The SPHINCS$^+$ Signature Framework

Full version, September 23, 2019

Daniel J. Bernstein
University of Illinois at Chicago
Ruhr University Bochum
djb@cr.yp.to

Andreas Hülsing
Eindhoven University of Technology
andreas@huelsing.net

Stefan Kölbl
Cybercrypt
stek@mailbox.org

Ruben Niederhagen
Fraunhofer SIT, Darmstadt
ruben@polycephaly.org

Joost Rijneveld
Radboud University
joost@joostrijneveld.nl

Peter Schwabe
Radboud University
peter@cryptojedi.org

ABSTRACT

We introduce SPHINCS$^+$, a stateless hash-based signature framework. SPHINCS$^+$ has significant advantages over the state of the art in terms of speed, signature size, and security, and is among the nine remaining signature schemes in the second round of the NIST PQC standardization project. One of our main contributions in this context is a new few-time signature scheme that we call FORS. Our second main contribution is the introduction of *tweakable hash functions* and a demonstration how they allow for a unified security analysis of hash-based signature schemes. We give a security reduction for SPHINCS$^+$ using this abstraction and derive secure parameters in accordance with the resulting bound. Finally, we present speed results for our optimized implementation of SPHINCS$^+$ and compare to SPHINCS-256, Gravity-SPHINCS, and Picnic.

CCS CONCEPTS

• **Security and privacy → Digital signatures**.

KEYWORDS

Post-quantum cryptography, SPHINCS, hash-based signatures, stateless, tweakable hash functions, NIST PQC, exact security

1 INTRODUCTION

Hash-based signature schemes are among the oldest designs to construct digital signatures. First introduced by Lamport [35] and refined by Merkle [37] in 1979, forty years later the basic constructions remain largely the same. With well-understood security and minimal assumptions, they are often considered to be the most conservative option available. Yet, it took the potentially imminent construction of a quantum computer for them to gain popularity and be considered for real-world applications. Today hash-based signature schemes are the first post-quantum signature schemes formally defined in two RFCs [31, 36], and SPHINCS$^+$, the scheme presented in this work, is among the nine remaining signature proposals in the second round of the NIST post-quantum cryptography standardization project [1].

The performance of hash-based signatures, in terms of both speed and size, has traditionally been an obstacle for adoption. Developments over the past decade have taken significant steps towards practicality, in particular through the design of XMSS [16]. Arguably the biggest hurdle towards practicality is of a more fundamental order: almost all hash-based signature schemes in literature (including the schemes described in RFCs above) are *stateful*; they need to keep track of all produced signatures. This was addressed in practice by SPHINCS [9] in 2015, building upon theoretical work by Goldreich [26, 27]. Merkle's design crucially relies on iterating over signing keys *in order*, to prevent reuse. Contrarily, the structure in the designs following Goldreich is so large that, roughly, one can pick a signing key at random each time and reasonably assume it has not been used before. This is essential for many real-world uses, where continuously updating a stateful key pair is often impossible: consider, e.g., distributed servers and backups.

In this work we make three contributions to evolve the state of the art in the area of hash-based signature schemes:

(1) We introduce SPHINCS$^+$, a stateless hash-based signature framework which has significant advantages over SPHINCS in several dimensions and meets the requirements of the NIST PQC project [40].
(2) We introduce the concept of *tweakable hash functions* and show how it allows us to unify the security analysis of hash-based signature schemes.
(3) We present speed results for our optimized implementation of SPHINCS$^+$ and a comparison with the other relevant symmetric-cryptography-based signature schemes: SPHINCS-256 [9], Gravity-SPHINCS [6], and Picnic [17].

Introducing SPHINCS$^+$. Although in a practical range, signature size and speed of SPHINCS are far off from what we are used to from RSA or ECDSA signatures. This work presents SPHINCS$^+$, a stateless hash-based signature framework which improves upon SPHINCS in terms of speed and signature size. This is achieved by introducing several improvements that strengthen the security of the scheme and thereby allow for smaller parameters. We introduce a signature framework instead of a specific signature scheme. The main reason for this is the large flexibility offered by the many parameter options. This allows users to make highly application-specific trade-offs with regards to the signature size, the signing speed, the required number of signatures and the desired security level, and even account for platform considerations such as memory limits or hardware support for specific hash functions.

As SPHINCS$^+$ resembles SPHINCS in many details, we refrain from giving a detailed description of the full scheme in this paper but rather focus on the aspects that differ from previous work. A full formal specification of SPHINCS$^+$ is available in the official submission to NIST [4]. We now briefly recall the high-level construction

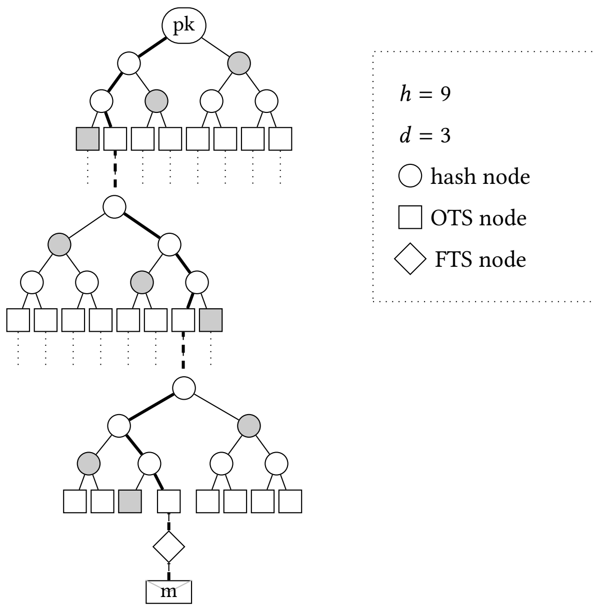


Figure 1: An illustration of a (small) SPHINCS structure.

of SPHINCS-like schemes to afterwards explain our improvements. See Section 3 for a more specific description.

Hash-based signature constructions center around a Merkle tree with one-time signature key pairs on its leaf nodes. For efficiency reasons, the XMSS^{MT} and SPHINCS constructions make use of a hypertree: a tree of trees, linked together using one-time signatures (OTS). As the leaf nodes of the SPHINCS tree are randomly selected, there is a trade-off to be made between the size of the tree and the likelihood of selecting the same leaf node twice. To sway this trade-off towards allowing smaller trees, SPHINCS uses a few-time signature scheme (FTS) at the bottom of the tree. The generic construction of such a hypertree is illustrated in Figure 1.

Among the main distinguishing contributions of SPHINCS$^+$ is the introduction of a new few-time signature scheme: FORS, introduced in Section 3.4. Another important change from SPHINCS to SPHINCS$^+$ is the way leaf nodes are chosen. SPHINCS$^+$ uses publicly verifiable index selection, described in Section 3.5. These two changes together make it harder to attack SPHINCS$^+$ via the few-time signature scheme and hence allow us to choose smaller parameters. With the same goal, we apply multi-target attack mitigation techniques as proposed in [33], making it harder to attack SPHINCS$^+$ using a (second-)preimage attack. We give a security reduction in Section 4 to formally show these claims. Analyzing the complexity of generic attacks against the required hash-function properties, we derive a formula for the bit security of a given parameter set from our security reduction (Section 5).

Tweakable hash functions. Over the last decade there was a line of work [15, 16, 19, 30, 33] focusing on reducing the assumptions that have to be made to prove a hash-based signature scheme secure. The first goal of this was to move away from collision resistance and towards collision resilient schemes. This leads to the use of targeted security notions like second-preimage and preimage resistance, making multi-target attacks a concern. Consequently, more recent proposals aimed at mitigating multi-target attacks [33].

Comparing these works, it turns out that the high-level constructions remain the same. What changes is the way nodes in hash chains and trees are computed. In some works, inputs first get XORed with random values, in others, functions are additionally keyed. Some proposals do both, and others just prepend or append additional data to the inputs before hashing. Although the differences in schemes are somewhat local, each work redid a full security analysis of the whole signature scheme. While these security analyses were already complex for stateful hash-based signature schemes, the case of stateless schemes adds further complexity.

We introduce an abstraction which we call *tweakable* hash functions in Section 2. Tweakable hash functions allow us to unify the description of hash-based signature schemes, abstracting away the details of how exactly nodes are computed. This allows us to separate the analysis of the high-level construction from the analysis of how this computation is done. We demonstrate the utility of this approach by proposing and analyzing three constructions of tweakable hash functions in Section 2, one of which is essentially the construction from [33]. Afterwards, the SPHINCS$^+$ security reduction in Section 4 bases security of large parts of SPHINCS$^+$ on the properties of the used tweakable hash functions and ignores how these are implemented (in addition security is based on properties of the initial message compression and the used PRFs). Hence, changing the way nodes are computed in SPHINCS$^+$ now only requires analyzing the hashing construction as a tweakable hash function. This also supports the design of dedicated constructions, as it provides a clear specification of the required properties.

Performance & comparison. Having defined a generic framework, we provide concrete parameters and instances (see Section 6) and evaluate the performance of the resulting signature scheme. Then we go for a comparison to similar signature schemes. The challenge here is that the schemes provide different levels of security under different assumptions. In a demonstration of the flexibility and competitiveness of our framework, we also define instances that carefully mimic the security level and properties of other signature schemes based on symmetric primitives and compare to these; see Section 7 for a discussion.

2 TWEAKABLE HASH FUNCTIONS

In this section we give a definition of tweakable hash functions, provide security notions, and discuss different instantiations. In Section 4 we then give a proof of security for the SPHINCS$^+$ framework using the properties of tweakable hash functions for the security of node computations.

2.1 Functional definition.

A *tweakable* hash function takes public parameters P and context information in form of a tweak T in addition to the message input. The public parameters might be thought of as a function key or index. The tweak might be interpreted as a nonce.

Definition 1 (Tweakable hash function). Let $n, \alpha \in \mathbb{N}$, $\mathcal{P}$ the public parameters space and $\mathcal{T}$ the tweak space. A *tweakable hash function*

is an efficient function

$$\mathbf{Th} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{\alpha} \rightarrow \{0,1\}^{n}, \quad \text{MD} \leftarrow \mathbf{Th}(P, T, M)$$

mapping an α-bit message M to an n-bit hash value MD using a function key called public parameter $P \in \mathcal{P}$ and a tweak $T \in \mathcal{T}$.

We sometimes write $\mathbf{Th}_{P,T}(M)$ in place of $\mathbf{Th}(P, T, M)$. We use the term public parameter for the function key to emphasize that it is intended to be public. Tweaks are used to define context and take the role of nonces when it comes to security. In SPHINCS$^+$ we use as public parameter a public seed **PK**.`seed` which is part of the SPHINCS$^+$ public key. As tweak we use a hash function address **ADRS** which identifies the position of the hash function call within the virtual structure defined by a SPHINCS$^+$ key pair. This allows us to make the hash-function calls for each SPHINCS$^+$ key pair and position in the virtual tree structure of SPHINCS$^+$ independent from each other.

2.2 Security notions.

Of course, this abstraction is only useful for us if it provides some security properties. What we require from tweakable hash functions are two properties, which we call *post-quantum single function, multi-target-collision resistance for distinct tweaks* (PQ-SM-TCR) and *post-quantum single function, multi-target decisional second-preimage resistance for distinct tweaks* (PQ-SM-DSPR).

PQ-SM-TCR. Essentially, SM-TCR is a variant of target-collision resistance. It is a two-stage game where an adversary $\mathcal{A}$ is allowed to adaptively specify p targets (multi-target) instead of a single one during the first stage. For this purpose $\mathcal{A}$ is given access to an oracle implementing the already keyed function (single-function as the same public parameters are used for all targets). The adversary's queries specify its targets for the second stage. In addition we require distinct tweaks, i.e., $\mathcal{A}$ is not allowed to use the same tweak for more than one query. Hence, $\mathcal{A}$ can only define one target per tweak. After specifying all targets, $\mathcal{A}$ receives the public parameters which are similar to a function key. The adversary wins if it finds a collision for one of the targets. It should be noted that as we are considering the post-quantum setting, we assume that adversaries have access to a quantum computer but honest parties do not. In consequence, all oracles in our definitions, except for random oracles, only allow classical access. A more detailed discussion of the post-quantum setting and quantum-accessible oracles can be found in Appendix A. We formalize the above in the following definition.

Definition 2 (PQ-SM-TCR). In the following let $\mathbf{Th}$ be a tweakable hash function as defined above. We define the success probability of any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the SM-TCR security of $\mathbf{Th}$. The definition is parameterized by the number of targets p for which it must hold that $p \leq |\mathcal{T}|$. In the definition, $\mathcal{A}_1$ is allowed to make p queries to an oracle $\mathbf{Th}(P, \cdot, \cdot)$. We denote the set of $\mathcal{A}_1$'s queries by $Q = \{(T_i, M_i)\}_{i=1}^{p}$ and define the predicate $\mathbf{DIST}(\{T_i\}_{i=1}^{p}) = (\forall i, k \in [1, p], i \neq k) : T_i \neq T_k$, i.e., $\mathbf{DIST}(\{T_i\}_{i=1}^{p})$ outputs 1 iff all tweaks are distinct.

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

We define the insecurity of a tweakable hash function against p target, time ξ, PQ-SM-TCR adversaries as the maximum success probability of any possibly quantum adversary $\mathcal{A}$ with p targets and running time $\leq \xi$:

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

As a special case, we refer to PQ-SM-TCR *with tweak advice* if $\mathcal{A}_1$ informs the oracle about all p tweaks it will use ahead of its queries.

SM-TCR is implied by collision resistance; a more detailed discussion of the relation between the two notions is given in Appendix B.

PQ-SM-DSPR. SM-TCR is a collision-finding notion. There are cases in the security reduction for SPHINCS$^+$ (and also XMSS-T [33]) where the adversary $\mathcal{A}$ works as a preimage finder. A reduction from a one-wayness notion leads to a non-tight reduction. The reason is that the reduction has to return preimages for some of the potential one-wayness targets as part of the answers to signing queries. If a preimage challenge was planted at a position for which a preimage is required to answer the signing query, the reduction fails. Consequently, the reduction has to guess where it may plant a preimage challenge and where it must not.

If we could instead ensure that (at least with high probability) the preimage returned by preimage finder $\mathcal{A}$ is different from the one we used to compute the image, we could turn $\mathcal{A}$ into a second-preimage finder that we might be able to use to break SM-TCR. The advantage of this approach is that the reduction now knows preimages for all targets and hence can answer all signing queries.

One way that was used before in [33] to ensure that the preimage finder $\mathcal{A}$ returns a second-preimage (and not the one already known to the reduction) is to assume that for every domain element of the function there exists at least one colliding domain element. As it is unknown to $\mathcal{A}$ which of the two or more preimages was used to compute the image its output must be independent of the used preimage. Hence, the returned preimage differs from the one already known to the reduction with probability at least $1/2$. The problem with this approach is that in the case of SPHINCS$^+$ and XMSS-T, the preimage finder works on a *length-preserving* hash function and a random length-preserving function does not have this property. Indeed, approximately $1/e$ of all domain elements do not have a colliding value in this case. Hence, we would expect cryptographic hash functions to also not have this property. It is possible to turn any length-preserving hash function into a hash function with this property [10], but this comes at the cost of a slight loss in security and a factor-two slowdown.

An alternative approach was recently proposed in [10] under the name decisional second-preimage resistance (DSPR). The intuition here is that while there might exist domain elements that do not have a colliding value, it is computationally hard to detect those. It was shown in [10] that for functions which are DSPR, a preimage finder can be used to find second-preimages with approximately

the same success probability. In the following, we formally define a version of DSPR adopted to the setting of tweakable hash functions which we call post-quantum single function, multi-target decisional second preimage resistance for distinct tweaks (PQ-SM-DSPR).

The definition of DSPR requires a definition of a second-preimage-exists predicate. We derive a workable definition for tweakable hash functions from the definition for keyed hash functions from [10] and use this definition to further define what it means for a tweakable hash function to be PQ-SM-DSPR.

Definition 3 (SPexists for tweakable hash functions). The second-preimage-exists predicate SPexists($\mathbf{Th}$) for a tweakable hash function $\mathbf{Th}$ is the function $\mathbf{SP} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha \rightarrow \{0,1\}$ defined as follows:

$$\mathbf{SP}_{P,T}(M) = \begin{cases} 1 & \text{if } |\mathbf{Th}^{-1}_{P,T}(\mathbf{Th}_{P,T}(M))| \geq 2 \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathbf{Th}^{-1}_{P,T}$ refers to the inverse of the function obtained by fixing the first two inputs to $\mathbf{Th}$ to the given values.

Definition 4 (PQ-SM-DSPR). In the following let $\mathbf{Th}$ be a tweakable hash function as defined above. We define the advantage of any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the SM-DSPR-security of $\mathbf{Th}$. The definition is parameterized by the number of targets p for which it must hold that $p \leq |\mathcal{T}|$. In the definition, $\mathcal{A}_1$ is allowed to make p queries to an oracle $\mathbf{Th}(P, \cdot, \cdot)$. The query set Q and predicate $\mathbf{DIST}(\{T_i\}_{i=1}^p)$, are defined as in Definition 2.

$$\mathrm{Adv}^{\text{SM-DSPR}}_{\mathbf{Th},p}(\mathcal{A}) = \max\{0, \mathrm{succ} - \mathrm{triv}\}$$

with

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

We define the PQ-SM-DSPR insecurity of a tweakable hash function against p target, time ξ adversaries as the maximum advantage of any (possibly quantum) adversary $\mathcal{A}$ with p targets and running time $\leq \xi$:

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

As a special case, we refer to PQ-SM-DSPR *with tweak advice* if $\mathcal{A}_1$ informs the oracle about all p tweaks it will use ahead of its queries.

The above definition of the DSPR advantage might look unfamiliar to the reader. The idea is the common concept that the advantage should be defined as the advantage of an adversary over the trivial algorithm that just guesses the right answer. Usually, the right answer is a uniformly random bit and hence we simply subtract $1/2$ as the guessing probability. For the case of DSPR, the guessing probability depends on the actual function used. E.g., for a random length-preserving function[1] $\mathbf{Th}$, the probability that $\mathbf{SP}_{P,T}(M) = 1$ is about $1 - 1/e$. This turns out to significantly complicate the definition of an advantage. To obtain a usable definition, the authors of [10] made some choices. Most importantly, the trivial attack to compare to was decided to be the algorithm that always outputs 1. This was justified by showing that for the overwhelming majority of functions $\Pr[\mathbf{SP}_{P,T}(M) = 1] > 1/2$ and for the cases where $\Pr[\mathbf{SP}_{P,T}(M) = 1] < 1/2$ DSPR turns out to not be useful. For a much more detailed discussion of the choices, see [10].

[1] We consider a tweakable hash function length-preserving if the message length equals the output length.

2.3 Generic constructions

In this section we give three generic constructions of tweakable hash functions. Our constructions make use of keyed hash functions $\mathrm{H} : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$. For key K and message M we sometimes write $\mathrm{H}_K(M)$ in place of $\mathrm{H}(K, M)$. The first construction is in the standard model but requires public parameters with size linear in the size of the tweak space. For SPHINCS$^+$ this would lead to exponential-size public parameters. This construction is thus mainly meant as an example to motivate the second construction, which is essentially the same as the first with the difference that it replaces the public parameters by a short public seed from which everyone can generate the required parameters using a keyed hash function H_2. While this massively reduces the public parameter size it comes at the cost of requiring the quantum accessible random oracle model (QROM) for the proof. If we assumed that H_2 was a PRF and if we just initialized the public parameters using H_2 and never output the used seed, we would still achieve security in the standard model. However, as we are handing out the seed, nothing can be derived from the PRF security of H_2 which requires the seed to be kept secret. Hence, we could either formulate a new, interactive security assumption or we use the QROM to show that this public-parameter compression is secure. We did the latter. The third construction goes even one step further and assumes that all hash functions used behave like quantum accessible random oracles (QROs).

CONSTRUCTION 5. *Given a keyed hash function H with n-bit keys, we construct* $\mathbf{Th}$ *as*

$$\begin{aligned} \mathbf{Th}(P, T, M) &= H(P[(\alpha + n)T, n], M^{\oplus}), \\ M^{\oplus} &= M \oplus (P[(\alpha + n)T + n, \alpha]), \end{aligned}$$

where P is a length-$(\alpha + n)|\mathcal{T}|$ bit string and $P[i, j]$ denotes the j-bit substring of P that starts with the ith bit.

CONSTRUCTION 6. *Given two hash functions $H_1 : \{0,1\}^{2n} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ with $2n$-bit keys, and $H_2 : \{0,1\}^{2n} \rightarrow \{0,1\}^\alpha$ we construct* $\mathbf{Th}$ *with $\mathcal{P} = \mathcal{T} = \{0,1\}^n$, as*

$$\mathbf{Th}(P, T, M) = H_1(P||T, M^{\oplus}), \text{ with } M^{\oplus} = M \oplus H_2(P||T).$$

CONSTRUCTION 7. *Given a hash function $H : \{0,1\}^{2n+\alpha} \rightarrow \{0,1\}^n$, we construct* $\mathbf{Th}$ *with $\mathcal{P} = \mathcal{T} = \{0,1\}^n$ as*

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

Construction 6 is essentially the construction used in [33] which was proven secure in the QROM using the post-quantum multi-function, multi-target second-preimage resistance (PQ-MM-SPR) of H. Construction 6 differs from [33] in that it does not key H with a (pseudo-)random bit string but just with $(P||T)$ which ensures distinct keys for distinct tweaks. Construction 7 is in spirit similar to the construction used for LMS signatures [36].

SM-TCR security. We first show under what conditions these constructions are PQ-SM-TCR. Afterwards, we look at PQ-SM-DSPR. We show that Construction 5 is PQ-SM-TCR if H is post-quantum multi-function, multi-target second-preimage resistant (PQ-MM-SPR), that Construction 6 is PQ-SM-TCR with tweak advice if H_1 is post-quantum distinct-function, multi-target second-preimage resistant (PQ-DM-SPR) and H_2 is modeled as QRO, and that Construction 7 is PQ-SM-TCR if H is modeled as QRO. We only achieve PQ-SM-TCR with tweak advice for Construction 6 for technical reasons. However, for the use in SPHINCS$^+$ and XMSS-T PQ-SM-TCR with tweak advice is sufficient. PQ-DM-SPR differs from PQ-MM-SPR in that it does not require the use of random but just distinct function keys:

Definition 8 (PQ-DM-SPR). Let $H : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ be a keyed hash function. We define the advantage of any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against distinct-function, multi-target second-preimage resistance (DM-SPR). This definition is parameterized by the number of targets p.

$$\begin{aligned}\mathrm{Succ}^{\text{DM-SPR}}_{\mathrm{H},p}(\mathcal{A}) = \Pr\Big[&\{K_i\}_{i=1}^p \leftarrow \mathcal{A}_1(\,), \{M_i\}_1^p \leftarrow_R (\{0,1\}^\alpha)^p;\\ &(j, M') \leftarrow_R \mathcal{A}_2(\{(K_i, M_i)\}_{i=1}^p) : M' \neq M_j\\ &\wedge \mathrm{H}(K_j, M_j) = \mathrm{H}(K_j, M') \wedge \mathbf{DIST}(\{K_i\}_{i=1}^p)\Big].\end{aligned}$$

where we assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ share state and $\mathbf{DIST}(\{K_i\}_1^p)$ is as in Definition 2.

We define the insecurity of a keyed hash function H against p target, time-ξ, PQ-DM-SPR adversaries as the maximum success probability of any possibly quantum adversary $\mathcal{A}$ with p targets and running time $\leq \xi$:

$$\mathrm{InSec}^{\text{PQ-DM-SPR}}(\mathrm{H}; \xi, p) = \max_{\mathcal{A}} \left\{\mathrm{Succ}^{\text{DM-SPR}}_{\mathrm{H},p}(\mathcal{A})\right\}.$$

The definition of MM-SPR as given in [33] is obtained from the above by replacing $\{K_i\}_1^p \leftarrow \mathcal{A}_1(\,)$ by $\{K_i\}_1^p \leftarrow_R \mathcal{K}^p$ and ignoring the **DIST** condition (and of course renaming $\mathcal{A}_2$ as $\mathcal{A}$).

THEOREM 9. *Let H be a hash function as in Construction 5 and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 5. Then the success probability of any time-ξ (quantum) adversary $\mathcal{A}$ against SM-TCR of* $\mathbf{Th}$ *is bounded by*

$$\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}) \leq \mathrm{InSec}^{\text{PQ-MM-SPR}}(H; \xi, p).$$

PROOF. Assume we are given access to an adversary $\mathcal{A}$ against SM-TCR of $\mathbf{Th}$. We show how to construct an oracle machine $\mathcal{M}^{\mathcal{A}}$ that breaks MM-SPR of H. Essentially, $\mathcal{M}^{\mathcal{A}}$ uses the MM-SPR challenge set $\{(K_i^*, M_i^*)\}_{i=1}^p$ to generate the public parameters P. For the ith query (M_i, T_i) made by $\mathcal{A}_1$, $\mathcal{M}^{\mathcal{A}}$ sets $P[(\alpha+n)T_i, n] = K_i^*$ and $P[(\alpha+n)T_i + n, \alpha] = M_i \oplus M_i^*$ and answers the query. After all p queries, $\mathcal{M}^{\mathcal{A}}$ fills the remaining spots in P with random bits and starts $\mathcal{A}_2$. When $\mathcal{A}_2$ outputs a target collision (j, M), $\mathcal{M}^{\mathcal{A}}$ outputs $(j, M \oplus P[(\alpha+n)T_j + n, \alpha])$ which by construction is a second preimage for M_j^* under $\mathrm{H}_{K_j^*}$. Hence, the success probability of $\mathcal{M}^{\mathcal{A}}$ is exactly that of $\mathcal{A}$ and it runs in essentially the same time. □

THEOREM 10. *Let H_1 and H_2 be hash functions as in Construction 6 and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 6. Then the success probability of any time-ξ (quantum) adversary $\mathcal{A}$ against SM-TCR of* $\mathbf{Th}$ *with tweak advice is bounded by*

$$\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}) \leq \mathrm{InSec}^{\text{PQ-DM-SPR}}(H_1; \xi, p),$$

when modeling H_2 as quantum-accessible random oracle and not giving $\mathcal{A}_1$ access to this oracle.

Note that the restriction that $\mathcal{A}_1$ does not get access to the random oracle is sufficient in later proofs, because when $\mathcal{A}_1$ is implemented by a reduction, it will only use the function oracle to generate the challenges.

PROOF. Assume we are given access to an adversary $\mathcal{A}$ against SM-TCR (with tweak advice) of $\mathbf{Th}$. We show how to construct an oracle machine $\mathcal{M}^{\mathcal{A}}$ that breaks DM-SPR of H_1. The idea is essentially the same as above. The main difference is that now instead of setting elements in P, we program the random oracle H_2. The reduction $\mathcal{M}^{\mathcal{A}}$ first receives the tweak advice which allows it to generate $\{K_i^*\}_1^p$ by first sampling a random $P \leftarrow_R \mathcal{P}$ and setting $K_i^* = P \| T_i$. With this, $\mathcal{M}^{\mathcal{A}}$ can request the DM-SPR challenge messages $M_1^*, \ldots, M_p^*$.

For the ith query (M_i, T_i) by $\mathcal{A}_1$, $\mathcal{M}^{\mathcal{A}}$ programs $\mathrm{H}_2(P||T_i) = M_i \oplus M_i^*$ and records the query. Then it applies the construction to answer the query. After all p queries were made, $\mathcal{M}^{\mathcal{A}}$ runs $\mathcal{A}_2$. When $\mathcal{A}_2$ outputs a target collision (j, M), $\mathcal{M}^{\mathcal{A}}$ outputs $(j, M \oplus \mathrm{H}_2(P||T_j))$ which by construction is a second preimage for M_j^* under $\mathrm{H}_1(K_j^*, \cdot)$. Hence, the success probability of $\mathcal{M}^{\mathcal{A}}$ is exactly that of $\mathcal{A}$ and it runs in essentially the same time. As all random oracle programming is done before $\mathcal{A}$ gets access to H_2, the reduction is history-free and thereby also works in the QROM. □

THEOREM 11. *Let H be a hash function as in Construction 7, modeled as quantum-accessible random oracle, and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 7. Then the success probability of any (quantum) adversary $\mathcal{A}$ making at most q-queries to H, against SM-TCR of* $\mathbf{Th}$ *is bounded by*

$$\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}) \leq 8(2q+1)^2/2^n,$$

when $\mathcal{A}_1$ is not given access to the random oracle.

The reason for not giving $\mathcal{A}_1$ access to the random oracle is the same as in Theorem 10. We delay the proof of Theorem 11 to Appendix D. The reason is that it is a direct proof of a quantum query complexity lower bound, which uses a framework from [33] that we only introduce in Appendix C.

SM-DSPR security. Now we take a look at SM-DSPR. We will reduce distinct function, multi-target decisional second-preimage resistance (DM-DSPR) of the used hash function to SM-DSPR of the tweakable hash. DM-DSPR needs the following definition from [10].

Definition 12 (SPexists for keyed hash functions). The second-preimage-exists predicate SPexists(H) for a keyed hash function H is the function $\mathbf{SP} : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}$ defined as follows:

$$\mathbf{SP}_K(M) = \begin{cases} 1 & \text{if } |\mathrm{H}_K^{-1}(\mathrm{H}_K(M))| \geq 2 \\ 0 & \text{otherwise,} \end{cases}$$

Definition 13 (PQ-DM-DSPR). In the following let H be a keyed hash function as defined above. We define the advantage of any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against DM-DSPR of H. The definition is parameterized by the number of targets p.

$$\mathrm{Adv}^{\text{DM-DSPR}}_{\mathrm{H},p}(\mathcal{A}) \stackrel{\text{def}}{=} \max\{0, \mathrm{succ} - \mathrm{triv}\},$$

where

$$\begin{aligned}\mathrm{succ} =& \Pr\big[\{K_i\}_1^p \leftarrow \mathcal{A}_1(\,); \{M_i\}_1^p \leftarrow_R (\{0,1\}^\alpha)^p;\\ &(j,b) \leftarrow \mathcal{A}_2(\{(K_i,M_i)\}_1^p) : \mathbf{SP}_{K_j}(M_j) = b \wedge \mathbf{DIST}(\{K_i\}_1^p)\big];\\ \mathrm{triv} =& \Pr\big[\{K_i\}_1^p \leftarrow \mathcal{A}_1(\,); \{M_i\}_1^p \leftarrow_R (\{0,1\}^\alpha)^p;\\ &(j,b) \leftarrow \mathcal{A}_2(\{(K_i,M_i)\}_1^p) : \mathbf{SP}_{K_j}(M_j) = 1 \wedge \mathbf{DIST}(\{K_i\}_1^p)\big];\end{aligned}$$

and where $\mathbf{DIST}(\{K_i\}_1^p)$ is defined as in Definition 2.

We define the PQ-DM-DSPR insecurity of a keyed hash function against p-target, time-ξ adversaries as the maximum advantage of any (possibly quantum) adversary $\mathcal{A}$ with p targets and running time $\leq \xi$:

$$\mathrm{InSec}^{\text{PQ-DM-DSPR}}(\mathrm{H}; \xi, p) = \max_{\mathcal{A}} \mathrm{Adv}^{\text{DM-DSPR}}_{\mathrm{H},p}(\mathcal{A}).$$

THEOREM 14. *Let H be a hash function as in Construction 5 and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 5. Then the advantage of any time-ξ (quantum) adversary $\mathcal{A}$ against SM-DSPR of* $\mathbf{Th}$ *is bounded by*

$$\mathrm{Adv}^{SM\text{-}DSPR}_{\mathbf{Th},p}(\mathcal{A}) \leq \mathrm{InSec}^{PQ\text{-}DM\text{-}SPR}(H; \xi, p)$$

PROOF. The reduction $\mathcal{M}^{\mathcal{A}}$ works exactly the same as in the proof of Theorem 9 with the single difference that here $\mathcal{M}^{\mathcal{A}}$ just forwards $\mathcal{A}$'s output. The extraction procedure in the proof of Theorem 9 shows that a collision under the function simulated towards $\mathcal{A}$ implies the existence of a collision under H. Hence, $\mathcal{M}^{\mathcal{A}}$ is correct with the same probability as $\mathcal{A}$. There also is no difference between triv for the two cases (which would imply a difference in advantage) as $\mathbf{SP}_{P,T_j}(M_j) = \mathbf{SP}_{K_j}(M_j^{\oplus})$. □

In the case of Construction 6 we again only achieve SM-DSPR with tweak advice, but again this is sufficient for the use in the context of hash-based signatures.

THEOREM 15. *Let H_1 and H_2 be hash functions as in Construction 6 and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 6. Then the advantage of any time-ξ (quantum) adversary $\mathcal{A}$ against SM-DSPR of* $\mathbf{Th}$ *with tweak advice is bounded by*

$$\mathrm{Adv}^{SM\text{-}DSPR}_{\mathbf{Th},p}(\mathcal{A}) \leq \mathrm{InSec}^{PQ\text{-}DM\text{-}DSPR}(H; \xi, p),$$

when modeling H_2 as quantum-accessible random oracle and not giving $\mathcal{A}_1$ access to this oracle.

PROOF. Again, the reduction $\mathcal{M}^{\mathcal{A}}$ works exactly the same as in the corresponding SM-TCR case, discussed in the proof of Theorem 10. Again, the single difference is that $\mathcal{M}^{\mathcal{A}}$ just forwards $\mathcal{A}$'s output in this case. The argument that $\mathcal{M}^{\mathcal{A}}$ is correct whenever $\mathcal{A}$ is correct and that the values of triv do not differ carries over from the proof of Theorem 14. □

For Construction 7 it is an open research question to prove SM-DSPR security. We conjecture the following bound.

CONJECTURE 16. *Let H be a hash function as in Construction 7, modeled as quantum-accessible random oracle and* $\mathbf{Th}$ *the tweakable hash function constructed by Construction 7. Then the advantage of any (quantum) adversary $\mathcal{A}$ making at most q-queries to H, against SM-DSPR of* $\mathbf{Th}$ *is bounded by*

$$\mathrm{Adv}^{SM\text{-}DSPR}_{\mathbf{Th},p}(\mathcal{A}) \leq O(q^2/2^n),$$

when $\mathcal{A}_1$ is not given access to the random oracle.

The reasoning behind this bound is similar to the reasoning behind the hardness of DM-DSPR given in Table 5. The difference here is that the adversary is allowed to choose messages and tweaks while the public parameters are hidden instead of choosing the function keys and getting the messages afterwards. However, given that we are considering a random oracle, the adversary does not gain any advantage from being able to partially determine the challenges in either way. This is the case as the behavior of the functions is hidden from it until after the challenge generation.

3 THE SPHINCS$^+$ FRAMEWORK

We now describe the SPHINCS$^+$ framework. We roughly follow the description in the SPHINCS$^+$ submission to NIST [4], often citing it literally in sections where precise definitions are required.

3.1 Cryptographic (Hash) Function Families

SPHINCS$^+$ makes use of several different function families with cryptographic properties. This description will use them generically, and we defer giving specific instantiations to Section 6.

SPHINCS$^+$ applies the multi-target mitigation technique from [33] using the abstraction of tweakable hash functions from above. In addition to several tweakable hash functions, SPHINCS$^+$ makes use of two PRFs and a keyed hash function. Input and output length are given in terms of the security parameter n and the message-digest length m, both to be defined more precisely below.

Tweakable hash functions. The constructions described in this work are built on top of a collection of tweakable hash functions with one function per input length. For SPHINCS$^+$ we fix $\mathcal{P} = \{0,1\}^n$ and $\mathcal{T} = \{0,1\}^{256}$, limit the message length to multiples of n, and use the same public parameter for the whole collection of tweakable hash functions. We write $\mathbf{Th}_\ell : \{0,1\}^n \times \{0,1\}^{256} \times \{0,1\}^{\ell n} \to \{0,1\}^n$, for the function with input length ℓn.

There are two special cases which we rename for consistency with previous descriptions of hash-based signature schemes: $\mathbf{F} : \{0,1\}^n \times \{0,1\}^{256} \times \{0,1\}^n \to \{0,1\}^n$, $\mathbf{F} \stackrel{\text{def}}{=} \mathbf{Th}_1$; $\mathbf{H} : \{0,1\}^n \times \{0,1\}^{256} \times \{0,1\}^{2n} \to \{0,1\}^n$, $\mathbf{H} \stackrel{\text{def}}{=} \mathbf{Th}_2$.

Pseudorandom functions and the message digest. SPHINCS$^+$ makes use of a pseudorandom function $\mathbf{PRF}$ for pseudorandom key generation, $\mathbf{PRF} : \{0,1\}^n \times \{0,1\}^{256} \to \{0,1\}^n$, and a pseudorandom function $\mathbf{PRF_{msg}}$ to generate randomness for the message compression: $\mathbf{PRF_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$. To compress the message to be signed, we use an additional keyed hash function $\mathbf{H_{msg}}$ that can process arbitrary length messages: $\mathbf{H_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^m$.

3.2 WOTS+

WOTS+ [30] is a one-time signature scheme: a private key must be used to sign exactly one message. When it is reused to sign multiple messages, security quickly degrades [14].

Parameters. WOTS+ has two parameters n and w. n is the security parameter; it is the message length as well as the length of a private key element, public key element, and signature element in bits. w is the Winternitz parameter; it can be used to make a trade-off between signing time and signature size: a higher value implies a smaller, slower, signature. w is typically restricted to 4, 16 or 256.

Define $\texttt{len}_1 = \lceil n/\log(w) \rceil$ and $\texttt{len}_2 = \lfloor \log(\texttt{len}_1(w-1))/\log(w) \rfloor + 1$. The sum of these, $\texttt{len}$, represents the number of n-bit values in an uncompressed WOTS+ private key, public key, and signature.

The WOTS+ key pair. In the context of SPHINCS+, the WOTS+ private key is derived from a secret seed **SK**.seed that is part of the SPHINCS+ private key, and the address of the WOTS+ key pair within the hypertree, using **PRF**.

The corresponding public key is derived by applying **F** iteratively for w repetitions to each of the n-bit values in the private key, effectively constructing len hash chains. Here, **F** is parameterized by the address of the WOTS+ key pair, as well as the height of the **F** invocation and its specific chain, in addition to a seed **PK**.seed that is part of the SPHINCS+ public key.

In contrast to previous definitions of WOTS+, and as a direct consequence of the use of tweakable hash functions to mitigate multi-target attacks, we do not use so-called ℓ-trees to compress the WOTS+ public key. Instead, the public key is compressed to an n-bit value using a single tweakable hash function call to $\mathbf{Th}_{\texttt{len}}$. We use 'WOTS+ public key' to refer to the compressed public key.

WOTS+ signature and verification. An input message m is interpreted as $\texttt{len}_1$ integers m_i, between 0 and $w-1$. We compute a checksum $C = \Sigma_{i=1}^{\texttt{len}_1}(w-1-m_i)$ over these values, represented as string of $\texttt{len}_2$ base-w values $C = (C_1, \ldots, C_{\texttt{len}_2})$. This checksum is necessary to prevent message forgery: an increase in at least one m_i leads to a decrease in at least one C_i and vice-versa.

Using these len integers as chain lengths, the chaining function **F** is applied to the private key elements. This leads to len n-bit values that make up the signature. The verifier can then recompute the checksum, derive the chain lengths, and apply **F** to complete each chain to its full length. This leads to the chain heads that are hashed using $\mathbf{Th}_{\texttt{len}}$ to compute the n-bit public key.

3.3 The hypertree

We first describe a single-tree hash-based signature that is essentially equivalent to the XMSS construction [16]. We then extend this to a multi-tree setting, in the same style as XMSS^{MT} [32].

A single tree. To be able to sign $2^{h'}$ messages, the signer derives $2^{h'}$ WOTS+ public keys. We use these keys as leaf nodes. To construct a binary tree, one repeatedly applies **H** on pairs of nodes, parameterized with the unique address of this application of **H** as well as the public seed **PK**.seed. We consider such a tree to be of height h', corresponding to the number of **H** applications to move from the leaves to the root. The root of this tree is what will now briefly serve as the public key of the single tree scheme.

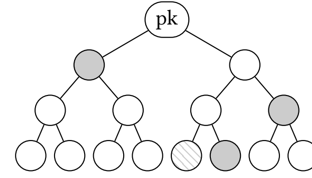


Figure 2: The authentication path to authenticate the fifth leaf is shown in gray.

One of the WOTS+ leaf nodes is used to create a signature on an n-bit message. Simply publishing the WOTS+ signature is not sufficient, as the verifier also requires information about the rest of the tree. For this, the signer includes the so-called 'authentication path' (see Figure 2). The verifier first derives the WOTS+ public key from the signature, and then uses the nodes included in the authentication path to reconstruct the root node.

A tree of trees. To make it sufficiently unlikely that random selection of a leaf node repeatedly results in the same leaf node being selected, a SPHINCS tree needs to be considerably large.

Rather than increasing h' (and incurring the insurmountable cost of computing $2^{h'}$ WOTS+ public keys per signing operation), we create a hypertree. For a more detailed discussion on this construction, refer to the paper introducing SPHINCS [9, Section 1].

This construction serves as a certification tree. The WOTS+ leaf nodes of the trees on the bottom layer are used to sign messages (or, in our case, FTS public keys), while the leaf nodes of trees on all other layers are used to sign the root nodes of the trees below. The WOTS+ signatures and authentication paths from a leaf at the bottom of the hypertree to the root of the top-most tree constitutes an authentication path. See Figure 1 on page 2 for an illustration.

Crucially, all leaf nodes of all intermediate trees are deterministically generated WOTS+ public keys that do not depend on any of the trees below it. This means that the complete hypertree is purely virtual: it never needs to be computed in full. During key generation, only the top-most subtree is computed to derive the public key. We define the total tree to be of height h and the number of intermediate layers to be d, retroactively setting h' to be h/d.

3.4 FORS

As the few-time signature scheme in SPHINCS+, we define FORS: Forest of Random Subsets, an improvement of HORST [9]. FORS security is captured in Section 4, where we introduce a new security notion for hash functions for this very reason. The new security notion strengthens the notion of target subset resilience as previously used to analyze HORS and HORST. FORS is defined in terms of integers k and $t = 2^a$, and can be used to sign strings of ka bits.

The FORS key pair. The FORS private key consists of kt random n-bit values, grouped together into k sets of t values each. In the context of SPHINCS+, these values are deterministically derived from **SK**.seed using **PRF** and the address of the key in the hypertree.

To construct the FORS public key, we first construct k binary hash trees on top of the sets of private key elements. Each of the t values is used as a leaf node, resulting in k trees of height a. We use **H**, addressed using the location of the FORS key pair in the

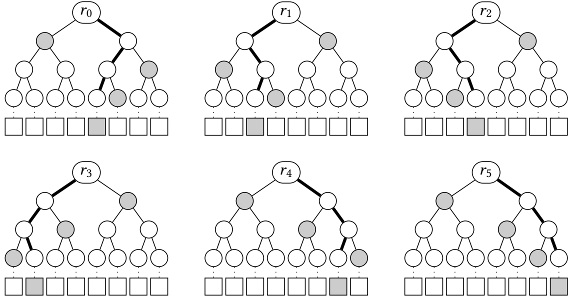


Figure 3: An illustration of a FORS signature with k = 6 and a = 3, for the message `100 010 011 001 110 111`.

hypertree and the unique position of the hash function call within the FORS trees. As in WOTS$^+$, we compress the root nodes using a call to $\mathbf{Th}_k$. The resulting n-bit value is the FORS public key.

FORS signatures. Given a message of ka bits, we extract k strings of a bits. Each of these bit strings is interpreted as the index of a single leaf node in each of the k FORS trees. The signature consists of these nodes and their respective authentication paths (see Figure 3).

The verifier reconstructs each of the root nodes using the authentication paths and uses $\mathbf{Th}_k$ to reconstruct the public key. As part of SPHINCS$^+$, a FORS signature is never verified explicitly. Rather, the resulting public key is used as a message, to be implicitly checked together with a WOTS$^+$ signature.

3.5 SPHINCS$^+$

Given the components defined above, we now construct SPHINCS$^+$.

The SPHINCS$^+$ key pair. Almost all elements that make up an SPHINCS$^+$ key pair have been introduced implicitly, above. The public key consists of two n-bit values: the root node of the top tree in the hypertree, and a random public seed $\mathbf{PK}.\mathtt{seed}$. In addition, the private key consists of two more n-bit random seeds: $\mathbf{SK}.\mathtt{seed}$, to generate the WOTS$^+$ and FORS secret keys, and $\mathbf{SK}.\mathtt{prf}$, used below for the randomized message digest.

The SPHINCS$^+$ signature. It should come as no surprise that 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. Two points have not yet been addressed: the computation of the message digest, and leaf selection. Here, SPHINCS$^+$ differs from the original SPHINCS in subtle but important details.

First, we pseudorandomly generate a randomizer $\mathbf{R}$, based on the message and $\mathbf{SK}.\mathtt{prf}$. $\mathbf{R}$ can optionally be made non-deterministic by adding additional randomness $\mathtt{OptRand}$. This may counteract side-channel attacks that rely on collecting several traces for the same computation. Note that setting this value to the all-zero string (or using a low-entropy value) does not negatively affect the pseudorandomness of $\mathbf{R}$. Formally, we say that $\mathbf{R} = \mathbf{PRF}(\mathbf{SK}.\mathtt{prf}, \mathtt{OptRand}, M)$. $\mathbf{R}$ is part of the signature. Using $\mathbf{R}$, we then derive the index of the leaf node that is to be used, as well as the message digest as $(\mathrm{MD}||\mathrm{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\mathtt{seed}, \mathbf{PK}.\mathtt{root}, M)$.

In contrast to SPHINCS, this method of selecting the index is publicly verifiable, preventing an attacker from freely selecting a seemingly random index and combining it with a message of their choice. Crucially, this counteracts multi-target attacks on the few-time signature scheme. As the index can now be computed by the verifier, it is no longer included in the signature.

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

	keypair	sign	verify
F	$2^{h/d}w \cdot \mathtt{len}$	$kt + d(2^{h/d})w \cdot \mathtt{len}$	$k + dw \cdot \mathtt{len}$
H	$2^{h/d} - 1$	$k(t-1) + d(2^{h/d} - 1)$	$k \log t + h$
PRF	$2^{h/d} \cdot \mathtt{len}$	$kt + d(2^{h/d}) \cdot \mathtt{len} + 1$	–
$\mathbf{Th}_{\mathtt{len}}$	$2^{h/d}$	$d2^{h/d}$	d

3.6 Theoretical Performance Estimates

In the following section we provide formulas to estimate computational cost and data sizes for a given SPHINCS$^+$ parameter set.

Key Generation. Generating the key seeds for SPHINCS$^+$ requires three calls to a random number generator. For the leaves of the top tree we need to perform $2^{h/d}$ WOTS$^+$ key generations ($\mathtt{len}$ calls to **PRF** to generate the sk and $w \cdot \mathtt{len}$ calls to **F** for the pk), and we have to compress the WOTS$^+$ public key (one call to $T_{\mathtt{len}}$). Computing the root of the top tree requires $(2^{h/d} - 1)$ calls to **H**.

Signing. For randomization and message compression we need one call to **PRF**, $\mathbf{PRF_{msg}}$ and $\mathbf{H_{msg}}$. The FORS signature requires kt calls to **PRF** and **F**. To compute the root of the k binary trees of height $\log t$, we add $k(t-1)$ calls to **H** and one call to $\mathbf{Th}_k$ to combine them. For the authentication paths, we compute d trees similarly to key generation. This implies $d(2^{h/d})$ times $\mathtt{len}$ calls to **PRF** and $w \cdot \mathtt{len}$ calls to **F** for the leaves, $d(2^{h/d})$ calls to $\mathbf{Th}_{\mathtt{len}}$ for key compressions, and $d(2^{h/d} - 1)$ calls to **H** for the nodes in the trees.

Verification. We first compute the message hash using $\mathbf{H_{msg}}$. We need to perform one FORS verification, which requires k calls to **F**, $k \log t$ calls to **H** and one call to $\mathbf{Th}_k$ to hash the roots. Next, we verify d layers in the hypertree, which takes $< w \cdot \mathtt{len}$ calls to **F** and one call to $\mathbf{Th}_{\mathtt{len}}$ each per WOTS$^+$ signature verification, as well as dh/d calls to **H** for the d root computations.

Table 1 summarizes the required calls to **F**, **H**, and **PRF**. The private and public key consist of $4n$ and $2n$ bits, respectively. The signature adds up to $(h + k(\log t + 1) + d \cdot \mathtt{len} + 1)n$ bits.

4 SECURITY EVALUATION

In this work we make an attempt to unify the security reductions for hash-based signature schemes. We move one of the main differences – the way hashing is done – out of the reduction for

the construction and hide it inside the notion of tweakable hash functions. This allows us to focus on the actual difference in the high-level construction here and discuss the difference in hashing in the tweakable-hash-function section.

In this section, we prove the following Theorem. Note that **F** and **H** are just renamings of **Th** for message lengths n and $2n$. We only treat **F** separately at one point as the length-preserving case needs additional attention in the proof.

THEOREM 17. *For parameters n, w, h, d, m, t, k 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,

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

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

For the definitions of PQ-EU-CMA and PQ-PRF we refer the reader to Appendix A.

4.1 (Post-quantum) interleaved target subset resilience.

Before we start with the proof, we need to define one new security property for hash functions. The security of HORST, the few-time signature scheme used in the original SPHINCS was based on the notion of target subset-resilience. Here, we define a strengthening of this notion called interleaved target subset resilience (ITSR), which captures the use of FORS in SPHINCS$^+$.

The idea for ITSR is that from a theoretical point of view, one can think of the 2^h FORS instances as a single huge HORS-style [43] signature scheme. The secret key consists of 2^h key sets, which in turn each consist of k key subsets of t secret n-byte values. The message digest function $\mathbf{H_{msg}}$ maps a message to a key set (by outputting the corresponding index) and a set of indexes such that each index is used to select one secret value per key subset of the selected key set.

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

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

The former two points will be shown to follow from the properties of **F**, **H**, and **Th** as well as those of **PRF**. The latter point is exactly what ITSR captures.

Definition 18 (PQ-ITSR). Let $\mathrm{H}:\{0,1\}^\kappa\times\{0,1\}^\alpha\rightarrow\{0,1\}^m$ be a keyed hash function. Further consider the mapping function $\mathrm{MAP}_{h,k,t}:\{0,1\}^m\rightarrow\{0,1\}^h\times[0,t-1]^k$ which, for parameters h,k,t, maps an m-bit string to a set of k indexes $((I,1,J_1),\ldots,(I,k,J_k))$, where I is chosen from $[0,2^h-1]$ and each J_i is chosen from $[0,t-1]$. Note that the same I is used for all tuples (I,i,J_i).

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

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

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

We define the PQ-ITSR insecurity of a keyed hash function against q-query, time-ξ adversaries as the maximum advantage of any quantum adversary $\mathcal{A}$ with running time $\le\xi$, that makes no more than q queries:

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

Note that this is actually a weakening of the target subset resilience assumption used in the analysis of SPHINCS in the multi-target setting. In the multi-target version of target subset resilience, $\mathcal{A}$ was able to freely choose the common index I for its output. In interleaved target subset resilience, I is determined by G and input M. We assess the hardness of PQ-ITSR through a complexity analysis of generic attacks against PQ-ITSR in Section 5.

4.2 Security reduction

The security reduction follows largely along the lines of the original SPHINCS security reduction. The new security properties shift certain details towards the analysis of the tweakable hash function and the message-digest function. In the remainder of this section we will prove Theorem 17.

PROOF (OF THEOREM 17). We want to bound the success probability of a (quantum) adversary $\mathcal{A}$ against the EU-CMA security of SPHINCS$^+$. Towards this end we use the following series of games.

We start with GAME.0 which is the EU-CMA experiment for SPHINCS$^+$ ($\mathrm{Exp}^{\text{EU-CMA}}_{\text{SPHINCS}^+}(\mathcal{A})$) as defined in Appendix A. Now consider a GAME.1 which is essentially GAME.0 but the experiment makes use of a SPHINCS$^+$ version where all the outputs of **PRF**, i.e., the WOTS$^+$ and FORS secret-key elements, get replaced by truly random values. Recall that in GAME.0 these were outputs of **PRF** on input secret **SK**.seed and a unique address per generated secret-key element.

Next, consider a game GAME.2, which is the same as GAME.1 but in the signing oracle $\mathbf{PRF_{msg}}(\mathbf{SK}.\mathtt{prf},\cdot)$ is replaced by a truly random function.

Afterwards, we consider GAME.3, which differs from GAME.2 in that we are considering the game lost if an adversary outputs a valid forgery (M, **SIG**) where the FORS signature part of **SIG** contains only secret values which were contained in previous signatures with that FORS key pair obtained by $\mathcal{A}$ via the signing oracle.

Finally, we consider game GAME.4, which differs from GAME.3 in that we are considering the game lost if an adversary outputs a valid forgery (M, **SIG**) which (implicitly or explicitly) contains a second preimage for an input to **Th** that was part of a signature returned as a signing-query response. By implicitly we here refer to a second preimage which is observed during the verification of the signature, e.g., when computing a root node from a leaf and an authentication path.

In the following we bound the differences in success probability of any adversary and the success probability of an adversary in the last game. The different numbers of queries refer to the quantities in the theorem statement.

Analyzing this sequence of games leads to the following five claims which we prove in Appendix E.

Claim 19.

$$\left|\mathrm{Succ}^{GAME.1}(\mathcal{A}) - \mathrm{Succ}^{GAME.0}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}PRF}(\mathbf{PRF}; \xi, q_1).$$

Claim 20.

$$\left|\mathrm{Succ}^{GAME.2}(\mathcal{A}) - \mathrm{Succ}^{GAME.1}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}PRF}\left(\mathbf{PRF_{msg}}; \xi, q_s\right).$$

Claim 21.

$$\left|\mathrm{Succ}^{GAME.3}(\mathcal{A}) - \mathrm{Succ}^{GAME.2}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}ITSR}\left(\mathbf{H_{msg}}; \xi, q_s\right).$$

Claim 22.

$$\left|\mathrm{Succ}^{GAME.4}(\mathcal{A}) - \mathrm{Succ}^{GAME.3}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}(\mathbf{Th}; \xi, q_2).$$

Claim 23.

$$\mathrm{Succ}^{GAME.4}(\mathcal{A}) \leq 3 \cdot \mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}(\mathbf{F}; \xi, q_3) + \mathrm{InSec}^{PQ\text{-}SM\text{-}DSPR}(\mathbf{F}; \xi, q_3).$$

We combine the bounds from the claims to obtain the bound of the theorem. □

5 SECURITY LEVEL / SECURITY AGAINST GENERIC ATTACKS

As shown in Theorem 17, the security reduction for Construction 6, and the security arguments for specific instantiations in the last section, the security of SPHINCS$^+$ relies on properties of the concrete instantiations of all the cryptographic functions and the way they are used. In the following we assume that there are no structural attacks against the used concrete instantiations of H_1, H_2, and H from Construction 6 and Construction 7 as well as for $\mathbf{H_{msg}}$, $\mathbf{PRF_{msg}}$, and **PRF**. We thus consider *generic* classical and quantum attacks against DM-SPR, PRF security, and ITSR. Runtime of adversaries is counted in terms of calls to the used functions. We summarize the bounds in Table 2.

Generic attacks against DM-SPR. To evaluate the complexity of generic attacks against hash-function properties, the hash functions are commonly modeled as random (keyed) functions. For random functions there is no difference between DM-SPR and multi-function multi-target second-preimage resistance (MM-SPR). Every key effectively selects a new random (unkeyed) function, independent of the key being random or not. Hence, the complexity of generic attacks is the same for both notions. We formally show this in Appendix C.

Table 2: Bounds for generic quantum and classical attacks against used function properties. We added a superscript (c) for conjectured bounds. We use $X = \sum_\gamma \left(1-\left(1-\frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma}\left(1-\frac{1}{2^h}\right)^{q_s-\gamma} \frac{1}{2^{h\gamma}}$.

	DM-SPR/ PRF	DM-DSPR$^{(c)}$	ITSR
classical	$\Theta\left(\frac{q+1}{2^n}\right)$	$\Theta\left(\frac{q+1}{2^n}\right)$	$\Theta\left((q+1)X\right)$
quantum	$\Theta\left(\frac{(q+1)^2}{2^n}\right)$	$\Theta\left(\frac{(q+1)^2}{2^n}\right)$	$\Theta\left((q+1)^2X\right)$

In [33] it was shown that the success probability of any classical q-query adversary against MM-SPR of a random function with range $\{0,1\}^n$ (and hence also against DM-SPR) is exactly $(q+1)/2^n$. For q-query quantum adversaries the success probability is $\Theta((q+1)^2/2^n)$. Note that these bounds are independent of the number of targets.

Generic attacks against DM-DSPR. As argued above, for random keyed functions there is no difference between the distinct-function and multi-function cases. In [10] it is shown that the success probability of a quantum adversary against (single-target) DSPR of an n-bit function is $O((q+1)^2/2^n)$. Considering a classical adversary this bound becomes $O(q/2^n)$. Moreover, the authors of [10] give a loose reduction from DSPR to DM-DSPR (which they call T-DSPR). The reduction loses exactly one over the number of targets. However, as also discussed in [10], the best attack against DSPR the authors could think of is executing a high-probability (second-)preimage attack. Given that multi-function multi-target attacks do not give an adversary any advantage over single-target attacks for PRE and SPR, we conjecture that the same holds for DPSR. Hence, we use the above bounds: $O((q+1)^2/2^n)$ for quantum and $O((q+1)/2^n)$ for non-quantum adversaries.

Generic attacks against PRF security. The best generic attack against the PRF security of a keyed function is commonly believed to be exhaustive search for the key. Hence, for a function with key space $\{0,1\}^n$ the success probability of a classical adversary that evaluates the function on q_k keys is bounded by $(q_k+1)/2^n$. For q_k-query quantum adversaries the success probability of exhaustive search in an unstructured space with $\{0,1\}^n$ elements is $\Theta((q_k+1)^2/2^n)$ as implicitly shown in [33] (this can be seen considering this as preimage search in a random function).

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

Recall the parameters h, k, and $t = 2^a$, which define the following process of choosing sets: generate independent uniformly random integers $I, J_1, \ldots, J_k$, with I chosen from $[0, 2^h-1]$ and each J_i chosen from $[0, t-1]$; then define $S = \{(I,1,J_1),(I,2,J_2),\ldots,(I,k,J_k)\}$.

(In the context of SPHINCS+, S is a set of positions of FORS private-key values revealed in a signature: I selects the FORS instance, and J_i selects the position of the value revealed from the ith set inside this FORS instance. To distinguish the number of queries to the oracle from the ITSR game from hash-function queries, we call the former q_s and the latter q_h.)

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

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

Define DarkSide_γ as the conditional probability that $(I_0, i, J_{0,i}) \in S_1 \cup \cdots \cup S_{q_s}$, given that the above number is γ. In other words, $1 - \text{DarkSide}_\gamma$ is the conditional probability that $(I_0, i, J_{0,i})$ is not in the set $\{(I_1, i, J_{1,i}), (I_2, i, J_{2,i}), \ldots, (I_{q_s}, i, J_{q_s,i})\}$. There are exactly γ choices of $\alpha \in \{1, 2, \ldots, q_s\}$ for which $I_\alpha = I_0$, and each of these has probability $1 - 1/t$ of $J_{\alpha,i}$ missing $J_{0,i}$. These probabilities are independent, so $1 - \text{DarkSide}_\gamma = (1 - 1/t)^\gamma$.

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

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

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

Hence, for any classical adversary which makes q_h queries to $\mathbf{H_{msg}}$ the success probability is

$$(q_h + 1) \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma}\left(1 - \frac{1}{2^h}\right)^{q_s-\gamma} \frac{1}{2^{h\gamma}}.$$

For random $\mathbf{H_{msg}}$ the task of finding a message digest that is covered by the previous signatures is search in unstructured data. Hence, we can reduce average search as defined in Definition 31 to this task. This can be shown along the lines of the proofs in Appendix C. This leads to a success probability for quantum adversaries of

$$O\left((q_h + 1)^2 \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma}\left(1 - \frac{1}{2^h}\right)^{q_s-\gamma} \frac{1}{2^{h\gamma}}\right).$$

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

Security Level of a Given Parameter Set. If we take the above success probabilities for generic attacks and plug them into Theorem 10, Theorem 15, and Theorem 17 we get a bound on the success probability of SPHINCS+-robust against generic attacks of classical and quantum adversaries. The final bounds are the same for SPHINCS+-simple up to small constant factors, hidden by the O-notation, given that our conjectures are true. Let q_s denote the number of adversarial signature queries. For classical adversaries that make no more than q_h queries to the cryptographic hash function used, this leads to

$$\text{InSec}^{\text{EU-CMA}}\left(\text{SPHINCS}^+; q_h\right) \leq \frac{q_h + 1}{2^n} + \frac{q_h + 1}{2^n}$$
$$+ \text{InSec}^{\text{ITSR}}\left(\mathbf{H_{msg}}; q_h\right) + \frac{q_h + 1}{2^n} + 3 \cdot \frac{q_h + 1}{2^n} + \frac{q_h + 1}{2^n}$$
$$= O\left(\frac{q_h}{2^n} + q_h \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma}\left(1 - \frac{1}{2^h}\right)^{q_s-\gamma} \frac{1}{2^{h\gamma}}\right).$$

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

$$\text{InSec}^{\text{PQ-EU-CMA}}\left(\text{SPHINCS}^+; q_h\right) \leq \frac{O(q_h + 1)^2}{2^n} + \frac{O(q_h + 1)^2}{2^n}$$
$$+ \text{InSec}^{\text{PQ-ITSR}}\left(\mathbf{H_{msg}}; q_h\right) + \frac{O(q_h + 1)^2}{2^n} + 3 \cdot \frac{O(q_h + 1)^2}{2^n} + \frac{O(q_h + 1)^2}{2^n}$$
$$= O\left(\frac{{q_h}^2}{2^n} + {q_h}^2 \sum_\gamma \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma}\left(1 - \frac{1}{2^h}\right)^{q_s-\gamma} \frac{1}{2^{h\gamma}}\right).$$

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

6 PARAMETER SELECTION AND SPHINCS+ INSTANCES

What is still missing to obtain concrete signature schemes from the SPHINCS+ framework, is choosing parameters and instantiating the tweakable hash functions. We explain our approach to addressing these two aspects in this section and then give examples of concrete instantiations in Section 7.

Selecting parameters. Our approach to selecting the hyper-tree parameters h and d, the FORS parameters b and k, and the Winternitz parameter w is to fix the maximum number of signatures and the target security level and then search through a large space of possible parameter sets. For each of those parameter sets we compute the probability ϵ that an attacker-provided message can be signed with the information known about FORS secret keys after the maximum number of messages has been signed (see Section 5).

For each of the parameter sets with a probability ϵ satisfying the desired security level, we accept the parameter set as a possibly interesting one and print the parameters together with the resulting signature size and an estimate of performance based on the total number of hash calls. In a post-processing step we use standard command-line tools to sort the output according to size or speed and pick the parameter set with the most favorable trade-off for the given application. The complete Python script we use to explore the parameter space is given in Listing 1 in Appendix G and available for download at https://sphincs.org/software.html.

Instantiating tweakable hash functions. Finally, we propose a total of 6 different instantiations of the tweakable hash functions. Concretely, we are using Construction 6 (in the following referred to as *robust*) and Construction 7 (in the following referred to as *simple*). For each of those we recommend three different instantiations of the underlying hash functions H_1 and H_2 (for Construction 6) and H (for Construction 7): SHA-256 [38], SHAKE256 [39], and Haraka [34]. Note that the instantiations using Haraka cannot reach the same security levels that can be reached with SHA-256 or SHAKE256. This is due to a generic meet-in-the-middle attack computing collisions in the internal state, which has (classical) complexity 2^{128}. For a full specification of the instantiations of tweakable hash functions see Appendix F.

7 PERFORMANCE AND COMPARISON

In order to illustrate the performance of signature schemes derived from the SPHINCS$^+$ framework we now give instantiations targeting the security level of other symmetric-crypto-based signature schemes. Specifically, we derive signature schemes to compare to the SPHINCS-256 scheme [9], to the NIST round-1 candidate Gravity-SPHINCS [6], and to the NIST round-2 candidate Picnic [17]. Generally all these stateless signature schemes based only on symmetric primitives do not reach the performance of, e.g., lattice-based signature schemes like Dilithium [22, 23], Falcon [25], or qTESLA [3, 12]. They are mainly interesting for applications without strong latency requirements, such as offline code signing or certificate signing. This makes signature size (and only to a smaller extent public-key size, signing speed, and verification speed) the most important optimization target. In the comparisons, we thus primarily focus on finding parameter sets with similar signature size and then compare computational performance. Note that for hash-based signatures, a rule of thumb is that a linear decrease in signature size comes with an exponential decrease in signing speed. See Table 3 for details on sizes and cycle counts.

Comparison to SPHINCS-256. SPHINCS-256 was the first signature scheme advertising a post-quantum security level of 128 bits. This claim is derived from an analysis of the security of individual building blocks and a theorem stating that the whole scheme is secure as long as each of the building blocks is secure. The statement ignores a significant tightness gap in the proof. Part of this tightness gap was later shown to be more than just a proof artifact, but actually due to attacks that compute a preimage to *one out of many* hashes inside the SPHINCS-256 tree [33]. As a consequence, the actual security of SPHINCS-256 is less than 190 bits classically and 95 bits post-quantum. SPHINCS$^+$ includes protections against such multi-target attacks and can thus achieve this security level with $n = 192$. The hash functions **H** and **F** used in SPHINCS-256 are built from the 512-bit ChaCha12 permutation [8] in a sponge mode with capacity 256 bits. This construction is susceptible to the same kind of meet-in-the-middle collision attacks with complexity 2^{128} that apply to Haraka. For the comparison to SPHINCS-256, we thus choose robust tweakable hash functions derived from Haraka. It should be noted that SPHINCS$^+$ in this case makes slightly stronger assumptions as Construction 6 requires a proof in the QROM to achieve compact public parameters (see Section 2.3 for a discussion of what slightly means in this context). Putting all this together, with parameters $n = 192, h = 51, d = 17, b = 7, k = 45, w = 16$ we obtain a signature scheme, which has signatures that are 25% shorter than SPHINCS-256 signatures, has a signing routine that is 1.7× faster than SPHINCS-256 signing and, like SPHINCS-256, guarantees security for up to 2^{50} signatures under the same key.

Comparison to Gravity-SPHINCS. The second natural comparison is with the NIST round-1 candidate Gravity-SPHINCS [6, 7]. Gravity-SPHINCS aims for a simpler scheme and increased speed at the cost of basing security on collision resistance. Like SPHINCS-256, it does not build in countermeasures against multi-target attacks. So the best attacks against Gravity-SPHINCS—which rely on computing a preimage for one out of many targets—are considerably more efficient than computing a preimage in the underlying hash function. Compared to SPHINCS$^+$ it does not make use of a ROM assumption which SPHINCS$^+$ needs even for the robust parameters (again, note that the assumption here is only necessary to prove the public parameter compression secure). A conservative instantiation of SPHINCS$^+$, which achieves a higher security level both in terms of what is proven and against best known attacks uses parameters $n = 192, h = 66, d = 22, b = 8, k = 33, w = 16$ with robust tweakable hash functions derived from Haraka (which is also used in Gravity-SPHINCS). As Table 3 shows, this instantiation has slightly larger signatures and slightly slower signing speed than Gravity-SPHINCS. However, this is due to a "caching mechanism" in Gravity-SPHINCS that is orthogonal to all design decisions discussed in this paper: Gravity-SPHINCS uses a higher top layer in the tree, computes this layer only once during key generation and stores it in the secret key. This design choice results in somewhat smaller signatures and faster signing at the cost of increased code complexity, much longer key-generation time and much bigger secret-key size.

More similar in spirit to Gravity-SPHINCS is SPHINCS$^+$ with the simple instantiation of tweakable hash functions, which are essentially exactly what Gravity-SPHINCS uses plus multi-target protection. This multi-target protection allows us to choose a smaller value of n to achieve the same level of security against known attacks (requiring second preimages) as Gravity-SPHINCS, but a lower level of security when following the reductions from collision resistance. With parameters $n = 192, h = 64, d = 16, b = 7, k = 49, w = 16$ and the simple construction for tweakable hash functions SPHINCS$^+$ achieves smaller signatures, only slightly slower signing speed, and (because it does not employ the caching mechanism) much faster key generation and smaller secret keys. Note that generally Gravity-SPHINCS has faster verification than SPHINCS$^+$. This is because Gravity-SPHINCS employs plain hashing for node computations, while SPHINCS$^+$ needs more costly calls to tweakable hashes.

Comparison to Picnic. Finally, we compare SPHINCS$^+$ to the only other symmetric-crypto-based NIST round-2 candidate, Picnic [17, 18]. Picnic has three variants, two based on the Fiat-Shamir transform [24] with a non-tight security reduction in the ROM and one based on the Unruh transform [44–46] with a non-tight reduction in the QROM. Signatures of Picnic with the Unruh transform are about 4× larger than those obtained from the SPHINCS$^+$ framework for comparable security levels. Also, the "Picnic1" Fiat-Shamir signatures are more than a factor 2 larger than the speed-optimized instantiations of SPHINCS$^+$ proposed to NIST (see below). The only

Table 3: Performance comparison of symmetric-crypto-based signature schemes on the Intel Haswell microarchitecture. All software is optimized using architecture-specific optimizations such as AESNI or AVX2 instructions.

Scheme	Cycles			Bytes		
	keypair	sign	verify	sig	pk	sk
Comparison to SPHINCS-256						
SPHINCS-256 [9]	2 868 464[a]	50 462 856[a]	1 672 652[a]	41 000	1 056	1 088
SPHINCS+ (Haraka, robust) ($n = 192, h = 51, d = 17, b = 7, k = 45, w = 16$)	1 254 968[b]	29 015 002[b]	2 739 770[b]	30 696	48	96
Comparison to Gravity-SPHINCS						
Gravity-SPHINCS [6] (parameter-set L)	30 729 044 392[a]	32 564 796[a]	625 752[a]	max: 35 168 avg: ?[c]	32	1 048 608
SPHINCS+ (Haraka, robust) ($n = 192, h = 66, d = 22, b = 8, k = 33, w = 16$)	1 257 826[b]	38 840 268[b]	3 467 192[b]	35 664	48	96
SPHINCS+ (Haraka, simple) ($n = 192, h = 64, d = 16, b = 7, k = 49, w = 16$)	1 892 462[b]	35 029 380[b]	1 460 204[b]	30 552	48	96
Comparison to Picnic						
Picnic2-L5-FS [17]	18 244[c]	904 189 188[c]	268 485 212[c]	max: 54 732 avg: 46 282	65	97
SPHINCS+ (SHA-256, simple) ($n = 256, h = 63, d = 9, b = 12, k = 29, w = 16$)	43 317 320[b]	527 413 100[b]	5 463 884[b]	33 408	64	128

[a] As reported by SUPERCOP [11] from 3.5GHz Intel Xeon E3-1275 V3 (Haswell)
[b] Median of 100 runs on 3.5GHz Intel Xeon E3-1275 V3 (Haswell), compiled with `gcc-5.4 -O3 -march=native -fomit-frame-pointer -flto`
[c] As reported by SUPERCOP [11] from 3.1GHz Intel Xeon E3-1220 V3 (Haswell)
[d] Neither [6] nor [7] report the average size of signatures; the analysis in [5] suggests that it is about 1KB smaller than the worst-case size.

Picnic variant that offers signatures with sizes in a similar ballpark as SPHINCS+ is the "Picnic2" instantiation. In Table 3 we compare the NIST level-5 parameter set of Picnic2 with SPHINCS+ using parameters $n = 256, h = 63, d = 9, b = 12, k = 29, w = 16$ and simple tweakable hash functions based on SHA-256. SPHINCS+ signatures with those parameters are 28% smaller than the average Picnic2 signatures and 39% smaller than the worst-case Picnic2 signatures. Signing of SPHINCS+ is more than 70% faster, key generation is much slower, but verification is almost 50 times faster than for Picnic2. This performance of SPHINCS+ is achieved with an instance that has a tight QROM proof and conservative choice of underlying symmetric primitive (SHA-256). The performance of Picnic2 on the other hand heavily relies on version 3 of the rather aggressively optimized symmetric encryption scheme LowMC, which was originally proposed in [2]. As far as we know, this latest version has not been intensively studied; earlier versions were shown to not offer the claimed security [20, 21, 42].

NIST instantiations. The instantiations of SPHINCS+ chosen for comparison to SPHINCS-256, Gravity-SPHINCS, and Picnic are not the *recommended* instantiations. The NIST submission of SPHINCS+ includes a total of 36 instantiations (3 security levels, 3 hash functions, simple and robust instantiations, speed and size optimized). The performance of all NIST instantiations in terms of sizes and cycle counts for optimized software is presented in Table 4 in Appendix H.

ACKNOWLEDGMENTS

The authors would like to thank Jean-Philippe Aumasson, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Tanja Lange, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger, for their support and comments, and the anonymous reviewers of CCS for finding a mistake in an earlier version of this work. This work has been supported by the European Research Council through Starting Grant No. 805031 (EPOQUE), by Cisco under the University Research Program, by the U.S. National Science Foundation under grant 1913167, and by the German Research Foundation under Cluster of Excellence 2092 "CASA: Cyber Security in the Age of Large-Scale Adversaries".

REFERENCES

[1] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. 2019. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. NISTIR 8240. available online at https://doi.org/10.6028/NIST.IR.8240.
[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Advances in Cryptology – EUROCRYPT 2015 (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9056. Springer, 430–454. https://eprint.iacr.org/2016/687.

[3] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Patrick Longa, and Jefferson E. Ricardini. 2019. The Lattice-Based Digital Signature Scheme qTESLA. Cryptology ePrint Archive, Report 2019/085. https://eprint.iacr.org/2019/085.

[4] Jean-Philippe Aumasson, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. 2019. SPHINCS$^+$. Submission to NIST's post-quantum crypto standardization project. http://sphincs.org/data/sphincs+-round2-specification.pdf.

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

[6] Jean-Philippe Aumasson and Guillaume Endignoux. 2017. Gravity-SPHINCS. Submission to the NIST PQC project. https://github.com/gravity-postquantum/gravity-sphincs/blob/master/Supporting_Documentation/submission.pdf.

[7] Jean-Philippe Aumasson and Guillaume Endignoux. 2018. Improving stateless hash-based signatures. In *Topics in Cryptology – CT-RSA 2018 (LNCS)*, Nigel P. Smart (Ed.), Vol. 10808. Springer, 219–242. https://eprint.iacr.org/2017/933.

[8] Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20. SASC 2008: The State of the Art of Stream Ciphers.

[9] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). LNCS, Vol. 9056. Springer, 368–397. https://eprint.iacr.org/2014/795.

[10] Daniel J. Bernstein and Andreas Hülsing. 2018. Decisional second-preimage resistance: When does SPR imply PRE? https://eprint.iacr.org/2019/492.pdf.

[11] Daniel J. Bernstein and Tanja Lange. accessed 2019-05-10. eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to.

[12] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Krämer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. 2019. Submission to NIST's post-quantum project (2nd round): lattice-based digital signature scheme qTESLA. Round-2 submission to the NIST PQC project. https://qtesla.org/wp-content/uploads/2019/04/qTESLA_round2_04.26.2019.pdf.

[13] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. 2011. Random Oracles in a Quantum World. In *ASIACRYPT 2011*, DongHoon Lee and Xiaoyun Wang (Eds.). LNCS, Vol. 7073. Springer, 41–69.

[14] Leon Groot Bruinderink and Andreas Hülsing. 2017. "Oops, I did it again" – Security of One-Time Signatures under Two-Message Attacks. In *International Conference on Selected Areas in Cryptography – SAC 2017 (LNCS)*, Carlisle Adams and Jan Camenisch (Eds.). Springer, 299–322. https://eprint.iacr.org/2016/1042.

[15] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. 2011. On the Security of the Winternitz One-Time Signature Scheme. In *Africacrypt 2011*, A. Nitaj and D. Pointcheval (Eds.). LNCS, Vol. 6737. Springer, 363–378.

[16] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. 2011. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In *Post-Quantum Cryptography*, Bo-Yin Yang (Ed.). LNCS, Vol. 7071. Springer, 117–129. https://eprint.iacr.org/2011/484.

[17] Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. 2019. The Picnic Signature Scheme – Design Document. Round-2 submission to the NIST PQC project. version 2.0, https://github.com/microsoft/Picnic/blob/master/spec/design-v2.0.pdf.

[18] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. 2017. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS'17*. ACM, 1825–1842. https://eprint.iacr.org/2017/279.

[19] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. 2008. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In *Post-Quantum Cryptography*, Johannes Buchmann and Jintai Ding (Eds.). LNCS, Vol. 5299. Springer, 109–123.

[20] Itai Dinur, Yunwen Liu, Willi Meier, and Qingju Wang. 2015. Optimized Interpolation Attacks on LowMC. In *Advances in Cryptology – ASIACRYPT 2015 (LNCS)*, Tetsu Iwata and Jung Hee Cheon (Eds.), Vol. 9558. Springer, 535–560. https://eprint.iacr.org/2015/418.

[21] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. 2015. Higher-Order Cryptanalysis of LowMC. In *Information Security and Cryptology – ICISC 2015 (LNCS)*, Soonhak Kwon and Aaram Yun (Eds.), Vol. 9558. Springer, 87–101. https://eprint.iacr.org/2015/407.

[22] Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, , and Damien Stehlé. 2019. CRYSTALS–Dilithium: Algorithm Specification and Supporting Documentation. Round-2 submission to the NIST PQC project. https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf.

[23] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. *Transactions on Cryptographic Hardware and Embedded Systems* 1 (2018), 238–268. Issue 2018.

[24] Amos Fiat and Adi Shamir. 1986. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology – CRYPTO '86 (LNCS)*, Andrew M. Odlyzko (Ed.), Vol. 263. Springer, 186–194.

[25] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2019. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU – Specifications v1.1. Round-2 submission to the NIST PQC project. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Falcon-Round2.zip.

[26] Oded Goldreich. 1987. Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme. In *Advances in Cryptology - CRYPTO '86*, Andrew M. Odlyzko (Ed.). LNCS, Vol. 263. Springer, 104–110.

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

[28] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.* 17, 2 (1988), 281–308.

[29] Andreas Hülsing. 2013. *Practical Forward Secure Signatures using Minimal Security Assumptions*. Ph.D. Dissertation. TU Darmstadt. http://tuprints.ulb.tu-darmstadt.de/3651.

[30] Andreas Hülsing. 2013. W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In *Progress in Cryptology – AFRICACRYPT 2013 (LNCS)*, Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien (Eds.), Vol. 7918. Springer, 173–188. https://eprint.iacr.org/2017/965.

[31] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. 2018. XMSS: eXtended Merkle Signature Scheme. RFC 8391. https://doi.org/10.17487/RFC8391 https://rfc-editor.org/rfc/rfc8391.txt.

[32] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. 2013. Optimal Parameters for XMSS^{MT}. In *Security Engineering and Intelligence Informatics*, Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu (Eds.). LNCS, Vol. 8128. Springer, 194–208. https://eprint.iacr.org/2017/966.

[33] Andreas Hülsing, Joost Rijneveld, and Fang Song. 2016. Mitigating Multi-target Attacks in Hash-Based Signatures. In *PKC 2016 (LNCS)*, Chen-Mou Cheng, Kai-Min Chung, Guiseppe Persiano, and Bo-Yin Yang (Eds.), Vol. 9614. Springer, 387–416. https://eprint.iacr.org/2015/1256.

[34] Stefan Kölbl, Martin Lauridsen, Florian Mendel, and Christian Rechberger. 2017. Haraka v2 – Efficient Short-Input Hashing for Post-Quantum Applications. *IACR Transactions on Symmetric Cryptology* 2016, 2 (2017), 1–29. https://doi.org/10.13154/tosc.v2016.i2.1-29 https://eprint.iacr.org/2016/098.

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

[36] David McGrew, Michael Curcio, and Scott Fluhrer. 2019. Leighton-Micali Hash-Based Signatures. RFC 8554. https://doi.org/10.17487/RFC8554

[37] Ralph Merkle. 1990. A Certified Digital Signature. In *Advances in Cryptology – CRYPTO '89 (LNCS)*, Gilles Brassard (Ed.), Vol. 435. Springer, 218–238.

[38] NIST 2015. FIPS PUB 180-4: Secure Hash Standard (SHS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[39] NIST 2015. FIPS PUB 202 – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[40] NIST. 2016. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[41] Bart Preneel and Paul C. van Oorschot. 1995. MDx-MAC and Building Fast MACs from Hash Functions. In *Advances in Cryptology – CRYPTO '95 (LNCS)*, Vol. 963. Springer, 1–14.

[42] Christian Rechberger, Hadi Soleimany, and Tyge Tiessen. 2018. Cryptanalysis of Low-Data Instances of Full LowMCv2. *IACR Transactions on Symmetric Cryptology* 2018, 3 (2018), 163–181. https://doi.org/10.13154/tosc.v2018.i3.163-181.

[43] Leonid Reyzin and Natan Reyzin. 2002. Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying. In *Information Security and Privacy 2002*, Lynn Batten and Jennifer Seberry (Eds.). LNCS, Vol. 2384. Springer, 1–47.

[44] Dominique Unruh. 2012. Quantum Proofs of Knowledge. In *Advances in Cryptology – EUROCRYPT 2012 (LNCS)*, David Pointcheval and Thomas Johansson (Eds.), Vol. 7237. Springer, 135–152.

[45] Dominique Unruh. 2015. Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model. In *Advances in Cryptology – EUROCRYPT 2015 (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9056. Springer, 755–784.

[46] Dominique Unruh. 2016. Computationally binding quantum commitments. In *Advances in Cryptology – EUROCRYPT 2016 (LNCS)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9666. Springer, 497–527.

A SECURITY MODELS AND DEFINITIONS

In the following we discuss post-quantum security and the quantum-accessible random oracle model (QROM). Afterwards we recall the definitions of post-quantum existential unforgeability under adaptive chosen-message attacks (PQ-EU-CMA) and post-quantum-secure pseudorandom functions (PQ-PRF).

Post-quantum security and the QROM. In this work we are concerned with the post-quantum security of the cryptographic schemes presented. In this setting, we assume that all honest parties use conventional hardware (often referred to as being 'classical'). Malicious parties, i.e., all adversaries, are generally assumed to have access to a large-scale quantum computer (often referred to as being 'quantum'). In consequence, all oracles that model an honest user and take an input unknown to the adversary are restricted to conventional queries. The only exception to this in a post-quantum setting are oracles that represent idealized, unkeyed primitives like random oracles (RO)[2] that do not take any input unknown to the adversary. These oracles model functions that in reality could be implemented by any the adversary locally on its quantum computer as – in contrast to the oracles discussed above – the adversary controls all inputs as well as a description of the function (in case of ROs, the description would be the publicly available code of the hash function used to instantiate it). Consequently, adversaries (and reductions) have to be granted quantum access to these oracles. Formally, this means that for the case of an RO F, executions of the unitaries describing the adversary are interleaved with executions of an oracle unitary

$$O_f : \sum_{x,y} \alpha_{x,y} |x\rangle|y\rangle \rightarrow \sum_{x,y} \alpha_{x,y} |x\rangle|y \oplus f(x)\rangle ,$$

i.e., the adversary is described by a sequence of unitaries $U_0, \ldots, U_q$ and executed as

$$U_q O_f U_{q-1} O_f \ldots O_f U_0 |0\rangle .$$

For more details on the QROM see the original paper [13]. In addition to introducing the QROM, the authors of [13] also showed that history-free ROM reductions imply a reduction in the QROM. The observation made there is that if the reduction finishes all possibly necessary manipulations of the RO before the adversary is executed, the RO can efficiently be simulated.

Post-quantum EU-CMA security. The standard security notion for digital signature schemes is existential unforgeability under adaptive chosen-message attacks (EU-CMA) [28]. The notion is defined using the following experiment for signature scheme SIG. In the experiment, the adversary $\mathcal{A}$ is given access to a signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ which is initialized with the target secret key. The q queries to $\mathsf{Sign}(\mathsf{sk}, \cdot)$ are denoted $\{M_i\}_1^{q_s}$. Following the reasoning above, even quantum adversaries are limited to classical queries to this oracle as it simulates an honest and hence classical user.

Experiment $\mathsf{Exp}_{\mathrm{SIG}}^{\mathrm{EU\text{-}CMA}}(\mathcal{A})$
 $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{kg}()$
 $(M^\star, \sigma^\star) \leftarrow \mathcal{A}^{\mathsf{sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$
 Return 1 iff $\mathsf{vf}(\mathsf{pk}, M^\star, \sigma^\star) = 1$ and $M^\star \notin \{M_i\}_1^{q_s}$.

[2] On a meta-level, another exception would be cryptographic primitives used to simulate idealized primitives in a reduction but that is not relevant for the work at hand.

Definition 24 (PQ-EU-CMA). Let SIG be a digital signature scheme. We define the success probability of an adversary $\mathcal{A}$ against the EU-CMA security of SIG as the probability that the above experiment outputs 1:

$$\mathrm{Succ}_{\mathrm{SIG}}^{\mathrm{EU\text{-}CMA}}(\mathcal{A}) = \Pr\left[\mathsf{Exp}_{\mathrm{SIG}}^{\mathrm{EU\text{-}CMA}}(\mathcal{A}) = 1\right].$$

We define the PQ-EU-CMA insecurity of a signature scheme SIG against q_s-query, time-ξ adversaries as the maximum advantage of any possibly quantum adversary that runs in time ξ and makes no more then q_s queries to its signing oracle:

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

Post-quantum PRF security. In the following we give the definition for PRF security of a keyed function $\mathrm{F} : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$. In the definition of the PRF distinguishing advantage the adversary $\mathcal{A}$ gets (classical) oracle access to either F_K for a uniformly random key $K \in \mathcal{K}$ or to a function G drawn from the uniform distribution over the set $\mathcal{G}(\alpha, n)$ of all functions with domain $\{0,1\}^\alpha$ and range $\{0,1\}^n$. The goal of $\mathcal{A}$ is to distinguish both cases.

Definition 25 (PQ-PRF). Let F be defined as above. We define the PRF distinguishing advantage of an adversary $\mathcal{A}$ as

$$\mathrm{Adv}_{\mathrm{F}}^{\mathrm{PRF}}(\mathcal{A}) = \left| \Pr_{K \leftarrow_R \mathcal{K}}\left[\mathcal{A}^{\mathrm{F}_K} = 1\right] - \Pr_{\mathrm{G} \leftarrow_R \mathcal{G}(m,n)}\left[\mathcal{A}^{\mathrm{G}} = 1\right] \right| .$$

We define the PQ-PRF insecurity of a keyed function F against q-query, time-ξ adversaries as the maximum advantage of any possibly quantum adversary that runs in time ξ and makes no more then q queries to its oracle:

$$\mathrm{InSec}^{\mathrm{PQ\text{-}PRF}}(\mathrm{F}; \xi, q) = \max_{\mathcal{A}} \left\{\mathrm{Adv}_{\mathrm{F}}^{\mathrm{PRF}}(\mathcal{A})\right\} .$$

B SM-TCR AND COLLISION RESISTANCE FOR TWEAKABLE HASH FUNCTIONS

The notion of SM-TCR is a collision-finding notion hence we give a comparison to collision resistance (CR) in the following. For this we briefly introduce collision resistance for tweakable hash functions, argue that it implies SM-TCR and afterwards show that SM-TCR is strictly weaker than CR under the assumption that a certain kind of SM-TCR hash functions exist.

Definition 26 (PQ-CR). Let $\mathbf{Th}$ be a tweakable hash function as defined above. We define the success probability of an adversary $\mathcal{A}$ against CR as

$$\mathrm{Succ}_{\mathbf{Th}}^{\mathrm{CR}}(\mathcal{A}) = \Pr[P \leftarrow_R \mathcal{P}, ((T_1, M_1), (T_2, M_2)) \leftarrow \mathcal{A}(\mathcal{P}) : \mathbf{Th}(P, T_1, M_1) = \mathbf{Th}(P, T_2, M_2) \wedge (T_1, M_1) \neq (T_2, M_2)] .$$

We define the PQ-CR insecurity of a tweakable hash function $\mathbf{Th}$ against time-ξ adversaries as the maximum advantage of any possibly quantum adversary that runs in time ξ:

$$\mathrm{InSec}^{\mathrm{PQ\text{-}CR}}(\mathbf{Th}; \xi) = \max_{\mathcal{A}} \left\{\mathrm{Succ}_{\mathbf{Th}}^{\mathrm{CR}}(\mathcal{A})\right\} .$$

We first argue that collision resistance implies SM-TCR.

THEOREM 27. *Let $\mathbf{Th}$ be a tweakable hash function. Then for any p, the success probability of any time-ξ (quantum) adversary $\mathcal{A}$ against SM-TCR of $\mathbf{Th}$ is bounded by*

$$\mathrm{Succ}_{\mathbf{Th},p}^{\mathit{SM\text{-}TCR}}(\mathcal{A}) \leq \mathrm{InSec}^{\mathit{PQ\text{-}CR}}(\mathbf{Th}; \xi) .$$

Proof. Towards a contradiction, assume there exists a time-ξ adversary $\mathcal{A}$ that succeeds in breaking SM-TCR of $\mathbf{Th}$ with probability greater $\mathrm{InSec}^{\text{PQ-CR}}(\mathbf{Th};\xi)$. We build an oracle machine $\mathcal{M}^{\mathcal{A}}$ against CR of $\mathbf{Th}$ as follows. Given a public parameter P, $\mathcal{M}^{\mathcal{A}}$ runs $\mathcal{A}_1$ and simulates the $\mathbf{Th}$ oracle using P. When $\mathcal{A}_1$ is done, $\mathcal{M}^{\mathcal{A}}$ runs $\mathcal{A}_2$. When $\mathcal{A}_2$ outputs a colliding M under P, for some tweak T_j and message M_j, $\mathcal{M}^{\mathcal{A}}$ outputs $\left((T_j, M), (T_j, M_j)\right)$. This is a valid collision for $\mathbf{Th}$, hence, $\mathcal{M}^{\mathcal{A}}$ succeeds whenever $\mathcal{A}$ succeeds. As $\mathcal{M}^{\mathcal{A}}$ can perfectly simulate the oracle to $\mathcal{A}$, the adversary succeeds with the same probability as in the original SM-TCR game. Moreover, $\mathcal{M}^{\mathcal{A}}$ runs in essentially the same time as $\mathcal{A}$. Hence, $\mathcal{M}^{\mathcal{A}}$ finds a collision with probability greater $\mathrm{InSec}^{\text{PQ-CR}}(\mathbf{Th};\xi)$ which contradicts the definition of $\mathrm{InSec}^{\text{PQ-CR}}(\mathbf{Th};\xi)$. □

For the other direction we give a result that shows a separation between CR and SM-TCR for tweakable hash functions which are key-one-way for distinct tweaks (KOW) as defined next.

Definition 28 (PQ-KOW). Let $\mathbf{Th}$ be a tweakable hash function as defined above. We define the advantage of any adversary $\mathcal{A}$ against KOW of $\mathbf{Th}$. The definition is parameterized by the number of queries q for which it must hold that $q \leq |\mathcal{T}|$. In the definition, $\mathcal{A}$ is allowed to make q queries to an oracle $\mathbf{Th}(P,\cdot,\cdot)$. The query set Q and predicate $\mathbf{DIST}(\{T_i\}_{i=1}^{q})$, are defined as in Definition 2.

$$\mathrm{Succ}^{\text{KOW}}_{\mathbf{Th},q}(\mathcal{A}) = \Pr\left[P \leftarrow_R \mathcal{P}, P' \leftarrow \mathcal{A}^{\mathbf{Th}(P,\cdot,\cdot)}() : \mathbf{DIST}(\{T_i\}_{i=1}^{q}) \wedge P = P'\right].$$

We define the PQ-KOW insecurity of a tweakable hash function $\mathbf{Th}$ against q-query, time-ξ adversaries as the maximum advantage of any possibly quantum adversary that runs in time ξ and makes no more than q queries to its oracle:

$$\mathrm{InSec}^{\text{PQ-KOW}}(\mathbf{Th};\xi,q) = \max_{\mathcal{A}}\left\{\mathrm{Succ}^{\text{KOW}}_{\mathbf{Th},q}(\mathcal{A})\right\}.$$

Using this we can show the following result:

Theorem 29. *Let $\mathbf{Th}$ be a tweakable hash function. Consider the tweakable hash function $\mathbf{Th}'$ defined using an additional bit B in the message as*

$$\mathbf{Th}'(P,T,B\|M) = \begin{cases} 0\|\mathbf{Th}(P,T,M) & \text{, if } M = P \\ B\|\mathbf{Th}(P,T,M) & \text{, otherwise.}\end{cases}$$

Then the algorithm that on input P selects an arbitrary tweak $T \in \mathcal{T}$ and outputs $((T, 0\|P), (T, 1\|P))$ is a constant time, success-probability-1 collision finder.

In addition, the success probability of any (quantum) adversary $\mathcal{A}$ against SM-TCR of $\mathbf{Th}'$ that runs in time ξ and makes at most p queries to its oracle is bounded by

$$\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th}',p}(\mathcal{A}) \leq \mathrm{InSec}^{\text{PQ-SM-TCR}}(\mathbf{Th};\xi,p) + p\cdot\mathrm{InSec}^{\text{PQ-KOW}}(\mathbf{Th};\xi,p).$$

Proof. The statement about the collision finder is true by construction. It remains to show the second statement of the theorem. Take any SM-TCR adversary $\mathcal{A}$ against $\mathbf{Th}'$. Now consider the following oracle machines $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{B}^{\mathcal{A}}$.

The oracle machine $\mathcal{M}^{\mathcal{A}}$ uses $\mathcal{A}$ to attack KOW of $\mathbf{Th}$. For this it answers $\mathcal{A}_1$'s oracle queries by first stripping off the bit B, then forwarding the query to its oracle and prepending B to the response. Eventually, $\mathcal{M}^{\mathcal{A}}$ samples an index $i \leftarrow_R [1,p]$ uniformly at random. When $\mathcal{A}_1$ is done, $\mathcal{M}^{\mathcal{A}}$ outputs M_i, where M_i is the message part (without the first bit B) of the ith oracle query.

The oracle machine $\mathcal{B}^{\mathcal{A}}$ uses $\mathcal{A}$ to attack SM-TCR of $\mathbf{Th}$. For this, $\mathcal{B}^{\mathcal{A}}$ runs $\mathcal{A}_1$ and answers $\mathcal{A}_1$'s oracle queries the same way $\mathcal{M}^{\mathcal{A}}$ does. Then it runs $\mathcal{A}_2$. When $\mathcal{A}_2$ outputs a SM-TCR solution $(j, B\|M)$, $\mathcal{B}^{\mathcal{A}}$ outputs (j, M).

Now we break down the case that $\mathcal{A}$ succeeds into two mutually exclusive cases. In the first case, the M part of at least one of $\mathcal{A}_1$'s oracle queries is P. In the second case, the M part of none of $\mathcal{A}_1$'s oracle queries is P. In the first case, $\mathcal{M}^{\mathcal{A}}$ outputs P with a probability of at least $1/p$. Note that although the simulation might not be perfect in case the query was $1\|P$, this does not alter $\mathcal{M}^{\mathcal{A}}$'s success probability.

In the second case, $\mathcal{B}^{\mathcal{A}}$ outputs a valid SM-TCR solution for $\mathbf{Th}$ with probability 1. For this note that conditioned on the second case $\mathcal{A}_1$ makes no query where the M part is P. Consequently, $\mathcal{B}^{\mathcal{A}}$'s $\mathbf{Th}$ oracle behaves identical to the $\mathbf{Th}'$ oracle in the real game. Now consider the colliding tweak-message pairs $(T_j, B_j\|M_j), (T_j, B\|M)$ referenced by the SM-TCR solution $(j, B\|M)$ returned by $\mathcal{A}$. For the very same reason as above, we have that $M_j \neq P$. In consequence, we know that $B_j = B$ as by construction the values would not collide otherwise. Therefore, $(T_j, M_j), (T_j, M)$ has to collide under $\mathbf{Th}(P,\cdot,\cdot)$ and so $(j, B\|M)$ is a valid SM-TCR solution for $\mathbf{Th}$.

In sum, the success probability of $\mathcal{A}$ is bound by $\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th}}\left(\mathcal{B}^{\mathcal{A}}\right) + p\cdot\mathrm{Succ}^{\text{KOW}}_{\mathbf{Th}}\left(\mathcal{M}^{\mathcal{A}}\right)$, which concludes the proof. □

This shows that if tweakable hash functions exist that are PQ-KOW and PQ-SM-TCR, then PQ-SM-TCR is a strictly weaker assumption then PQ-CR.

C HARDNESS OF PQ-MM-SPR AND PQ-DM-SPR

In Definition 8 we defined DM-SPR and explained the difference with MM-SPR. In Section 5 we discuss that the query complexity of generic attacks against these two notions is the same. In the following we formally prove this.

The following result on the hardness of MM-SPR is shown in [33]:[3]

Lemma 30 ([33]). *For any q-query quantum adversary $\mathcal{A}$, it holds that*

$$\mathrm{Succ}^{\text{MM-SPR}}_{\mathcal{H}_n}(\mathcal{A}) \leq 8(2q+1)^2/2^n.$$

The proof follows a framework that starts with an average case search problem. The problem makes use of the following distribution D_λ over boolean functions.

Definition 31 ([33]). Let $\mathcal{F} \overset{\text{def}}{=} \{f : \{0,1\}^m \rightarrow \{0,1\}\}$ be the collection of all boolean functions on $\{0,1\}^m$. Let $\lambda \in [0,1]$ and $\varepsilon > 0$. Define a family of distributions D_λ on $\mathcal{F}$ such that $f \leftarrow_R D_\lambda$ satisfies

$$f : x \mapsto \begin{cases} 1 & \text{with prob. } \lambda, \\ 0 & \text{with prob. } 1-\lambda \end{cases}$$

for any $x \in \{0,1\}^m$.

[3] The bound stated in [33] actually was $16(q+1)^2/2^n$. This missed that the factor 2 overhead in queries also gets squared.

Using this distribution we can define the average case search problem Avg-Search$_\lambda$ as the problem that given oracle access to $f \leftarrow D_\lambda$, finds an x such that $f(x) = 1$. For any q-query quantum algorithm $\mathcal{A}$

$$\mathrm{Succ}^{\text{Avg-Search}_\lambda}(\mathcal{A}) := \Pr_{f \leftarrow D_\lambda}\left[f(x) = 1 : x \leftarrow \mathcal{A}^f(\cdot)\right].$$

For this average case search problem the authors give a quantum query bound.

Lemma 32 ([33]). *For any quantum algorithm $\mathcal{A}$ with q queries it holds that* $\mathrm{Succ}^{\text{Avg-Search}_\lambda}(\mathcal{A}) \leq 8\lambda(q+1)^2$.

The reduction then generates the mm-spr challenge as described in Figure 4.

Given: $f \leftarrow D_\lambda : [p] \times \{0,1\}^\alpha \rightarrow \{0,1\}$, $\lambda = 1/2^n$.

(1) For $i = 1, \ldots, p$, sample $x_i \leftarrow \{0,1\}^\alpha$ and $y_i \leftarrow \{0,1\}^n$ independently and uniformly at random. Denote $S = \{x_i\}_1^p$.

(2) For $i = 1, \ldots, p$, let $g_i : \{0,1\}^\alpha \rightarrow \{0,1\}^n \backslash \{y_i\}$ be a random function. We construct $\tilde{H}_i : \{0,1\}^\alpha \rightarrow \{0,1\}^n$ as follows: for any $x \in \{0,1\}^\alpha$

$$x \mapsto \begin{cases} y_i & \text{if } x = x_i \\ y_i & \text{if } x \neq x_i \wedge f(i\|x) = 1 \\ g_i(x) & \text{otherwise.} \end{cases}$$

Output: mm-spr instance $(S, \{\tilde{H}_i\}_{i=1}^p)$. Namely an adversary is given x_i and oracle access to $\tilde{H}_i$, and the goal is to find (i^*, x^*) such that $x^* \neq x_{i^*}$ and $\tilde{H}_{i^*}(x^*) = \tilde{H}_{i^*}(x_{i^*}) = y_{i^*}$.

Figure 4: Reducing Avg-Search **to mm-spr.**

We will now show that the same bound applies for dm-spr:

Theorem 33. *For any q-query quantum adversary $\mathcal{A}$, it holds that*

$$\mathrm{Succ}^{\text{DM-SPR}}_{\mathcal{H},p}(\mathcal{A}) \leq 8(2q+1)^2/2^n.$$

The proof of Theorem 33 is a straightforward combination of Lemma 32 and the following Lemma.

Lemma 34. *Let $\mathcal{H}_n$ as defined above be a family of random functions. Any quantum adversary $\mathcal{A}$ that solves dm-spr making q quantum queries to $\mathcal{H}_n$ can be used to construct a quantum adversary $\mathcal{B}$ that makes $2q$ queries to its oracle and solves* Avg-Search$_{\frac{1}{2^n}}$ *with success probability*

$$\mathrm{Succ}^{\text{Avg-Search}_{\frac{1}{2^n}}}(\mathcal{B}) \geq \mathrm{Succ}^{\text{DM-SPR}}_{\mathcal{H},p}(\mathcal{A}).$$

Proof. In general the proof follows exactly the same reasoning, as the mm-spr proof. For dm-spr things are in general slightly more complicated when considering a random function family. The reason is that we have to give both $\mathcal{A}_2$ *and* $\mathcal{A}_1$ oracle access to the function family to select the target functions.

Hence, $\mathcal{B}$ has to simulate the full function family. Although we are interested in query complexity, we decided to give a reduction $\mathcal{B}$ that simulates the function family efficiently. The random functions e_0, e_1, and g can be efficiently simulated using $2q$-wise independent hash functions as discussed in [33].

The reduction $\mathcal{B}$ generates function (family) $\tilde{H}$ as shown in Figure 5. Then it runs $\mathcal{A}_1$ with $\tilde{H}$ as oracle.

Given: $f \leftarrow D_\lambda : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}$, $\lambda = \frac{1}{2^n}$.

(1) Let $e_0 : \mathcal{K} \rightarrow \{0,1\}^\alpha$ and $e_1 : \mathcal{K} \rightarrow \{0,1\}^n$ be two random functions.

(2) Let $g = \{g_K : \{0,1\}^\alpha \rightarrow \{0,1\}^n \backslash \{e_1(K)\} \mid K \in \mathcal{K}\}$ be a family of random functions. We construct $\tilde{H}_K : \{0,1\}^\alpha \rightarrow \{0,1\}^n$ as follows: for any $X \in \{0,1\}^\alpha$

$$x \mapsto \begin{cases} e_1(K) & \text{if } X = e_0(K) \\ e_1(K) & \text{if } X \neq e_0(K) \wedge f(K\|X) = 1 \\ g_K(X) & \text{otherwise.} \end{cases}$$

Output: Function family $\tilde{H}$, e_0, e_1.

Figure 5: Reducing Avg-Search **to dm-spr.**

When $\mathcal{A}_1$ returns the target key set $\{K_i\}_1^q$, $\mathcal{B}$ completes the dm-spr challenge adding $e_0(K_i)$ to each K_i. Then $\mathcal{B}$ runs $\mathcal{A}_2$ on input $\{K_i, e_0(K_i)\}_1^q$, again giving oracle access to $\tilde{H}$. When $\mathcal{A}_2$ returns (j, x'), $\mathcal{B}$ outputs $K_j\|x'$.

Per construction, $f(K_j\|X') = 1$ whenever (j, X') is a valid dm-spr solution. Moreover, the combined distribution of $\tilde{H}$ and $\{K_i, e_0(K_i)\}_1^q$ is exactly that of a dm-spr challenge. Hence, $\mathcal{B}$ succeeds exactly with $\mathcal{A}$'s success probability in the dm-spr game. $\mathcal{B}$ makes twice the number of oracle queries as it has to uncompute the oracle results after use. □

D PROOF OF Theorem 11

Recall Construction 7:

Construction 7. *Given a hash function $H : \{0,1\}^{2n+\alpha} \rightarrow \{0,1\}^n$, we construct* **Th** *with $\mathcal{P} = \mathcal{T} = \{0,1\}^n$ as*

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

We now give the proof for Theorem 11:

Theorem 11. *Let H be a hash function as in Construction 7, modeled as quantum-accessible random oracle, and* **Th** *the tweakable hash function constructed by Construction 7. Then the success probability of any (quantum) adversary $\mathcal{A}$ making at most q-queries to H, against sm-tcr of* **Th** *is bounded by*

$$\mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}) \leq 8(2q+1)^2/2^n,$$

when $\mathcal{A}_1$ is not given access to the random oracle.

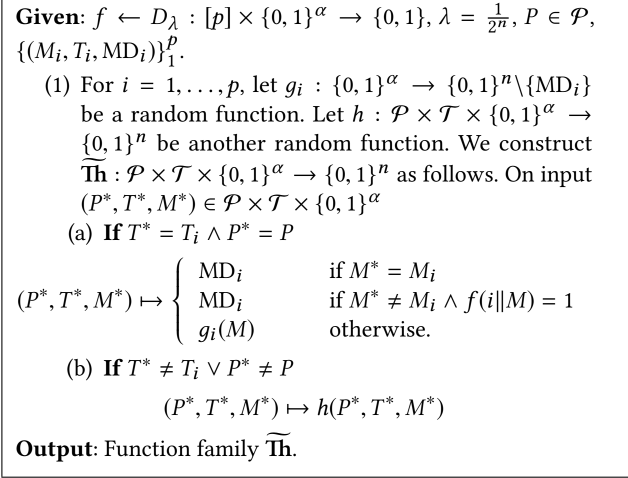
The proof of Theorem 11 is a straightforward combination of Lemma 32 and the following Lemma.

Lemma 35. *Let* **Th** *be the tweakable hash function as given by Construction 7. Any quantum adversary $\mathcal{A}$ that solves sm-tcr making q quantum queries to* **Th** *can be used to construct a quantum adversary $\mathcal{B}$ that makes $2q$ queries to its oracle and solves* Avg-Search$_{\frac{1}{2^n}}$ *with success probability*

$$\mathrm{Succ}^{\text{Avg-Search}_{\frac{1}{2^n}}}(\mathcal{B}) \geq \mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th},p}(\mathcal{A}).$$

Proof. The proof follows exactly the same outline, as the previous proofs. For SM-TCR things are more complicated as we have an initial challenge generation phase, interacting with $\mathcal{A}_1$.

However, the interaction with $\mathcal{A}_1$ is straight forward. For every query (M_i, T_i), $\mathcal{B}$ samples a random output MD_i and stores the tuple $(M_i, T_i, \mathrm{MD}_i)$ in a list. When $\mathcal{A}_1$ did all its p queries, $\mathcal{B}$ samples a random $P \leftarrow_R \mathcal{P}$ and generates $\widetilde{\mathbf{Th}}$ as shown in Figure 6.



Given: $f \leftarrow D_\lambda : [p] \times \{0,1\}^\alpha \rightarrow \{0,1\}$, $\lambda = \frac{1}{2^n}$, $P \in \mathcal{P}$, $\{(M_i, T_i, \mathrm{MD}_i)\}_1^p$.

(1) For $i = 1, \ldots, p$, let $g_i : \{0,1\}^\alpha \rightarrow \{0,1\}^n \backslash \{\mathrm{MD}_i\}$ be a random function. Let $h : \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ be another random function. We construct $\widetilde{\mathbf{Th}} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ as follows. On input $(P^*, T^*, M^*) \in \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha$

(a) **If** $T^* = T_i \wedge P^* = P$

$$(P^*, T^*, M^*) \mapsto \begin{cases} \mathrm{MD}_i & \text{if } M^* = M_i \\ \mathrm{MD}_i & \text{if } M^* \neq M_i \wedge f(i \| M) = 1 \\ g_i(M) & \text{otherwise.} \end{cases}$$

(b) **If** $T^* \neq T_i \vee P^* \neq P$

$$(P^*, T^*, M^*) \mapsto h(P^*, T^*, M^*)$$

Output: Function family $\widetilde{\mathbf{Th}}$.

Figure 6: Reducing Avg-Search **to SM-TCR.**

□

The construction essentially assigns a manipulated random function to every combination of P with a tweak used by $\mathcal{A}_1$ and a uniformly random function to any other combination of public-parameters and tweak. The manipulated random functions are manipulated in essentially the same way as in Figure 4. When $\widetilde{\mathbf{Th}}$ is generated, $\mathcal{B}$ runs $\mathcal{A}_2$ on input $(\{(T_i, M_i)\}_{i=1}^p, P)$, giving oracle access to $\widetilde{\mathbf{Th}}$. When $\mathcal{A}_2$ returns (j, M), $\mathcal{B}$ outputs $j \| M$.

Per construction, $f(j \| M) = 1$ whenever (j, M) is a valid SM-TCR solution. The distribution of $\widetilde{\mathbf{Th}}$ and $(\{(T_i, M_i)\}_{i=1}^p, P)$ is exactly that of a SM-TCR challenge. Hence, $\mathcal{B}$ succeeds exactly with $\mathcal{A}$'s success probability in the SM-TCR game. $\mathcal{B}$ makes twice the number of oracle queries as it has to uncompute the oracle results after use.

E PROOFS OF CLAIMS IN SECTION 4.2

Claim 19.

$$\left|\mathrm{Succ}^{GAME.1}(\mathcal{A}) - \mathrm{Succ}^{GAME.0}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}PRF}(\mathbf{PRF}; \xi, q_1).$$

Proof. For any forger $\mathcal{A}$, the difference in success probability between playing in GAME.0 and GAME.1 is bounded by the PRF insecurity of $\mathbf{PRF}$, $\mathrm{InSec}^{\mathrm{PQ\text{-}PRF}}(\mathbf{PRF}; \xi, q_1)$, where $q_1 < 2^{h+1}(kt + \mathtt{len})$ is the number of $\mathbf{PRF}$ outputs used for one SPHINCS$^+$ keypair. Otherwise, we could use $\mathcal{A}$ to break the PRF security of $\mathbf{PRF}$ with a success probability greater $\mathrm{InSec}^{\mathrm{PQ\text{-}PRF}}(\mathbf{PRF}; \xi, q_1)$. For this, we replace $\mathbf{PRF}$ in GAME.0 by the oracle provided by the PRF game and output 1 whenever $\mathcal{A}$ succeeds. The two cases to be distinguished in the PRF game differ by the function implemented by the provided oracle. In one case, the oracle is the real function $\mathbf{PRF}$ keyed with a random secret key. For $\mathcal{A}$, replacing $\mathbf{PRF}$ with this oracle is identical to GAME.0. In the other case, the oracle is a truly random function. Replacing $\mathbf{PRF}$ with this oracle is exactly GAME.1. Given that the addresses used to generate the secret key values of WOTS$^+$ and FORS are all distinct by construction, the outputs of a random function on these addresses leads to independent, uniformly distributed random values. Consequently, the difference of the probabilities that the reduction outputs one in either of the two cases is exactly the difference of the success probabilities of $\mathcal{A}$ in the two games. □

Claim 20.

$$\left|\mathrm{Succ}^{GAME.2}(\mathcal{A}) - \mathrm{Succ}^{GAME.1}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}PRF}\left(\mathbf{PRF_{msg}}; \xi, q_s\right).$$

Proof. The difference in success probability of any adversary $\mathcal{A}$ playing the two games must be bounded by the PRF insecurity of $\mathbf{PRF_{msg}}$, $\mathrm{InSec}^{\mathrm{PQ\text{-}PRF}}\left(\mathbf{PRF_{msg}}; \xi, q_s\right)$, where q_s denotes the number of signing queries made by $\mathcal{A}$. Otherwise, we can construct an oracle machine $\mathcal{M}^{\mathcal{A}}$ which uses $\mathcal{A}$ to break the PRF security of $\mathbf{PRF_{msg}}$. For this $\mathcal{M}^{\mathcal{A}}$ just replaces all calls to $\mathbf{PRF_{msg}}$ by calls to the oracle given in the PRF game and outputs 1 whenever $\mathcal{A}$ succeeds. If the oracle implements $\mathbf{PRF_{msg}}$ for a random key, this is identical to GAME.1. If $\mathbf{PRF_{msg}}$ is a random function, this is identical to GAME.2. □

Claim 21.

$$\left|\mathrm{Succ}^{GAME.3}(\mathcal{A}) - \mathrm{Succ}^{GAME.2}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}ITSR}\left(\mathbf{H_{msg}}; \xi, q_s\right).$$

Proof. The only source for a difference in success probability between these two games are the success cases which got excluded in GAME.3. These success cases are exactly the cases where $\mathcal{A}$ breaks the ITSR security of $\mathbf{H_{msg}}$. Hence, we can build a reduction $\mathcal{M}^{\mathcal{A}}$ which uses $\mathcal{A}$ to break ITSR and it will succeed exactly with the difference in success probabilities between these two games. The reduction $\mathcal{M}^{\mathcal{A}}$ makes use of the ITSR challenge function family to instantiate a SPHINCS$^+$ key pair. Then, for every signature query M_i by $\mathcal{A}$, it uses its oracle $\mathcal{O}$ to obtain $K_i, \mathrm{G}(K_i, M_i)$ instead of computing this itself. Otherwise, signatures are computed using the regular SPHINCS$^+$ algorithms. Note that here we are using the ITSR notation; in SPHINCS$^+$ the function key K is called the randomizer and denoted R. The resulting signatures follow the correct distribution as the function keys K are uniformly random in both cases and the signatures are otherwise computed exactly the same as in GAME.2. When $\mathcal{A}$ outputs a forgery $(M, \mathbf{SIG})$, $\mathcal{M}^{\mathcal{A}}$ extracts the function key K from $\mathbf{SIG}$ and outputs (K, M). The reduction $\mathcal{M}^{\mathcal{A}}$ makes one oracle query per signature query by $\mathcal{A}$, so at most q_s oracle queries in total. By construction we got $\left|\mathrm{Succ}^{\mathrm{GAME.3}}(\mathcal{A}) - \mathrm{Succ}^{\mathrm{GAME.2}}(\mathcal{A})\right| = \mathrm{Succ}^{\mathrm{ITSR}}_{\mathcal{H}, q_s}\left(\mathcal{M}^{\mathcal{A}}\right)$ and so the claim follows. □

Claim 22.

$$\left|\mathrm{Succ}^{GAME.4}(\mathcal{A}) - \mathrm{Succ}^{GAME.3}(\mathcal{A})\right| \leq \mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}(\mathbf{Th}; \xi, q_2).$$

Proof. Similar to above, the only source for a difference in success probability between these two games are the success cases which got excluded in GAME.4. All these success cases are cases where $\mathcal{A}$ breaks the SM-TCR security of $\mathbf{Th}$. Hence, we can build a

reduction $\mathcal{M}^{\mathcal{A}}$ that breaks SM-TCR of **Th**. The reduction $\mathcal{M}^{\mathcal{A}}$ builds the whole SPHINCS+ structure of a key pair (the key pair plus the whole hypertree including all FORS and WOTS key pairs) during set-up using the SM-TCR oracle for **Th** and stores all computed values. Thereby it defines all inputs to **Th** as targets. In total, $\mathcal{M}^{\mathcal{A}}$ makes $q_2 = \left(\sum_{i=0}^{d-1} 2^{ih/d}(2^{h/d}(w \cdot \mathtt{len} + 1) + 2^{h/d} - 1\right) + 2^h k(2t - 1) < 2^{h+2}(w \cdot \mathtt{len} + 2kt)$ queries to its oracle.

When $\mathcal{M}^{\mathcal{A}}$ is done, it obtains the public parameters from the challenger and puts these into the public key together with the generated root. Then it runs $\mathcal{A}$ with this public key as input. $\mathcal{M}^{\mathcal{A}}$ can answer all signature queries and perfectly simulates the EU-CMA game for SPHINCS+.

When $\mathcal{A}$ returns a forgery, $\mathcal{M}^{\mathcal{A}}$ runs verification and compares all computed values to the values it computed during set-up. If $\mathcal{M}^{\mathcal{A}}$ finds a second preimage it outputs it together with its query index (indicating when it was sent to the SM-TCR oracle). We get $\left|\mathrm{Succ}^{\mathrm{GAME.4}}(\mathcal{A}) - \mathrm{Succ}^{\mathrm{GAME.3}}(\mathcal{A})\right| = \mathrm{Succ}^{\text{SM-TCR}}_{\mathbf{Th}, q_2}\left(\mathcal{M}^{\mathcal{A}}\right)$ which implies the claim. □

Claim 23.

$$\mathrm{Succ}^{GAME.4}(\mathcal{A}) \le 3 \cdot \mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}(\mathbf{F}; \xi, q_3) + \mathrm{InSec}^{PQ\text{-}SM\text{-}DSPR}(\mathbf{F}; \xi, q_3).$$

Proof. In GAME.4 we excluded all cases but those where $\mathcal{A}$ gives us a preimage under **F** of a value it learned from one of the queries. The argument is essentially the same as in previous proofs like the original SPHINCS proof [9]: As we already excluded the ITSR case, the FORS signature in a forgery must include the preimage of a FORS leaf node that was not previously revealed to it – which is a preimage under **F**. However, the leaf might be different from the leaf that was used by the signer (or the reduction below), in which case it would not match the statement. By a pigeonhole argument it can be reasoned that this either means that the signature leads to a second-preimage for **Th** or one of the root values that can be derived from the forgery differs from the one the signer used. The former case is exactly the case that got excluded in GAME.4. The latter implies a WOTS+ forgery which in turn either implies that the forgery leads to a second preimage under **F** for a value $\mathcal{A}$ learned from a signature query, or a preimage under **F** for a value $\mathcal{A}$ learned from a signature query as shown in [29]. The former again is what got excluded in GAME.4 and the latter is exactly the case we are interested in.

Now we can apply the technique from [10] to show that we can use $\mathcal{A}$ to either break SM-TCR or SM-DSPR of **F**. We consider two reductions. Reduction $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ targets SM-TCR of **F**. It essentially works like the reduction in the last proof. It uses the SM-TCR oracle for **F** to generate the whole structure of a SPHINCS+ key pair. For this it first computes all outputs of **F**, then obtains the public parameters and afterwards uses those to do the **Th** computations. As above, $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ then runs $\mathcal{A}$. Note that it can answer all signing queries and the generated public key, as well as the generated signatures follow the right distribution. The reduction $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ makes $q_3 = \left(\sum_{i=0}^{d-1} 2^{ih/d}(2^{h/d} w \cdot \mathtt{len})\right) + 2^h kt < 2^{h+1}(kt + w \cdot \mathtt{len})$ queries to its oracle.

When $\mathcal{A}$ outputs a valid forgery, $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ extracts the preimage under **F** which must exist according to the argument above. Let this preimage be x'. Now, $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ also used a value x to compute the image that **F** got inverted on. Let j be the index of the query with message x. $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ outputs the pair (j, x'). If $\mathcal{A}$ fails, $\mathcal{M}^{\mathcal{A}}$ outputs a random entry (j, x_j) from its challenge list.

The second reduction $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ aims at breaking SM-DSPR. Up to the point where the forgery is obtained, $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ does exactly the same as $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$. Given the forgery, $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ compares the given preimage x' to the value x that it used to compute the image that **F** got inverted on. Let j again be the index of the query that was used to compute the inverted image. If the two values are the same, i.e., if $\mathcal{A}$ returned the preimage that $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ already knew, it returns $(0, j)$. In any other case (including the case that $\mathcal{A}$ fails), $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ returns $(1, j)$.

Now the proof proceeds essentially as in [10]. We split the universe of possible events into mutually exclusive events across two dimensions: the number of preimages of $\mathbf{F}_{P,T_j}(x)$, and whether $\mathcal{A}$ succeeds or fails in forging a signature and thereby in finding a preimage. Specifically, define

$$S_i \stackrel{\text{def}}{=} \left[\left|\mathbf{F}^{-1}_{P,T_j}(\mathbf{F}_{P,T_j}(x))\right| = i \wedge \mathbf{F}_{P,T_j}(x') = \mathbf{F}_{P,T_j}(x_j)\right],$$

as the event that there are exactly i preimages and that $\mathcal{A}$ succeeds, and define

$$F_i \stackrel{\text{def}}{=} \left[\left|\mathbf{F}^{-1}_{P,T_j}(\mathbf{F}_{P,T_j}(x_j))\right| = i \wedge \left(\mathbf{F}_{P,T_j}(x') \neq \mathbf{F}_{P,T_j}(x)\right)\right]$$

as the event that there are exactly i preimages and that $\mathcal{A}$ fails. Note that there are only finitely many i for which the events S_i and F_i can occur.

Define s_i and f_i as the probabilities of S_i and F_i respectively. The probability space here includes the random choices of P, and any random choices made inside $\mathcal{A}$.

$\mathcal{A}$'s success probability. By definition, $\mathrm{Succ}^{\mathrm{GAME.4}}(\mathcal{A})$ is the probability that x' is a preimage of $\mathbf{F}_{P,T_j}(x)$; i.e., that $\mathbf{F}_{P,T_j}(x') = \mathbf{F}_{P,T_j}(x)$. This event is the union of the events S_i, so the combined probability is $\mathrm{Succ}^{\mathrm{GAME.4}}(\mathcal{A}) = \sum_i s_i$.

SM-DSPR success probability. By definition $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ outputs the pair (j, b), where $b = (x' \neq x)$ or 1 if $\mathcal{A}$ fails.

Define succ as in the definition of $\mathrm{Adv}^{\text{SM-DSPR}}_{\mathbf{F}, q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}\right)$. Then succ is the probability that $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ is correct, i.e., $b = \mathbf{SP}_{P,T_j}(x)$. There are four cases:

- If the event S_1 occurs, then there is exactly 1 preimage of $\mathbf{F}_{P,T_j}(x)$, so $\mathbf{SP}_{P,T_j}(x) = 0$ by definition of **SP**. Also, $\mathcal{A}$ succeeds: i.e., x' is a preimage of $\mathbf{F}_{P,T_j}(x)$, forcing $x' = x$. Hence $b = 0 = \mathbf{SP}_{P,T_j}(x)$.
- If the event F_1 occurs, then again $\mathbf{SP}_{P,T_j}(x) = 0$, but now $\mathcal{A}$ fails: i.e., $\mathcal{A}$ did not return a valid forgery in the sense of GAME.4. In this case $b = 1 \neq \mathbf{SP}_{P,T_j}(x)$.
- If the event S_i occurs for $i > 1$, then $\mathbf{SP}_{P,T_j}(x) = 1$ and $\mathcal{A}$ succeeds. Hence x' is a preimage of $\mathbf{F}_{P,T_j}(x)$, so $x' = x$ with conditional probability exactly $\frac{1}{i}$ as x is information theoretically hidden from $\mathcal{A}$ within a set of i elements. Hence $b = 1 = \mathbf{SP}_{P,T_j}(x)$ with conditional probability exactly $\frac{i-1}{i}$.

- If the event F_i occurs for $i > 1$, then $\mathbf{SP}_{P,T_j}(x) = 1$ and $\mathcal{A}$ fails. Failure means that $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ outputs $b = 1 = \mathbf{SP}_{P,T_j}(x)$. So $\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}$ is correct.

To summarize, succ $= s_1 + \sum_{i>1} \frac{i-1}{i} s_i + \sum_{i>1} f_i$.

SM-DSPR advantage. Define triv as we did before, in the definition of $\text{Adv}^{\text{SM-DSPR}}_{\mathbf{F},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}\right)$. Then we get $\text{Adv}^{\text{SM-DSPR}}_{\mathbf{F},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}\right) = \max\{0, \text{succ} - \text{triv}\}$.

The analysis of triv is the same as the analysis of succ above, except that we compare $\mathbf{SP}_{P,T_j}(x)$ to 1 instead of comparing it to b. We have $1 = \mathbf{SP}_{P,T_j}(x)$ exactly for the events S_i and F_i with $i > 1$. Hence triv $= \sum_{i>1} s_i + \sum_{i>1} f_i$. Subtract to see that

$$\text{Adv}^{\text{SM-DSPR}}_{\mathbf{F},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}\right) = \max\{0, \text{succ} - \text{triv}\} \geq \text{succ} - \text{triv} = s_1 - \sum_{i>1} \frac{1}{i} s_i.$$

SM-TCR success probability. By definition $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ outputs (j, x'). The SM-TCR success probability $\text{Succ}^{\text{SM-TCR}}_{\mathbf{Th},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}\right)$ is the probability that x' is a second preimage of x under $\mathbf{F}_{P,T_j}$, i.e., that $\mathbf{F}_{P,T_j}(x') = \mathbf{F}_{P,T_j}(x)$ while $x' \neq x$.

Assume that event S_i occurs with $i > 1$. Then x' is a preimage of $\mathbf{F}_{P,T_j}(x)$. Furthermore, $\mathcal{A}$ did not learn x from a previous query, so x is not known to $\mathcal{A}$ except via $\mathbf{F}_{P,T_j}(x)$. There are i preimages, so $x' = x$ with conditional probability exactly $\frac{1}{i}$. Hence $\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}$ succeeds with conditional probability $\frac{i-1}{i}$.

To summarize, $\text{Succ}^{\text{SM-TCR}}_{\mathbf{Th},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}\right) \geq \sum_{i>1} \frac{i-1}{i} s_i$.

Combining the probabilities. We conclude:

$$\begin{aligned}
&\text{Adv}^{\text{SM-DSPR}}_{\mathbf{F},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-DSPR}}\right) + 3\text{Succ}^{\text{SM-TCR}}_{\mathbf{Th},q_3}\left(\mathcal{M}^{\mathcal{A}}_{\text{SM-TCR}}\right) \\
&\quad \geq s_1 - \sum_{i>1} \frac{1}{i} s_i + 3 \sum_{i>1} \frac{i-1}{i} s_i = s_1 + \sum_{i>1} \frac{3i-4}{i} s_i \\
&\quad \geq s_1 + \sum_{i>1} s_i = \text{Succ}^{\text{GAME.4}}(\mathcal{A}).
\end{aligned}$$

□

F INSTANTIATIONS OF HASH FUNCTIONS

In this section we define different signature schemes, which are obtained by instantiating the cryptographic function families of SPHINCS$^+$ with SHA-256, SHAKE256, and Haraka. To instantiate the tweakable hash functions, we present two different constructions. Leading to a total of six instantiations. For the 'robust' instances, we first generate pseudorandom *bitmasks* which are then XORed with the input message. The masked messages are denoted as $M^{\oplus}$. For the 'simple' instances, we take an approach inspired by the LMS proposal for stateful hash-based signatures [36], and omit the bitmasks. We make this difference explicit in the instances defined below. The 'simple' instances are faster as they omit the calls to **PRF** to generate bitmasks. When combined with compressed addresses in the SHA-256 case this can lead to an estimated reduction of the number of compression function calls by a factor of almost 4. In return, this comes at the cost of a security argument that entirely relies on the random oracle model.

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

F.1 SPHINCS$^+$-SHAKE256

For SPHINCS$^+$-SHAKE256 we define

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

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

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

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

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

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

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

F.2 SPHINCS$^+$-SHA-256

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

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

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

$$\begin{aligned}
&\mathbf{F}(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M_1) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M_1^{\oplus}), \\
&\mathbf{H}(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M_1||M_2) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M_1^{\oplus}||M_2^{\oplus}), \\
&\mathbf{Th}_\ell(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M^{\oplus}),
\end{aligned} \tag{5}$$

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

$$\begin{aligned}
&\mathbf{F}(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M_1) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M_1), \\
&\mathbf{H}(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M_1||M_2) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M_1||M_2), \\
&\mathbf{Th}_\ell(\mathbf{PK}.\mathsf{seed}, \mathbf{ADRS}, M) = \\
&\quad \text{SHA-256}(\mathbf{PK}.\mathsf{seed}||\text{toByte}(0, 64 - n/8)||\mathbf{ADRS}^c||M),
\end{aligned} \tag{6}$$

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

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

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

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

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

Shorter Outputs. If a parameter set requires an output length $n < 256$ bits for **F**, **H**, **PRF**, and $\mathbf{PRF_{msg}}$ we take the first n bits of the output and discard the remaining.

F.3 SPHINCS+-Haraka

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

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

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

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

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

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

For a more efficient construction we generate the round constants of Haraka using **PK**.seed.[4] As **PK**.seed is the same for all hash function calls for a given key pair we expand **PK**.seed using HarakaS and use the result for the round constants in all instantiations of Haraka used in SPHINCS+. In total there are 40 128-bit round constants defined by

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

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

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

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

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

[4] This is similar to the ideas used for the MDx-MAC construction [41].

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

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

For $\mathbf{F}$ we pad M_1 and $M_1^{\oplus}$ with zero if $n < 256$. Note that $\mathbf{H}$ and $\mathbf{H_{msg}}$ will always have a different **ADRS** and we therefore do not need any further domain separation.

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

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

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

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

Shorter Outputs. If a parameter set requires an output length $n < 256$ bits for $\mathbf{F}$ and $\mathbf{PRF}$, we take the first n bits of the output and discard the remaining.

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

G PARAMETER-SPACE EXPLORATION

The Python script that we use for parameter-space exploration is given in Listing 1.

H PERFORMANCE OF THE NIST INSTANTIATIONS

For an overview of the performance of all SPHINCS$^+$ instantiations submitted to NIST see Table 4. For details of the parameters of these instances, please refer to [4].

Listing 1 Parameter-exploration script

```
# Set variables in the following three lines
tsec        = 192    # Pr[one attacker hash call works] <= 1/2^tsec
maxsigs     = 2^64   # at most 2^72
maxsigbytes = 64000  # Don't print parameters if signature size is larger
#### Don't edit below this line ####

#### Generic caching layer to save time

import collections
class memoized(object):
  def __init__(self,func):
    self.func = func
    self.cache = {}
    self.__name__ = 'memoized:' + func.__name__
  def __call__(self,*args):
    if not isinstance(args,collections.Hashable):
      return self.func(*args)
    if not args in self.cache:
      self.cache[args] = self.func(*args)
    return self.cache[args]

#### SPHINCS+ analysis

F = RealIntervalField(tsec+100)
sigmalimit = F(2^(-tsec))
donelimit  = 1-sigmalimit/2^20
hashbytes  = tsec/8 # length of hashes in bytes

# Pr[exactly r sigs hit the leaf targeted by this forgery attempt]
@memoized
def qhitprob(leaves,qs,r):
    p = 1/F(leaves)
    return binomial(qs,r)*p^r*(1-p)^(qs-r)

# Pr[FORS forgery given that exactly r sigs hit the leaf] = (1-(1-1/F(2^b))^r)^k
@memoized
def forgeryprob(b,r,k):
    if k == 1: return 1-(1-1/F(2^b))^r
    return forgeryprob(b,r,1)*forgeryprob(b,r,k-1)

# Number of WOTS chains
@memoized
def wotschains(m,w):
    la = ceil(m / log(w,2))
    return la + floor(log(la*(w-1), 2) / log(w,2)) + 1

s = log(maxsigs,2)
for h in range(s-8,s+20):                                   # Iterate over total tree height
    leaves = 2^h
    for b in range(3,24):                                   # Iterate over height of FORS trees
        for k in range(1,64):                               # Iterate over number of FORS trees
            sigma = 0
            r = 1
            done = qhitprob(leaves,maxsigs,0)
            while done < donelimit:
                t = qhitprob(leaves,maxsigs,r)
                sigma += t*forgeryprob(b,r,k)
                if sigma > sigmalimit: break
                done  += t
                r     += 1
            sigma += min(0,1-done)
            if sigma > sigmalimit: continue
            sec = ceil(log(sigma,2))
            for d in range(4,h):                            # Iterate over number of sub-trees
                if h % d == 0 and h <= 64+(h/d):
                    for w in [16,256]:                      # Try different Winternitz parameters
                        wots = wotschains(8*hashbytes,w)
                        sigsize = ((b+1)*k+h+wots*d+1)*hashbytes
                        speed   = k*2^(b+1) + d*(2^(h/d)*(wots*w+1))  # Rough speed estimate based on #hashes
                        if sigsize < maxsigbytes:
                            print h,d,b,k,w,                # SPHINCS+ parameters
                            print sec,                      # FORS forgery probability
                            print sigsize,                  # Sig size in bytes
                            print speed                     # Signing speed estimate (based on #hashes)
```

Table 4: Performance of optimized software for all SPHINCS$^+$ signature instantiations proposed to NIST. As required by the NIST PQC project, all parameter sets support up to 2^{64} signatures under the same key. All cycle counts are the median of 100 runs on a 3.5GHz Intel Xeon E3-1275 V3 (Haswell). Software is compiled with `gcc-5.4 -O3 -march=native -fomit-frame-pointer -flto`. The "sec" column specifies the security level as defined in Section 4.A.5 of [40].

Scheme		Cycles			Bytes		
	sec	keypair	sign	verify	sig	pk	sk
SPHINCS$^+$-SHAKE256-128s-simple	L1	128 154 676	2 041 365 350	3 951 142	8 080	32	64
SPHINCS$^+$-SHAKE256-128s-robust	L1	250 818 474	3 701 426 810	7 615 270	8 080	32	64
SPHINCS$^+$-SHAKE256-128f-simple	L1	4 018 144	131 989 768	9 557 542	16 976	32	64
SPHINCS$^+$-SHAKE256-128f-robust	L1	7 851 034	245 065 142	18 993 432	16 976	32	64
SPHINCS$^+$-SHAKE256-192s-simple	L3	194 000 638	4 378 342 330	5 923 106	17 064	48	96
SPHINCS$^+$-SHAKE256-192s-robust	L3	374 059 710	7 584 715 214	11 398 728	17 064	48	96
SPHINCS$^+$-SHAKE256-192f-simple	L3	6, 079, 376	173 513 530	15 523 074	35 664	48	96
SPHINCS$^+$-SHAKE256-192f-robust	L3	11 695 144	326 736 564	29 729 294	35 664	48	96
SPHINCS$^+$-SHAKE256-256s-simple	L5	253 651 290	3 086 754 562	7 783 684	29 792	64	128
SPHINCS$^+$-SHAKE256-256s-robust	L5	480 242 128	5 551 086 830	15 116 818	29 792	64	128
SPHINCS$^+$-SHAKE256-256f-simple	L5	15 875 308	373 185 700	15 397 090	49 216	64	128
SPHINCS$^+$-SHAKE256-256f-robust	L5	30 041 464	682 683 022	30 727 218	49 216	64	128
SPHINCS$^+$-SHA-256-128s-simple	L1	49 078 104	835 272 076	2 348 916	8 080	32	64
SPHINCS$^+$-SHA-256-128s-robust	L1	94 988 100	1 624 566 118	4 700 588	8 080	32	64
SPHINCS$^+$-SHA-256-128f-simple	L1	1 602 368	51 805 308	5 676 578	16 976	32	64
SPHINCS$^+$-SHA-256-128f-robust	L1	2 978 018	96 974 576	11 401 188	16 976	32	64
SPHINCS$^+$-SHA-256-192s-simple	L3	69 860 954	1 737 629 602	3 662 790	17 064	48	96
SPHINCS$^+$-SHA-256-192s-robust	L3	134 664 612	3 024 929 742	7 784 118	17 064	48	96
SPHINCS$^+$-SHA-256-192f-simple	L3	2 116 010	66 380 214	9 611 814	35 664	48	96
SPHINCS$^+$-SHA-256-192f-robust	L3	4 390 738	133 192 018	19 219 918	35 664	48	96
SPHINCS$^+$-SHA-256-256s-simple	L5	85 946 882	1 121 074 298	4 903 926	29 792	64	128
SPHINCS$^+$-SHA-256-256s-robust	L5	350 260 762	4 064 645 574	13 790 402	29 792	64	128
SPHINCS$^+$-SHA-256-256f-simple	L5	5 298 662	133 374 038	9 408 596	49 216	64	128
SPHINCS$^+$-SHA-256-256f-robust	L5	21 672 826	495 051 104	26 825 462	49 216	64	128
SPHINCS$^+$-Haraka-128s-simple	L1	19 984 598	383 658 068	545 352	8 080	32	64
SPHINCS$^+$-Haraka-128s-robust	L1	25 340 702	526 821 772	829 266	8 080	32	64
SPHINCS$^+$-Haraka-128f-simple	L1	643 370	22 936 196	1 188 352	16 976	32	64
SPHINCS$^+$-Haraka-128f-robust	L1	809 006	30 719 668	1 890 584	16 976	32	64
SPHINCS$^+$-Haraka-192s-simple	L2	29 838 170	830 939 210	764 448	17 064	48	96
SPHINCS$^+$-Haraka-192s-robust	L2	39 650 538	1 312 001 676	1 451 896	17 064	48	96
SPHINCS$^+$-Haraka-192f-simple	L2	956 708	27 551 500	1 906 088	35 664	48	96
SPHINCS$^+$-Haraka-192f-simple	L2	1 260 024	38 911 468	3 482 634	35 664	48	96
SPHINCS$^+$-Haraka-256s-simple	L2	40 094 962	572 899 448	1 091 290	29 792	64	128
SPHINCS$^+$-Haraka-256s-robust	L2	51 961 586	807 399 570	1 799 156	29 792	64	128
SPHINCS$^+$-Haraka-256f-simple	L2	2 528 384	65 363 906	2 037 918	49 216	64	128
SPHINCS$^+$-Haraka-256f-robust	L2	3 268 332	90 442 914	3 351 188	49 216	64	128