SPHINCS$^+$

Submission to the NIST post-quantum project, v.3

Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens,
Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag,
Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange,
Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger,
Joost Rijneveld, Peter Schwabe, Bas Westerbaan

October 1, 2020

Contents

1. Introduction

Hash-based signature schemes were developed as one-time signature schemes in the late 1970s by Lamport [14] and extended to many-times signatures by Merkle [16]. The security of these schemes is easy to analyze and relies solely on the properties of the used hash function. However, Merkle's tree-based signature scheme required fixing at key-generation time the number of signatures to be made, keeping this number small for performance. Most importantly, the system required users to remember a state: some information to remember how many signatures were already made with the key.

In the 40 years since Lamport's scheme, many ideas improved the performance, practicality, and theoretical foundations of hash-based signatures, culminating in XMSS [7], which is by now – as the first post-quantum signature scheme – published as an RFC [8] by the CFRG. A strong point of these systems is that they need very few security assumptions – the hash function even need not be collision resistant. The only downside of XMSS is that it is stateful, which makes it not fit the standard definition of signature schemes as, e.g., stated in the NIST call for submissions.

SPHINCS [5] was designed by Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, and Wilcox-O'Hearn as a *stateless* hash-based signature scheme and was the first signature scheme to propose parameters to resist quantum cryptanalysis. SPHINCS uses many components from XMSS but works with larger keys and signatures to eliminate the state.

This document is about the SPHINCS$^+$ construction. At a high level, SPHINCS$^+$ works like SPHINCS. The basic idea is to authenticate a huge number of few-time signature (FTS) key pairs using a so-called hypertree. FTS schemes are signature schemes that allow a key pair to produce a small number of signatures, e.g., in the order of ten for our parameter sets.

For each new message, a (pseudo)random FTS key pair is chosen to sign the message. The signature consists then of the FTS signature and the authentication information for that FTS key pair. The authentication information is roughly a hypertree signature, i.e. a signature using a certification tree of Merkle tree signatures.

More specifically, a hypertree is a tree of hash-based many-time signatures (MTS). These many-time signatures allow a key pair to sign a fixed number N of messages – for SPHINCS$^+$ N is a power of 2, for example 256. The MTS key pairs themselves are organized in an N-ary tree with d layers. On the top layer $d-1$ there is a single MTS key pair which is used to sign the public keys of N MTS key pairs that form layer $d-2$. Each of these N MTS key pairs is used to sign another N MTS public keys forming layer $d-3$. This goes on down to the N^{d-1} key pairs on the bottom layer which are used to sign N FTS public keys, each, leading to a total number of N^d authenticated FTS key pairs. The authentication information for an FTS key pair consists of the d MTS signatures that build a path from the FTS key pair to the top MTS tree.

An MTS signature is just a classical Merkle-tree signature in the case of SPHINCS$^+$. It consists of a one-time signature (OTS) on the given message plus the authentication path in the binary hash-tree, authenticating the N OTS key pairs of one MTS key pair.

The public key of SPHINCS$^+$ is essentially the public key of the top level MTS which is just the root node of its binary hash tree and hence, a single hash value. However, actual SPHINCS$^+$ public keys additionally contain a public seed value of the same length as the root node. This is due to technical reasons explained in the detailed specification below.

The SPHINCS$^+$ secret key is just a single secret seed value. From this, all the OTS and

FTS secret keys are generated in a pseudorandom manner. The OTS and FTS secret keys together fully determine the whole virtual structure of an SPHINCS$^+$ key pair. Again, actual SPHINCS$^+$ secret keys contain an additional secret value of the same size as the secret seed as well as a copy of the public key. The additional value is used to key a PRF used in randomized hashing as detailed in the comparison below.

1.1. SPHINCS$^+$ vs SPHINCS

SPHINCS$^+$ builds on SPHINCS by introducing several improvements:

- Multi-target attack protection: We apply the mitigation techniques from [11] using keyed hash functions. Each hash function call is keyed with a different key and applies different bitmasks. Keys and bitmasks are pseudorandomly generated from an address specifying the context of the call, and a public seed. For this we introduce the notion of tweakable hash functions which in addition to the input value take a public seed and an address.

- Tree-less WOTS$^+$ public key compression: The last nodes of the WOTS$^+$ chains are not compressed using an L-tree but using a single tweakable hash function call. This call again receives an address and a public seed to key this call and to generate a bitmask as long as the input.

- FORS: A HORST key pair does not consist anymore of a single monolithic tree. Instead it consists of k trees of height a The leaves of these trees are the hashes of the 2^a secret key elements. The public key is the tweakable hash of the concatenation of all the root nodes as for the WOTS$^+$ public key.

  A FORS key pair can be used to sign $k2^a$bit message digests. The digest is first split into k strings m_i of length 2^a bits each. Next, every m_i is interpreted as an integer in $[0, 2^a - 1]$. Here m_i selects the m_i-th secret key element of the i-th tree for the signature. The signature also contains the authentication paths for all the selected secret key elements, which means one path of length a per tree. Verification uses the signature to reconstruct the root nodes and compresses them using the tweakable hash.

- Verifiable index selection: The message digest is now computed as follows. First, we deterministically generate randomness

  $$\mathbf{R} = \mathbf{PRF_{msg}}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M).$$

  Where `OptRand` is a 256 bit value, per default 0 but can be filled with random bits e.g. taken from a TRNG to avoid deterministic signing (this might be desirable to counter side channel attacks). Then we compute message digest and index as

  $$(\text{md}||\text{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}, M)$$

  where $\mathbf{PK} = (\mathbf{PK}.\texttt{seed}, \mathbf{PK}.\texttt{root})$ contains the top root node and the public seed. Hence, we can omit the index in the SPHINCS signature as it would be redundant. This allows to tighten HORST security.

- SPHINCS$^+$-'robust' and SPHINCS$^+$-'simple': The updated, Round 2 submission of SPHINCS$^+$ adds new, more simple instantiations of the tweakable hash functions similar to those in the LMS proposal for stateful hash-based signatures [15]. This splits the

instantiations into the new 'simple' instantiations and the established 'robust' instantiations. The 'simple' instantiations have the advantage of better speed and the drawback of a security argument which in its entirety only applies in the random oracle model. Consequently, the 'robust' instantiations have a more conservative security argument but are slower.

1.2. Organization

In this document we give a formal specification of the SPHINCS$^+$ construction. We follow a bottom-up approach to specify SPHINCS$^+$. We start with basic notation. Afterwards we define WOTS$^+$, the OTS used in SPHINCS$^+$. Next, we specify XMSS, the MTS used in SPHINCS$^+$, and how it is used to do HT signatures. Then, we define FORS, the FTS used, to finally specify SPHINCS$^+$. Afterwards we discuss different instantiations and explain the design rationale. Then we present a security analysis, give performance values and conclude with a discussion of advantages and limitations.

2. Notation

In the following we start defining basic mathematical operations on integers and bit strings. From that we work our way to more specific basic methods used later in the specification.

2.1. Data Types

Bytes and byte strings are the fundamental data types. A byte is a sequence of eight bits. The set of bytes is denoted as $\mathbb{B}$. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string of length 3. An array of byte strings is an ordered, indexed set starting with index 0 in which all byte strings have identical length. We assume big-endian representation for any data types or structures.

2.2. Functions

We define the following functions:

$\lceil x \rceil$(or ceil(x)) : for x a real number returns the smallest integer greater than or equal to x.

$\lfloor x \rfloor$(or floor(x)) : for x a real number returns the largest integer less than or equal to x.

$\log(x)$: for x a non-negative real number returns the logarithm to base 2 of x. In pseudocode this function is written as `lg`.

$\text{Trunc}_\ell(x)$: truncates the bit-string x to the first ℓ bits.

2.3. Operators

When a and b are integers, mathematical operators are defined as follows:

$\hat{}$: a^b denotes the result of a raised to the power of b.

$\cdot$: $a \cdot b$ denotes the product of a and b. This operator is sometimes omitted in the absence of ambiguity, as in usual mathematical notation.

$/$: a/b denotes the quotient of a by non-zero b.

$\%$: $a \% b$ denotes the non-negative remainder of the integer division of a by b.

$+$: $a + b$ denotes the sum of a and b.

$-$: $a - b$ denotes the difference of a and b.

$++$: $a++$ denotes incrementing a by 1, i.e., $a = a + 1$.

$<<$: $a << b$ denotes a logical left shift of a by b positions, for b being non-negative, i.e., $a \cdot 2^b$.

$>>$: $a >> b$ denotes a logical right shift of a by b positions, for b being non-negative, i.e. $\text{floor}(a/2^b)$.

The standard order of operations is used when evaluating arithmetic expressions.

Arrays are used in the common way, where the i-th element of an array A is denoted $A[i]$. Byte strings are treated as arrays of bytes where necessary: If X is a byte string, then $X[i]$ denotes its i-th byte, where $X[0]$ is the leftmost, highest order byte.

If A and B are byte strings of equal length, then:

A AND B denotes the bitwise logical conjunction operation.

A XOR B (or $A \oplus B$) denotes the bitwise logical exclusive disjunction operation.

When B is a byte and i is an integer, then $B >> i$ denotes the logical right-shift by i positions.

If X is an x-byte string and Y a y-byte string, then $X||Y$ denotes the concatenation of X and Y, with $X||Y = X[0] \ldots X[x-1]Y[0] \ldots Y[y-1]$.

2.4. Integer to Byte Conversion (Function `toByte`)

For x and y non-negative integers, we define $Z = \texttt{toByte}(x, y)$ to be the y-byte string containing the binary representation of x in big-endian byte-order.

2.5. Strings of Base-w Numbers (Function `base_w`)

A byte string can be considered as a string of base w numbers, i.e. integers in the set $\{0, \ldots, w-1\}$. The correspondence is defined by the function $\texttt{base\_w}(X, w, \texttt{out\_len})$ as follows. Let X be a `len_X`- byte string, and w is an element of the set $\{4, 16, 256\}$, then $\texttt{base\_w}(X, w, \texttt{out\_len})$ outputs an array of `out_len` integers between 0 and $w - 1$ (Figure 1). The length `out_len` is REQUIRED to be less than or equal to $8 * \texttt{len\_X} / \log(w)$.

```
# Input: len_X-byte string X, int w, output length out_len
# Output: out_len int array basew

base_w(X, w, out_len) {
```

```
    int in = 0;
    int out = 0;
    unsigned int total = 0;
    int bits = 0;
    int consumed;

    for ( consumed = 0; consumed < out_len; consumed++ ) {
        if ( bits == 0 ) {
            total = X[in];
            in++;
            bits += 8;
        }
        bits -= lg(w);
        basew[out] = (total >> bits) AND (w - 1);
        out++;
    }
    return basew;
}
```

Algorithm 1: `base_w` – Computing the base-w representation

2.6. Member Functions (Functions `set`, `get`)

To simplify algorithm descriptions, we assume the existence of member functions. If a complex data structure like a public key PK contains a variable X then `PK.getX()` returns the value of X for this public key. Accordingly, `PK.setX(Y)` sets variable X in PK to the value held by Y. Since camelCase is used for member function names, a value z may be referred to as Z in the function name, e.g. `getZ`.

2.7. Cryptographic (Hash) Function Families

SPHINCS$^+$ makes use of several different function families with cryptographic properties. Every SPHINCS$^+$ instantiation MUST describe how to implement each of the following functions. For the main instantiations given in this document, this will be done using a single (hash) function, i.e., SHA2-256 or SHAKE-128. Specific instantiations are given in Section 7.

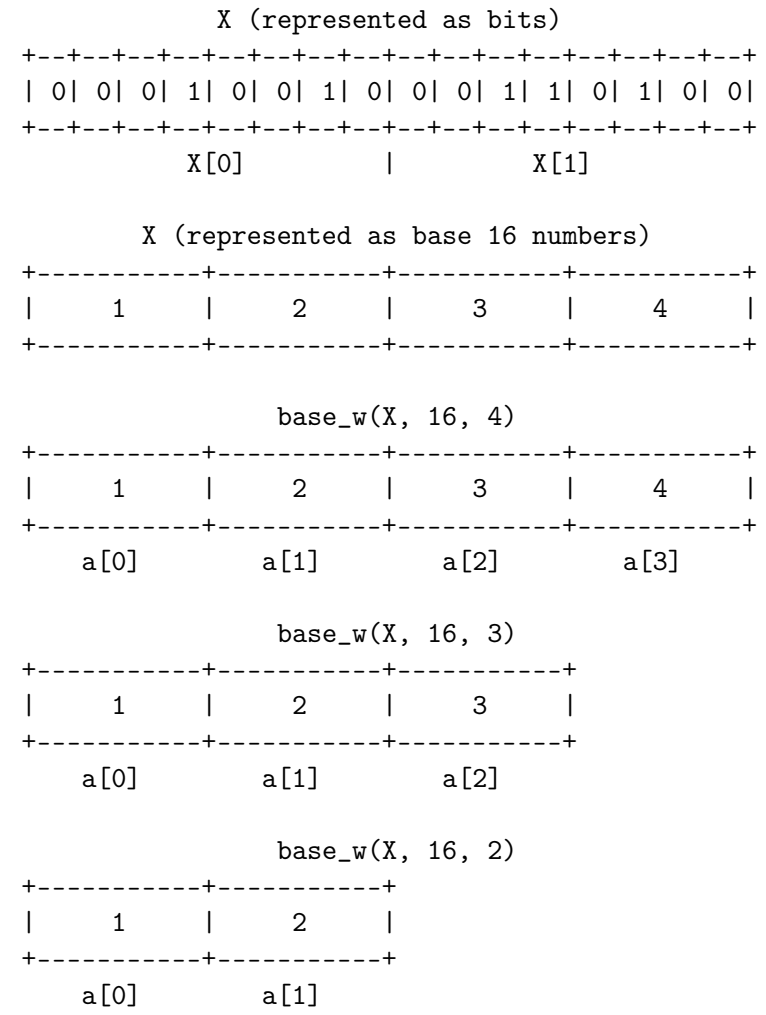
SPHINCS$^+$ applies the multi-target mitigation technique from [11], independently keying and randomizing each hash function call in the original SPHINCS. The implementation of this randomization and keying differs for different instantiations as different function families (e.g., SHA2 or SHAKE) have different properties. Hence, we introduce *tweakable* hash functions as a layer of abstraction. All algorithms in this specification use *tweakable* hash functions in place of traditional hash functions. Later, in Section 7, we describe how to implement the tweakable hash functions.

In addition to several tweakable hash functions, SPHINCS$^+$ makes use of two PRFs and a keyed hash function. Input and output length are given in terms of the security parameter n and the message digest length m, both to be defined more precisely in the coming sections.

2.7.1. Tweakable Hash Functions (Functions `T_l`, `F`, `H`)

A *tweakable* hash function takes a public seed **PK**.`seed` and context information in form of an address **ADRS** in addition to the message input. This allows to make the hash function

Figure 1: For example, if X is the (big-endian) byte string 0x1234, then `base_w`$(X, 16, 4)$ returns the array $a = \{1, 2, 3, 4\}$.



calls for each key pair and position in the virtual tree structure of SPHINCS$^+$ independent from each other. The addressing scheme will be described in Section 2.7.3.

The schemes described in this specification build upon several instantiations of tweakable hash functions of the form

$$\mathbf{T}_\ell : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n} \to \mathbb{B}^n,$$
$$\text{md} \leftarrow \mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M)$$

mapping and ℓn-byte message M to an n-byte hash value md using an n-byte seed $\mathbf{PK}.\texttt{seed}$ and a 32-byte address $\mathbf{ADRS}$. The function $\mathbf{T}_\ell$ is denoted by `T_l` in pseudocode.

There are two special cases which we rename for consistency with previous descriptions of

hash-based signature schemes:

$$\mathbf{F} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n \to \mathbb{B}^n,$$
$$\mathbf{F} \stackrel{\text{def}}{=} \mathbf{T}_1$$
$$\mathbf{H} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n} \to \mathbb{B}^n$$
$$\mathbf{H} \stackrel{\text{def}}{=} \mathbf{T}_2$$

2.7.2. PRF and Message Digest (Functions `PRF`, `PRF_msg`, `H_msg`)

SPHINCS$^+$ makes use of a pseudorandom function **PRF** for pseudorandom key generation:

$$\mathbf{PRF} : \mathbb{B}^n \times \mathbb{B}^{32} \to \mathbb{B}^n.$$

In addition, SPHINCS$^+$ uses a pseudorandom function $\mathbf{PRF_{msg}}$ to generate randomness for the message compression:

$$\mathbf{PRF_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^n.$$

To compress the message to be signed, SPHINCS$^+$ uses an additional keyed hash function $\mathbf{H_{msg}}$ that can process arbitrary length messages:

$$\mathbf{H_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^m.$$

2.7.3. Hash Function Address Scheme (Structure of ADRS)

An address **ADRS** is a 32-byte value that follows a defined structure. In addition, it comes with `set` methods to manipulate the address. We explain the generation of addresses in the following sections where they are used. Essentially, all functions have to keep track of the current context, updating the addresses after each hash call.

There are five different types of addresses for the different use cases. One type is used for the hashes in WOTS$^+$ schemes, one is used for compression of the WOTS$^+$ public key, the third is used for hashes within the main Merkle tree construction, another is used for the hashes in the Merkle tree in FORS, and the last is used for the compression of the tree roots of FORS. These types largely share a common format. We describe them in more detail, below.

The structure of an address complies with word borders, with a word being 32 bits long in this context. Only the tree address (i.e. the index of a specific subtree in the main tree) is too long to fit a single word: for this, we reserve three words. An address is structured as follows. It always starts with a layer address of one word in the most significant bits, followed by a tree address of three words. These addresses describe the position of a tree within the hypertree. The layer address describes the height of a tree within the hypertree starting from height zero for trees on the bottom layer. The tree address describes the position of a tree within a layer of a multi-tree starting with index zero for the leftmost tree. The next word defines the type of the address. It is set to 0 for a WOTS$^+$ hash address, to 1 for the compression of the WOTS$^+$ public key, to 2 for a hash tree address, to 3 for a FORS address, and to 4 for the compression of FORS tree roots.

We first describe the WOTS$^+$ address (Figure 2). In this case, the type word is followed by the key pair address that encodes the index of the WOTS$^+$ key pair within the specified

tree. The next word encodes the chain address (i.e. the index of the chain within WOTS$^+$), followed by a word that encodes the address of the hash function call within the chain. Note that the address of the bottom of the chain is also used to generate the secret keys based on **SK**.seed.

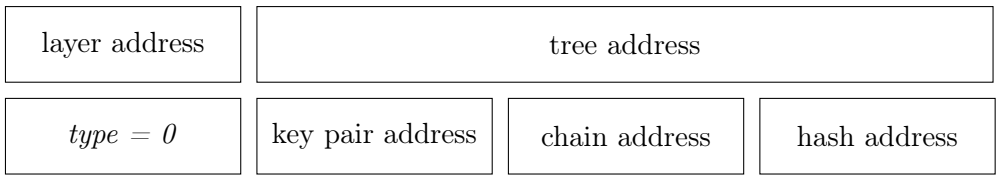


Figure 2: WOTS$^+$ hash address.

The second type (Figure 3) is used to compress the WOTS$^+$ public keys. The type word is set to 1. Similar to the address used within WOTS$^+$, the next word encodes the key pair address. The remaining two words are not needed, and thus remain zero. We zero pad the address to the constant length of 32 bytes.

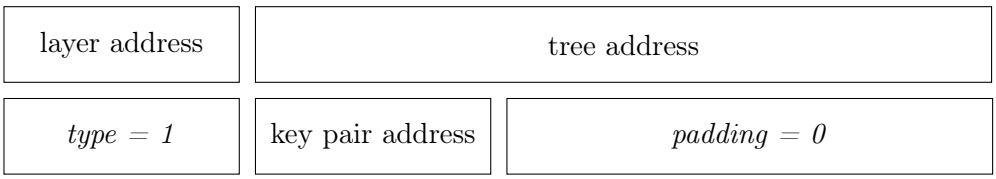


Figure 3: WOTS$^+$ public key compression address.

The third type (Figure 4) addresses the hash functions in the main tree. In this case the type word is set to 2, followed by a zero padding of one word. The next word encodes the height of the tree node that is being computed, followed by a word that encodes the index of this node at that height.

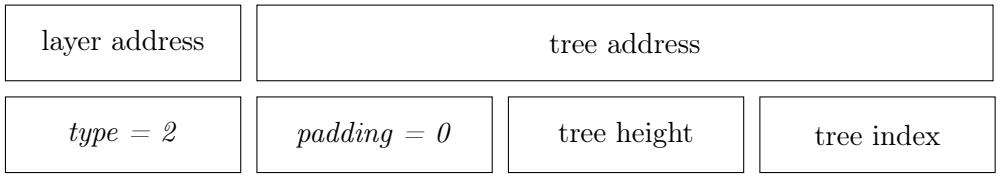


Figure 4: hash tree address.

The next type (Figure 5) is of a similar format, and is used to describe the hash functions in the FORS tree. The type word is set to 3. The key pair address is used to signify which FORS key pair is used, identical to the key pair address in the WOTS$^+$ hash addresses. Its value is the same as that of the WOTS$^+$ key pair that is used to authenticate it, i.e. its index as a leaf in the specified tree. The tree height and tree index fields are used to address the hashes within the FORS tree. This is done like for the above-mentioned hashes in the main tree, with the additional consideration that the tree indices are counted continuously across the different FORS trees. The addresses at tree height 0 are used to generate the leaf nodes from **SK**.seed.

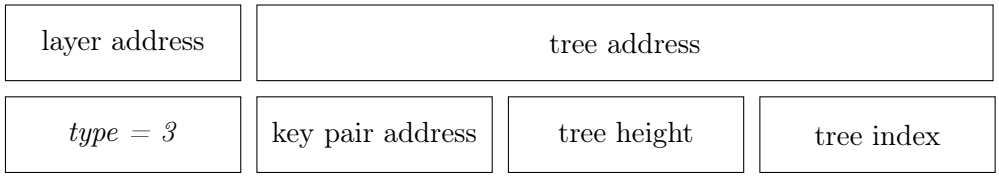


Figure 5: FORS tree address.

The final type (Figure 6) is used to compress the tree roots of the FORS trees. The type word is set to 4. Like the WOTS$^+$ public key compression address, it contains only the address of the FORS key pair, but is padded to the full length.

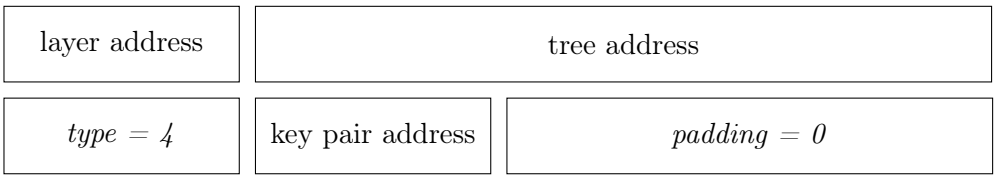


Figure 6: FORS tree roots compression address.

All fields within these addresses encode unsigned integers. When describing the generation of addresses we use `set` methods that take positive integers and set the bits of a field to the binary representation of that integer, in big-endian notation. Throughout this document, we adhere to the convention of assuming that changing the type word of an address (indicated by the use of the `setType()` method) initializes the subsequent three words to zero.

In order to make keeping track of the types easier throughout the pseudo-code in the rest of this document, we refer to them respectively using the constants `WOTS_HASH`, `WOTS_PK`, `TREE`, `FORS_TREE` and `FORS_ROOTS`.

3. WOTS$^+$ One-Time Signatures

This section describes the WOTS$^+$ scheme, in a version similar to [9]. WOTS$^+$ is a OTS scheme; while a private key can be used to sign any message, each private key MUST NOT be used to sign more than a single message. In particular, if a private key is used to sign two different messages, the scheme becomes insecure.

The description given here is tailored to the use inside of SPHINCS$^+$. It assumes that the scheme is used as a subroutine inside a higher order scheme and is not sufficient for a standalone implementation of WOTS$^+$. The section starts with an explanation of parameters. Afterwards, the so-called chaining function, which forms the main building block of the WOTS$^+$ scheme, is explained. A description of the algorithms for key generation and signing follows. Finally, we give an algorithm to compute a WOTS$^+$ public key from a WOTS$^+$ signature. This will be used as a subroutine in SPHINCS$^+$ signature verification.

3.1. WOTS$^+$ Parameters

WOTS$^+$ uses the parameters n and w; they both take positive integer values. These parameters are summarized as follows:

- n: the security parameter; it is the message length as well as the length of a private key, public key, or signature element in bytes.
- w: the Winternitz parameter; it is an element of the set $\{4, 16, 256\}$.

These parameters are used to compute values `len`, $\texttt{len}_1$ and $\texttt{len}_2$:

- `len`: the number of n-byte-string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{8n}{\log(w)} \right\rceil, \quad \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

The security parameter n is the same as the security parameter n for SPHINCS$^+$. The value of n determines the in- and output length of the tweakable hash function used for WOTS$^+$. The value of n also determines the length of messages that can be processed by the WOTS$^+$ signing algorithm. The parameter w can be chosen from the set $\{4, 16, 256\}$. A larger value of w results in shorter signatures but slower operations; it has no effect on security. Choices of w are limited to the values 4, 16, and 256 since these values yield optimal trade-offs and easy implementation. WOTS$^+$ parameters are implicitly included in algorithm inputs as needed.

3.2. WOTS$^+$ Chaining Function (Function `chain`)

The chaining function (Algorithm 2) computes an iteration of **F** on an n-byte input using a WOTS$^+$ hash address **ADRS** and a public seed **PK**.`seed`. The address **ADRS** MUST have the first seven 32-bit words set to encode the address of this chain. In each iteration, the address is updated to encode the current position in the chain before **ADRS** is used to process the input by **F**.

In the following, **ADRS** is a 32-byte WOTS$^+$ hash address as specified in Section 2.7.3 and **PK**.`seed` is a n-byte string. The chaining function takes as input an n-byte string X;, a start index i, a number of steps s, as well as **ADRS** and **PK**.`seed`. The chaining function returns as output the value obtained by iterating **F** for s times on input X.

```
#Input: Input string X, start index i, number of steps s, public seed PK.seed,
    address ADRS
#Output: value of F iterated s times on X

chain(X, i, s, PK.seed, ADRS) {
  if ( s == 0 ) {
    return X;
  }
  if ( (i + s) > (w - 1) ) {
    return NULL;
  }
  byte[n] tmp = chain(X, i, s - 1, PK.seed, ADRS);

  ADRS.setHashAddress(i + s - 1);
  tmp = F(PK.seed, ADRS, tmp);
  return tmp;
}
```

Algorithm 2: `chain` – Chaining function used in WOTS$^+$.

3.3. WOTS$^+$ Private Key (Function `wots_SKgen`)

The WOTS$^+$ private key, denoted by sk (s for secret), is a length `len` array of n-byte strings. This private key MUST NOT be used to sign more than one message. This private key is only implicitly used. Therefore, the following is just to support a better understanding of the following algorithms. Each n-byte string in the WOTS$^+$ private key is derived from a secret seed **SK**.`seed` which is part of the SPHINCS$^+$ secret key and a WOTS$^+$ address **ADRS** using **PRF**. The same secret seed is used to generate all secret key values within SPHINCS$^+$. The address used to generate the i-th n-byte string of sk MUST encode the position of the i-th hash chain of this WOTS$^+$ instance within the SPHINCS$^+$ structure.

The following pseudocode (Algorithm 3) describes an algorithm to generate a WOTS$^+$ private key.

```
#Input: secret seed SK.seed, address ADRS
#Output: WOTS+ private key sk

wots_SKgen(SK.seed, ADRS) {
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk[i] = PRF(SK.seed, ADRS);
  }
  return sk;
}
```

Algorithm 3: `wots_SKgen` – Generating a WOTS$^+$ private key.

3.4. WOTS$^+$ Public Key Generation (Function `wots_PKgen`)

A WOTS$^+$ key pair defines a virtual structure that consists of len hash chains of length w. Each of the len stings of n-bytes in the private key defines the start node for one hash chain. The public key is the tweakable hash of the end nodes of these hash chains. To compute the hash chains, the chaining function (Algorithm 2) is used. A WOTS$^+$ hash address **ADRS** and a seed **PK**.`seed` have to be provided by the calling algorithm as well as a secret seed **SK**.`seed`. The address **ADRS** MUST encode the address of the WOTS$^+$ key pair within the SPHINCS$^+$ structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.`seed` used here is public information also available to a verifier. The following pseudocode (Algorithm 4) describes an algorithm for generating the public key pk.

```
#Input: secret seed SK.seed, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk

wots_PKgen(SK.seed, PK.seed, ADRS) {
  wotspkADRS = ADRS; // copy address to create OTS public key address
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk = PRF(SK.seed, ADRS);
    tmp[i] = chain(sk[i], 0, w - 1, PK.seed, ADRS);
  }
  wotspkADRS.setType(WOTS_PK);
  wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_len(PK.seed, wotspkADRS, tmp);
```

```
  return pk;
}
```

Algorithm 4: `wots_PKgen` – Generating a WOTS$^+$ public key.

3.5. WOTS$^+$ Signature Generation (Function `wots_sign`)

A WOTS$^+$ signature is a length `len` array of n-byte strings. The WOTS$^+$ signature is generated by mapping a message M to `len` integers between 0 and $w-1$. To this end, the message is transformed into $\texttt{len}_1$ base-w numbers using the `base_w` function defined in Section 2.5. Next, a checksum over M is computed and appended to the transformed message as $\texttt{len}_2$ base-w numbers using the `base_w` function. Note that the checksum may reach a maximum integer value of $\texttt{len}_1 \cdot (w-1)$ and therefore depends on the parameters n and w. For the parameter sets given in Section 7, a 32-bit unsigned integer is sufficient to hold the checksum. If other parameter sets are used, the size of the variable holding the integer value of the checksum MUST be sufficiently large. Each of the base-w integers is used to select a node from a different hash chain. The signature is formed by concatenating the selected nodes. A WOTS$^+$ hash address **ADRS**, a public seed **PK**.`seed`, and a secret seed **SK**.`seed` have to be provided by the calling algorithm. The address will encode the address of the WOTS$^+$ key pair within a greater structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.`seed` used here is public information also available to a verifier while the secret seed **SK**.`seed` is private information. The pseudocode for generating a WOTS$^+$ signature **sig** is shown below (Algorithm 5).

```
#Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
#Output: WOTS+ signature sig

wots_sign(M, SK.seed, PK.seed, ADRS) {
  csum = 0;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  if( (lg(w) % 8) != 0) {
    csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  }
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);

  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk = PRF(SK.seed, ADRS);
    sig[i] = chain(sk, 0, msg[i], PK.seed, ADRS);
  }
  return sig;
```

```
}
```

Algorithm 5: `wots_sign` – Generating a WOTS+ signature on a message M.

The data format for a signature is given in Figure 7.

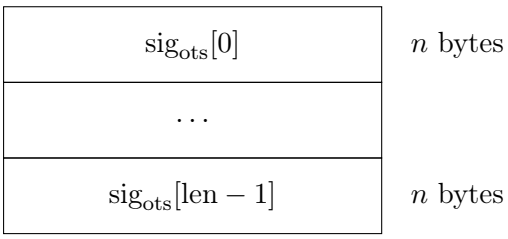


Figure 7: WOTS$^+$ Signature data format.

3.6. WOTS$^+$ Compute Public Key from Signature (Function `wots_pkFromSig`)

SPHINCS$^+$ uses implicit signature verification for WOTS$^+$. In order to verify a WOTS$^+$ signature **sig** on a message M, the verifier computes a WOTS$^+$ public key value from the signature. This can be done by "completing" the chain computations starting from the signature values, using the base-w values of the message hash and its checksum. This step, called `wots_pkFromSig`, is described below in Algorithm 6. The result of `wots_pkFromSig` then has to be verified. In a standalone version, this would be done by simple comparison. When used in SPHINCS$^+$ the output value is verified by using it to compute a SPHINCS$^+$ public key.

A WOTS$^+$ hash address **ADRS** and a public seed **PK**.`seed` have to be provided by the calling algorithm. The address will encode the address of the WOTS$^+$ key pair within the SPHINCS$^+$ structure. Hence, a WOTS$^+$ algorithm MUST NOT manipulate any parts of **ADRS** other than the last three 32-bit words. Note that the **PK**.`seed` used here is public information also available to a verifier.

```
#Input: Message M, WOTS+ signature sig, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk_sig derived from sig

wots_pkFromSig(sig, M, PK.seed, ADRS) {
  csum = 0;
  wotspkADRS = ADRS;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    tmp[i] = chain(sig[i], msg[i], w - 1 - msg[i], PK.seed, ADRS);
```

```
    }

    wotspkADRS.setType(WOTS_PK);
    wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
    pk_sig = T_len(PK.seed, wotspkADRS, tmp);
    return pk_sig;
  }
```

Algorithm 6: `wots_pkFromSig` – Computing a WOTS+ public key from a message and its signature.

4. The SPHINCS$^+$ Hypertree

In this section, we explain how the SPHINCS$^+$ hypertree is built. We first explain how WOTS$^+$ gets combined with a binary hash tree, leading to a fixed input-length version of the eXtended Merkle Signature Scheme (XMSS). Afterwards, we explain how to go to a hypertree from there. The hypertree might be viewed as a fixed input-length version of multi-tree XMSS (XMSS^{MT}).

4.1. (Fixed Input-Length) XMSS

XMSS is a method for signing a potentially large but fixed number of messages. It is based on the Merkle signature scheme. It authenticates $2^{h'}$ WOTS$^+$ public keys using a binary tree of height h'. Hence, an XMSS key pair for height h' can be used to sign $2^{h'}$ different messages. Each node in the binary tree is an n-byte value which is the tweakable hash of the concatenation of its two child nodes. The leaves are the WOTS$^+$ public keys. The XMSS public key is the root node of the tree. In SPHINCS$^+$, the XMSS secret key is the single secret seed that is used to generate all WOTS$^+$ secret keys.

An XMSS signature in the context of SPHINCS$^+$ consists of the WOTS$^+$ signature on the message and the so-called authentication path. The latter is a vector of tree nodes that allow a verifier to compute a value for the root of the tree starting from a WOTS$^+$ signature. A verifier computes the root value and verifies its correctness. A standalone XMSS signature also contains the index of the used WOTS$^+$ key pair. In the context of SPHINCS$^+$ this is not necessary as the SPHINCS$^+$ signature allows to compute the index for each XMSS signature contained.

4.1.1. XMSS Parameters

XMSS has the following parameters:

h' : the height (number of levels - 1) of the tree.

n : the length in bytes of messages as well as of each node.

w : the Winternitz parameter as defined for WOTS$^+$ in the previous Section.

There are $2^{h'}$ leaves in the tree. XMSS signatures are denoted by $\mathbf{SIG}_{\text{XMSS}}$ (`SIG_XMSS` in pseudocode). WOTS$^+$ signatures are denoted by sig.

XMSS parameters are implicitly included in algorithm inputs as needed.

4.1.2. XMSS Private Key

In the context of SPHINCS$^+$, an XMSS private key is the single secret seed **SK**.seed contained in the SPHINCS$^+$ secret key. It is used to generate the WOTS$^+$ secret keys within the structure of an XMSS key pair as described in Section 3.

4.1.3. TreeHash (Function treehash)

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine treehash (Algorithm 7) accepts a secret seed **SK**.seed, a public seed **PK**.seed, an unsigned integer s (the start index), an unsigned integer z (the target node height), and an address **ADRS** that encodes the address of the containing tree. For the height of a node within a tree, counting starts with the leaves at height zero. The treehash algorithm returns the root node of a tree of height z with the leftmost leaf being the WOTS$^+$ pk at index s. It is REQUIRED that $s \% 2^z = 0$, i.e. that the leaf at index s is a leftmost leaf of a sub-tree of height z. Otherwise the algorithm fails as it would compute non-existent nodes. The treehash algorithm described here uses a stack holding up to $(z-1)$ nodes, with the usual stack functions push() and pop(). We furthermore assume that the height of a node (an unsigned integer) is stored alongside a node's value (an n-byte string) on the stack.

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack

treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       ADRS.setType(WOTS_HASH);
       ADRS.setKeyPairAddress(s + i);
       node = wots_PKgen(SK.seed, PK.seed, ADRS);
       ADRS.setType(TREE);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

Algorithm 7: treehash – The TreeHash algorithm.

4.1.4. XMSS Public Key Generation (Function xmss_PKgen)

The XMSS public key is computed as described in xmss_PKgen (Algorithm 10). In the context of SPHINCS$^+$ the XMSS public key PK is the root of the binary hash tree. The root is computed using treehash. The public key generation takes a secret seed **SK**.seed, a public seed **PK**.seed, and an address **ADRS**. The latter encodes the position of this XMSS instance within the SPHINCS$^+$ structure.

```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: XMSS public key PK

xmss_PKgen(SK.seed, PK.seed, ADRS) {
     pk = treehash(SK.seed, 0, h', PK.seed, ADRS)
     return pk;
}
```

Algorithm 8: `xmss_PKgen` – Generating an XMSS public key.

4.1.5. XMSS Signature

An XMSS signature is a $((\texttt{len} + h') * n)$-byte string consisting of

- a WOTS$^+$ signature **sig** taking $\texttt{len} \cdot n$ bytes,
- the authentication path **AUTH** for the leaf associated with the used WOTS$^+$ key pair taking $h' \cdot n$ bytes.

The authentication path is an array of h' n-byte strings. It contains the siblings of the nodes in on the path from the used leaf to the root. It does not contain the nodes on the path itself. These nodes in **AUTH** are needed by a verifier to compute a root node for the tree from a WOTS$^+$ public key. A node N is addressed by its position in the tree. $N(x, y)$ denotes the yth node on level x with $y = 0$ being the leftmost node on a level. The leaves are on level 0, the root is on level h'. An authentication path contains exactly one node on every layer $0 \leq x \leq (h' - 1)$. For the ith WOTS$^+$ key pair, counting from zero, the jth authentication path node is

$$\textbf{AUTH}[j] = N\left(j, \lfloor \frac{i}{2^j} \rfloor \oplus 1\right)$$

The computation of the authentication path is discussed in Section 4.1.6.
The data format for a signature is given in Figure 8.

sig ($\texttt{len} \cdot n$ bytes)
AUTH [0] (n bytes)
...
AUTH [h-1] (n bytes)

Figure 8: XMSS Signature

4.1.6. XMSS Signature Generation (Function `xmss_sign`)

To compute the XMSS signature of a message M in the context of SPHINCS$^+$, the secret seed **SK**.seed, the public seed **PK**.seed, the index idx of the WOTS$^+$ key pair to be used, and the address **ADRS** of the XMSS instance are needed. First, a WOTS$^+$ signature of the message digest is computed using the WOTS$^+$ instance at index idx. Next, the authentication path is computed.

The node values of the authentication path MAY be computed in any way. The least memory-intensive method is to compute all nodes using the `treehash` algorithm (Algorithm 7). This is described here. Note that the details of how this step is implemented are not relevant to interoperability; it is not necessary to know any of these details in order to perform the signature verification operation.

```
# Input: n-byte message M, secret seed SK.seed, index idx, public seed PK.seed,
    address ADRS
# Output: XMSS signature SIG_XMSS = (sig || AUTH)

xmss_sign(M, SK.seed, idx, PK.seed, ADRS)
     // build authentication path
     for ( j = 0; j < h'; j++ ) {
       k = floor(idx / (2^j)) XOR 1;
       AUTH[j] = treehash(SK.seed, k * 2^j, j, PK.seed, ADRS);
     }

     ADRS.setType(WOTS_HASH);
     ADRS.setKeyPairAddress(idx);
     sig = wots_sign(M, SK.seed, PK.seed, ADRS);
     SIG_XMSS = sig || AUTH;
     return SIG_XMSS;
}
```

Algorithm 9: `xmss_sign` – Generating an XMSS signature.

4.1.7. XMSS Compute Public Key from Signature (Function `xmss_pkFromSig`)

SPHINCS$^+$ makes use of implicit signature verification of XMSS signatures. An XMSS signature is used to compute a candidate XMSS public key, i.e., the root of the tree. This is used in further computations (signature of the tree above) and implicitly verified by the outcome of that computation. Hence, this specification does not contain an `xmss_verify` method but the method `xmss_pkFromSig`.

The method `xmss_pkFromSig` takes an n-byte message M, an XMSS signature $\mathbf{SIG}_{\text{XMSS}}$, a signature index idx, a public seed **PK**.seed, and an address **ADRS**. The latter encodes the position of the current XMSS instance within the virtual structure of the SPHINCS$^+$ key pair. First, `wots_pkFromSig` is used to compute a candidate WOTS$^+$ public key. This in turn is used together with the authentication path to compute a root node which is then returned. The algorithm `xmss_pkFromSig` is given as Algorithm 10.

```
# Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte message M,
    public seed PK.seed, address ADRS
# Output: n-byte root value node[0]

xmss_pkFromSig(idx, SIG_XMSS, M, PK.seed, ADRS){
```

```
        // compute WOTS+ pk from WOTS+ sig
        ADRS.setType(WOTS_HASH);
        ADRS.setKeyPairAddress(idx);
        sig = SIG_XMSS.getWOTSSig();
        AUTH = SIG_XMSS.getXMSSAUTH();
        node[0] = wots_pkFromSig(sig, M, PK.seed, ADRS);

        // compute root from WOTS+ pk and AUTH
        ADRS.setType(TREE);
        ADRS.setTreeIndex(idx);
        for ( k = 0; k < h'; k++ ) {
          ADRS.setTreeHeight(k+1);
          if ( (floor(idx / (2^k)) % 2) == 0 ) {
            ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
            node[1] = H(PK.seed, ADRS, (node[0] || AUTH[k]));
          } else {
            ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
            node[1] = H(PK.seed, ADRS, (AUTH[k] || node[0]));
          }
          node[0] = node[1];
        }
        return node[0];
    }
```

Algorithm 10: `xmss_pkFromSig` – Computing an XMSS public key from an XMSS signature.

4.2. HT: The Hypertee

The SPHINCS$^+$ hypertree HT is a variant of XMSS^{MT}. It is essentially a certification tree of XMSS instances. A HT is a tree of several layers of XMSS trees. The trees on top and intermediate layers are used to sign the public keys, i.e., the root nodes, of the XMSS trees on the respective next layer below. Trees on the lowest layer are used to sign the actual messages, which are FORS public keys in SPHINCS$^+$. All XMSS trees in HT have equal height.

Consider a HT of total height h that has d layers of XMSS trees of height $h' = h/d$. Then layer $d-1$ contains one XMSS tree, layer $d-2$ contains $2^{h'}$ XMSS trees, and so on. Finally, layer 0 contains $2^{h-h'}$ XMSS trees.

4.2.1. HT Parameters

In addition to all XMSS parameters, a HT requires the hypertree height h and the number of tree layers d, specified as an integer value that divides h without remainder. The same tree height $h' = h/d$ and the same Winternitz parameter w are used for all tree layers.

4.2.2. HT Key Generation (Function `ht_PKgen`)

The HT private key is the secret seed **SK**.`seed` which is used to generate all the WOTS$^+$ private keys within the virtual structure spanned by the HT.

The HT public key is the public key (root node) of the single XMSS tree on the top layer. Its computation is explained below. The public key generation takes as input a private and a public seed.

```
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT

ht_PKgen(SK.seed, PK.seed){
     ADRS = toByte(0, 32);
     ADRS.setLayerAddress(d-1);
     ADRS.setTreeAddress(0);
     root = xmss_PKgen(SK.seed, PK.seed, ADRS);
     return root;
}
```

Algorithm 11: `ht_PKgen` – Generating an HT public key.

4.2.3. HT Signature

A HT signature $\mathbf{SIG_{HT}}$ is a byte string of length $(h + d * \texttt{len}) * n$. It consists of d XMSS signatures (of $(h/d + \texttt{len}) * n$ bytes each).

The data format for a signature is given in Figure 9

XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer 0) $((h/d + \texttt{len}) \cdot n$ bytes)
XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer 1) $((h/d + \texttt{len}) \cdot n$ bytes)
...
XMSS signature $\mathbf{SIG}_{\text{XMSS}}$ (layer $d-1$) $((h/d + \texttt{len}) \cdot n$ bytes)

Figure 9: HT signature

4.2.4. HT Signature Generation (Function `ht_sign`)

To compute a HT signature $\mathbf{SIG_{HT}}$ of a message M using, `ht_sign` (Algorithm 12) described below uses `xmss_sign` as defined in Section 4.1.6. The algorithm `ht_sign` takes as input a message M, a private seed $\mathbf{SK}$.`seed`, a public seed $\mathbf{PK}$.`seed`, and an index idx. The index identifies the leaf of the hypertree to be used to sign the message. The HT signature then consists of a stack of XMSS signatures using the XMSS trees on the path from the leaf with index idx to the top tree. Note that idx is passed as two separate arguments, split into an index to address the specific tree and the leaf index within that tree. This allows for a somewhat higher hypertree, as one can use a 64-bit integer for tree_idx to support parameters that conform to $h < 64 + h/d$. This matches the parameters in this specification If other parameter sets are used that allow greater h, the data type of tree_idx MUST be adapted accordingly.

Algorithm `ht_sign` uses `xmss_pkFromSig` to compute the root node of an XMSS instance after that instance was used for signing. An alternative is to use `xmss_PKgen`. However, `xmss_PKgen` rebuilds the whole tree while `xmss_pkFromSig` only does one call to `wots_pkFromSig` and $(h' - 1)$ calls to $\mathbf{H}$. The algorithm `ht_sign` as described below is just one way to generate a HT signature. Other methods MAY be used as long as they generate the same output.

```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index
    idx_tree, leaf index idx_leaf
# Output: HT signature SIG_HT

ht_sign(M, SK.seed, PK.seed, idx_tree, idx_leaf) {
     // init
     ADRS = toByte(0, 32);

     // sign
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     SIG_tmp = xmss_sign(M, SK.seed, idx_leaf, PK.seed, ADRS);
     SIG_HT = SIG_HT || SIG_tmp;
     root = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx_tree;
        idx_tree = (h - (j + 1) * (h / d)) most significant bits of idx_tree;
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idx_tree);
        SIG_tmp = xmss_sign(root, SK.seed, idx_leaf, PK.seed, ADRS);
        SIG_HT = SIG_HT || SIG_tmp;
        if ( j < d - 1 ) {
           root = xmss_pkFromSig(idx_leaf, SIG_tmp, root, PK.seed, ADRS);
        }
     }
     return SIG_HT;
}
```

Algorithm 12: `ht_sign` – Generating an HT signature

4.2.5. HT Signature Verification (Function `ht_verify`)

HT signature verification (Algorithm 13) can be summarized as d calls to `xmss_pkFromSig` and one comparison with a given value. HT signature verification takes a message M, a signature $\mathbf{SIG_{HT}}$, a public seed $\mathbf{PK}$.seed, an index idx (split into a tree index and a leaf index, as above), and a HT public key $\mathbf{PK}_{\mathrm{HT}}$.

```
# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree,
    leaf index idx_leaf, HT public key PK_HT.
# Output: Boolean

ht_verify(M, SIG_HT, PK.seed, idx_tree, idx_leaf, PK_HT){
     // init
     ADRS = toByte(0, 32);

     // verify
     SIG_tmp = SIG_HT.getXMSSSignature(0);
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     node = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx_tree;
        idx_tree = (h - (j + 1) * h / d) most significant bits of idx_tree;
        SIG_tmp = SIG_HT.getXMSSSignature(j);
        ADRS.setLayerAddress(j);
```

```
            ADRS.setTreeAddress(idx_tree);
            node = xmss_pkFromSig(idx_leaf, SIG_tmp, node, PK.seed, ADRS);
        }
        if ( node == PK_HT ) {
          return true;
        } else {
          return false;
        }
    }
```

Algorithm 13: `ht_verify` – Verifying a HT signature $\mathbf{SIG_{HT}}$ on a message M using a HT public key $\mathbf{PK}_{\mathrm{HT}}$

5. FORS: Forest Of Random Subsets

The SPHINCS$^+$ hypertree HT is not used to sign the actual messages but the public keys of FORS instances which in turn are used to sign message digests. FORS (pronounced [fɔːrs]), short for forest of random subsets, is a few-time signature scheme (FTS). FORS is an improvement of HORST [5] which in turn is a variant of HORS [20]. For security it is essential that the input to FORS is the output of a hash function. In the following we describe FORS as acting on bit strings.

FORS uses parameters k and $t = 2^a$ (example parameters are $t = 2^{15}, k = 10$). FORS signs strings of length ka bits. Here, we deviate from defining sizes in bytes as the message length in bits might not be a multiple of eight. The private key consists of kt random n-byte strings grouped into k sets, each containing t n-byte strings. The private key values are pseudorandomly generated from the main private seed $\mathbf{SK}$.`seed` in the SPHINCS$^+$ private key. In SPHINCS$^+$, the FORS private key values are only temporarily generated as an intermediate result when computing the public key or a signature.

The FORS public key is a single n-byte hash value. It is computed as the tweakable hash of the root nodes of k binary hash trees. Each of these binary hash trees has height a and is used to authenticate the t private key values of one of the k sets. Accordingly, the leaves of a tree are the (tweakable) hashes of the values in its private key set.

A signature on a string M consists of k private key values – one per set of private key elements – and the associated authentication paths. To compute the signature, md is split into k a-bit strings. As md is a sequence of bytes, we first convert to a bit-string by enumerating the bytes in md, internally enumerating the bits within a byte from least to most significant. Next, each of these bit strings is interpreted as an integer between 0 and $t - 1$. Each of these integers is used to select one private key value from a set. I.e., if the first integer is i, the ith private key element of the first set gets selected and so on. The signature consists of the selected private key elements and the associated authentication paths.

SPHINCS$^+$ uses implicit verification for FORS, only using a method to compute a candidate public key from a signature. This is done by computing root nodes of the k trees using the indices computed from the input string as well as the private key values and authentication paths form the signature. The tweakable hash of these roots is then returned as candidate public key.

We now describe the parameters and algorithms for FORS.

5.1. FORS Parameters

FORS uses the parameters n, k, and t; they all take positive integer values. These parameters are summarized as follows:

- n: the security parameter; it is the length of a private key, public key, or signature element in bytes.
- k: the number of private key sets, trees and indices computed from the input string.
- t: the number of elements per private key set, number of leaves per hash tree and upper bound on the index values. The parameter t MUST be a power of 2. If $t = 2^a$, then the trees have height a and the input string is split into bit strings of length a.

Inputs to FORS are bit strings of length $k \log t$.

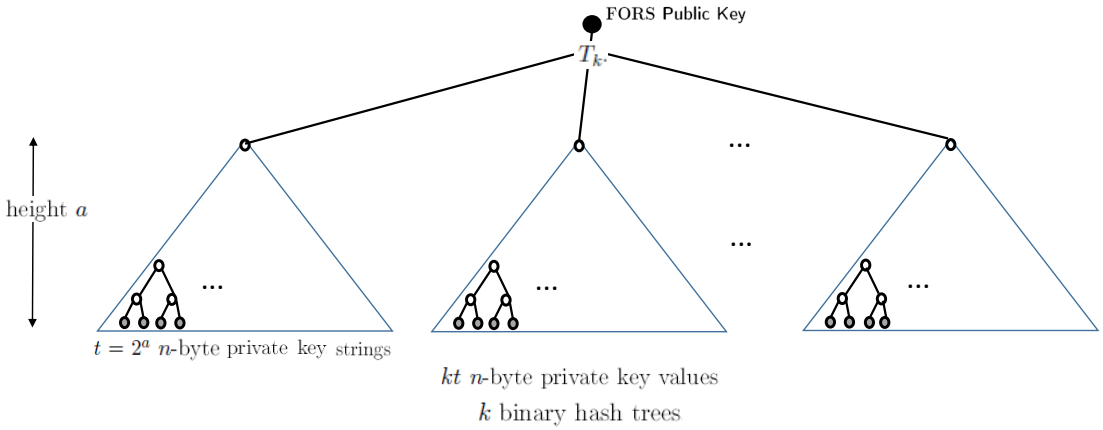


Figure 10: FORS trees and PK

5.2. FORS Private Key (Function `fors_SKgen`)

In the context of SPHINCS$^+$, a FORS private key is the single private seed **SK**.`seed` contained in the SPHINCS$^+$ private key. It is used to generate the kt n-byte private key values using **PRF** with an address. While these values are logically grouped into a two-dimensional array, for implementations it makes sense to assume they are in a one-dimensional array of length kt. The jth element of the ith set is then stored at $\mathsf{sk}[it + j]$. To generate one of these elements, a FORS address **ADRS** is used, that encodes the position of the FORS key pair within SPHINCS$^+$ and has tree height set to 0 and leaf index set to $it + j$:

```
#Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
#Output: FORS private key sk

fors_SKgen(SK.seed, ADRS, idx) {
  ADRS.setTreeHeight(0);
  ADRS.setTreeIndex(idx);
  sk = PRF(SK.seed, ADRS);
```

```
  return sk;
}
```

Algorithm 14: `fors_SKgen` – Computing a FORS private key value.

5.3. FORS TreeHash (Function `fors_treehash`)

Before coming to the FORS public key, we have to discuss computation of the trees. For the computation of the n-byte nodes in the FORS hash trees, the subroutine `fors_treehash` is used. It is essentially the same algorithm as `treehash` (Algorithm 7) in Section 4.1. The two differences are how the leaf nodes are computed and how addresses are handled. However, as the addresses are similar, an implementation can implement both algorithms in the same routine easily.

Algorithm `fors_treehash` accepts a secret seed **SK**.`seed`, a public seed **PK**.`seed`, an unsigned integer s (the start index), an unsigned integer z (the target node height), and an address **ADRS** that encodes the address of the FORS key pair. As for `treehash`, the `fors_treehash` algorithm returns the root node of a tree of height z with the leftmost leaf being the hash of the private key element at index s. Here, s is ranging over the whole kt private key elements. It is REQUIRED that $s \,\%\, 2^z = 0$, i.e. that the leaf at index s is a leftmost leaf of a sub-tree of height z. Otherwise the algorithm fails as it would compute non-existent nodes.

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack

fors_treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       ADRS.setTreeHeight(0);
       ADRS.setTreeIndex(s + i);
       sk = PRF(SK.seed, ADRS);
       node = F(PK.seed, ADRS, sk);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

Algorithm 15: The `fors_treehash` algorithm.

5.4. FORS Public Key (Function `fors_PKgen`)

In the context of SPHINCS$^+$, the FORS public key is never generated alone. It is only generated together with a signature. We include `fors_PKgen` for completeness, a better understanding, and testing. Algorithm `fors_PKgen` takes a private seed **SK**.`seed`, a public seed

PK.seed, and a FORS address **ADRS**. The latter encodes the position of the FORS instance within SPHINCS$^+$. It outputs a FORS public key.

```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK

fors_PKgen(SK.seed, PK.seed, ADRS) {
     forspkADRS = ADRS; // copy address to create FTS public key address

     for(i = 0; i < k; i++){
         root[i] = fors_treehash(SK.seed, i*t, a, PK.seed, ADRS);
     }
     forspkADRS.setType(FORS_ROOTS);
     forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
     pk = T_k(PK.seed, forspkADRS, root);
     return pk;
}
```

Algorithm 16: `fors_PKgen` – Generate a FORS public key.

5.5. FORS Signature Generation (Function `fors_sign`)

A FORS signature is a length $k(\log t + 1)$ array of n-byte strings. It contains k private key values, n-bytes each, and their associated authentication paths, $\log t$ n-byte values each.

The algorithm `fors_sign` takes a $(k \log t)$-bit string M, a private seed **SK**.seed, a public seed **PK**.seed, and an address **ADRS**. The latter encodes the position of the FORS instance within SPHINCS$^+$. It outputs a FORS signature $\mathbf{SIG}_{\text{FORS}}$.

```
#Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
#Output: FORS signature SIG_FORS

fors_sign(M, SK.seed, PK.seed, ADRS) {
  // compute signature elements
  for(i = 0; i < k; i++){
    // get next index
    unsigned int idx = bits i*log(t) to (i+1)*log(t) - 1 of M;

    // pick private key element
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    SIG_FORS = SIG_FORS || PRF(SK.seed, ADRS);

    // compute auth path
    for ( j = 0; j < a; j++ ) {
      s = floor(idx / (2^j)) XOR 1;
      AUTH[j] = fors_treehash(SK.seed, i * t + s * 2^j, j, PK.seed, ADRS);
    }
    SIG_FORS = SIG_FORS || AUTH;
  }
  return SIG_FORS;
}
```

Algorithm 17: `fors_sign` – Generating a FORS signature on string M.

The data format for a signature is given in Figure 11.

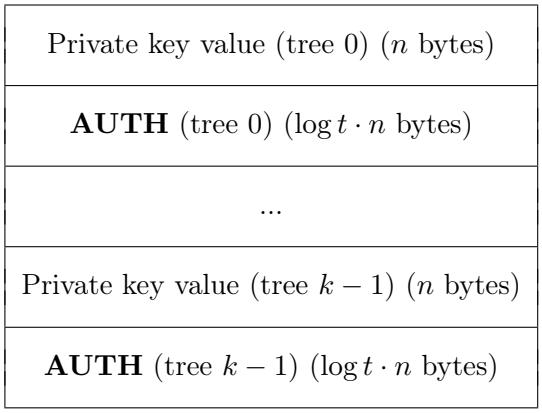


Figure 11: FORS signature

5.6. FORS Compute Public Key from Signature (Function `fors_pkFromSig`)

SPHINCS$^+$ makes use of implicit signature verification of FORS signatures. A FORS signature is used to compute a candidate FORS public key. This public key is used in further computations (message for the signature of the XMSS tree above) and implicitly verified by the outcome of that computation. Hence, this specification does not contain a `fors_verify` method but the method `fors_pkFromSig`.

The method `fors_pkFromSig` takes a $k \log t$-bit string M, a FORS signature $\mathbf{SIG}_{\text{FORS}}$, a public seed $\mathbf{PK}$.`seed`, and an address **ADRS**. The latter encodes the position of the FORS instance within the virtual structure of the SPHINCS$^+$ key pair. First, the roots of the k binary hash trees are computed using `fors_treehash`. Afterwards the roots are hashed using the tweakable hash function $\mathbf{T}_k$. The algorithm `fors_pkFromSig` is given as Algorithm 18. The method `fors_pkFromSig` makes use of functions $\mathbf{SIG}_{\text{FORS}}$`.getSK(i)` and $\mathbf{SIG}_{\text{FORS}}$`.getAUTH(i)`. The former returns the ith secret key element stored in the signature, the latter returns the ith authentication path stored in the signature.

```
# Input: FORS signature SIG_FORS, (k lg t)-bit string M, public seed PK.seed,
    address ADRS
# Output: FORS public key

fors_pkFromSig(SIG_FORS, M, PK.seed, ADRS){

  // compute roots
  for(i = 0; i < k; i++){
    // get next index
    unsigned int idx = bits i*log(t) to (i+1)*log(t) - 1 of M;

    // compute leaf
    sk = SIG_FORS.getSK(i);
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    node[0] = F(PK.seed, ADRS, sk);

    // compute root from leaf and AUTH
    auth = SIG_FORS.getAUTH(i);
```

```
    ADRS.setTreeIndex(i*t + idx);
    for ( j = 0; j < a; j++ ) {
      ADRS.setTreeHeight(j+1);
      if ( (floor(idx / (2^j)) % 2) == 0 ) {
        ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
        node[1] = H(PK.seed, ADRS, (node[0] || auth[j]));
      } else {
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
        node[1] = H(PK.seed, ADRS, (auth[j] || node[0]));
      }
      node[0] = node[1];
    }
    root[i] = node[0];
  }

  forspkADRS = ADRS; // copy address to create FTS public key address
  forspkADRS.setType(FORS_ROOTS);
  forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_k(PK.seed, forspkADRS, root);
  return pk;
}
```

Algorithm 18: `fors_pkFromSig` – Compute a FORS public key from a FORS signature.

6. SPHINCS$^+$

We now have all ingredients to describe our main construction SPHINCS$^+$. Essentially, SPHINCS$^+$ is an orchestration of the methods and schemes described before. It only adds randomized message compression and verifiable index generation.

6.1. SPHINCS$^+$ Parameters

SPHINCS$^+$ has the following parameters:

n : the security parameter in bytes.

w : the Winternitz parameter as defined in Section 3.1.

h : the height of the hypertree as defined in Section 4.2.1.

d : the number of layers in the hypertree as defined in Section 4.2.1.

k : the number of trees in FORS as defined in Section 5.1.

t : the number of leaves of a FORS tree as defined in Section 5.1.

All the restrictions stated in the previous sections apply. Recall that we use $a = \log t$. Moreover, from these values the values m and `len` are computed as

- m: the message digest length in bytes. It is computed as

$$m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor.$$

  While only $h + k \log t$ bits would be needed, using the longer m as defined above simplifies implementations significantly.

- `len`: the number of n-byte string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{8n}{\log(w)} \right\rceil, \quad \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

In the following, we assume that all algorithms have access to these parameters.

6.2. SPHINCS$^+$ Key Generation (Function `spx_keygen`)

The SPHINCS$^+$ private key contains two elements. First, the n-byte secret seed **SK**.`seed` which is used to generate all the WOTS$^+$ and FORS private key elements. Second, an n-byte PRF key **SK**.`prf` which is used to deterministically generate a randomization value for the randomized message hash.

The SPHINCS$^+$ public key also contains two elements. First, the HT public key, i.e. the root of the tree on the top layer. Second, an n-byte public seed value **PK**.`seed` which is sampled uniformly at random.

As `spx_sign` does not get the public key, but needs access to **PK**.`seed` (and possibly to **PK**.`root` for fault attack mitigation), the SPHINCS$^+$ secret key contains a copy of the public key.

The description of algorithm `spx_keygen` assumes the existence of a function `sec_rand` which on input i returns i-bytes of cryptographically strong randomness.

```
# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)

spx_keygen( ){
     SK.seed = sec_rand(n);
     SK.prf = sec_rand(n);
     PK.seed = sec_rand(n);
     PK.root = ht_PKgen(SK.seed, PK.seed);
     return ( (SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root) );
}
```

Algorithm 19: `spx_keygen` – Generate a SPHINCS$^+$ key pair.

The format of a SPHINCS$^+$ private and public key is given in Figure 12.

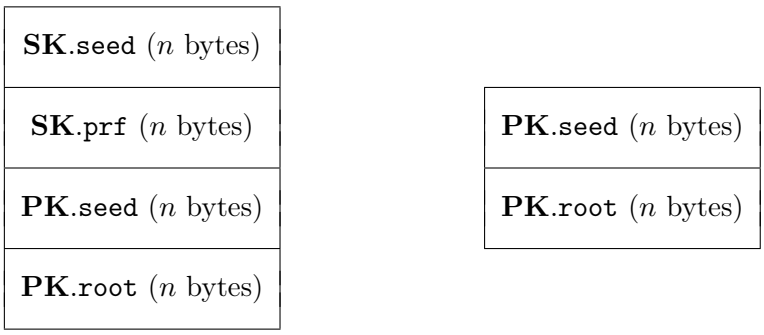


Figure 12: Left: SPHINCS$^+$ secret key. Right: SPHINCS$^+$ public key.

6.3. SPHINCS$^+$ Signature

A SPHINCS$^+$ signature $\mathbf{SIG_{HT}}$ is a byte string of length $(1+k(a+1)+h+d\texttt{len})n$. It consists of an n-byte randomization string R, a FORS signature $\mathbf{SIG_{FORS}}$ consisting of $k(a+1)$ n-byte strings, and a HT signature $\mathbf{SIG_{HT}}$ of $(h+d\texttt{len})n$ bytes.

The data format for a signature is given in Figure 9

Randomness $\mathbf{R}$ (n bytes)
FORS signature $\mathbf{SIG_{FORS}}$ ($k(a+1)\cdot n$ bytes)
HT signature $\mathbf{SIG_{HT}}$ ($(h+d\texttt{len})n$ bytes)

Figure 13: SPHINCS$^+$ signature

6.4. SPHINCS$^+$ Signature Generation (Function `spx_sign`)

Generating a SPHINCS$^+$ signature consists of four steps. First, a random value $\mathbf{R}$ is pseudorandomly generated. Next, this is used to compute a m byte message digest which is split into a $\lfloor(k\log t+7)/8\rfloor$-byte partial message digest `tmp_md`, a $\lfloor(h-h/d+7)/8\rfloor$-byte tree index `tmp_idx_tree`, and a $\lfloor(h/d+7)/8\rfloor$-byte leaf index `tmp_idx_leaf`. Next, the actual values md, `idx_tree`, and `idx_leaf` are computed by extracting the necessary number of bits. The partial message digest md is then signed with the `idx_leaf`-th FORS key pair of the `idx_tree`-th XMSS tree on the lowest HT layer. The public key of the FORS key pair is then signed using HT. As described in Section 4.2.3, the index is never actually used as a whole, but immediately split into a tree index and a leaf index, for ease of implementation.

When computing $\mathbf{R}$, the PRF takes a n-byte string `opt` which is initialized with zero but can be overwritten with randomness if the global variable RANDOMIZE is set. This option is given as otherwise SPHINCS$^+$ signatures would be always deterministic. This might be problematic in some settings. See Section 9 and Section 11 for more details.

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG

spx_sign(M, SK){
     // init
     ADRS = toByte(0, 32);

     // generate randomizer
     opt = toByte(0, n);
     if(RANDOMIZE){
       opt = rand(n);
     }
     R = PRF_msg(SK.prf, opt, M);
     SIG = SIG || R;

     // compute message digest and index
```

```
    digest = H_msg(R, PK.seed, PK.root, M);
    tmp_md = first floor((ka +7)/ 8) bytes of digest;
    tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
    tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;

    md = first ka bits of tmp_md;
    idx_tree = first h - h/d bits of tmp_idx_tree;
    idx_leaf = first h/d bits of tmp_idx_leaf;

    // FORS sign
    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idx_tree);
    ADRS.setType(FORS_TREE);
    ADRS.setKeyPairAddress(idx_leaf);

    SIG_FORS = fors_sign(md, SK.seed, PK.seed, ADRS);
    SIG = SIG || SIG_FORS;

    // get FORS public key
    PK_FORS = fors_pkFromSig(SIG_FORS, M, PK.seed, ADRS);

    // sign FORS public key with HT
    ADRS.setType(TREE);
    SIG_HT = ht_sign(PK_FORS, SK.seed, PK.seed, idx_tree, idx_leaf);
    SIG = SIG || SIG_HT;

    return SIG;
}
```

Algorithm 20: `spx_sign` – Generating a SPHINCS$^+$ signature

6.5. SPHINCS$^+$ Signature Verification (Function `spx_verify`)

SPHINCS$^+$ signature verification (Algorithm 21) can be summarized as recomputing message digest and index, computing a candidate FORS public key, and verifying the HT signature on that public key. Note that the HT signature verification will fail if the FORS public key is not matching the real one (with overwhelming probability). SPHINCS$^+$ signature verification takes a message M, a signature **SIG**, and a SPHINCS$^+$ public key **PK**.

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean

spx_verify(M, SIG, PK){
    // init
    ADRS = toByte(0, 32);
    R = SIG.getR();
    SIG_FORS = SIG.getSIG_FORS();
    SIG_HT = SIG.getSIG_HT();


    // compute message digest and index
    digest = H_msg(R, PK.seed, PK.root, M);
    tmp_md = first floor((ka +7)/ 8) bytes of digest;
    tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
    tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;
```

```
        md = first ka bits of tmp_md;
        idx_tree = first h - h/d bits of tmp_idx_tree;
        idx_leaf = first h/d bits of tmp_idx_leaf;

        // compute FORS public key
        ADRS.setLayerAddress(0);
        ADRS.setTreeAddress(idx_tree);
        ADRS.setType(FORS_TREE);
        ADRS.setKeyPairAddress(idx_leaf);

        PK_FORS = fors_pkFromSig(SIG_FORS, md, PK.seed, ADRS);

        // verify HT signature
        ADRS.setType(TREE);
        return ht_verify(PK_FORS, SIG_HT, PK.seed, idx_tree, idx_leaf, PK.root);
    }
```

Algorithm 21: `spx_verify` – Verify a SPHINCS$^+$ signature **SIG** on a message M using a SPHINCS$^+$ public key **PK**

7. Instantiations

This section discusses instantiations for SPHINCS$^+$. SPHINCS$^+$ can be viewed as a signature template. It is a way to build a signature scheme by instantiating the cryptographic function families used. We consider different ways to implement the cryptographic function families as different signature systems. Orthogonal to instantiating the cryptographic function families are parameter sets. Parameter sets assign specific values to the SPHINCS$^+$ parameters described in Section 7.1 below.

In this section, we first define the requirements on parameters and discuss existing trade-offs between security, sizes, and speed controlled by the different parameters. Then we propose 6 different parameter sets that match NIST security levels I, III, and V (2 parameter sets per security level). Afterwards we propose three different instantiations for the cryptographic function families of SPHINCS$^+$. These instantiation are indeed three different signature schemes. We propose SPHINCS$^+$-SHAKE256, SPHINCS$^+$-SHA-256, and SPHINCS$^+$-Haraka. The former two use the cryptographic hash functions defined in FIPS PUB 202, respectively FIPS PUB 180, to instantiate the cryptographic function families. The latter uses a new cryptographic (hash) function called Haraka, proposed in [13].

7.1. SPHINCS$^+$ Parameter Sets

SPHINCS$^+$ is described by the following parameters already described in the previous sections. All parameters take positive integer values.

n : the security parameter in bytes.

w : the Winternitz parameter.

h : the height of the hypertree.

d : the number of layers in the hypertree.

k : the number of trees in FORS.

t : the number of leaves of a FORS tree.

Recall that we use $a = \log t$. Moreover, from these values the values m and $\texttt{len}$ are computed as

- m: the message digest length in bytes. It is computed as $m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$.
- $\texttt{len}$: the number of n-byte string elements in a WOTS$^+$ private key, public key, and signature. It is computed as $\texttt{len} = \texttt{len}_1 + \texttt{len}_2$, with

$$\texttt{len}_1 = \left\lceil \frac{8n}{\log(w)} \right\rceil, \; \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

We now repeat the roles of, requirements on, and properties of these parameters. Afterwards, we give several formulas that show their exact influence on performance and security.

The security parameter n is also the output length of all cryptographic function families besides $\mathbf{H_{msg}}$. Therefore, it largely determines which security level a parameter set reaches. It is also the size of virtually any node within the SPHINCS$^+$ structure and thereby also the size of all elements in a signature, i.e., the signature size is a multiple of n.

The Winternitz parameter w determines the number and length of the hash chains per WOTS$^+$ instance. A greater value for w linearly increases the length of the hash chains but logarithmically reduces their number. The number of hash chains exactly corresponds to the number of n-byte values in a WOTS$^+$ signature. Thereby it largely influences the size of a SPHINCS$^+$ signature. The product of the number and the length of hash chains directly correlates with signing speed as essentially all time in HT signature generation is spent computing WOTS$^+$ public keys. Therefore, greater w means shorter signatures but slower signing. However, note the exponential gap. The bigger w gets, the more expensive is the signature size reduction. The Winternitz parameter does not influence SPHINCS$^+$ security.

The height of the hypertree h determines the number of FORS instances. Hence, it determines the probability that a FORS key pair is used several times, given the number of signatures made with a SPHINCS$^+$ key pair. Hence, the height has a direct impact on security: A taller hypertree gives more security. On the other hand, a taller tree leads to larger signatures.

The number of layers d is a pure performance trade-off parameter and does not influence security. It determines the number of layers of XMSS trees in the hypertree. Hence, d must divide h without remainder. The parameter d thereby defines the height of the XMSS trees used. The greater d, the smaller the subtrees, the faster signing. However, d also controls the number of layers and thereby the number of WOTS$^+$ signatures within a HT and thereby a SPHINCS$^+$ signature.

The parameters k and t determine the performance and security of FORS. The number of leaves of a tree in FORS t must be a power of two while k can be chosen freely. A smaller t generally leads to smaller and faster signatures. However, for a given security level a smaller t requires a greater k which increases signature size and slows down signing. Hence, it is important to balance these two parameters. This is best done using the formulas below.

The message digest length m is the output length of $\mathbf{H_{msg}}$ in bytes. It is $\lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$ bytes.

The number $\texttt{len}$ of chains in a WOTS$^+$ key pair determines the WOTS$^+$ signature size.

Table 1: Overview of the number of function calls we require for each operation. We omit the single calls to $\mathbf{H_{msg}}, \mathbf{PRF_{msg}}$, and $\mathbf{T}_k$ for signing and single calls to $\mathbf{H_{msg}}$ and $\mathbf{T}_k$ for verification as they are negligible when estimating speed.

	F	**H**	**PRF**	$T_{\texttt{len}}$
Key Generation	$2^{h/d}w\texttt{len}$	$2^{h/d}-1$	$2^{h/d}\texttt{len}$	$2^{h/d}$
Signing	$kt + d(2^{h/d})w\texttt{len}$	$k(t-1) + d(2^{h/d}-1)$	$kt + d(2^{h/d})\texttt{len}$	$d2^{h/d}$
Verification	$k + dw\texttt{len}$	$k \log t + h$	–	d

Table 2: Key and signature sizes

	SK	PK	Sig
Size	$4n$	$2n$	$(h + k(\log t + 1) + d \cdot len + 1)n$

7.1.1. Influence of Parameters on Security and Performance

In the following we provide formulas to compute speed, size and security for a given SPHINCS$^+$ parameter set. This supports parameter selection. We also provide a SAGE script in Appendix A.

Key Generation. Generating the SPHINCS$^+$ private key and $\mathbf{PK}.\texttt{seed}$ requires three calls to a secure random number generator. Next we have to generate the top tree. For the leaves we need to do $2^{h/d}$ WOTS$^+$ key generations ($\texttt{len}$ calls to $\mathbf{PRF}$ for generating the $\texttt{sk}$ and $w\texttt{len}$ calls to $\mathbf{F}$ for the $\texttt{pk}$) and we have to compress the WOTS$^+$ public key (one call to $T_{\texttt{len}}$). Computing the root of the top tree requires $(2^{h/d}-1)$ calls to $\mathbf{H}$.

Signing. For randomization and message compression we need one call to $\mathbf{PRF_{msg}}$, and one to $\mathbf{H_{msg}}$. The FORS signature requires kt calls to $\mathbf{PRF}$ and $\mathbf{F}$. Further, we have to compute the root of k binary trees of height $\log t$ which adds $k(t-1)$ calls to $\mathbf{H}$. Finally, we need one call to T_k. Next, we compute one HT signature which consists of d trees similar to the key generation. Hence, we have to do $d(2^{h/d})$ times $\texttt{len}$ calls to $\mathbf{PRF}$ and $w\texttt{len}$ calls to $\mathbf{F}$ as well as $d(2^{h/d})$ calls to $\mathbf{T}_{\texttt{len}}$. For computing the root of each tree we get additionally $d(2^{h/d}-1)$ calls to $\mathbf{H}$.

Verification. First we need to compute the message hash using $\mathbf{H_{msg}}$. We need to do one FORS verification which requires k calls to $\mathbf{F}$ (to compute the leaf nodes from the signature elements), $k \log t$ calls to $\mathbf{H}$ (to compute the root nodes using the leaf nodes and the authentication paths), and one call to T_k for hashing the roots. Next, we have to verify d XMSS signatures which takes $< w\texttt{len}$ calls to $\mathbf{F}$ and one call to $T_{\texttt{len}}$ each for WOTS$^+$ signature verification. It also needs dh/d calls to $\mathbf{H}$ for the d root computations.

The size of the SPHINCS$^+$ private and public keys along with the signature can be deduced from Section 6 and is shown in Table 2.

The classical security level, or bit security of SPHINCS$^+$ against generic attacks can be

computed as

$$b = -\log\left(\frac{1}{2^{8n}} + \sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right).$$

The quantum security level, or bit security of SPHINCS$^+$ against generic attacks can be computed as

$$b = -\frac{1}{2}\log\left(\frac{1}{2^{8n}} + \sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right).$$

Here, we are neglecting the small constant factors inside the logarithm. For details see Section 9.

7.1.2. Proposed Parameter Sets and Security Levels

As explained in the previous subsection, even for a fixed security level the design of SPHINCS$^+$ supports many different tradeoffs between signature size and speed. In Table 3 we list 6 parameter sets that—together with the cycle counts given in Table 4— illustrates how these tradeoffs can be used to obtain concrete parameter sets optimizing for signature size and concrete parameter sets optimizing for speed. Specifically, we propose parameter sets achieving security levels 1, 3, and 5; for each of these security levels propose one size-optimized (ending on 's' for "small") and one speed-optimized (ending on 'f' for "fast") parameter set. The parameter sets were obtained with the help of a Sage script that we list in Appendix A. In the first line of that script, set the "target bit security" to a desired value (in our case, close to 128 for security level 1, close to 192 for security level 3, and close to 256 for security level 5). The output of the script will be a long list of possible parameters achieving this security level together with the signature size and an estimate of the performance, using the formulas from Section 7.1.1 above.

Note that we did *not* obtain our proposed parameter sets simply by searching this output for the smallest or the fastest option. The reason is that, for example, optimizing for size without caring about speed at all results in signatures of a size of $\approx 15\,\text{KB}$ for a bit security of 256, but computing one signature takes more than 20 minutes on our benchmark platform. Such a tradeoff might be interesting for very few select applications, but we cannot think of many applications that would accept such a large time for signing. Instead, the proposed parameter sets are what we consider "non-extreme"; i.e., with a signing time of at most a few seconds in our non-optimized implementation.

The choice of these parameters is orthogonal to the choice of hash function. In Section 7.2 we describe three different instantiations of the underlying hash function, each with a simple and a robust variant. Together with the six parameter sets listed in Table 3 we obtain 36 different instantiations of SPHINCS$^+$.

7.2. Instantiations of Hash Functions

In this section we define different signature schemes, which are obtained by instantiating the cryptographic function families of SPHINCS$^+$ with SHA-256, SHAKE256, and Haraka. To instantiate the tweakable hash functions, we present two different constructions. Leading to a total of six instantiations. For the 'robust' instances, we first generate pseudorandom *bitmasks*

Table 3: Example parameter sets for SPHINCS$^+$ targeting different security levels and different tradeoffs between size and speed. Note that these parameter sets have been update for round 3. The column labeled "bitsec" gives the bit security computed as described in Section 9; the column labeled "sec level" gives the security level according to the levels specified in Section 4.A.5 of the Call for Proposals. As explained later, **for Haraka the security level is limited to 2:** i.e., it is 1 for $n = 16$, and 2 for $n = 24$ or $n = 32$.

	n	h	d	$\log(t)$	k	w	bitsec	sec level	sig bytes
SPHINCS$^+$-128s	16	63	7	12	14	16	133	**1**	7 856
SPHINCS$^+$-128f	16	66	22	6	33	16	128	**1**	17 088
SPHINCS$^+$-192s	24	63	7	14	17	16	193	**3**	16 224
SPHINCS$^+$-192f	24	66	22	8	33	16	194	**3**	35 664
SPHINCS$^+$-256s	32	64	8	14	22	16	255	**5**	29 792
SPHINCS$^+$-256f	32	68	17	9	35	16	255	**5**	49 856

which are then XORed with the input message. The masked messages are denoted as $M^{\oplus}$. For the 'simple' instances, we take an approach inspired by the LMS proposal for stateful hash-based signatures [15], and omit the bitmasks. We make this difference explicit in the instances defined below. The 'simple' instances are faster as they omit the calls to **PRF** to generate bitmasks. When combined with compressed addresses in the SHA-256 case this can lead to an estimated reduction of the number of compression function calls by a factor of almost 4. In return, this comes at the cost of a security argument that entirely relies on the random oracle model.

Recall that n and m are the security parameter and the message digest length, in bytes.

7.2.1. SPHINCS$^+$-SHAKE256

For SPHINCS$^+$-SHAKE256 we define

$$\begin{aligned}
\mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M) &= \mathrm{SHAKE256}(\mathbf{R}||\mathbf{PK}.\mathtt{seed}||\mathbf{PK}.\mathtt{root}||M, 8m), \\
\mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \mathrm{SHAKE256}(\mathbf{SEED}||\mathbf{ADRS}, 8n), \\
\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M) &= \mathrm{SHAKE256}(\mathbf{SK}.\mathtt{prf}||\mathtt{OptRand}||M, 8n).
\end{aligned} \tag{1}$$

For the robust variant, we further define the tweakable hash functions as

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}, 8n), \\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1||M_2) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1^{\oplus}||M_2^{\oplus}, 8n), \\
\mathbf{T}_\ell(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M^{\oplus}, 8n),
\end{aligned} \tag{2}$$

For the simple variant, we instead define the tweakable hash functions as

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1, 8n), \\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M_1||M_2) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M_1||M_2, 8n), \\
\mathbf{T}_\ell(\mathbf{PK}.\mathtt{seed}, \mathbf{ADRS}, M) &= \mathrm{SHAKE256}(\mathbf{PK}.\mathtt{seed}||\mathbf{ADRS}||M, 8n),
\end{aligned} \tag{3}$$

Generating the Masks. SHAKE256 can be used as an XOF which allows us to generate the bitmasks for arbitrary length messages directly. For a message M with l bits we compute

$$M^{\oplus} = M \oplus \text{SHAKE256}(\mathbf{PK}.\texttt{seed}||\mathbf{ADRS}, l).$$

7.2.2. SPHINCS$^+$-SHA-256

In a similar way we define the functions for SPHINCS$^+$-SHA-256 as

$$\begin{aligned}
\mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\texttt{seed}, \mathbf{PK}.\texttt{root}, M) &= \text{MGF1-SHA-256}(\text{SHA-256}(\mathbf{R}||\mathbf{PK}.\texttt{seed}||\mathbf{PK}.\texttt{root}||M), m),\\
\mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \text{SHA-256}(\mathbf{SEED}||\mathbf{ADRS}^c),\\
\mathbf{PRF_{msg}}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M) &= \text{HMAC-SHA-256}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}||M).
\end{aligned} \tag{4}$$

For the robust variant, we further define the tweakable hash functions as

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M_1^{\oplus}),\\
\mathbf{H}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1||M_2) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M_1^{\oplus}||M_2^{\oplus}),\\
\mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M^{\oplus}),
\end{aligned} \tag{5}$$

For the simple variant, we instead define the tweakable hash functions as

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M_1),\\
\mathbf{H}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1||M_2) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M_1||M_2),\\
\mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M) &= \text{SHA-256}(\mathbf{PK}.\texttt{seed}||\text{toByte}(0, 64-n)||\mathbf{ADRS}^c||M),
\end{aligned} \tag{6}$$

Here, we use MGF1 as defined in RFC 2437 and HMAC as defined in FIPS-198-1. Note that MGF1 takes as the last input the output length in bytes.

Generating the Masks. SHA-256 can be turned into a XOF using MGF1 which allows us to generate the bitmasks for arbitrary length messages directly. For a message M with l bytes we compute

$$M^{\oplus} = M \oplus \text{MGF1-SHA-256}(\mathbf{PK}.\texttt{seed}||\mathbf{ADRS}^c, l).$$

Padding PK.seed. Each of the instances of the tweakable hash function take $\mathbf{PK}.\texttt{seed}$ as its first input, which is constant for a given key pair – and, thus, across a single signature. This leads to a lot of redundant computation. To remedy this, we pad $\mathbf{PK}.\texttt{seed}$ to the length of a full 64-byte SHA-256 input block. Because of the Merkle-Damgård construction that underlies SHA-256, this allows for reuse of the intermediate SHA-256 state after the initial call to the compression function which improves performance.

Compressing ADRS. To ensure that we require the minimal number of calls to the SHA-256 compression function, we use a compressed $\mathbf{ADRS}$ for each of these instances. Where possible, this allows for the SHA2 padding to fit within the last input block. Rather than storing the layer address and type field in a full 4-byte word each, we only include the least-significant byte of each. Similarly, we only include the least-significant 8 bytes of the 12-byte tree address. This reduces the address from 32 to 22 bytes. We denote such compressed addresses as $\mathbf{ADRS}^c$.

Shorter Outputs. If a parameter set requires an output length $n < 32$-bytes for $\mathbf{F}$, $\mathbf{H}$, $\mathbf{PRF}$, and $\mathbf{PRF_{msg}}$ we take the first n bytes of the output and discard the remaining.

7.2.3. SPHINCS$^+$-Haraka

Our third instantiation is based on the Haraka short-input hash function. Haraka is not a NIST-approved hash function, and since it is new it needs further analysis. We specify SPHINCS$^+$-Haraka as third signature scheme to demonstrate the possible speed-up by using a dedicated short-input hash function.

As the Haraka family only supports input sizes of 256 and 512 bits we extend it with a sponge-based construction based on the 512-bit permutation π. The sponge has a rate of 256-bit respectively a capacity of 256-bit and the number of rounds used in π is 5. The padding scheme is the same as defined in FIPS PUB 202 for SHAKE256.

We denote this sponge as HarakaS(M, d), where M is the padded message and d is the length of the message digest in bits. A 256-bit message block M_i is absorbed into the state S by

$$\text{Absorb}(M, S) : S = \pi(S \oplus (M||\texttt{toByte}(0, 32))). \tag{7}$$

The d-bit hash output h is computed by squeezing blocks of r bits

$$\begin{aligned} \text{Squeeze}(S) : h &= h||\text{Trunc}_{256}(S) \\ S &= \pi(S). \end{aligned} \tag{8}$$

For a more efficient construction we generate the round constants of Haraka using $\mathbf{PK}.\texttt{seed}$.[1] As $\mathbf{PK}.\texttt{seed}$ is the same for all hash function calls for a given key pair we expand $\mathbf{PK}.\texttt{seed}$ using HarakaS and use the result for the round constants in all instantiations of Haraka used in SPHINCS$^+$. In total there are 40 128-bit round constants defined by

$$RC_0, \ldots, RC_{39} = \text{HarakaS}(\mathbf{PK}.\texttt{seed}, 5120). \tag{9}$$

This only has to be done once for each key pair for all subsequent calls to Haraka hence the costs for this are amortized. We denote Haraka with the round constants derived from $\mathbf{PK}.\texttt{seed}$ as $\text{Haraka}_{\mathbf{PK}.\texttt{seed}}$. We can now define all functions we need for SPHINCS$^+$-Haraka as

$$\begin{aligned} \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\texttt{seed}, \mathbf{PK}.\texttt{root}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{R}||\mathbf{PK}.\texttt{root}||M, 8m), \\ \mathbf{PRF}(\mathbf{SEED}, \mathbf{ADRS}) &= \text{Haraka256}_{\mathbf{SEED}}(\mathbf{ADRS}), \\ \mathbf{PRF_{msg}}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{SK}.\texttt{prf}||\texttt{OptRand}||M, 8n). \end{aligned} \tag{10}$$

For the robust variant, we further define the tweakable hash functions as

$$\begin{aligned} \mathbf{F}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1) &= \text{Haraka512}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M_1^{\oplus}), \\ \mathbf{H}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1||M_2) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M_1^{\oplus}||M_2^{\oplus}, 8n), \\ \mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M^{\oplus}, 8n), \end{aligned} \tag{11}$$

For the simple variant, we instead define the tweakable hash functions as

$$\begin{aligned} \mathbf{F}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1) &= \text{Haraka512}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M_1), \\ \mathbf{H}(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M_1||M_2) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M_1||M_2, 8n), \\ \mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \mathbf{ADRS}, M) &= \text{HarakaS}_{\mathbf{PK}.\texttt{seed}}(\mathbf{ADRS}||M, 8n), \end{aligned} \tag{12}$$

[1] This is similar to the ideas used for the MDx-MAC construction [19].

For $\mathbf{F}$ we pad M_1 and $M_1^{\oplus}$ with zero if $n < 32$. Note that $\mathbf{H}$ and $\mathbf{H_{msg}}$ will always have a different $\mathbf{ADRS}$ and we therefore do not need any further domain separation.

Generating the Masks. The mask for the message used in $\mathbf{F}$ is generated by computing

$$M_1^{\oplus} = M_1 \oplus \text{Haraka256}_{\mathbf{PK.seed}}(\mathbf{ADRS}) \tag{13}$$

For all other purposes the masks are generated using HarakaS. For a message M with l bytes we compute

$$M^{\oplus} = M \oplus \text{HarakaS}_{\mathbf{PK.seed}}(\mathbf{ADRS}, l).$$

Shorter Outputs. If a parameter set requires an output length $n < 32$-bytes for $\mathbf{F}$ and $\mathbf{PRF}$, we take the first n bytes of the output and discard the remaining.

Security Restrictions. Note that our instantiation using Haraka employs the sponge construction with a capacity of 256-bits. Hence, in contrast to SPHINCS$^+$-SHA-256 and SPHINCS$^+$-SHAKE256, SPHINCS$^+$-Haraka reaches security level 2 for 32- and 24-byte outputs and security level 1 for 16-byte outputs.

8. Design rationale

The design rationale behind SPHINCS$^+$ is to follow the original SPHINCS construction and apply several results from more recent research. The idea behind SPHINCS was as follows. One can build a stateless hash-based signature scheme using a massive binary certification tree and selecting a leaf at random for each message to be signed. The problem with this approach is that the tree has to be extremely high, i.e., a height of about twice the security level would be necessary. This leads to totally unpractical signature sizes. Using a hypertree instead of a binary certification tree allows to trade speed for signature size. However, this is still not sufficient to get practical sizes and speed.

The main new idea in SPHINCS was to not use the leaves directly to sign messages but to use the leaves to certify FTS key pairs. This allowed to massively reduce the total tree height (by a factor about 4). This is due to the fact that the security of an FTS instance degrades with every signature a key pair is used for. Hence, the height of the tree does not have to be such that collisions do only occur with negligible probability anymore. Instead, it has to be ensured that the product of the probability of a γ-times collision on a leaf and the forging probability of an adversary after seeing γ FTS signatures (with the same key pair) is negligible.

From this, it is mainly a question of balancing parameters to find a practical scheme. For the full original reasoning see [5].

In the following we give a more detailed reasoning regarding the changes made to SPHINCS in SPHINCS$^+$, and changes that were discussed by the SPHINCS$^+$ team but got discarded.

8.1. Changes Made

We changed several details of SPHINCS leading to SPHINCS$^+$. The reasoning behind those changes is discussed in the following.

8.1.1. Multi-Target Attack Protection

SPHINCS was designed to be collision-resilient i.e., to not be vulnerable to collision attacks against the used hash function. This had two reasons. First, it allowed to choose a smaller output length at the same security level which led to smaller signatures. Second, collision resistance is a far stronger assumption than the used (second-)preimage resistance and pseudorandomness assumptions.

However, the use of (second-)preimage resistance introduced a new issue as pointed out in [11]: Multi-target attacks. Preimage resistance properties are targeted properties. An adversary is asked to invert the function on a given target value, or to find a second-preimage for a given target value. If it suffices to break the given property for one out of many targets, the adversarial effort is reduced by a factor of the number of targets. To prevent this in our we apply the mitigation techniques from [11] using keyed hash functions. Each hash function call is keyed with a different key and applies different bitmasks. Keys are derived from, and bitmasks are pseudorandomly generated from a public seed and an address specifying the context of the call. For this we introduce the notion of tweakable hash functions which take in addition to the input value a public seed and an address.

This pseudorandom generation of bitmasks comes at the cost of introducing a random oracle assumption for the PRF used to generate the bitmasks. However, this only applies to the pseudorandom generation of the bitmasks. I.e., if all bitmasks would be stored in the public key, the scheme would have a standard model security proof (even if these bitmasks where generated using exactly the same way but without giving away the seed). Hence, the security reduction in [11] is in the quantum-accessible random oracle model.

One difference to [11] is that in all instantiations of SPHINCS$^+$, keys are not pseudorandomly generated. Instead, the concatenation of public seed and address is used to practically key the functions. Given how the tweakable hash functions are instantiated, this means that we assume that there do not exist any (exponentially large) subsets of the domain on which second-preimage finding is easy. This assumption holds for any hash function based on the sponge or Merkle-Dåmgard construction, assuming the block or compression function behaves like a random function.

8.1.2. Tree-less WOTS$^+$ Public Key Compression

SPHINCS$^+$ compresses the end nodes of the WOTS$^+$ hash chains with a single call to a tweakable hash function, while SPHINCS used a so called L-tree. The reason to use L-trees in SPHINCS was that this required only two n-byte bitmasks per layer, i.e., $2\lceil \log \texttt{len} \rceil$ bitmasks. A single call to a tweakable hash requires `len` n-byte bitmasks. As the bitmasks were stored in the public key, this meant smaller public keys. Now, that bitmasks are pseudorandomly generated anyway and hence are not stored in the public key anymore, this argument does not apply. On the opposite, tree based compression is slower than using a single call to a tweakable hash with longer input.

8.1.3. FORS

FORS was used to replace HORST. HORST, as its predecessor HORS, had the problem that weak messages existed as recently independently pointed out in [1]. More specifically, in HORST the message is also split into k indexes as for FORS. However, these indexes all selected values from the same single set of secret key values. Hence, if the same index appeared

multiple times in a signature, still only a single secret value would be required. In extreme cases this means that for the signature of a message only a single secret value has to be know. FORS prevents this using separate secret value sets per index obtained from the message. Even if a message maps k-times to the same index, the signature now contains k different secret values.

For the same parameters k and t this would mean an increase in signature size and worse speed as now k trees of height $\log t$ have to be computed instead of one and for each signature value an authentication path of length $(\log t) - 1$ is needed. However, due to the strengthened security, we can choose different values for k and t. This in the end leads to smaller signatures than for HORST.

We also considered a method similar to Octopus [2]. The idea is that authentication paths in HORST largely overlap. Hence, it becomes possible to reduce the signature size removing any redundancy in the authentication paths. This comes at the cost of a rather involved method to collect the right nodes as well as variable size signatures. In practice this means that one still has to prepare for the worst case. This worst case indeed still has smaller signatures than HORST. We decided against this option as the FORS signature size matches that of Octopus' worst case signature size. At the same time, FORS gives more flexibility in the choice of k and t, and comes with a far simpler signature and verification method that Octopus.

8.1.4. Verifiable Index Selection

In SPHINCS the index of the HORST instance to be used was pseudorandomly selected. This had the drawback that the index appeared random to a verifier and it was impossible to verify that the index was indeed generated that way. This allowed an adversary a multi-target attack on HORST (similarly for FORS in SPHINCS$^+$). An adversary could first map a message to an index set and then check if the necessary secret values were already uncovered for some HORST key pair. Then it would just select the index of that HORST key pair as index and succeed in forging a signature.

To prevent this attack, we decided to make index generation verifiable. More specifically, we generate the index together with the message digest:

We compute message digest and index as

$$(\text{md}||\text{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}, M)$$

where $\mathbf{PK} = (\mathbf{PK}.\texttt{seed}||\mathbf{PK}.\texttt{root})$ contains the top root node and the public seed.

This way, an adversary can no longer freely choose an index. Indeed, selecting a message immediately also fixes the index. This method has another advantage in addition to avoiding the multi-target attack against FORS/HORST. We can omit the index in the SPHINCS signature as it would be redundant.

8.1.5. Making Deterministic Signing Optional

The pseudorandom generation of randomizer $\mathbf{R}$ now allows to use additional randomness. It takes a n-byte value `OptRand`. Per default `OptRand` is set to 0 but it can be filled with random bits e.g. taken from a TRNG. The randomizer is then computed as

$$\mathbf{R} = \mathbf{PRF}(\mathbf{SK}.\texttt{prf}, \texttt{OptRand}, M).$$

That way, deterministic signing becomes optional. Deterministic signing can be a problem for devices which are susceptible to side-channel attacks as it allows to collect several traces for the exactly same computation by just asking for a signature on the same message multiple times.

We could of course also have replaced **R** by a truly random value on default. This would have caused the scheme to become susceptible to bad randomness. The new method prevents this. If `OptRand` is a high entropy string, **R** has as much entropy as that string. If `OptRand` is left as zero or has only little entropy, **R** is just a pseudorandom value as in SPHINCS.

8.1.6. SPHINCS$^+$-'simple' and SPHINCS$^+$-'robust'

The updated, Round 2 submission of SPHINCS$^+$ introduces instantiations of the tweakable hash functions similar to those of the LMS proposal for stateful hash-based signatures [15]. These instantiations are called 'simple' (compared to the established instantiations which we now call 'robust'). The 'simple' instantiations omit the use of bitmasks, i.e., no bitmasks have to be generated and XORed with the message input of the tweakable hash functions **F**, **H** or **T**. This has the advantage of better speed since the calls to the underlying hash function (needed in order to generate the bitmasks for each tweakable hash calculation) are saved. However, the resulting drawback is a security argument which in its entirety only applies in the random oracle model.

Another reason to propose these simple instantiations is the possibility to align the construction with the stateful scheme [15] such that clients can easily implement the verification procedure for both with a small code-base, as for the robust instantiations and XMSS. However, the simple instantiations of SPHINCS$^+$ are so far not compatible with the LMS signature scheme as described in [15]. The simple instantiations of SPHINCS$^+$ uses **PK**.`seed` and **ADRS** to distinguish hash calls. LMS uses a specially crafted security string which has the same purpose, is similar, but differs in the details.

Most of the time in SPHINCS$^+$, XMSS, and LMS, is spent on **F** computations. The LMS proposal [15] optimized the length of their security string for SHA-256 to ensure that the **F** computations of the OTS signatures can be done with a single compression function call. We use a similar approach, applied to our SPHINCS$^+$-SHA-256 instantiation. For this purpose we compress the hash addresses in case of SHA-256 instantiations and pad **PK**.`seed` to fit a full compression function block (with an exception of the mask generation). As **PK**.`seed` is constant for a key pair, this allows to precompute the internal state of SHA-256 after absorbing this block and reduce the necessary online computations to a single compression function call for the SHA-256-simple instances. Also for the robust instantiations this saves a factor of two in compression function calls. For SHAKE256 and Haraka such an optimization is of no effect as one **F** computation already takes only a single call to the inner function.

8.2. Discarded Changes

In Section 8.1.3, we already explained that we discarded the use of an Octopus-like method as we found a better alternative.

One more idea which we discarded on the way was a signature - secret key size trade-off. To further shrink the SPHINCS$^+$ signature size, the top z layers of the hypertree can be merged together into a a single tree of height zh'. That way an SPHINCS$^+$ signature includes $z-1$ less WOTS$^+$ signatures. This decreases the signature size by $n \cdot len(z-1)$ bytes, but typically

comes at the cost of speed as now a tree of height zh' has to be computed for each signature generation. This can be prevented by storing the nodes at height ih', where $0 < i < z$, as part of the secret key. These nodes (auxiliary data) can be used to build the authentication paths to the root of the merged tree without actually computing the whole tree. Indeed, authentication path computation in this case gets faster than computing the authentication paths for z tree layers in the original hypertree. The size of the auxiliary data is $n \sum_{i=1}^{z-1} 2^{ih'}$. While this already grows extremely fast, the real problem turned out to be key generation time. As the full tree still has to be computed once during key generation, key generation time increases. Key generation would now take $2^{zh'}$ WOTS$^+$ key generations.

Initial experiments suggested that key generation time easily moves into the order of minutes already for $z = 2$ while the benefit in signature size is 1KB or 2KB for $w = 256$ and $w = 16$ respectively. In addition, this optimization significantly complicates implementations as the top tree has to be handled differently than the remaining trees. Hence, this idea was discarded.

9. Security Evaluation (including estimated security strength and known attacks)

The security of SPHINCS$^+$ is based on standard properties of the used function families. These in turn can be derived from the properties of the hash functions used to instantiate those function families. For the robust instantiations, these properties can be derived from standard model properties of the used hash function and the assumption that the PRF used within the instantiations of the tweakable hash functions (to generate the bitmasks) can be modeled as a random oracle. We want to emphasize again that this assumption about the random oracle is limited to the pseudorandom generation of bitmasks. For the simple instantiations, these properties can be derived from the assumption that the used hash function behaves like a random oracle even in the presence of quantum adversaries which are given quantum oracle access to the function.

Disclaimer: The following two subsections present an attempt for a tight security reduction for SPHINCS$^+$ that turned out to be flawed; we keep them here for reference. The flaw is an artifact of the attempt to prove a tight security reduction for the variant of the Winternitz one-time signature scheme used by SPHINCS$^+$ taken from [11]. It should be noted that the non-tight proof for WOTS$^+$ from [9] still applies. Also, the flaw does not translate into an attack but just demonstrates that the proof made false assumptions. Indeed, at the time of writing we are positive that the problem can be circumvented. Hence, this does not influence our security estimates at all. In the following we briefly outline the flaw and a previous issue with the security reduction and discuss the solution.

A first issue was fixed since version 2 of this specification and appeared in the scientific publication on SPHINCS$^+$ [3]. That work also discusses the security of the simple instantiations, not discussed below. The issue was as follows. The security reduction makes a statistical assumption about the used hash function which does not hold for a random function and, consequently, should not hold for a good cryptographic hash function. This assumption essentially states that every possible input to $\mathbf{F}$ has at least one colliding value under $\mathbf{F}$ (which we call sibling). It is trivial to construct a hash function for which it is reasonable to conjecture this property. Just take for example SHA-256, apply it once, truncate the result to 248 bits and apply SHA-256 again. However, this would have to be paid for by a factor 2 penalty in speed.

The flaw which persisted also in [3] is related to the same part of the proof. It is concerned with arguing about the hardness of finding x given $y = \mathbf{F}(x)$. The above assumption was used to argue that if y has at least two preimages x, x', it is information theoretically hidden from an adversary which preimage was used to compute y. This argument applies if x is chosen uniformly at random from the whole domain, and if no side-information about x exists. It does not necessarily apply if the input is known to be an output of another function (in our case $\mathbf{F}$). Intuitively what is required under this condition is that this side-information does still not allow an adversary to determine which preimage was used to compute y. This can be shown using the additional assumption that $\mathbf{F}$ is undetectable, as used in previous works (e.g., [9]). Undetectability says that an image of a function on a random input is indistinguishable from a random element in its codomain.

The revised proof will be made available via the NIST pqc-forum mailing list, as well as via the SPHINCS$^+$ website `https://sphincs.org`, upon publication.

Reductionist proof. In this section we give a security reduction for SPHINCS$^+$ underpinning the above claim. The security reduction essentially combines the original SPHINCS security reduction from [5], the XMSS-T security reduction from [11], and a new security analysis for multi-instance FORS.

In our technical specification of SPHINCS$^+$ we used the abstraction of tweakable hash functions to allow for different ways of keying a function and generating bitmasks. In the security reduction we will remove this abstraction and assume that each call to the hash function used to instantiate the tweakable hash is keyed with a different value and inputs are XORed with a bitmask before being processed. Moreover, we assume that the bitmasks are generated using a third PRF called $\mathbf{PRF_{BM}}$. The PRF $\mathbf{PRF_{BM}}$ is the single function assumed to behave like a random oracle. Finally, we make a statistical assumption on the hash function F. Informally we require that every element in the image of F has at least two preimages, i.e.,

$$(\forall k \in \{0,1\}^n)(\forall y \in \mathrm{IMG}(\mathrm{F}_k))(\exists x, x' \in \{0,1\}^n) : x \neq x' \wedge \mathrm{F}_k(x) = f_k(x'). \tag{14}$$

Informally, we will prove the following Theorem where F, H, and T are the cryptographic hash functions used to instantiate $\mathbf{F}$ and $\mathbf{H}$, respectively.

Theorem 9.1 *For security parameter $n \in \mathbb{N}$, parameters w, h, d, m, t, k as described above, SPHINCS$^+$ is existentially unforgeable under post-quantum adaptive chosen message attacks if*

- F, H, *and* T *are post-quantum distinct-function multi-target second-preimage resistant function families,*
- F *fulfills the requirement of Eqn. 14,*
- $\mathbf{PRF}, \mathbf{PRF_{msg}}$ *are post-quantum pseudorandom function families,*
- $\mathbf{PRF_{BM}}$ *is modeled as a quantum-accessible random oracle, and*
- $\mathbf{H_{msg}}$ *is a post-quantum interleaved target subset resilient hash function family.*

More specifically, the insecurity function $\mathrm{InSec}^{\text{PQ-EU-CMA}}\left(\mathit{SPHINCS}^{+};\xi,2^{h}\right)$ *describing the maximum success probability over all adversaries running in time* $\leq \xi$ *against the* PQ-EU-CMA *security of SPHINCS*$^{+}$ *is bounded by*

$$\begin{aligned}\mathrm{InSec}^{\text{PQ-EU-CMA}}\left(\mathit{SPHINCS}^{+};\xi\right) \leq 2(&\mathrm{InSec}^{\text{PQ-PRF}}\left(\mathbf{PRF};\xi\right)+\mathrm{InSec}^{\text{PQ-PRF}}\left(\mathbf{PRF_{msg}};\xi\right)\\ &+\mathrm{InSec}^{\mathit{pq\text{-}itsr}}\left(\mathbf{H_{msg}};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{F};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{H};\xi\right)+\mathrm{InSec}^{\text{PQ-DM-SPR}}\left(\mathrm{T};\xi\right))\end{aligned} \tag{15}$$

9.1. Preliminaries

Before we start with the proof, we have to provide two definitions. In general, we refer the reader to [11] for formal definitions of the above properties with two exceptions. First, we use a variant of post-quantum multi-function multi-target second-preimage resistance called post-quantum *distinct*-function multi-target second-preimage resistance. The distinction here is that the targets are given for distinct but predefined functions from the family while for the multi-function notion, the functions are sampled together with the target, uniformly at random.

Second, we define a variant of subset-resilience which captures the use of FORS in SPHINCS^{+} which we call (post-quantum) interleaved target subset resilience. The idea is that from a theoretical point of view, one can think of the 2^{h} FORS instances as a single huge HORS-style signature scheme. The secret key consists of 2^{h} key-sets which in turn consist of k key-subsets of t secret n-byte values, each. The message digest function $\mathbf{H_{msg}}$ maps a message to a key-set (by outputting the index) and a set of indexes such that each index is used to select one secret value per key-subset of the selected key-set.

Formally, the security of this multi-instance FORS boils down to the inability of an adversary

- to learn actual secret values which were not disclosed before,
- to replace secret values by values of its choosing, and
- to find a message which is mapped to a key-set and a set of indexes such that the adversary has already seen the secret values indicated by the indexes for that key-set.

The former two points will be shown to follow from the properties of F, H, and T as well as those of **PRF**. The latter point is exactly what (post-quantum) interleaved target subset resilience captures.

We define those properties in the following.

Post-quantum distinct-function, multi-target second-preimage resistance (PQ-DM-SPR). In the following let $\lambda \in \mathbb{N}$ be the security parameter, $\alpha = \mathrm{poly}(\lambda), \kappa = \mathrm{poly}(\lambda)$, and $\mathcal{H}_{\lambda} = \{\mathrm{H}_K : \{0,1\}^{\alpha} \to \{0,1\}^{\lambda}\}_{K\in\{0,1\}^{\kappa}}$ be a family of functions. We define the success probability of any (quantum) adversary $\mathcal{A}$ against PQ-MM-SPR. This definition is parameterized by the number of targets

$$\begin{aligned}\mathrm{Succ}^{\text{PQ-DM-SPR}}_{\mathcal{H}_{\lambda},p}\left(\mathcal{A}\right) = \mathsf{Pr}\big[(\forall\,\{K_i\}_1^q \subset (\{0,1\}^{\kappa})^q), M_i \xleftarrow{\$} \{0,1\}^{\alpha}, 0<i\leq p;\\ (j,M') \xleftarrow{\$} \mathcal{A}((K_1,M_1),\ldots,(K_p,M_p)) :\\ M' \neq M_j \wedge \mathrm{H}_{K_j}(M_j) = \mathrm{H}_{K_j}(M')\big]\ .\end{aligned} \tag{16}$$

(Post-quantum) interleaved target subset resilience. In the following let $\lambda \in \mathbb{N}$ be the security parameter, $\alpha = \text{poly}(\lambda), \kappa = \text{poly}(\lambda)$, and $\mathcal{H}_\lambda = \{\text{H}_K : \{0,1\}^\alpha \to \{0,1\}^\lambda\}_{K \in \{0,1\}^\kappa}$ be a family of functions. Further consider the mapping function $\text{MAP}_{h,k,t} : \{0,1\}^\lambda \to \{0,1\}^h \times [0, t-1]^k$ which for parameters h, k, t maps an λ-bit string to a set of k indexes $((I, 1, J_1), \ldots, (I, k, J_k))$ where I is chosen from $[0, 2^h - 1]$ and each J_i is chosen from $[0, t-1]$. Note that the same I is used for all tuples (I, i, J_i).

We define the success probability of any (quantum) adversary $\mathcal{A}$ against PQ-MM-SPR of $\mathcal{H}_\lambda$. Let $\text{G} = \text{MAP}_{h,k,t} \circ \mathcal{H}_\lambda$. This definition uses an oracle $\mathcal{O}(\cdot)$ which upon input of a α-bit message M_i samples a key $K_i \xleftarrow{\$} \{0,1\}^\kappa$ and returns K_i and $\text{G}(K_i, M_i)$. The adversary may query this oracle with messages of its choosing. The adversary would like to find another G input whose output is covered by the G outputs produced by the oracle, without the input being one of the inputs used by the oracle. Note that the adversary knows the description of G and can evaluate it on randomizer-message pairs of its choosing. However, these queries do not count into the set of values which need to cover the adversary's output.

$$\text{Succ}^{\text{pq-itsr}}_{\mathcal{H},q}(\mathcal{A}) = \mathsf{Pr}\Big[(K, M) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(1^\lambda) \quad \text{s.t.} \quad \text{G}(K, M) \subseteq \bigcup_{j=1}^{q} \text{G}(K_j, M_j) \\ \wedge (K, M) \notin \{(K_j, M_j)\}_1^q\Big]$$

where q denotes the number of oracle queries of $\mathcal{A}$ and the pairs $\{(K_j, M_j)\}_1^q$ represent the responses of oracle $\mathcal{O}$.

Note that this is actually a strengthening of (post-quantum) target subset resilience in the multi-target setting. In the multi-target version of target subset resilience, $\mathcal{A}$ was able to freely choose the common index I for its output. In interleaved target subset resilience, I is determined by G and input M.

9.2. Security Reduction

The security reduction is essentially an application of techniques used especially in [11]. Hence, we will only roughly sketch it here.

We want to bound the success probability of an adversary $\mathcal{A}$ against the PQ-EU-CMA security of SPHINCS$^+$. We start with GAME.0 which is the original PQ-EU-CMA game. Now consider a second game GAME.1 where all outputs of **PRF** are replaced by truly random values. The difference in success probability of any forger $\mathcal{A}$ must be bound by $\text{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$ otherwise we could use $\mathcal{A}$ to break the pseudorandomness of **PRF** with a success probability greater $\text{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$ which would contradict the definition of $\text{InSec}^{\text{PQ-PRF}}(\mathbf{PRF}; \xi)$.

Next, consider a game GAME.2 which is the same as GAME.1 but all outputs of $\mathbf{PRF_{msg}}$ are replaced by truly random values. Following the same reasoning as above, the difference in success probability of any adversary $\mathcal{A}$ playing in the two games must be bounded by $\text{InSec}^{\text{PQ-PRF}}(\mathbf{PRF_{msg}}; \xi)$.

Next, we consider GAME.3 where we consider the game lost if $\mathcal{A}$ outputs a valid forgery $(M, \mathbf{SIG})$ where the FORS signature part of **SIG** differs from the signature which would be obtained by signing M with the secret key of the challenger. The difference of any $\mathcal{A}$ in winning the two games must be bounded by $\text{InSec}^{\text{PQ-DM-SPR}}(\text{F}; \xi) + \text{InSec}^{\text{PQ-DM-SPR}}(\text{H}; \xi) + \text{InSec}^{\text{PQ-DM-SPR}}(\text{T}; \xi)$. Otherwise, we could use $\mathcal{A}$ to break the post-quantum distinct-function,

multi-target second-preimage resistance of F, H, or T. A detailed proof of this follows exactly along the lines of the security reduction for XMSS-T in [11]. Given distinct challenges for each call to F, H or T for the key-set defined by $\mathbf{PK}.\mathtt{seed}$ and the address space, we program $\mathbf{PRF_{BM}}$ to output bitmasks which are the XOR of the input to the according tweakable hash function and the given challenge. That way we program the actual input to the hash function to be the challenge value. This allows us to extract a second preimage if a collision happens between the forgery and the honestly generated signature. A pigeon hole argument can be used to show that such a collision must exist in this case.

Next, we consider GAME.4 which differs from GAME.3 in that we are considering the game lost if an adversary outputs a valid forgery $(M, \mathbf{SIG})$ where the FORS signature part of $\mathbf{SIG}$ contains a secret value which is the same as that of an honestly generated signature of M but was not contained in any of the signatures obtained by $\mathcal{A}$ via the singing oracle. The difference of any (unbounded) $\mathcal{A}$ in the two games is bounded by $1/2$ times the success probability of $\mathcal{A}$ in GAME.3. The reason is that the secret values which were not disclosed to $\mathcal{A}$ before still contain 1 bit of entropy, even for an unbounded $\mathcal{A}$.

Finally, we have to bound the success probability of $\mathcal{A}$ in GAME.4. But GAME.4 can be viewed as the (post-quantum) interleaved target subset resilience game. Because, if $\mathcal{A}$ returns a valid signature and succeeds in the GAME, the FORS signature must be valid and consist only of values that have been observed by $\mathcal{A}$ in previous signatures. Hence, the success probability of $\mathcal{A}$ in GAME.4 is bounded by $\mathrm{InSec}^{\text{pq-itsr}}(\mathbf{H_{msg}}; \xi)$ per definition.

Putting things together we obtain the claimed bound. □

9.3. Security Level / Security Against Generic Attacks

As shown in Theorem 9.1, the security of SPHINCS$^+$ relies on the properties of the functions used to instantiate all the cryptographic function families (and the way they are used to instantiate the function families). In the following we assume that there do not exist any structural attacks against the used functions SHA-256, SHAKE256, and Haraka. In later sections we justify this assumption for each of the function familes.

For now, we only consider generic attacks. We now consider generic classical and quantum attacks against distinct-function multi-target second-preimage resistance, pseudorandomness (of function families), and interleaved target subset resilience. Runtime of adversaries is counted in terms of calls to the cryptographic function families.

9.3.1. Distinct-Function Multi-Target Second-Preimage Resistance

To evaluate the complexity of generic attacks against hash function properties the hash functions are commonly modeled as (family of) random functions. Note, that for random functions there is no difference between distinct-function multi-target second-preimage resistance and multi-function multi-target second-preimage resistance. Every key just selects a new random function, independent of the key being random or not. In [11] it was shown that the success probability of any classical q_{hash}-query adversary against multi-function multi-target second-preimage resistance of a random function with range $\{0,1\}^{8n}$ (and hence also against distinct-function multi-target second-preimage resistance) is exactly $\frac{q_{\text{hash}}+1}{2^{8n}}$. For q_{hash}-query quantum adversaries the success probability is $\Theta(\frac{(q_{\text{hash}}+1)^2}{2^{8n}})$. Note that these bounds are independent of the number of targets.

9.3.2. Pseudorandomness of Function Families

The best generic attack against the pseudorandomness of a function family is commonly believed to be exhaustive key search. Hence, for a function family with key space $\{0,1\}^{8n}$ the success probability of a classical adversary that evaluates the function family on q_{key} keys is again bounded by $\frac{q_{\text{key}}+1}{2^{8n}}$. For q_{key}-query quantum adversaries the success probability of exhaustive search in an unstructured space with $\{0,1\}^{8n}$ elements is $\Theta(\frac{(q_{\text{key}}+1)^2}{2^{8n}})$ as implicitly shown in [11] (just consider this as preimage search of a random function).

9.3.3. Interleaved Target Subset Resilience

To evaluate the attack complexity of generic attacks against interleaved target subset resilience we again assume that the used hash function family is a family of random functions.

Recall that there are parameters h, k, t where $t = 2^a$. These parameters define the following process of choosing sets: generate independent uniform random integers $I, J_1, \ldots, J_k$, where I is chosen from $[0, 2^h - 1]$ and each J_i is chosen from $[0, t-1]$; then define $S = \{(I, 1, J_1), (I, 2, J_2), \ldots, (I, k, J_k)\}$. (In the context of SPHINCS$^+$, S is a set of positions of FORS private key values revealed in a signature: I selects the FORS instance, and J_i selects the position of the value revealed from the ith set inside this FORS instance.)

The core combinatorial question here is the probability that $S_0 \subset S_1 \cup \cdots \cup S_q$, where each S_i is generated independently by the above process. (In the context of SPHINCS$^+$, this is the probability that a new message digest selects FORS positions that are covered by the positions already revealed in q signatures.) Write S_α as $\{(I_\alpha, 1, J_{\alpha,1}), (I_\alpha, 2, J_{\alpha,2}), \ldots, (I_\alpha, k, J_{\alpha,k})\}$.

For each α, the event $I_\alpha = I_0$ occurs with probability $1/2^h$, and these events are independent. Consequently, for each $\gamma \in \{0, 1, \ldots, q\}$, the number of indices $\alpha \in \{1, 2, \ldots, q\}$ such that $I_\alpha = I_0$ is γ with probability $\binom{q}{\gamma}(1 - 1/2^h)^{q-\gamma}/2^{h\gamma}$.

Define DarkSide_γ as the conditional probability that $(I_0, i, J_{0,i}) \in S_1 \cup \cdots \cup S_q$, given that the above number is γ. In other words, $1 - \text{DarkSide}_\gamma$ is the conditional probability that $(I_0, i, J_{0,i}) \notin \{(I_1, i, J_{1,i}), (I_2, i, J_{2,i}), \ldots, (I_q, i, J_{q,i})\}$. There are exactly γ choices of $\alpha \in \{1, 2, \ldots, q\}$ for which $I_\alpha = I_0$, and each of these has probability $1 - 1/t$ of $J_{\alpha,i}$ missing $J_{0,i}$. These probabilities are independent, so $1 - \text{DarkSide}_\gamma = (1 - 1/t)^\gamma$.

The conditional probability that $S_0 \subset S_1 \cup \cdots \cup S_q$, again given that the above number is γ, is the kth power of the DarkSide_γ quantity defined above. Hence the total probability ϵ that $S_0 \subset S_1 \cup \cdots \cup S_q$ is

$$\sum_\gamma \text{DarkSide}_\gamma^k \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}} = \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}}.$$

For example, if $t = 2^{14}$, $k = 22$, $h = 64$, and $q = 2^{64}$, then $\epsilon \approx 2^{-256.01}$ (with most of the sum coming from γ between 7 and 13). The set S_0 thus has probability $2^{-256.01}$ of being covered by 2^{64} sets $S_1, \ldots, S_q$. (In the SPHINCS$^+$ context, a message digest chosen by the attacker has probability $2^{-256.01}$ of selecting positions covered by 2^{64} previous signatures.)

Hence, for any classical adversary which makes q_{hash} queries to function family $\mathcal{H}_n$ the success probability is

$$(q_{\text{hash}} + 1) \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}}.$$

As this for random $\mathcal{H}_n$ is search in unstructured data, the best a quantum adversary can do is Grover search. This leads to a success probability of

$$\mathcal{O}\left((q_{\text{hash}}+1)^2 \sum_{\gamma} \left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k} \binom{q}{\gamma} \left(1-\frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}}\right).$$

For computations, note that the $\mathcal{O}$ is small, and that $(1-1/t)^{\gamma}$ is well approximated by $1-\gamma/t$.

9.3.4. Security Level of a Given Parameter Set

If we take the above success probabilities for generic attacks and plug them into Theorem 9.1 we get a bound on the success probability of SPHINCS$^+$ against generic attacks of classical and quantum adversaries. Let q denote the number of adversarial signature queries. For classical adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$\begin{aligned}
\text{InSec}^{\text{EU-CMA}}\left(\text{SPHINCS}^+; q_{\text{hash}}\right) &\leq 2\Big(\frac{q_{\text{hash}}+1}{2^{8n}} + \frac{q_{\text{hash}}+1}{2^{8n}} \\
&\quad + \text{InSec}^{\text{pq-itsr}}\left(\mathbf{H_{msg}}; q_{\text{hash}}\right) + \frac{q_{\text{hash}}+1}{2^{8n}} + \frac{q_{\text{hash}}+1}{2^{8n}} + \frac{q_{\text{hash}}+1}{2^{8n}}\Big) \\
&= 10\frac{q_{\text{hash}}+1}{2^{8n}} + 2(q_{\text{hash}}+1)\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}} \\
&= \mathcal{O}\left(\frac{q_{\text{hash}}}{2^{8n}} + (q_{\text{hash}})\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right). \quad (17)
\end{aligned}$$

Similarly, for quantum adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$\begin{aligned}
\text{InSec}^{\text{PQ-EU-CMA}}\left(\text{SPHINCS}^+; q_{\text{hash}}\right) &\leq 2\Big(\frac{(q_{\text{hash}}+1)^2}{2^{8n}} + \frac{(q_{\text{hash}}+1)^2}{2^{8n}} \\
&\quad + \text{InSec}^{\text{pq-itsr}}\left(\mathbf{H_{msg}}; q_{\text{hash}}\right) + \frac{(q_{\text{hash}}+1)^2}{2^{8n}} + \frac{(q_{\text{hash}}+1)^2}{2^{8n}} + \frac{(q_{\text{hash}}+1)^2}{2^{8n}}\Big) \\
&= 10\frac{(q_{\text{hash}}+1)^2}{2^{8n}} + \mathcal{O}\left(2(q_{\text{hash}}+1)^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right) \\
&= \mathcal{O}\left(\frac{(q_{\text{hash}})^2}{2^{8n}} + 2(q_{\text{hash}})^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right). \quad (18)
\end{aligned}$$

To compute the security level also known as bit security one sets this bound on the success probability to equal 1 and solves for q_{hash}.

9.4. Implementation Security and Side-Channel Protection

Timing attacks. Typical implementations of SPHINCS$^+$ are naturally free of any secret-dependent branches or secretly indexed loads or stores. SPHINCS$^+$ implementations are

thus free of the two most notorious sources of timing variation. An exception is potentially SPHINCS$^+$-Haraka, because Haraka is based on AES, which is well known to exhibit timing vulnerabilities in software implementations [4, 18, 6, 17]. Clearly, SPHINCS$^+$-Haraka should only be used in environments that support AES in hardware (like almost all modern 64-bit Intel and AMD and many ARMv8a processors). On *some* processors also certain arithmetic instructions do not run in constant time; examples are division instructions on Intel processors and the `UMULL` multiplication instruction on ARM Cortex-M3 proceesors. Again, typical implementations of SPHINCS$^+$ naturally do not use these instructions with secret data as input – secret data is only processed by symmetric cryptographic primitives that are *designed* to not make use of such potentially dangerous arithmetic.

Differential and fault attacks. We expect that any implementation of SPHINCS$^+$ without dedicated protection against differential power or electromagnetic radiation (EM) attacks or against fault-injection attacks will be vulnerable to such attacks. Deployment scenarios of SPHINCS$^+$ in which an attacker is assumed to have the power to mount such attacks require specially protected implementations. For protection against differential attacks this will typically require masking of the symmetric primitives; for protection against fault-injection attacks countermeasures on the hardware level. One additional line of defense against such advanced implementation attacks is included in the specification of SPHINCS$^+$, namely the option to randomize the signing procedure via the value `OptRand` (see Subsection 8.1.5).

9.5. Security of SPHINCS$^+$-SHAKE256

NIST has standardized several applications of the Keccak permutation, such as the SHA3-256 hash function and the SHAKE256 extendable-output function, after a multi-year Cryptographic Hash Algorithm Competition involving extensive public input. All of these standardized Keccak applications have a healthy security margin against all attacks known.

Discussions of the theory of cryptographic hash functions typically identify a few important properties such as collision resistance, preimage resistance, and second-preimage resistance; and sometimes include a few natural variants of the attack model such as multi-target attacks and quantum attacks. It is important to understand that cryptanalysts engage in a much broader search for any sort of behavior that is feasible to detect and arguably "non-random". NIST's call for SHA-3 submissions highlighted preimage resistance etc. but then stated the following:

> Hash algorithms will be evaluated against attacks or observations that may threaten existing or proposed applications, or demonstrate some fundamental flaw in the design, such as exhibiting nonrandom behavior and failing statistical tests.

It is, for example, non-controversial to use Keccak with a partly secret input as a PRF: any attack against such a PRF would be a tremendous advance in SHA-3 cryptanalysis, even though the security of such a PRF is not implied by properties such as preimage resistance. Similarly, a faster-than-generic attack against the interleaved-target-subset-resilience property, being able to find an input with various patterns of output bits, would be a tremendous advance.

The particular function SHAKE256 used in SPHINCS$^+$-SHAKE256 has an internal "capacity" of 512 bits. There are various attack strategies that search for 512-bit internal collisions, but this is not a problem even at the highest security category that we aim for. There is

also progress towards showing the hardness of generic quantum attacks against the sponge construction. Of course, second-preimage resistance is limited by the n-byte output length that we use.

9.6. Security of SPHINCS$^+$-SHA-256

NIST's SHA-2 family has been standardized for many more years than SHA-3. The standardization and popularity of SHA-2 mean that these functions are attractive targets for cryptanalysts, but this has not produced any attacks of concern: each of the members of this family has a comfortable security margin against all known attacks.

The broad cryptanalytic goal of finding non-random behavior (see above) is not a new feature of SHA-3. For example, the security analysis of the popular HMAC-SHA-256 message-authentication code is based on the security analysis of NMAC-SHA-256, which in turn is based on a pseudorandomness assumption for SHA-256.

The particular function SHA-256 used in SPHINCS$^+$-SHA-256 has a "chaining value" of only 256 bits, making it slightly weaker in some metrics than SHAKE256 with 256-bit output. However, it is still suitable for all of our target security categories.

9.7. Security of SPHINCS$^+$-Haraka

Both Haraka-256 and Haraka-512 provide a (second)-preimage resistance of 256-bit in the pre-quantum setting and the best known quantum attack is Grover's search on 256-bit. However, the sponge construction we use for HarakaS has a capacity of 256-bit which allows at most security level 2. The best attack breaking any of the security properties required for SPHINCS$^+$ is a preimage attack which corresponds to a collision search on 256-bit for the sponge construction we use. Instances with larger output size are limited by this and provide a less efficient trade-off between security and efficiency.

Another aspect is that we pseudo-randomly generate round constants derived from a seed. An attacker cannot influence the values of the constants for one instance, but can search for instances having weak constants. As shown by Jean [12], a weak choice of round constants can lead to more efficient preimage attacks. In general, a bad choice of round constants does not break the symmetry of a single round. In the case of Haraka, which combines several calls of two rounds of AES-128 per round to create bigger blocks, the round constants have to break the symmetry within two rounds of AES, but also between the different calls of the two rounds. Let us first focus on Haraka-256.

To break the symmetry within one round of AES, we require that the value of the round constant is not the same for each column. For round constants generated via an extendable-output function from a random 256-bit seed, we consider this event to happen with a probability of 2^{-96}. Moreover, that the symmetry of two rounds of AES is not broken by round-constants happens with 2^{-192}. In other words, since one instance of Haraka-256 uses 10 times 2-round AES, only for a fraction of $10 \cdot 2^{-192}$ instances/keys, we expect that the symmetry within one call of 2 rounds of AES is not broken. Even if this happens, all other 2 round AES calls used in Haraka-256 have with a high probability constants that break the symmetry of 2 rounds of AES for all other calls. Hence, we do not expect any negative consequences for the security.

Haraka-256 processes two 2-round AES-calls in parallel per round. So, we also do not want to have the same round constants in these calls. This condition happens with probability $5 \cdot 2^{-256}$. Furthermore, the probability that two rounds have the same round constants is

$10 \cdot 2^{-512}$. Similar observations are also valid for Haraka-512. Hence, we conclude that it is very unlikely, that a pseudo-random generation of the round constants per instance leads to weak round constants.

10. Performance

In order to obtain benchmarks, we evaluate our reference implementation on a machine using the Intel x86-64 instruction set. In particular, we use a single core of a 3.1 GHz Intel Xeon E3-1220 CPU (Haswell). We follow the standard practice of disabling TurboBoost and hyper-threading. Furthermore, it has 32GiB of RAM and the system ran on Ubuntu 18.04 with Linux kernel is 4.15.0-96-generic. We compiled the code using gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0, with the compiler optimization flag `-O3`.

10.1. Runtime

For the defined parameter sets, the resulting cycle counts are listed in Table 4.

We also provide optimized implementations for platforms supporting the AVX2 instruction set. For Haraka, it is especially relevant to also examine platforms that have the AES-NI instruction set available. We used the same system as described above, this time including the `march=native` compiler flag, as well as `flto` and `fomit-frame-pointer`. Performance results are listed in Table 5, Table 6 and Table 7.

10.2. Space

In Table 8, we list the key and signature sizes (in bytes) for the defined parameter sets. In terms of memory consumption, we remark that the reference implementation tends towards low stack usage. This shows for example in procedures such as computing authentication paths and tree roots, which is done using the treehash algorithm (which requires stack usage linear in the tree height, rather than the naive exponential approach of first computing the entire tree and then cherry-picking the relevant nodes).

11. Advantages and Limitations

The advantages and limitations of SPHINCS$^+$ can be summarized in one sentence: On the one hand, SPHINCS$^+$ is probably the most conservative design of a post-quantum signature scheme, on the other hand, it is rather inefficient in terms of signature size and speed. In the following we discuss disadvantages and advantages in some more detail.

Disadvantage: Signature size and speed. The clear drawback of SPHINCS$^+$ is signing speed and signature size. SPHINCS$^+$ is clearly not competing to be the smallest or fastest signature scheme. However, as shown in Section 7.1.1 there exists a magnitude of possible trade-offs allowing to tweak SPHINCS$^+$ as long as one can tolerate at least one of the two, i.e., somewhat slow signing *or* somewhat large signatures.

Advantage: "Minimal Security Assumptions". In contrast to other post-quantum crypto schemes (including signatures as well as public-key encryption schemes), SPHINCS$^+$ does not introduce a new intractability assumption. The security of SPHINCS$^+$ is solely based on

	key generation	signing	verification
SPHINCS$^+$-SHAKE256-128s-simple	616 484 336	4 682 570 992	4 764 084
SPHINCS$^+$-SHAKE256-128s-robust	1 195 409 786	8 995 481 640	9 232 084
SPHINCS$^+$-SHAKE256-128f-simple	9 649 130	239 793 806	12 909 924
SPHINCS$^+$-SHAKE256-128f-robust	18 726 982	460 757 304	28 152 828
SPHINCS$^+$-SHAKE256-192s-simple	898 362 434	8 091 419 556	6 465 506
SPHINCS$^+$-SHAKE256-192s-robust	1 753 646 932	15 306 007 790	13 509 022
SPHINCS$^+$-SHAKE256-192f-simple	14 215 518	386 861 992	19 876 926
SPHINCS$^+$-SHAKE256-192f-robust	27 463 376	734 072 042	39 295 686
SPHINCS$^+$-SHAKE256-256s-simple	594 081 566	7 085 272 100	10 216 560
SPHINCS$^+$-SHAKE256-256s-robust	1 156 363 648	13 198 544 260	19 292 734
SPHINCS$^+$-SHAKE256-256f-simple	36 950 136	763 942 250	19 886 032
SPHINCS$^+$-SHAKE256-256f-robust	72 503 094	1 467 095 732	39 555 542
SPHINCS$^+$-SHA-256-128s-simple	358 061 994	2 721 595 944	2 712 044
SPHINCS$^+$-SHA-256-128s-robust	714 027 022	5 363 065 742	5 561 194
SPHINCS$^+$-SHA-256-128f-simple	5 590 602	138 610 500	7 757 942
SPHINCS$^+$-SHA-256-128f-robust	11 135 058	273 836 364	16 101 098
SPHINCS$^+$-SHA-256-192s-simple	524 116 024	5 012 149 284	4 333 066
SPHINCS$^+$-SHA-256-192s-robust	1 059 562 738	9 893 267 932	8 557 224
SPHINCS$^+$-SHA-256-192f-simple	8 227 944	232 973 880	11 768 382
SPHINCS$^+$-SHA-256-192f-robust	16 581 076	458 983 816	24 679 894
SPHINCS$^+$-SHA-256-256s-simple	346 844 762	4 499 800 456	6 060 438
SPHINCS$^+$-SHA-256-256s-robust	1 006 905 074	12 382 647 014	18 784 558
SPHINCS$^+$-SHA-256-256f-simple	21 763 590	468 188 036	11 934 164
SPHINCS$^+$-SHA-256-256f-robust	62 599 672	1 312 473 846	37 139 082
SPHINCS$^+$-Haraka-128s-simple	576 421 410	4 594 239 682	5 211 916
SPHINCS$^+$-Haraka-128s-robust	1 095 155 240	8 555 157 606	9 749 770
SPHINCS$^+$-Haraka-128f-simple	9 137 070	232 172 172	13 148 448
SPHINCS$^+$-Haraka-128f-robust	17 119 708	430 223 622	27 216 072
SPHINCS$^+$-Haraka-192f-simple	13 399 816	392 561 468	20 424 354
SPHINCS$^+$-Haraka-192f-robust	25 376 582	736 487 034	40 090 578
SPHINCS$^+$-Haraka-192s-simple	857 570 254	8 710 115 544	7 572 424
SPHINCS$^+$-Haraka-192s-robust	1 625 369 822	16 200 067 256	15 276 532
SPHINCS$^+$-Haraka-256f-simple	35 650 224	832 534 808	22 061 746
SPHINCS$^+$-Haraka-256f-robust	67 266 388	1 519 602 658	42 244 366
SPHINCS$^+$-Haraka-256s-simple	569 851 046	8 717 853 894	11 740 252
SPHINCS$^+$-Haraka-256s-robust	1 077 657 774	15 566 614 908	22 516 650

Table 4: Runtime benchmarks for SPHINCS$^+$

	key generation	signing	verification
SPHINCS$^+$-Haraka-128s-simple	30 075 604	240 763 926	308 774
SPHINCS$^+$-Haraka-128s-robust	37 113 806	304 905 780	432 066
SPHINCS$^+$-Haraka-128f-simple	482 332	12 196 792	799 808
SPHINCS$^+$-Haraka-128f-robust	587 548	15 176 760	1 072 774
SPHINCS$^+$-Haraka-192s-simple	46 369 950	481 682 614	480 264
SPHINCS$^+$-Haraka-192s-robust	63 387 838	718 896 354	759 952
SPHINCS$^+$-Haraka-192f-simple	732 770	21 433 286	1 205 698
SPHINCS$^+$-Haraka-192f-robust	998 446	30 866 288	1 799 300
SPHINCS$^+$-Haraka-256s-simple	28 822 310	451 164 660	696 980
SPHINCS$^+$-Haraka-256s-robust	40 954 800	677 039 436	1 046 096
SPHINCS$^+$-Haraka-256f-simple	1 809 078	41 973 226	1 252 598
SPHINCS$^+$-Haraka-256f-robust	2 599 368	61 706 762	1 854 540

Table 5: Runtime benchmarks for SPHINCS$^+$-Haraka on AES-NI

	key generation	signing	verification
SPHINCS$^+$-SHA-256-128s-simple	84 964 790	644 740 090	861 478
SPHINCS$^+$-SHA-256-128s-robust	175 257 460	1 328 848 352	1 827 104
SPHINCS$^+$-SHA-256-128f-simple	1 334 220	33 651 546	2 150 290
SPHINCS$^+$-SHA-256-128f-robust	2 748 026	68 541 846	4 801 338
SPHINCS$^+$-SHA-256-192s-simple	125 310 788	1 246 378 060	1 444 030
SPHINCS$^+$-SHA-256-192s-robust	260 903 972	2 517 396 082	3 103 732
SPHINCS$^+$-SHA-256-192f-simple	1 928 970	55 320 742	3 492 210
SPHINCS$^+$-SHA-256-192f-robust	4 063 066	113 484 456	7 552 358
SPHINCS$^+$-SHA-256-256s-simple	80 943 202	1 025 721 040	1 986 974
SPHINCS$^+$-SHA-256-256s-robust	339 101 780	3 912 132 754	8 294 732
SPHINCS$^+$-SHA-256-256f-simple	5 067 546	109 104 452	3 559 052
SPHINCS$^+$-SHA-256-256f-robust	21 327 470	435 984 168	14 938 510

Table 6: Runtime benchmarks for SPHINCS$^+$-SHA-256 on AVX2

	key generation	signing	verification
SPHINCS$^+$-SHAKE256-128s-simple	143 900 796	1 102 470 520	1 189 102
SPHINCS$^+$-SHAKE256-128s-robust	274 483 474	2 076 548 104	2 408 782
SPHINCS$^+$-SHAKE256-128f-simple	2 249 444	56 933 788	3 346 068
SPHINCS$^+$-SHAKE256-128f-robust	4 272 402	106 032 762	6 677 094
SPHINCS$^+$-SHAKE256-192s-simple	206 105 502	1 910 461 606	1 653 314
SPHINCS$^+$-SHAKE256-192s-robust	397 548 084	3 549 895 260	3 300 788
SPHINCS$^+$-SHAKE256-192f-simple	3 220 902	89 875 552	4 783 424
SPHINCS$^+$-SHAKE256-192f-robust	6 175 702	167 173 520	9 330 848
SPHINCS$^+$-SHAKE256-256s-simple	136 190 230	1 650 717 926	2 559 892
SPHINCS$^+$-SHAKE256-256s-robust	258 430 892	2 982 404 428	4 669 406
SPHINCS$^+$-SHAKE256-256f-simple	8 535 534	176 951 378	5 030 988
SPHINCS$^+$-SHAKE256-256f-robust	16 296 098	329 696 258	9 716 888

Table 7: Runtime benchmarks for SPHINCS$^+$-SHAKE256 on AVX2

	public key size	secret key size	signature size
SPHINCS$^+$-128s	32	64	7 856
SPHINCS$^+$-128f	32	64	17 088
SPHINCS$^+$-192s	48	96	16 224
SPHINCS$^+$-192f	48	96	35 664
SPHINCS$^+$-256s	64	128	29 792
SPHINCS$^+$-256f	64	128	49 856

Table 8: Key and signature sizes in bytes

assumptions about the used hash function. A secure hash function is required by *any efficient signature scheme* that supports arbitrary input lengths.

Moreover, a collision attack against the hash function does not suffice to break the security of SPHINCS$^+$. We consider this an important feature given the successful collision attacks on MD5 and SHA1. Especially given that even for MD5 second-preimage resistance has not been broken, yet.

Finally, the cryptographic community has a good understanding of (exact) hash-function security, especially after the recent SHA3 competition. This is in contrast to the relatively new problems used in other areas of post-quantum cryptography. Even though some of those problems are known already for a long time, estimating the hardness of solving specific problem instances is far less understood.

Advantage: State-of-the-art attacks are easily analyzed. The most efficient attacks known against SPHINCS$^+$ are easy to state and analyze, such as searching for a hash input that has a particular pattern of output bits. The analogous quantum attacks are also easy to state and analyze, such as using Grover's algorithm to accelerate the same search. This allows precise quantification of the security levels provided by SPHINCS$^+$.

Advantage: Small key sizes. Another advantage of SPHINCS$^+$ is the small size of the keys, in particular the public-key size. In many applications public keys are transmitted frequently; almost as frequently as signatures. This is typically the case for certificates (or certificate chains) as used, for example, in TLS.

Advantage: Overlap with XMSS. One more feature of SPHINCS$^+$ is the large overlap with the stateful hash-based signature scheme XMSS. Especially the verification code of XMSS is almost entirely contained within the SPHINCS$^+$ verification code. Hence, in scenarios like virtual private networks where clients authenticate towards a gateway using signatures it is easy to combine these two. While every client that actually can support to handle a state can use XMSS, every other client can use SPHINCS$^+$. Only the gateway has to support verification of both, XMSS and SPHINCS$^+$ signatures. This becomes especially interesting as SPHINCS$^+$ is not particularly well suited for resource-constrained devices (although it was shown that it is in principle possible to implement SPHINCS$^+$ on such devices [10]). However, most resource-constrained devices can deal with a state and XMSS is far better suited for these devices.

Advantage: Reuse of established building blocks. SPHINCS$^+$ uses the basic hash function as building block many times. Any speedup to implementations of SHA-256, SHAKE256 or Haraka directly benefits the SPHINCS$^+$ speed. In particular hardware support for hash functions in the CPU, cryptographic coprocessors, or via instruction-set extensions instantly leads to faster SPHINCS$^+$ signatures (or to smaller SPHINCS$^+$ signatures via tuning w).

12. Acknowledgements

The authors would like to thank Wouter Boschmann, Andrew Bulychev, Lena Heimberger, and Hassan Mohseni for feedback on this proposal.

References

[1] Jean-Philippe Aumasson and Guillaume Endignoux. Clarifying the subset-resilience problem. Cryptology ePrint Archive, Report 2017/909, 2017. https://eprint.iacr.org/2017/909. 42

[2] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. Cryptology ePrint Archive, Report 2017/933, 2017. https://eprint.iacr.org/2017/933. 43

[3] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe. The SPHINCS$^+$ Signature Framework. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 2019. https://doi.org/10.1145/3319535.3363229. 45, 46

[4] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. Document ID: cd9faae9bd5308c440df50fc26a517b4, https://cr.yp.to/papers.html#cachetiming. 52

[5] 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 Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer Berlin Heidelberg, 2015. 5, 25, 41, 46

[6] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer-Verlag Berlin Heidelberg, 2006. http://www.jbonneau.com/AES_timing_full.pdf. 52

[7] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011. 5

[8] A. Hülsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018. 5

[9] Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013. 13, 45, 46

[10] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016*, volume 9614 of *LNCS*, pages 446–470, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 58

[11] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016*, volume 9614 of *LNCS*, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. 6, 9, 42, 45, 46, 47, 48, 49, 50

[12] Jérémy Jean. Cryptanalysis of Haraka. *IACR Trans. Symmetric Cryptol.*, 2016(1):1–12, 2016. 53

[13] Stefan Kölbl, Martin Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 – efficient short-input hashing for post-quantum applications. volume 2016, pages 1–29, 2017. 34

[14] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979. 5

[15] David McGrew, Michael Curcio, and Scott Fluhrer. Hash-based signatures. Internet Draft, IETF Crypto Forum Research Group, 2019. 6, 38, 44

[16] Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1990. 5

[17] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag Berlin Heidelberg, 2007. 52

[18] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. http://eprint.iacr.org/2005/271/. 52

[19] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995. 40

[20] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Batten and Jennifer Seberry, editors, *Information Security and Privacy 2002*, volume 2384 of *LNCS*, pages 1–47. Springer, 2002. 25

A. Parameter-evaluation Sage script

```
tsec,hashbytes = 125,16
#tsec,hashbytes = 192,24
#tsec,hashbytes = 253,32
maxsigs=2**64

F = RealField(100)

def ld(r):
  return -F(log(1/F(2**(8*hashbytes))+F(r)) / log2)

def pow(p,e):
  return F(p)**e

def qhitprob(qs,r):
  p = F(1/leaves)
  return binomial(qs,r)*(pow(p,r))*(pow(1-p,qs-r))

def la(m,w):
  return ceil(m / log(w,2))

def lb(m,w):
  return floor( log(la(m,w)*(w-1), 2) / log(w,2)) + 1

def lc(m,w):
  return la(m,w) + lb(m,w)


for h in range(35,74,2):
  leaves = 2**h
  for b in range(4,17):
    for k in range(30,32):
      sigma=0
      r = 1
      while True:
        r = F(r)
        p = min(1,F((r/F(2**b)))**k)
        q = qhitprob(maxsigs,long(r))*p
        sigma += q
        r += 1
        if(r > maxsigs/leaves and q < F(2)**(-10*tsec)): # beyond expected number of collisions and
          break
      if(sigma<2**-tsec):
        for d in range(4,h):
          if(h % d == 0 and h <= 64+(h/d)):
            for w in [16,256]:
              wots = lc(8*hashbytes,w)
              sigsize = ((b+1)*k+h+wots*d+1)*hashbytes
              if(sigsize < 50000):
                print h, # total tree height
                print d, # number of tree layers, subtree height is h/d
                print b, # height of FORS trees
```

```
print k, # number of trees for FORS
print w, # Winternitz parameter
print round(ld(sigma)),
print sigsize,
# Speed estimate based on (rough) hash count
print (k*2**(b+1) + d*(2**(h/d)*(wots*w+1)))
```