsf{w}-1)$. *Further, we use the notation* $\mathcal{L}_{[d_1:d_2]} := \bigcup_{d=d_1}^{d_2} \mathcal{L}_d$ *to denote the union of multiple adjacent layers.*

Remark A.2. *For any two distinct vertices* $x, x' \in \mathcal{L}_d$, *we neither have* $x \le x'$ *nor* $x \ge x'$, *where* $x \le x'$ *if each coordinate of* x *is smaller or equal to the corresponding coordinate in* x'.

Definition A.3 (Sizes of Hypercube Layers)**.** *Let* w *and* v *be integers and consider the hypercube* $\{0,\dots,\mathsf{w}-1\}^v$. *We denote the size of the* d*th layer of the hypercube as* $\ell_d = |\mathcal{L}_d|$. *Note that the size* ℓ_d *of the* d*th layer is given by the coefficient of* $X^{v(\mathsf{w}-1)-d}$ *in the polynomial* $(1+X+X^2+\cdots+X^{\mathsf{w}-1})^v$. *Further, we use the notation* $\ell_{[d_1:d_2]} = \sum_{d=d_1}^{d_2} \ell_d$ *to denote the size of multiple adjacent layers.*

Construction A.1 (Top Level Message Hash)**.** *Let* $v, \mathsf{w}, D, T \in \mathbb{N}$ *be integers such that* $T \ge v(\mathsf{w}-1)-D$. *Let* $\mathsf{Th}^{msg} : \mathcal{P}\times\mathcal{T}\times(\{0,1\}^{l_{msg}}\times\mathcal{R}) \to \{0,\dots,A-1\} \subseteq \mathbb{N}$ *be a tweakable hash function. We define a new tweakable hash function* $\mathsf{TL}^D[\mathsf{Th}^{msg}] : \mathcal{P}\times\mathcal{T}\times(\{0,1\}^{l_{msg}}\times\mathcal{R}) \to \mathcal{L}_{[0:D]}$. *It is defined as follows for inputs* $(P, T, (M, R))$*:*

1. *Sample* $\rho \leftarrow_{\$} \mathcal{R}$.
2. *Set* $x := \mathsf{Th}^{msg}(P, T, (m, \rho))$.
3. *Set* $y := x \bmod \ell_{[0:D]}$.
4. *Map* y *to a vertex* $a \in \mathcal{L}_{[0:D]}$ *via the following steps:*
   (a) *Find* d *such that* $\ell_{[0:d-1]} \leq y < \ell_{[0:d]}$.
   (b) *Set* $a := \mathsf{MapToVertex}_{\mathsf{w},v,d}(y - \ell_{[0:d-1]})$
5. *Output* a.

where MapToVertex is defined as:

- $\mathsf{MapToVertex}_{\mathsf{w},v,d}(x) \to a$ *for* $0 \leq x < \ell_d^{\langle v \rangle}$ *and* $a \in \{0, \ldots, \mathsf{w}-1\}^v$
  1. *Set* $x_1 := x$ *and* $d_1 := d$.
  2. *For* i *from* 1 *to* $v-1$*:*
     (a) *Find* $j_i \in \{\max(0, d_i - (\mathsf{w}-1)(v-i)), \ldots, \min(\mathsf{w}-1, d_i)\}$ *such that*
     $$\sum_{\max(0,d_i-(\mathsf{w}-1)(v-i))\leq j<j_i} \ell_{d_i-j}^{\langle v-i \rangle} \leq x_i < \sum_{\max(0,d_i-(\mathsf{w}-1)(v-i))\leq j\leq j_i} \ell_{d_i-j}^{\langle v-i \rangle}.$$
     (b) *Set* $a_i := \mathsf{w}-1-j_i \in [\mathsf{w}]$.
     (c) *Set* $d_{i+1} := d_i - j_i = d_i - (\mathsf{w}-1-a_i)$.
     (d) *Set* $x_{i+1} := x_i - \sum_{\max(0,d_i-(\mathsf{w}-1)(v-i))\leq j<j_i} \ell_{d_i-j}^{\langle v-i \rangle}$.
  3. *Set* $a_v := \mathsf{w}-1-x_v-d_v$.
  4. *Output* $(a_1, a_2, \ldots, a_v)$.

An example of the MapToVertex algorithm is presented in Fig. 19.

To search for the parameters, one can utilize the previous work or use the Python scripts:

https://github.com/b-wagn/hypercube-hashsig-parameters.

Further, the authors have implemented a prototype implementation of their encoding:

https://github.com/b-wagn/hash-sig.

B Stateful Hash-Based Signature Schemes

In this section, we examine the option of deploying stateful hash-based signature schemes (HBS) in the Bitcoin setting. We begin with a broader discussion of their applicability, followed by potential optimizations that could make stateful constructions more practical.

Challenges of stateful schemes. The primary obstacle for stateful HBS lies in their reliance on flawless state management. Several concerns are particularly relevant for Bitcoin:

- **State management risks.** The security of stateful schemes critically depends on ensuring that each state is used at most once. State reuse enables forgeries. Implementations must ensure that state is never lost, corrupted, or rolled back.
- **Backup and recovery.** Users need to be able to back up their keys. Restoring from an outdated backup without the current state could lead to state reuse and enable forgeries.
- **State storage constraints.** In many wallet implementations, state can only be stored in locations accessible to users. Users might inadvertently restore from old backups, effectively rewinding the state.
- **New threat model.** An adversary could trick the victim into replaying old state and enable forgeries.

Despite these challenges, it is technically possible to manage state securely [44]. Moreover, some Layer 2 protocols already require maintaining state correctly; for example, Lightning Network nodes must track channel state to avoid losing funds. For such applications, stateful hash-based signatures could potentially be applicable. Several techniques exist that could improve the performance and safety of stateful schemes.

B.1 One-Time Signatures

One possibility is to extend Bitcoin with an opcode that validates a single WOTS-TW or WOTS+C instance. Such signatures are very small ($\approx$ 256 bytes) and could be combined using Taproot. However, the key drawback is that the user can produce only one valid signature per key. If multiple signatures are ever generated for the same public key, even if they do not appear on-chain, the security of the scheme collapses. This restriction is especially problematic in off-chain protocols. Moreover, while Taproot could combine such one-time keys, the resulting scheme would remain stateful and less efficient than an XMSS-based construction, since XMSS can be instantiated with shorter hash outputs (e.g., 128 bits).

B.2 Tree Traversal

Tree-based stateful schemes such as XMSS and XMSS^{MT} benefit from efficient traversal algorithms that reduce signing costs. The BDS algorithm [9] balances leaf computations across signing steps, reducing worst-case runtime to nearly match average runtime. More recently, BDSFix [19] was introduced to handle recovery from "tearing events" (e.g., power loss) while maintaining correctness of the traversal state. Such techniques make stateful schemes more efficient and robust in practice.

B.3 Unbalanced Trees

Another approach is to use unbalanced Merkle trees, where leaves closer to the root correspond to shorter authentication paths. For example, the first leaf may be only one hash away from the root, the second leaf two hashes away, and so forth. This structure prioritizes efficiency for the earliest signatures under a given public key, which aligns with Bitcoin's expected usage pattern: keys are generally used only once before being replaced. The structure is presented in Fig. 20a. The tradeoff is that later signatures require longer authentication paths, but this is less relevant if key reuse is rare.

To provide a secure fallback in the case of restoring from a backup one can use one branch of a Merkle tree for the unbalanced XMSS, and the second one for the stateless scheme (see Fig. 20b).

C Octopus Algorithm

In this section we describe the Octopus algorithm [3]. The aim of the algorithm is to give a minimal authentication set for a given set of leaves in a Merkle tree. The algorithm takes as an input a set of leaves that need to be authenticated. Octopus maintains three sets:

- *Indices*: the nodes that need to be authenticated in this iteration
- *NewIndices*: nodes that we will have to authenticate in the next iteration, and
- *Auth*: authentication set.

We start by putting all the leaves into the *Indices* set. For each entry in the *Indices* we check if the sibling node is also contained in this set. If there is no sibling in *Indices*, then we add the sibling node to the *Auth* set. Given the leaf and its sibling we can compute the parent node. Hence, we add the parent of the node into the *NewIndices* set. When we checked all the entries in the *Indices* set, we can continue to the next iteration, by letting *Indices* = *NewIndices*, and emptying *NewIndices*.

Consider an encoding where nodes are represented as binary strings. If node v has encoding string s, then its left child has encoding $s\|0$ and its right child has encoding $s\|1$ (where $\|$ denotes concatenation). The root node has empty string ε as its encoding. The sibling of a node with encoding s is obtained by flipping the last bit: $s \oplus 1$. The parent of a node with encoding $s\|b$ (where $b \in \{0, 1\}$) is the node with encoding s. The pseudocode of Octopus is given in Algorithm 2. An example is presented in Fig. 21.

Algorithm 2: Octopus

Input: $[x_1, \ldots, x_k]$, tree height h
Output: Authentication set *Auth*.

```
1  Indices ← sorted([x_1, ..., x_k])
2  Auth ← [ ]
3  for ℓ = h − 1 to 0 do
4      NewIndices ← [ ]
5      j ← 0
6      while j < length(Indices) do
7          x ← Indices[j]
8          NewIndices.append(⌊x/2⌋)
9          sibling ← x ⊕ 1
10         if j + 1 < length(Indices) and Indices[j + 1] = sibling then
11             j ← j + 2
12         else
13             Auth.append((ℓ + 1, sibling))
14             j ← j + 1
15     Indices ← NewIndices
16 return Auth
```

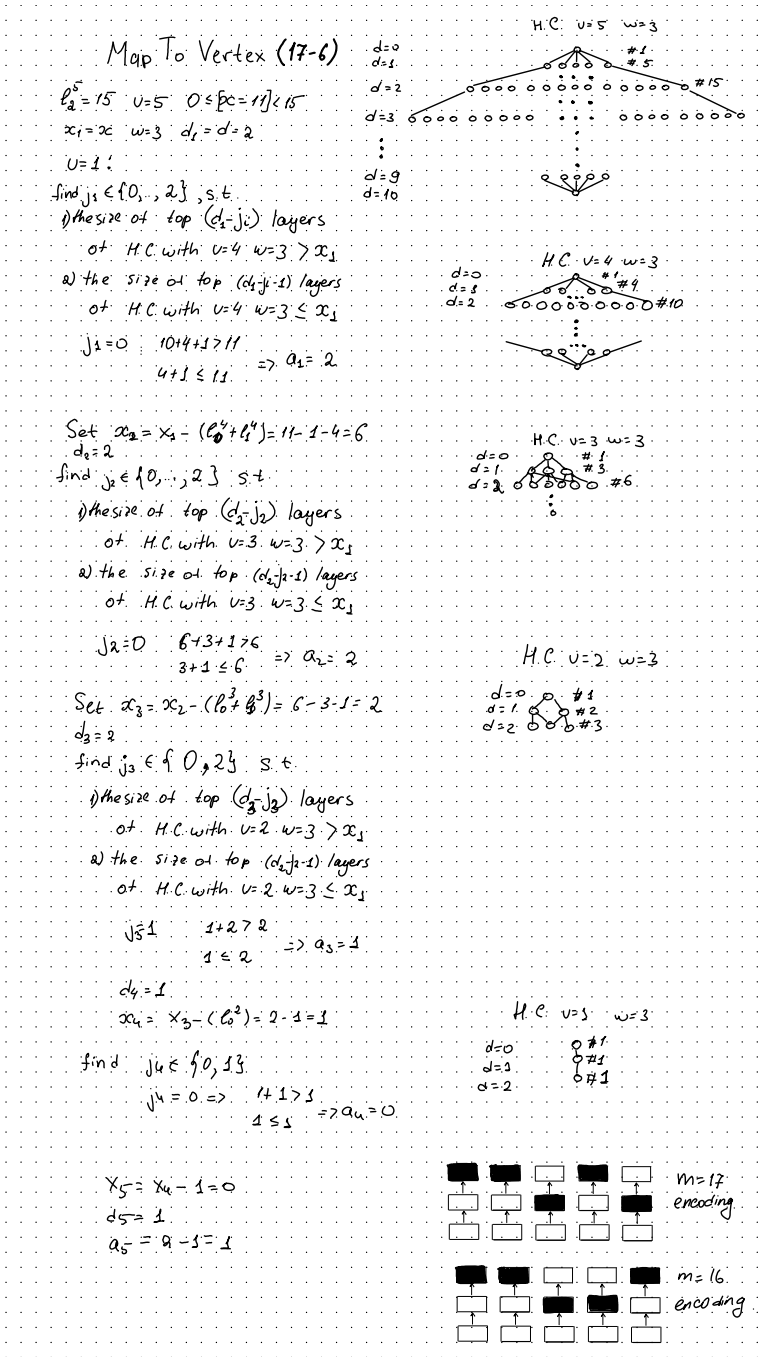
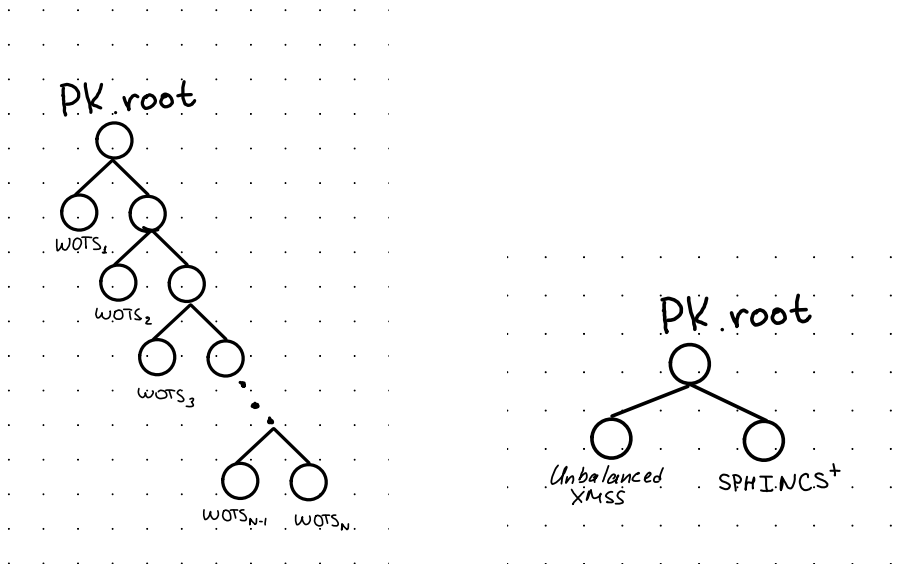


Figure 19: Example of embedding computation for TL-WOTS-TW.



(a) Unbalanced XMSS tree

(b) Stateful + stateless combination

Figure 20: Application of an Unbalanced XMSS.

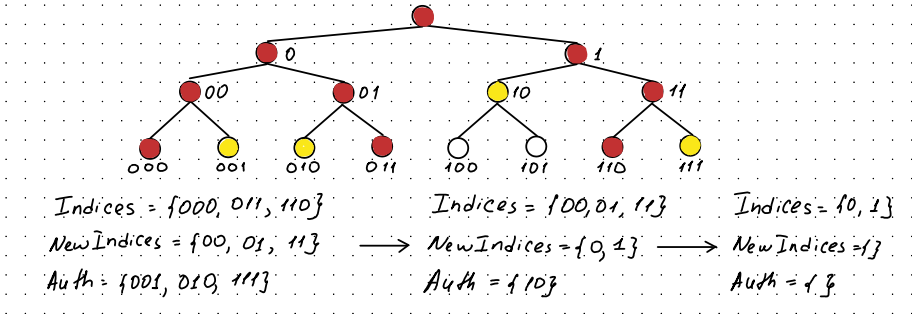


Figure 21: Example of Octopus algorithm for leaves 000, 011, 110.