SPHINCS-α: A Compact Stateless Hash-Based Signature Scheme

Kaiyi Zhang[1], Hongrui Cui[1], and Yu Yu[1,2]

[1]Department of Computer Science, Shanghai Jiao Tong University, 200240 Shanghai
[2]Shanghai Qizhi Institute, 200232 Shanghai
{*kzoacn,rickfreeman*}*@sjtu.edu.cn*
yuyu@yuyu.hk

January 17, 2022

Abstract

Hash-based signatures offer a conservative alternative to post-quantum signatures with arguably better-understood security than other post-quantum candidates. Nevertheless, a major drawback that makes it less favorable to deploy in practice is the (relatively) large size of the signatures, and long signing and verification time.

In this paper, we introduce SPHINCS-α, a stateless hash-based signature scheme, which benefits from a twofold improvement. First, we provide an improved Winternitz one-time signature with an efficient size-optimal encoding, which might be of independent interest. Second, we give a variant of the few-time signature scheme, FORC, by applying the Winternitz method. Plugging the two improved components into the framework of the state-of-the-art (stateless) hash-based SPHINCS$^+$, with carefully chosen parameter choices, yields a certain degree of performance improvement. In particular, under the "small" series parameter set aiming for compact signatures, our scheme reduces signature size and signing time by 8-11% and 3-15% respectively, compared to SPHINCS$^+$ at all security levels. For the "fast" series that prioritizes computation time, our scheme exhibits a better performance in general. E.g., when instantiating the simple tweakable hash function with SHA-256, our scheme reduces the signing and verification time by 7-10% and up to 10% respectively, while keeping roughly the same signature size. The security proofs/estimates follow the framework of SPHINCS$^+$. To facilitate a fair comparison, we give the implementation of SPHINCS-α by adapting that of SPHINCS$^+$, and we provide a theoretical estimate in the number of hash function calls.

Keywords— Hash-Based Signature, Post-Quantum Cryptography, SPHINCS$^+$

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 [Mer90] that builds upon Lamport's one-time signature (OTS) [Lam79]. The recent efforts towards improving stateful signatures lead to the eXtended Merkle Signature Scheme (XMSS) [HBG$^+$18] and the Leighton-Micali Signature (LMS) [MCF19], standardized by NIST [CAD$^+$20] 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 [Gol87, Gol04] 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 [BHH$^+$15] offered a practical instantiation of the Goldreich-style stateless hash-based signature. SPHINCS serves as a basis for subsequent works driven by the NIST PQC standardization process, including Gravity-SPHINCS [AE18], SPHINCS-Simpira [GM17] and SPHINCS$^+$ [BHK$^+$19]. As a NIST PQC Round-3-Alternate candidate, SPHINCS$^+$ [BHK$^+$19] 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.

Performance. Despite the desirable features such as conservative assumptions, well-understood security, and flexibility in choosing the underlying hash functions, the performance of hash-based signatures, in terms of signing speed and signature size, is less competitive than other (e.g., lattice-based) candidates, which makes it less favorable for adoption. The performance of a hash-based signature can benefit from new structures [BM96a, BM96b, BDH11], more efficient underlying components (e.g., one-time signature [DSS05, Hül13], few-time signature [Per01, RR02], lightweight hash function [KLMR16]), and hardware accelerations [HRS16a, Köl18, WJW$^+$19, ACZ18, BHRv21, SZM20]. The main focus of this paper is algorithmic optimization of stateless hash-based signatures.

Table 1: Performance comparison between SPHINCS$^+$ and SPHINCS-α, with simple tweakable hash function instantiated with `sha256`. 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	2.0×10^6	2.3×10^7	1.9×10^6	17088	1.7×10^6	2.1×10^7	1.7×10^6	17040	−13.44%	−7.11%	−9.99%	−0.28%
192f	2.8×10^6	3.9×10^7	2.9×10^6	35664	1.4×10^6	3.5×10^7	2.9×10^6	35640	−50.72%	−10.28%	−1.52%	−0.07%
256f	7.2×10^6	7.9×10^7	3.1×10^6	49856	3.8×10^6	7.2×10^7	3.1×10^6	49696	−47.87%	−8.64%	+1.70%	−0.32%
128s	6.2×10^7	4.7×10^8	7.6×10^5	7856	4.8×10^7	4.6×10^8	1.2×10^6	6960	−23.41%	−3.34%	+58.48%	−11.41%
192s	9.2×10^7	8.7×10^8	1.2×10^6	16224	8.0×10^7	7.2×10^8	2.0×10^6	14784	−12.66%	−16.85%	+65.25%	−8.88%
256s	6.1×10^7	7.8×10^8	1.8×10^6	29792	6.1×10^7	6.6×10^8	3.3×10^6	27104	+0.31%	−15.15%	+84.76%	−9.02%

Our contributions. In this paper, we propose SPHINCS-α, a stateless hash-based signature scheme, which improves the performance of state-of-the-art of stateless hash-based signatures. Our optimization stems from the enhancement of two components.

- We introduce an improved variant of the Winternitz OTS scheme, referred to as "Balanced WOTS$^+$", using an efficient and balanced encoding algorithm. Using

[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.

order theoretic techniques, we prove that the encoding scheme is size-optimal, which might be of independent interest.

- We compose the Winternitz method to the few-time signature scheme FORS. While the resulting scheme, which we call FORC, does not constitute an overall improvement, it offers more room for the tradeoffs between the security and efficiency parameters when used to build the signatures.

Following the frameworks of SPHINCS$^+$ we provide security proof, and security estimates based on carefully chosen and optimized parameters for the above components and the resulting signatures. 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 8-11% and 3-15% respectively at all security levels while under the "fast" series (optimized towards fast signing operations) our scheme exhibits better performance in general. When instantiating the "simple" variant of the tweakable hash function with `sha256`, our scheme reduces the signing time and verification time by 7-10% and up to 10% respectively under roughly the same signature size.

We refer to more detailed comparisons for the full spectrum of parameter choices in Section 5.4 and Appendix A. 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 122% increase in verification time. Nevertheless, since the verification only amounts to 1-9% of the signing time we argue that this sacrifice is worthwhile considering efficiency gain in signing time and signature size.

Paper structure. We organize the rest of this paper as follows: In Section 2 we revisit the SPHINCS$^+$ signature whose structure our construction will mostly follow. In Section 3 and Section 4, we describe our optimized one-time signature 'Balanced WOTS$^+$" and our variant of few-time signature "FORC". In Section 5, we present the security proof of our scheme and the implementation details, and report the performance compared to SPHINCS$^+$.

2 Revisiting SPHINCS$^+$

In this section, we briefly review the SPHINCS$^+$ scheme.

2.1 Notations

We use $[w] \stackrel{\text{def}}{=} \{0, 1, \ldots, w-1\}$ for $w \in \mathbb{N}^+$. We denote the i-th element of a vector v by v_i. By $\log(x)$ we refer to the binary logarithm, i.e., $\log_2(x)$. We denote the concatenation of strings (vectors) a and b by $a\|b$ or (a, b). For a set $\mathcal{S}$, we denote the size of $\mathcal{S}$ and the power set of $\mathcal{S}$ by $|\mathcal{S}|$ and $P(\mathcal{S})$ respectively. We let λ be the security parameter, and refer to a λ-bit value as a block.

2.2 The SPHINCS$^+$ Framework

The structure of SPHINCS$^+$ forms a hypertree, a tree of Merkle trees. Each Merkle tree is linked to the next tree using a one-time signature. And the leaf node of the hypertree is the public key of a few-time signature. The whole tree is generated on the fly, which means the secret key holder can obtain any part of the hypertree from private randomness with a pseudorandom function. An illustration is shown in Figure 1.

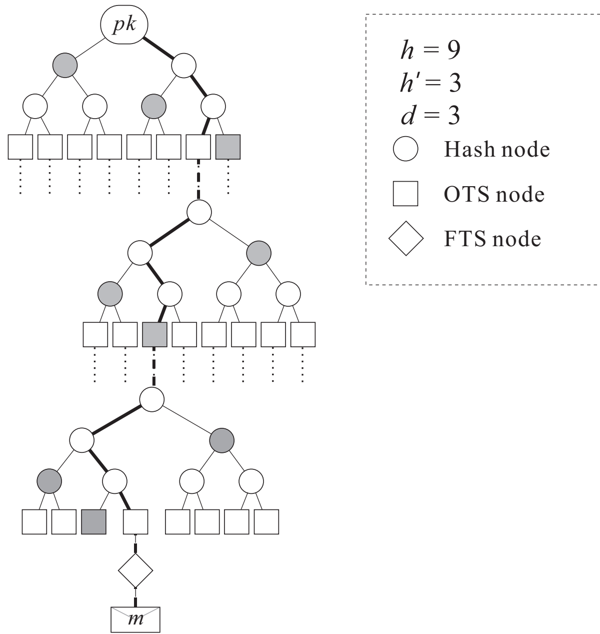

Figure 1: An overview of the SPHINCS$^+$ structure from [BHK$^+$19].

2.3 Cryptographic (Hash) Function Families

Tweakable hash functions Tweakable hash functions are the basic building blocks in SPHINCS$^+$. A tweakable hash function takes public parameters P, a tweak T, and a message m as inputs. The public parameters can be interpreted as a function key, and the tweak can be thought of like a salt/nonce. The definition is quoted from SPHINCS$^+$ [BHK$^+$19].

Definition 1 (Tweakable hash function) *Let $\alpha \in \mathbb{N}$, and let $\mathcal{P}$ and $\mathcal{T}$ be the public parameter space the tweak space respectively. A tweakable hash function is an efficient function mapping an α-bit message M to an λ-bit hash value* MD *using a function key called public parameter $P \in \mathcal{P}$ and a tweak $T \in \mathcal{T}$.*

$$\mathbf{Th} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{\alpha} \rightarrow \{0,1\}^{\lambda}, \ \ \mathrm{MD} \leftarrow \mathbf{Th}(P,T,M) \ \ .$$

We sometimes write $\mathbf{Th}(M)$ instead of $\mathbf{Th}(P,T,M)$ wherever P and T are obvious from the context. A straightforward instantiation of the tweakable hash function in the QROM can be simply

$$\mathbf{Th}(P,T,M) = H(P||T||M) \ \ ,$$

where the hash function H is assumed to behave like a quantum accessible random oracle. This construction is used in SPHINCS$^+$ as a “simple” version tweakable hash function, as opposed to a “robust” version.

Following SPHINCS$^+$, we use the following simplifying notations. We write $\mathbf{Th}_l : \{0,1\}^{\lambda} \times \{0,1\}^{256} \times \{0,1\}^{l\lambda} \rightarrow \{0,1\}^{\lambda}$, for the tweakable hash function with input length $l\lambda$. We further define $\mathbf{F} \stackrel{\text{def}}{=} \mathbf{Th}_1, \mathbf{H} \stackrel{\text{def}}{=} \mathbf{Th}_2$.

Pseudorandom functions and message digest. SPHINCS$^+$ uses a pseudorandom function $\mathbf{PRF}$ to generate pseudorandom keys, $\mathbf{PRF} : \{0,1\}^{\lambda} \times \{0,1\}^{256} \rightarrow \{0,1\}^{\lambda}$, and another pseudorandom function $\mathbf{PRF_{msg}}$ to generate randomness for message

compression: $\mathbf{PRF_{msg}} : \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$. SPHINCS$^+$ applies a keyed hash function $\mathbf{H_{msg}} : \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ to the message and then signs its compressed output (digest).

2.4 WOTS$^+$

We show WOTS$^+$ [Hül13], a one-time signature scheme which is used in SPHINCS$^+$. One-time signature restricts a private key to be used exactly one message, otherwise its security quickly decreases [BH17].

The security parameter / message length is denoted by λ. The Winternitz parameter is w. Let l be the number of blocks in an uncompressed WOTS$^+$ private key, public key, and signature, where

$$l = l_1 + l_2, l_1 = \left\lceil \frac{\lambda}{\log(w)} \right\rceil, l_2 = \left\lfloor \frac{\log(l_1(w-1))}{\log(w)} \right\rfloor + 1 \ .$$

The WOTS$^+$ key pair. The WOTS$^+$ private key sk consists of l random blocks (i.e. $sk = (sk_1, sk_2, \ldots, sk_l), sk_i \in \{0,1\}^\lambda$). The uncompressed WOTS$^+$ public keys $\{pk_i\}_{i=1}^l$ is derived by applying $\mathbf{F}$ iteratively for $w-1$ times to each of the blocks in the private key (i.e., $pk_i = \mathbf{F}^{w-1}(sk_i)$). Note each $\mathbf{F}$ has a different tweak. Finally, the public keys are compressed to a block using a single tweakable hash function $\mathbf{Th}_l$ (i.e., $pk = \mathbf{Th}_l(pk_1, pk_2, \ldots, pk_l)$). We use WOTS$^+$ public key to refer to the compressed public key.

In the context of the SPHINCS$^+$, the WOTS$^+$ private key sk is derived from the SPHINCS$^+$ private key and the address of the WOTS$^+$ key pair within the hypertree using **PRF**.

WOTS$^+$ signature and verification. A message $m \in \{0,1\}^\lambda$ is interpreted as l_1 integers $m_i \in \{0, 1, \ldots, w-1\}$. We compute a checksum $C = \sum_{i=1}^{l_1}(w - 1 - m_i)$, represented as a string of l_2 base-w values $C = (C_1, C_2, \ldots, C_{l_2})$. Let $M = (m_1, \ldots, m_{l_1}, C_1, \ldots, C_{l_2})$. Apply the hash function $\mathbf{F}$ to each of the blocks in the private key sk_i for M_i times (i.e. $\sigma_i = \mathbf{F}^{M_i}(sk_i)$). The l blocks σ_i make up the signature σ. The verifier can them recompute the checksum and apply $\mathbf{F}$ to each block for $w - 1 - M_i$ times (i.e. $pk_i' = \mathbf{F}^{w-1-M_i}(\sigma_i)$). Finally compress those values to an λ-bit public key pk' using $\mathbf{Th}_l$ (i.e. $pk' = \mathbf{Th}_l(pk_1', pk_2', \ldots, pk_l')$). The verifier accepts this signature iff $pk = pk'$.

2.5 The hypertree

The hypertree structure generalizes both the Merkle tree [Mer90] and Goldreich tree [Gol87]. It was adopted in XMSS [BDH11, HRB13] and SPHINCS$^+$ [BHK$^+$19].

A single Merkle tree. In order to sign $2^{h'}$ messages, the signer generates $2^{h'}$ WOTS$^+$ key pairs and constructs a binary tree using these $2^{h'}$ public keys as leaf nodes. That is, he repeatedly applies $\mathbf{H}$ to each pair of child nodes to generate the corresponding parent node until reaching the root of the tree, which is the public key for this single tree scheme. Note that each $\mathbf{H}$ is parameterized with a unique address of as well as the SPHINCS$^+$ public seed. The height of the tree, h', corresponds to the number of $\mathbf{H}$ applications from any leaf to the root.

To sign a message, the signer picks one of the WOTS$^+$ leaf nodes and publishes the WOTS$^+$ signature as well as all siblings of the nodes on the path from the leaf to the root, which is referred to as the "authentication path". The verifier first derives the WOTS$^+$ public key from the signature and then uses the nodes on the authentication path and their siblings to reconstruct the root.

Notice that in order to generate a root node that is able to authenticate $2^{h'}$ messages, one needs to hash all of the leaf nodes. This becomes impractical if we want to sign all messages ($h' = n$) in a single Merkle tree. We thus need to use a hypertree to be explained next.

A tree of trees. A hypertree consists of d layers. The leaf nodes of the trees on the bottom layer are used to sign messages (in SPHINCS$^+$, the message is a FORS public key), while the leaf nodes of trees on other layers are used to sign the root nodes of the trees immediately beneath them. We refer to Figure 1 for a visualization of the structure, where SPHINCS$^+$ defines h', d and sets the total tree height $h = h'd$. During key generation, only the top-most tree is generated to derive the public key. The rest of the trees can be generated "on the fly" when needed.

2.6 FORS

A few-time signature allows a private key to sign multiple messages, with decreasing security for the increasing number of signatures. Forest of Random Subsets (FORS) is a few-time signature scheme proposed in SPHINCS$^+$[BHK$^+$19]. FORS is parameterized by integers k and $t = 2^a$, and can be used to sign messages of ka bits.

The FORS key pair. The FORS public key consists of k Merkle trees where each tree has t leaves. The forest is derived from SPHINCS$^+$ private key using **PRF** and the address of the key in the hypertree. To construct the FORS public key, we first construct k Merkel trees from their respective t random blocks and then compress the root nodes using $\mathbf{Th}_k$. The resulting block is the FORS public key.

FORS signature and verification. Given a message of ka bits, we split it into k chunks of a bits. Each chunk value is interpreted as the index of a single leaf node in the corresponding Merkle tree. The signature consists of these nodes and their respective authentication paths. The verifier reconstructs all the root nodes from the signature, compresses the root nodes using $\mathbf{Th}_k$, and compares the result value against the public key. See Figure 2 for an illustration.

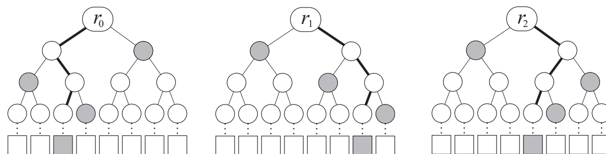

Figure 2: An example of a FORS signature with $k = 3$ and $a = 3$, on message 010 110 100.

2.7 The SPHINCS$^+$

The SPHINCS$^+$ is built upon the aforementioned components.

The SPHINCS$^+$ key pair. The public key consists of two blocks: the root node of the hypertree, and a random public seed PK.seed. In addition, the private key consists of two more random seeds: SK.seed, to generate the WOTS$^+$ and FORS private keys, and SK.prf, used for the message digest.

The SPHINCS$^+$ signature and verification. The signature consists of a FORS signature on a digest of the message, a WOTS$^+$ signature on the corresponding FORS public key, and a series of authentication paths and WOTS$^+$ signatures to authenticate that WOTS$^+$ public key. To verify this chain of paths and signatures, the verifier

iteratively reconstructs the public keys and root nodes until the root node at the top of the SPHINCS$^+$ hypertree is reached.

Message digest and Index selection. SPHINCS$^+$ generates a randomizer $R = \mathbf{PRF}(\text{SK.prf}, \text{OptRand}, M)$ and includes R as part of the signature, where OptRand is some optional randomness. It then derives the index of the leaf node idx and the message digest MD

$$(\text{MD}||\text{idx}) = \mathbf{H_{msg}}(R, \text{PK.seed}, \text{PK.root}, M) \ ,$$

where the relation is publicly verifiable, preventing an adversary from selecting an index at his own choice.

Maximum number of signatures. SPHINCS$^+$ restricts the number of signatures to no more than 2^{64}. This limit suffices for most practical applications, and it sets the maximum height of the hypertree for performance reasons.

3 Balanced WOTS$^+$

In this section, we present our Balanced WOTS$^+$, whose encoding scheme is slightly more optimized than the counterpart in WOTS$^+$.

3.1 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.

3.2 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 2). 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 2 ((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 [BC93] studied this approach for the special case of $w = 2$. Vaudenay [Vau93] generalized it to arbitrary w, but he did not provide an encoding algorithm. Perin et al. provided a similar encoding algorithm in [PZC$^+$21], but they did not present a size-optimal proof.

Theorem 3.1 *For any* $m \in [l(w-1)+1]$, $\mathcal{C}_m \stackrel{\text{def}}{=} \{v \in [w]^l : \sum_{i=1}^{l} v_i = m\}$ *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. [PZC+21] 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 3.2 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 3.2 (Size-optimal encoding) *For every incomparable* $\mathcal{C}^* \in P([w]^l)$, *it holds that*

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

We defer its proof to Theorem 3.4, which rephrases Theorem 3.2 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

$$D_{1,m} = 1, \text{ for } m \in \{0, 1, \dots, w-1\}$$
$$D_{n,m} = 0, \text{ for } 2 \leq n \in \mathbb{Z}, m \in \mathbb{Z}^- \ ,$$

and recurrence relation

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

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 [Eul01] has studied $w = 3, 4, 5$, known as trinomial, quadrinomial and quintinomial coefficients respectively. The generalized form was studied in the literature,

e.g., [And76, War97, BI20]. Actually, we can use an inclusion-exclusion argument to express it as a function of binomial coefficients [hz11]

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

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.

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

```
Function Encode(x)
    Let v be an array of size l;
    m ← ⌊l(w − 1)/2⌋;
    for i ← l ... 1 do
        for j ← 0 ... min(w − 1, m) do
            if x ≥ D_{i−1,s−j} then
                x ← x − D_{i−1,s−j};
            else
                v_i ← j;
                break;
        m ← m − v_i;
    return v;
```

Let us explain the encoding algorithm. As previously stated, the problem can be divided into several subproblems by considering the value of the last element v_i. To encode a natural number $x \in [0, D_{i,m})$, we can simply determine $v_i = j$ 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_i is determined, we proceed to its preceding 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 [BM94].

Preliminaries of Order Theory

Definition 3 (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 $y \le x$).*

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 4 ((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 3.3 (Dilworth's theorem [Dil09]) *For any finite poset P, the size of P's maximum antichain equals the size of a minimum chain decomposition of P.*

Let $P = ([w]^l, \le)$ 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 3.4 *$\mathcal{C}$ is the maximum antichain, i.e., $\mathcal{C}$ is size-optimal.*

Proof: We have proved that $\mathcal{C}$ is an antichain in Theorem 3.1. 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 [Spe28], 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 $|(a_1, ..., a_n)| \stackrel{\text{def}}{=} a_1 + ... + a_n$.

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 Figure 3 below:

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

Figure 3: 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).

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 $(c_i, 0), ..., (c_i, w-1)$ 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 $c = \{c_1, \ldots, c_m\}$. We discuss the following cases.

Case $m \leq w$: We have $k = m-1 \leq w-1$. $[(c_i, 0), \ldots, (c_i, k+1-i)]$ appears as the first $(k+2-i)$ elements of the i-th column, and then $[(c_i, k+1-i), \ldots, (c_i, w-1)]^T$ as the last $(w+i-k-1)$ elements of the $(k+2-i)$-th row in Figure 3.

Case $m > w$: We have $k = w-1 < m-1$. If $1 \leq i \leq m-w+1$, then $[(c_i, 0), (c_i, w-1)]$ appears as the i-th column in Figure 3. Otherwise, $m-w+1 < i \leq m$. $[(c_i, 0), \ldots, (c_i, m-i)]$ and $[(c_i, m-i), \ldots, (c_i, w-1)]^T$ 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 $a \in [w]^{n-1}$, $(a, 0)$, $\ldots$,$(a, w-1)$ 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. ∎

3.3 Theoretical Performance

Our Balanced 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.

Table 2: Comparison of length l between WOTS$^+$ and Balanced 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

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$^+$.

4 FORC

We present a variant of the few time signature, called Forest of Random Chains (FORC). Intuitively, the Winternitz method manages to encode multiple-bit messages into a single block using hash chains. Therefore we decide to add hash chains in the leaf node of the origin FORS signature. Let w' be a Winternitz parameter (w is reserved for our balanced WOTS$^+$), indicating the length of the hash chain at the bottom. Our construction, as shown in Figure 4, is therefore parameterized by integers k , $t = 2^a$ and w'.

Key pair The public key consists of k trees where each tree has t chains hanging from its leaves. To construct the public key, we first sample t random block, iteratively apply $\mathbf{F}$ to each block w' times. Finally, treat the t the end-points blocks as leaves and compress them down to the root of the Merkle tree.

Signature and verification Given a message of $k(a + \log w')$ bits,we split it into k chunks of $a + \log w'$ bits, denoted by (J, L). Each chunk is interpreted as the index of

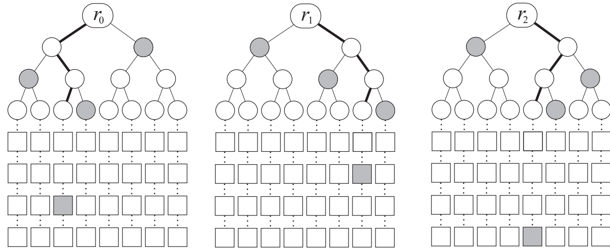

Figure 4: An illustration of a FORC signature with $k = 3, a = 3$ and $w' = 4$, for the message 010 10 110 01 100 11.

a single leaf node with a chain position for each of the k Merkle trees. The signature consists of the selected node and their respective authentication paths. The verifier reconstructs each of the root nodes using the authentication paths and uses $\mathbf{Th}_k$ to reconstruct the public key.

Collision probability. We specifiy the position of the FORC scheme by a pair $(J, L) \in [t]\times[w']$. The probability that under the same tree the corresponding signature of a random pair (J, L) is efficiently implied by another random one (J', L') is

$$\Pr[J = J'] \cdot \Pr[L \leq L'] = \frac{1}{t}\Big(\frac{1}{w'}\sum_{i=1}^{w'}\frac{i}{w'}\Big) = \frac{1}{t}\cdot\frac{w'+1}{2w'} \ .$$

Therefore, compared to the original FORS scheme (which can be seen as the special of ours for $w' = 1$), we improve the collision probability by a factor of $\frac{w'+1}{2w'}$. While the improvement seems not substantial at all (i.e., $\frac{w'+1}{2w'} > 1/2$ regardless of w'), it adds one more degree of freedom for tradeoffs and thus creates more possibilities for the performance improvement of the resulting signature.

5 The SPHINCS-α Signature Scheme

We describe more details of the SPHINCS-α signature scheme in this section. Our signature follows the overall structure of SPHINCS$^+$ (see Fig. 1) except for replacing its underlying WOTS$^+$ and FORS with our Balanced WOTS$^+$ and adapted FORC respectively.

5.1 Security Notions

We recall the relevant security properties about the tweakable hash functions used in the security analysis of SPHINCS$^+$ as well as the standard (post-quantum) security notions such as digital signature and pseudorandom function. We defer the (adapted) notion and analysis of interleaved target subset resilience to Section 5.2.

PQ-SM-TCR. This variant of target collision resistance is modelled by a two-stage game: an adversary first (denoted by $\mathcal{A}_1$) makes multiple adaptive queries $Q = \{(T_i, M_i)\}_{i=1}^{p}$ to the single function (keyed with the same P hidden from $\mathcal{A}_1$), one query per distinct tweak T_j. Then, he (i.e., $\mathcal{A}_2$) gets to see P, specifies which target (i.e., j), and wins the game iff he finds a collision for the specified tweak T_j. This notion is formally given in the following definition.

Definition 5 (PQ-SM-TCR) *Let* **Th** *be a tweakable hash function. We define the success probability of a quantum adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *as*

$$\begin{aligned}&\mathrm{Succ}^{\textit{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}) = \Pr\big[P \leftarrow \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathbf{Th}(P,\cdot,\cdot)};\\&(j, M) \leftarrow \mathcal{A}_2(Q, S, P) : \mathbf{Th}(P, T_j, M_j) = \mathbf{Th}(P, T_j, M)\\&M \neq M_j \wedge \mathsf{DIST}(\{T_i\}_{i=1}^p)\big] \ ,\end{aligned}$$

where $Q = \{(T_i, M_i)\}_{i=1}^p$ *denotes the set of* $\mathcal{A}_1$*'s query and the predicate* DIST *returns 1 iff all tweaks are distinct. Further, we define the PQ-SM-TCR insecurity*

$$\mathrm{InSec}^{\textit{PQ-SM-TCR}}(\mathbf{Th}; \xi, p) = \max_{\mathcal{A}}\{\mathrm{Succ}^{\textit{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A})\} \ ,$$

as the maximum success probability of all quantum adversaries with running time ξ *and query complexity* q*. By "PQ-SM-TCR with tweak advice" we refer to the special case that adversary* $\mathcal{A}_1$ *commits to the oracle all distinct tweaks ahead of its queries.*

PQ-SM-DSPR. This property upper bounds the probability of a quantum adversary determining whether the tweakable hash function has multiple preimages or not.

Definition 6 (PQ-SM-DSPR) *Let* $\mathbf{Th}, \mathsf{DIST}, Q$ *be as defined in Definition 5. We define the success probability of a quantum adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *as* $\mathrm{Adv}^{\textit{SM-DSPR}}_{\mathbf{Th},p}(\mathcal{A}) = \max\{0, \mathrm{succ} - \mathrm{triv}\}$ *with*

$$\begin{aligned}\mathrm{succ} = \Pr\big[&P \leftarrow \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathbf{Th}(P,\cdot,\cdot)}(); (j, b) \leftarrow \mathcal{A}_2(Q, S, P) :\\&\mathsf{SP}_{P,T_j}(M_j) = b \wedge \mathsf{DIST}(\{T_i\}_{i=1}^p)\big];\end{aligned}$$

$$\begin{aligned}\mathrm{triv} = \Pr\big[&P \leftarrow \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathbf{Th}(P,\cdot,\cdot)}(); (j, b) \leftarrow \mathcal{A}_2(Q, S, P) :\\&\mathsf{SP}_{P,T_j}(M_j) = 1 \wedge \mathsf{DIST}(\{T_i\}_{i=1}^p)\big] \ ,\end{aligned}$$

where $\mathsf{SP}_{P,T}(M)$ *is the predicate that returns 1 if there exists another* M' *s.t.* $\mathbf{Th}(P, T, M) = \mathbf{Th}(P, T, M')$*. We define the PQ-SM-DSPR insecurity*

$$\mathrm{InSec}^{\textit{PQ-SM-DSPR}}(\mathbf{Th}; \xi, p) = \max_{\mathcal{A}}\{\mathrm{Adv}^{\textit{SM-DSPR}}_{\mathbf{Th},p}(\mathcal{A})\} \ ,$$

as the maximum success probability of every quantum adversary with running time ξ *and query complexity* q*. As a special case, we use the term "with tweak advice" to denote the case where the adversary* $\mathcal{A}_1$ *sends all the tweaks to the oracle ahead of its queries.*

PQ-EU-CMA. This is an enhancement to the classical existential unforgeability under adaptive chosen message attack by allowing the adversary to have access to a quantum computer. We defer the formal definition to Appendix B.

PQ-PRF. This extends the classical definition of pseudorandom functions to quantum adversaries, except that the access to the random/pseudorandom function remains classical. We defer the formal definition to Appendix B.

5.2 Security Evaluation

Since our overall structure remains unaltered compared to SPHINCS$^+$, the security analysis largely follows the original framework. The only adaption needed is the Interleaved Target Subset Resilience security of the FORS construction. Since we deviate from the original structure, this value needs to be re-computed. We first recall the original main theorem of the SPHINCS$^+$ signature.

Theorem 5.1 *For parameters n, w, h, d, m, t, k, l as described above,* SPHINCS$^+$ *is PQ-EU-CMA secure if*

- **Th***(and thereby also* **F** *and* **H***) is post-quantum single-function multi-target-collision resistant for distinct tweaks (with tweak advice)*
- **F** *is post-quantum single-function multi-target decisional second-preimage resistant for distinct tweaks (with tweak advice)*
- **PRF** *and* $\mathbf{PRF_{msg}}$ *are post-quantum pseudorandom function families, and*
- $\mathbf{H_{msg}}$ *is post-quantum interleaved target subset resilient*

More concretely,

$$\begin{aligned}
&\mathrm{InSec}^{\textit{PQ-EU-CMA}}(\mathrm{SPHINCS}^+; \xi, q_s) \\
&\leq \mathrm{InSec}^{\textit{PQ-PRF}}(\mathbf{PRF}; \xi, q_1) + \mathrm{InSec}^{\textit{PQ-PRF}}(\mathbf{PRF_{msg}}; \xi, q_s) \\
&+ \mathrm{InSec}^{\textit{PQ-ITSR}}(\mathbf{H_{msg}}; \xi, q_s) + \mathrm{InSec}^{\textit{PQ-SM-TCR}}(\mathbf{Th}; \xi, q_2) \\
&+ 3 \cdot \mathrm{InSec}^{\textit{PQ-SM-TCR}}(\mathbf{F}; \xi, q_3) + \mathrm{InSec}^{\textit{PQ-SM-DSPR}}(\mathbf{F}; \xi, q_3) \ ,
\end{aligned}$$

where $q_1 < 2^{h+1}(kt + l)$, $q_2 < 2^{h+2}(w \cdot l + 2kt)$, and $q_3 < 2^{h+1}(kt + w \cdot l)$.

Following the security evaluation methodology of SPHINCS$^+$ we first estimate the above insecurity levels using the generic attacks fixing the number of signatures (and thus the number of oracle queries). Then, we calculate the computational complexity of an adversary in order to get a constant success probability in winning the EU-CMA game. We therefore review the generic attacks in the following subsection.

5.3 Concrete Security

We follow the route of the previous works [HRS16b, BH19] to derive the bounds on PQ-SM-TCR, PQ-SM-DSPR, and PQ-PRF as $\mathcal{O}(q + 1/2^\lambda)$ (classical) and $\mathcal{O}((q+1)^2/2^\lambda)$ (quantum).

With our revised few-time signature FORC, we slightly adapt the definition of PQ-ITSR from [BHK$^+$19] and adjust the collision probability in the analysis of its InSec$^{\text{PQ-ITSR}}$ accordingly. It captures the generic attacks against interleaved target subset resilience (ITSR), i.e., a new message digest selects FORC positions that are covered (and whose values are efficiently implied) by the q_s signature queries.

Definition 7 (PQ-ITSR) *Let $H : \{0,1\}^\kappa \times \{0,1\}^\alpha \rightarrow \{0,1\}^m$ be a keyed hash function. Further consider the mapping function $MAP_{h,k,t} : \{0,1\}^m \rightarrow \{0,1\}^h \times ([t] \times [w'])^k$ which, for parameters h, k, t, w', maps an m-bit string to a set of k indexes $\{(I, 1, J_1, L_1), (I, 2, J_2, L_2), \ldots, (I, k, J_k, L_k)\}$, where I is chosen from $[2^h]$ and each (J_i, L_i) pair is chosen from $[t] \times [w']$. Note that the same I is used for all tuples (I, i, J_i, L_i).*

We call a pair (J, L) covered by another pair (J', L') if $J = J'$ and $L \leq L'$, and a set S_0 is covered by another set S_1 if they have the same I, and all (J, L) pairs of S_0 are covered by those of S_1.

We define the success probability of any (quantum) adversary A against $ITSR$ of H. Let $G = MAP_{h,k,t} \circ H$. The definition uses an oracle $O(\cdot)$ which on input an α-bit message M_i samples a key $K_i \leftarrow \{0,1\}^\kappa$ and returns K_i and $G(K_i, M_i)$. The adversary can query this oracle with messages of his choice.

$$\begin{aligned}
&\mathrm{Succ}^{ITSR}_{H,q}(A) = \Pr[(K, M) \leftarrow A^{O(\cdot)}(1^n) \\
&\textit{s.t. } G(K, M) \textit{ covered by } \bigcup_{j=1}^{q} G(K_j, M_j) \wedge (K, M) \notin \{(K_j, M_j)\}_{j=1}^{q}],
\end{aligned}$$

where q denotes the number of oracle queries of A and the pairs $\{(K_j, M_j)\}_{j=1}^{q}$ represent the responses of oracle O.

The PQ-ITSR insecurity of a keyed hash function against q-query, time-ξ adversaries is the maximum advantage of any quantum adversary A with running time $\leq \xi$, and making up to q queries:

$$\mathrm{InSec}^{PQ\text{-}ITSR}(H;\xi,q) = \max_A \mathrm{Succ}_{H,q}^{ITSR}(A) \ .$$

Following SPHINCS$^+$ we assume idealized hash functions when evaluating concrete values for $\mathrm{InSec}^{\text{PQ-ITSR}}$. Recall the parameters h, k, $t = 2^a$ and w'. The process is as follows: generate independent uniformly random integers $I, J_1, L_1, J_2, L_2, \ldots, J_k, L_k$. In the context of SPHINCS-α , I selects the FORC instance, and each (J_i, L_i) pair selects the position of the value revealed from the i-th set inside this FORC instance.

Notice that the position in the FORC is now specified by (J, L) (instead of just J in [BHK$^+$19]). Thus, a position (J, L) is covered by another one (J', L') iff $J = J'$ and $L \leq L'$, which occurs with probability

$$\Pr[\mathsf{Col}] = \Pr[J = J'] \cdot \Pr[L \leq L'] = \frac{1}{t}\Big(\frac{1}{w'}\sum_{i=1}^{w'}\frac{i}{w'}\Big) = \frac{1}{t}\cdot\frac{w'+1}{2w'} \ .$$

Let q_s denote the number of adversarial signature queries. Let S_i be the generated set by the i-th signing processing. The probability that S_0 is covered by the union of $S_1, S_2, \ldots, S_{q_s}$ is shown in [BHK$^+$19] as

$$\sum_{\gamma=0}^{q_s}(1-(1-\Pr[\mathsf{Col}])^{\gamma})^k \binom{q_s}{\gamma}\left(1-\frac{1}{2^h}\right)^{q_s-\gamma}\frac{1}{2^{h\gamma}} \ .$$

We substitute $\Pr[\mathsf{Col}] = \frac{1}{t}\cdot\frac{w'+1}{2w'}$ into the above to get

$$\sum_{\gamma=0}^{q_s}\left(1-\left(1-\frac{1}{t}\cdot\frac{w'+1}{2w'}\right)^{\gamma}\right)^k \binom{q_s}{\gamma}\left(1-\frac{1}{2^h}\right)^{q_s-\gamma}\frac{1}{2^{h\gamma}} \ .$$

In summary, the overall insecurity for classical adversaries that make no more than q_h queries to the underlying hash function is

$$\mathrm{InSec}^{\text{EU-CMA}}(\text{SPHINCS-}\alpha; q_h) \leq$$
$$\mathcal{O}\left(\frac{q_h}{2^n} + q_h\sum_{\gamma=0}^{q_s}\left(1-\left(1-\frac{1}{t}\cdot\frac{w'+1}{2w'}\right)^{\gamma}\right)^k \binom{q_s}{\gamma}\left(1-\frac{1}{2^h}\right)^{q_s-\gamma}\frac{1}{2^{h\gamma}}\right).$$

Similarly with quantum adversaries, this is

$$\mathrm{InSec}^{\text{PQ-EU-CMA}}(\text{SPHINCS-}\alpha; q_h) \leq$$
$$\mathcal{O}\left(\frac{q_h^2}{2^n} + q_h^2\sum_{\gamma=0}^{q_s}\left(1-\left(1-\frac{1}{t}\cdot\frac{w'+1}{2w'}\right)^{\gamma}\right)^k \binom{q_s}{\gamma}\left(1-\frac{1}{2^h}\right)^{q_s-\gamma}\frac{1}{2^{h\gamma}}\right).$$

A small constant factor is hidden in the $\mathcal{O}$-notation. To estimate the security, one sets the bound to 1 and solves for q_h.

5.4 Implementation

We describe implementation details of the SPHINCS-α signature scheme in this section.

Implementation. We adapt the official SPHINCS$^+$ implementation on github [Tea21] to ours [aT21], where we reuse most of its basic modules such as hash functions, and

implement from scratch only those added, i.e., the encoding algorithm. Therefore, this facilitates a fair comparison of the performance.

Instantiation Similar to SPHINCS$^+$, we provide 36 combinations of parameter choices and instantiations. The classic security level includes 128,192 or 256 bits. The hash functions can be haraka [KLMR16], shake256 [Dwo15] or sha256 [D$^+$15]. The tweakable hash function has simple or robust version. We also offer a small or fast option towards either small signatures or fast signature generation. Note that the instantiations using haraka cannot reach the same security levels like shake256 or sha256, due to a generic meet-in-the-middle attack.

Parameter Sets. The parameter sets are listed in Table 3. Note "bitsec" represents classic security level. Readers can also find the parameter estimation code in our open source implementation. Please open para.ipynb in Jupyter Notebook with SageMath.

Table 3: Parameter Sets for SPHINCS-α

Parameter Set	n	h	d	$\log t$	k	w	w'	bitsec
sphincs-α-128f	128	66	22	7	23	13	2	127
sphincs-α-192f	192	66	22	7	37	15	2	192
sphincs-α-256f	256	64	16	8	48	16	2	255
sphincs-α-128s	128	64	8	13	11	32	2	127
sphincs-α-192s	192	64	8	12	19	38	2	192
sphincs-α-256s	256	63	9	12	27	46	2	255

Environment. 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.

Performance. We briefly report the performance of SPHINCS-α with SPHINCS$^+$ as a baseline in Table 5 and Table 6 respectively. All reported instances are optimized using architecture-specific instructions such as AESNI or AVX2.

As shown in Table 1, we summarize a performance comparison between SPHINCS$^+$ and SPHINCS-α when instantiating the simple tweakable hash function with sha256. In addition, we also provide in Table 4 a theoretic performance comparison in terms of the total number of hash function calls, where count only the dominating hash functions $\mathbf{F}$ and $\mathbf{H}$ (and omitting others like $\mathbf{PRF}, \mathbf{PRF_{msg}}, \mathbf{H_{msg}}$, and $\mathbf{Th}_l$). Note that the real performance (in terms of the number of CPU cycles) mostly agrees with the theoretic estimate.

Table 4: Theoretic performance comparison between the signing algorithms in SPHINCS$^+$ and SPHINCS-α, in term of the number of hash calls and signature size.

	SPHINCS$^+$		SPHINCS-α		Relative Change	
Param.	#Hash	Size	#Hash	Size	#Hash	Size
128f	102960	17088	93664	17040	−9.03%	−0.28%
192f	160688	35664	149024	35640	−7.26%	−0.07%
256f	327696	49856	307456	49696	−6.18%	−0.32%
128s	2125312	7856	2041856	6960	−3.93%	−11.41%
192s	3485184	16224	3192832	14784	−8.39%	−8.88%
256s	2918400	29792	2876544	27104	−1.43%	−9.02%

We refer to Table 5 and Table 6 for comprehensive performance summaries of SPHINCS$^+$ and SPHINCS-α respectively for all the 36 parameter sets. As summarized in Table 7, SPHINCS-α reduces both signing time and signature size for most parameter sets. We pay a price of up to 122% increase in verification time, which seems worthwhile since verification only takes 1-9% of the signing time.

6 Conclusion

In this paper, we improve the performance of state-of-the-art stateless hash-based signature. This is achieved by improving/modifying the underlying components WOTS$^+$ and FORS, and optimizing the parameter choices. The resulting scheme, which we call SPHINCS-α, yields a general improvement in most parameter settings.

References

[ACZ18] Dorian Amiet, Andreas Curiger, and Paul Zbinden. FPGA-based accelerator for SPHINCS-256. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):18–39, 2018. `https://tches.iacr.org/index.php/TCHES/article/view/831`.

[AE18] 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.

[And76] Désiré André. Mémoire sur les combinaisons régulières et leurs applications. In *Annales scientifiques de l'École Normale Supérieure*, volume 5, pages 155–198, 1876.

[aT21] The SPHINCS alpha Team. sphincs-a. `https://github.com/sphincs-alpha/sphincs-a`, 2021.

[BC93] 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.

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 117–129, Tapei, Taiwan, November 29 – December 2 2011. Springer, Heidelberg, Germany.

[BH17] Leon Groot Bruinderink and Andreas Hülsing. "Oops, I did it again" - security of one-time signatures under two-message attacks. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017: 24th Annual International Workshop on Selected Areas in Cryptography*, volume 10719 of *Lecture Notes in Computer Science*, pages 299–322, Ottawa, ON, Canada, August 16–18, 2017. Springer, Heidelberg, Germany.

[BH19] Daniel J. Bernstein and Andreas Hülsing. Decisional second-preimage resistance: When does SPR imply PRE? In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 33–62, Kobe, Japan, December 8–12, 2019. Springer, Heidelberg, Germany.

[BHH$^+$15] 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.

[BHK$^+$19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ 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.

[BHRv21] Joppe W. Bos, Andreas Hülsing, Joost Renes, and Christine van Vredendaal. Rapidly verifiable XMSS signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):137–168, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/8730`.

[BI20] 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.

[BM94] 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.

[BM96a] Daniel Bleichenbacher and Ueli M. Maurer. On the efficiency of one-time digital signatures. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology – ASIACRYPT'96*, volume 1163 of *Lecture Notes in Computer Science*, pages 145–158, Kyongju, Korea, November 3–7, 1996. Springer, Heidelberg, Germany.

[BM96b] Daniel Bleichenbacher and Ueli M Maurer. Optimal tree-based one-time digital signature schemes. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 361–374. Springer, 1996.

[CAD$^+$20] 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.

[D$^+$15] Quynh H Dang et al. Secure hash standard. 2015.

[Dil09] Robert P Dilworth. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*, pages 139–144. Springer, 2009.

[DSS05] C. Dods, Nigel P. Smart, and Martijn Stam. Hash based digital signature schemes. In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115, Cirencester, UK, December 19–21, 2005. Springer, Heidelberg, Germany.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[Eul01] 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.

[GM17] 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.

[Gol87] 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.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. Xmss: extended merkle signature scheme. In *RFC 8391*. IRTF, 2018.

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for xmss mt. In *International Conference on Availability, Reliability, and Security*, pages 194–208. Springer, 2013.

[HRS16a] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 9614 of *Lecture Notes in Computer Science*, pages 446–470, Taipei, Taiwan, March 6–9, 2016. Springer, Heidelberg, Germany.

[HRS16b] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 9614 of *Lecture Notes in Computer Science*, pages 387–416, Taipei, Taiwan, March 6–9, 2016. Springer, Heidelberg, Germany.

[Hül13] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *AFRICACRYPT 13: 6th International Conference on Cryptology in Africa*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188, Cairo, Egypt, June 22–24, 2013. Springer, Heidelberg, Germany.

[hz11] 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).

[KLMR16] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 - Efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology*, 2016(2):1–29, 2016. http://tosc.iacr.org/index.php/ToSC/article/view/563.

[Köl18] Stefan Kölbl. Putting wings on SPHINCS. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 205–226, Fort Lauderdale, Florida, United States, April 9–11 2018. Springer, Heidelberg, Germany.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, October 1979.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-micali hash-based signatures. In *RFC 8554*. IRTF, 2019.

[Mer90] 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.

[Per01] Adrian Perrig. The BiBa one-time signature and broadcast authentication protocol. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001: 8th Conference on Computer and Communications Security*, pages 28–37, Philadelphia, PA, USA, November 5–8, 2001. ACM Press.

[PZC+21] 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.

[RR02] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Margaret Batten and Jennifer Seberry, editors, *ACISP 02: 7th Australasian Conference on Information Security and Privacy*, volume 2384 of *Lecture Notes in Computer Science*, pages 144–153, Melbourne, Victoria, Australia, July 3–5, 2002. Springer, Heidelberg, Germany.

[Spe28] Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

[SZM20] Shuzhou Sun, Rui Zhang, and Hui Ma. Efficient parallelism of post-quantum signature scheme sphincs. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2542–2555, 2020.

[Tea21] The SPHINCS+ Team. Sphincs+. `https://github.com/sphincs/sphincsplus`, 2021.

[Vau93] Serge Vaudenay. One-time identification with low memory. In *Eurocode'92*, pages 217–228. Springer, 1993.

[War97] S Ole Warnaar. The andrews–gordon identities and q-multinomial coefficients. *Communications in mathematical physics*, 184(1):203–232, 1997.

[WJW+19] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 523–550, Waterloo, ON, Canada, August 12–16, 2019. Springer, Heidelberg, Germany.

A More Detailed Comparisons

We benchmarked the performance of SPHINCS-α under all 36 parameter settings ($\{\text{haraka}, \text{shake256}, \text{sha256}\} \times \{128, 192, 256\} \times \{\text{fast}, \text{small}\} \times \{\text{robust}, \text{simple}\}$). To facilitate a fair comparison, we tested our implementation (adapted from the SPHINCS$^+$ codes) along with the original SPHINCS$^+$. The test results are reported in Table 5 and Table 6 with a comparison in Table 7.

Table 5: Runtime benchmarks for optimized 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	Tweakable hash	Key generation	Signing	Verification	Pk. Size	Sk. Size	Sig. size
sphincs-haraka-128f	robust	443556	5705136	512676	32	64	17088
sphincs-haraka-128f	simple	380016	4924638	419454	32	64	17088
sphincs-haraka-192f	robust	809640	12774942	902610	48	96	35664
sphincs-haraka-192f	simple	590814	8668764	645066	48	96	35664
sphincs-haraka-256f	robust	1104750	26008722	980586	64	128	49856
sphincs-haraka-256f	simple	1397070	17004978	641934	64	128	49856
sphincs-haraka-128s	robust	14450958	119774736	235890	32	64	7856
sphincs-haraka-128s	simple	11825712	94249962	170424	32	64	7856
sphincs-haraka-192s	robust	26697672	300607686	408006	48	96	16224
sphincs-haraka-192s	simple	19067202	196759800	261360	48	96	16224
sphincs-haraka-256s	robust	17549622	283877928	586008	64	128	29792
sphincs-haraka-256s	simple	11645640	187857558	369702	64	128	29792
sphincs-shake256-128f	robust	4161078	96265278	7285968	32	64	17088
sphincs-shake256-128f	simple	2140128	50028048	3450348	32	64	17088
sphincs-shake256-192f	robust	5990310	153338490	9922194	48	96	35664
sphincs-shake256-192f	simple	6262956	78820344	5035356	48	96	35664
sphincs-shake256-256f	robust	16049250	314417178	10418976	64	128	49856
sphincs-shake256-256f	simple	9390924	166614426	4989636	64	128	49856
sphincs-shake256-128s	robust	266192604	1999936872	2577816	32	64	7856
sphincs-shake256-128s	simple	136044954	1034544870	1312974	32	64	7856
sphincs-shake256-192s	robust	386893530	3366483696	3596472	48	96	16224
sphincs-shake256-192s	simple	197779788	1784871360	1869642	48	96	16224
sphincs-shake256-256s	robust	256996026	2912051970	5316408	64	128	29792
sphincs-shake256-256s	simple	130144878	1539752436	2585250	64	128	29792
sphincs-sha256-128f	robust	3911976	45293760	4078386	32	64	17088
sphincs-sha256-128f	simple	1995822	22989240	1899450	32	64	17088
sphincs-sha256-192f	robust	5888736	76478148	6478434	48	96	35664
sphincs-sha256-192f	simple	2768706	39118374	2949516	48	96	35664
sphincs-sha256-256f	robust	15628122	310998042	12441420	64	128	49856
sphincs-sha256-256f	simple	7213410	79211412	3085938	64	128	49856
sphincs-sha256-128s	robust	122212602	928916964	1495872	32	64	7856
sphincs-sha256-128s	simple	62257986	473796252	759060	32	64	7856
sphincs-sha256-192s	robust	184899276	1706237460	2642274	48	96	16224
sphincs-sha256-192s	simple	91933686	869077224	1240470	48	96	16224
sphincs-sha256-256s	robust	248710140	2912502672	6669810	64	128	29792
sphincs-sha256-256s	simple	60785262	780042312	1795986	64	128	29792

B Definitions

Definition 8 (PQ-EU-CMA) *Let* $\mathsf{SIG} = (\mathsf{kg}, \mathsf{sign}, \mathsf{vf})$ *be a digital signature scheme. We define the success probability of a quantum adversary* $\mathcal{A}$ *as*

$$\mathrm{Succ}^{EU\text{-}CMA}_{\mathsf{SIGN}}(\mathcal{A}) = \Pr[\mathsf{vf}(\mathsf{pk}, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q_s} | \\ (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}(); (M^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{sign}(\mathsf{sk}, \cdot)}] \ ,$$

where $M_1, ..., M_{q_s}$ *are the messages that* $\mathcal{A}$ *submitted to the signing oracle. We define the PQ-EU-CMA insecurity of* SIGN *as the maximum success probability among all quantum adversaries with* ξ *computing time and* q_s *query complexity:*

$$\mathrm{InSec}^{PQ\text{-}EU\text{-}CMA}(\mathsf{SIGN}; \xi, q_s) = \max_{\mathcal{A}}\{\mathrm{Succ}^{EU\text{-}CMA}_{\mathsf{SIGN}}(\mathcal{A})\} \ .$$

Definition 9 (PQ-PRF) *Let* $F : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ *be a keyed function and* $\mathcal{O} : \{0,1\}^\alpha \rightarrow \{0,1\}^n$ *be the set of all functions with domain* $\{0,1\}^\alpha$ *and range* $\{0,1\}^n$*. We define the advantage of a quantum adversary* $\mathcal{A}$ *as*

$$\mathrm{Adv}^{PRF}_F(\mathcal{A}) = \left| \Pr_{K \leftarrow \mathcal{K}}[\mathcal{A}^{F(K,\cdot)} = 1] - \Pr_{O \leftarrow \mathcal{O}}[\mathcal{A}^{O(\cdot)} = 1] \right| \ .$$

We define the PQ-PRF insecurity of F *as the maximum advantage among all quantum adversaries with* ξ *computing time and* q_s *query complexity:*

$$\mathrm{InSec}^{PQ\text{-}PRF}(F; \xi, q_s) = \max_{\mathcal{A}}\{\mathrm{Adv}^{PRF}_F(\mathcal{A})\} \ .$$

Table 6: Runtime benchmarks for optimized 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	Tweakable hash	Key generation	Signing	Verification	Pk. Size	Sk. Size	Sig. size
sphincs-α-haraka-128f	robust	393246	5580054	509238	32	64	17040
sphincs-α-haraka-128f	simple	325800	4418550	410184	32	64	17040
sphincs-α-haraka-192f	robust	408276	11190078	951408	48	96	35640
sphincs-α-haraka-192f	simple	565146	7753266	717264	48	96	35640
sphincs-α-haraka-256f	robust	2089836	23245110	1086318	64	128	49696
sphincs-α-haraka-256f	simple	1395198	15511896	728010	64	128	49696
sphincs-α-haraka-128s	robust	11066076	121551804	338364	32	64	6960
sphincs-α-haraka-128s	simple	8993628	96050520	278406	32	64	6960
sphincs-α-haraka-192s	robust	23202630	223899948	642960	48	96	14784
sphincs-α-haraka-192s	simple	16920270	157002696	483390	48	96	14784
sphincs-α-haraka-256s	robust	17758350	212131746	1072116	64	128	27104
sphincs-α-haraka-256s	simple	11742912	139256118	744120	64	128	27104
sphincs-α-shake256-128f	robust	5323986	88214796	5930352	32	64	17040
sphincs-α-shake256-128f	simple	1806174	44921592	3032856	32	64	17040
sphincs-α-shake256-192f	robust	5619420	139043214	9267030	48	96	35640
sphincs-α-shake256-192f	simple	5848578	77229594	4998438	48	96	35640
sphincs-α-shake256-256f	robust	15730344	290428074	9726318	64	128	49696
sphincs-α-shake256-256f	simple	8182728	152799588	4975470	64	128	49696
sphincs-α-shake256-128s	robust	205750530	1945905372	4163886	32	64	6960
sphincs-α-shake256-128s	simple	101552796	986881392	2129256	32	64	6960
sphincs-α-shake256-192s	robust	345090096	3017876580	6685596	48	96	14784
sphincs-α-shake256-192s	simple	183203352	1623353148	3490110	48	96	14784
sphincs-α-shake256-256s	robust	264304782	2743480044	11130858	64	128	27104
sphincs-α-shake256-256s	simple	134336934	1418581440	5736798	64	128	27104
sphincs-α-sha256-128f	robust	3144024	41544720	3504726	32	64	17040
sphincs-α-sha256-128f	simple	1727604	21354462	1709622	32	64	17040
sphincs-α-sha256-192f	robust	4067424	69338052	6151158	48	96	35640
sphincs-α-sha256-192f	simple	1364364	35095410	2904768	48	96	35640
sphincs-α-sha256-256f	robust	15402294	291966840	12457404	64	128	49696
sphincs-α-sha256-256f	simple	3760272	72368298	3138498	64	128	49696
sphincs-α-sha256-128s	robust	94289472	909234018	2333664	32	64	6960
sphincs-α-sha256-128s	simple	47681964	457985448	1202958	32	64	6960
sphincs-α-sha256-192s	robust	162853884	1452755592	4231278	48	96	14784
sphincs-α-sha256-192s	simple	80291520	722659932	2049930	48	96	14784
sphincs-α-sha256-256s	robust	258965946	2715656976	12925548	64	128	27104
sphincs-α-sha256-256s	simple	60974802	661839894	3318174	64	128	27104

Table 7: Performance comparison between optimized SPHINCS$^+$and SPHINCS-α in terms of relative changes.

Parameter Set	Parameter Set	Tweakable hash	Key generation	Signing	Verification	Sig. Size
sphincs-haraka-128f	sphincs-α-haraka-128f	robust	−11.34%	−2.19%	−0.67%	−0.28%
sphincs-haraka-128f	sphincs-α-haraka-128f	simple	−14.27%	−10.28%	−2.21%	−0.28%
sphincs-haraka-192f	sphincs-α-haraka-192f	robust	−49.57%	−12.41%	+5.41%	−0.07%
sphincs-haraka-192f	sphincs-α-haraka-192f	simple	−4.34%	−10.56%	+11.19%	−0.07%
sphincs-haraka-256f	sphincs-α-haraka-256f	robust	+89.17%	−10.63%	+10.78%	−0.32%
sphincs-haraka-256f	sphincs-α-haraka-256f	simple	−0.13%	−8.78%	+13.41%	−0.32%
sphincs-haraka-128s	sphincs-α-haraka-128s	robust	−23.42%	+1.48%	+43.44%	−11.41%
sphincs-haraka-128s	sphincs-α-haraka-128s	simple	−23.95%	+1.91%	+63.36%	−11.41%
sphincs-haraka-192s	sphincs-α-haraka-192s	robust	−13.09%	−25.52%	+57.59%	−8.88%
sphincs-haraka-192s	sphincs-α-haraka-192s	simple	−11.26%	−20.21%	+84.95%	−8.88%
sphincs-haraka-256s	sphincs-α-haraka-256s	robust	+1.19%	−25.27%	+82.95%	−9.02%
sphincs-haraka-256s	sphincs-α-haraka-256s	simple	+0.84%	−25.87%	+101.28%	−9.02%
sphincs-shake256-128f	sphincs-α-shake256-128f	robust	+27.95%	−8.36%	−18.61%	−0.28%
sphincs-shake256-128f	sphincs-α-shake256-128f	simple	−15.60%	−10.21%	−12.10%	−0.28%
sphincs-shake256-192f	sphincs-α-shake256-192f	robust	−6.19%	−9.32%	−6.60%	−0.07%
sphincs-shake256-192f	sphincs-α-shake256-192f	simple	−6.62%	−2.02%	−0.73%	−0.07%
sphincs-shake256-256f	sphincs-α-shake256-256f	robust	−1.99%	−7.63%	−6.65%	−0.32%
sphincs-shake256-256f	sphincs-α-shake256-256f	simple	−12.87%	−8.29%	−0.28%	−0.32%
sphincs-shake256-128s	sphincs-α-shake256-128s	robust	−22.71%	−2.70%	+61.53%	−11.41%
sphincs-shake256-128s	sphincs-α-shake256-128s	simple	−25.35%	−4.61%	+62.17%	−11.41%
sphincs-shake256-192s	sphincs-α-shake256-192s	robust	−10.80%	−10.36%	+85.89%	−8.88%
sphincs-shake256-192s	sphincs-α-shake256-192s	simple	−7.37%	−9.05%	+86.67%	−8.88%
sphincs-shake256-256s	sphincs-α-shake256-256s	robust	+2.84%	−5.79%	+109.37%	−9.02%
sphincs-shake256-256s	sphincs-α-shake256-256s	simple	+3.22%	−7.87%	+121.90%	−9.02%
sphincs-sha256-128f	sphincs-α-sha256-128f	robust	−19.63%	−8.28%	−14.07%	−0.28%
sphincs-sha256-128f	sphincs-α-sha256-128f	simple	−13.44%	−7.11%	−9.99%	−0.28%
sphincs-sha256-192f	sphincs-α-sha256-192f	robust	−30.93%	−9.34%	−5.05%	−0.07%
sphincs-sha256-192f	sphincs-α-sha256-192f	simple	−50.72%	−10.28%	−1.52%	−0.07%
sphincs-sha256-256f	sphincs-α-sha256-256f	robust	−1.45%	−6.12%	+0.13%	−0.32%
sphincs-sha256-256f	sphincs-α-sha256-256f	simple	−47.87%	−8.64%	+1.70%	−0.32%
sphincs-sha256-128s	sphincs-α-sha256-128s	robust	−22.85%	−2.12%	+56.01%	−11.41%
sphincs-sha256-128s	sphincs-α-sha256-128s	simple	−23.41%	−3.34%	+58.48%	−11.41%
sphincs-sha256-192s	sphincs-α-sha256-192s	robust	−11.92%	−14.86%	+60.14%	−8.88%
sphincs-sha256-192s	sphincs-α-sha256-192s	simple	−12.66%	−16.85%	+65.25%	−8.88%
sphincs-sha256-256s	sphincs-α-sha256-256s	robust	+4.12%	−6.76%	+93.79%	−9.02%
sphincs-sha256-256s	sphincs-α-sha256-256s	simple	+0.31%	−15.15%	+84.76%	−9.02%