SPHINCS: practical stateless hash-based signatures

Daniel J. Bernstein[1,3], Daira Hopwood[2], Andreas Hülsing[3], Tanja Lange[3], Ruben Niederhagen[3], Louiza Papachristodoulou[4], Peter Schwabe[4], and Zooko Wilcox O'Hearn[2] *

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Least Authority
3450 Emerson Ave.
Boulder, CO 80305–6452 USA
daira@leastauthority.com,zooko@leastauthority.com
[3] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
tanja@hyperelliptic.org, ruben@polycephaly.org, andreas.huelsing@googlemail.com
[4] Radboud University Nijmegen
Digital Security Group
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
louiza@cryptologio.org, peter@cryptojedi.org

Abstract. This paper introduces a high-security post-quantum stateless hash-based signature scheme that signs hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU.

Keywords: post-quantum cryptography, one-time signatures, few-time signatures, hypertrees, vectorized implementation

1 Introduction

It is not at all clear how to securely sign operating-system updates, web-site certificates, etc. once an attacker has constructed a large quantum computer:

- RSA and ECC are perceived today as being small and fast, but they are broken in polynomial time by Shor's algorithm. The polynomial is so small that scaling up to secure parameters seems impossible.
- Lattice-based signature schemes are reasonably fast and provide reasonably small signatures and keys for proposed parameters. However, their quantitative security levels are highly unclear. It is unsurprising for a lattice-based scheme to promise "100-bit" security for a parameter set in 2012 and to correct this promise to only "75-80 bits" in 2013 (see [18, footnote 2]). Furthermore, both of these promises are only against pre-quantum attacks, and it seems likely that the same parameters will be breakable in practice by quantum computers.
- Multivariate-quadratic signature schemes provide extremely short signatures, are reasonably fast, and in some cases have public keys short enough for typical applications. However, the long-term security of these schemes is even less clear than the security of lattice-based schemes.

* This work was supported by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005 and Veni 2013 project 13114. Permanent ID of this document: 5c2820cfddf4e259cc7ea1eda384c9f9. Date: 2014-05-27

– Code-based signature schemes provide short signatures, and in some cases have been studied enough to support quantitative security conjectures. However, the schemes that have attracted the most security analysis have keys of many megabytes, and would need even larger keys to be secure against quantum computers.

Hash-based signature schemes are perhaps the most attractive answer. These schemes are parametrized by a choice of hash function; many of these schemes offer security proofs relative to comprehensible, and plausible, properties of the hash function, properties that have not been broken even when the hash function is MD5. We do not mean to suggest that MD5 is a good choice of hash function; it is easy to make, and we recommend, much more conservative parameter choices. Hash-based signing is reasonably fast, even without hardware acceleration; verification is faster; signatures and keys are reasonably small. Even more, every signature scheme uses a cryptographic hash function; hash-based signatures need nothing but a cryptographic hash function.

However, every practical hash-based signature scheme in the literature is *stateful*. Signing reads a secret key and a message and generates a signature but also generates an updated secret key. This does not fit standard APIs; it does not even fit the standard *definition* of signatures in cryptography. If the update fails (for example, if a key is copied from one device to another, or backed up and later restored) then security disintegrates.

It has been known for many years that, as a theoretical matter, one can build hash-based signature schemes without a state. What we show in this paper is that high-security post-quantum stateless hash-based signature systems are *practical*, and in particular that they can sign hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU using parameters that provide 128bit security against quantum attacks. In particular, we

– introduce SPHINCS, a new method to do randomized tree-based stateless signatures;
– propose SPHINCS-256, a specific instantiation of SPHINCS that offers 128 bit of security against quantum attacks;
– introduce HORS with trees (HORST), an improvement of the HORS few-time signature scheme; and
– describe a fast vectorized implementation of SPHINCS-256.

Our construction, SPHINCS, is carefully designed so that its security can be based on weak standard-model assumptions, avoiding collision resistance and the random-oracle model.

Hash-based Signatures. The basic idea of a hash-based signature scheme as proposed by Merkle [28] is as follows. It starts with a one-time signature scheme (OTS), i.e. a signature scheme where a key pair must only be used once. To construct a many-time signature scheme, 2^h OTS key pairs are authenticated using a binary hash tree of height h. The leafs of this tree are the hashes of the OTS public keys. The OTS secret keys become the secret key of the new scheme and the root of the tree the public key. Such a key pair can be used to sign 2^h messages.

A signature of the many-time signature scheme is also called a *full* signature if necessary to distinguish it from other kinds of signatures. A full signature contains the index of the used OTS key pair in the tree, the OTS public key[5], the OTS signature, and the so called authentication path. This is the set of sibling nodes on the path from the OTS

[5] If a Winternitz style OTS is used, the OTS public key can be computed from the OTS signature. Hence, it can be omitted in the full signature

public key to the root. To guarantee that each OTS key pair is only used once, the OTS key pairs are used in a predefined order, using the leaves of the tree from left to right. To verify the signature, first the OTS signature on the message is verified. Afterwards, the authenticity of the OTS key pair is checked. Therefor, the hash of the OTS public key is computed and is used with the authentication path to compute a root value. The result is then compared to the public key.

This approach generates comparatively small signatures and keys (using pseudorandom key generation). However, key generation and signature time are exponential in h as the whole tree has to be built. Recent practical variants [24,14,17,15,16] solve these two problems. On the one hand, key generation time is significantly reduced using a hyper-tree of several layers of trees, i.e. a certification tree where a single hash-based key pair with height h_1 is used to sign the public keys of 2^{h_1} hash-based key pairs and so on. During key generation only one tree on each layer has to be generated. So, using d layers of trees with height h/d, the key generation time can be reduced from $\mathcal{O}(2^h)$ to $\mathcal{O}(d2^{h/d})$. On the other hand, signing time is reduced from $\mathcal{O}(2^h)$ to $\mathcal{O}(h)$ using stateful algorithms that exploit the ordered use of the OTS key pairs. When combined with hyper-trees, the ordered use of the trees allows to reduce the signing time even further to $\mathcal{O}(h/d)$.

From Stateful to Stateless. We are aware of a single proposal for stateless hash-based signatures: Goldreich [21] proposes to use a binary certification tree out of one-time signature keys. So, each OTS key pair corresponding to a non-leaf node is used to sign the hash of the public keys of its two child nodes. The leaf OTS key pairs are used to sign the messages. The OTS public key of the root becomes the overall public key. The secret key is a seed value that is used to pseudorandomly generate all the OTS key pairs of the tree.

Assume that the scheme is used to sign n-bit messages where n is the security parameter (hash-and-sign can be used to extend the message space). To ensure that each OTS key pair is never used to sign two different messages, a tree of height n is used. The leaf nodes are labeled from 0 to $2^n - 1$. To sign message M, its integer value m is computed and the leaf with index m is used to sign. The full signature contains all the OTS public keys in the path from leaf m to the root, all the public keys of the sibling nodes on this path, and the one-time signatures on the message and on the public keys in the path.

For this scheme, key generation requires a single OTS key generation. Signing takes $2n$ OTS key generations and n OTS signatures. This can be done in reasonable time for secure parameters. Keys are also very short, i.e. one OTS public key ($\mathcal{O}(n^2)$) for the public key and a single seed value ($\mathcal{O}(n)$) for the secret key. However, the signature size is far from being practical as they are cubic in the security parameter. For example, using the Winternitz OTS construction from [23], which has small signatures for a hash-based OTS, $n = 256$ as we do for SPHINCS-256, and some straight forward optimizations, the signature size is > 1 MB.

The SPHINCS Approach. To achieve practicality, SPHINCS takes a different approach to get rid of the state. More specifically, there are three main differences compared to Goldreich's construction. First, we show how to construct a stateless hash-based signature scheme with a total tree height h that is much smaller than n. Instead of deterministically using the message to determine the index of the OTS key pair, we select a random index (actually, we show how to securely derandomize this choice later, such that the SPHINCS signature algorithm is deterministic). With this change we can not ensure anymore that no OTS key pair is used to sign two or more messages. However, we

can choose h big enough that the probability that the same index is chosen twice is sufficiently small. While this change decouples the tree height from the security parameter, we gain nothing for $n = 256$ and targeted 128 bits of security as we do for SPHINCS-256.

Second, to benefit from the above change, we use the OTS key pairs on the bottom layer to sign the public keys of a hash-based few-time signature scheme (FTS). An FTS is a signature scheme where security degrades with every signature that an adversary learns. When combining the use of FTS with randomized index selection we can further reduce the tree height. We now only have to make sure that for all $\gamma \in \mathbb{N}$, the probability that the same index is used γ-times times the maximum success probability of an adversary that sees γ signatures is sufficiently small. For example, for a targeted 128 bits of security against quantum attackers, we can reduce the total tree height from 256 to 60 this way.

Goldreich's construction might be viewed as a hyper-tree construction with h layers of trees of height 0. The third change is that for SPHINCS we propose to use a hyper tree with fewer layers, thereby introducing a trade-off between signature size and time controlled by the number of layers d. Using a balanced hyper-tree with $d > 0$ layers where all trees have height h/d, signature size is $|\sigma| \approx d * |\sigma_{\mathrm{OTS}}| + h * n$ assuming a hash function with n-bit outputs. Recall that the size of a one-time signature $|\sigma_{\mathrm{OTS}}|$ is roughly $\mathcal{O}(n^2)$, hence decreasing the number of layers we get smaller full signatures. However, signing time increases exponentially in the decrease of layers. In detail, signing takes $d2^{h/d}$ OTS key generations and $d2^{h/d} - d$hash computations. So, this trade-off has to be used carefully.

We accompany our construction with a security reduction to some standard-model properties of hash functions. Towards parameter selection, we analyze the costs of generic attacks against these properties when the attacker has access to a large-scale quantum computer. For SPHINCS-256 we select parameters that provide 128 bits of security against quantum attackers and keep a balance between signature size and time.

HORST. As FTS for SPHINCS we introduce HORS with trees (HORST). As implied by the name, HORST is based on the hash-based FTS HORS [30]. HORS has a good performance in terms of speed but it has large keys and, when used in a hash-based signature scheme, the public key has to be part of each full signature. For message hashes of length m, HORS uses two parameters $t = 2^\tau$ for $\tau \in \mathbb{N}$ and $k \in \mathbb{N}$ such that $m = k \log t$. For practical secure parameters $t \gg k$. HORS uses a secret key consisting of t random values. The public key consists of the t hashes of these values. A signature consists of k secret key elements, selected through some mapping function.

HORST sacrifices runtime to reduce the public key size and the combined size of a signature and a public key. A HORST public key is the root node of a binary hash tree of height $\log t$, where the leaves are the public key elements of a HORS key. This reduces the public key size to a single hash value. For this to work, a HORST signature contains not only the k secret key elements but also one authentication path per secret key element. Now the public key can be computed given a signature. So, when used in a hash-based signature scheme, instead of including t hash values in the full signature, we only have to include $k \log t$. Indeed, we present some optimizations that further reduce this value. For the SPHINCS parameters this is a reduction of the FTS part of the full signature from 2^{16} hash values to less then $16 \cdot 32 = 2^9$ hash values. This is a reduction from 2 MB to 16 KB when using a 256-bit hash function.

The changes that we made can also be applied to HORS ++ [29], a variant of HORS that gives stronger security guarantees at the cost of bigger keys, to reduce the key size.

Vectorized software implementation. We describe an implementation of SPHINCS-256. Almost all hash computations in SPHINCS are highly parallel and we make extensive use of vector instructions. On an Intel Xeon E3-1275 (Haswell) CPU, parallel hashing runs at about 1.6 cycles/byte. Signing with SPHINCS-256 takes $47,466,005$ cycles; this corresponds to 294 signatures per second on all 4 cores of the 3.5 GHz CPU. Verification takes only $1,369,060$ cycles; key-pair generation takes $3,051,562$ cycles.

We placed the software described in this paper into the public domain to maximize reusability of our results. We submitted the software to eBACS [10] for independent benchmarking; the software is also available from `http://cryptojedi.org/crypto/#sphincs`.

Notation. We always use the logarithm with base 2 and hence write log instead of $\log_2$. We write $[x]$ for the set $0, \ldots, x$. Given a bit string x we write $x(i)$ for the ith bit of x and $x(i, j)$ for the j bit substring of x that starts with the ith bit.

2 The SPHINCS Construction

In this section we describe our main construction. We first start with a description of the main building blocks. Then we explain how key generation, signing and verification work. We first explain the used parameters. Then we describe the one-time signature scheme WOTS$^+$, binary hash trees, and the few-time signature scheme HORST.

Parameters. SPHINCS uses several parameters and several functions; the main security parameter is $n \in \mathbb{N}$. It uses two short-input cryptographic hash functions $\mathrm{F} : \{0,1\}^n \to \{0,1\}^n$ and $\mathrm{H} : \{0,1\}^{2n} \to \{0,1\}^n$, one arbitrary input randomized hash function $\mathcal{H} : \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^m$, for $m = \mathrm{poly}(n)$, a family of pseudorandom generators $\mathrm{G}_\lambda : \{0,1\}^n \to \{0,1\}^{\lambda n}$ for different values of λ, an ensemble of pseudorandom function families $\mathcal{F}_\lambda : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$ and a pseudorandom function family $\mathcal{F} : \{0,1\}^* \times \{0,1\}^n \to \{0,1\}^{2n}$ that supports arbitrary input lengths. SPHINCS uses a hyper-tree (a tree of trees) of total height $h \in \mathbb{N}$, where h is a multiple of d, and the hyper-tree consists of d layers of trees, each having height h/d.

The components of SPHINCS have additional parameters which influence performance and size of the signature and keys. The Winternitz one-time signature WOTS naturally allows for a space-time trade-off using the Winternitz parameter $w \in \mathbb{N}, w > 1$. The tree-based few-times signature scheme HORST (based on HORS) has a space-time trade-off which is controlled by two parameters $k \in \mathbb{N}$ and $t = 2^\tau$ with $\tau \in \mathbb{N}$ and $k\tau = m$. For SPHINCS-256 we use $n = 256, m = 512, h = 60, d = 12, w = 16, t = 2^{16}, k = 32$.

WOTS$^+$. We now describe the Winternitz one-time signature (WOTS$^+$) from [23]. We deviate slightly from the description in [23] to describe the algorithms as they are used in SPHINCS. Specifically, we include pseudorandom key generation and fix the message length to be n, meaning that a seed value takes the place of a secret key in our description. Given n and w, we define

$$\ell_1 = \left\lceil \frac{n}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

For the SPHINCS-256 parameters this leads to $\ell = 67$. WOTS$^+$ uses the function F to construct the following chaining function.

$c^i(x, \mathbf{r})$*:* On input of value $x \in \{0,1\}^n$, iteration counter $i \in \mathbb{N}$, and bitmasks $\mathbf{r} = (r_1, \ldots, r_j) \in \{0,1\}^{n \times j}$ with $j \geq i$, the chaining function works the following way. In case $i = 0$, c returns x ($c^0(x, \mathbf{r}) = x$). For $i > 0$ we define c recursively as

$$c^i(x, \mathbf{r}) = \mathrm{F}(c^{i-1}(x, \mathbf{r}) \oplus r_i),$$

i.e. in every round, the function first takes the bitwise xor of the intermediate value and bitmask r_i and evaluates F on the result. We write $\mathbf{r}_{a,b}$ for the substring $(r_a, \ldots, r_b)$ of $\mathbf{r}$. In case $b < a$ we define $\mathbf{r}_{a,b}$ to be the empty string. Now we describe the three algorithms of WOTS$^+$.

Key Generation Algorithm ($\mathsf{sk} \longleftarrow \mathrm{WOTS.}\mathsf{kg}(\mathcal{S}, \mathbf{r})$)*:* On input of seed $\mathcal{S} \in \{0,1\}^n$ and bitmasks $\mathbf{r} \in \{0,1\}^{n \times (w-1)}$ the key generation algorithm computes the internal secret key as $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell) \longleftarrow \mathrm{G}_\ell(\mathcal{S})$, i.e., the n bit seed is expanded to ℓ values of n bits. The public key pk is computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (c^{w-1}(\mathsf{sk}_1, \mathbf{r}), \ldots, c^{w-1}(\mathsf{sk}_\ell, \mathbf{r})).$$

Signature Algorithm ($\sigma \longleftarrow \mathrm{WOTS.}\mathsf{sign}(M, \mathcal{S}, \mathbf{r})$)*:* On input of an n-bit message M, seed $\mathcal{S}$ and the bitmasks $\mathbf{r}$, the signature algorithm first computes a base-w representation of M: $M = (M_1 \ldots M_{\ell_1})$, $M_i \in \{0, \ldots, w-1\}$. That is, M is treated as the binary representation of a natural number x and then the w-ary representation of x is computed. Next it computes the checksum

$$C = \sum_{i=1}^{\ell_1} (w - 1 - M_i)$$

and its base w representation $C = (C_1, \ldots, C_{\ell_2})$. The length of the base w representation of C is at most ℓ_2 since $C \leq \ell_1(w-1)$. We set $B = (b_1, \ldots, b_\ell) = M \parallel C$, the concatenation of the base w representations of M and C. Then the internal secret key is generated using $\mathrm{G}_\ell(\mathcal{S})$ the same way as during key generation. The signature is computed as

$$\sigma = (\sigma_1, \ldots, \sigma_\ell) = (c^{b_1}(\mathsf{sk}_1, \mathbf{r}), \ldots, c^{b_\ell}(\mathsf{sk}_\ell, \mathbf{r})).$$

Verification Algorithm ($\mathsf{pk}' \longleftarrow \mathrm{WOTS.}\mathsf{vf}(M, \sigma, \mathbf{r})$)*:* On input of an n-bit message M, a signature σ, and bitmasks $\mathbf{r}$, the verification algorithm first computes the b_i, $1 \leq i \leq \ell$ as described above. Then it returns:

$$\mathsf{pk}' = (\mathsf{pk}'_1, \ldots, \mathsf{pk}'_\ell) = (c^{w-1-b_1}(\sigma_1, \mathbf{r}_{b_1+1,w-1}), \ldots, c^{w-1-b_\ell}(\sigma_\ell, \mathbf{r}_{b_\ell+1,w-1})).$$

A formally correct verification algorithm would compare pk' to a given public key and output true on equality and false otherwise. In SPHINCS this comparison is delegated to the overall verification algorithm.

Binary Hash Trees. The central elements of our construction are full binary hash trees. We use the construction proposed in [17] shown in Figure 1.

In SPHINCS, a binary hash tree of height h always has 2^h leaves which are n bit strings L_i, $i \in [2^h - 1]$. Each node $N_{i,j}$, for $0 < j \leq h$, $0 \leq i < 2^{h-j}$, of the tree stores an n-bit string. To construct the tree, h bit masks $\mathbf{Q}_j \in \{0,1\}^{2n}$, $0 < j \leq h$, are used. Define $N_{i,0} = L_i$. The values of the internal nodes $N_{i,j}$ are computed as

$$N_{i,j} = \mathrm{H}((N_{2i,j-1} || N_{2i+1,j-1}) \oplus \mathbf{Q}_j).$$

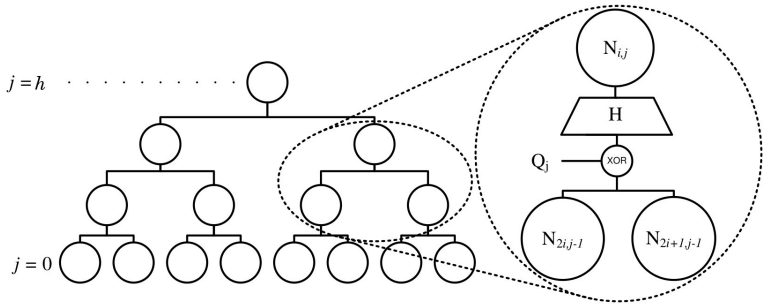

Fig. 1: The binary hash tree construction

We also denote the root as $\text{ROOT} = N_{0,h}$.

An important notion is the authentication path $\mathsf{Auth}_i = (\mathsf{A}_0, \ldots, \mathsf{A}_{h-1})$ of a leaf L_i shown in Figure 2. Auth_i consists of all the sibling nodes of the nodes contained in the path from L_i to the root. For a discussion on how to compute authentication paths, see Section 5. Given a leaf L_i together with its authentication path Auth_i, the root of the tree can be computed using the following algorithm:

Input: Leaf index i, leaf L_i, authentication path $\mathsf{Auth}_i = (\mathsf{A}_0, \ldots, \mathsf{A}_{h-1})$ for L_i.
Output: Root node ROOT of the tree that contains L_i.

Set $P_0 \leftarrow L_i$;
for $j \leftarrow 1$ **up to** h **do**

$$P_j = \begin{cases} \mathrm{H}((P_{j-1}||\mathsf{A}_{j-1}) \oplus \mathbf{Q}_j), & \text{if } \lfloor i/2^{j-1} \rfloor \equiv 0 \mod 2; \\ \mathrm{H}((\mathsf{A}_{j-1}||P_{j-1}) \oplus \mathbf{Q}_j), & \text{if } \lfloor i/2^{j-1} \rfloor \equiv 1 \mod 2; \end{cases}$$

end
return P_h

Algorithm 1: Root Computation

L-Tree. In addition to the full binary trees above, we also use unbalanced binary trees called L-Trees as in [17]. These are exclusively used to hash WOTS$^+$ public keys. The ℓ leaves of an L-Tree are the elements of a WOTS$^+$ public key and the tree is constructed as described above but with one difference: A left node that has no right sibling is lifted to a higher level of the L-Tree until it becomes the right sibling of another node. The L-Trees have height $\lceil \log \ell \rceil$ and hence need $\lceil \log \ell \rceil$ bitmasks.

2.1 HORST

Besides WOTS we also need a few-time signature scheme. For this purpose we introduce HORS-with-Tree (HORST). HORST signs messages of length m and uses parameters k and $t = 2^\tau$ with $k\tau = m$ (typical values as used in SPHINCS-256 are $t = 2^{16}, k = 32$). HORST improves HORS [30] using a binary hash-tree to reduce the public key size from tn bits to n bits[6] and the combined signature and public key size from tn bits to $(k(\log t - x + 1) + 2^x)n$ bits for some $x \in \mathbb{N}\backslash\{0\}$. The value x is determined based on t and k such that $k(\log t - x + 1) + 2^x$ is minimal. It might happen that the expression takes its minimum for two successive values. In this case the greater value is used. For SPHINCS-256 this results in $x = 6$.

In contrast to a one-time signature scheme like WOTS, HORS can be used to sign more than one message with the same key pair. However, with each signature the security

[6] Here we assume that the used bitmasks are given as they are used for several key pairs. Otherwise, public key size is $(2\tau + 1)n$ bit including bitmasks, which is still less then tn bits.

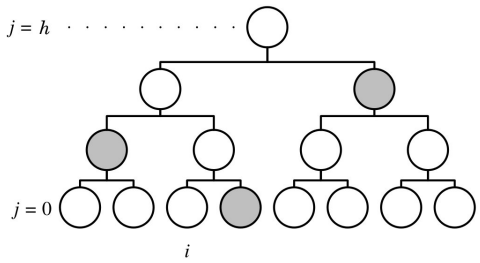

Fig. 2: The authentication path for leaf L_i

decreases. See Section 3 for more details. Like for WOTS$^+$ our description includes pseudorandom key generation. We now describe the algorithms for HORST:

Key Generation Algorithm ($\mathsf{pk} \longleftarrow \text{HORST.}\mathsf{kg}(\mathcal{S}, \mathbf{Q})$)*:* On input of seed $\mathcal{S} \in \{0,1\}^n$ and bitmasks $\mathbf{Q} \in \{0,1\}^{2n \times \log t}$ the key generation algorithm first computes the internal secret key $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_t) \longleftarrow \mathrm{G}_t(\mathcal{S})$. The public key pk is computed as the root node of a binary tree of height $\log t$. The leaves of the tree are computed as $L_i = \mathrm{F}(\mathsf{sk}_i)$ for $i \in [t-1]$ and the tree is constructed using bitmasks $\mathbf{Q}$.

Signature Algorithm ($(\sigma, \mathsf{pk}) \longleftarrow \text{HORST.}\mathsf{sign}(M, \mathcal{S}, \mathbf{Q})$)*:* On input of a message $M \in \{0,1\}^m$, seed $\mathcal{S} \in \{0,1\}^n$, and bitmasks $\mathbf{Q} \in \{0,1\}^{2n \times \log t}$ first the internal secret key sk is computed as described above. Then, let $M = (M_0, \ldots, M_{k-1})$ denote the k numbers obtained by splitting M into k strings of length $\log t$ bits each and interpreting each as an unsigned integer. The signature $\sigma = (\sigma_0, \ldots, \sigma_{k-1}, \sigma_k)$ consists of k blocks $\sigma_i = (\mathsf{sk}_{M_i}, \mathsf{Auth}_{M_i})$ for $i \in [k-1]$ containing the M_ith secret key element and the lower $\tau - x$ elements of the authentication path of the corresponding leaf $(\mathsf{A}_0, \ldots, \mathsf{A}_{\tau-1-x})$. The block σ_k contains all the 2^x nodes of the binary tree on level $\tau - x$ $(N_{0,\tau-x}, \ldots, N_{2^x-1,\tau-x})$. In addition to the signature, HORST.sign also outputs the public key.

Verification Algorithm ($\mathsf{pk}' \longleftarrow \text{HORST.}\mathsf{vf}(M, \sigma, \mathbf{Q})$)*:* On input of message $M \in \{0,1\}^m$, a signature σ, and bitmasks $\mathbf{Q} \in \{0,1\}^{2n \times \log t}$, the verification algorithm first computes the M_i, as described above. Then, for $i \in [k-1]$, $y_i = \lfloor M_i / 2^\tau - x \rfloor$ it computes $N'_{y_i,\tau-x}$ using Algorithm 1 with index M_i, $L_{M_i} = \mathrm{F}(\sigma_i^1)$, and $\mathsf{Auth}_{M_i} = \sigma_i^2$. It then checks that $\forall i \in [k-1] : N'_{y_i,\tau-x} = N_{y_i,\tau-x}$, i.e., that the computed nodes match those in σ_k. If all comparisons hold it uses σ_k to compute and then return $\textsc{Root}_0$, otherwise it returns fail.

Theoretical Performance. In the following we give rough theoretical performance values for HORST when used in a many-time signature scheme. We ignore the space needed for bitmasks, assuming they are provided. For runtimes we only count PRG calls and the number of hash evaluations without distinguishing the different hash functions.

Sizes: A HORST secret key consists of a single n bit seed. The public key contains a single n bit hash. A signature contains k secret key elements and authentication paths of length $(\log t) - x$ (Recall $t = 2^\tau$ is a power of two). In addition it contains 2^x nodes in σ_k, adding up to a total of $(k((\log t) - x + 1) + 2^x)n$ bits.

Runtimes: Key generation needs one evaluation of G_t and t hashes to compute the leaf values and $t-1$ hashes to compute the public key, leading to a total of $2t-1$. Signing takes the same time as we require the root is part of the output. Verification takes k

times one hash to compute a leaf value plus $(\log t) - x$ hashes to compute the node on level $(\log t) - x$. In addition, $2^x - 1$ hashes are needed to compute the root from σ_k. Together these are $k((\log t) - x + 1) + 2^x - 1$ hashes.

2.2 SPHINCS

Given all of the above we can finally describe the algorithms of the SPHINCS construction. A SPHINCS keypair completely defines a "virtual" structure which we explain first. SPHINCS works on a hyper-tree of height h that consists of d layers of trees of height h/d. Each of these trees looks as follows. The leaves of a tree are $2^{(h/d)}$ L-Tree root nodes that each compress the public key of a WOTS$^+$ key pair. Hence, a tree can be viewed as a key pair that can be used to sign $2^{(h/d)}$ messages. The hyper-tree is structured into d layers. On layer $d-1$ it has a single tree. On layer $d-2$ it has $2^{(h/d)}$ trees. The roots of these trees are signed using the WOTS$^+$ key pairs of the tree on layer $d-1$. In general, layer i consists of $2^{(d-1-i)(h/d)}$ trees and the roots of these trees are signed using the WOTS$^+$ key pairs of the trees on layer $i+1$. Finally, on layer 0 each WOTS$^+$ key pair is used to sign a HORST public key. We talk about a "virtual" structure as all values within are determined choosing a seed and the bitmasks, and as the full structure is never computed. The seed is part of the secret key and used for pseudorandom key generation. For a better intuition, Appendix B shows the virtual structure for one SPHINCS signature.

We use a simple addressing scheme for pseudorandom key generation. An address is a bit string of length $a = \lceil \log(d+1) \rceil + (d-1)(h/d) + (h/d) = \lceil \log(d+1) \rceil + h$. The address of a WOTS$^+$ key pair is obtained by encoding the layer of the tree it belongs to as a $\log(d+1)$-bit string (using $d-1$ for the top layer with a single tree). Then, appending the index of the tree in the layer encoded as a $(d-1)(h/d)$-bit string (we number the trees from left to right, starting with 0 for the left-most tree). Finally, appending the index of the WOTS$^+$ key pair within the tree encoded as a (d/h)-bit string (again numbering from left to right, starting with 0). The address of the HORST key pair is obtained using the address of the WOTS$^+$ key pair used to sign its public key and place d as the layer value in the address string, encoded as $\lceil \log(d+1) \rceil$ bit string. To give an example: In SPHINCS-256, an address needs 64 bits.

Key Generation Algorithm $((\mathsf{SK}, \mathsf{PK}) \longleftarrow \mathsf{kg}(1^n))$*:* The key generation algorithm first samples two secret key values $(\mathsf{SK}_1, \mathsf{SK}_2) \in \{0,1\}^n \times \{0,1\}^n$. The value SK_1 is used for pseudorandom key generation. The value SK_2 is used to generate an unpredictable index in sign and pseudorandom values to randomize the message hash in sign. Also, p uniformly random n-bit values $\mathbf{Q} \xleftarrow{\$} \{0,1\}^{p \times n}$ are sampled as bitmasks where $p = \max\{w-1, 2(h + \lceil \log \ell \rceil), 2 \log t\}$. These bitmasks are used for all WOTS$^+$ and HORST instances as well as for the trees. In the following we use $\mathbf{Q}_{\text{WOTS}^+}$ for the first $w-1$ bitmasks (of length n) in $\mathbf{Q}$, $\mathbf{Q}_{\text{HORST}}$ for the first $2 \log t$, $\mathbf{Q}_{L-\text{Tree}}$ for the first $2\lceil \log \ell \rceil$, and $\mathbf{Q}_{\text{Tree}}$ for the $2h$ strings of length n in $\mathbf{Q}$ that follow $\mathbf{Q}_{L-\text{Tree}}$.

The remaining part of kg consists of generating the root node of the tree on layer $d-1$. Towards this end the WOTS$^+$ key pairs for the single tree on layer $d-1$ are generated. The seed for the key pair with address $A = (d-1||0||i)$ where $i \in [2^{(h/d)} - 1]$ is computed as $\mathcal{S}_A \longleftarrow \mathcal{F}_a(A, \mathsf{SK}_1)$, evaluating the PRF on input A with key SK_1. In general, the seed for a WOTS$^+$ key pair with address A is computed as $\mathcal{S}_A \longleftarrow \mathcal{F}_a(A, \mathsf{SK}_1)$ and we will assume from now on that these seeds are known to any algorithm that knows SK_1. The WOTS$^+$ public key is computed as $\mathsf{pk}_A \longleftarrow \text{WOTS}.\mathsf{kg}(\mathcal{S}_A, \mathbf{Q}_{\text{WOTS}^+})$. The ith leaf L_i of

the tree is the root of an L-Tree that compresses pk_A using bit masks $\mathbf{Q}_{L-\text{Tree}}$. Finally, a binary hash tree is built using the constructed leaves and its root node becomes PK_1.

The SPHINCS secret key is $\mathsf{SK} = (\mathsf{SK}_1, \mathsf{SK}_2, \mathbf{Q})$, the public key is $\mathsf{PK} = (\mathsf{PK}_1, \mathbf{Q})$. kg returns the key pair $((\mathsf{SK}_1, \mathsf{SK}_2, \mathbf{Q}), (\mathsf{PK}_1, \mathbf{Q}))$.

Signature Algorithm ($\Sigma \longleftarrow \mathsf{sign}(M, \mathsf{SK})$)*:* On input of a message $M \in \{0,1\}^*$ and secret key $\mathsf{SK} = (\mathsf{SK}_1, \mathsf{SK}_2, \mathbf{Q})$, sign computes a randomized message digest $D \in \{0,1\}^m$: First, a pseudorandom $R = (R_1, R_2) \in \{0,1\}^n \times \{0,1\}^n$ is computed as $R \longleftarrow \mathcal{F}(M, \mathsf{SK}_2)$. Then, $D \longleftarrow \mathcal{H}(R_1, M)$ is computed as the randomized hash of M using the first n bits of R as randomness. The later n bits of R are used to select a HORST keypair, computing an h bit index $i \longleftarrow \textsc{Chop}(R_2, h)$ as the first h bits of R_2.

Given index i, the HORST key pair with address $A_{\text{HORST}} = (d||i(0,(d-1)h/d)||i((d-1)h/d, h/d))$ is used to sign the message digest D, i.e., the first $(d-1)h/d$ bits of i are used as tree index and the remaining bits for the index within the tree. The HORST signature and public key $(\sigma_{\text{H}}, \mathsf{pk}_{\text{H}}) \longleftarrow (D, \mathcal{S}_{A_{\text{HORST}}}, \mathbf{Q}_{\text{HORST}})$ are computed using the HORST bitmasks and seed $\mathcal{S}_{A_{\text{HORST}}} \longleftarrow \mathcal{F}_a(A_{\text{HORST}}, \mathsf{SK}_1)$.

The full SPHINCS signature $\Sigma = (i, R_1, \sigma_{\text{H}}, \sigma_{\text{W},0}, \mathsf{Auth}_{A_0}, \ldots, \sigma_{\text{W},d-1}, \mathsf{Auth}_{A_{d-1}})$ contains besides index i, randomness R_1 and HORST signature σ_{H} also one WOTS$^+$ signature and one authentication path $\sigma_{\text{W},i}, \mathsf{Auth}_{A_i}$, $i \in [d-2]$ per layer. These are computed as follows: The WOTS$^+$ key pair with address A_0 is used to sign pk_{H}, where A_0 is the address obtained taking A_{HORST} and setting the first $\lceil \log(d+1) \rceil$ bits to zero. This is done running $\sigma_{\text{W},1} \longleftarrow (\mathsf{pk}_{\text{H}}, \mathcal{S}_{A_0}, \mathbf{Q}_{\text{WOTS}^+})$ using the WOTS$^+$ bitmasks. Then the authentication path $\mathsf{Auth}_{i((d-1)h/d,h/d))}$ of the used WOTS$^+$ key pair is computed. Next, the WOTS$^+$ public key $\mathsf{pk}_{\text{W},0}$ is computed running $\mathsf{pk}_{\text{W},0} \longleftarrow \text{WOTS}.\mathsf{vf}(\mathsf{pk}_{\text{H}}, \sigma_{\text{W},0}, \mathbf{Q}_{\text{WOTS}^+})$. The root node $\textsc{Root}_0$ of the tree is computed by first compressing $\mathsf{pk}_{\text{W},0}$ using an L-Tree. Then Algorithm 1 is applied using the index of the WOTS$^+$ key pair within the tree, the root of the L-Tree and $\mathsf{Auth}_{i((d-1)h/d,h/d))}$.

This procedure gets repeated for layers 1 to $d-1$ with the following two differences. On layer $1 \leq j < d$, WOTS$^+$ is used to sign $\textsc{Root}_{j-1}$, the root computed at the end of the previous iteration. The address of the WOTS$^+$ key pair used on layer j is computed as $A_j = (j||i(0,(d-1-j)h/d)||i((d-1-j)h/d, h/d))$, i.e. on each layer the last (h/d) bits of the tree address become the new leaf address and the remaining bits of the former tree address become the new tree address.

Finally, sign outputs $\Sigma = (i, R_1, \sigma_{\text{H}}, \sigma_{\text{W},0}, \mathsf{Auth}_{A_0}, \ldots, \sigma_{\text{W},d-1}, \mathsf{Auth}_{A_{d-1}})$.

Verification Algorithm ($b \longleftarrow \mathsf{vf}(M, \Sigma, \mathsf{PK})$)*:* On input of a message $M \in \{0,1\}^*$, a signature Σ, and a public key PK, the algorithm computes the message digest $D \longleftarrow \mathcal{H}(R_1, M)$ using the randomness R_1 contained in the signature. The message digest D and the HORST bitmasks $\mathbf{Q}_{\text{HORST}}$ are used to compute the HORST public key $\mathsf{pk}_{\text{H}} \longleftarrow \text{HORST}.\mathsf{vf}(D, \sigma_{\text{H}}, \mathbf{Q}_{\text{HORST}})$ from the HORST signature. If HORST.vf returns fail, verification returns false. The HORST public key in turn is used together with the WOTS$^+$ bit masks and the WOTS$^+$ signature to compute the first WOTS$^+$ public key $\mathsf{pk}_{\text{W},0} \longleftarrow \text{WOTS}.\mathsf{vf}(\mathsf{pk}_{\text{H}}, \sigma_{\text{W},0}, \mathbf{Q}_{\text{WOTS}^+})$. An L-Tree is used to compute the leaf $L_{i((d-1)h/d,h/d)}$ corresponding to $\mathsf{pk}_{\text{W},0}$. Then, the root $\textsc{Root}_0$ of the respective tree is computed using Algorithm 1 with index $i((d-1)h/d, h/d)$, leaf $L_{i((d-1)h/d,h/d)}$ and authentication path Auth_0.

Then, this procedure gets repeated for layers 1 to $d-1$ with the following two differences. First, on layer $1 \leq j < d$ the root of the previously processed tree $\textsc{Root}_{j-1}$ is used to compute the WOTS$^+$ public key $\mathsf{pk}_{\text{W},j}$. Second, the leaf computed from $\mathsf{pk}_{\text{W},j}$

using an L-Tree is $L_{i((d-1-j)h/d,h/d)}$, i.e., the index of the leaf within the tree can be computed cutting of the last $j(h/d)$ bits of i and then using the last (h/d) bits of the resulting bit string.

The result of the final repetition on layer $d-1$ is a value ROOT_{d-1} for the root node of the single tree on the top layer. This value is compared to the first element of the public key, i.e., $\mathsf{PK}_1 \stackrel{?}{=} \text{ROOT}_{d-1}$. If the comparison holds, vf returns true, otherwise it returns false.

Theoretical Performance. In the following we give rough theoretical performance values. We count runtimes only counting the number of PRF, PRG and hash evaluations without distinguishing the different PRFs, PRGs, and hash functions.

Sizes: A SPHINCS secret key consists of two n bit seeds and the $p = \max\{w-1, 2(h+\lceil\log\ell\rceil), 2\log t\}$ n bit bitmasks. Summing up to $(2+p)n$ bits. The public key contains a single n bit hash and the bitmasks: $(1+p)n$ bits. A signature contains one h bit index and a n bit randomness. Moreover, it contains a HORST signature $((k((\log t)-x+1)+2^x)n$ bits), d WOTS signatures (ℓn bits each), and a total of h authentication path nodes (n bits each). This gives a signature size of $((k((\log t)-x+1)+2^x)+d\ell+h+1)\,n+h$ bits.

Runtimes: SPHINCS key generation consists of building the top tree. This takes for leaf generation $2^{(h/d)}$ times: One PRF call, one PRG call, one WOTS$^+$ key generation (ℓw hashes), and one L-Tree ($\ell-1$ hashes). Building the tree adds another $2^{(h/d)}-1$ hashes. Together these are $2^{(h/d)}$ PRF and PRG calls and $(\ell(w+1))2^{(h/d)}-1$ hashes. Signing requires one PRF call to generate the index and the randomness for the message hash as well as the message hash itself. Then one PRF call to generate a HORST seed and a HORST signature. In addition, d trees have to be built, adding d times the time for key generation. The WOTS$^+$ signatures can be extracted while running WOTS$^+$ key generation, hence they add no extra cost. This sums up to $d2^{(h/d)}+2$ PRF calls, $d2^{(h/d)}+1$ PRG calls, and $2t+d((\ell(w+1))2^{(h/d)}-1)$ hashes. Finally, verification needs the message hash, one HORST verification, and d times a WOTS$^+$ verification ($<\ell w$ hashes), computing an L-Tree, and $h/d-1$ hashes to compute the root. This leads to a total of $k((\log t)-x+1)+2^x+d(\ell(w+1)-2)+h$ hashes.

3 Security Analysis

We now discuss the security of SPHINCS. We first give a reduction from standard hash function properties. Afterwards we discuss the best generic attacks on these properties using quantum computers. Definitions of the used properties can be found in Appendix A. For our security analysis we group the message hash and the mapping used within HORST to a function $\mathcal{H}_{k,t}$ that maps bit strings of arbitrary length to a subset of $\{0,...,t-1\}$ with at most k elements.

3.1 Security Reduction

We will now prove our main theorem which states that SPHINCS is secure as long as the used function (families) provide certain standard security properties. These properties are fulfilled by secure cryptographic hash functions, even against quantum attacks, as shown in the next subsection.

Theorem 1. *SPHINCS is existentially unforgeable under q_s-adaptive chosen message attacks if*

- F *is a second-preimage resistant, undetectable one-way function,*
- H *is a second-preimage resistant hash function,*
- G_λ *is a secure pseudorandom generator for values* $\lambda \in \{\ell, t\}$,
- $\mathcal{F}_\lambda$ *is a pseudorandom function family for* $\lambda = a$,
- $\mathcal{F}$ *is a pseudorandom function family, and*
- *for the subset-resilience of* $\mathcal{H}_{k,t}$ *it holds that*

$$\sum_{\gamma=1}^{\infty} \min\left\{2^{\gamma(\log q_s - h)+h}, 1\right\} \cdot \mathrm{Adv}_{\gamma\text{-}sr}(\mathcal{H}_{k,t}) = \mathrm{negl}(n)$$

where $\mathrm{Adv}_{\gamma\text{-}sr}$ *denotes the maximum over the success probability of all probabilistic polynomial-time adversary against the* γ*-subset resilience of* $\mathcal{H}_{k,t}$.

Proof. In the following we show that the success probability of any probabilistic polynomial-time adversary $\mathcal{A}$ that attacks the EU-CMA security of SPHINCS is negligible in the security parameter. First consider the following six games:

Game 1 is the original EU-CMA game against SPHINCS.

Game 2 differs from Game 1 in that the values R used to randomize the message hash and to choose the index i are chosen uniformly at random instead of using $\mathcal{F}$.

Game 3 is similar to Game 2 but this time all used WOTS$^+$ and HORST seeds are generated uniformly at random and stored in some list for reuse instead of generating them using $\mathcal{F}_a$.

Game 4 is similar to Game 3 but this time no pseudorandom key generation is used inside WOTS$^+$. Instead, all WOTS$^+$ secret key elements are generated uniformly at random and stored in some list for reuse.

Game 5 is similar to Game 4 but this time no pseudorandom key generation is used at all. Instead, also all HORST secret key elements are generated uniformly at random and stored in some list for reuse.

The difference in the success probability of $\mathcal{A}$ between playing Game 1 and Game 2 must be negligible. Otherwise we could use $\mathcal{A}$ as an distinguisher against the pseudorandomness of $\mathcal{F}$. Similarly, the difference in the success probability of $\mathcal{A}$ between playing Game 2 and Game 3 must be negligible. Otherwise, we could use $\mathcal{A}$ as an distinguisher against the pseudorandomness of $\mathcal{F}_a$. Also the difference in the success probability of $\mathcal{A}$ between playing Game 3 and Game 4 and playing Game 4 and Game 5 must be negligible. Otherwise, $\mathcal{A}$ could be used to distinguish the outputs of the PRG G_ℓ (resp. G_t) from uniformly random bit strings.

It remains to limit the success probability of $\mathcal{A}$ running in Game 5. Assume that $\mathcal{A}$ makes q_s queries to the signing oracle before outputting a valid forgery

$$M^*, \Sigma^* = (i^*, R^*, \sigma^*_{\mathrm{H}}, \sigma^*_{\mathrm{W},0}, \mathsf{Auth}^*_{A_0}, \ldots, \sigma^*_{\mathrm{W},d-1}, \mathsf{Auth}^*_{A_{d-1}}).$$

The index i^* was used to sign at least one of the query messages with overwhelming probability. Otherwise, $\mathcal{A}$ could be turned into an inverter for H that succeeds with non-negligible probability. Hence, we assume from now on i^* was used before. While running $\mathsf{vf}(M^*, \Sigma^*, \mathsf{PK})$ we can extract the computed HORST public key $\mathsf{pk}^*_{\mathrm{H}}$ as well as the computed WOTS$^+$ public keys $\mathsf{pk}^*_{\mathrm{W},j}$ and the root nodes of the trees containing

these WOTS$^+$ public keys ROOT^*_j for all levels $j \in [d-1]$. In addition, we compute the respective values pk_{H}, $\mathsf{pk}_{\text{W},j}$ and ROOT_j using the list of secret key elements. All required elements must be contained in the lists as i^* was used before.

Next we compare these values in reverse order of computation, i.e., starting with $\mathsf{pk}_{\text{W},d-1} \stackrel{?}{=} \mathsf{pk}^*_{\text{W},d-1}$, then $\text{ROOT}_{d-2} \stackrel{?}{=} \text{ROOT}^*_{d-2}$, and so forth. Then one of the following four mutually exclusive cases must appear:

Case 1: The first occurrence of a difference happens for a WOTS$^+$ public key. As shown in [17] this can only happen with negligible probability. Otherwise, we can use $\mathcal{A}$ to compute second-preimages for H with non-negligible success probability.

Case 2: The first difference occurs for two root nodes $\text{ROOT}_j \neq \text{ROOT}^*_j$. This implies a forgery for the WOTS$^+$ key pair used to sign ROOT_j. As shown in [23] this can only happen with negligible advantage. Otherwise, we could use $\mathcal{A}$ to either break the one-wayness, the second-preimage resistance, or the undetectability of F with non-negligible success probability.

Case 3: The first spotted difference is two different HORST public keys. As for Case 2, this implies a WOTS$^+$ forgery and can hence only appear with negligible probability.

Case 4: All the public keys and root nodes are equal, i.e. no difference occurs.

We already bound the probability for all cases but Case 4 which we analyze now. The analysis consists of a sequence of mutually exclusive cases. Please recall that the secret key elements for this HORST key pair are already fixed and contained in the secret value list as $i*$ was used in the query phase. First, we compare the values of all leaf nodes that can be derived from σ^*_{H} with the respective values derived from the list entries. These are the hashes of the secret key elements in the signature and the authentication path nodes for level 0. The case that there exists a difference can only appear with negligible probability, as otherwise $\mathcal{A}$ could be used to compute second-preimages for H with non-negligible probability following the proof in [17]. Hence, we assume from now on all of these are equal.

Second, the indices of the secret key values contained in σ^*_{H} have either all been published as parts of query signatures or at least one index has not been published before. The latter case can only appear with negligible probability. Otherwise, $\mathcal{A}$ could be turned into a preimage finder for F that has non-negligible success probability. Finally, we can limit the probability that all indices have been published as parts of previous signatures.

Recall, when computing the signatures on the query messages, the indices were chosen uniformly at random. Hence, the probability that a given index reoccurs γ times, i.e., is used for γ signatures, is equal to the probability of the event $C(2^h, q_s, \gamma)$ that after q_s samples from a set of size 2^h at least one value was sampled γ times. This probability can in turn be bound by

$$\mathsf{Pr}[C(2^h, q_s, \gamma)] \leq \frac{1}{2^{h(\gamma-1)}} \binom{q_s}{\gamma} \leq \frac{q_s^\gamma}{2^{h(\gamma-1)}} = 2^{\gamma(\log q_s - h)+h} \tag{1}$$

as shown in [31]. Using Equation 1, the probability for this last case can be written as

$$\sum_{\gamma=1}^{\infty} \min\left\{2^{\gamma(\log q_s - h)+h}, 1\right\} \cdot \text{Adv}_{\gamma\text{-sr}}(\mathcal{H}_{k,t}),$$

i.e. the sum over the probabilities that there exists at least one index that was used γ times multiplied by the γ-subset resilience of $\mathcal{H}_{k,t}$. This probability is negligible per

assumption. Hence the success probability of $\mathcal{A}$ is negligible which concludes the proof. □

3.2 Generic attacks

As a complement to the above reduction, we now analyze the concrete complexity of various attacks, both pre-quantum and post-quantum. Please recall that $\mathcal{H}_{k,t}(R, M)$ applied to message $M \in \{0,1\}^*$ and randomness $R \in \{0,1\}^n$ works as follows. First, the message digest is computed as $M' = \mathcal{H}(R, M) \in \{0,1\}^m$. Then, M' is split into k bit strings, each of length $\log t$. Finally, each of these bit strings is interpreted as an unsigned integer. Thus, the output of $\mathcal{H}_{k,t}$ is an ordered subset of k values out of the set $[t-1]$ (possibly with repetitions).

Subset-Resilience. The main attack vector against SPHINCS is targeting subset-resilience. The obvious first attack is to simply replace (R, M) in a valid signature with (R', M'), hoping that $\mathcal{H}_{k,t}(R, M) = \mathcal{H}_{k,t}(R', M')$. This violates strong unforgeability if $(R, M) \neq (R', M')$, and it violates existential unforgeability if $M \neq M'$. Finding a second preimage of (R, M) under $\mathcal{H}_{k,t}$ costs 2^m pre-quantum but only $2^{m/2}$ post-quantum (Grover's algorithm). Success probability drops quadratically with lower cost.

More generally, if $\mathcal{H}_{k,t}(R', M')$ contains the same indices as $\mathcal{H}_{k,t}(R, M)$, then the attacker can permute the HORST signature for (R, M) accordingly to obtain the HORST signature for (R', M'). If k^2 is considerably smaller than t then the k indices in a hash are unlikely to contain any collisions, so there are about $2^m/k!$ equivalence classes of hashes under permutations. It is easy to map each hash to a numerical representative of its equivalence class, effectively reducing the pre-quantum second-preimage cost from 2^m to $2^m/k!$, and the post-quantum second-preimage cost from $2^{m/2}$ to $\sqrt{2^m/k!}$.

More generally, when γ valid signatures use the same HORST key, the attacker can mix and match the HORST signatures. All the attacker needs is to break γ-subset-resilience: i.e., for the set of k indices in $\mathcal{H}_{k,t}(R', M')$ to be a subset of the union of the indices in the γ valid signatures. The union has size about γk (at most γk, and usually close to γk if γk is not very large compared to t), so a uniform random number has probability about $\gamma k/t$ of being in the union, and if the k indices were independent uniform random numbers then they would have probability about $(\gamma k)^k/t^k$ of all being in the union. The expected cost of a pre-quantum attack is about $t^k/(\gamma k)^k$, and the expected cost of a post-quantum attack is about $t^{k/2}/(\gamma k)^{k/2}$.

Of course, this attack cannot start unless the signer in fact produced γ valid signatures using the same HORST key. After a total of q signatures, the probability of any particular HORST key being used exactly γ times is exactly $\binom{q}{\gamma}(1-1/2^h)^{q-\gamma}(1/2^h)^\gamma$. This probability is bounded above by $(q/2^h)^\gamma$ in the above proof; a much tighter approximation is $(q/2^h)^\gamma \exp(-q/2^h)/\gamma!$. If the ratio $\rho = q/2^h$ is significantly smaller than 1 then there will be approximately q keys used once, approximately $\rho q/2$ keys used twice, approximately $\rho^2 q/6$ keys used three times, and so on. The chance that some key will be used γ times, for $\gamma = h/\log(1/\rho) + \delta$, is approximately $\rho^\delta/\gamma!$.

For example, consider $t = 2^{16}$, $k = 32$, $h = 60$, and $q = 2^{50}$. There is a noticeable chance, approximately $2^{-9.5}$, that some HORST key is used 6 times. For $\gamma = 6$ the expected cost of a pre-quantum attack is 2^{269} and the expected cost of a post-quantum attack is 2^{134}. Increasing γ to 9 reduces the post-quantum cost below 2^{128}, specifically to $2^{125.3}$, but the probability of a HORST key being used 9 times is below 2^{-48}. Increasing γ to $10, 11, 12, 13, 14, 15$ reduces the cost to $2^{122.8}, 2^{120.6}, 2^{118.6}, 2^{116.8}, 2^{115.1}, 2^{113.5}$ respectively, but reduces the probability to $2^{-61}, 2^{-75}, 2^{-88}, 2^{-102}, 2^{-116}, 2^{-130}$ respectively.

Security degrades as q grows closer to 2^h. For example, for $q = 2^{60}$ the attacker finds $\gamma = 26$ with probability above 2^{-30}, and then a post-quantum attack costs only about 2^{100}. Of course, the signer is in control of the number of messages signed: for example, even if the signer's key is shared across enough devices to continuously sign 2^{20} messages per second, signing 2^{50} messages would take more than 30 years.

One-Wayness. The attacker can also try to work backwards from a hash output to an n-bit hash input that was not revealed by the signer (or a n-bit half of a $2n$-bit hash input where the other half was revealed by the signer). If the hash inputs were independent uniform random n-bit strings then this would be a standard preimage problem; generic pre-quantum preimage search costs 2^n, and generic post-quantum preimage search (again Grover) costs $2^{n/2}$.

The attacker can also merge preimage searches for n-bit-to-n-bit hashes. (For $2n$-bit-to-n-bit hashes the known n input bits have negligible chance of repeating.) For pre-quantum attacks the cost of generic T-target preimage attacks is well known to drop by a factor of T; here T is bounded by approximately 2^h (the exact bound depends on q), for a total attack cost of approximately 2^{n-h}. For post-quantum attacks the cost of generic T-target preimage attacks is a matter of dispute. It is well known that $2^{n/2}/\sqrt{T}$ quantum *queries* are necessary and sufficient (assuming $T < 2^{n/3}$), but there is overhead beyond the queries. An analysis of this overhead by Bernstein [8] concludes that all known post-quantum collision-finding algorithms cost at least $2^{n/2}$, implying that the post-quantum cost of multi-target preimage attacks is also $2^{n/2}$. For example, for $n = 256$ and $T = 2^{56}$ the best post-quantum attacks use only 2^{100} queries but still cost 2^{128}.

Second-Preimage Resistance. As for the message hash, finding a second preimage of either a message $M \in \{0,1\}^n$ under F or a message $\mathcal{M} \in \{0,1\}^{2n}$ under H costs 2^n pre-quantum and $2^{n/2}$ post-quantum (Grover's algorithm). Again, success probability drops quadratically with lower cost.

PRF, PRG, and Undetectability. The hash inputs are actually obtained from a chain of PRF outputs, PRG outputs, and lower-level hash outputs. The attacker can try to find patterns in these inputs, for example by guessing the PRF key, the PRG seed, or an input to F that ends up at a target value after a number of rounds of the chaining function. All generic attacks again cost 2^n pre-quantum and $2^{n/2}$ post-quantum.

Security of SPHINCS-256. SPHINCS-256 uses $n = 256, m = 512, h = 60, d = 12, w = 16, t = 2^{16}, k = 32$ as parameters. Hence, considering attackers that have access to a large scale quantum computer this means the following. Assuming the best attacks against the used hash functions are generic attacks. $\mathcal{H}_{k,t}$ provides 134 bit security regarding the subset resilience, F and H provide 128 bit security against preimage, second-preimage and in case of F undetectability attacks. Similarly, the used PRFs and PRGs provide 128 bits of security. Summing up, SPHINCS-256 provides 128 bits of security against post-quantum attackers.

4 SPHINCS-256

In addition to the general construction of SPHINCS, we propose a specific instantiation called SPHINCS-256. The parameters for SPHINCS-256 were selected with two goals in mind. On the one hand, 128 bit security against attackers with access to quantum computers. On the other hand, a good trade-off between signature size and speed.

The selected parameters are $n = 256, m = 512, h = 60, d = 12, w = 16, t = 2^{16}, k = 32$, leading to $\ell = 67, x = 6$, and $a = 64$. The security parameter n is determined by the first goal, and in turn determined the name SPHINCS-256. As we are not aware of a metric that allows to compare speed and size for signature schemes, there exists no optimal solution for the remaining parameters. Hence, the remaining parameters are our proposal for a good trade-off between speed and signature size. Depending on the application, different instantiations might be defined changing the remaining parameters in favor of either speed or signature size.

The second part of SPHINCS-256 are the instantiations of the used function (families). To reduce code size, we built everything out of a small code base. For the message hash and the PRFs we use BLAKE [3]. The BLAKE compression algorithm is based on the stream cipher ChaCha [6]. Hence, we make use of ChaCha12 as PRG. Finally, we construct the fixed input size hash functions F and H using the ChaCha permutation with standard constructions.

To be more detailed, the message hash and PRF functions are implemented as follows:

- For the message hash we use $\mathcal{H}(R, M) = \text{BLAKE-512}(R\|M)$ to compute the message digest of message M using randomness R,
- for the n bit output PRF we use $\mathcal{F}_a(A, K) = \text{BLAKE-256}(K\|A)$ to compute the output for address A and key K,
- for the $2n$ bit output PRF we use $\mathcal{F}(M, K) = \text{BLAKE-512}(K\|M)$ to compute the output for address A and key K.

Here we follow the proposal of the BLAKE authors to build a PRF ensemble.

As a stream cipher is the practical implementation of a PRG we can use plain ChaCha12 to implement G_λ. We use $\mathrm{G}_\lambda(\textsc{Seed}) = \text{ChaCha12}_{\textsc{Seed}}(0)_{0,\ldots,\lambda-1}$, i.e. we run ChaCha12 with key $\textsc{Seed}$ and initialization vector 0 and take the first λ output bits. For details on the decision to use ChaCha with 12 rounds see the discussion about choosing a permutation for F and H below.

For the fixed-size hash functions F and H we use the ChaCha permutation. Let $\pi_{ChaCha} : \{0,1\}^{512} \to \{0,1\}^{512}$ denote the ChaCha permutation, let C be the bytes of the ascii representation of "expand 32-byte to 64-byte state!" and let $\textsc{Chop}(M, i)$ be the function that returns the i first bits of the bit string M. Then, for 256 bit strings M_1, M_2 we implement F and H as

$$\begin{aligned} \mathrm{F}(M_1) &= \textsc{Chop}\left(\pi_{ChaCha}\left(M_1\|C\right), 256\right), \text{ and} \\ \mathrm{H}(M_1\|M_2) &= \textsc{Chop}\left(\pi_{ChaCha}\left(\pi_{ChaCha}\left(M_1\|C\right) \oplus \left(M_2\|0^{256}\right)\right), 256\right). \end{aligned}$$

For the reasoning behind this construction see the following subsection.

The parameters, functions, and resulting key and signature sizes of SPHINCS-256 are summarized in Table 1

4.1 Fast fixed-size hashing

The primary cost metric in the literature on cryptographic hash functions, for example in the SHA-3 competition, is performance for long inputs. However, what is most important for SPHINCS and hash-based signatures in general is performance for short inputs. The hashing in SPHINCS consists primarily of applying F to n-bit inputs and secondarily of applying H to $2n$-bit inputs. Recall that to achieve long-term security we take $n = 256$ for SPHINCS-256.

Table 1: SPHINCS-256 parameters and functions for the 128-bit post-quantum security level and resulting signature and key sizes.

Parameter	Value	Meaning
n	256	bitlength of hashes in HORST and WOTS
m	512	bitlength of the message hash
h	60	height of the hyper tree
d	12	layers of the hyper tree
w	16	Winternitz parameter used for WOTS signatures
t	2^{16}	number of secret-key elements of HORST
k	32	number of revealed secret-key elements per HORST signature
Functions		
Hash $\mathcal{H}$:	$\mathcal{H}(R, M) = \text{BLAKE-512}(R\|M)$	
PRF $\mathcal{F}_a$:	$\mathcal{F}_a(A, K) = \text{BLAKE-256}(K\|A)$	
PRF $\mathcal{F}$:	$\mathcal{F}(M, K) = \text{BLAKE-512}(K\|M)$	
PRG G_λ:	$\text{G}_\lambda(\text{SEED}) = \text{ChaCha12}_{\text{SEED}}(0)_{0,\ldots,\lambda-1}$	
Hash F:	$\text{F}(M_1) = \text{CHOP}(\pi_{ChaCha}(M_1\|C), 256)$	
Hash H:	$\text{H}(M_1\|M_2) = \text{CHOP}\left(\pi_{ChaCha}\left(\pi_{ChaCha}\left(M_1\|C\right) \oplus \left(M_2\|0^{256}\right)\right), 256\right)$	
Sizes		
Signature size:	41000 bytes	
Public-key size:	1056 bytes	
Private-key size:	1088 bytes	

As an analogy, stream ciphers are normally optimized for long outputs, but what is most important for many PRF applications is performance for short outputs. Short-input performance was emphasized in a recent PRF design [1] from Aumasson and Bernstein. We propose short-input performance as an interesting target for hash-function designers.

Designing a new hash function is not within the scope of this paper: we limit ourselves to evaluating the short-input performance of previously designed components that appear to have received adequate study. This section explains our selection of specific functions $\text{F} : \{0,1\}^n \to \{0,1\}^n$ and $\text{H} : \{0,1\}^{2n} \to \{0,1\}^n$ for $n = 256$.

Review of permutation-based cryptography. Rivest suggested strengthening the DES cipher by "whitening" the input and output: i.e., encrypting a block M under key (K, K_1, K_2) as $E_K(M \oplus K_1) \oplus K_2$, where E_K means DES using key K. Even and Mansour [20] suggested eliminating the original key K: i.e., encrypting a block M under key (K_1, K_2) as $E(M \oplus K_1) \oplus K_2$, where E is an *unkeyed* public permutation. Kilian and Rogaway [25, Section 4] suggested taking $K_1 = K_2$.

Combining all of these suggestions means encrypting M under key K as $E(M \oplus K) \oplus K$; see, e.g., [26], [7], and [19]. Trivial 0-padding allows M and K to be shorter than the block length of E: for example, the "Salsa20" cipher from [7] actually produces $E(K, M, C) \oplus (K, M, C)$, where C is a constant. The PRF security of Salsa20 is tightly equivalent to the PRF security of $E(K, M, C) \oplus (K, 0, 0)$, which in turn implies the PRF security of the "HSalsa20" stream cipher [9] obtained by truncating $E(K, M, C)$.

Bertoni, Daemen, Peeters, and Van Assche [11] proposed building cryptographic hash functions from unkeyed permutations, and later proposed a specific "Keccak" hash function. The "sponge" construction used in [11], and in Keccak, hashes a $(b-c)$-bit message K_1 to a $(b-c)$-bit truncation of $E(K_1, C)$, where C is a c-bit constant; hashes a $2(b-c)$-bit message (K_1, K_2) to a $(b-c)$-bit truncation of $E(E(K_1, C) \oplus (K_2, 0))$;etc. Sponges

have been reused in many subsequent designs and studied in many papers. We ended up selecting the sponge structure for both F and H.

Note that the single-block hash here, a truncation of $E(K_1, C)$, is the same as an encryption of a constant nonce using a truncated-$E(K_1, M, C)$ cipher. Of course, there is no logical connection between the PRF security of this cipher and (e.g.) second-preimage resistance, but designers use the same techniques to build E for either context: consider, for example, the reuse of the Salsa20 permutation in the "Rumba20" [5] compression function, the reuse of a tweaked version of the ChaCha20 permutation [6] in the "BLAKE" and "BLAKE2" [4] hash functions, and the reuse of the Keccak permutation in the "Keyak" [13] authenticated-encryption scheme.

Many other hash-function designs use input blocks as cipher keys, but in most cases the underlying ciphers use complicated "key schedules" rather than wrapping simple key addition around an unkeyed permutation. Both [7] and [12] state reasons to believe that unkeyed permutations provide the best performance-security tradeoff. Performance obviously played a large role in the selection of Salsa20/12 (Salsa20 reduced to 12 rounds) for the eSTREAM portfolio, the deployment of ChaCha20 in TLS [27], and the selection of Keccak as SHA-3. We did not find any non-permutation-based hash-function software competitive in performance with the permutation that we selected.

Choice of permutation for $n = 256$. A sponge function using a b-bit permutation E and a c-bit "capacity" takes $b - c$ bits in each input block and produces $b - c$ bits of output. We require $b - c \geq 256$ so that a single call to E hashes 256 bits to 256 bits (and two calls to E hash 512 bits to 256 bits). The attacker can compute preimages by guessing the c missing bits and applying E^{-1}, so we also require $c \geq 256$.

We considered using the Keccak permutation, which has $b = 1600$, but this is overkill: it takes as long to hash a 256-bit block as it does to hash a 1000-bit block. There is a scaled-down version of Keccak with $b = 800$, but this is not part of SHA-3, and we do not know how intensively it has been analyzed.

After considering various other permutations we settled on ChaCha, which has $b = 512$. ChaCha is a slightly modified version of Salsa, advertising faster diffusion and at the same time better performance. The best key-recovery attacks known are from Aumasson, Fischer, Khazaei, Meier, and Rechberger [2] and are slightly faster than 2^{256} operations against 8 rounds of Salsa and 7 rounds of ChaCha, supporting the security advertisement. The eSTREAM portfolio recommends 12 rounds of Salsa20 as having a "comfortable margin for security" so we selected 12 rounds of ChaCha.

The Salsa and ChaCha permutations are not designed to simulate ideal permutations: they are designed to simulate ideal permutations with certain symmetries, i.e., ideal permutations of the orbits of the state space under these symmetries. The Salsa and ChaCha stream ciphers add their inputs to only part of the block and specify the rest of the block as asymmetric constants, guaranteeing that different inputs lie in different orbits. For the same reason we specify an asymmetric constant for C.

5 Fast software implementation

The fastest arithmetic units of most modern microprocessors are vector units. Instead of performing a certain arithmetic operation on scalar inputs, they perform the same operation in parallel on multiple values kept in vector registers. Not surprisingly, many speed records for cryptographic algorithms are held by implementations that make efficient use of these vector units. Also not surprisingly, many modern cryptographic primitives

are designed with vectorizability in mind. In this section we describe how to efficiently implement SPHINCS-256 using vector instructions, more specifically the AVX2 vector instructions in Intel Haswell processors. All cycle counts reported in this section are measured on one core of an Intel Xeon E3-1275 CPU running at 3.5 GHz. We followed the standard practice of turning off Turbo Boost and hyper-threading for our benchmarks. The parameters and functions used for SPHINCS-256 as well as resulting signature size and key sizes are summarized in Table 1.

The AVX2 instruction set. The Advanced Vector Extensions (AVX) were introduced by Intel in 2011 with the Sandy Bridge microarchitecture. The extensions feature 16 vector registers of size 256 bits. In AVX, those registers can only be used as vectors of 8 single-precision floating-point values or vectors of 4 double-precision floating points values. This functionality was extended in AVX2, introduced with the Haswell microarchitecture, to also support arithmetic on 256-bit vectors of integers of various sizes. We use these AVX2 instructions for 8-way parallel computations on 32-bit integers.

Vectorizing hash computations. The two low-level operations in SPHINCS that account for most of the computations are the fixed-input-size hash functions F and H. The SPHINCS-256 instantiation of F and F internally uses the ChaCha permutation. The same permutation has already been used for in the SHA-3 finalist BLAKE [3]. The description of BLAKE discusses parallelism in [3, Section 3.1.3] and notes that the core operation "can be computed in four parallel branches". Most high-speed implementations of BLAKE use this 4-way parallelism for vector computations. It may seem obvious to also follow this approach in the implementation of F and H. However it is much more efficient to vectorize across multiple independent computations of F or H. The most obvious reason is that the ChaCha permutation operates on 32-bit integers which means that 4-way-parallel computation can only make use of half of the 256-bit AVX vector registers. A second reason is that internal vectorization of ChaCha requires relatively frequent shuffling of values in vector registers. Those shuffles do not incur a serious performance penalty, but they are noticeable. A third reason is that vectorization is not the only way how modern microprocessors exploit parallelism. Instruction-level parallelism is used on pipelined processors to hide latencies and superscalar CPUs can even execute multiple independent instruction in the same cycle. A non-vectorized implementation of ChaCha has 4-way instruction-level parallelism which makes very good use of pipelining and superscalar execution. A vectorized implementation of ChaCha has almost no instruction-level parallelism and suffers from serious instruction-latency penalties.

Finally, vectorizing across hash computations does not *rely* on hash-internal parallelism – it works with any hash function. The techniques described in this section thus also apply for hash-based signatures that use other instantiations of F and H. Our 8-way parallel implementation of F takes 405 cycles to hash 8 independent 256-bit inputs to 8 256-bit outputs. Our 8-way parallel implementation of H takes 801 cycles to hash 8 independent 512-bit inputs to 8 256-bit outputs. These speeds assume that the inputs are interleaved in memory. Interleaving and de-interleaving data means transposing an 8×8 32-bit-word matrix.

In the following we describe our approach to hash vectorization in the two main components in the SPHINCS signature generation: HORST signing and WOTS authentication-path computations.

HORST signing. The first step in HORST signature generation is to expand the secret seed into a stream of $t \cdot n = 16,777,216$ bits (or 2 MB). This pseudorandom stream forms the 2^{16} secret keys of HORST. We use the Chacha12 for this seed expansion, more

specifically the `moon` implementation of ChaCha12 by floodyberry, which SUPERCOP identifies as the fastest implementation for Haswell CPUs. The seed expansion costs about $1,650,000$ cycles.

The costly part of HORST signing is to first evaluate $\mathrm{F}(\mathsf{sk}_i)$ for $i = 0, \ldots, t-1$, and then build the binary hash tree on top of the $\mathrm{F}(\mathsf{sk}_i)$ and extract nodes which are required for the 32 authentication paths. SPHINCS-256 uses $t = 2^{16}$ so we need a total of $65,536$ evaluations of F and $65,535$ evaluations of H. A streamlined vectorized implementation treats the HORST binary tree up to level 13 (the level with 8 nodes) as 8 independent subtrees and vectorizes computations across these sub-trees. Data needs to be interleaved only once at the beginning (the HORST secret keys sk_i) and de-interleaved at the very end (the 8 nodes on level 13 of the HORST tree). All computations in between are streamlined 8-way parallel computations of F and H on interleaved data. The final tree hashing from level 13 to the HORST root at level 16 needs only 7 evaluations of H. This is a negligible cost, even when using a slower non-vectorized implementation of H.

Note that, strictly speaking, we do not have to interleave input at all; we can simply treat the 2 MB output of Chacha12 as already interleaved random data. However, this complicates compatible non-vectorized or differently vectorized implementations on other platforms.

WOTS authentication paths. Computing a WOTS authentication path consists of 32 WOTS key generations, each followed by an L-Tree computation. This produces 32 WOTS public keys, which form the leaves of a binary tree. Computing this binary tree and extracting the nodes required for the authentication path finishes this computation. The costly part of this operation is the computation of 32 WOTS public keys ($32 \cdot 15 \cdot 67 = 32,160$ evaluations of F) and of 32 L-Tree computations ($32 \cdot 66 = 2,112$ evaluations of H). For comparison, the binary tree at the end costs only 31 computations of H. Efficient vectorization parallelizes across 8 independent WOTS public-key computations with subsequent L-Tree computations. Data needs to be interleaved only once at the very beginning (the WOTS secret key) and de-interleaved once at the very end (the roots of the L-Trees). Again, all computations in between are streamlined 8-way parallel computations of F and H on interleaved data.

SPHINCS signing performance. Our software still uses some more transpositions of data than the almost perfectly streamlined implementation described above. With these transpositions and some additional overhead to xor hash inputs with masks, update pointers and indices etc., HORST signing takes $14,176,422$ cycles. The lower bound from $65,536$ evaluations of F and $65,535$ evaluations of H is $9,879,451$ cycles. The computation of one WOTS authentication path takes $2,391,574$ cycles. The lower bound from $32,160$ evaluations of F and $2,143$ evaluations of H is $1,842,667$ cycles. The complete SPHINCS-256 signing takes $47,466,005$ cycles; almost all of these cycles are explained by one HORST signature and 12 WOTS authentication paths.

SPHINCS key generation and verification. The by far most costly operation in SPHINCS is signing so we focused our optimization efforts on this operation. Some easily vectorizable parts of key generation and verification also use our high-speed 8-way vectorized implementations of F and H, but other parts still use relatively slow non-vectorized versions based on the ChaCha12 reference implementation in eBACS [10].

Our implementation of key generation takes $3,051,562$ cycles. Our implementation of signature verification takes $1,369,060$ cycles.

Acknowledgement Thanks to Michael Schneider, Christian Rechberger and Andrew Miller for helpful discussions on the topic.

References

1. Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012. 17
2. Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of Latin dances: Analysis of Salsa, ChaCha, and Rumba. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 470–488. Springer, 2008. 18
3. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST, 2008. http://131002.net/blake/blake.pdf. 16, 19
4. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013. 18
5. Daniel J. Bernstein. What output size resists collisions in a xor of independent expansions? ECRYPT Hash Workshop, 2007. 18
6. Daniel J. Bernstein. ChaCha, a variant of Salsa20. SASC 2008: The State of the Art of Stream Ciphers, 2008. 16, 18
7. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew J. B. Robshaw and Olivier Billet, editors, *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008. 17, 18
8. Daniel J. Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems, 2009. 15
9. Daniel J. Bernstein. Extending the Salsa20 nonce. Symmetric Key Encryption Workshop 2011, 2011. 17
10. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to (accessed 2014-05-25). 5, 20
11. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. ECRYPT Hash Workshop, 2007. 17
12. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The road from Panama to Keccak via RadioGatún. Dagstuhl Seminar Proceedings, 2009. 18
13. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Keyak v1, 2014. 18
14. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin / Heidelberg, 2011. 3
15. Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin / Heidelberg, 2007. 3
16. Johannes Buchmann, L. C. Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS - an improved Merkle signature scheme. In *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2006. 3
17. Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin / Heidelberg, 2008. 3, 6, 7, 13
18. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013. 1
19. Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in cryptography: The Even-Mansour scheme revisited. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2012. 17
20. Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. In *Journal of Cryptology*, pages 151–161. Springer-Verlag, 1991. 17

21. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, Cambridge, UK, 2004. 3
22. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988. 22
23. Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and AboulElla Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2013. http://huelsing.files.wordpress.com/2013/05/wotsspr.pdf. 3, 5, 13
24. Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, DimitrisE. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2013. 3
25. Joe Kilian and Phillip Rogaway. How to protect DES against exhaustive key search (an analysis of DESX). *J. Cryptology*, 14(1):17–35, 2001. 17
26. Kaoru Kurosawa. Power of a public random permutation and its application to authenticated-encryption. Cryptology ePrint Archive, Report 2002/127, 2002. http://eprint.iacr.org/. 17
27. Adam Langley. TLS symmetric crypto, 2014. https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html. 18
28. Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer Berlin / Heidelberg, 1990. 2
29. Josef Pieprzyk, Huaxiong Wang, and Chaoping Xing. Multiple-time signature schemes against adaptive chosen message attacks. In Mitsuru Matsui and Robert Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 88–100. Springer Berlin / Heidelberg, 2004. 4
30. Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Batten and Jennifer Seberry, editors, *Information Security and Privacy*, volume 2384 of *Lecture Notes in Computer Science*, pages 1–47. Springer Berlin / Heidelberg, 2002. 4, 7, 23
31. Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In Min Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 29–40. Springer Berlin / Heidelberg, 2006. 13

A Security Properties

In this section we give the basic definitions for security properties we use.

Existential Unforgeability under Adaptive Chosen Message Attacks The standard security notion for digital signature schemes is existential unforgeability under adaptive chosen message attacks (EU-CMA) [22] which is defined using the following experiment. By $\mathrm{Dss}(1^n)$ we denote a signature scheme with security parameter n.

Experiment $\mathsf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathrm{Dss}(1^n)}(\mathcal{A})$
$(\mathsf{sk}, \mathsf{pk}) \longleftarrow \mathsf{kg}(1^n)$
$(M^\star, \sigma^\star) \longleftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$
Let $\{(M_i, \sigma_i)\}_1^q$ be the query-answer pairs of $\mathsf{sign}(\mathsf{sk}, \cdot)$.
Return 1 iff $\mathsf{vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$ and $M^\star \notin \{M_i\}_1^q$.

A signature scheme is called existentially unforgeable under a q adaptive chosen message attack if any PPT adversary making at most q queries, has only negligible success probability in winning the above game.

An EU-CMA secure one-time signature scheme (OTS) is a signature scheme that is existentially unforgeable under a 1-adaptively chosen message attack.

A.1 Hash Function Families

In this section we provide definitions of the security properties of hash function families that we use, namely one-wayness, second-preimage resistance, undetectability and pseudorandomness. In the following let $n \in \mathbb{N}$ be the security parameter, $m, k = \text{poly}(n)$, $\mathcal{H}_n = \{\mathrm{H}_K : \{0,1\}^m \to \{0,1\}^n \,|\, K \in \{0,1\}^k\}$ a family of functions. (In the description of SPHINCS we actually omit the key K in many cases for readability.) In theory these functions are modeled as a random element chosen from some function family).

We define the security properties in terms of the success probability of an adversary $\mathcal{A}$ against the respective property. A function family $\mathcal{H}_n$ is said to provide a property if the success probability of any probabilistic polynomial-time adversary against this property is negligible. We begin with the success probability of an adversary $\mathcal{A}$ against the one-wayness (OW) of a function family $\mathcal{H}_n$.

$$\begin{aligned}\mathrm{Succ}^{\mathrm{OW}}_{\mathcal{H}_n}(\mathcal{A}) = \mathsf{Pr}[\, K \xleftarrow{\$} \{0,1\}^k; M \xleftarrow{\$} \{0,1\}^m, Y \longleftarrow \mathrm{H}_K(M),\\ M' \longleftarrow \mathcal{A}(K,Y) : Y = \mathrm{H}_K(M')] \ . \end{aligned} \tag{2}$$

We next define the success probability of an adversary $\mathcal{A}$ against second-preimage resistance (SPR).

$$\begin{aligned}\mathrm{Succ}^{\mathrm{SPR}}_{\mathcal{H}_n}(\mathcal{A}) =& \mathsf{Pr}[\, K \xleftarrow{\$} \{0,1\}^k; M \xleftarrow{\$} \{0,1\}^m, M' \longleftarrow \mathcal{A}(K,M) :\\ & (M \neq M') \wedge (\mathrm{H}_K(M) = \mathrm{H}_K(M'))] \ . \end{aligned} \tag{3}$$

To define undetectability, assume the following two distributions over $\{0,1\}^n \times \{0,1\}^k$. A sample (U,K) from the first distribution $\mathcal{D}_{\mathrm{UD},\mathcal{U}}$ is obtained by sampling $U \xleftarrow{\$} \{0,1\}^n$ and $K \xleftarrow{\$} \{0,1\}^k$ uniformly at random from the respective domain. A sample (U,K) from the second distribution $\mathcal{D}_{\mathrm{UD},\mathcal{H}}$ is obtained by sampling $K \xleftarrow{\$} \{0,1\}^k$ and then evaluating H_K on a uniformly random bit string, i.e. $U \longleftarrow \mathrm{H}_K(\mathcal{U}_m)$. The success probability of an adversary $\mathcal{A}$ against the undetectability of $\mathcal{H}_n$ is defined as:

$$\mathrm{Succ}^{\mathrm{UD}}_{\mathcal{H}_n}(\mathcal{A}) = \left|\mathsf{Pr}[\mathcal{A}^{\mathcal{D}_{\mathrm{UD},\mathcal{U}}} = 1] - \mathsf{Pr}[\mathcal{A}^{\mathcal{D}_{\mathrm{UD},\mathcal{H}}} = 1]\right| ,$$

where $\mathcal{A}^{\mathrm{dist}}$ denotes that $\mathcal{A}$ has oracle access to some oracle that outputs samples from distribution dist.

The fourth notion we use is pseudorandomness of a function family (PRF). In the definition of the success probability of an adversary against pseudorandomness the adversary gets black-box access to an oracle Box. Box is either initialized with a function from $\mathcal{H}_n$ or a function from the set $\mathcal{G}(m,n)$ of all functions with domain $\{0,1\}^m$ and range $\{0,1\}^n$. The goal of the adversary is to distinguish both cases:

$$\mathrm{Succ}^{\mathrm{PRF}}_{\mathcal{H}_n}(\mathcal{A}) = \left|\mathsf{Pr}[\mathsf{Box} \xleftarrow{\$} \mathcal{H}_n : \mathcal{A}^{\mathsf{Box}(\cdot)} = 1] - \mathsf{Pr}[\mathsf{Box} \xleftarrow{\$} \mathcal{G}(m,n) : \mathcal{A}^{\mathsf{Box}(\cdot)} = 1]\right| .$$

A.2 Subset-Resilient Functions

We now recall the definition of subset resilience from [30]. Let $\mathcal{H} = \{\mathrm{H}_{i,t,k}\}$ be a family of functions, where $\mathrm{H}_{i,t,k}$ maps a bit string of arbitrary length to a subset of size at most k of the set $\{0,1,...,t-1\}$. (As for hash functions in the description of SPHINCS we omit the key and assume the used function is randomly selected from a family using the uniform distribution.) Moreover, assume that there is a polynomial-time algorithm

that, given $i, 1^t, 1^k$ and M, computes $\mathrm{H}_{i,t,k}(M)$. Then $\mathcal{H}$ is called γ-subset resilient if the following success probability is negligible for any probabilistic polynomial-time adversary $\mathcal{A}$:

$$\mathrm{Succ}_{\mathcal{H}}^{\gamma\text{-sr}}(\mathcal{A}) = \mathsf{Pr}_i\left[(M_1, M_2, \ldots, M_{\gamma+1}) \longleftarrow \mathcal{A}(i, 1^t, 1^k) \right.$$
$$\left. \text{s.t. } \mathrm{H}_{i,t,k}(M_{\gamma+1}) \subseteq \bigcup_{j=1}^{\gamma} \mathrm{H}_{i,t,k}(M_j)\right]$$

B The virtual structure of a SPHINCS signature

To support easier understanding, Figure 3 shows the virtual structure of a SPHINCS signature, i.e. of one path inside the hyper-tree. It contains d trees $\textsc{Tree}_i$ $i \in [d-1]$ (each consisting of a binary hash tree that authenticates the root nodes of $2^{h/d}$ L-Trees which in turn each have the public key nodes of one WOTS$^+$ keypair as leaves). Each tree authenticates the tree below using a WOTS$^+$ signature $\sigma_{W,i}$. The only exception is $\textsc{Tree}_0$ which authenticates a HORST public key using a WOTS$^+$ signature. Finally, the HORST key pair is used to sign the message. Which trees inside the hyper-tree are used (which in turn determines the WOTS$^+$ key pairs used for the signature) and which HORST key pair is determined by the pseudorandomly generated index not shown here.

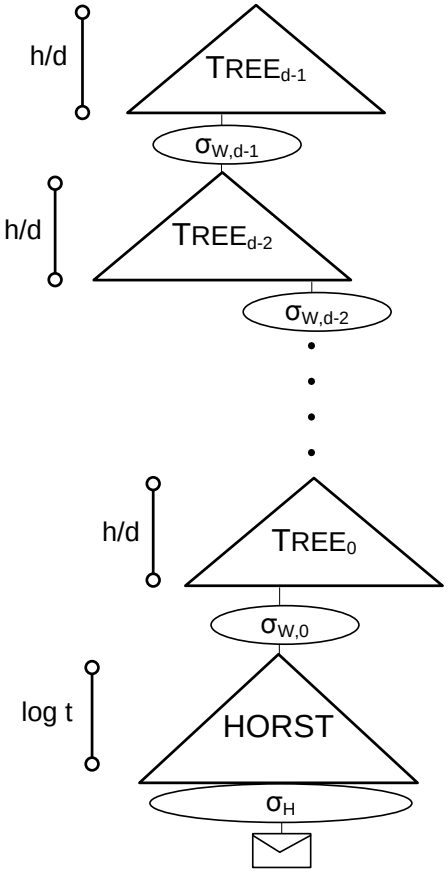

Fig. 3: Virtual structure of a SPHINCS signature