Improving Stateless Hash-Based Signatures

Jean-Philippe Aumasson[1] and Guillaume Endignoux[2]

[1] Kudelski Security, Switzerland
[2] `firstname.surname@m4x.org`

Abstract. We present several optimizations to SPHINCS, a stateless hash-based signature scheme proposed by Bernstein et al. in 2015: PORS, a more secure variant of the HORS few-time signature scheme used in SPHINCS; secret key caching, to speed-up signing and reduce signature size; batch signing, to amortize signature time and reduce signature size when signing multiple messages at once; mask-less constructions to reduce the key size and simplify the scheme; and Octopus, a technique to eliminate redundancies from authentication paths in Merkle trees. Based on a refined analysis of the subset resilience problem, we show that SPHINCS' parameters can be modified to reduce the signature size while retaining a similar security level and computation time. We then propose Gravity-SPHINCS, our variant of SPHINCS embodying the aforementioned tricks. Gravity-SPHINCS has shorter keys (32 and 64 bytes instead of ≈ 1 KB), shorter signatures (≈ 30 KB instead of 41 KB), and faster signing and verification for a same security level as SPHINCS.

1 Introduction

In 2015 Bernstein et al. presented SPHINCS [5], a stateless hash-based signature scheme combining Goldreich's [12, §6.4.2] binary authentication tree of one-time signatures (OTS) and Reyzin[2]'s [19] hash to obtain a random subset (HORS) construction of a few-time signature (FTS) scheme, with two main optimizations.

1. Inner nodes of the tree are not OTSs but *Merkle trees* whose leaves are OTSs, namely Winternitz OTS (WOTS) [17,13] instances. Each node can then sign up to 2^x children nodes instead of 2, where x is the height of the Merkle tree—SPHINCS thus uses a *hyper-tree*. This change increases signing time because each Merkle tree on the path to a leave needs to be generated for every signature, but reduces the signature size because fewer OTS instances are included in the signature.
2. Leaves of the hyper-tree are not OTSs but FTSs, namely HORS with tree instances (HORST), a version of HORS that uses a binary tree to compress the HORS public key. Leaves can then sign more than one message, which increases the resilience to path collisions, hence reducing the height needed for the hyper-tree.

SPHINCS-256, the main SPHINCS instance proposed in [5], has $\approx 1\,\text{KB}$ private and public keys, 41 KB signatures, and offers 128-bit quantum security provided that no more than 2^{50} signatures are issued. A 41 KB signature is fine in some use cases, but can induce significant extra cost if many signatures are stored, compared to pre-quantum constructions. Likewise, key sizes—especially public keys—can be prohibitive in low-memory platforms. For this reason, it makes sense to try to shave off a few bytes from SPHINCS signatures and keys.

This Paper. After briefly describing HORST and SPHINCS in §2, we present the following optimizations to SPHINCS in §3.

- **PORS**, a more secure variant of HORS (§§3.1).
- **Secret key caching**, to speed-up signing and reduce signature size (§§3.2).
- **Batch signing**, to amortize signature time and reduce signature size (§§3.3).
- **Mask-less hashing**, to reduce the key size (§§3.4).
- **Octopus**, to avoid redundancies in Merkle tree authentication paths of HORST, and thus reduce signature size (§§3.5).

Based on these optimizations and on refined analyses of the subset resilience problem and of mask-less constructions—whose details could not be included for lack of space, but have been made public [2,10]—we show that SPHINCS parameters can be modified to reduce the signature size while retaining a similar security level. §4 then describes Gravity-SPHINCS, a signature scheme based on SPHINCS with shorter keys (32 and 64 bytes instead of $\approx$ 1KB), shorter signatures ($\approx 30\,\text{KB}$ instead of 41 KB), and faster signing and verification.

2 HORST and SPHINCS

We briefly describe how HORST and SPHINCS work, however a comprehensive specification of SPHINCS would take too much space so we refer readers to the original paper [5]. Note that contrary to the SPHINCS paper, we use the more common and practical convention that a tree's level 0 is the root, and not the leaf level.

2.1 HORST

HORST is a few-time signature scheme proposed by the SPHINCS authors as a variant of HORS [19], which works as follows.

A HORS private key is a list of t values $(\mathsf{ek}_i)_{i=0\dots t-1}$, where $t = 2^\tau$ for $\tau \in \mathbb{N}^*$. The public key is the list $(\mathsf{pk}_i = \mathsf{hash}_0(\mathsf{ek}_i))_{i=0\dots t-1}$ where hash_0 is a one way function. For example, 128-bit secure HORS may use 256-bit ek_i's.

Signing a message M with HORS works as follows, given a parameter $k < t$.

- Derive a set of k indices $\{V_i\}_{i=0,\dots,k-1}$ from $\mathsf{hash}_1(M)$, where hash_1 is a hash function, by splitting the hash into k chunks of τ bits converted into integers in $\{0, \dots, t-1\}$.

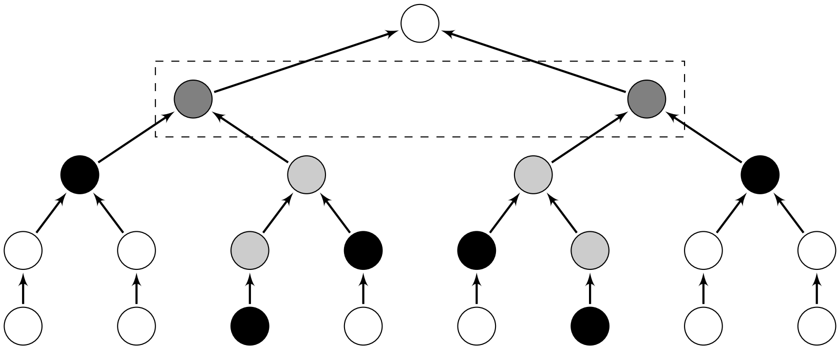


Fig. 1: Binary hash tree of a HORST signature, with $t = 8$ hashes (thus a tree depth of $\tau = 3$), a subset of $k = 2$ hashes, $C = 2$ subtrees (with roots in dark grey in the dashed rectangle), and indices $V_0 = 2, V_1 = 5$. The nodes in black and dark grey are part of the signature, the nodes in pale grey are computed during the verification, and the root node is the public key.

- Return $\mathsf{sig} = (\mathsf{sig}_i)_{i=0,\ldots,k-1}$ where $\mathsf{sig}_i = \mathsf{ek}_{V_i}$.

Verification computes the k indices V_i from M and checks that $\mathsf{hash}_0(\mathsf{sig}_i)$ equals pk_{V_i} for all i.

HORS with trees (HORST) as used in SPHINCS replaces the t-value public key with a single value. This value is the root of the Merkle tree whose leaves are the pk_i's. A HORST signature consists of k ek_i's along with their k authentication paths, i.e. the list of sibling nodes required to "connect" each pk_i to the root. Because the k authentication paths will likely share high-level authentication nodes, an optimization made in SPHINCS is to include in the signature all nodes at some level. This avoids storing authentication nodes above that level. Figure 1 shows a simplistic example of HORST with the latter optimization.

The more HORST signatures are issued, the more private ek_i's are revealed to an attacker, and they will eventually be able to forge signatures by finding a message that hashes to a set of known indices. The cost of such an attack is analyzed in [19] and [5], and in further details in [2], to cover the case of adaptive attacks.

2.2 SPHINCS

SPHINCS is a complex scheme, and the description in [5] may not be sufficient to fully understand it. In our experience the best way to understand SPHINCS is to look at an implementation—such as the simple Python version at `https://github.com/joostrijneveld/SPHINCS-256-py`, or others listed at `https://ianix.com/pqcrypto/pqcrypto-deployment.html`—or, better, to write one. Yet we'll try here to introduce the main ideas of SPHINCS, by describing it as a combination of four types of trees. The four types of trees are the following (see Figure 2).

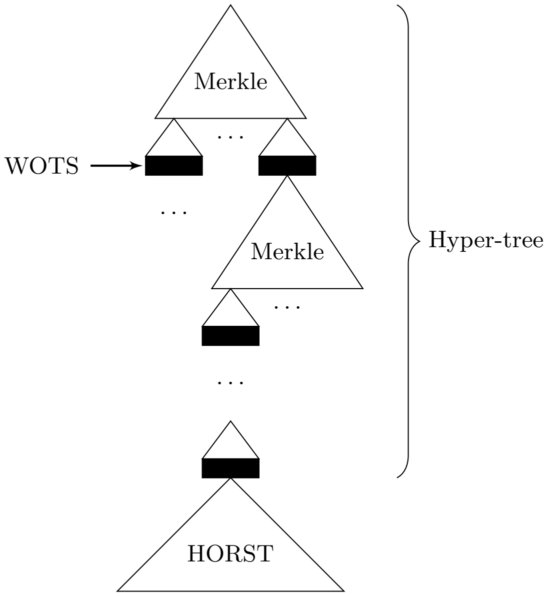


Fig. 2: Sketch of the SPHINCS construction.

1. The main hyper-tree, of height h (60 in SPHINCS-256). The root of this tree is part of the public key. The leaves of this tree are HORST instances (type-4 trees). This hyper-tree is divided into d (12 in SPHINCS-256) layers of type-2 trees.
2. The subtrees, which are Merkle trees of height h/d ($60/12 = 5$ in SPHINCS-256). The leaves of these trees are roots of type-3 trees; said roots are compressed public keys of WOTS instances, that connect to a tree at the next layer.
3. The WOTS public key compression trees, which are L-trees (and not necessarily complete binary trees), of height $\lceil \log_2 \ell \rceil$ when there are ℓ leaves. The leaves of this tree are components of a WOTS public key (67 values of 256 bits each in SPHINCS-256). The associated WOTS instance signs a tree root at the next layer.
4. At the bottom of the hyper-tree, the HORST public key compression trees are Merkle trees of height $\tau = \log_2 t$, where t is the number of public key elements in the HORST instances (2^{16} in SPHINCS-256).

Signing with SPHINCS then works like this:

1. Derive a *leaf index* from the message and the private key. This index identifies one of the 2^h HORST instances (relative to the main hyper-tree), that will be used to sign the message.
2. Generate the HORST instance whose seed is derived from the private key and from the leaf index, and sign the message with this HORST instance. The

HORST signature includes k keys and their respective authentication paths, and is part of the SPHINCS signature. Obtain the HORST tree-compressed public key p.

3. For each layer of the hyper-tree, sign the public key p (obtained from the lower layer) using the correct WOTS instance (derived from the leaf index); add this WOTS signature and associated type-3 authentication path to the SPHINCS signature. Compute the authentication path of this WOTS instance within the type-2 subtree; add this path to the SPHINCS signature and let p be the subtree root.

This is really a bird eye's view of SPHINCS, and we omitted many details. See the paper [5] for a more formal description.

3 Improvements

In this section, we present our optimizations for SPHINCS and similar hash-based signature schemes.

3.1 From HORS to PORS

HORS was only partially studied, as [19] only considered non-adaptive attacks. But [2] recently showed that the textbook version of HORS is susceptible to adaptive attacks, and that its simplicity can be exploited to further reduce its security. Indeed, nothing prevents some of the k indices to collide (yielding only $\kappa < k$ distinct indices), reducing the size of the obtained subset and making forgeries easier.

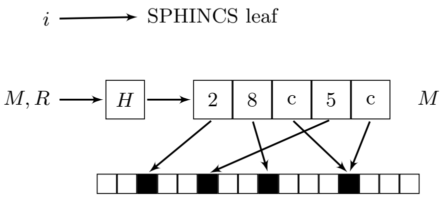
HORS' original hash-based index generation is simple and fast, yet its speed is not critical in SPHINCS, where Winternitz OTS (WOTS) and Merkle trees dominate the computational cost. We therefore propose a slightly more complex construction, PORS, for *PRNG to obtain a random subset*. Instead of using a hash function, we seed a PRNG from the message (and salt) and query it until we obtain k distinct indices (Figure 3). The computational overhead is minimal, for a significant security increase.

In the original SPHINCS, adversaries have full control over the selected leaf in the hyper-tree. Instead, we propose to generate this leaf index with the PRNG, in order to reduce the attack surface. This increased security level allows to reduce the hyper-tree height by 2 layers of WOTS, saving 4616 bytes.

More details and a security analysis are given in [2].

Remark. In SPHINCS, the public salt R is computed by the signer as $\mathsf{hash}(\mathsf{salt} \| M)$ for a secret salt.[3] This means that if the message M is long, the signer needs to compute two long hashes: $R = \mathsf{hash}(\mathsf{salt} \| M)$ and the HORST subset as $\mathsf{hash}(R \| M)$. Instead, with PORS we propose to compute a long hash $H = \mathsf{hash}(M)$ and then two small hashes as $R = \mathsf{hash}(\mathsf{salt} \| H)$ and $\mathsf{seed} = \mathsf{hash}(R \| H)$

[3] Here hash means "some hash function", not necessarily the same in all places.



(a) HORS.

(b) PORS.

Fig. 3: Comparison of the HORS and PORS constructions to obtain a random subset from a message M and a salt R. In HORS (left), the hash function output is split into τ-bit blocks that may collide. If the scheme is used in SPHINCS, the signer (or attacker) provides the leaf index i. In PORS (right), a PRNG is used to produce as many τ-bit numbers as necessary, as well as the leaf index i if used in SPHINCS.

as a seed for the PRNG. This halves the computational overhead for long messages.

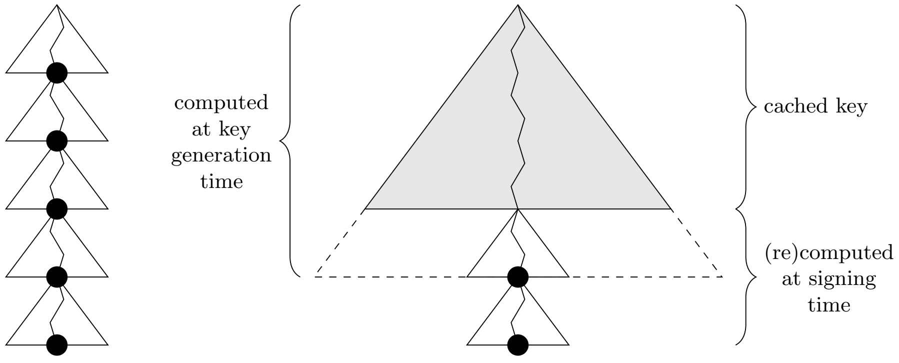
3.2 Secret Key Caching

XMSS [6] is a signature scheme similar to SPHINCS but with shorter signatures at the expense of being stateful. For example, the XMSS-T variant [14] produces signatures of 8.8 KB for a capacity of 2^{60} messages and 128-bit quantum security. The main difference with SPHINCS is that the hyper-tree of SPHINCS is divided in many layers ($d = 12$) because these trees have to be recomputed on-the-fly for each signature.

But XMSS benefits from efficient incremental algorithms to amortize the computational cost over many signatures [15,20,7]. Consequently, SPHINCS authors proposed to divide a hyper-tree of height 60 into 12 layers of Merkle trees, each of height 5, meaning that there are 12 WOTS signatures to connect these layers. Most of the size of a SPHINCS signature is used by WOTS signatures, each containing $\ell = 67$ hash values, or 2144 bytes per WOTS signature. In contrast, an authentication path in a Merkle tree of height 5 requires only 5 hash values, or 160 bytes.

However, the root layer of SPHINCS contains only one tree, recomputed for every signature independently of the selected path in the hyper-tree. The signer can thus cache this layer during key generation in order to save computation time later. Further, we can choose a greater height for this root tree than the other layers, because the cost of key generation is amortized over many signatures (up to 2^{50} for SPHINCS), and in practice key generation does not have the same latency constraints as signing. For the same hyper-tree height, caching reduces the number of layers, which means fewer WOTS per signature, thus smaller signatures.

For example, for a total hyper-tree height of 60 we can use a top layer of height 20, and 8 other layers of height 5, saving 3 WOTS instances. In the top



(a) SPHINCS.

(b) Secret key caching.

Fig. 4: Secret key caching. Triangles represent Merkle trees, black circles represent WOTS signatures. In SPHINCS (left) the hyper-tree is simply made of equal-height trees. With secret key caching (right), a large root Merkle tree is computed at key generation (dashed triangle) and part of it is cached in the secret key (grey triangle). At signature time, the relevant lower part of this root tree is recomputed, along with lower layers in the hyper tree.

tree, the signer can cache the first 15 levels (that is, $2^{16} - 1$ hashes of 32 bytes) with 2 MB of memory. At signature time the signer regenerates the 8 lower layers and the bottom 5 levels of the top layer, as on Figure 4. Compared to SPHINCS, this saves 201 hashes per signature, or 6432 bytes, and speeds up signature and verification due to fewer WOTS instances.

Note that caching does not make the scheme stateful. Contrary to the state of XMSS, our cache is static and not modified by the signing process. The cache can be recomputed on-demand from a small secret seed. This means that one can easily set-up new signing machines by sending a secret seed; there is no need to send the full cache. Similarly, the cache doesn't need to be stored in persistent memory, it can be regenerated after a reboot, a machine reinstallation, etc.

Last, parameters are easy to adapt to the user's configuration. If the signing machine is not powerful enough—such as an embedded device with low memory and power—a smaller cache can be used. On the contrary, more powerful machines can use a larger cache to further reduce signature size.

3.3 Batch Signing

To amortize the cost of signing over many messages, several *batching* methods have been developed. Some methods leverage the algebraic structure of the signature scheme [11,3], but others are more generic: in 1999, Pavlovski et al. proposed a *generic* batch signing method [18] that gathers all the messages, computes a Merkle tree from their respective hashes, and signs only the Merkle tree root with a traditional signature scheme. The signature of each message

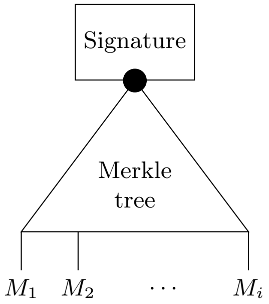


Fig. 5: Pavlovski's batch signing [18]. A batch of messages $M_1, \ldots, M_i$ are hashed together in a Merkle tree. The root of this tree is authenticated with an expensive signature scheme.

then contains the signature of the Merkle tree root and the authentication path for the corresponding message (Figure 5).

In the context of hash-based signatures, batch signing has additional advantages, because of the limits on the number of signatures imposed by WOTS and HORS/PORS. A way to increase the number of signatures is to use a large hyper-tree structure as in XMSS and SPHINCS, but this increases the signature size.

With batch signing, the total number of messages signed can be reduced, and consequently each signature can be made smaller. For example, one can define a signing period W (e.g. a millisecond for TLS connections, a day for software updates), group all messages within each period, and release a single signature at the end of the period. For a given use case, the frequency of signatures $1/W$ is predictable and allows to adapt the signature parameters to the life duration of a key pair. Further, such a scheme is still flexible and allows to occasionally shorten a period (e.g. if an emergency security update must be issued before the end of the day). We note that the precise signing period W is private to the signer, so there is no need to synchronize clocks with recipients.

Batch signing also allows to allocate more computing power to each signature, because this cost is amortized among many messages. Indeed, if N messages are signed within a period, computing N signatures each in time t (without batching) takes the same resources as computing 1 signature in time Nt (with batch signing). In SPHINCS, this allows to increase the height of each layer in the hyper tree, hence reducing the total number of WOTS signatures and the signature size.

Practical Parameters. With a hyper-tree of height 60, SPHINCS authors targeted at most 2^{50} messages per key pair, arguing that it would take more than 30 years to exhaust a key at a rate of 2^{20} messages per second. Even for highly interactive environments, a period W of one millisecond reduces the target to 2^{40} signatures for more than 30 years per key pair, with 2^{10} messages per batch. The latency overhead of one millisecond seems acceptable, given that signing time is an order of magnitude larger on a single CPU[4].

With that in mind, the hyper-tree height of SPHINCS can be reduced by 10, hence removing 2 layers of WOTS signatures, saving 144 hashes, or 4608 bytes. On the other hand, the batching Merkle tree adds 10 authentication nodes per signature, i.e. 320 bytes. A batch index must also be sent, for example on 2 bytes. Overall, batch signing saves 4286 bytes. The height of internal Merkle trees can also be increased to save additional WOTS signatures.

Real-Time Deployment. Batch signing also offers advantages for highly interactive environments (e.g. TLS servers). First, the marginal cost of signing an additional message in a given period is in the order of a few hash evaluations, much faster than computing a full signature. Consequently, batch signing can easily adapt to load variations by gathering more messages in a single Merkle tree, which reduces the risk of denial-of-service attacks that flood the signer with messages.

Second, the overall memory footprint for a signer transmitting signatures to N recipients is reduced to a single signature and a Merkle tree with N leaves, instead of N signatures. Here again, the marginal memory cost of signing an additional message is in the order of a few hashes, thwarting denial-of-service attacks.

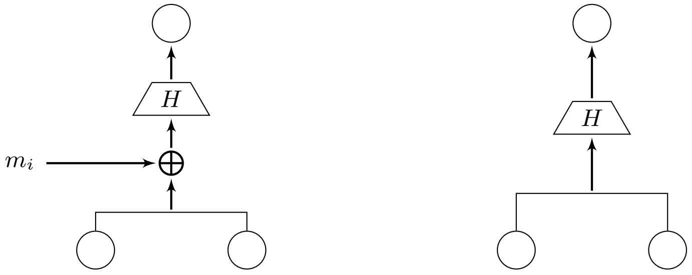
Multi-Authentication. Additionally, Pavlovski's batch signing allows to authenticate a subset of the messages with a single signature. Consider the case of a software repository with daily updates, where each user wants a subset of the packages. After downloading the packages, they only need to fetch one signature for the day and a Merkle authentication path for each package in their chosen subset. They can even use an authentication octopus, as described in §§3.5. This amortizes the signature size compared to fetching a full signature for each package. The marginal cost for authenticating an additional package is $O(\log N)$ where N is the total number of packages in the repository.

3.4 Mask Off

Recent versions of Merkle tree and Winternitz OTS constructions [9,13] interleave hash evaluations with masking. The public key contains a list of uniformly generated random masks, and each hash evaluation is preceded by XOR-ing a mask. The mask to select depends on the location of this hash evaluation in the tree.

Masking allows to relax security requirements to second-preimage resistance instead of collision resistance, but this reduction is less tight and security degrades with the total number of hash evaluations in the construction (for example, 2^h for a Merkle tree of height h). Moreover, against quantum computers,

[4] In SPHINCS, signing takes of the order of 50 million cycles [5].



(a) Masked hashing in SPHINCS.

(b) Mask off.

Fig. 6: Mask-less hashing in Merkle trees. In the masked construction (left), the hash function H is assumed to be second-preimage-resistant. In the mask-less construction (right), H is assumed to be collision-resistant.

collision resistance and second-preimage resistance have approximately the same generic security of $O(2^{n/2})$ for n bits of output [4]. A recent paper described new collision-finding quantum algorithms [8], that are faster but at the expense of more memory use, which makes them less efficient than (parallel) classical ones.

We thus propose to remove masks in these constructions (Figure 6), to obtain a simpler design and reduce the size of public keys. Security proofs for mask-less constructions are given in [10, Ch.6].

3.5 Eliminating Redundancies: Octopus

A significant part of the size of a SPHINCS signature is taken by a single HORST instance, and in particular by $k = 32$ authentication paths in a HORST tree of height 16. SPHINCS already shortened these authentication paths from length 16 to 10 by including all nodes at level 6, as there is a lot of redundancy next to the root. In total, authentication paths contain 384 values, or 12 288 bytes. Yet, on average most of the nodes at level 6 can already be inferred from authentication paths, so there is still some redundancy. Besides, some authentication paths may merge even below this threshold, introducing even more redundancy. We thus propose to use a dynamic strategy to include only necessary values, in what we call an *authentication octopus*. As we show, this approach saves 1909 bytes on average for SPHINCS signatures, and 1024 bytes in the worst case.

Algorithm. The algorithm on Figure 7 computes the optimal authentication octopus, that is, the optimal set of nodes to be included in the signature, given a list of leaf indices. This algorithm works as follows. First, the set of leaf indices is sorted to facilitate identification of siblings, with the convention that we count indices from 0 to $2^h - 1$ from left to right. Then, for each level of the Merkle tree, in a bottom-up order, the algorithm converts the sequence of indices to authenticate at level $\ell + 1$ into a sequence of authentication nodes at level $\ell + 1$ and a sequence of indices to authenticate at level ℓ (Figure 8).

```
proc Octopus($[x_1, \ldots, x_k], h$)
    $Indices \leftarrow$ sorted($[x_1, \ldots, x_k]$)
    $Auth \leftarrow []$
    for $\ell = h - 1$ down to $0$
        $NewIndices \leftarrow []$
        $j \leftarrow 0$
        while $j < Indices$.length()
            $x \leftarrow Indices[j]$
            $NewIndices$.append($\lfloor x/2 \rfloor$)
            $sibling \leftarrow x \oplus 1$
            if $j + 1 < Indices$.length() $\wedge$ $Indices[j + 1] = sibling$
                $j \leftarrow j + 1$
            else
                $Auth$.append($(\ell + 1, sibling)$)
            $j \leftarrow j + 1$
        $Indices \leftarrow NewIndices$
    return $Auth$
```

Fig. 7: Algorithm to compute the optimal authentication octopus. The inputs are the list of leaf indices to authenticate and the Merkle tree height; the result is the list of authentication nodes. Each authentication node contains a level $0 \le \ell \le h$ and an index $0 \le i < 2^\ell$. The sorted() function takes as input a list of integers and returns this list sorted in increasing order.

More precisely, at a given level, for each index we add its parent to the list of indices at the upper level. We then compute the index of its sibling, by flipping the least significant bit. If the next index to authenticate happens to be the sibling, then we skip the sibling, to avoid adding their common parent twice. Otherwise, we add the sibling to the list of authentication nodes. Because the list of indices is always sorted in increasing order, checking the next index is sufficient to identify siblings.

Signature and verification algorithms are easily derived from this algorithm.

Best and Worst Cases. To analyze the extreme cases, we can rephrase the problem as follows. Starting from k tentacles (authentication paths) at the bottom of the tree, we obtain a single root. This means that there must be $k - 1$ merges in the octopus. We can now analyze each merge individually.

First, note that if two tentacles merge at level ℓ, they have identical authentication nodes between level ℓ and the root. Their authentication nodes at level $\ell + 1$ are mutual siblings, hence redundant. This means that if a merge occurs at level ℓ then $\ell + 2$ authentication nodes are redundant (Figure 9). To count the total number of redundant nodes in an octopus, we can simply add the redundant nodes of each merge. Indeed, we can construct an octopus by successively adding tentacles; each new tentacle merges at some level ℓ and saves $\ell + 2$ nodes.

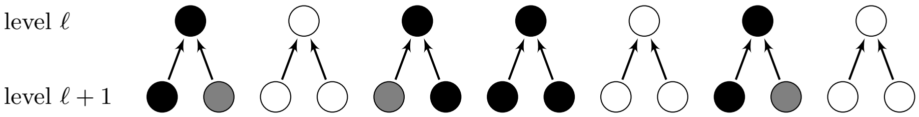


Fig. 8: One iteration of the optimal octopus algorithm. Starting form a set of nodes to authenticate at level $\ell+1$ (black), an iteration computes the set of authentication nodes at level $\ell+1$ (dark grey), and the set of nodes to authenticate at level ℓ, by identification of siblings.

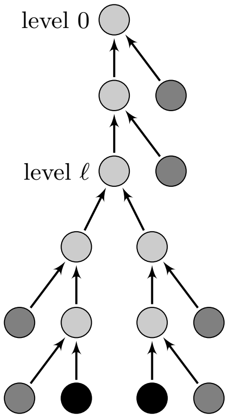


Fig. 9: Merging of two tentacles at level ℓ. The merge removed $\ell+2$ authentication nodes (dark grey). The authentication nodes are identical at levels 1 to ℓ, and no authentication node is needed at level $\ell+1$.

In the best case, all merges are close to the leaves, whereas in the worst case all merges are close to the root. There are however some constraints because the octopus is embedded in a Merkle tree.

- There cannot be more than 2^ℓ merges at level ℓ.
- If there are $k_{\ell+1}$ tentacles at level $\ell+1$, there cannot be more than $\lfloor k_{\ell+1}/2 \rfloor$ merges at level ℓ.

To simplify the analysis, we first assume that k is a power of two.

Lemma 1. *Let k and h be integers such that k is a positive power of 2 and $k \leq 2^h$. Then, given k leaves to authenticate in a Merkle tree of height h, the octopus authentication algorithm of Figure 7 outputs between $h - \log_2 k$ and $k(h - \log_2 k)$ authentication nodes (inclusive).*

To give a visual interpretation, in the best case the octopus is shaped like a broom with a stick of length $h - \log_2 k$ at the top, whereas in the worst case it is shaped like a rake with k teeth of length $h - \log_2 k$ at the bottom.

Proof. In the worst case, all levels up to $\log_2 k - 1$ are saturated with merges, and the number of redundant nodes is:

$$\sum_{\ell=0}^{\log_2 k-1} 2^\ell(\ell+2) = k \log_2 k$$

In the best case, there are $k/2$ merges at level $h-1$, $k/4$ merges at level $h-2$, $\ldots$, and 1 merge at level $h - \log_2 k$. The number of redundant nodes is:

$$\sum_{\ell=1}^{\log_2 k} \frac{k}{2^\ell}(h-\ell+2) = (k-1)h + \log_2 k$$

The result follows. □

We now prove the following theorem, in the general case:

Theorem 1. *Let k and h be integers such that $k \leq 2^h$. Then, given k leaves to authenticate in a Merkle tree of height h, if the octopus authentication algorithm of Figure 7 outputs oct authentication nodes, then:*

$$h - \lceil \log_2 k \rceil \leq \mathsf{oct} \leq k(h - \lfloor \log_2 k \rfloor)$$

Proof. We let $k' = 2^{\lfloor \log_2 k \rfloor}$ be the largest power of two smaller than or equal to k. In the worst case, all levels up to $\log_2 k' - 1$ are saturated with merges, and level $\log_2 k'$ contains $k - k'$ merges. By Lemma 1, the number of redundant nodes is:

$$k' \log_2 k' + (k-k')(\log_2 k' + 2) \geq k \log_2 k'$$

so the number of authentication nodes is at most $k(h - \lfloor \log_2 k \rfloor)$.

In the best case, all merges are at the bottom levels. In particular, it is possible to merge k tentacles in the $\lceil \log_2 k \rceil$ bottom levels. The only remaining tentacle at level $h - \lceil \log_2 k \rceil$ needs $h - \lceil \log_2 k \rceil$ authentication nodes. The lower bound follows. □

Corollary 1. *Compared to the SPHINCS construction, octopus authentication saves at least k hash values—assuming that x is rounded to $\log_2 k$ in SPHINCS.*

Average Case. To study the average case, we denote $\mathsf{mH}(h,k)$ the random variable equal to the minimal number of hash values necessary to authenticate k uniformly distributed leaves in a Merkle tree of height h. We denote $\mathbb{E}_{\mathsf{mH}}(h,k)$ the expectation of $\mathsf{mH}(h,k)$, i.e. the average number of hash values. We follow a bottom-up approach to derive a recurrence relation between consecutive levels, i.e. $\mathbb{E}_{\mathsf{mH}}(h,\cdot)$ and $\mathbb{E}_{\mathsf{mH}}(h+1,\cdot)$, and can then solve the problem by dynamic programming.

We also denote by $P(h,k,i)$ the probability that given k uniformly distributed tentacles at level $h+1$, i merges occur at level h.

Lemma 2. *The probability $P(h,k,i)$ is equal to:*

$$P(h,k,i) = \binom{2^{h+1}}{k}^{-1} \binom{2^h}{k-i} \binom{k-i}{i} 2^{k-2i}$$

Proof. There are $\binom{2^{h+1}}{k}$ sets of k distinct indices at level $h+1$. At level h, there are $\binom{2^h}{k-i}$ choices of $k-i$ merged slots, out of which $\binom{k-i}{i}$ choices of i slots that contain a merge. For each of the $k-2i$ non merged slots at level h, there are two possible slots at level $h+1$. □

Theorem 2. $\mathbb{E}_{\mathsf{mH}}(h,k)$ *satisfies the following recurrence relation:*

$$\begin{aligned} &\mathbb{E}_{\mathsf{mH}}(0,1) = 0 \\ &\mathbb{E}_{\mathsf{mH}}(h+1,k) = \sum_{i=0}^{\lfloor k/2 \rfloor} \left(k - 2i + \mathbb{E}_{\mathsf{mH}}(h,k-i)\right) P(h,k,i) \end{aligned}$$

Proof. First, $\mathbb{E}_{\mathsf{mH}}(0,1) = 0$, because no authentication node is needed for a tree reduced to one node.

We now remark that if k leaf indices are uniformly distributed, and that they have merged into t tentacles at some upper level ℓ, these t tentacles are also uniformly distributed at level ℓ. This is independent of how the k merged into t, so we can view the subtree above level ℓ as a standalone tree of height ℓ.

This allows to derive the recurrence relation between consecutive levels. Indeed, i merges occur at level h with probability $P(h,k,i)$. In that case, $k-2i$ authentication nodes are necessary at level $h+1$ to authenticate the $k-2i$ non-merged tentacles, and $\mathbb{E}_{\mathsf{mH}}(h,k-i)$ authentication nodes are necessary at upper levels on average. □

We can also derive a recurrence relation to compute the standard deviation of $\mathsf{mH}(h,k)$. Recall that it is equal to $\sqrt{\mathbb{E}^{(2)}_{\mathsf{mH}}(h,k) - \mathbb{E}_{\mathsf{mH}}(h,k)^2}$, where $\mathbb{E}^{(2)}_{\mathsf{mH}}(h,k)$ is the expectation of $\mathsf{mH}(h,k)^2$. We can compute $\mathbb{E}^{(2)}_{\mathsf{mH}}$ with the following recurrence relation, and then derive the standard deviation.

$$\mathbb{E}^{(2)}_{\mathsf{mH}}(h{+}1,k) = \sum_{i=0}^{\lfloor k/2 \rfloor} \Big((k-2i)^2 + 2(k-2i)\mathbb{E}_{\mathsf{mH}}(h,k-i) + \mathbb{E}^{(2)}_{\mathsf{mH}}(h,k-i) \Big) P(h,k,i)$$

Application to SPHINCS. Solving the recurrences by dynamic programming, we obtain $\mathbb{E}_{\mathsf{mH}}(h = 16, k = 32) \approx 324.3$ for the proposed SPHINCS parameters, with a standard deviation of 7.1. In contrast, the HORST construction in SPHINCS uses $k(h-x) + 2^x = 384$ authentication values (both for $x = 5$ and $x = 6$). Even in the worst case, our octopus authentication uses only $k(h - \log_2 k) = 352$ authentication values.

Octopus authentication thus saves 1024 bytes in the worst case and 1909 bytes on average, compared to the threshold method proposed for HORST in SPHINCS. Compared to the naive method of using k full authentication paths, octopus authentication saves 6005 bytes on average.

4 Gravity-SPHINCS

Gravity-SPHINCS is our improved version of SPHINCS, which incorporates the improvements discussed. For lack of space in this paper we can't provide a complete specification, so we'll focus on the main differences with SPHINCS and the optimizations implemented in Gravity-SPHINCS.

4.1 Parameters

Gravity-SPHINCS inherits some parameters from SPHINCS (hash length, Winternitz depth, etc.), and has new ones. In the list below h denotes the height of subtrees, rather than the main tree height (like in SPHINCS), and below $B_n = \{0,1\}^n$ denotes the set of n-bit strings:

- the **hash output bit length** n, a positive integer
- the **Winternitz depth** w, a power of two such that $w \geq 2$ and $\log_2 w$ divides n
- the **PORS set size** t, a positive power of two
- the **PORS subset size** k, a positive integer such that $k \leq t$
- the **internal Merkle tree height** h, a positive integer
- the **number of internal Merkle trees** d, a non-negative integer
- the **cache height** c, a non-negative integer
- the **batching height** b, a non-negative integer
- the **message space** $\mathcal{M}$, usually a subset of bit strings $\{0,1\}^*$

From these parameters are derived:

- the **Winternitz width** $\ell = \mu + \lfloor \log_2(\mu(w-1))/\log_2 w \rfloor + 1$ where $\mu = n/\log_2 w$
- the **PORS set** $T = \{0,\dots,t-1\}$
- the **address space** $\mathcal{A} = \{0,\dots,d\} \times \{0,\dots,2^{c+dh}-1\} \times \{0,\dots,\max(\ell,t)-1\}$
- the **public key space** $\mathcal{PK} = B_n$
- the **secret key space** $\mathcal{SK} = B_n^2$
- the **signature space** $\mathcal{SG} = B_n \times B_n^k \times B_n^{\leq k(\log_2 t - \lfloor \log_2 k \rfloor)} \times (B_n^\ell \times B_n^h)^d \times B_n^c$
- the **batched signature space** $\mathcal{SG}_B = B_n^b \times \{0,\dots,2^b-1\} \times \mathcal{SG}$
- the **public key size**, of n bits
- the **secret key size**, of $2n$ bits
- the **maximal signature size**, of

$$\mathsf{sigsz} = (1 + k + k(\log_2 t - \lfloor \log_2 k \rfloor) + d(\ell + h) + c)n \text{ bits}$$

- the **maximal batched signature size**, of $\mathsf{sigsz} + bn + b$ bits

4.2 Primitives

An instance of Gravity-SPHINCS signature scheme is based on four primitives that depend on the parameters n and $\mathcal{M}$:

- a length-preserving hash function $F : B_n \to B_n$
- a length-halving hash function $H : B_n^2 \to B_n$
- a pseudo-random function $G : B_n \times \mathcal{A} \to B_n$ (that takes as input a seed and address)
- a general-purpose hash function $H^* : \mathcal{M} \to B_n$

4.3 Internal Algorithms

We describe the algorithms used for functionalities that are not in the original SPHINCS.

Octopus Authentication. $\mathsf{Octopus\text{-}auth}_h : B_n^{2^h} \times \{0, \dots, 2^h - 1\}^k \to B_n^* \times B_n$

This function takes as input 2^h leaf hashes x_i and $1 \leq k \leq 2^h$ distinct leaf indices $0 \leq j_i < 2^h$ sorted in increasing order, and outputs the associated octopus authentication nodes $oct \in B_n^*$ and the octopus root $r \in B_n$. It is defined by recurrence on h as:

- $\mathsf{Octopus\text{-}auth}_0(x_0, j_1) = (\emptyset, x_0)$,
- $\mathsf{Octopus\text{-}auth}_{h+1}(x_0, x_1, \dots, x_{2i}, x_{2i+1}, j_1, \dots, j_k)$ is computed as

$$\begin{cases} j'_1, \dots, j'_\kappa \leftarrow \mathsf{unique}(\lfloor j_1/2 \rfloor, \dots, \lfloor j_k/2 \rfloor) \\ oct', r \leftarrow \mathsf{Octopus\text{-}auth}_h(H(x_0, x_1), \dots, H(x_{2i}, x_{2i+1}), j'_1, \dots, j'_\kappa) \\ z_1, \dots, z_{2\kappa-k} \leftarrow (j_1 \oplus 1, \dots, j_k \oplus 1) \setminus (j_1, \dots, j_k) \\ a_1, \dots, a_{2\kappa-k} \leftarrow (x_{z_1}, \dots, x_{z_{2\kappa-k}}) \\ oct \leftarrow (a_1, \dots, a_{2\kappa-k}, oct') \end{cases}$$

where $\mathsf{unique}()$ removes duplicates in a sequence, and $A \setminus B$ denotes the set difference.

Octopus Root Extraction. $\mathsf{Octopus\text{-}extract}_{h,k} : B_n^k \times \{0, \dots, 2^h - 1\}^k \times B_n^* \to B_n \cup \{\perp\}$

This function (with $1 \leq k \leq 2^h$) takes as input k leaf hashes $x_i \in B_n$, k leaf indices $0 \leq j_i < 2^h$ and an authentication octopus $oct \in B_n^*$, and outputs the associated Merkle tree root $r \in B_n$, or $\perp$ if the number of hashes in the authentication octopus is invalid. It is defined by recurrence on h as:

- $\mathsf{Octopus\text{-}extract}_{0,1}(x_1, j_1, oct) = \begin{cases} x_1 & \text{if } oct = \emptyset \\ \perp & \text{otherwise} \end{cases}$

- $\mathsf{Octopus\text{-}extract}_{h+1,k}(x_1, \ldots, x_k, j_1, \ldots, j_k, \mathit{oct})$ is computed as

$$\begin{cases} j'_1, \ldots, j'_\kappa \leftarrow \mathsf{unique}(\lfloor j_1/2 \rfloor, \ldots, \lfloor j_k/2 \rfloor) \\ L \leftarrow \mathsf{Octopus\text{-}layer}((x_1, j_1), \ldots, (x_k, j_k), \mathit{oct}) \\ \bot & \text{if } L = \bot \\ \mathsf{Octopus\text{-}extract}_{h,\kappa}(x'_1, \ldots, x'_\kappa, j'_1, \ldots, j'_\kappa, \mathit{oct}') & \text{if } L = (x'_1, \ldots, x'_\kappa, \mathit{oct}') \end{cases}$$

where $\mathsf{Octopus\text{-}layer}()$ is defined by recurrence as:

- $\mathsf{Octopus\text{-}layer}(x_1, j_1, \mathit{oct}) = \begin{cases} \bot & \text{if } \mathit{oct} = \emptyset \\ H(x_1, a), \mathit{oct}' & \text{if } \mathit{oct} = (a, \mathit{oct}') \wedge j_1 \mod 2 = 0 \\ H(a, x_1), \mathit{oct}' & \text{if } \mathit{oct} = (a, \mathit{oct}') \wedge j_1 \mod 2 = 1 \end{cases}$
- $\mathsf{Octopus\text{-}layer}(x_1, j_1, x_2, j_2, \ldots, x_k, j_k, \mathit{oct})$ is

$$\begin{cases} H(x_1, x_2), \mathsf{Octopus\text{-}layer}(x_3, j_3, \ldots, x_k, j_k, \mathit{oct}) & \text{if } j_1 \oplus 1 = j_2 \\ \bot & \text{if } j_1 \oplus 1 \neq j_2 \wedge \mathit{oct} = \emptyset \\ H(x_1, a), \mathsf{Octopus\text{-}layer}(x_2, j_2, \ldots, x_k, j_k, \mathit{oct}') & \text{if } \mathit{oct} = (a, \mathit{oct}') \wedge j_1 \mod 2 = 0 \\ H(a, x_1), \mathsf{Octopus\text{-}layer}(x_2, j_2, \ldots, x_k, j_k, \mathit{oct}') & \text{if } \mathit{oct} = (a, \mathit{oct}') \wedge j_1 \mod 2 = 1 \end{cases}$$

Operations on Addresses. Before describing PORS as used in Gravity-SPHINCS, we need to describe operations on *addresses* within the hyper-tree. Each WOTS and PORST instance has a unique address that allows to generate its secret values on demand, and each address contains:

- a layer $0 \leq i \leq d$ in the hyper-tree, where 0 is the root layer, $d - 1$ is the last WOTS layer and d is the PORST layer;
- an instance index j in the layer, with $0 \leq j < 2^{c+(i+1)h}$ if $i < d$ and $0 \leq j < 2^{c+dh}$ if $i = d$;
- a counter λ in the instance, with $0 \leq \lambda < \ell$ if $i < d$ and $0 \leq \lambda < t$ if $i = d$.

We define the following functions to manipulate addresses.

- The function $\mathsf{make\text{-}addr} : \{0, \ldots, d\} \times \mathbb{N} \to \mathcal{A}$ takes as input a layer $i \in \{0, \ldots, d\}$ and an index $j \in \mathbb{N}$ and returns the address $a = (i, j \mod 2^{c+dh}, 0) \in \mathcal{A}$.
- The function $\mathsf{incr\text{-}addr} : \mathcal{A} \times \mathbb{N} \to \mathcal{A}$ takes as input an address $a = (i, j, \lambda)$ and an integer x and returns the address $a' = (i, j, \lambda + x) \in \mathcal{A}$ with the counter incremented by x.

PRNG to Obtain a Random Subset. The function $\mathsf{PORS} : B_n \times B_n \to \mathbb{N} \times T^k$ takes as input a salt $s \in B_n$ and a hash $x \in B_n$, and outputs a hyper-tree index $\lambda \in \mathbb{N}$ and k distinct indices x_i, computed as follows.

- Compute $g \leftarrow H(s, x)$.
- Let $a \leftarrow \mathsf{make\text{-}addr}(0, 0)$.

- Compute $b \leftarrow G(g, a)$ and interpret it as the big-endian encoding of an integer $\beta \in \{0, \dots, 2^n - 1\}$.
- Compute $\lambda \leftarrow \beta \mod 2^{c+dh}$. In other words, λ is the big-endian interpretation of the $c + dh$ last bits of the block b.
- Initialize $X \leftarrow \emptyset$ and $j \leftarrow 0$.
- While $|X| < k$ do the following:
  - increment $j \leftarrow j + 1$,
  - compute $b \leftarrow G(g, \textsf{incr-addr}(a, j))$,
  - split b into $\nu = \lfloor n / \log_2 t \rfloor$ blocks of $\log_2 t$ bits, as $b_i = \textsf{substr}(b, (i - 1)\log_2 t, \log_2 t)$,
  - for $i \in \{1, \dots, \nu\}$ interpret b_i as the big-endian encoding of an integer $\overline{b_i} \in T$,
  - for $i \in \{1, \dots, \nu\}$, if $|X| < k$ compute $X \leftarrow \textsf{unique}(X, \overline{b_i})$.
- Compute $(x_1, \dots, x_k) \leftarrow \textsf{sorted}(X)$.

PORST Signature. The function $\textsf{PORST-sign} : B_n \times \mathcal{A} \times T^k \rightarrow B_n^k \times B_n^* \times B_n$ takes as input a secret $\textsf{seed} \in B_n$, a base address $a \in \mathcal{A}$ and k sorted indices $x_i \in T$, and outputs the associated PORST signature $(\sigma, oct) \in B_n^k \times B_n^*$ and PORST public key $p \in B_n$, computed as follows.

- For $i \in \{1, \dots, t\}$ compute the secret value $s_i \leftarrow G(\textsf{seed}, \textsf{incr-addr}(a, i - 1))$.
- For $j \in \{1, \dots, k\}$ set the signature value $\sigma_j = s_{x_j}$.
- Compute the authentication octopus and root as

$$oct, p \leftarrow \textsf{Octopus-auth}_{\log_2 t}(s_1, \dots, s_t, x_1, \dots, x_k)$$

PORST Public Key Extraction. The function $\textsf{PORST-extractpk} : T^k \times B_n^k \times B_n^* \rightarrow B_n \cup \{\perp\}$ takes as input k indices $x_i \in T$ and a PORST signature $(\sigma, oct) \in B_n^k \times B_n^*$, and outputs the associated PORST public key $p \in B_n$, or $\perp$ if the authentication octopus is invalid, computed as follows.

- Compute the octopus root $p \leftarrow \textsf{Octopus-extract}_{\log_2 t, k}(\sigma, x_1, \dots, x_k, oct)$.

4.4 Batch Operations

Single-message signature and verification in Gravity-SPHINCS is similar, though not identical, to SPHINCS'. We thus only describe the new batch operations.

Batch Signature. The batch signature procedure takes as input a sequence of messages $(M_1, \dots, M_i) \in \mathcal{M}^i$ with $0 < i \leq 2^b$ and a secret key $sk = (\textsf{seed}, \textsf{salt})$ along with its secret cache, and outputs i signatures σ_j computed as follows.

- For $j \in \{1, \dots, i\}$ compute the message digest $m_j \leftarrow H^*(M_j)$.
- For $j \in \{i + 1, \dots, 2^b\}$ set $m_j \leftarrow m_1$.
- Compute $m \leftarrow \textsf{Merkle-root}_b(m_1, \dots, m_{2^b})$.
- Compute $\sigma \leftarrow \mathcal{S}(sk, m)$.
- For $j \in \{1, \dots, i\}$ the j-th signature is $\sigma_j \leftarrow (j, \textsf{Merkle-auth}_b(m_1, \dots, m_{2^b}, j), \sigma)$.

Batch Verification. The batch verification procedure takes as input a public key pk, a message $M \in \mathcal{M}$ and a signature (j, A, σ), and works as follows.

- Compute the message digest $m \leftarrow H^*(M)$.
- Compute the Merkle root $m \leftarrow \mathsf{Merkle\text{-}extract}_b(m, j, A)$.
- Return $\mathcal{V}(pk, m, \sigma)$, the result of the non-batch verification procedure.

4.5 Instances Proposed

We propose parameters and primitives for Gravity-SPHINCS, with

- hash output of $n = 256$ bits to aim for 128-bit collision-resistance, both classical and quantum
- Winternitz depth $w = 16$, a good trade-off between size and speed often chosen in similar constructions (XMSS, SPHINCS)
- PORS set size $t = 2^{16}$, here again a good trade-off between size and speed chosen in SPHINCS

For the hash functions, we use 6-round version Haraka-v2-256 as F and 6-round Haraka-v2-512 as H. We thus extend the original Haraka-v2 construction [16] with an additional round.

For the general-purpose hash function H^* we use SHA-256. For G we use a construction that is essentially AES-256-CTR.

name	$\log_2 t$	k	h	d	c	sigsz	capacity
NIST-fast	16	28	5	10	14	35 168	2^{64}
NIST	16	28	8	6	16	26 592	2^{64}
NIST-slow	16	28	14	4	8	22 304	2^{64}
fast	16	32	5	7	15	28 928	2^{50}
batched	16	32	8	3	16	20 032	2^{40}
small	16	24	5	1	10	12 640	2^{10}

Table 1: Proposed Gravity-SPHINCS parameters for 128-bit quantum security. The capacity is the number of messages (or batches thereof) that can be signed per key pair. The maximal signature size sigsz is in bytes and does not include batching. Public keys are always 32-byte, secret keys are always 64-byte.

We propose the following instances, summarized in Table 1.

- Three modes suitable for the NIST call for proposals for post-quantum signature schemes. Submission requirements mandate a capacity of at least

2^{64} messages per key pair [1, Section 4.A.4]. We propose several trade-offs between signing time and signature size.
- A mode suitable to sign up to 2^{50} messages, for comparison with SPHINCS [5].
- A batched mode, suitable to sign up to 2^{40} batches. This is a reasonable alternative for a capacity of 2^{50} messages (with batches of 2^{10} messages), for applications that can handle batching.
- A small mode with a capacity of 2^{10} messages, for applications that don't need to sign many messages.

Verification times are similar in all cases, and much faster than signing.

References

1. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. NIST (Dec 2016), http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf
2. Aumasson, J.P., Endignoux, G.: Clarifying the subset resilience problem. Cryptology ePrint Archive, Report 2017/909 (2017)
3. Bellare, M., Garay, J.A., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. In: EUROCRYPT (1998)
4. Bernstein, D.J.: Cost analysis of hash collisions: Will quantum computers make sharcs obsolete? SHARCS09 Special-purpose Hardware for Attacking Cryptographic Systems p. 105 (2009)
5. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: EUROCRYPT (2015)
6. Buchmann, J.A., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Post-Quantum Cryptography (2011)
7. Buchmann, J.A., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: Post-Quantum Cryptography (2008)
8. Chailloux, A., Naya-Plasencia, M., Schrottenloher, A.: An efficient quantum collision search algorithm and implications on symmetric cryptography. Cryptology ePrint Archive, Report 2017/847 (2017)
9. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Post-Quantum Cryptography (2008)
10. Endignoux, G.: Design and implementation of a post-quantum hash-based cryptographic signature scheme. Master's thesis, EPFL (2017)
11. Fiat, A.: Batch RSA. In: CRYPTO (1989)
12. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press (2004)
13. Hülsing, A.: W-OTS+ - shorter signatures for hash-based signature schemes. In: AFRICACRYPT (2013)
14. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: PKCS (2016)
15. Jakobsson, M., Leighton, F.T., Micali, S., Szydlo, M.: Fractal merkle tree representation and traversal. In: CT-RSA (2003)

16. Klbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 - efficient short-input hashing for post-quantum applications. Cryptology ePrint Archive, Report 2016/098 (2016)
17. Merkle, R.C.: A certified digital signature. In: CRYPTO (1989)
18. Pavlovski, C., Boyd, C.: Efficient batch signature generation using tree structures. In: International Workshop on Cryptographic Techniques and E-Commerce, CrypTEC. vol. 99 (1999)
19. Reyzin, L., Reyzin, N.: Better than BiBa: Short one-time signatures with fast signing and verifying. In: ACISP (2002)
20. Szydlo, M.: Merkle tree traversal in log space and time. In: EUROCRYPT (2004)