SPHINCS-alpha
Submission to the NIST post-quantum project

The SPHINCS-alpha Team

May 31, 2023

Contents

1 Introduction

Hash-based signature is one of the most promising candidates for (and perhaps the most conservative approach to) post-quantum digital signatures. An advantage of hash-based signatures is that its (classical as well as quantum) security strength is better understood (and easier to evaluate) than other candidates, by solely relying on the idealized hardness[1] of the cryptographic hash functions.

Stateful signatures. Ralph Merkle proposed a hash-based signature [20] that builds upon Lamport's one-time signature (OTS) [18]. The recent efforts towards improving stateful signatures lead to the eXtended Merkle Signature Scheme (XMSS) [16] and the Leighton-Micali Signature (LMS) [19], standardized by NIST [9] and IETF.

Stateless signatures. In a typical stateful signature, e.g., Merkle's signature scheme (MSS), the signer keeps track of which private key of the OTS has been used to avoid security issues from subsequent reuse. This is however not always possible in many practical scenarios. Goldreich proposed a stateless hash-based signature construction [12, 13] which removes the need for maintaining a local state but results in prohibitively large signatures. Recently, this line of research gets renewed interest. By incorporating the hypertree structure, SPHINCS [5] offered a practical instantiation of the Goldreich-style stateless hash-based signature. SPHINCS serves as a basis for subsequent works driven by the NIST PQC standardization process, including Gravity-SPHINCS [3], SPHINCS-Simpira [14] and SPHINCS$^+$ [6]. Among these schemes, SPHINCS$^+$ employs a new design framework based on few-time signature (FTS), and a new security analysis framework from "tweakable hash function". It is generally considered as the current state-of-the-art of stateless hash-based signatures and is to be standardized by NIST.

Table 1: Performance comparison between SPHINCS$^+$ and SPHINCS-α, with simple tweakable hash function instantiated with `shake`. Key generation, signing and verification time are in terms of CPU cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

	SPHINCS$^+$				SPHINCS-α				Relative Change			
Param.	KeyGen	Sign	Verify	Size	KeyGen	Sign	Verify	Size	KeyGen	Sign	Verify	Size
128f	1143558	26872236	2204802	17088	1036602	26635716	2028186	16720	−9.35%	−0.88%	−8.01%	−2.15%
192f	1662498	45405504	3003534	35664	2199276	45218790	1744038	34896	32.29%	−0.41%	−41.93%	−2.15%
256f	4327632	92059542	2967642	49856	4286574	91335474	3175290	49312	−0.95%	−0.79%	7.00%	−1.09%
128s	72597852	551233638	846486	7856	51421086	537033762	2689650	6880	−29.17%	−2.58%	217.74%	−12.42%
192s	105310692	1022229270	1201230	16224	78050718	988899534	3845970	14568	−25.89%	−3.26%	220.17%	−10.21%
256s	69033492	918473904	1701324	29792	52048332	764352612	6005448	27232	−24.60%	−16.78%	252.99%	−8.59%

Summary of our Work. We propose SPHINCS-α, a stateless hash-based signature scheme, which improves upon the state-of-the-art stateless hash-based signature scheme SPHINCS$^+$, while preserving the core elements that made the original SPHINCS$^+$ a standout project.

Our optimization mainly stems from the optimization of the one-time signature scheme, which we prove to have a size-optimal encoding scheme among all tree structured one-time signatures. Moreover, no DAG-based construction offers better size efficiency, making our proposal size-optimal among all known schemes to the best of our knowledge [27].

Since our modification upon the SPHINCS$^+$ scheme is limited to the encoding scheme of the underlying WOTS$^+$one-time signature, the security analysis of our new scheme remains largely the same as the original analysis. Based on the security analysis of the original work, we estimate the security level and carefully choose parameters in order to achieve optimized performance.

To facilitate a fair comparison, we implement SPHINCS-α by adapting the code of SPHINCS$^+$ and compare their performance on a desktop computer. As shown in Table 1, under the "small" series parameters (optimized towards small signature size) our scheme reduces signature size and running time by 8-12% and 2-16% respectively at all security levels while under the "fast" series (optimized towards fast signing operations) our scheme exhibits better performance in general.

We refer to more detailed comparisons for the full spectrum of parameter choices in Section 11. As summarized in Table 7, our scheme offers an overall performance improvement for most parameter settings, in terms of signing time and signature size. On the downside, we experience an up to 253%

[1]The design philosophy of symmetric primitives (including hash functions) is that they should only admit generic attacks, otherwise the design is considered to be flawed.

increase in verification time. Nevertheless, we argue that for specific scenarios where verification time is critical, we can re-tune the parameters towards fast verification.

Other Changes Made. Following the decisions made by NIST [21], we remove haraka and robust version from tweakable hash function.

Acknowledgement. We acknowledge the SPHINCS$^+$ team for their great work on the original SPHINCS$^+$ project. A significant portion of our code and documentation derives from their commendable efforts.

2 SPHINCS-α vs. SPHINCS$^+$

In this section we highlight our improvement upon the original SPHINCS$^+$ scheme, namely, the one-time signature component. Our optimized scheme, CS-WOTS$^+$, allows a larger message space / signature size ratio as compared to the original WOTS$^+$ scheme, allowing a smaller one-time signature size. As a result, the parameters of the hash-based signature scheme can thus be re-tuned to reduce the signature size without exacerbating the computation time.

We explain the details of our optimization as well as some technical proofs in this section.

2.1 The WOTS$^+$ Encoding

The reason that the WOTS$^+$(as well as other Winternitz-type OTS) scheme introduces the checksum is that in absence of the checksum the adversary can efficiently forge signatures given a single pair of valid message signature. That is, given (σ, m) he forges any m' satisfying $\forall i(m_i \leq m'_i)$ by computing $\mathbf{F}^{m'_i}(sk_i) = \mathbf{F}^{m'_i - m_i}\big(\mathbf{F}^{m_i}(sk_i)\big)$.

The checksum addresses the issue: an increase in any m_i leads to a decrease in at least one C_i (recall $C = \sum_{i=1}^{l_1}(w-1-m_i)$). Therefore, the adversary cannot forge any (m', C') satisfying both $\forall i(m_i \leq m'_i)$ and $\forall i(C_i \leq C'_i)$ at the same time.

2.2 Size-Optimal Encoding

More formally, the problem of constructing one-time signature reduces to that of building an efficient encoding scheme $\mathsf{Enc} : \mathcal{M} \to \mathcal{C} \subseteq [w]^l$ for some incomparable codeword set $\mathcal{C}$ (see Definition 1). In case of WOTS$^+$, the encoding function Enc simply appends the checksum to the original message. Note that WOTS$^+$ fixes the size of the message to l_1 (i.e., $\mathcal{M} = [w]^{l_1}$) and then constructs as small codewords as possible (minimizing $l - l_1$).

Definition 1 ((In)comparability)**.** *For $a, b \in [w]^l$, we denote by $a \leq b$ if for every $i \in [l]$ we have $a_{i+1} \leq b_{i+1}$. If $a \leq b$ or $b \leq a$ we say that a and b comparable, or otherwise a and b are incomparable. A set $S \subseteq \{a : a \in [w]^l\}$ is said to be incomparable (or called an "antichain" in order theory terminology) if any two elements of S are incomparable.*

We take a slightly different approach to encoding the messages. That is, we first fix the size of the codewords to l, $\mathcal{C} \subseteq [w]^l$, and strive to accommodate as large message space $\mathcal{M}$ as possible. Given that Enc is an injection it is essentially to maximize the size of $\mathcal{C} \subseteq [w]^l$. A natural approach is to encode the codewords such that all elements of every codeword sum to the same value, and therefore the checksum is not explicitly needed. Bos and Chaum [8] studied this approach for the special case of $w = 2$. Vaudenay [25] generalized it to arbitrary w, but he did not provide an encoding algorithm. Perin et al. provided a similar encoding algorithm in [22], but they did not present a size-optimal proof.

Theorem 2.1. *For any $m \in [l(w-1)+1]$, $\mathcal{C}_m \overset{\text{def}}{=} \{v \in [w]^l : \sum_{i=1}^{l} v_i = m\}$ is incomparable.*

Proof. Suppose towards contradiction that $\mathcal{C}_m$ (for some fixed $m \in [l(w-1)+1]$) is not incomparable, then there exist distinct $a, b \in \mathcal{C}_m$ s.t. $a \leq b$. There must be an index j such that $a_j < b_j$ (otherwise $a = b$). However, due to equal sum $\sum_i a_i = \sum_i b_i$ we have $\sum_{1 \leq i \leq l \wedge i \neq j}(a_i - b_i) > 0$, and there must exist some $1 \leq k \leq l$ such that $a_k > b_k$, which is a contradiction to $a \leq b$. □ □

Every $\mathcal{C}_m$ gives an encoding scheme but with different size. For $m = 0$ or $m = l(w-1)$, $\mathcal{C}_m$ consists of only a single codeword. We argue that the size of $\mathcal{C}_m$ reaches its maximal in the middle, i.e., when $m = \lfloor \frac{l(w-1)}{2} \rfloor$. One easily verifies that this holds in the binary case (i.e., $w = 2$) where $|\mathcal{C}_m| = \binom{l}{m}$. Perin et al. [22] proved that $|\mathcal{C}_m|$ reaches its maximum when $m = \lfloor \frac{l(w-1)}{2} \rfloor$. We prove a stronger optimality result in Theorem 2.2 that the size of $\mathcal{C}_m$, when $m = \lfloor \frac{l(w-1)}{2} \rfloor$, is not only the largest in all $\mathcal{C}_m$ for $m \in [l(w-1)+1]$ but the largest among all valid sets of codewords.

Theorem 2.2 (Size-optimal encoding)**.** *For every incomparable $\mathcal{C}^* \in P([w]^l)$, it holds that*

$$|\mathcal{C}^*| \le |\mathcal{C}_{\lfloor \frac{l(w-1)}{2} \rfloor}| \ .$$

We defer its proof to Theorem 2.4, which rephrases Theorem 2.2 in the language of order theory. Prior to that, we discuss how to compute $|\mathcal{C}_m|$ by recursion, and give an explicit construction of encoding messages into $\mathcal{C}_m$ for $m = \lfloor \frac{l(w-1)}{2} \rfloor$. Hereafter, we denote such $\mathcal{C}_m$ with maximal size by $\mathcal{C}$ for brevity.

Counting the size. Now we need to figure out the size of $\mathcal{C}$. As a special case, $|\mathcal{C}| = \binom{l}{\lfloor l/2 \rfloor}$ when $w = 2$. Fix w, let

$$D_{n,m} = |\{v \in [w]^n : \sum_{i=1}^{n} v_i = m\}| \ ,$$

we have their initial values

$$\begin{aligned} D_{1,m} &= 1, \text{ for } m \in \{0, 1, \dots, w-1\} \\ D_{n,m} &= 0, \text{ for } 2 \le n \in \mathbb{Z}, m \in \mathbb{Z}^- \ , \end{aligned}$$

and recurrence relation

$$D_{n,m} = \sum_{i=0}^{w-1} D_{n-1,m-i}, 2 \le n \in \mathbb{Z}, m \in \{0, 1, \dots, l(w-1)\} \ . \tag{1}$$

Note when $w = 2$, this method is equivalent to recurrence relation of binomial coefficient, i.e., $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$.

Let us explain the recurrence relation. To compute $D_{n,m}$, consider the value of its last summand, which could be any value in $\{0, 1, \dots, w-1\}$. If this value is set to i, the sum of the first $n-1$ elements must be $m-i$. Therefore, we notice that the problem "n elements with sum to m" into those "$n-1$ elements with sum to $m-i$". Thus we can simply count $D_{n,m}$ by accumulating $D_{n-1,m-i}$. Following this method, $D_{l,\lfloor l(w-1)/2 \rfloor}$ gives the size of $\mathcal{C}$.

Related works. $D_{n,m}$ is also the m-th coefficient of $(1 + x + x^2 + \cdots + x^{w-1})^n$. Euler [11] has studied $w = 3, 4, 5$, known as trinomial, quadrinomial and quintinomial coefficients respectively. The generalized form was studied in the literature, e.g., [1, 26, 4]. Actually, we can use an inclusion-exclusion argument to express it as a function of binomial coefficients [15]

$$D_{n,m} = \sum_{s=0}^{\lfloor m/w \rfloor} (-1)^s \binom{n}{s} \binom{m+n-sw-1}{n-1} \ .$$

Encoding algorithm. Now we make the construction explicit by giving an efficient encoding algorithm, which maps a message $x \in [|\mathcal{C}|]$ into an element in $\mathcal{C}$. We give the pseudocode of the encoding algorithm in Algorithm 1.

Let us explain the encoding algorithm. As previously stated, the problem can be divided into several subproblems by considering the value of the first element v_{l-i}. To encode a natural number $x \in [0, D_{i,m})$, we can simply determine $v_{l-i} = j$ by seeking which j satisfies $x \in [\sum_{k<j} D_{i-1,m-k}, \sum_{k \le j} D_{i-1,m-k})$. Once the value of v_{l-i} is determined, we proceed to the next terms until all elements are decided.

In order to prove the optimality of the encoding, we need some prerequisites about the order theory. The relationship between one-time signature and order theory has been investigated in [7].

Preliminaries of Order Theory

Definition 2 (Poset)**.** *A poset $(\mathcal{S},\le)$ consists of a set $\mathcal{S}$ together with an antisymmetric, transitive and reflexive binary relation '$\le$', where are certain pairs $(x,y) \in \mathcal{S}$ are comparable ($x \le y$ or $y \le x$).*

Algorithm 1: Encode:$[|\mathcal{C}|] \to \mathcal{C}$.

```
Function Encode(x)
    Let $v$ be an array of size $l$;
    $m \leftarrow \lfloor l(w-1)/2 \rfloor$;
    for $i \leftarrow l \ldots 1$ do
        for $j \leftarrow 0 \ldots \min(w-1, m)$ do
            if $x \geq D_{i-1,s-j}$ then
                $x \leftarrow x - D_{i-1,s-j}$;
            else
                $v_{l-i} \leftarrow j$;
                break;
        $m \leftarrow m - v_{l-i}$;
    return $v$;
```

Note that a poset does not require all pairs in $\mathcal{S}$ to be comparable, and thus it is also known as a partially ordered set.

Definition 3 ((Anti)chain and decomposition)**.** *A chain (resp., antichain) refers to a subset of a poset, whose every pair of elements is comparable (resp., incomparable). A chain decomposition is a partition of a poset into disjoint chains.*

Theorem 2.3 (Dilworth's theorem [10])**.** *For any finite poset P, the size of P's maximum antichain equals the size of a minimum chain decomposition of P.*

Let $P = ([w]^l, \leq)$ be a finite poset. According to Dilworth's theorem, we can prove that $\mathcal{C}$ is the maximum antichain of P by arguing that (1) $\mathcal{C}$ is an antichain and (2) we can find a chain decomposition whose size equals to $|\mathcal{C}|$.

Theorem 2.4. *$\mathcal{C}$ is the maximum antichain, i.e., $\mathcal{C}$ is size-optimal.*

Proof. We have proved that $\mathcal{C}$ is an antichain in Theorem 2.1. It remains to construct the chain decomposition of size $|\mathcal{C}|$ as follows. Our proof can be viewed as a generalization of the proof of Sperner's theorem [24], which considers the special case for $w = 2$.

Consider poset $S_n = ([w]^n, \leq)$, and we sometimes denote an element of S_n by $(a_1, ..., a_n) \in [w]^n$ or by $c_i \in [w]^n$. We slightly abuse the notation $|(a_1, ..., a_n)| \overset{\text{def}}{=} a_1 + ... + a_n$.

We construct the chain decomposition for S_n by induction, where every chain $\{c_1, \ldots, c_m\}$ satisfies the following two properties:

- $|c_{i+1}| = |c_i| + 1, \forall i \in \{1, 2, \ldots, m-1\}$,
- $|c_1| + |c_m| = n \cdot (w-1)$.

The case for $n = 1$ is trivial, i.e., $D_{1,\lfloor (w-1)/2 \rfloor} = 1$, which correspond to the chain of $S_1 = \{(0), (1), \ldots, (w-1)\}$.

Assume that we have a chain decomposition for S_{n-1} satisfying the above two properties, we proceed to the construction of a chain decomposition for S_n. For each chain $c = \{c_1, c_2, \ldots, c_m\}$ (from the chain decomposition of S_{n-1}) satisfying the two properties, we build $k+1$ chains for S_n as follows, where $k = \min(w-1, m-1)$. That is, for every $j \in \{0, ..., k\}$ the j-th chain consists of:

$$(c_1, j) \leq \ldots \leq (c_{m-j}, j) \leq (c_{m-j}, j+1) \leq ... \leq (c_{m-j}, w-1) \enspace .$$

This yields the $k+1$ chains as shown in Fig. 1 below:

It is easy to verify that $|(c_1, j)| + |(c_{m-j}, w-1)| = |(c_1, 0)| + j + |(c_m, 0)| - j + (w-1) = n(w-1)$, and every subsequent element increase the sum value of its predecessor by one. Namely, the two properties are preserved for all the constructed chains of S_n.

It remains to argue that all the chains constructed (from the decomposed chains of S_{n-1}) constitute a partition of $[w]^n$. That is, for every $c_i \in S_{n-1}$, each of its augmented elements $(c_i, 0), ..., (c_i, w-1)$

$$\begin{array}{ccccccc}
(c_1,0) & \dots & \dots & \dots & (c_m,0) & \dots & (c_m,w-1) \\
\vdots & \dots & \dots & \iddots & \dots & \dots & \vdots \\
(c_1,k) & \dots & (c_{m-k},k) & \dots & \dots & \dots & (c_{m-k},w-1)
\end{array}$$

Figure 1: A demonstration of how a chain from S_{n-1} is expanded into $k+1$ chains for S_n, where every row is an expanded chain. Note that it is not a rectangular matrix (every row has two less elements than the previous).

Table 2: Comparison of length l between WOTS$^+$ and CS-WOTS$^+$ for different values of Winternitz parameters w and security parameter λ.

	128bit		192bit		256bit	
w	WOTS$^+$	Ours	WOTS$^+$	Ours	WOTS$^+$	Ours
8	46	45	67	66	90	88
16	35	34	51	50	67	66
24	31	30	45	44	59	58
32	28	27	42	40	55	53
40	27	26	39	38	52	50
48	25	25	37	36	48	48

appears in the constructed chains exactly once. Note that every c_i belongs to (and can only belong to) one of the decomposed chains of S_{n-1}, say $c = \{c_1, \dots, c_m\}$. We discuss the following cases.

Case $m \le w$: We have $k = m-1 \le w-1$. $[(c_i,0), \dots, (c_i, k+1-i)]$ appears as the first $(k+2-i)$ elements of the i-th column, and then $[(c_i, k+1-i), \dots, (c_i, w-1)]^T$ as the last $(w+i-k-1)$ elements of the $(k+2-i)$-th row in Fig. 1.

Case $m > w$: We have $k = w-1 < m-1$. If $1 \le i \le m-w+1$, then $[(c_i,0),(c_i,w-1)]$ appears as the i-th column in Fig. 1. Otherwise, $m-w+1 < i \le m$. $[(c_i,0), \dots, (c_i, m-i)]$ and $[(c_i, m-i), \dots, (c_i, w-1)]^T$ are the first $m-i+1$ elements of the i-th column, and the last $(w+i-m)$ elements of the $(m-i+1)$-th row respectively.

Therefore, we have shown that for every $a \in [w]^{n-1}$, $(a,0)$, …,$(a,w-1)$ appears exactly once in the newly constructed chains, namely, the chains constitutes as a chain decomposition for S_n. Finally, it remains to count the number of chains in the decomposition. The two properties guarantee that every chain contains exactly one element c_{mid} with $|c_{\mathsf{mid}}| = \lfloor l(w-1)/2 \rfloor$ (i.e., $c_{\mathsf{mid}} \in \mathcal{C}$). Thus, the size of chain decomposition $|\mathcal{C}| = D_{l,\lfloor l(w-1)/2 \rfloor}$. This completes the proof that $\mathcal{C}$ is the maximum antichain. $\square$

2.3 Theoretical Performance

Constant Sum WOTS$^+$ has two advantages over WOTS$^+$.

- Stable computing time. The number of hash function calls is fixed in our construction, in contrast to possibly variable numbers for the signing and verification algorithm of WOTS$^+$. While no timing attacks are identified against the implementations of our construction and WOTS$^+$, stable computing time is always preferable (especially for signing algorithms whose computation involves a private key).

- Reduced signature size and number of hash calls. For instance, the SPHINCS$^+$-256s parameter set suggests $w = 16$ and $l = 67$. In our construction, for $w = 16$ we require $l = 66$, which reduces 1.5% in both running time (in terms of the expected number of hash function calls) and size. We refer to Table 2 for more details.

Although our encoding algorithm costs slightly more than the checksum method, it is less dominant compared to the number of hash function calls used in the signature scheme, which will be confirmed in the performance comparison to SPHINCS$^+$.

3 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.

3.1 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.

3.2 Functions

We define the following functions:

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

$\lfloor x \rfloor$ (or $\mathsf{floor}(x)$) 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.

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

3.3 Operations

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., $\mathsf{floor}(a/2^b)$.

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 $X \| Y = X[0] ... X[x-1] Y[0] ... Y[y-1]$.

3.4 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.

3.5 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`.

3.6 Cryptographic (Hash) Function Families

3.6.1 Tweakable Hash Functions (Functions `T`, `F`, `H`)

Following SPHINCS$^+$, a tweakable hash function takes a public seed **PK**`.seed` and an address `ADRS` in addition to the message input. This effectively renders hash function calls for each key pair and position in the virtual tree independent. The addressing scheme will be described in Section 3.6.3.

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

$$\begin{aligned} \mathbf{T}_\ell &: \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n} \rightarrow \mathbb{B}^n \\ \texttt{md} &\leftarrow \mathbf{T}_\ell(\mathbf{PK}.\texttt{seed}, \texttt{ADRS}, M) \ , \end{aligned}$$

mapping and ℓn-byte message M to an n-byte hash value md using an n-byte seed **PK**`.seed` and a 32-byte address `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:

$$\begin{aligned} &\mathbf{F} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n \rightarrow \mathbb{B}^n \\ &\mathbf{F} \overset{\text{def}}{=} \mathbf{T}_1 \\ &\mathbf{H} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n} \rightarrow \mathbb{B}^n \\ &\mathbf{H} \overset{\text{def}}{=} \mathbf{T}_2 \end{aligned}$$

3.6.2 PRF and Message Digest (Functions `PRF`, `PRF_msg`, `H_msg`)

SPHINCS-α uses a pseudorandom function **PRF** for pseudorandom key generation:

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

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

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

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

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

3.6.3 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 CS-WOTS$^+$ schemes, one is used for compression of the CS-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 CS-WOTS$^+$ hash address, to 1 for the compression of the CS-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 CS-WOTS$^+$ address (Fig. 2). In this case, the type word is followed by the key pair address that encodes the index of the CS-WOTS$^+$ key pair within the specified tree. The next word encodes the chain address (i.e. the index of the chain within CS-WOTS$^+$), followed by a word that encodes the address of the hash function call within the chain. Note that for the generation of the secret keys based on **SK**.`seed` a different type of address is used (see below).

Layer Address	Tree Address		
type = 0	Key Pair Address	Chain Address	Hash Address

Figure 2: CS-WOTS$^+$ hash address.

The second type (Fig. 3) is used to compress the CS-WOTS$^+$ public keys. The type word is set to 1. Similar to the address used within CS-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.

Layer Address	Tree Address	
type = 1	Key Pair Address	Padding = 0

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

The third type (Fig. 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.

Layer Address	Tree Address		
type = 2	Padding = 0	Tree Height	Tree Index

Figure 4: hash tree address.

The next type (Fig. 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 CS-WOTS$^+$ hash addresses. Its value is the same as that of the CS-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. To generate the leaf nodes from **SK**.`seed` a different type of address is used (see below).

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

The final two types are used for secret key value generation in CS-WOTS$^+$ and FORS. A CS-WOTS$^+$ key generation address (Fig. 7) is the same as a CS-WOTS$^+$ hash address with two differences. First, the

Layer Address	Tree Address		
type = 3	Key Pair Address	Tree Height	Tree Index

Figure 5: FORS tree address.

Layer Address	Tree Address		
type = 4	Key Pair Address	Padding = 0	

Figure 6: FORS tree roots compression address.

type word is set to 5. Second, the hash address word is constantly set to 0. When generating the secret key value for a given chain, the remaining words have to be set the same way as for the CS-WOTS$^+$ hash addresses used for this chain.

Layer Address	Tree Address		
type = 5	Key Pair Address	Chain Address	Hash Address = 0

Figure 7: CS-WOTS$^+$ key generation address.

Similarly, the FORS key generation type (Fig. 8) is the same as the FORS tree address type, except that the type word is set to 6, and the tree height word is set to 0. As for the CS-WOTS$^+$ key generation address, the remaining words have to be set as for the FORS tree address used when processing the generated value.

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`, `FORS_ROOTS`, `WOTS_PRF`, and `FORS_PRF`.

4 CS-WOTS$^+$ One-Time Signatures

This section describes the CS-WOTS$^+$ 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.

4.1 CS-WOTS$^+$ Parameters

CS-WOTS$^+$ uses the parameters n, w and `len`; They are 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;
- `len`: the number of n-byte-string elements in a CS-WOTS$^+$ private key, public key, and signature.

4.2 CS-WOTS$^+$ Chaining Function (Function `chain`)

The chaining function (Algorithm 2) computes an iteration of **F** on an n-byte input using a CS-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**.

Layer Address	Tree Address		
type = 6	Key Pair Address	Tree Height = 0	Tree Index

Figure 8: FORS key generation address.

In the following, `ADRS` is a 32-byte CS-WOTS$^+$ hash address as specified in Section 3.6.3 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.

Algorithm 2: `chain` — Chaining function used in CS-WOTS$^+$.

```
# 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;
}
```

4.3 CS-WOTS$^+$ Private Key (Function `wots_SKgen`)

The CS-WOTS$^+$ private key, denoted by **SK**, 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 CS-WOTS$^+$ private key is derived from a secret seed **SK**.`seed` which is part of the SPHINCS-α secret key and a CS-WOTS$^+$ key generation address `skADRS` using **PRF**. The same secret seed is used to generate all secret key values within SPHINCS-α. 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 CS-WOTS$^+$ instance within the SPHINCS-α structure.

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

4.4 CS-WOTS$^+$ Public Key Generation (Function `wots_PKgen`)

A CS-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 CS-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 CS-WOTS$^+$ key pair within the SPHINCS-α structure. Hence, a CS-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**.

Algorithm 3: `wots_SKgen` — Generating a CS-WOTS$^+$ private key.

```
# Input: secret seed SK.seed, address ADRS
# Output: CS-WOTS+ private key sk
wots_SKgen(SK.seed, ADRS) {
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
    skADRS.setChainAddress(i);
    skADRS.setHashAddress(0);
    sk[i] = PRF(SK.seed, skADRS);
  }
  return sk;
}
```

Algorithm 4: `wots_PKgen` — Generating a CS-WOTS$^+$ public key.

```
# Input: secret seed SK.seed, address ADRS, public seed PK.seed
# Output: CS-WOTS+ public key pk
wots_PKgen(SK.seed, PK.seed, ADRS) {
  wotspkADRS = ADRS; // copy address to create OTS public key address
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
  skADRS.setChainAddress(i);
    skADRS.setHashAddress(0);
    sk[i] = PRF(SK.seed, skADRS);
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    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;
}
```

4.5 CS-WOTS$^+$ Encoding (Function `wots_encode`)

Let us explain the encoding algorithm (Algorithm 5). The problem can be divided into several subproblems by considering the value of the first element c_{l-i}. To encode a natural number $x \in [0, D_{i,m})$, we can simply determine $c_{l-i} = j$ by seeking which j satisfies $x \in [\sum_{k<j} D_{i-1,m-k}, \sum_{k\leq j} D_{i-1,m-k})$. Once the value of v_{l-i} is determined, we proceed to its next terms until all elements are decided. All arithmetic operations MUST be compute in unsigned 256-bit. Array D can be precomputed to avoid computing D for each encoding function call.

Algorithm 5: `wots_encode` — Encoding message to constant-sum encoding.

```
#Input: Message M
#Output: Constant-Sum Encoding of M
wots_encode(M) {

  s=floor(len*(w-1)/2)
  Let D be an 2d array of size (len+1)*(s+1) initialized to 0

  D[0][0]=1
  for( i = 1 ; i <= len ; i++ ){
    for( j = 0 j <= s ; j++ ){
      for( k = 0 ; k < w and k <= j ; k++ ){
        D[i][j] = D[i][j] + D[i-1][j-k]
      }
    }
  }
  // array D can be precomputed

  Let c be an array of size len
  for( i = len ; i >= 1 ; i-- ){
    for( j = 0 ; j < w and j <= s  ; j++ ){
      if( M >= D[i-1][s-j] ){
        M = M - D[i-1][s-j]
      } else {
        c[len-i] = j
        break
      }
    }
    s = s - c[len-i]
  }

  return c;
}
```

4.6 CS-WOTS$^+$ Signature Generation (Function `wots_sign`)

A CS-WOTS$^+$ signature is a length `len` array of n-byte strings. The CS-WOTS$^+$ signature is first generated by mapping a message M to `len` integers between 0 and $w-1$, and the sum of them is equals to $\lfloor \texttt{len} \cdot (w-1)/2 \rfloor$. A CS-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 CS-WOTS$^+$ key pair within a greater structure. Hence, a CS-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 CS-WOTS$^+$ signature `SIG` is shown below (Algorithm 6).

The data format for a signature is given in Fig. 9.

Algorithm 6: `wots_sign` — Generating a CS-WOTS$^+$ signature on a message M.

```
# Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
# Output: CS-WOTS+ signature sig
wots_sign(M, SK.seed, PK.seed, ADRS) {
  msg = wots_encode(M);
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
    skADRS.setChainAddress(i);
    skADRS.setHashAddress(0);
    sk = PRF(SK.seed, skADRS);
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sig[i] = chain(sk, 0, msg[i], PK.seed, ADRS);
  }
  return sig;
}
```

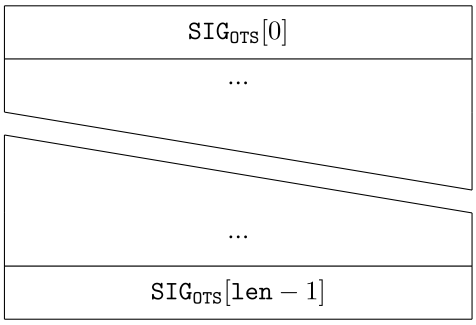

Figure 9: CS-WOTS$^+$ Signature Data Format.

4.7 CS-WOTS$^+$ Compute Public Key from Signature (Function `wots_pkFromSig`)

SPHINCS-α uses implicit signature verification for CS-WOTS$^+$. In order to verify a CS-WOTS$^+$ signature `SIG` on a message M, the verifier computes a CS-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 7. 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-α the output value is verified by using it to compute a SPHINCS-α public key.

A CS-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 CS-WOTS$^+$ key pair within the SPHINCS-α structure. Hence, a CS-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.

5 The SPHINCS-α Hypertree

In this section, we explain how the SPHINCS-α hypertree is built. We first explain how CS-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}).

Algorithm 7: `wots_pkFromSig` — Computing a CS-WOTS$^+$ public key from a message and its signature.

```
# Input: Message M, CS-WOTS+ signature sig, address ADRS, public seed PK.seed
# Output: CS-WOTS+ public key pk_sig derived from sig
wots_pkFromSig(sig, M, PK.seed, ADRS) {
  wotspkADRS = ADRS;
  msg = wots_encode(M)
  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;
}
```

5.1 (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 $2^{h'}$ CS-WOTS$^+$ public keys using a binary tree of height h'. Hence, an XMSS key pair for height h' can be used to sign 2 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 CS-WOTS$^+$ public keys. The XMSS public key is the root node of the tree. In SPHINCS-α, the XMSS secret key is the single secret seed that is used to generate all CS-WOTS$^+$ secret keys.

An XMSS signature in the context of SPHINCS-α consists of the CS-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 CS-WOTS$^+$ signature. A verifier computes the root value and verifies its correctness. A standalone XMSS signature also contains the index of the used CS-WOTS$^+$ key pair. In the context of SPHINCS-α this is not necessary as the SPHINCS-α signature allows to compute the index for each XMSS signature contained.

5.1.1 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 CS-WOTS$^+$ in the previous Section.

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

XMSS parameters are implicitly included in algorithm inputs as needed.

5.1.2 XMSS Private Key

In the context of SPHINCS-α, an XMSS private key is the single secret seed **SK**.`seed` contained in the SPHINCS-α secret key. It is used to generate the CS-WOTS$^+$ secret keys within the structure of an XMSS key pair as described in Section 4.

5.1.3 TreeHash (Function `treehash`)

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine `treehash` (Algorithm 8) 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 CS-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.

Algorithm 8: `treehash` — The TreeHash algorithm.

```
# 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();
}
```

5.1.4 XMSS Public Key Generation (Function `xmss_PKgen`)

The XMSS public key is computed as described in `xmss_PKgen` (Algorithm 9). In the context of SPHINCS-α 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-α structure.

Algorithm 9: `xmss_PKgen` — Generating an XMSS public key.

```
# 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;
}
```

5.1.5 XMSS Signature

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

- a CS-WOTS$^+$ signature $\texttt{SIG}_{\texttt{OTS}}$ taking $\texttt{len} \cdot n$ bytes,
- the authentication path `AUTH` for the leaf associated with the used CS-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 CS-WOTS$^+$ public key. A node N is addressed by its position in the tree. $N(x, y)$ denotes the y-th 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 i-th CS-WOTS$^+$ key pair, counting from zero, the j-th authentication path node is

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

The computation of the authentication path is discussed in Section 5.1.6. The data format for a signature is given in Fig. 10.

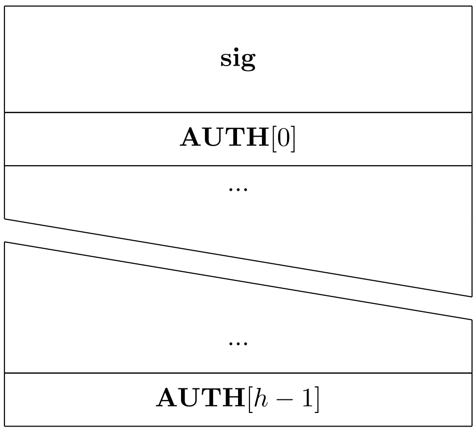

Figure 10: XMSS Signature

5.1.6 XMSS Signature Generation (Function `xmss_sign`)

To compute the XMSS signature of a message M in the context of SPHINCS-α, the secret seed **SK**.`seed`, the public seed **PK**.`seed`, the index `idx` of the CS-WOTS$^+$ key pair to be used, and the address `ADRS` of the XMSS instance are needed. First, a CS-WOTS$^+$ signature of the message digest is computed using the CS-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 8). 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.

5.1.7 XMSS Compute Public Key from Signature (Function `xmss_pkFromSig`)

SPHINCS-α 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 $\texttt{SIG}_{\texttt{XMSS}}$, 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-α key pair. First, `wots_pkFromSig` is used to compute a candidate CS-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 11.

5.2 HT: The Hypertree

The SPHINCS-α hypertree HT is a variant of XMSS$^{\mathrm{MT}}$. 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

Algorithm 10: `xmss_sign` — Generating an XMSS signature.

```
# 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 11: `xmss_pkFromSig` — Computing an XMSS public key from an XMSS signature.

```
# 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];
}
```

below. Trees on the lowest layer are used to sign the actual messages, which are FORS public keys in SPHINCS-α. 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.

5.2.1 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.

5.2.2 HT Key Generation (Function `ht_PKgen`)

The HT private key is the secret seed **SK**.`seed` which is used to generate all the CS-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.

Algorithm 12: `ht_PKgen` — Generating an HT public key.

```
# 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;
}
```

5.2.3 HT Signature

A HT signature $\texttt{SIG}_{\texttt{HT}}$ 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 Fig. 11.

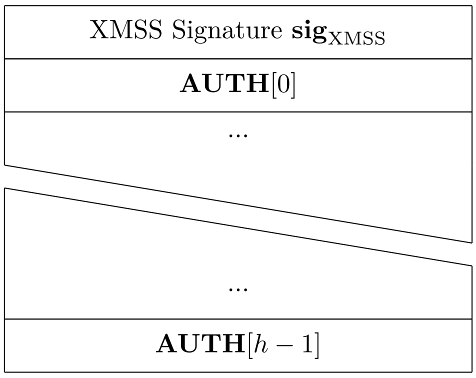

Figure 11: HT Signature.

5.2.4 HT Signature Generation (Function `ht_sign`)

To compute a HT signature $\texttt{SIG}_{\texttt{HT}}$ of a message M using, `ht_sign` (Algorithm 13) described below uses `xmss_sign` as defined in Section 5.1.6. The algorithm `ht_sign` takes as input a message M, a private seed **SK**.`seed`, a public seed **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 **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.

Algorithm 13: `ht_sign` — Generating an HT signature.

```
# 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;
}
```

5.2.5 HT Signature Verification (Function `ht_verify`)

HT signature verification (Algorithm 14) 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 $\texttt{SIG}_{\texttt{HT}}$, a public seed **PK**.seed, an index `idx` (split into a tree index and a leaf index, as above), and a HT public key $\mathbf{PK}_{\texttt{HT}}$.

6 FORS: Forest Of Random Subsets

The SPHINCS-α 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, 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 [23]. 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

Algorithm 14: `ht_verify` — Verifying a HT signature $\mathbf{SIG_{HT}}$ on a message M using a HT public key $\mathbf{PK}_{HT}$.

```
# 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;
  }
}
```

be a multiple of eight. The private key consists of kt random n- byte strings grouped into k sets, each containing tn-byte strings. The private key values are pseudorandomly generated from the main private seed **SK**.`seed` in the SPHINCS-α private key. In SPHINCS-α, 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 i-th 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-α 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.

6.1 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 $k \log t$.

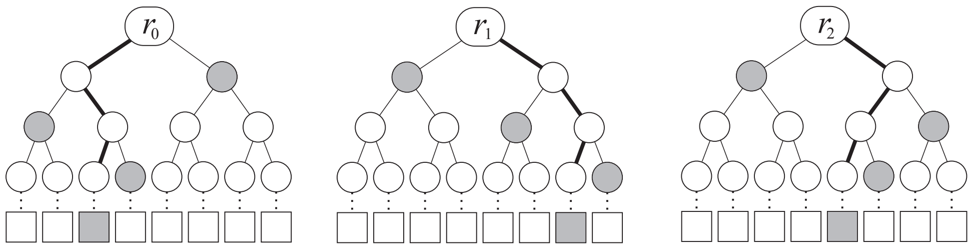

Figure 12: FORS trees and PK.

6.2 FORS Private Key (Function `fors_SKgen`)

In the context of SPHINCS-α, a FORS private key is the single private seed **SK**.`seed` contained in the SPHINCS-α private key. It is used to generate the kt n-byte private key values using **PRF** with a FORS key generation 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 i-th set is then stored at **SK**$[it + j]$. To generate one of these elements, a FORS key generation address skADRS is used, that encodes the position of the FORS key pair within SPHINCS-α and has tree height set to 0 and leaf index set to $it + j$:

Algorithm 15: `fors_SKgen` — Computing a FORS private key value.

```
#Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
#Output: FORS private key sk
fors_SKgen(SK.seed, ADRS, idx) {
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(FORS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());

  skADRS.setTreeHeight(0);
  skADRS.setTreeIndex(idx);
  sk = PRF(SK.seed, skADRS);
  return sk;
}
```

6.3 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 8) in Section 5.1. 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.

Algorithm 16: The `fors_treehash algorithm`.

```
# 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++ ) {
    sk = fors_SKgen(SK.seed, ADRS, s+i)
    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();
}
```

6.4 FORS Public Key (Function `fors_PKgen`)

In the context of SPHINCS-α, 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-α. It outputs a FORS public key.

Algorithm 17: `fors_PKgen` — Generate 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;
}
```

6.5 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-α. It outputs a FORS signature $\texttt{SIG}_{\texttt{FORS}}$.

Algorithm 18: `fors_sign` — Generate a FORS signature on string M.

```
# 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
    SIG_FORS = SIG_FORS  fors_SKgen(SK.seed, ADRS, i*t + idx) ;
    // 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;
}
```

The data format for a signature is given in Fig. 13.

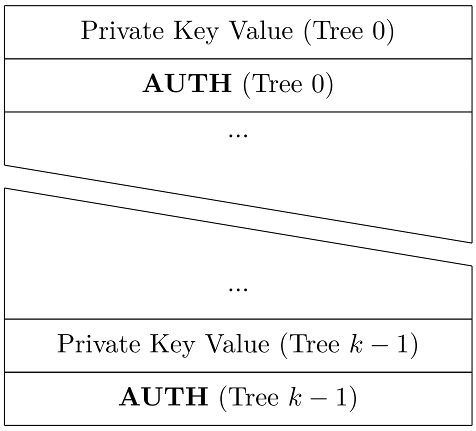

Figure 13: FORS Signature.

6.6 FORS Compute Public Key from Signature (Function `fors_pkFromSig`)

SPHINCS-α 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 $\texttt{SIG}_{\texttt{FORS}}$, a public seed **PK**.`seed`, and an address `ADRS`. The latter encodes the position of the FORS instance within the virtual structure of the SPHINCS-α 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 T_k. The algorithm `fors_pkFromSig` is given as Algorithm 19. The method `fors_pkFromSig` makes use of functions $\texttt{SIG}_{\texttt{FORS}}$`.getSK(i)` and $\texttt{SIG}_{\texttt{FORS}}$`.getAUTH(i)`. The former returns the i-th secret key element stored in the signature, the latter returns the i-th authentication path stored in the signature.

Algorithm 19: `fors_pkFromSig` — Compute a FORS public key from a FORS 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;
}
```

7 SPHINCS-α

We now have all ingredients to describe our main construction SPHINCS-α. Essentially, SPHINCS-α follows the fundamental structure of SPHINCS-α with the only difference being the one-time signature component.

7.1 SPHINCS-α Parameters

SPHINCS-α has the following parameters:

- n: the security parameter in bytes.
- w: the Winternitz parameter as defined in Section 4.1.
- h: the height of the hypertree as defined in Section 5.2.1.
- d: the number of layers in the hypertree as defined in Section 5.2.1.
- k: the number of trees in FORS as defined in Section 6.1.
- t: the number of leaves of a FORS tree as defined in Section 6.1.

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 CS-WOTS$^+$ private key, public key, and signature. The value `len` is chosen so that the size of the following poset is larger than the desired message space $\{0,1\}^n$ (cf. Section 2).

  $$\left| D_{\mathtt{len}, \lfloor \frac{\mathtt{len} \cdot (w-1)}{2} \rfloor} \right| \geq 2^n \ .$$

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

7.2 SPHINCS-α Key Generation (Function `spx_keygen`)

The SPHINCS-α private key contains two elements. First, the n-byte secret seed **SK**.`seed` which is used to generate all the CS-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-α 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 `spa_sign` does not get the public key, but needs access to **PK**.`seed` (and possibly to **PK**.`root` for fault attack mitigation), the SPHINCS-α secret key contains a copy of the public key.

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

The format of a SPHINCS-α private and public key is given in Fig. 14.

7.3 SPHINCS-α Signature

A SPHINCS-α signature $\mathtt{SIG}_{\mathtt{HT}}$ is a byte string of length $(1 + k \cdot (a+1) + h + d \cdot \mathtt{len})n$. It consists of an n-byte randomization string `R`, a FORS signature $\mathtt{SIG}_{\mathtt{FORS}}$ consisting of $k(a+1)n$-byte strings, and a HT signature $\mathtt{SIG}_{\mathtt{HT}}$ of $(h + d \cdot \mathtt{len})n$ bytes.

The data format for a signature is given in Fig. 15.

Algorithm 20: `spa_keygen` — Generate a SPHINCS-α key pair.

```
# Input: (none)
# Output: SPHINCS-α key pair (SK,PK)
spa_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) );
}
```

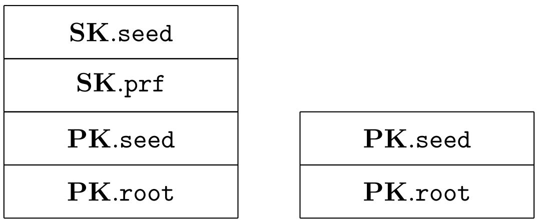

Figure 14: Left: SPHINCS-α secret key. Right: SPHINCS-α public key.

7.4 SPHINCS-α Signature Generation (Function `spa_sign`)

Generating a SPHINCS-α signature consists of four steps. First, a random value `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.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 `R`, the PRF takes a n-byte string opt which is initialized with **PK**.`seed` but can be overwritten with randomness if the global variable `RANDOMIZE` is set. This option is given as otherwise SPHINCS-α signatures would be always deterministic.

7.5 SPHINCS-α Signature Verification (Function `spa_verify`)

SPHINCS-α signature verification (Algorithm 22) 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-α signature verification takes a message M, a signature `SIG`, and a SPHINCS-α public key **PK**.

8 Instantiations

This section discusses instantiations for SPHINCS-α. SPHINCS-α 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-α parameters described 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-α. These instantiation are indeed three different signature schemes. We propose SPHINCS-α-SHAKE and

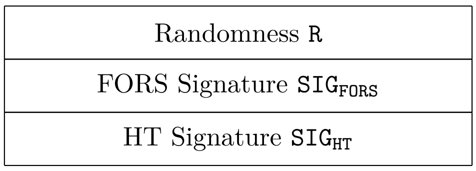

Figure 15: SPHINCS-α signature.

SPHINCS-α-SHA2, which utilize the cryptographic hash functions defined in FIPS PUB 202, respectively FIPS PUB 180, to instantiate the cryptographic function families.

8.1 SPHINCS-α Parameter Sets

SPHINCS-α 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 len are computed as

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

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

- `len`: the number of n-byte string elements in a CS-WOTS$^+$ private key, public key, and signature.

$$\texttt{len} = \min \left\{ \ell : \left| D_{\ell, \lfloor \frac{\ell \cdot (w-1)}{2} \rfloor} \right| \geq 2^n \right\} \quad ,$$

where the size of $D_{\ell, \lfloor \frac{\ell \cdot (w-1)}{2} \rfloor}$ is evaluated using the recurrence relation in Eq. (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 $\mathbf{H}_{\mathbf{msg}}$. Therefore, it largely determines which security level a parameter set reaches. It is also the size of virtually any node within the SPHINCS-α 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 CS-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 CS-WOTS$^+$ signature. Thereby it largely influences the size of a SPHINCS-α 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 CS-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-α 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-α 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

Algorithm 21: `spa_sign` — Generating a SPHINCS-α signature.

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS-α signature SIG
spa_sign(M, SK){ // init
ADRS = toByte(0, 32);
  // generate randomizer
  opt = PK.seed;
  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, md, 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;
}
```

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 CS-WOTS$^+$ signatures within a HT and thereby a SPHINCS-α 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 $\mathbf{H}_{\mathbf{msg}}$ 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 len of chains in a CS-WOTS$^+$ key pair determines the CS-WOTS$^+$ signature size.

8.1.1 Influence of Parameters on Security and Performance

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

Key Generation. Generating the SPHINCS-α private key and **PK**.`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 $2^{h/d}$ CS-WOTS$^+$ key generations (`len` calls to PRF for generating the **SK** and w`len` calls to **F** for the

Algorithm 22: `spa_verify` — Verify a SPHINCS-α signature **SIG** on a message M using a SPHINCS-α public key **PK**.

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean
spa_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);
}
```

PK) and we have to compress the CS-WOTS$^+$ public key (one call to $\mathbf{T}_{\texttt{len}}$). Computing the root of the top tree requires $(2^{h/d} - 1)$ calls to **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 **PRF** and **F**. Further, we have to compute the root of k binary trees of height $\log t$ which adds $k(t-1)$ calls to **H**. Finally, we need one call to $\mathbf{T}_k$. 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 len calls to **PRF** and $w\texttt{len}$ calls to **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 **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 **F** (to compute the leaf nodes from the signature elements), $k \log t$ calls to **H** (to compute the root nodes using the leaf nodes and the authentication paths), and one call to $\mathbf{T}_k$ for hashing the roots. Next, we have to verify d XMSS signatures which takes $< w\texttt{len}$ calls to **F** and one call to $\mathbf{T}_{\texttt{len}}$ each for CS-WOTS$^+$ signature verification[2]. It also needs dh/d calls to **H** for the d root computations.

The size of the SPHINCS-α private and public keys along with the signature are $4n, 2n, (h + k(a+1) + d \cdot \texttt{len} + 1)n$ respectively.

The classical security level, or bit security of SPHINCS-α against generic attacks can be computed as

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

[2]It should be noted that the $w\texttt{len}$ bound for calls to **F** is a worst-case bound. This is a bound on the cost for CS-WOTS$^+$ signature verification. Given that the messages are hash values which can assumed to be close to uniformly distributed, this value will be closer to the average-case bound $(w/2)$. len in actual measurements.

Table 3: Parameter sets for the SPHINCS-α scheme.

Parameter Set	n	h	d	$\log t$	k	w	l	bitsec	sec level	sig bytes
sphincs-a-128s	16	63	9	13	12	73	22	128	I	6880
sphincs-a-128f	16	63	21	8	25	14	36	128	I	16720
sphincs-a-192s	24	63	9	14	17	77	32	192	III	14568
sphincs-a-192f	24	64	16	8	37	8	66	192	III	34896
sphincs-a-256s	32	66	11	13	23	79	42	255	V	27232
sphincs-a-256f	32	68	17	9	35	16	66	255	V	49312

The quantum security level, or bit security of SPHINCS-α against generic attacks can be computed as

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

Here, we are neglecting the small constant factors inside the logarithm.

8.1.2 Proposed Parameter Sets and Security Levels

As explained in the previous subsection, even for a fixed security level the design of SPHINCS-α 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 5 — 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. 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 8.1.1 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 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 8.2 we describe two different instantiations of the underlying hash function. Together with the six parameter sets listed in Table 3 we obtain 12 different instantiations of SPHINCS-α.

8.2 Instantiations of Hash Functions

In this section we define different signature schemes, which are obtained by instantiating the cryptographic function families of SPHINCS-α with SHA2 and SHAKE.

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

Table 4: Example parameter sets for SPHINCS-α 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 $n = 32$.

8.2.1 SPHINCS-α-SHAKE

For SPHINCS-α-SHAKE we define

$$\begin{aligned}
\mathbf{H_{msg}}(\mathtt{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M) &= \text{SHAKE256}(\mathtt{R}\|\mathbf{PK}.\mathtt{seed}\|\mathbf{PK}.\mathtt{root}\|M, m),\\
\mathbf{PRF}(\mathbf{PK}.\mathtt{seed}, \mathbf{SK}.\mathtt{seed}, \mathtt{ADRS}) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}\|\mathtt{ADRS}\|\mathbf{SK}.\mathtt{seed}, 8n),\\
\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M) &= \text{SHAKE256}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}\|M, 8n)\ \ .\\
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M_1) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}\|\mathtt{ADRS}\|M_1, 8n),\\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M_1\|M_2) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}\|\mathtt{ADRS}\|M_1\|M_2, 8n),\\
\mathbf{T}_\ell(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M) &= \text{SHAKE256}(\mathbf{PK}.\mathtt{seed}\|\mathtt{ADRS}\|M, 8n)\ \ .
\end{aligned}$$

8.2.2 SPHINCS-α-SHA2

In a similar way we define the functions for SPHINCS-α-SHA2. In some places we use SHA2-256 for $n = 16$ and SHA2-512 for $n = 24$ and $n = 32$. For this we use the shorthand SHA-X.

$$\begin{aligned}
\mathbf{H_{msg}}(\mathtt{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M) &= \text{MGF1-SHA-X}(\mathtt{R}\|\mathbf{PK}.\mathtt{seed}\|\text{SHA-X}(\mathtt{R}\|\mathbf{PK}.\mathtt{seed}\|\mathbf{PK}.\mathtt{root}\|M), m),\\
\mathbf{PRF}(\mathbf{PK}.\mathtt{seed}, \mathbf{SK}.\mathtt{seed}, \mathtt{ADRS}) &= \text{SHA2-256}(\mathtt{BlockPad}(\mathbf{PK}.\mathtt{seed})\|\mathtt{ADRS}^c\|\mathbf{SK}.\mathtt{seed}),\\
\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M) &= \text{HMAC-SHA-X}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}\|M)\ \ .
\end{aligned}$$

For $n = 32$, we only take the first 32 bytes of output of **PRF** and discard the rest.

$$\begin{aligned}
\mathbf{F}(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M_1) &= \text{SHA2-256}(\mathtt{BlockPad}(\mathbf{PK}.\mathtt{seed})\|\mathtt{ADRS}^c\|M_1),\\
\mathbf{H}(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M_1\|M_2) &= \text{SHA-X}(\mathtt{BlockPad}(\mathbf{PK}.\mathtt{seed})\|\mathtt{ADRS}^c\|M_1\|M_2),\\
\mathbf{T}_\ell(\mathbf{PK}.\mathtt{seed}, \mathtt{ADRS}, M) &= \text{SHA-X}(\mathtt{BlockPad}(\mathbf{PK}.\mathtt{seed})\|\mathtt{ADRS}^c\|M)\ \ ,
\end{aligned}$$

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

Padding PK.seed. Each of the instances of the tweakable hash function take **PK**.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 **PK**.seed to the length of a full 64-/128-byte SHA2 input block using

$$\mathtt{BlockPad}(\mathbf{PK}.\mathtt{seed}) = \mathbf{PK}.\mathtt{seed}\|\mathtt{toByte}(0, bl - n)\ \ .$$

where $bl = 64$ for SHA2-256 and $bl = 128$ for SHA2-512. Because of the Merkle-Damgård construction that underlies SHA2, this allows for reuse of the intermediate SHA2 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 SHA2 compression function, we use a compressed 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 ADRS.

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

9 Design Rationale

The design rationale behind SPHINCS-α is to follow the original SPHINCS$^+$ construction and apply the optimized CS-WOTS$^+$ one time signature scheme. 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.

Since one-time signature is extensively used in the SPHINCS$^+$ hypertree structure, optimizing the size of the one-time signature scheme has potentially great influence on the overall performance. The reason is that with CS-WOTS$^+$, we can re-tune the parameters of the signature scheme in order to have more flexibility, e.g., to achieve smaller (resp. faster) signatures with comparable running time (resp. signature size) as the original SPHINCS$^+$ scheme.

We are able to prove that the encoding scheme in CS-WOTS$^+$ is size optimal among all encoding scheme over tree structures (even including those not necessarily efficiently computable). Moreover, to the best of our knowledge, no existing DAG-based scheme achieves better performance in terms of signature size, which renders our scheme the size optimal one among all known schemes [27]. We are aware that the existence of probabilistic encoding schemes based on rejection sampling [17], and we consider the techniques thereof as of independent interest, since one can reduce the message length using the proof-of-work-style protocol in [17] and apply our scheme on the shorter digest.

10 Security Evaluation and Analysis with Respect to Known Attacks.

We follow the security analysis of SPHINCS$^+$ and adopt the suggestion of NIST [21] to discard the robust version of the tweakable hash functions as well as the haraka hash function. As a result, the EUF-CMA security of the signature scheme can be reduced to the security of different properties of the tweakable hash functions. Since the tweakable hash functions are modelled as (quantum) random oracles, the respective security properties can only be thwarted via generic attacks.

In particular, the EUF-CMA security of the signature scheme relies on the following security properties of the underlying components:

- The PRF security of $\mathbf{PRF}$ and $\mathbf{PRF_{msg}}$,
- The interleaved target subset resilience (ITSR) of $\mathbf{H_{msg}}$,
- The single function, multi-target undetectability (SM-UD), target-collision (SM-TCR), and decisional second-preimage resistance (SM-DSPR) of $\mathbf{F}$,
- The SM-TCR of $\mathbf{H}$ and $\mathbf{T}_\ell$.

Security Level of a Given Parameter Set. The security analysis is identical to SPHINCS$^+$. The classical security level, or bit security of SPHINCS-α 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-α 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.

As we have mentioned above, the assumption that the underlying tweakable hash function behaving like a random oracle effectively renders forgery attacks against the signature scheme being limited to generic attacks, whose success probability is captured in the above formulas.

Now we discuss the security of the two instantiations of hash functions, namely SHAKE and SHA2.

Security of SPHINCS-α-SHAKE. 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-α-SHAKE 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.

Security of SPHINCS-α-SHA2. NISTs 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 SHA2-256 used in SPHINCS-α-SHA2 has a chaining value of only 256 bits, making it slightly weaker in some metrics than SHAKE256 with 256-bit output. Therefore we make use of SHA2-512 in some cases to achieve the target security level.

11 Performance

We conduct our benchmarks on a Ubuntu 20.04 machine with Ryzen 5 3600 CPU and 16GB RAM, compiled with `gcc-9.3.0 -O3 -march=native -fomit-frame-pointer -flto`. We also provide optimized implementations for platforms supporting the AVX2 instruction set.

For the defined parameter sets, the resulting cycle counts, the key and signature sizes (in bytes) are listed in Table 5. Performance results are listed in Table 5, Table 6 and Table 7. In terms of memory consumption, we remark that the reference implementation tends towards low stack usage.

Table 5: Runtime benchmarks for SPHINCS-α. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

Parameter Set	Impl.	KeyGen	Sign	Verify	Pk	Sk	Sig
sphincs-a-shake-128f	ref	6861114	176440590	9035874	32	64	16720
sphincs-a-shake-192f	ref	14555628	281397294	7353450	48	96	34896
sphincs-a-shake-256f	ref	29112588	586596492	15740802	64	128	49312
sphincs-a-shake-128s	ref	347407200	3628303722	12591234	32	64	6880
sphincs-a-shake-192s	ref	533064942	6209945190	19292382	48	96	14568
sphincs-a-shake-256s	ref	362125278	4942646316	31932360	64	128	27232
sphincs-a-sha2-128f	ref	4157334	106933014	5569452	32	64	16720
sphincs-a-sha2-192f	ref	8837622	173603070	4654278	48	96	34896
sphincs-a-sha2-256f	ref	17439858	357693966	9646776	64	128	49312
sphincs-a-sha2-128s	ref	208068264	2172320100	7584102	32	64	6880
sphincs-a-sha2-192s	ref	319426722	3827118258	11723508	48	96	14568
sphincs-a-sha2-256s	ref	215034228	3011033142	19147662	64	128	27232
sphincs-a-shake-128f	avx2	2218014	57069090	3558492	32	64	16720
sphincs-a-shake-192f	avx2	4614804	92073114	3028500	48	96	34896
sphincs-a-shake-256f	avx2	9563742	191187306	5983920	64	128	49312
sphincs-a-shake-128s	avx2	108983646	1139743980	4891482	32	64	6880
sphincs-a-shake-192s	avx2	171004500	1996754616	7254738	48	96	14568
sphincs-a-shake-256s	avx2	115604604	1582371720	11677806	64	128	27232
sphincs-a-sha2-128f	avx2	1036602	26635716	2028186	32	64	16720
sphincs-a-sha2-192f	avx2	2199276	45218790	1744038	48	96	34896
sphincs-a-sha2-256f	avx2	4286574	91335474	3175290	64	128	49312
sphincs-a-sha2-128s	avx2	51421086	537033762	2689650	32	64	6880
sphincs-a-sha2-192s	avx2	78050718	988899534	3845970	48	96	14568
sphincs-a-sha2-256s	avx2	52048332	764352612	6005448	64	128	27232

Table 6: Runtime benchmarks for SPHINCS$^+$. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

Parameter Set	Impl.	KeyGen	Sign	Verify	Pk	Sk	Sig
sphincs-shake-128f	ref	7622514	178188408	10775124	32	64	17088
sphincs-shake-192f	ref	11240172	290022120	15972588	48	96	35664
sphincs-shake-256f	ref	29488050	593083386	15949980	64	128	49856
sphincs-shake-128s	ref	493648758	3747092580	3602178	32	64	7856
sphincs-shake-192s	ref	717515010	6427813662	5332932	48	96	16224
sphincs-shake-256s	ref	470748762	5584718124	7709508	64	128	29792
sphincs-sha2-128f	ref	4600566	107749800	6402438	32	64	17088
sphincs-sha2-192f	ref	6705198	181354752	9365400	48	96	35664
sphincs-sha2-256f	ref	17695728	362443014	9947394	64	128	49856
sphincs-sha2-128s	ref	294665274	2237140404	2282346	32	64	7856
sphincs-sha2-192s	ref	428811300	3954195432	3266478	48	96	16224
sphincs-sha2-256s	ref	283132530	3503794590	4785678	64	128	29792
sphincs-shake-128f	avx2	2494854	58500990	4063716	32	64	17088
sphincs-shake-192f	avx2	3541392	91863954	5919426	48	96	35664
sphincs-shake-256f	avx2	9676188	193273884	6019830	64	128	49856
sphincs-shake-128s	avx2	159844320	1210947264	1491894	32	64	7856
sphincs-shake-192s	avx2	233254134	2093058036	2165706	48	96	16224
sphincs-shake-256s	avx2	153274212	1799699922	3085776	64	128	29792
sphincs-sha2-128f	avx2	1143558	26872236	2204802	32	64	17088
sphincs-sha2-192f	avx2	1662498	45405504	3003534	48	96	35664
sphincs-sha2-256f	avx2	4327632	92059542	2967642	64	128	49856
sphincs-sha2-128s	avx2	72597852	551233638	846486	32	64	7856
sphincs-sha2-192s	avx2	105310692	1022229270	1201230	48	96	16224
sphincs-sha2-256s	avx2	69033492	918473904	1701324	64	128	29792

Table 7: Performance comparison between the original and improved SPHINCS$^+$ in terms of relative changes.

Parameter Set			Runtime			
SPHINCS$^+$	SPHINCS-α	Impl.	KeyGen	Sign	Verify	Sig Size
sphincs-shake-128f	sphincs-a-shake-128f	ref	−9.99%	−0.98%	−16.14%	−2.15%
sphincs-shake-192f	sphincs-a-shake-192f	ref	29.50%	−2.97%	−53.96%	−2.15%
sphincs-shake-256f	sphincs-a-shake-256f	ref	−1.27%	−1.09%	−1.31%	−1.09%
sphincs-shake-128s	sphincs-a-shake-128s	ref	−29.62%	−3.17%	249.55%	−12.42%
sphincs-shake-192s	sphincs-a-shake-192s	ref	−25.71%	−3.39%	261.76%	−10.21%
sphincs-shake-256s	sphincs-a-shake-256s	ref	−23.07%	−11.50%	314.19%	−8.59%
sphincs-sha2-128f	sphincs-a-sha2-128f	ref	−9.63%	−0.76%	−13.01%	−2.15%
sphincs-sha2-192f	sphincs-a-sha2-192f	ref	31.80%	−4.27%	−50.30%	−2.15%
sphincs-sha2-256f	sphincs-a-sha2-256f	ref	−1.45%	−1.31%	−3.02%	−1.09%
sphincs-sha2-128s	sphincs-a-sha2-128s	ref	−29.39%	−2.90%	232.29%	−12.42%
sphincs-sha2-192s	sphincs-a-sha2-192s	ref	−25.51%	−3.21%	258.90%	−10.21%
sphincs-sha2-256s	sphincs-a-sha2-256s	ref	−24.05%	−14.06%	300.10%	−8.59%
sphincs-shake-128f	sphincs-a-shake-128f	avx2	−11.10%	−2.45%	−12.43%	−2.15%
sphincs-shake-192f	sphincs-a-shake-192f	avx2	30.31%	0.23%	−48.84%	−2.15%
sphincs-shake-256f	sphincs-a-shake-256f	avx2	−1.16%	−1.08%	−0.60%	−1.09%
sphincs-shake-128s	sphincs-a-shake-128s	avx2	−31.82%	−5.88%	227.87%	−12.42%
sphincs-shake-192s	sphincs-a-shake-192s	avx2	−26.69%	−4.60%	234.98%	−10.21%
sphincs-shake-256s	sphincs-a-shake-256s	avx2	−24.58%	−12.08%	278.44%	−8.59%
sphincs-sha2-128f	sphincs-a-sha2-128f	avx2	−9.35%	−0.88%	−8.01%	−2.15%
sphincs-sha2-192f	sphincs-a-sha2-192f	avx2	32.29%	−0.41%	−41.93%	−2.15%
sphincs-sha2-256f	sphincs-a-sha2-256f	avx2	−0.95%	−0.79%	7.00%	−1.09%
sphincs-sha2-128s	sphincs-a-sha2-128s	avx2	−29.17%	−2.58%	217.74%	−12.42%
sphincs-sha2-192s	sphincs-a-sha2-192s	avx2	−25.89%	−3.26%	220.17%	−10.21%
sphincs-sha2-256s	sphincs-a-sha2-256s	avx2	−24.60%	−16.78%	252.99%	−8.59%

12 Advantages and Limitations

Since the SPHINCS-α signature scheme is an optimization upon the original SPHINCS$^+$ scheme, it inherits almost all the advantages and limitations of the original scheme, as discussed in [2]. Nevertheless, our new CS-WOTS$^+$ introduces new changes to the overall scheme and thus we recapture different points briefly.

- **Advantage: Stable computing time.** The number of hash function calls is fixed in our construction, in contrast to possibly variable numbers for the signing and verification algorithm of WOTS$^+$. While no timing attacks are identified against the implementations of our construction and WOTS$^+$, stable computing time is always preferable (especially for signing algorithms whose computation involves a private key).
- **Disadvantage: Incompatibility with XMSS.** Since we changed the underlying one-time signature component, the SPHINCS-α is no longer compatible with XMSS as SPHINCS$^+$ is.

The rest of the points are the same as SPHINCS$^+$.

- **Disadvantage: Signature size and speed.**
- **Advantage: "Minimal Security Assumptions".**
- **Advantage: State-of-the-art attacks are easily analyzed.**
- **Advantage: Small key sizes.**
- **Advantage: Reuse of established building blocks.**

References

[1] Désiré André. Mémoire sur les combinaisons régulières et leurs applications. In *Annales scientifiques de l'École Normale Supérieure*, volume 5, pages 155–198, 1876.

[2] 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, and Bas Westerbaan. SPHINCS+ submission to the nist post-quantum project, v.3.1, 2022.

[3] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, volume 10808 of *Lecture Notes in Computer Science*, pages 219–242, San Francisco, CA, USA, April 16–20, 2018. Springer, Heidelberg, Germany.

[4] Hacene Belbachir and Oussama Igueroufa. Congruence properties for bi s nomial coefficients and like extended ram and kummer theorems under suitable hypothesis. *Mediterranean Journal of Mathematics*, 17(1):1–14, 2020.

[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, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[6] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2129–2146. ACM Press, November 11–15, 2019.

[7] Daniel Bleichenbacher and Ueli M. Maurer. Directed acyclic graphs, one-way functions and digital signatures. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 75–82, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Heidelberg, Germany.

[8] Jurjen N. Bos and David Chaum. Provably unforgeable signatures. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 1–14, Santa Barbara, CA, USA, August 16–20, 1993. Springer, Heidelberg, Germany.

[9] David A Cooper, Daniel C Apon, Quynh H Dang, Michael S Davidson, Morris J Dworkin, Carl A Miller, et al. Recommendation for stateful hash-based signature schemes. *NIST Special Publication*, 800:208, 2020.

[10] Robert P Dilworth. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*, pages 139–144. Springer, 2009.

[11] Leonhard Euler. De evolutione potestatis polynomialis cuiuscunque (1+ x+ x 2+ x 3+ x 4+ etc.) n. *Nova Acta Academiae Scientiarum Imperialis Petropolitanae*, pages 47–57, 1801.

[12] Oded Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.

[13] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[14] Shay Gueron and Nicky Mouha. SPHINCS-simpira: Fast stateless hash-based signatures with post-quantum security. Cryptology ePrint Archive, Report 2017/645, 2017. `http://eprint.iacr.org/2017/645`.

[15] Douglas Zare (https://math.stackexchange.com/users/8345/douglas zare). How to express $(1 + x + x^2 + \cdots + x^m)^n$ as a power series? Mathematics Stack Exchange, 2011. URL:https://math.stackexchange.com/q/28861 (version: 2011-11-15).

[16] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. Xmss: extended merkle signature scheme. In *RFC 8391*. IRTF, 2018.

[17] Mikhail Kudinov, Andreas Hülsing, Eyal Ronen, and Eylon Yogev. Sphincs+c: Compressing sphincs+ with (almost) no cost. Cryptology ePrint Archive, Paper 2022/778, 2022. `https://eprint.iacr.org/2022/778`.

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

[19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-micali hash-based signatures. In *RFC 8554*. IRTF, 2019.

[20] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany.

[21] Dustin Moody. Parameter selection for the selected algorithms, 2022.

[22] Lucas Pandolfo Perin, Gustavo Zambonin, Ricardo Custódio, Lucia Moura, and Daniel Panario. Improved constant-sum encodings for hash-based signatures. *Journal of Cryptographic Engineering*, pages 1–23, 2021.

[23] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. Cryptology ePrint Archive, Report 2002/014, 2002. `http://eprint.iacr.org/2002/014`.

[24] Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

[25] Serge Vaudenay. One-time identification with low memory. In *Eurocode'92*, pages 217–228. Springer, 1993.

[26] S Ole Warnaar. The andrews–gordon identities and q-multinomial coefficients. *Communications in mathematical physics*, 184(1):203–232, 1997.

[27] Kaiyi Zhang, Hongrui Cui, and Yu Yu. Revisiting the constant-sum winternitz one-time signature with applications to sphincs+ and xmss. In *Advances in Cryptology–CRYPTO 2023: 43th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–24, 2023, Proceedings, to appear*, pages xxx–xxx. Springer, 2023.

A Parameter Evaluation Sage Script

```
maxsigs=2**64
F = RealField(256+100)

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 run(l,w):
    s=l*(w-1)//2
    dp=[]
    for i in range(0,l+1):
        dp.append([])
        for j in range(0,s+1):
            dp[i].append(0)
    dp[0][0]=1
    for i in range(1,l+1):
        for j in range(0,s+1):
            for k in range(0,w):
                if k>j:
                    continue
                dp[i][j]+=dp[i-1][j-k]
    return dp[l][s]

lenmap={}

for osec in [128,192,256]:
    lenmap[osec]={}
    lastl=100
    for w in range(8,129):
        lenmap[osec][w]=99999
        ans = lastl
        for l in range(lastl-1,1,-1):
            if F(log(F(run(l,w)))/log2)>osec:
                ans=l
            else:
                break
        lenmap[osec][w]=ans
        lastl=ans
        print(osec,w,ans,w*ans/2.0)
print('preprocess end')


def size(sec,h,d,b,k,w1,hashbytes):
    l=lenmap[sec][w1]
    return ((b+1)*k+h+l*d+1)*hashbytes


def split(h,d):
    ans=[0 for i in range(d)]
    for i in range(h):
        ans[i%d]+=1
    return ans

def speed(sec,h,d,b,k,w1):
    l=lenmap[sec][w1]
    ans=k*(2**b)*(2)
    ans+=d*(2**(h/d)*(l*w1+1))
    return ans


def sec(h,d,b,k,w1):
    s=F(0)
    for r in range(100): # in fact, qhitprob(maxsigs,r)<2^{-256}*2^{-64} when r>100.
    ↪   Thus we can omit all r>100
        p=F(1-(1-1/F(2**b))^r)**k
        s+=qhitprob(maxsigs,r)*p
    return -F(log(F(s))/log2)

def factor(h):
    ans=[]
    for d in range(6,h):
        if h%d==0:
            ans.append(d)
    return ans


paras={}
paras[128]={}
paras[192]={}
paras[256]={}

paras[128]['small']=(63,7,12,14,16)
paras[128]['fast']=(66,22,6,33,16)

paras[192]['small']=(63,7,14,17,16)
paras[192]['fast']=(66,22,8,33,16)

paras[256]['small']=(64,8,14,22,16)
paras[256]['fast']=(68,17,9,35,16)


for osec in [128,192,256]:
    D={}
    for w in range(8,129):
        l=lenmap[osec][w]
        if not l in D:
            D[l]=w
    good_w=list(D.values())
    print(good_w)
    for ty in ['small','fast']:
        h,d,b,k,w1=paras[osec][ty]
        max_size=size(osec,h,d,b,k,w1,osec//8)
        max_hashcalls=speed(osec,h,d,b,k,w1)
        print(osec,ty,max_size,max_hashcalls)
        best_szsp=(max_size,max_hashcalls)
        best_spsz=(max_hashcalls,max_size)
        para_szsp=(h,d,b,k,w1)
        para_spsz=(h,d,b,k,w1)
        for h in [63,64,65,66,68]:
            leaves=2**h
            for b in range(3,15):
                for k in range(6,40):
                    if sec(h,0,b,k,0)>=osec-1:
                        for d in factor(h):
                            for w1 in good_w:
                                sz=size(osec,h,d,b,k,w1,osec//8)
                                sp=speed(osec,h,d,b,k,w1)
                                if sec(h,d,b,k,w1)>=osec-1 and sz<=max_size and
                                ↪  sp<max_hashcalls:
                                    if (sz,sp)<best_szsp:
                                        best_szsp=(sz,sp)
                                        para_szsp=(h,d,b,k,w1,lenmap[osec][w1])
                                    if (sp,sz)<best_spsz:
                                        best_spsz=(sp,sz)
                                        para_spsz=(h,d,b,k,w1,lenmap[osec][w1])
            print(h)
        print(para_szsp,best_szsp)
        print(para_spsz,best_spsz)
```