Gravity-SPHINCS

v1, November 20, 2017

Inventor, submitter, main contact:

Jean-Philippe Aumasson, Kudelski Security

Route de Genève 22, 1033 Cheseaux-sur-Lausanne, Switzerland

`jeanphilippe.aumasson@gmail.com`, +41 79 726 05 08

Inventor, submitter, backup contact:

Guillaume Endignoux

`guillaume.endignoux@m4x.org`

Contents

1 Introduction

Gravity-SPHINCS is a hash-based signature scheme that is a variant of SPHINCS [3]. Like SPHINCS, Gravity-SPHINCS is stateless, as required by NIST.

Gravity-SPHINCS brings a number of optimizations and new features to SPHINCS:

- **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 hashing** to reduce the key size and simplify the scheme.
- **Octopus authentication**, a technique to eliminate redundancies from authentication paths in Merkle trees, and thus reduce signature size.

Detailed analyses related to Gravity-SPHINCS are available in [1, 2, 4].

1.1 Advantages and Limitations

Advantages:

- **High-assurance**: Security essentially depends on the collision resistance of hash functions, an assumption unlikely to fail for the proposed functions.
- **Speed/size trade-offs**: Gravity-SPHINCS parameters and secret key caching allow for a range of trade-offs between 1) the key generation and signing time and 2) the size of keys and of signatures.
- **Batch signing** allows to reduce the per-message signing time and signature size.

Limitations:

- **Complexity**: Gravity-SPHINCS isn't the simplest signature scheme ever.
- **Signature size**: Signatures aren't small (around 20–30 KiB), but this size remains manageable for many applications.

1.2 High-Level View

Like the original SPHINCS, Gravity-SPHINCS can be seen as an extension of Goldreich's [5, §6.4.2] constuction of a stateless hash-base signature scheme that is a binary authentication tree of one-time signatures (OTS). SPHINCS is essentially Goldreich's construction with the following modifications.

1. Inner nodes of the tree are not OTSs but *Merkle trees* whose leaves are OTSs, namely Winternitz OTS (WOTS) [8, 6] instances. Thanks to Merkle trees, each node can sign up to 2^x children nodes, where x is the height of the Merkle tree. SPHINCS thus uses a tree of trees, or *hyper-tree*. This change increases signing time compared to Goldreich's scheme, 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 few-time signature (FTS) schemes. The FTS in SPHINCS is Reyzin—Reyzin's [9] hash-to-obtain-a-random-subset (HORS) construction where public keys are "compressed" thanks to a Merkle tree. This variant is called *HORST*, for HORS with trees. Leaves can sign more than one message, which increases the resilience to path collisions, hence reducing the height needed for the hyper-tree.

Like SPHINCS, Gravity-SPHINCS can be described as the combination of four types of trees (see Figure 1.1)

- **Type 1: The hyper-tree**, whose root is part of the public key. The nodes of this tree are type-2 trees, and its leaves are type-4 trees.
- **Type 2: The subtrees**, which are Merkle trees whose leaves are roots of type-3 trees; said roots connect a type-2 tree to another type-2 tree at a lower layer.
- **Type 3: The WOTS public key compression trees**, which are L-trees (and not necessarily complete binary trees). The leaves of this tree are components of a WOTS public key. The associated WOTS instance signs a type-2 tree's root.
- **Type 4: The PORST public key compression trees**, at the bottom of the hyper-tree, which are Merkle trees whose root is a compressed representation of the actual PORS public key (that is, the Type-4 tree's leaves).

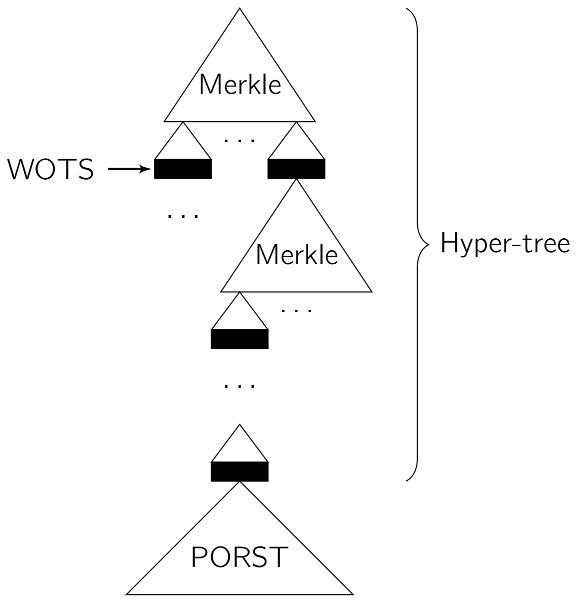

Figure 1.1: Sketch of the Gravity-SPHINCS construction.

2 Specification

This chapter includes a formal specification of Gravity-SPHINCS. For more context and an introduction to hash-based signature schemes, we refer to [4, 2].

2.1 Parameters

An instance of Gravity-SPHINCS requires the following parameters:

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

From these parameters are derived the following values, where $B_n = \{0,1\}^n$ denotes the set of n-bit strings:

- **Winternitz width** $\ell = \mu + \lfloor \log_2(\mu(w-1))/\log_2 w \rfloor + 1$ where $\mu = n/\log_2 w$
- **PORS set** $T = \{0, \dots, t-1\}$
- **Address space** $\mathcal{A} = \{0, \dots, d\} \times \{0, \dots, 2^{c+dh} - 1\} \times \{0, \dots, \max(\ell, t) - 1\}$
- **Public key space** $\mathcal{PK} = B_n$
- **Secret key space** $\mathcal{SK} = B_n^2$
- **Signature space** $\mathcal{SG} = B_n \times B_n^k \times B_n^{\leq k(\log_2 t - \lfloor \log_2 k \rfloor)} \times (B_n^\ell \times B_n^h)^d \times B_n^c$
- **Batched signature space** $\mathcal{SG}_B = B_n^b \times \{0, \dots, 2^b - 1\} \times \mathcal{SG}$

- **Public key size**, of n bits
- **Secret key size**, of $2n$ bits
- **Maximal signature size**, of

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

- **Maximal batched signature size**, of $\text{sigsz} + bn + b$ bits

2.2 Primitives

An instance of Gravity-SPHINCS needs four primitives, which 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 pseudorandom function $G : B_n \times \mathcal{A} \to B_n$ (takes as input a seed and an address)
- A general-purpose hash function $H^* : \mathcal{M} \to B_n$

2.3 Internal Algorithms

We first define algorithms that are building blocks of Gravity-SPHINCS.

2.3.1 Operations on Addresses

Within the hyper-tree, each WOTS and PORST instance is assigned a unique address in order to generate its secret values on demand. Each address contains:

- A layer index $0 \leq i \leq d$ 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 within 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.

- make-addr : $\{0,\dots,d\} \times \mathbb{N} \to \mathcal{A}$, which takes as input a layer $i \in \{0,\dots,d\}$ and an index $j \in \mathbb{N}$ and returns the address $a = (i, j \mod 2^{c+dh}, 0) \in \mathcal{A}$.
- incr-addr : $\mathcal{A} \times \mathbb{N} \to \mathcal{A}$, which 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.

2.3.2 L-Tree

The function L-tree : $B_n^+ \to B_n$ takes as input a sequence of hashes $x_i \in B_n$ and returns the associated L-tree root $r \in B_n$, defined by recurrence as follows.

$$\begin{cases} \text{L-tree}(x_1) = x_1 \\ \text{L-tree}(x_1,\dots,x_{2i+2}) = \text{L-tree}(H(x_1,x_2),\dots,H(x_{2i+1},x_{2i+2})) \\ \text{L-tree}(x_1,\dots,x_{2i+3}) = \text{L-tree}(H(x_1,x_2),\dots,H(x_{2i+1},x_{2i+2}),x_{2i+3}) \end{cases}$$

2.3.3 Winternitz Checksum

The function checksummed : $B_n \to \{0,\dots,w-1\}^\ell$ takes as input a hash $x \in B_n$ and returns ℓ integers x_i, computed as follows.

- For $i \in \{1,\dots,\mu\}$ compute $z_i \leftarrow \text{substr}(x, (i-1)\log_2 w, \log_2 w)$, where $\text{substr}(x, j, m)$ denotes the substring of x of length m bits starting at bit index $0 \le j < |x|$.
- For $i \in \{1,\dots,\mu\}$ interpret z_i as the big-endian encoding of a number $0 \le x_i < w$.
- Compute the checksum $C = \sum_{i=1}^{\mu} w - 1 - x_i$.
- For $i \in \{\mu+1,\dots,\ell\}$ compute $x_i = \lfloor C/w^{i-\mu-1} \rfloor \mod w$. In other words, $(x_{\mu+1},\dots,x_\ell)$ is the base-w little-endian encoding of the checksum C.

2.3.4 Winternitz Public Key Generation

The function WOTS-genpk : $B_n \times \mathcal{A} \to B_n$ takes as input a secret seed $\in B_n$ and a base address $a \in \mathcal{A}$, and outputs the associated Winternitz public key $p \in B_n$, computed as follows.

- For $i \in \{1,\dots,\ell\}$ compute the secret value $s_i \leftarrow G(\text{seed}, \text{incr-addr}(a, i-1))$.
- For $i \in \{1,\dots,\ell\}$ compute the public value $p_i \leftarrow F^{w-1}(s_i)$ where the F^{w-1} denotes the function F iterated $w-1$ times.
- Compute $p \leftarrow \text{L-tree}(p_1,\dots,p_\ell)$.

2.3.5 Winternitz Signature

The function WOTS-sign : $B_n \times \mathcal{A} \times B_n \to B_n^\ell$ takes as input a secret seed $\in B_n$, a base address $a \in \mathcal{A}$ and a hash $x \in B_n$, and outputs the associated Winternitz signature $\sigma \in B_n^\ell$, computed as follows.

- For $i \in \{1, \ldots, \ell\}$ compute the secret value $s_i \leftarrow G(\text{seed}, \text{incr-addr}(a, i-1))$.
- Compute $(x_1, \ldots, x_\ell) \leftarrow \text{checksummed}(x)$.
- For $i \in \{1, \ldots, \ell\}$ compute the signature value $\sigma_i \leftarrow F^{x_i}(s_i)$.

2.3.6 Winternitz Public Key Extraction

The function WOTS-extractpk : $B_n \times B_n^\ell \to B_n$ takes as input a hash $x \in B_n$ and a signature $\sigma \in B_n^\ell$, and outputs the associated Winternitz public key $p \in B_n$, computed as follows.

- Compute $(x_1, \ldots, x_\ell) \leftarrow \text{checksummed}(x)$.
- For $i \in \{1, \ldots, \ell\}$ compute the public value $p_i \leftarrow F^{w-1-x_i}(\sigma_i)$.
- Compute $p \leftarrow \text{L-tree}(p_1, \ldots, p_\ell)$.

2.3.7 Merkle Tree Root

The function Merkle-root$_h$: $B_n^{2^h} \to B_n$ takes as input 2^h leaf hashes x_i, and outputs the associated Merkle tree root $r \in B_n$. It is defined by recurrence on h as:

- $\text{Merkle-root}_0(x_0) = x_0$,
- $\text{Merkle-root}_{h+1}(x_0, x_1, \ldots, x_{2i}, x_{2i+1}) = \text{Merkle-root}_h(H(x_0, x_1), \ldots, H(x_{2i}, x_{2i+1}))$.

2.3.8 Merkle Tree Authentication

The function Merkle-auth$_h$: $B_n^{2^h} \times \{0, \ldots, 2^h - 1\} \to B_n^h$ takes as input 2^h leaf hashes x_i and a leaf index $0 \le j < 2^h$, and outputs the associated Merkle tree authentication path $(a_1, \ldots, a_h) \in B_n^h$. It is defined by recurrence on h as:

- $\text{Merkle-auth}_1(x_0, x_1, j) = a_1 \leftarrow x_{j \oplus 1}$ where $\oplus$ denotes the bitwise XOR operation on non-negative integers,

- Merkle-auth$_{h+1}(x_0, x_1, \ldots, x_{2i}, x_{2i+1}, j)$ is

$$\begin{cases} a_1 \leftarrow x_{j \oplus 1} \\ a_2, \ldots, a_{h+1} \leftarrow \text{Merkle-auth}_h(H(x_0, x_1), \ldots, H(x_{2i}, x_{2i+1}), \lfloor j/2 \rfloor) \end{cases}$$

2.3.9 Merkle Tree Root Extraction

The function Merkle-extract$_h : B_n \times \{0, \ldots, 2^h - 1\} \times B_n^h \to B_n$ takes as input a leaf hash $x \in B_n$, a leaf index $0 \le j < 2^h$ and an authentication path $(a_1, \ldots, a_h) \in B_n^h$, and outputs the associated Merkle tree root $r \in B_n$. It is defined by recurrence on h as:

- Merkle-extract$_0(x, j) = x$,
- Merkle-extract$_{h+1}(x, j, a_1, \ldots, a_{h+1}) =$ Merkle-extract$_h(x', \lfloor j/2 \rfloor, a_2, \ldots, a_{h+1})$ where

$$x' = \begin{cases} H(x, a_1) & \text{if } j \mod 2 = 0 \\ H(a_1, x) & \text{if } j \mod 2 = 1 \end{cases}$$

2.3.10 Octopus Authentication

The function Octopus-auth$_h : B_n^{2^h} \times \{0, \ldots, 2^h - 1\}^k \to B_n^* \times B_n$ takes as input 2^h leaf hashes x_i and $1 \le k \le 2^h$ distinct leaf indices $0 \le j_i < 2^h$ 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:

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

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

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

In other words, Octopus-auth$_h$ is a collective Merkle tree authentication for multiple leaves, that takes care to remove redundant authentication nodes.

2.3.11 Octopus Root Extraction

The function $\text{Octopus-extract}_{h,k} : B_n^k \times \{0, \ldots, 2^h - 1\}^k \times B_n^* \to B_n \cup \{\perp\}$ (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:

- $\text{Octopus-extract}_{0,1}(x_1, j_1, oct) = \begin{cases} x_1 & \text{if } oct = \emptyset \\ \perp & \text{otherwise} \end{cases}$,
- $\text{Octopus-extract}_{h+1,k}(x_1, \ldots, x_k, j_1, \ldots, j_k, oct)$ is computed as

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

where Octopus-layer() is defined by recurrence as:

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

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

In other words, Octopus-layer() consumes authentication values from the octopus oct along with nodes x_i and their indices j_i to obtain nodes at the upper layer.

2.3.12 PRNG to Obtain a Random Subset

The function $\text{PORS} : B_n \times B_n \to \mathbb{N} \times T^k$ 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 $a \leftarrow \text{make-addr}(0, 0)$.

- 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 $b \leftarrow G(g, \text{incr-addr}(a, j))$,
  - split b into $\nu = \lfloor n / \log_2 t \rfloor$ blocks of $\log_2 t$ bits, as $b_i = \text{substr}(b, (i-1) \log_2 t, \log_2 t)$,
  - 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 $X \leftarrow \text{unique}(X, \overline{b_i})$.
- Compute $(x_1, \dots, x_k) \leftarrow \text{sorted}(X)$.

2.3.13 PORST Signature

The function PORST-sign : $B_n \times \mathcal{A} \times T^k \to B_n^k \times B_n^* \times B_n$ takes as input a secret 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 $s_i \leftarrow G(\text{seed}, \text{incr-addr}(a, i-1))$.
- For $j \in \{1, \dots, k\}$ set the signature value $\sigma_j = s_{x_j}$.
- Compute the authentication octopus and root as

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

2.3.14 PORST Public Key Extraction

The function PORST-extractpk : $T^k \times B_n^k \times B_n^* \to B_n \cup \{\perp\}$ 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 $p \leftarrow \text{Octopus-extract}_{\log_2 t, k}(\sigma, x_1, \dots, x_k, oct)$.

2.4 Signature Scheme

We now specify the $(\mathcal{KG}, \mathcal{S}, \mathcal{V})$ algorithms for Gravity-SPHINCS, as well as batched variants $(\mathcal{KG}, \mathcal{S}_B, \mathcal{V}_B)$. To simplify, we specify them without secret key caching by the signer. Indeed, this caching optimization is internal to the signer – to increase signing speed – and does not change the public results (public key, signature). We discuss this optimization in §4.2.

2.4.1 Key Generation

$\mathcal{KG}$ takes as input $2n$ bits of randomness and outputs the secret key $sk \in B_n^2$ and the public key $pk \in B_n$.

- Generate the secret key from $2n$ bits of randomness $sk = (\mathrm{seed}, \mathrm{salt}) \overset{\$}{\leftarrow} B_n^2$.
- For $0 \le i < 2^{c+h}$ generate a Winternitz public key

$$x_i \leftarrow \text{WOTS-genpk}(\mathrm{seed}, \text{make-addr}(0, i))$$

- Generate the public key $pk \leftarrow \text{Merkle-root}_{c+h}(x_0, \dots, x_{2^{c+h}-1})$.

2.4.2 Signature

$\mathcal{S}$ takes as input a hash $m \in B_n$ and a secret key $sk = (\mathrm{seed}, \mathrm{salt})$, and outputs a signature computed as follows.

- Compute the public salt $s \leftarrow H(\mathrm{salt}, m)$.
- Compute the hyper-tree index and random subset as $j, (x_1, \dots, x_k) \leftarrow \mathrm{PORS}(s, m)$.
- Compute the PORST signature and public key

$$(\sigma_d, oct, p) \leftarrow \text{PORST-sign}(\mathrm{seed}, \text{make-addr}(d, j), x_1, \dots, x_k)$$

- For $i \in \{d-1, \dots, 0\}$ do the following:
  - compute the WOTS signature $\sigma_i \leftarrow \text{WOTS-sign}(\mathrm{seed}, \text{make-addr}(i, j), p)$,
  - compute $p \leftarrow \text{WOTS-extractpk}(p, \sigma_i)$,
  - set $j' \leftarrow \lfloor j/2^h \rfloor$,
  - for $u \in \{0, \dots, 2^h - 1\}$ compute the WOTS public key

$$p_u \leftarrow \text{WOTS-genpk}(\mathrm{seed}, \text{make-addr}(i, 2^h j' + u))$$

  - compute the Merkle authentication $A_i \leftarrow \text{Merkle-auth}_h(p_0, \dots, p_{2^h-1}, j - 2^h j')$,

    - set $j \leftarrow j'$.

- For $0 \leq u < 2^{c+h}$ compute the WOTS public key

$$p_u \leftarrow \text{WOTS-genpk}(\text{seed}, \text{make-addr}(0, u))$$

- Compute the Merkle authentication

$$(a_1, \ldots, a_{h+c}) \leftarrow \text{Merkle-auth}_{h+c}(p_0, \ldots, p_{2^{h+c}-1}, 2^h j)$$

- Set $A_c \leftarrow (a_{h+1}, \ldots, a_{h+c})$.
- The signature is $(s, \sigma_d, oct, \sigma_{d-1}, A_{d-1}, \ldots, \sigma_0, A_0, A_c)$.

2.4.3 Verification

$\mathcal{V}$ takes as input a hash $m \in B_n$, a public key $pk \in B_n$ and a signature

$$(s, \sigma_d, oct, \sigma_{d-1}, A_{d-1}, \ldots, \sigma_0, A_0, A_c)$$

and verifies it as follows.

- Compute the hyper-tree index and random subset as $j, (x_1, \ldots, x_k) \leftarrow \text{PORS}(s, m)$.
- Compute the PORST public key $p \leftarrow \text{PORST-extractpk}(x_1, \ldots, x_k, \sigma_d, oct)$.
- If $p = \perp$, then abort and return 0.
- For $i \in \{d-1, \ldots, 0\}$ do the following:
    - compute the WOTS public key $p \leftarrow \text{WOTS-extractpk}(p, \sigma_i)$,
    - set $j' \leftarrow \lfloor j/2^h \rfloor$,
    - compute the Merkle root $p \leftarrow \text{Merkle-extract}_h(p, j - 2^h j', A_i)$,
    - set $j \leftarrow j'$.
- Compute the Merkle root $p \leftarrow \text{Merkle-extract}_c(p, j, A_c)$.
- The result is 1 if $p = pk$, and 0 otherwise.

2.4.4 Batch Signature

$\mathcal{S}_B$ takes as input a sequence of messages $(M_1, \ldots, M_i) \in \mathcal{M}^i$ with $0 < i \leq 2^b$ and a secret key $sk = (\text{seed}, \text{salt})$ along with its secret cache, and outputs i signatures σ_j, computed as follows.

- For $j \in \{1, \ldots, i\}$ compute the message digest $m_j \leftarrow H^*(M_j)$.

- For $j \in \{i+1, \ldots, 2^b\}$ set $m_j \leftarrow m_1$.
- Compute $m \leftarrow \text{Merkle-root}_b(m_1, \ldots, m_{2^b})$.
- Compute $\sigma \leftarrow \mathcal{S}(sk, m)$.
- For $j \in \{1, \ldots, i\}$ the j-th signature is $\sigma_j \leftarrow (j, \text{Merkle-auth}_b(m_1, \ldots, m_{2^b}, j), \sigma)$.

For $b = 0$, we simplify $\mathcal{S}_B(sk, M)$ to $\mathcal{S}(sk, H^*(M))$.

2.4.5 Batch Verification

$\mathcal{V}_B$ takes as input a public key pk, a message $M \in \mathcal{M}$ and a signature (j, A, σ), and verifies it as follows.

- Compute the message digest $m \leftarrow H^*(M)$.
- Compute the Merkle root $m \leftarrow \text{Merkle-extract}_b(m, j, A)$.
- The result is $\mathcal{V}(pk, m, \sigma)$.

For $b = 0$, we simplify $\mathcal{V}_B(pk, M, \sigma)$ to $\mathcal{V}(pk, H^*(M), \sigma)$.

2.5 Proposed Instances

We now propose concrete instances of parameters and primitives for Gravity-SPHINCS.

2.5.1 Parameters

We propose the following parameters, common to all our proposed instances.

- Hash output $n = 256$ bits, to aim for 128 bits of security for 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).
- A PORS set size $t = 2^{16}$, here again a good trade-off between size and speed chosen in SPHINCS.

Given these, Table 2.1 gives the parameters of our three proposed instances:

name	$\log_2 t$	k	h	d	c	sig	sk	capacity
S	16	24	5	1	10	12 640	64 KiB	2^{10}
M	16	32	5	7	15	28 929	2 MiB	2^{50}
L	16	28	5	10	14	35 168	1 MiB	2^{64}

Table 2.1: Proposed parameters for Gravity-SPHINCS, suitable for 128 bits of post-quantum security. The capacity is the number of messages (or batches of messages) that can be signed per key pair. The value under "sig" is the maximal signature byte size. The value under "sk" includes the 64-byte secret key plus the cached data. Public keys are always 32-byte.

- **Instance S** produces signatures of at most 12 640 bytes and provides 128-bit security if no more than 2^{10} messages are signed. This version is for use cases where a limited number of signatures is issued (firmware signatures, certificate authorities, and so on).
- **Instance M** produces signatures of at most 28 929 bytes and provides 128-bit security if no more than 2^{50} messages are signed. This version is suitable for most use cases.
- **Instance L** produces signatures of at most 35 168 bytes and provides 128-bit security if no more than 2^{64} messages are signed. This version is for use cases where a single secret key may issue more than 2^{50} signatures.

If more than the authorized number of messages are signed, then the security level slowly degrades, and may eventually allow attackers to forge signatures efficiently. Said forgeries would not be based on the real secret key, however, and could be distinguished from legitimately issued signatures.

2.5.2 Primitives

2.5.2.1 Hash Functions

For the hash functions, we propose to use a 6-round version of Haraka-v2-256 as F, and a 6-round version of Haraka-v2-512 as H. We extend the original Haraka-v2 construction with an additional round, whose constants are the following, computed with the same formula defined in [7]:

$$\begin{aligned}
RC_{40} &= \texttt{2ff372380de7d31e367e4778848f2ad2}\\
RC_{41} &= \texttt{08d95c6acf74be8bee36b135b73bd58f}\\
RC_{42} &= \texttt{5880f434c9d6ee9866ae1838a3743e4a}\\
RC_{43} &= \texttt{593023f0aefabd99d0fdf4c79a9369bd}\\
RC_{44} &= \texttt{329ae3d1eb606e6fa5cc637b6f1ecb2a}\\
RC_{45} &= \texttt{e00207eb49e01594a4dc93d6cb7594ab}\\
RC_{46} &= \texttt{1caa0c4ff751c880942366a665208ef8}\\
RC_{47} &= \texttt{02f7f57fdb2dc1ddbd03239fe3e67e4a}
\end{aligned}$$

For the general-purpose hash function H^*, we propose to use SHA-256, which is a NIST standard and widely available.

2.5.2.2 Pseudorandom Function

We propose a construction based on AES-256 for G, valid as long as the parameters verify the constraints $c + dh \leq 64$ and $\max(\ell, t) \leq 2^{31}$. More precisely, given a seed $s \in B_n$ and an address $a = (i, j, \lambda) \in \mathcal{A}$, we compute $G(s, a)$ as follows.

- Compute $P_0 \leftarrow \lceil j \rfloor_{64} || \lceil i \rfloor_{32} || \lceil 2\lambda \rfloor_{32}$ and $P_1 \leftarrow \lceil j \rfloor_{64} || \lceil i \rfloor_{32} || \lceil 2\lambda + 1 \rfloor_{32}$, where $\lceil x \rfloor_m$ denotes the (bytewise) big-endian encoding of x as an m-bit number.
- The result is AES-256$(s, P_0)||$AES-256(s, P_1), i.e. the encryption of P_0 and P_1 with key s.

We recall that due to the constraints on (c, d, h), the in-layer index j satisfies $0 \leq j < 2^{c+dh} \leq 2^{64}$, and the counter λ satisfies $2\lambda + 1 < 2^{32}$.

This construction is essentially AES-256 in counter mode, except that we need two AES blocks for a 256-bit result. Note that the seed s is the same throughout the hyper-tree, which allows a signer to cache the AES round keys.

3 Security

The security of Gravity-SPHINCS relies on the collision resistance of F, H, H^*, on the undetectability and one-wayness of F, and on the pseudo-randomness of G. Security reductions in [4, Ch.6] give lower bounds on the complexity of attacks. We now describe some concrete attack strategies and Gravity-SPHINCS's expected strength against them:

- **Find two messages that collide for** H^*, because their signatures would be identical. A generic birthday attack has a complexity of 128 bits for $n = 256$.
- **Break the non-adaptive subset-resilience of** G**.** Here again, our choices of parameters guarantee a complexity of at least 128 bits for known generic attacks, see [1] or [4, Ch.4].
- **Exploit a collision in WOTS or PORST instances**: if two secret values (in any of the WOTS and PORST instances) are identical, knowing one allows to forge another. With $n = 256$, this gives 128 bits of security if the secret values are chosen independently and uniformly. However, our construction with AES-256 guarantees that all secret values are distinct throughout the construction, because G is in fact a permutation.

These security estimates hold both against classical and quantum attacks.

This security level corresponds to NIST's category 2 (find a collision for a 256-bit hash).

4 Performance

We now describe optimization techniques to implement Gravity-SPHINCS efficiently.

4.1 Primitives

We can first apply optimizations specific to the chosen primitives.

4.1.1 Caching of AES Round Keys

To generate secret values with G, the same seed is used throughout the construction. With our implementation based on AES-256 in counter mode, this seed is the AES key. To avoid recomputing them for each block, we can cache the AES round keys throughout the scheme.

4.1.2 Haraka Pipelining

The Haraka hash function [7] was designed to support parallel computation on several inputs for CPUs supporting optimized instructions, e.g. Intel's Haswell and Skylake micro-architectures. Typically, a CPU core can evaluate Haraka on 4 to 8 inputs at the same time, depending on the micro-architecture. Hence, careful scheduling of hash evaluations is a way to improve the speed of Gravity-SPHINCS.

In particular, the WOTS construction uses many long chains of hashes. A naive implementation evaluates the chains one after another, which does not leverage 4-way (or 8-way) hashing. On the other end, a more clever implementation evaluates the chains level by level, using the pipelined versions of Haraka. Likewise, Merkle trees can be compressed level by level.

An even more efficient strategy is to fully compute the first 4 chains (using 4-way Haraka), then the next 4 chains, and so on. Indeed, this avoids expensive loads and stores between CPU registers and the rest of the memory. This is even more effective for mask-less constructions, because there is no need to load a mask from memory after each iteration. We improved the optimized Haraka implementation[1] to support computation of mask-less hash chains, removing useless store and load instructions at each iteration. This proved to be the most efficient strategy.

[1] Available at `https://github.com/kste/haraka`

4.1.3 AES-NI

Gravity-SPHINCS spends most of its time computing Haraka hashes, that is, AES rounds (plus some mixing operations). AES-NIs, now available in most mainstream processors, are mandatory to achieve tolerable speeds, for they make AES rounds orders of magnitude faster than with dedicated implementations.

On Skylake CPUs, the AES round instruction AESENC has a latency of four cycles and a reciprocal throughput of one. In Gravity-SPHINCS, many AES rounds can be pipelined in order to maximize the effective throughput and return one AESENC result per cycle: Haraka-512 computes up to four independent AES rounds, and rounds within four AES-256-CTR instances can be interleaved.

These independent AES round instances can't really be parallelized on current microarchitectures that have a single AES unit. But AMD's new Ryzen CPUs have two AES units, and Intel is expected to follow in future microarchitectures versions and include two AES units as well.

Our optimized implementation thus includes 4-way and 8-way interleaved versions of Haraka for computing the trees and its leaves[2], as well as 4-way interleaved AES-256-CTR.

Furthermore, future Intel microarchitectures (from Ice Lake) will include VAES (vector AES) instructions, which will compute four AES rounds simultaneously within a ZMM register. This will further speed-up Haraka-based hashing.

4.2 Secret Cache

As analyzed in [2], the top levels of the root Merkle tree can be cached by the signer, as they are shared among all signatures. In particular, given the threshold c, the 2^c hash values at level c can be cached with $2^c n$ bits of memory. Further, the levels above it total only $2^c - 1$ additional hash values, so a good strategy is to save all values from levels 0 to c, with $(2^{c+1} - 1)n$ bits of memory. For our sets of parameters proposed in Table 2.1, this represents 16 KiB to 2 MiB of secret cache.

4.3 Multithreading

To reduce signature size, the slower versions of Gravity-SPHINCS use larger Merkle trees, at the expense of key generation and signing times. To reduce the latency of these operations, we can leverage multithreading, especially in Merkle trees. Indeed, computing the root of a Merkle tree of height h can be distributed among 2^τ threads as follows: split the tree into 2^τ subtrees of height $h - \tau$ (starting from the leaves), compute each subtree in a different thread, and then

[2]Based on Haraka's authors code at `https://github.com/kste/haraka/`, with a few optimizations and bug fixes.

compute the top τ layers in a single thread. The latency of this computation is now in the order of $2^{h-\tau} + 2^{\tau}$, instead of 2^h.

This strategy is especially relevant in a batching context: instead of computing many independent signatures in parallel the signer computes a single signature, which means that many parallel threads of computation are available for one signature.

4.4 Cost Estimation

We estimate the cost of each operation (key generation, signing and verification) in terms of function calls. We let aside calls to the general-purpose hash function H^*, whose performance depends on the length of the message being signed.

Key Generation We compute the top Merkle tree:

- $2^{c+h}\ell$ calls to G to generate the WOTS secret values,
- $2^{c+h}\ell(w-1)$ calls to F to evaluate WOTS chains,
- $2^{c+h}-1$ calls to H to compress the Merkle tree.

The bottleneck is the evaluation of WOTS hash chains.

Signing Assuming that the top c levels of the hyper-tree are cached, we compute a PORST signature and d Merkle trees:

- 2 calls to H and a few calls to G to obtain the random subset of PORST,
- t calls to G to generate the PORST secret values,
- $t-1$ calls to H to compress the PORST tree,
- $d2^h\ell$ calls to G to generate the WOTS secret values,
- $\leq d2^h\ell(w-1)$ calls to F to evaluate partial WOTS chains,
- $d(2^h-1)$ calls to H to compress the d Merkle trees.

Here again the bottleneck is the evaluation of many WOTS hash chains.

Verification We verify a PORST instance and d Merkle trees:

- 1 call to H and a few calls to G to obtain the random subset of PORST,
- k calls to F to compute PORST public values,
- $\leq k(\log_2 t - \lfloor \log_2 k \rfloor)$ calls to H to compress octopus authentication nodes,
- $\leq d\ell(w-1)$ calls to F to evaluate partial WOTS chains,
- $c + dh$ calls to H to compress Merkle authentication paths.

The bottleneck is again the evaluation of WOTS hash chains, but verification is much faster than signing and key generation.

4.5 Benchmarks

We measured the execution time of the three operations—key generation, sign, verify—for each of the three proposed instances. It doesn't make much sense to count CPU cycles here since all operations are relatively slow, and signing/verification do a different number of operations depending on the message signed anyway. So we measure the wall time, reported here in microseconds on an Intel Core i5-6360U CPU @ 2.00 GHz.

The measurements below are directly copied from executions of our `bench` program, included in the source code published. We measured three rounds of sign–verify, to show the variability of the measured time.

Version S:

```
# crypto_sign_keypair
390823.00 usec

# crypto_sign
5982.00 usec

# crypto_sign_open
52.00 usec

# crypto_sign
4222.00 usec

# crypto_sign_open
32.00 usec

# crypto_sign
4118.00 usec

# crypto_sign_open
35.00 usec
```

Version M:

```
# crypto_sign_keypair
12114856.00 usec

# crypto_sign
9450.00 usec

# crypto_sign_open
126.00 usec

# crypto_sign
5920.00 usec

# crypto_sign_open
116.00 usec

# crypto_sign
6752.00 usec

# crypto_sign_open
115.00 usec
```

Version L:

```
# crypto_sign_keypair
5894540.00 usec

# crypto_sign
10527.00 usec

# crypto_sign_open
169.00 usec

# crypto_sign
6744.00 usec

# crypto_sign_open
175.00 usec

# crypto_sign
8087.00 usec

# crypto_sign_open
162.00 usec
```

As expected from the theoretical estimates in §4.4, key generation is super slow, while signing is a few milliseconds and verification is sub-millisecond. A very rough estimate of the CPU cycles count is obtained by multiplying the microsecond figures by 2000, since 2000 cycles are elapsed within a microsecond, at the CPU's nominal frequency (but note that we didn't disable Turbo Boost).

When CPUs include two AES units instead of one, Gravity-SPHINCS will be at most twice as fast, since most of the time is spent computing AES rounds.

Bibliography

[1] Jean-Philippe Aumasson and Guillaume Endignoux. Clarifying the subset resilience problem. Cryptology ePrint Archive, Report 2017/909, 2017.

[2] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. Cryptology ePrint Archive, Report 2017/933, 2017.

[3] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *EUROCRYPT*, 2015.

[4] Guillaume Endignoux. Design and implementation of a post-quantum hash-based cryptographic signature scheme. Master's thesis, EPFL, 2017.

[5] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004.

[6] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In *AFRICACRYPT*, 2013.

[7] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 - efficient short-input hashing for post-quantum applications. Cryptology ePrint Archive, Report 2016/098, 2016.

[8] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[9] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *ACISP*, 2002.