SPHINCS+C: Compressing SPHINCS+ With (Almost) No Cost

Andreas Hülsing
TU Eindhoven
andreas@huelsing.net

Mikhail Kudinov
TU Eindhoven
m.kudinov@tue.nl

Eyal Ronen
Tel Aviv University
eyal.ronen@cs.tau.ac.il

Eylon Yogev
Bar-Ilan University
eylon.yogev@biu.ac.il

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179381

***Abstract*—SPHINCS+ [CCS '19] is one of the selected post-quantum digital signature schemes of NIST's post-quantum standardization process. The scheme is a hash-based signature and is considered one of the most secure and robust proposals. The proposal includes a fast (but larger) variant and a small (but slower) variant for each security level. The main problem that might hinder its adoption is its large signature size. Although SPHINCS+ supports a trade-off between signature size and the computational cost of signing, further reducing the signature size (below the small variants) results in a prohibitively high computational cost for the signer.**

This paper presents several novel methods for further compressing the signature size while requiring negligible added computational costs for the signer and further reducing verification time. Moreover, our approach enables a much more efficient trade-off curve between signature size and the computational costs of the signer. In many parameter settings, we achieve small signatures and faster running times simultaneously. For example, for 128-bit (classical) security, the small signature variant of SPHINCS+ is 7856 bytes long, while our variant is only 6304 bytes long: a compression of approximately 20% while still reducing the signer's running time. However, other trade-offs that focus, e.g., on verification speed, are possible.

The main insight behind our scheme is that there are predefined specific subsets of messages for which the WOTS+ and FORS signatures (that SPHINCS+ uses) can be compressed, and generation can be made faster while maintaining the same security guarantees. Although most messages will not come from these subsets, we can search for suitable hashed values to sign. We sign a hash of the message concatenated with a counter that was chosen such that the hashed value is in the subset. The resulting signature is both smaller and faster to sign and verify.

Our schemes are simple to describe and implement. We provide an implementation, a theoretical analysis of speed and security, as well as benchmark results.

Keywords: SPHINCS+; Hash based signatures; Post-quantum security

I. Introduction

Hash-based signatures are among the most secure digital signature scheme proposals today. They are believed to resist quantum computer-aided attacks, and breaking a hash-based signature scheme would imply devastating and unlikely attacks in large areas of cryptography. They have solid and well-understood security guarantees, flexibility in choosing the underlying hash functions, and enjoy fast signature generation and verification times. The major drawback of hash-based signatures, and what is slowing wide deployment, is the signature size, which is large compared to other alternatives. Most other signature schemes are either not post-quantum secure (e.g., discrete-log based), or rely on assumptions that have not been extensively studied and are often prone to new attacks [9]. The only post-quantum secure signature schemes that are efficient in terms of all speeds and sizes are based on structured lattice problems. As digital signatures are a crucial cryptographic tool in practice, putting all eggs in one basket is a dangerous setup. Therefore, reducing the size of hash-based signatures is of the utmost importance.

Hash-based signatures have a long history starting with the one-time signatures (OTS) proposed by Lamport [18] and improved by Winternitz [19]. In 2015, Bernstein et al. presented a stateless hash-based signature scheme called SPHINCS [5]. Their proposal had a significant impact on the area of hash-based signatures. Their scheme combined several known and novel techniques that managed to get a fast digital signature candidate and a reasonably small signature size. The SPHINCS scheme is a practical take on Goldreich's proposal to turn stateful schemes into stateless schemes [12]. Goldreich suggested using a binary authentication tree of one-time signatures, which removed the need to maintain a local state but practically yields prohibitively large signatures. The SPHINCS scheme combines a (hyper) tree of one-time signatures while replacing the one-time signatures in the leaves of the tree with few-time signatures [22]. This modification allowed reducing the size of the tree, resulting in a significantly smaller signature size.

Since this proposal, hash-based signatures have received renewed interest, and various improvements and variations have been suggested [23], [13], [3], [15], [4], [10], [21], [24]. Most notable is the suggested proposal SPHINCS+ [6], which is one of the selected schemes in NIST's post-quantum standardization process for digital signatures. The SPHINCS+ scheme introduces a new few-time signature scheme called FORS, uses WOTS-TW [16] as the one-time signature component, and a new security analysis framework that uses "tweakable hash functions". The SPHINCS+ scheme is generally considered the current state-of-the-art of stateless hash-based signatures. It supports a trade-off between signature size and the computational cost of the signature. SPHINCS+ allows for up to 2^{64} signatures with the same private key, and includes a fast (but larger) variant and a small (but slower) variant for each security level. For example, for 128 bits of classical security (NIST level 1), it has signatures of size ≈ 8KB for the small variant and ≈ 17KB for the fast variant.

DOI 10.1109/SP46215.2023.00146

Unfortunately, further reduction in the signature size is not considered practical due to prohibitively high computational cost for the signer.

A. Our results

In this paper, we propose SPHINCS+C, a stateless hash-based signature scheme based on SPHINCS+, which improves the best-known results for stateless hash-based signatures. We reduce the signature size while maintaining (and in some cases even improving) the speed of signature generation. Furthermore, the security of our scheme and the number of supported signatures follows from the security of the components of the original SPHINCS+ scheme. Our main contributions and new techniques are summarized in the following points:

- Improved one-time signatures (WOTS+C): We introduce a new variant of the Winternitz one-time signature scheme (WOTS), which we refer to as WOTS+C. The scheme reduces the number of chains in WOTS, which reduces the signature size and the running time of the verifier *without increasing the running time of the signer* or the key-generation time. The scheme is described in Section III.
- Improved few-time signatures (FORS+C): We introduce a new variant of the FORS scheme, called FORS+C. As for WOTS, our variant reduces the signature size and (slightly) reduces the verification time while maintaining the same signing and key-generation time. The scheme is described in Section IV.
- Improved stateless hash-based signature scheme (SPHINCS+C): We propose a novel variant of SPHINCS+, called SPHINCS+C that uses WOTS+C and FORS+C. We show how to integrate the new one-time and few-time signatures into SPHINCS+, which allows us to reduce the signature size while maintaining speeds, and enables a more comprehensive range of trade-offs for the whole signature scheme. This is described in Section VI.

We provide a security proof of our scheme, a reference implementation based on the code of SPHINCS+,[1] and propose several parameter sets that we benchmarked. On the one hand, we analyze the impact of our improvement when using the same parameters as proposed in the SPHINCS+ submission. On the other hand, we select parameters that optimize for size and demonstrate how far one can go in terms of compression without sacrificing signing speed. We note that users may want to look for other trade-offs depending on the use-case. Table I provides a comparison for the signature sizes of the SPHINCS+ variants submitted to round 3 of the NIST competition [2], and our proposed variants using the size optimized parameters which maintain a comparable (and sometimes slightly better) signing speed. For example, for 128-bit security, the small signature variant of the SPHINCS+ signature is 7856 bytes long, while our variant is only 6304 bytes long: a compression of approximately 20% while additionally slightly improving signing speed (see benchmark at Table V).

[1]https://github.com/eyalr0/sphincsplusc.

Security	Small Signature Size		Fast Signature Size	
Level	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C
128-bit	7856	6304 (-20%)	17088	14904 (-13%)
192-bit	16224	13776 (-16%)	35664	33016 (-8%)
256-bit	29792	26096 (-13%)	49856	46884 (-6%)

TABLE I: Comparison of signature sizes (in bytes) between SPHINCS+ and SPHINCS+C (our scheme). The table compares all three security levels, and for each it compares the small and fast variants. The reduction percentages in size is shown in the parenthesis. The signature generation times are (approximately) the same in both scheme for all parameter settings.

Paper organization

The organization of the rest of the paper is as follows. In Section II, we give the relevant background on the SPHINCS+ signature scheme and its components. In Section III, we describe WOTS+C, our new variant of WOTS+, along with its theoretical analysis. In Section IV, we describe our FORS+C scheme, along with its theoretical analysis. In Section V, we introduce SPHINCS+C and prove it secure. We then discuss parameter selection and propose concrete parameter sets in Section VI. In Section VII, we describe our implementation and provide a comparison with the SPHINCS+ scheme (in terms of speed and signature size). In Section VIII we provide a general discussion of further considerations that are relevant to our scheme. Finally, Section IX suggests future work.

II. Background

In this section, we provide relevant background on the SPHINCS+ signature scheme, covering its main building blocks. Readers familiar with SPHINCS+ may safely skip this section. we abbreviate $\log_2$ as $\log$ as we only take logarithms base 2.

A. WOTS

The Winternitz one-time signature scheme (WOTS) and its variants (e.g., WOTS+) are one-time signature schemes [14]. The core idea of WOTS (and its variants) is to use len function chains starting from random values. These random values together act as the secret key. The public key consists of the ends of all chains. The signature is computed by mapping the message to one intermediate value of each function chain. In this section, we present the original WOTS scheme. We use this basic WOTS scheme to present our ideas. Later in the paper, we show that our optimization can also be used with more complex variants, including WOTS+ and WOTS-TW.

WOTS has two parameters, n and w; n is the security parameter, and the length of the message; w is the Winternitz parameter, which defines the length of the chains, and is usually set to 4, 16 or 256. Let

$$\mathsf{len}_1 = \left\lceil \frac{n}{\log(w)} \right\rceil \text{ and } \mathsf{len}_2 = \left\lfloor \frac{\log(\mathsf{len}_1 \cdot (w-1))}{\log(w)} \right\rfloor + 1 \ ,$$

and $\mathsf{len} = \mathsf{len}_1 + \mathsf{len}_2$. Then the WOTS scheme is:

$\mathsf{KeyGen}(1^n)$:

1) The secret key is random strings $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{len}})$.

2) The public key is $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}})$, where $\mathsf{pk}_i = F^{w-1}(\mathsf{sk}_i)$.

$\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$:

1) Interpret the message m as a number and transfer it into base w representation: $\mathsf{m} = a_1, \ldots, a_{\mathsf{len}_1}$, where $a_i \in [w]$.
2) Compute a checksum $C = \sum_{i=1}^{\mathsf{len}_1}(w - 1 - a_i)$, represented as a string of len_2 base-w values $C = (C_1, \ldots, C_{\mathsf{len}_2})$.
3) Let $(b_1, \ldots, b_{\mathsf{len}}) = (a_1, \ldots, a_{\mathsf{len}_1}, C_1, \ldots, C_{\mathsf{len}_2})$
4) For $i \in [\mathsf{len}]$ compute $\sigma_i = F^{b_i}(\mathsf{sk}_i)$.
5) Output $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$.

$\mathsf{Verify}(1^n, \mathsf{m}, \sigma, \mathsf{pk})$:

1) Parse σ as $(\sigma_1, \ldots, \sigma_{\mathsf{len}})$.
2) Parse $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}})$.
3) Compute the message and checksum encoding $b_1, \ldots, b_{\mathsf{len}} \in [w]$.
4) Verify that for all $i \in [\mathsf{len}]$, it holds that $\mathsf{pk}_i = F^{w-b_i-1}(\sigma_i)$.

Various variants of WOTS were introduced in the literature, including WOTS+ [15] and WOTS-TW [16]. They mainly differ in the exact manner in which the chains are computed. For example, WOTS+, replaces the simple hash function with a more complex chaining function which involves random masking values r and a family of hash functions f_k. In every iteration, the function first takes the bitwise xor of the intermediate value and bitmask r and evaluates f_k on the result. This results in a tight security analysis using weak (inversion) properties of the hash functions. WOTS-TW is built on tweakable hash functions with different security notions.

B. FORS

One-time signature schemes lose all security after doing more than one signature. In contrast, few-time signatures (FTS) can maintain some level of security even after a few signatures are signed. The security of the scheme (usually) deteriorates with each added signature.

The FORS (Forest of Random Subsets) signature scheme is the few-time signature scheme used in SPHINCS+ [6], introduced as an improvement to HORST [5]. The FORS scheme is defined in terms of integers k and $t = 2^b$, and can be used to sign message digests of $k \cdot b$ bits. The private key contains $k \cdot t$ random n-bit values (for security parameter n). These values are viewed as the leaves of k trees, each with t leaves. A Merkle tree is computed for each tree based on the hashes of secret values, resulting in k roots. The public-key consists of the hash of all these roots together. Given a message digest of $k \cdot b$ bits, the signature algorithm reveals k secret values – one from each tree. Each of these k bit strings of length b is interpreted as the index of a leaf in one of the k FORS trees. The signature contains an authentication path for each leaf. As each index is used in a different tree, FORS solves the problem of weak messages in HORST [3], that is due to a collision of two or more indices in the single tree used by HORST.

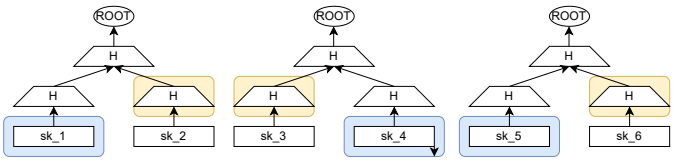

FIG. 1: A toy example of a FORS signature. The image presents a FORS instance for $k = 3, b = 1, t = 2$. It is possible to sign a message of 3-bits length. The image shows a signature for message $m = 010$, blue blocks represent the revealed leafs and orange blocks represent the authentication paths.

An example of the FORS signature is shown in Figure 1.

C. SPHINCS+

The SPHINCS+ [6] scheme is a stateless hash-based signature improving upon the previous version called SPHINCS [5]. To understand SPHINCS+, consider a Merkle tree over the public keys of WOTS+ instances of height h'. Such construction allows us to sign $2^{h'}$ messages. We group the trees in layers. Instead of signing messages, each WOTS+ instance is used to sign a root of another Merkle tree on all but the lowest layer. We call this tree of Merkle trees a hyper-tree. Having d layers in our hyper-tree structure allows us to generate $2^{h' \cdot d}$ signatures on the lowest layer. Since the secret elements for WOTS+ are generated pseudorandomly, each Merkle tree can be generated independently of others. For security reasons and to allow for more message signatures, WOTS+ instances on the bottom layer of the hyper-tree are used to sign public keys of FORS instances. Finally, one of the FORS instances will be used to sign the message. An example of the SPHINCS$^+$ structure is shown in Figure 2.

During key generation, a seed and a PRF-key are chosen, which define the secret key. The former is used to derive all the FORS and WOTS+ secret key values, while the latter is used to deterministically generate randomness for the message hash during signing. To compute the public key, a hash function key called public parameters is chosen that is used for all hash function calls and is the first half of the public key. The second half is the root of the top tree of the hyper-tree, which can be computed by generating just this top tree.

To sign a message, its digest $= (m, i)$ is computed using a (pseudo)-random salt, which prevents collision-finding attacks. The second part i of the digest determines the leaf of the hyper-tree – and thereby the FORS keypair – used to sign the other part m. The FORS public key is signed by the corresponding WOTS+ key pair of the hyper-tree. The Merkle tree containing that WOTS+ key pair is computed to obtain its root. This root is signed using the respective WOTS+ instance from the next layer and the containing tree is computed. This is repeated until the root of the top-level tree is reached, which is an element of the public key. The SPHINCS+ signature includes all the generated FORS and WOTS+ signatures and the corresponding authentication paths in the Merkle trees (these are the sibling nodes for the path from the used leaf to the root).

The verification process tries to recompute the root of the top-level tree from the signature. This is possible as FORS and WOTS+ allow to recompute their public keys given a

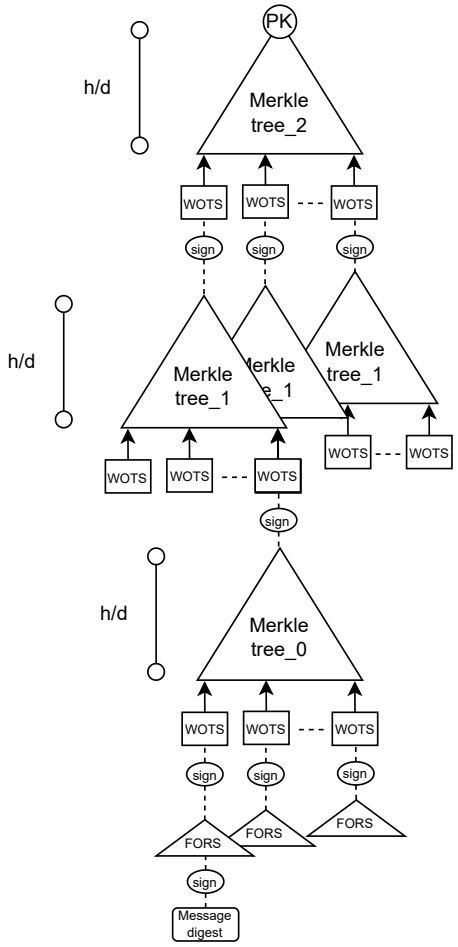

FIG. 2: Example of the SPHINCS$^+$ structure

signature and the signed message. Similarly, given a leaf and an authentication path, Merkle trees allow to compute their root. Together this allows to derive a candidate root value from a SPHINCS+ signature. The signature is accepted if the computed root matches the public key value.

See [6] for more details and security proofs. A slightly more formal description can also be found in Appendix A.

III. WOTS+C

In this section, we present a modification of the WOTS [20] construction (also applicable to other variants). Our modification reduces the size of the signature, *without making the chains longer* and without increasing the running time of verification (indeed, the verification times are actually reduced).

Recall that the WOTS scheme has two parameters n and w. The parameter n is the security parameter and the length of the message. The parameter w is the Winternitz parameter, which defines the length of the chains, and is usually set to 4, 16 or 256. Our scheme introduces two additional parameters $S_{w,n}, z \in \mathbb{N}$. Instead of signing the message m, we sign $d = H(s\|\mathsf{m})$ where s is a short random bit string (a salt). We choose s such that the resulting bit string d satisfies the following property: d is mapped to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$ with:

1) **Fixed sum:** $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$.
2) **Additional zero-chains:** $\forall i \in [z] : a_i = 0$.

The signing algorithm is tasked with finding such a suitable salt s. This is done by enumerating over s until the two conditions hold. We analyze the cost of this process in the next subsection, but first we describe the benefits we gain:

- The first condition means the sum of all words is always equal to a fixed value (that might depend on w and n). As this sum is a fixed parameter of the scheme and can be easily checked by the verifier, we *do not need to sign its value*, which is what is done in WOTS(+). This significantly reduces the size of the signature, as well as the verification time.
- The second condition allows us to further compress the signature size, by not including elements for the first z chains. Again, z is a parameter of the scheme and this condition can be checked by the verifier.

Our scheme can be viewed as a variant of WOTS or WOTS+ (or other variants as well), where we have less chains to sign and verify. Instead of having $\mathsf{len} = \mathsf{len}_1 + \mathsf{len}_2$ chains, we only need to sign and verify $\mathsf{len}_1 - z$ chains (c.f. Figure 3).

Let $\ell = \mathsf{len}_1 - z$. Our scheme is implemented as follows:

$\mathsf{KeyGen}(1^n)$:

1) The secret key is random strings $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$.
2) The public key is $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell)$, where $\mathsf{pk}_i = F^{w-1}(\mathsf{sk}_i)$.

$\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$:

1) Sample a salt $s \in \{0,1\}^n$ at random, until $d = H(s\|\mathsf{m})$ satisfies the above two conditions (if none exists then abort).
2) Compute $d = H(s\|\mathsf{m})$.
3) Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$.
4) For $i \in [\ell]$ compute $\sigma_i = F^{a_i}(\mathsf{sk}_i)$.
5) Output $\sigma = (\sigma_1, \ldots, \sigma_\ell, s)$.

$\mathsf{Verify}(1^n, \mathsf{m}, \sigma, \mathsf{pk})$:

1) Parse σ as $(\sigma_1, \ldots, \sigma_\ell, s)$.
2) Parse $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell)$.
3) Compute $d = H(s\|\mathsf{m})$.
4) Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$.
5) Verify that $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$ and that $\forall i \in [z] : a_i = 0$.
6) Verify that for all $i \in [\ell]$, it holds that $\mathsf{pk}_i = F^{w-a_i-1}(\sigma_i)$.

Note that in the description we have two different hash functions: H and F. H is used to compute a digest of the message m and F is used to construct chains in WOTS. In more complex constructions such as WOTS+ and WOTS-TW instead of iteratively applying a single hash function more complex functions are used. Such functions are usually called chaining functions. Note that our modification is independent of the way these chains are computed. For now we focus on this simple construction with one hash function. Later in the paper we discuss a construction with a more complex chaining function. The analysis of this construction is conceptually the same and differs only in the way we handle the message hash

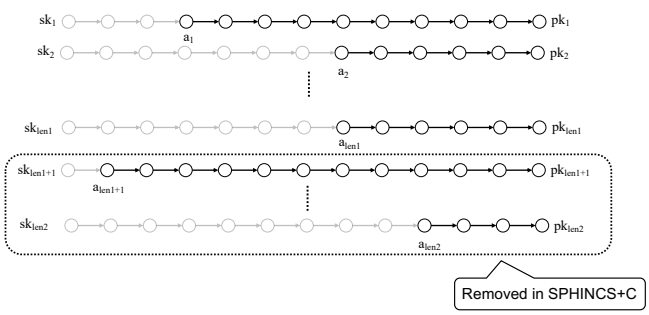

FIG. 3: An illustration of the WOTS chains that are removed in the WOTS+C scheme in the case that $z = 0$. If $z > 0$ then additional chains will be removed.

function (in our example H). Hence, we conclude that our modification works for any WOTS-like scheme that has the chaining structure. It does not depend on how the chain is computed, or which functions are used for chaining. However, we stress that our implementation and experimental results are for the WOTS-TW scheme. The description below uses WOTS solely for ease of presentation.

Once we remove the checksum chains, the running time of the verification becomes much faster (as it does not need to compute hashes for these chains). Signing has an additional step of finding the right counter value, but this is compensated for by the smaller number of chains that have to be computed. Overall, the new scheme allows for smaller signature size, faster verification, and keeping the signature generation time (almost) the same.

Choosing to further compress the signature by also requiring $z > 0$ additional zero-chains can probabilistically increase the overall signature generation time. However, as we discuss in Section VIII-C, it decreases the overall time for SPHINCS+C signature generation in expectation.

A. Security of WOTS

We tightly relate the EU-CMA security of our WOTS+C scheme to that of the WOTS+ [15] scheme and the security of the used hash function. The EU-CMA security is defined using the following experiment (where $\mathsf{Dss}(1^n)$ denotes a signature scheme with security parameter n).

Experiment $\mathsf{Exp}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A})$:
- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^n)$.
- $(\mathsf{m}^*, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$.
- Let $\{(\mathsf{m}_i, \sigma_i)\}_{i\in[q]}$ be the query-answer pairs of $\mathsf{Sign}(\mathsf{sk}, \cdot)$.
- Return 1 iff $\mathsf{Verify}(\mathsf{pk}, \mathsf{m}^*, \sigma^*) = 1$ and $\mathsf{m}^* \notin \{\mathsf{m}_i\}_{i\in[q]}$.

For the success probability of an adversary $\mathcal{A}$ in the above experiment we write

$$\mathsf{Succ}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A}) = \Pr[\mathsf{Exp}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A}) = 1] \ .$$

Using this, we define EU-CMA security the following way.

Definition III.1. *Let $n, t, q \in \mathbb{N}$, $t, q = \mathrm{poly}(n)$, $\mathsf{Dss}(1^n)$ a digital signature scheme. We call $\mathsf{Dss}(1^n)$ EU-CMA-secure, if for any adversary $\mathcal{A}$ with running time t, making at most q queries to* Sign *in the above experiment, the success probability $\mathsf{Succ}^{\text{EU-CMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A})$ is negligible in n.*

An EU-CMA secure one-time signature scheme (OTS) is a $\mathsf{Dss}(1^n)$ that is EU-CMA secure as long as the number of oracle queries of the adversary is limited to one, i.e. $q = 1$. This is the case for WOTS and WOTS+.

B. Security of standalone WOTS+C

Intuitively, the security of WOTS+C follows from the hardness of forging a signature for WOTS and the hardness of finding a colliding message for a message that hashes to some subspace of the image of the hash function. This is the case as the forgery message m^* either is colliding with the message m used in the signature query, or it is not. If it is not, the forgery is a valid WOTS forgery as it is on a fresh message ($H(m) \neq H(m^*)$). If the two messages collide, m^* clearly is a colliding message for m.

To complete the picture, it is important to note that, at least for a random function, an adversary does not gain anything from knowing that it will have to find a collision for a message that hashes into a given subset of the image. This is not surprising as a similar case was already analyzed in [10]. Intuitively, the reason is that we are considering a form of target-collision resistance where the adversary is allowed to choose the message and only afterwards is told under which function key a colliding message has to be found. Hence, putting a restriction on the messages which are considered valid targets rather constrains the adversary than easing its task. We give a full security proof in Appendix B.

C. Complexity analysis of WOTS+C

In this section we provide a mathematical analysis of the time requirements for generating WOTS+C signatures. We count time in number of hash-function calls. As in WOTS-TW, the cost of computing the chains is essentially equal to the length of the chains times the number of the chains. However, we still need to analyze the cost of finding a counter value such that the message digest satisfies the WOTS+C constraints. We assume that Th behaves like a random function and output values are uniformly distributed. Let us recall that to generate a WOTS+C signature for a message m under public seed P and tweak T^* one should find a value i such that $\mathsf{Th}(P, T^*, \mathsf{m}\|i)$ satisfies the following requirements:

1) $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$.
2) $\forall i \in [z] : a_i = 0$,

for parameters $S_{w,n}$, and z. We can replace the second condition with the more general one that simply requires that the last z_b bits are equal to zero. This is useful when $\log w$ does not divide n. This requirement will decrease the probability of hitting a good hash by a factor of 2^{-z_b}.

We will now focus on the $S_{w,n}$ requirement. To calculate the probability one has to compute the number of good hashes. One can see that the number of strings that satisfy our restrictions is equal to the number of ways the value $S_{w,n}$ can be represented as a sum having exactly len terms, where each term is in

$[0, w-1]$ and the order of the terms is important. According to [1] (Section 3, equation E) this number can be computed as:[2]

$$\nu = \sum_{j=0}^{\mathsf{len}} (-1)^j \binom{\mathsf{len}}{j} \binom{(S_{w,n} + \mathsf{len}) - jw - 1}{\mathsf{len} - 1} .$$

Then the probability of hitting a good hash is

$$p_\nu = \frac{\nu}{w^{\mathsf{len}} \cdot 2^{z_b}} .$$

We can view each hash evaluation as an independent biased coin toss. The number of evaluations until the first success follows a geometric distribution, and the expected number of required evaluations is $\frac{1}{p_\nu}$. The probability that it will require more than k hashes is $(1-p_\nu)^k$. A script that can compute these probabilities can be found in the full version of the paper.We experimentally verified the theoretical results by generating a large number of random messages and checking the probability of getting the expected average checksum.

In Section VI-B, we show how to bound the variance in the signature generation time.

IV. FORS+C

In this section, we present the FORS+C scheme, as an improved variant to the FORS few-time signature scheme (see Section II-B for an overview of the original FORS). Recall that the FORS scheme uses k trees, where each tree contains $t = 2^b$ leaves. The signature is a collection of preimages of leaves with their authentication paths, one for each tree. The main idea behind the security of FORS, is that the best strategy of an attacker, is to find a message/salt pair that hashes to a set of leaves that were already revealed as part of the previous signatures. The amount of required hash function evaluations is related to the probability that such event happens for a single message/salt pair.

A. Removing trees from the forest

The first improvement of FORS+C over FORS is to reduce the number of authentication paths while maintaining the same level of security (and the same running time of all algorithms). The idea is to force the hash for the last tree to always open the first leaf (leaf with index 0). This is equivalent to requiring that the last b bits of the digest of the message that is signed by FORS are all zeros. How can this be enforced? We hash a concatenation of a counter i and the message we want to sign. The signer then enumerates over values of i until it finds one where the digest of the message with the counter ends with b bits of zero and signs it.

Once this is enforced, the verifier knows that only signatures that reveal leaf 0 in the last tree are valid, and she can check it directly in the resulting digest value. Thus, the signature itself *does not require the actual authentication path*. This means that we now only need to store the counter in the signature instead of the whole authentication path of the last tree. Moreover, there is no need for the signer to compute the last tree. See an illustration in Figure 4.

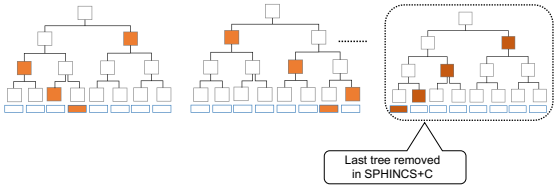

FIG. 4: An illustration of the FORS tree that is removed in the FORS+C scheme.

On average, this will require trying 2^b hash function evaluations before finding a suitable counter and digest. However, as we mentioned, the signing process saves back the 2^b hashes since it does not need to generate the last tree, thus keeping the running time of the signer approximately the same. Moreover, as a result, verification is also (slightly) faster, as it has one less tree to verify. This results in signatures that are strictly better, allowing us to generate smaller signatures that are faster to verify with the same signature time.

To gain further, we can make the last tree bigger before removing it. Equivalently, this means finding an index i where its hash (with the message) begins with b' 0's, for some $b' > b$. This increases the amount of work needed for signing but reduces the signature size. This will be useful when considering different parameter trade-offs (in the overall hyper-tree of the SPHINCS+C scheme).

1) Security: The security analysis is the same as the security analysis of FORS. One can (virtually) imagine that all k trees exist, where in the last tree we always open the same leaf. The probability of the attacker to find a message/salt pair that hashes to k leaves that are all opened is not higher in our scheme. On the contrary, we gain better security guarantees since, in the last tree, the attacker always has only a single leaf open, *regardless* of the number of signatures provided. There is no degradation in security for the tree.

In [6] it is shown that the security analysis of FORS factors into two parts: Either an adversary is able to break the security properties of the hash functions used to compute the leaves and the trees, or the adversary is able to break what is called the interleaved target subset resilience (ITSR) of the message digest function. Informally, ITSR states that it is hard for an adversary that has seen some γ FORS signatures, to find a message that is hashed to leaf indices $x_0, \ldots, x_k$ such that the adversary has seen openings for all the leaves indexed by the x_i in previous signatures.

In more detail, the analysis of FORS shows that the overall success probability of the attacker against ITSR is the product of the success probability for each separate tree. The probability depends only on the number of opened leaves per tree and not on the locations of the opened leaves. Thus, the security of our scheme follows verbatim. Moreover, the analysis remains the same if not all trees are the same size. This means that we can choose a different tree size $t' = 2^{b'}$ for the last tree that is not $t = 2^b$.

[2]We use $(S_{w,n} + \mathsf{len})$ to compensate for the fact that [1] assumes values range of $[1, w]$.

We now describe the security analysis of ITSR. Assume that the adversary has seen γ signatures. We use the analysis from [6] to bound the probability that a new message digest selects FORS positions that are covered by the positions already revealed in previous signatures in a specific tree i. In [6], the probability is denoted by $\mathsf{DarkSide}_\gamma$. The probability that a digest hits all covered positions can be computed as follows. The probability that all the γ messages miss the location of the message digest is $(1-\frac{1}{t})^\gamma$. Thus, as shown in [6], we have that

$$\mathsf{DarkSide}_\gamma = 1 - \left(1 - \frac{1}{t}\right)^\gamma \quad .$$

The probability of being covered in all k trees is:

$$(\mathsf{DarkSide}_\gamma)^k = \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \quad .$$

In the case of FORS+C, the analysis is similar, where we can squeeze out more security by leveraging the fact that for the last tree, all previous γ signatures collide. For the first $k-1$ trees, the probability is the same as above. For the last tree, the probability of choosing the first leaf is merely $1/t'$ (where t' is the size of the last tree), which is independent of γ. Thus, we get that the probability of the digest to be all covered is

$$(\mathsf{DarkSide}_\gamma)^{(k-1)} \cdot \frac{1}{t'} = \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^{(k-1)} \cdot \frac{1}{t'} \quad .$$

Note here that

$$(\mathsf{DarkSide}_\gamma)^{k-1} \cdot \frac{1}{t'} \leq (\mathsf{DarkSide}_\gamma)^k$$

when $t' \geq t$. Hence, we can use the previous ITSR analysis to bound the security of FORS+C.

B. Interleaving trees

In the full version of the paper, we present another variant of FORS, which we call "Interleaving trees". This new technique allows us to enjoy the "path pruning" method to reduce the signature size that was suggested in [4] as "Octopus" while still retaining the simple security analysis of FORS. We employ the same counter technique we use for tree removal to tackle the variable signature size issue from [4] and ensure we always enjoy the average size reduction.

C. Complexity analysis of FORS+C

The number of hash calls required for FORS+C signature generation is the sum of the calls required in the original FORS signature (for the authentication paths we provide) and the cost of finding a suitable counter that removes the last tree. Assuming that our hash function is a random function and the last tree has t' leaves. The probability of hitting the first leaf is $1/t'$; hence the estimated number of tries to get a good hash is t'. The probability of not hitting the needed leaf after k' tries is $(1-1/t')^{k'}$. The script for calculating the probabilities can be found in the full version of the paper.

In Section VI-B, we show how to bound the variance in the signature generation time.

V. SPHINCS+C

At its simplest form, our proposed SPHINCS+C is the SPHINCS+ scheme where we use WOTS+C instead of WOTS+ and FORS+C instead of FORS (later we describe additional optimizations). The usage of FORS+C is straightforward, but WOTS+C in SPHINCS+ requires some work to obtain a tight security proof as in [16]. In this section we discuss WOTS+C in the context of SPHINCS+ and show the security bound for SPHINCS+C.

A. WOTS+C in the context of SPHINCS+

In Section III a standalone version of WOTS+C is described. In the context of SPHINCS+ the messages signed using WOTS-TW are roots of binary hash tress that already have the right length. Therefore, we do not require a hash function to compress the messages. However, to apply the WOTS+C idea, we have to hash the message with a counter until we find a hash that fulfills the WOTS+C requirements. There are several options to do this. For example, one could incorporate a counter into the last hash of the binary hash tree. The least invasive option seems to be to simply hash the binary hash tree root once more with a counter.

As we discussed earlier the general idea behind the security of WOTS+C is based on the security of the underlying WOTS scheme and on the complexity of finding a collision under H. In Appendix B a security proof based on the m-eTCR property of H is given. The complexity of generic attacks against m-eTCR shrinks linearly in the number of targets an attacker gets. When using WOTS+C in SPHINCS+, an attacker gets to see more than 2^{60} WOTS signatures in the worst case. That makes more than 2^{60} target hash values, and consequently a security loss of 60 bits. In [6], the SPHINCS+ team proposed what is called a tweakable hash function (THF, see definition below). Using THFs it is possible to prevent this degradation of security with the number of targets and to give a tight reduction. After defining THFs, we give a construction of the WOTS+C concept with a THF for message hashing. This approach allows to integrate WOTS+C in SPHINCS+ without a degradation of the security level (see Section V-B).

Definition V.1 (Tweakable hash function; [16, Definition 1])**.** *Let $n, m \in \mathbb{N}$, let $\mathcal{P}$ be the public parameters space and let $\mathcal{T}$ be the tweak space. A tweakable hash function (THF) is an efficient function*

$$\mathsf{Th}\colon \mathcal{P} \times \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n,\ MD \leftarrow \mathsf{Th}(P,T,M)$$

mapping an m-bit message M to an n-bit hash value MD using a function key called public parameter $P \in \mathcal{P}$ and a tweak $T \in \mathcal{T}$. For brevity, we write $\mathsf{Th}_{P,T}(M) = \mathsf{Th}(P,T,M)$.

The public parameter is called Seed in the context of WOTS-TW. A Seed is associated with one or several instances of WOTS-TW. To get optimal security, each hash invocation uses a different tweak. We follow the WOTS-TW structure and have a function that creates a unique tweak for each hash invocation. The tweak associated with the j-th function call in the i-th chain is defined as $T_{i,j}$. WOTS+C adds a unique

tweak to compute the hash of the message with the counter, which we denote T^*. In different applications, one would want to use several WOTS instances. In this case, the tweaks that are used in different instances should also be different. To denote this distinction, we follow the approach of [16], [6] and define an addressing system. Tweaks contain two parts: the prefix $\mathbf{ADRS}$ which defines the instance and a suffix that defines the exact position in that instance. For example, we use T^*_{ADRS} to denote the tweak for hashing the message for the WOTS instance that corresponds to the address ADRS, and we require that the hash with the counter has the desired fixed sum. We will denote the tweaks in the chains for an instance corresponding to address $\mathbf{ADRS}$ as $T_{\text{ADRS},i,j}$. If we use only one instance of WOTS+C, we omit the ADRS.

a) Chaining function $c^{j,k}_{\text{ADRS}}(\mathsf{m}, i, \mathsf{Seed})$: The main difference between WOTS-TW and previous variants is how the hashing is performed. This is described by the chaining function. The chaining function takes as inputs a message $\mathsf{m} \in \{0,1\}^n$, the address $\mathbf{ADRS}$ of the instance, iteration counter $k \in \mathbb{N}$, start index $j \in \mathbb{N}$, chain index i, and public parameters Seed. The chaining function then works the following way. In case $k \leq 0$, c returns m. For $k > 0$ we define c recursively as

$$c^{j,k}(\mathsf{m}, i, \mathsf{Seed}) = \mathsf{Th}(\mathsf{Seed}, T_{\text{ADRS},i,j+k-1}, c^{j,k-1}_{\text{ADRS}}(\mathsf{m}, i, \mathsf{Seed})) \ .$$

We assume the existence of context information $\mathcal{C}$ (which usually contains a public seed, and a specific address inside the SPHINCS+ scheme to distinguish the hash invocations). We assume implicitly that all procedures have access to the context information. The space which we enumerate to obtain a good hash is denoted as $\{0,1\}^r$. Assume that the security parameter for our scheme is λ, then the value r we also set to be λ. This requirement allows us to assume that there always exists a counter that satisfies the WOTS+C conditions. See Appendix C for more details. Then, the new scheme is:

$\mathsf{KeyGen}(1^n)$:

1) The secret key is random strings $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$.
2) The public key is $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell)$, where $\mathsf{pk}_i = c^{0,w-1}_{\text{ADRS}}(\mathsf{sk}_i, i, \mathsf{Seed})$.

$\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$:

1) Find $\mathsf{count} \in \{0,1\}^r$ such that $d = \mathsf{Th}(\mathsf{Seed}, T^*_{\text{ADRS}}, \mathsf{m} \| \mathsf{count})$ satisfies the two conditions.
2) Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$.
3) For $i \in [\ell]$ compute $\sigma_i = c^{0,a_i}_{\text{ADRS}}(\mathsf{sk}_i, i, \mathsf{Seed})$.
4) Output $\sigma = (\sigma_1, \ldots, \sigma_\ell, \mathsf{count})$.

$\mathsf{Verify}(\mathsf{m}, \sigma, \mathsf{pk})$:

1) Parse σ as $(\sigma_1, \ldots, \sigma_\ell, \mathsf{count})$.
2) Parse $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell)$.
3) Compute $d = \mathsf{Th}(\mathsf{Seed}, T^*_{\text{ADRS}}, \mathsf{m} \| \mathsf{count})$.
4) Map d to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$.
5) Verify that $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$ and that $\forall i \in [z] : a_i = 0$.
6) Verify that for all $i \in [\ell]$, it holds that $\mathsf{pk}_i = c^{a_i, w-1-a_i}_{\text{ADRS}}(\sigma_i, i, \mathsf{Seed})$.

Security of WOTS+C in the EU-naCMA model. Previously we showed EU-CMA security of the standalone WOTS+C scheme. In the SPHINCS+ construction, WOTS is used to sign messages that are generated by the honest user and not by the adversary (roots of binary trees). Thus, it was observed in [16] that it is sufficient to prove a weaker, non-adaptive notion of security for WOTS, called existential unforgeability under non-adaptive chosen message attacks (EU-naCMA) where the adversary receives the public key *only after* it made its signature query. The formal definition is given in Appendix C. In [16] the tweakable WOTS (WOTS-TW) variant used in SPHINCS+ is proven secure under this notion.

It is straightforward to extend our above proof to the notion of EU-naCMA and base it also on the EU-naCMA notion of WOTS(-TW). In this case, the adversary does not expect the public key before picking the signature query. Also, our reduction only needs the public key when the signature query is answered. Note that the reduction in Appendix B does not depend on how the chaining function is calculated. The chaining function can be implemented using a keyed hash function, a tweakable hash function, or any other function. The security reduction is dependent on the positions in the chains and not on the way these are calculated. So the switch to tweakable hash functions does not affect the result.

Hence, the only thing left to do to obtain a proof of security for WOTS+C with tweakable hash functions is to handle the modified message hash. Towards this end, we introduce a security definition for a tweakable hash function which we call special target collision resistance (S-TCR(Prop)). This security definition is parameterized by some boolean predicate Prop. The idea behind this notion is that even if an adversary is promised to only receive targets for which the images satisfy Prop, it should still be hard to find collisions for these targets.

In the security reduction we follow the ideas from [16] and make use of a collection of tweakable hash functions which we call Th_λ, which means the adversary has access to the Th_λ oracle. The oracle is used with a challenger for some security notion of a THF. $\mathsf{Th}_\lambda(P, \cdot, \cdot)$ accepts tweaks and messages of arbitrary length and computes a corresponding tweakable hash function on the provided inputs. It is important to note that Th_λ shares the public parameter with the challenger. The main purpose of this oracle is to prepare for a challenge query. So the natural restriction we make is that queries to Th_λ should use different tweaks from the ones that are used for challenge queries.

A more detailed discussion and a full proof of security can be found in Appendix C.

B. SPHINCS+C security

The SPHINCS$^+$ proof proceeds in a sequence of game hops where the difference between any two of those is bounded by the security of one of the building blocks (e.g., security of FORS, security of WOTS-TW, security of the tree hashing, or security of PRF). Hence, to argue security of SPHINCS+C, we have to give new bounds for the security of the building blocks that we changed (WOTS-TW), with respect to the right security notion (d-EU-naCMA) – note that for FORS+C we

showed above that the security bound does not change despite our modifications.

For WOTS+C, we show EU-naCMA security in Appendix C. The notion required for SPHINCS$^+$ is a multi-user security version of it called d-EU-naCMA introduced in [16] (see Appendix D). We give a proof sketch for the d-EU-naCMA security of WOTS+C in the full paper together with an exact bound. The below theorem is then obtained by replacing the terms coming from the d-EU-naCMA bound of WOTS-TW by those of the bound for WOTS+C. Namely, a single term for the S-TCR(+C) security of the tweakable hash function is added (see Appendix C for the definition). This is a variant of target collision resistance (more precisely the SM-TCR property used everywhere in SPHINCS+) where the adversary is tasked to find a target collision for hash values that have a certain form (here, they fulfill the "+ check", i.e., their base-w representation sums up to $S_{w,n}$ and the first z values are 0). For this the adversary gets access to an oracle that produces hashes of the right form, defining the targets. In the full paper we show that generic attacks against S-TCR(+C) have the same complexity as those against SM-TCR.

Combining the results of this paper and the result from [16] we get the following bound:

Theorem V.2. *For parameters n, w, h, d, m, t, k as described in [6] and $l = \mathsf{len}_1 + \mathsf{len}_2$ the following security bound can be obtained for SPHINCS+C:*

$$
\begin{aligned}
&\mathrm{InSec}^{\mathrm{EU-CMA}}(\text{SPHINCS+C}; \xi, q_s) \leq \\
&\quad \mathrm{InSec}^{\text{PRF}}(\mathbf{PRF}, \xi, q_1) + \mathrm{InSec}^{\text{PRF}}(\mathbf{PRF_{msg}}, \xi, q_s) \\
&+ \mathrm{InSec}^{\text{ITSR}}(\mathbf{H_{msg}}, \xi, q_s) + w \cdot \mathrm{InSec}^{\text{SM-UD}}(\mathbf{F} \in \mathsf{Th}_\lambda; \xi, q_2) \\
&+ \mathrm{InSec}^{\text{SM-TCR}}(\mathbf{F} \in \mathsf{Th}_\lambda; \xi, q_3 + q_7) + \mathrm{InSec}^{\text{SM-PRE}}(\mathbf{F} \in \mathsf{Th}_\lambda; \xi, q_2) \\
&+ \mathrm{InSec}^{\text{SM-TCR}}(\mathbf{H} \in \mathsf{Th}_\lambda; \xi, q_4) + \mathrm{InSec}^{\text{SM-TCR}}(\mathsf{Th}_k \in \mathsf{Th}_\lambda; \xi, q_5) \\
&+ \mathrm{InSec}^{\text{SM-TCR}}(\mathsf{Th}_l \in \mathsf{Th}_\lambda; \xi, q_6) + 3 \cdot \mathrm{InSec}^{\text{SM-TCR}}(\mathbf{F} \in \mathsf{Th}_\lambda; \xi, q_8) \\
&+ \mathrm{InSec}^{\text{SM-DSPR}}(\mathbf{F} \in \mathsf{Th}_\lambda; \xi, q_8) + \mathsf{InSec}^{\text{S-TCR(+C)}}(\mathsf{Th}^{+C} \in \mathsf{Th}_\lambda, \xi, q_6)
\end{aligned}
$$

where $q_1 < 2^{h+1}(kt + l)$, $q_2 < 2^{h+1} \cdot l$, $q_3 < 2^{h+1} \cdot l \cdot w$, $q_4 < 2^{h+1}k \cdot 2t$, $q_5 < 2^h$, $q_6 < 2^{h+1}$, $q_7 < 2^{h+1}kt$, $q_8 < 2^h \cdot kt$ and q_s denotes the number of signing queries made by $\mathcal{A}$.

Recall, the ITSR property refers to the security of FORS as explained above. The PRF property refers to the standard indistinguishability of a PRF from a random function. Moreover, the bound contains several terms that refer to properties of tweakable hash functions (THF). All these THF properties are single-function, multi-target (SM) properties, i.e., the adversary receives or defines multiple targets for the same public parameters but with different tweaks. The PRE(image finding) property refers to the ability of an adversary to find a preimage for given images. For Target Collision Resistance (TCR) the adversary is asked to find a collision for targets it defines before it knows the public parameters. The Undetectability (UD) notion represents the ability of an adversary to distinguish outputs of a tweakable hash function on random inputs from random strings. Decisional Second Preimage Resistance (DSPR) tasks the adversary with detecting an input for a THF that does not have a second preimage. The formal definitions of these properties can be found in Appendix D. The used functions are defined as follows:

$\mathbf{F} := \mathsf{Th}_1 : \mathcal{P} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$;
$\mathbf{H} := \mathsf{Th}_2 : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{2n} \to \{0,1\}^n$;
$\mathsf{Th}_l : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{ln} \to \{0,1\}^n$;
$\mathsf{Th}_k : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{kn} \to \{0,1\}^n$;
$\mathbf{PRF} : \{0,1\}^n \times \{0,1\}^{256} \to \{0,1\}^n$;
$\mathbf{PRF_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$;
$\mathbf{H_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^m$.
$\mathsf{Th}^{+C} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^n \times \{0,1\}^r \to \{0,1\}^n$

Lastly the possibility of a denial-of-service attack is worth considering. One could imagine an adversary submitting a message to the signing oracle which requires a huge amount of time to sign. This case is eliminated by the use of a (pseudo)-random salt in the message digest computation. This salt is a part of the original SPHINCS+ design and hence does not introduce new complexities.

VI. Parameter sets for SPHINCS+C

Recall, that for each security level, SPHINCS+ provides two instantiations or sets of parameters that give us a trade-off between faster and smaller signatures. This allows us to sign in settings where speed is crucial with one set (e.g., when the scheme is implemented in Javascript to run in the browser for email security), and to use the other set when signing speed is not that much of an issue or if the size of signatures is the limiting factor. Following this rationale, we also provide two such options, while using the fact that SPHINCS+C gives us a better trade-off curve between signature size and the running time of the signer.

For our concrete parameter choices, we searched for parameters that will minimize the signature size while maintaining the same signing speed, security level, and number of supported signatures (2^{64}) as in the original variants proposed in the NIST submission of SPHINCS+ [2]. Note that this is an arbitrary choice to allow for a simple comparison with SPHINCS+. In Section VIII, we discuss the effect of the parameters on different optimizations, such as minimizing verification time or reducing the variance in signing speed.

A. Parameters search

To search for the best parameters, we follow the example of SPHINCS+. We edited the latest sage script published by the SPHINCS+ team[3] and modified it to support SPHINCS+C.

We recall the parameters notation from SPHINCS+:

n: the security parameter in bytes.

w: the Winternitz parameter.

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

d: the number of layers in the hypertree.

k: the number of trees in FORS.

t: the number of leaves of a FORS tree.

For our scheme we add a parameter:

t': the number of leaves of the extra FORS tree we remove.

[3]http://sphincs.org/data/spx_parameter_exploration.sage

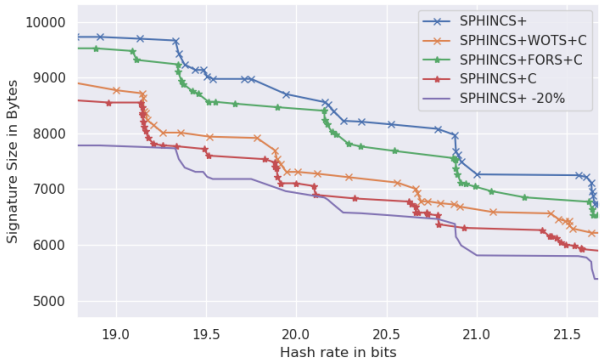

FIG. 5: Hash rate (computational cost) to signature size in bytes trade-off curves for SPHINCS+ and our various compression options for (WOTS+C, FORS+C, and the full SPHINCS+C).

Figure 5 shows part of the trade-off curve between the computational cost (approximated by the number of calls to the hash function) and the signature size. It can be seen that at times we are able to get more than a 20% improvement in size over the original SPHINCS+ scheme for the same computational cost. As expected, most of our gain comes from the WOTS+C compression, but adding FORS+C for the full SPHINCS+C still gives us a significant improvement. The script used to find the trade-off curve can be found in in the full version of the paper. To simplify the script and implementation, we did not incorporate the relatively complex interleaving trees optimization.

In Table II we provide a summary of the concert parameters of our proposed variants. We follow the approach of [2] and propose one size-optimized (ending on "s" for "small") and one speed-optimized (ending on "f" for "fast") parameter set for each security level. For each variant, the signature generation time is similar to its counterpart in the SPHINCS+ NIST submission, while resulting in a smaller signature size.

B. Bounding the signature generation time

Another related concern is that some signatures might take longer to run than the average. In an unfortunate event, one of the WOTS+C or FORS+C signatures will take longer than expected to find a suitable counter. We will now show how to find the probability that the signature generation time takes more than a factor f of the expected running time as a function of the specific parameters of the scheme.

As shown in Section III-C, if the probability of finding a good counter in a hash evaluation of the WOTS+C signature is p_ν, then the expected number of evaluations is $1/p_\nu$, and the probability that finding a good counter takes more than k hash evaluation is $(1-p_\nu)^k$. If we want to find the number of hash evaluations k such that we will only need more than k evaluation with probability p for some small probability p (e.g., $p = 2^{-32}$), we get:

$$k = \frac{\log(p)}{\log(1-p_\nu)}$$

However, in SPHINCS+C we have d WOTS+C signatures. We want to find k_d, the number of hash evaluations such that we will only need more than k_d evaluations to find all d good counters with probability p. We can provide the trivial upper bound $k_d < d \cdot k$, or to find k_d by using the exact calculation of $p_d = 1-p$, the probability of success after k_d hash evaluation:

$$p_d = \sum_{i=d-1}^{k_d} \binom{i}{d-1} \cdot p_\nu^d \cdot (1-p_\nu)^{i+1-d}$$

As shown in Section IV-C, the probability of finding a good counter is $1/t'$ where t' is the number of leaves in the tree we remove. As we did for WOTS+C, we can find k', the number of hash evaluations such that we will only need more than k' evaluations with probability p to find a good counter for the FORS+C signature:

$$k' = \frac{\log(p)}{\log(1-1/t')}$$

We can upper bound k_{all}, such we will only need more than k_{all} evaluations to find all good counters in the SPHINCS+C scheme with probability p by $k_{all} < k_d + k'$. Now we can find the factor $f(p)$, such that only probability p the total number of hash evaluations required to generate the full SPHINCS+C signature will be more than $f(p)$ times the expected number.

Table III shows the results for our proposed SPHINCS+C parameter sets. We note that the dominating factor here is t', the size of the tree we remove at the FORS+C signature. If we want less variance in the signature generation time, we can simply search for parameter sets with smaller values of t' (at the cost of slightly larger signatures).

Even with our current parameter set choices, taking the parameter set with the largest running time variance, the probability that the signature generation time will take more than 5 times the expected time is less than 2^{-32}.

VII. IMPLEMENTATION

We based our implementation on the latest official version of the SPHINCS+ code.[4] We modified the original code to add our optimizations and to allow for performance comparison with the original version. We will make the full code available with the publication of this paper.

A. Implementing FORS+C

To implement FORS+C we simply added an extra 4 bytes counter value at the end of the first hashing of the message to be signed. We try different values of the counter, until the first t' bits of the resulting messages to be signed by FORS+C are zero. We store the counter value that we found as part of the signature. This means that the verifier can simply read the counter value and check that the resulting bits are zero, instead of searching for the counter value again. If the resulting bits are not zero, the validation fails.

The counter value can be replaced with a different one that results in zero bits (the whole computation only requires public

[4]https://github.com/sphincs/sphincsplus/

	n	h	d	log(t)	k	w	log(t')	bitsec	sig bytes
SPHINCS+C-128s	16	66	11	13	9	128	18	128	6304 (−20%)
SPHINCS+C-128f	16	63	21	9	19	16	8	128	14904 (−13%)
SPHINCS+C-192s	24	66	11	15	13	128	12	192	13776 (−16%)
SPHINCS+C-192f	24	63	21	9	30	16	13	192	33016 (−8%)
SPHINCS+C-256s	32	66	11	14	19	64	19	256	26096 (−13%)
SPHINCS+C-256f	32	64	16	10	34	16	10	256	46884 (−6%)

TABLE II: Example parameter sets for SPHINCS+C targeting different security levels and different tradeoffs between size and speed. The signer running time for each variant is better than the ones in the SPHINCS+ NIST submission, and the size reduction in percentages is shown in the parenthesis. We note that k described the number of FORS+C trees authentication paths included in the signature, not including the last tree of size t' that is signed implicitly by the zero bits in the signature.

	expected		$f(p)$ for probability							
	hash calls	$\log(t')$	2^{-8}	2^{-16}	2^{-24}	2^{-32}	2^{-40}	2^{-48}	2^{-56}	2^{-64}
SPHINCS+C-128s	$2^{20.9}$	18	1.6	2.3	3.1	3.8	4.5	5.3	6.0	6.7
SPHINCS+C-128f	$2^{16.7}$	8	1.0	1.0	1.1	1.1	1.1	1.1	1.1	1.1
SPHINCS+C-192s	$2^{21.7}$	12	1.0	1.0	1.1	1.1	1.1	1.1	1.1	1.1
SPHINCS+C-192f	$2^{17.4}$	13	1.2	1.5	1.8	2.0	2.3	2.6	2.9	3.1
SPHINCS+C-256s	$2^{21.5}$	19	1.8	2.8	3.7	4.7	5.7	6.6	7.6	8.6
SPHINCS+C-256f	$2^{18.4}$	10	1.0	1.0	1.1	1.1	1.1	1.1	1.1	1.2

TABLE III: Factor $f(p)$, such that only with probability p the total number of hash calls required to generate the full SPHINCS+C signature will be more than $f(p)$ times the expected number.

parameters). However, we use SPHINCS+C to sign a message that is simply the concatenation of the counter and the original message, and forging either the counter or the message is as hard as forging the message in the original SPHINCS+ scheme.

B. Implementing WOTS+C

Implementing WOTS+C is a bit more involved and is tailored to the implementation of SPHINCS+, and its "tweaked" hash functions. To minimize the modification to the existing code and maintain the size of the input to the hash functions (larger input may reduce performance), we encode counters for WOTS+C inside the address structure used in SPHINCS+. Recall that in SPHINCS+, each "tweaked" hash call includes a unique "address" to make each call in a virtual tree structure independent of each other. We refer the reader to the SPHINCS+ [6] paper for more details on how tweaked hash function and the address structures are defined and implemented.

In each WOTS+C in the virtual tree, we need to hash the message we want to sign (the root of a Merkle tree) together with a counter. As we want the hashes of the different WOTS+C signatures in the tree to be independent, we use an address structure that is unique for each WOTS+C signature. We add a new address structure "WOTS+C message compression address" with `type` values 7 to separate it from all other address types. It is similar to the WOTS+ public key compression address in that it is unique for each WOTS+C signature, but we also add a 4-bytes counter. As in the WOTS+ public key compression address, the resulting structure has 32-bit layer address field, 72-bit tree address field, 32-bit type field (with value 7), and 32-bit key pair address. To do this, we add a 32-bit counter field, and the remaining 64-bits are padded with zeros.

To save computation time, the unique bitmask used in the tweaked hash function is generated once with a counter value of zero. After that we try incremented values of the counter, until the resulting digest of the addresses concatenated with the masked message have the required checksum value. Again we refer the reader to the SPHINCS+ paper for more details on how the bitmask is generated and used.

Similarly to what we did in FORS+C, the counter value for each WOTS+C signature is stored as part of the signature to speed up the running time of the verifier. The verifier reads the counter value and validates that the checksum is correct. If not, the public key of the WOTS+C signature is set to all zeros. This will result in a generation of an incorrect root of the Merkle tree and, in the end, generation of an incorrect public root of the SPHINCS+C signature. As the calculated root is compared with the public key, the validation will fail.

C. Benchmark

The script we used for searching parameters gives us only an approximation of the running time of the signer. To test the actual running time, the official code of SPHINCS+ includes a framework for benchmarking the code. We use this framework to benchmark both the original SPHINCS+ code with the parameters sets submitted to NIST, and our modified SPHINCS+C code with the parameters sets we chose.

The tests were run on a Intel(R) Core(TM) i7-8550U CPU 1.80GHz with 16GB of RAM, running Ubuntu 20.04.4 LTS. The measurements were done using the RDTSC command with `constant_tsc` assuring that the command measures the passage of time on a single-threaded code. SPHINCS+ supports 3 different underlining hash constructions, we chose to benchmark the reference code for the "robust" SHAKE [11] variant based on the Keccak permutation [8]. Our main goal is to compare the running time of SPHINCS+C with SPHINCS+ [6]. For each parameter set we ran the key generation,

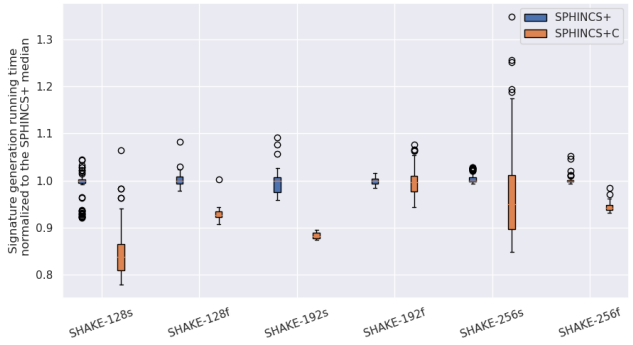

FIG. 6: Comparison of variability in signature generation time between SPHINCS+ and SPHINCS+C variants. The results of each variant include 100 repeated runs of randomized signatures and are normalized to the median of the SPHINCS+ variant.

randomized signature generation and verification procedure, for 100 times.

First, we show how our optimization can improve both the running time and signature size. We compared the original SPHINCS+ variants with the "compressed" versions. The "compressed" variants use the same parameters as in the original SPHINCS+ NIST submission but use WOTS+C to remove the checksum and FORS+C to remove the last tree in the signature. Table IV shows the results of the benchmark comparison. It shows that while reducing the signature sizes by as much as 7%, the optimization also reduces the run time for all variants for key generation, signature generation, and verification.

Table V shows the average results of our second benchmark comparing the original SPHINCS+ variants with our new SPHINCS+C. In all parameter regimes, our new SPHINCS+C variant has smaller signature size compared to its corresponding SPHINCS+ variant, while the signature generation time is approximately the same. Note that although we increase the verification time for our new "small" variants, it is still more than two orders of magnitude faster than the signature generation time.

We also experimentally benchmarked the run-time variability of our SPHINCS+C implementations compared to the original SPHINCS+ over 100 measurements. As the results in Figure 6 show, although the variability in the signature generation time of the SPHINCS+C variants is larger than the SPHINCS+ variants, it is not very big. In all our experiments, the longest signature generation time was at most 40% slower than the median of the SPHINCS+ variant. Most use cases can handle a small number of somewhat slower signatures. We note that at the same time, the average values that determine the total signature generation throughput are very similar (see Table V).

VIII. DISCUSSION

We discuss several properties and trade-offs of our schemes that are somewhat different then in SPHINCS+.

A. Variance in signature generation time

The signing speed is not constant due to the search for good counters. In Section VI-B, we show how to bound the generation time and show that for some of our parameter sets, the variance is negligible, whereas, for others, it is a small factor. The variance is mostly determined by the ratio between the expected time to find a good counter and the total signing speed. This means that to get negligible variance, we need to use slightly smaller values for t' at the cost of a slight increase in signature size. Note that due to the fact that we have many WOTS+C signatures, their contribution to the variance is negligible.

Due to a maybe unlucky naming when coining the term "constant time", one may think that the variable signing speed may enable side-channel attacks. However, constant-time refers to the independence of running time and secret inputs. A variance in speed caused by public values does not cause any vulnerability. In our case, the variance only depends on the message and public values that are revealed as part of the signature (i.e., the public key and public roots of the Merkle trees) and is completely independent of any secret values. Consequently, it can also not leak information about those.

B. Signature verification time

The "small" variants that we proposed for SPHINCS+C have a longer verification time. This is because we optimize for signature size and therefore use large values for w. The SPHINCS+ scheme only supports w values of 16 and 256 as the resulting encoding is of 4 and 8 bits per chain. Otherwise, the last chain will not be "wasteful" and will encode less than $\log(w)$. In practice, all the parameter sets only used $w = 16$.

In WOTS+C, we support a wider range of w values, and some of our variants also use w values of 64 and 128. This is because we can simply zero out the last "partial" chain. This allows for smaller signatures at the cost of longer verification. To optimize for shorter signature verification time, we can use smaller values of w. As a further optimization, we can use different values of w for different WOTS instances of the scheme (e.g., a combination of $w = 16$ and $w = 64$). As the value of w is essentially a trade-off between signature size and running time, this will give us more options on the curve.

C. WOTS+C public key vs. signature generation time trade-off

In WOTS+C, merely removing the checksum keeps the signature run time approximately the same while reducing the signature size and verification time. In addition, it also reduces the key generation time. Trying to find additional zero chains will increase the overall run time of the signer. However, in SPHINCS+ and SPHINCS+C we calculate Merkle trees of WOTS+C signatures. For each tree in the signature, we need to generate only one signature but a large number of public keys for all the other WOTS+C signatures in the tree. This means that in SPHINCS+C, there is a tradeoff between the WOTS+C public key and the signature generation times. In some cases, it may be better to try and find additional zero chains, as this will *decrease* the total signature time of SPHINCS+C while also reducing the signature size.

	Key Generation		Signature		Verification		Size	
	SPHINCS+	Compressed	SPHINCS+	Compressed	SPHINCS+	Compressed	SPHINCS+	Compressed
SHAKE-128s	382.2	344.3 (−10%)	2860.0	2630.3 (−8%)	2.6	2.6 (−1%)	7856	7344 (−7%)
SHAKE-128f	5.7	5.1 (−11%)	135.8	123.1 (−9%)	8.7	7.5 (−13%)	17088	16012 (−7%)
SHAKE-192s	566.1	509.8 (−10%)	4839.0	4388.8 (−9%)	4.5	3.9 (−12%)	16224	15392 (−6%)
SHAKE-192f	8.0	7.1 (−12%)	201.7	184.2 (−9%)	11.6	10.3 (−12%)	35664	33956 (−5%)
SHAKE-256s	345.7	343.4 (−1%)	4012.3	3903.6 (−3%)	6.0	6.0 (−1%)	29792	28580 (−5%)
SHAKE-256f	23.5	20.4 (−13%)	455.8	404.4 (−11%)	12.7	11.2 (−12%)	49856	47976 (−4%)

TABLE IV: Comparison of average running time in milliseconds between the original SPHINCS+ NIST submission and the "compressed" version for the SHAKE robust variants. The compressed versions use the *same* parameter sets but apply WOTS+C to remove the checksum and FORS+C to remove the last FORS tree.

	Key Generation		Signature		Verification		Size	
	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C
SHAKE-128s	382.2	180.7 (−53%)	2860.0	2438.4 (−15%)	2.6	16.3 (+518%)	7856	6304 (−20%)
SHAKE-128f	5.7	4.6 (−20%)	135.8	125.9 (−7%)	8.7	6.4 (−27%)	17088	14904 (−13%)
SHAKE-192s	566.1	265.5 (−53%)	4839.0	4295.7 (−11%)	4.5	23.8 (+429%)	16224	13776 (−16%)
SHAKE-192f	8.0	6.8 (−15%)	201.7	201.0 (−0%)	11.6	9.9 (−15%)	35664	33016 (−8%)
SHAKE-256s	345.7	228.9 (−34%)	4012.3	3888.6 (−3%)	6.0	19.1 (+218%)	29792	26096 (−13%)
SHAKE-256f	23.5	20.0 (−15%)	455.8	429.4 (−6%)	12.7	10.5 (−18%)	49856	46884 (−6%)

TABLE V: Comparison of average running time in milliseconds between the original SPHINCS+ NIST submission and our new SPHINCS+C for the different variants. Here the used SPHINCS+C parameters *differ* from those of SPHINCS+ and are chosen with a focus on size reduction.

D. Signature time vs. signature size and verification time tradeoff

Our scheme allows a server to create smaller signatures that are also faster to verify with the cost of increasing the signing time. This trade-off can be beneficial for CAs that sign a relatively small number of certificates but have their signatures included in a very large number of certificates and verified for many connections. For these servers, it may be possible to find other parameter sets even with a very large signature generation time.

As mentioned in [4], a signing server can batch together several signatures. This is done by first calculating a Merkle tree on all of the batched signatures and then signing the root of the Merkle tree. Moreover, the current parameter sets can support up to 2^{64} signatures. As batching (and longer signature generation time) can reduce the total number of possible signatures, it might also be possible to use smaller tree sizes that will lead to smaller signatures.

In many use cases (e.g., CAs, Certificates for IoT devices) where we want to have smaller signatures and may be willing to pay the price of extra computational cost for the signer or a smaller number of possible signatures. Exploring the potential trade-offs and parameter sets for these use cases may help to facilitate the deployment of hash-based signature schemes.

E. Constant verification time

Similar to [21], [24] our WOTS+C variant ensures a constant verification time. The number of hash calls required by the verifier is determined by the checksum. As in WOTS+C signatures the value of the checksum is always the same, the verification time is constant. We note that the counters for both WOTS+C and FORS+C are stored inside the signature and provided to the verifier. This means the running time for the verifier is constant for a fixed set of parameters.

IX. Future work

In this work, we explored new trade-offs opened up by our optimizations. We were able to reduce up to 20% of the signature size, but there are still opportunities to reduce the remaining 80%. We believe there is room for further exploration, which can exploit additional optimizations we described but did not implement. For example, we did not implement the interleaved trees, but this could be useful in some settings of parameters. Furthermore, one can look at other working points (e.g., higher signature generation time, a smaller number of supported signatures) and find novel approaches for further compression and improving the computational complexity.

As we believe that the main factor limiting the wide deployment of hash-based signatures is the signature size, any advance in this direction can have a big impact on the practicality of these schemes.

For future work, we present two optimization directions. Although in our initial analysis, their contribution to our new variants is not very significant, they may be useful for other parameter sets and might be improved upon in future work.

A. Small trees of FORS+C

In the current SPHINCS+ design, the bottom layer of the hyper-tree signs the root of a FORS signature. This means that the total number of FORS signature is equal to the number of leaves in the tree. Suppose we want to support a larger number of signatures (for increased security or small FORS signature size). In that case, we need to make the tree larger (which will either increase the signature generation time or the signature size).

However, we note that usually, the FORS signature generation time is only a small part of the full signature generation time. This means we can potentially calculate the root of several

FORS signatures with a small increase in the overall signature generation time.

In the bottom layer of the tree, instead of signing the root of a FORS signature, we can instead sign multiple FORS signatures using a Merkle tree. This means that we increase the total number of FORS signatures, with only a negligible increase in verification time and signature size (the added cost of the Merkle authentication path). For example, if we sign a Merkle tree over 4 FORS signatures, we increase the FORS signature generation time by 4 but only add two hash values to the signature size and two calls to the hash function in verification.

Note that we could simply increase the size of the FORS signature, but that would require a significant increase in the signature size (due to longer or more authentication paths) and a small increase in verification time.

Taking this approach to the extreme, we can imagine a "soft state-full" variant to XMSS. In this variant, we don't use any WOTS+ signatures but only use a tree of FORS signatures. This will allow us to support a small number of signatures while only bounding the total number of signatures without maintaining an exact state or counter.

B. Reducing the verification time for WOTS+C

In our proposed WOTS+C, we find a digest such that $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$, where $S_{w,n}$ is the expected value. Instead, we can use larger values for $S_{w,n}$. As long as the value we used is not much larger than the expected value, the added cost for finding a good counter is not high. On the other hand, as $S_{w,n}$ gets larger, the number of hash evaluations required by the verifier gets smaller, reducing the total verification time.

ACKNOWLEDGMENTS

We thank Ofek Bransky for his contribution to the implementation and benchmark of the code. Eyal Ronen is supported in part by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik ICRC, and Robert Bosch Technologies Israel Ltd. He is a member of CPIIS. Eylon Yogev is supported by an Alon Young Faculty Fellowship, by the Israel Science Foundation (Grant No. 2893/22), and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. Andreas Hülsing and Mikhail Kudinov are supported by an NWO VIDI grant (Project No. VI.Vidi.193.066).

REFERENCES

[1] Morton Abramson. Restricted combinations and compositions. *Fibonacci Quart*, 14(5):439–452, 1976.

[2] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. Sphincs+ submission to the nist post-quantum project, v.3.1. 2022.

[3] Jean-Philippe Aumasson and Guillaume Endignoux. Clarifying the subset-resilience problem. *IACR Cryptol. ePrint Arch.*, page 909, 2017.

[4] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In *Cryptographers' Track at the RSA Conference*, pages 219–242. Springer, 2018.

[5] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.

[6] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs$^+$ signature framework. In *CCS*, pages 2129–2146. ACM, 2019.

[7] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. pages 2129–2146, 2019.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30):320–337, 2009.

[9] Ward Beullens. Breaking rainbow takes a weekend on a laptop. *IACR Cryptol. ePrint Arch.*, page 214, 2022.

[10] Joppe W. Bos, Andreas Hülsing, Joost Renes, and Christine van Vredendaal. Rapidly verifiable xmss signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):137–168, Dec. 2020.

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

[12] Oded Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, pages 104–110, 1986.

[13] Shay Gueron and Nicky Mouha. Sphincs-simpira: Fast stateless hash-based signatures with post-quantum security. *IACR Cryptol. ePrint Arch.*, page 645, 2017.

[14] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, pages 173–188, 2013.

[15] Andreas Hülsing. WOTS+ - shorter signatures for hash-based signature schemes. *IACR Cryptol. ePrint Arch.*, page 965, 2017.

[16] Andreas Hülsing and Mikhail A. Kudinov. Recovering the tight security proof of sphincs$^+$. *IACR Cryptol. ePrint Arch.*, page 346, 2022.

[17] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. pages 387–416, 2016.

[18] Leslie Lamport. Constructing digital signatures from a one way function. Technical report, 1979. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010.

[19] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

[20] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY, 1990. Springer New York.

[21] Lucas Pandolfo Perin, Gustavo Zambonin, Ricardo Custódio, Lucia Moura, and Daniel Panario. Improved constant-sum encodings for hash-based signatures. *Journal of Cryptographic Engineering*, 11(4):329–351, 2021.

[22] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. In *Information Security and Privacy, 7th Australian Conference, ACISP 2002, Melbourne, Australia, July 3-5, 2002, Proceedings*, pages 144–153, 2002.

[23] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. In *ACISP*, volume 2384 of *Lecture Notes in Computer Science*, pages 144–153. Springer, 2002.

[24] Kaiyi Zhang, Hongrui Cui, and Yu Yu. Sphincs-α: A compact stateless hash-based signature scheme. *IACR Cryptol. ePrint Arch.*, page 59, 2022.

APPENDIX

A. Brief description of SPHINCS+

Here we briefly describe the SPHINCS$^+$ signature scheme from [16]. This description is the same as in the referred paper and is presented here to make the paper self-sufficient. An example of the SPHINCS$^+$ structure was presented in Figure 2. A detailed description can be found in [7].

The public key consists of two n-bit values: a random public seed $\mathbf{PK}$.seed and the root of the top tree in the hypertree

structure. $\mathbf{PK}$.seed is used as a first argument for all tweakable hash function calls. The private key contains two more n-bit values $\mathbf{SK}$.seed and $\mathbf{SK}$.prf. We now discuss the main parts of SPHINCS$^+$ starting with the addressing scheme. As SPHINCS$^+$ uses Tweakable Hash Functions (THF), different tweaks are required for all calls to THFs. The tweaks are instantiated by addresses. An address is a 32-byte value that describes the position of a hash function call within the virtual structure of a SPHINCS$^+$ key pair. Addresses are built hierarchically and a prefix describes to which part of the SPHINCS$^+$ structure the primitive that uses the hash belongs. We denote this prefix as $\mathbf{ADRS}$.

WOTS-TW is a one-time signature scheme used in the SPHINCS$^+$ structure. The secret keys of WOTS-TW are pseudorandomly generated using $\mathbf{SK}$.seedand tweaks for appropriate addresses. The public key is formed by a sequence of hashing applied to the elements of the WOTS-TW secret key.

Another building block is binary trees. In the SPHINCS$^+$ algorithm, binary trees of height γ always have 2^γ leaves. Each leaf $L_i,\ i \in [0, 2^\gamma - 1]$ is a bit string of length n. Each node of the tree $N_{i,j},\ 0 < j \leq \gamma, 0 \leq i < 2^{\gamma - j}$ is also a bit string of length n. The values of an internal nodes of the tree are calculated from the children of that node using a THF. A leaf of a binary tree is the output of a THF that takes the elements of a WOTS-TW public key as input.

Binary trees and WOTS-TW signature schemes construct a hypertree structure. WOTS-TW public keys form the leaves of the binary trees. These instances are then used to sign the roots of binary trees on lower levels. WOTS-TW instances on the lowest level are used to sign the public key of a FORS (Forest of Random Subsets) few-time signature scheme. FORS is defined with the following parameters: $k \in \mathbb{N}$, $t = 2^a$. This algorithm can sign message digests of length ka-bits.

FORS key pair. The private key of FORS consists of kt pseudorandomly generated n-bit values grouped into k sets of t elements each. To get the public key, k binary hash trees are constructed. The leaves in these trees are k sets (one for each tree) of t values each. Thus, we get k trees of height a. As roots of k binary trees are calculated, they are compressed using a THF. The resulting value will be the FORS public key.

FORS signature. A ka bits message digest is divided into k lines of a bits. Each line is interpreted as a leaf index corresponding to one of the k trees. The signature consists of these leaves and their authentication paths. An authentication path for a leaf is the set of siblings of the nodes on the path from this leaf to the root. The verifier reconstructs the tree roots, compresses them, and verifies them against the public key. If all match, the signature is declared valid. Otherwise, it is declared invalid.

The last thing to discuss is the way the message digest is calculated. First, a pseudorandom value $\mathbf{R}$ is prepared as $\mathbf{R} = \mathbf{PRF}_{msg}(\mathbf{SK}_{\mathsf{prf}}, \mathsf{OptRand}, M)$ using a dedicated secret key element $\mathbf{SK}$.prf and the message. This function can be made non-deterministic, initializing the value OptRand with randomness. The $\mathbf{R}$ value is part of the signature. Using $\mathbf{R}$, we calculate the index of the FORS key pair with which the message will be signed and the message digest itself: $(\mathsf{MD}||\mathsf{idx}) = \mathbf{H}_{\mathbf{msg}}(\mathbf{R}, \mathbf{PK}.\mathsf{seed}, \mathbf{PK}.\mathsf{root}, M)$.

The key generation algorithm generates the WOTS-TW instances of the top-level binary tree from $\mathbf{SK}$.seedand uses the public keys of those instances to compute the root of the binary tree. $\mathbf{PK}$.seed together with the root form a public key.

The signature algorithm computes the digest. Then uses the idx-th FORS instance to produce a FORS signature. The WOTS-TW signature on the FORS public key is computed. The authentication path that helps compute this binary tree's root will be added to the signature. The same procedure is repeated up to the top-level binary tree. The signature consists of the randomness $\mathbf{R}$, the FORS signature (under idx from $\mathbf{H}_{\mathbf{msg}}$) of the message digest, the WOTS-TW signature of the corresponding FORS public key, and a set of authentication paths and WOTS-TW signatures of tree roots.

To verify this chain, the verifier iteratively reconstructs the public keys and tree roots until it gets the root of the top tree. The signature is accepted if this matches the root given in the SPHINCS$^+$ public key.

B. Security proof for standalone WOTS+C

To prove the security of WOTS+C we give a reduction from multi-target extended target collision resistance (m-eTCR) [17]. Assume a keyed hash function $H : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$. For such a hash function we define m-eTCR below. To keep the definition readable we use a challenge oracle $\mathsf{Box}(\cdot)$ that on input of a message outputs a random function key K.

Definition A.1 (Multi-target extended target collision resistance (m-eTCR) [17])**.** *The success probability of an adversary $\mathcal{A}$ against* M-ETCR *that makes no more than p queries to* $\mathsf{Box}(\cdot)$ *is defined as:*

$$\mathsf{Succ}^{\text{M-ETCR}}_{H,p}(\mathcal{A}) = \Pr\left[(M', K', i) \xleftarrow{\$} \mathcal{A}^{\mathsf{Box}(\cdot)}(1^n) : M' \neq M_i \wedge H_{K_i}(M_i) = H_{K'}(M')\right].$$

We base the security of WOTS+C on the security of the original signature scheme (either WOTS+ or WOTS-TW) and the M-ETCR property of H. We do so giving a game hopping proof. GAME.0 is the original WOTS+C game in the EU-CMA model. GAME.1 is the same as GAME.0, but we consider the game lost if the forgery together with a signature query response presents a collision under H. We limit the difference between those two games by the M-ETCR property of H. Then we show that the success of the adversary in GAME.1 is upper bounded by the security of the original signature scheme WOTS* (WOTS-TW or WOTS+).

Theorem A.2. *Let WOTS* be either WOTS+ or WOTS-TW. Let H be a keyed hash function. Then the insecurity of WOTS+C against one-time EU-CMA attacks is bounded by*

$$\mathsf{InSec}^{\textit{EU-CMA}}(\textit{WOTS+C}; t; 1) \leq 1/\epsilon \cdot \mathsf{InSec}^{\text{M-ETCR}}(H; \tilde{t}, q) + \mathsf{InSec}^{\textit{EU-CMA}}(\textit{WOTS*}; t', 1)$$

Algorithm 1: M-ETCR reduction

Input : M-ETCR challenger C, adversary $\mathcal{A}$ against WOTS+C
Output : (K^*, M^*, i) or $\perp$

1 Generate a random public seed $P \leftarrow_{\$} \{0,1\}^n$
2 Generate the context information for WOTS+C instance T
3 Run KeyGen of WOTS* providing P and T if needed
4 Give the public key Pub to the adversary $\mathcal{A}$
5 Receive a signing query m from $\mathcal{A}$
6 **for** $i = 1;\ i++;\ i < q + 1$ **do**
7 Query C with $m' = (m, \text{context information})$
8 Get K_i from C
9 Set $seed = K_i$ Compute $d_i = H(seed, m')$
10 **if** d_i *satisfies the properties for WOTS+C* **then**
11 Break;
12 $seed = 0$
13 **if** $seed == 0$ **then**
14 **return** $\perp$
15 Map d_i to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$
16 For $i \in [\ell]$ compute σ_i as is Sign algorithm of WOTS*
17 Send $\sigma = (\sigma_1, \ldots, \sigma_\ell, seed), m$ to $\mathcal{A}$
18 Obtain a forgery $\sigma^* = [\sigma^* = (\sigma_1^*, \ldots, \sigma_\ell^*, seed^*), m^*]$
19 Set $m'' = (m^*, \text{context information})$
20 **if** $m^* \neq m \wedge \mathsf{Verify}(m, \sigma^*, Pub) \wedge H(seed^*, m'') = H(seed, m')$ **then**
21 **return** $seed^*, m'', seed, m'$
22 **else**
23 **return** $\perp$

Algorithm 2: WOTS* reduction

Input : WOTS* challenger C, adversary $\mathcal{A}$ against WOTS+C
Output : message, signature

1 Given $\mathsf{pk} \leftarrow \mathsf{KeyGen}(1^n)$ (generated by the WOTS* scheme), take $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}})$ and create a public key $\mathsf{pk}' = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}_1 - z})$ for $\mathcal{A}$ by removing the len_2 words encoding the sum and to the first z words
2 Obtain a signature query m from the adversary $\mathcal{A}$
3 Sample a random seed $seed \leftarrow_{\$} \{0,1\}^n$ until $d = H(seed, \mathsf{m}, \text{context information})$ satisfies the properties of WOTS+C
4 Send d as a signature query for C
5 Obtain a signature for d: $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$
6 Set $\sigma' = (\sigma_1, \ldots, \sigma_{\mathsf{len}_1 - z}, seed)$
7 Send σ' to the adversary $\mathcal{A}$
8 Obtain a forgery (m^*, σ^*) from $\mathcal{A}$
9 Compute $d^* = H(seed^*, \mathsf{m}^*, \text{context information})$. Set $\tilde{\sigma} = (\sigma_1^*, \ldots, \sigma_{\mathsf{len}_1 - z}^*, \sigma_{\mathsf{len}_1 - z + 1}, \ldots, \sigma_{\mathsf{len}})$ by adding the truncated parts from the signature we received from the WOTS* challenger
10 **return** $d^*, \tilde{\sigma}$

where ϵ is the probability of finding a hash that satisfies the conditions of WOTS+C with q queries, $\widetilde{t} = t + q$, and t' is the time needed to find a hash that satisfies the conditions of WOTS+C, where time is given in number of hash function calls.

Proof. As we mentioned above let us define GAME.0 as the original game. GAME.1 differs from GAME.0 in that we consider the game lost if one can extract a collision under H from a signature query and the forgery. Assume the signature query was for message m. Assume the response was $\sigma = (\sigma_1, \ldots, \sigma_{len_1 - z}, s)$ and the valid forgery for message m^* is $\sigma^* = S(\sigma_1^*, \ldots, \sigma_{len_1 - z}^*, s^*)$. We assume that GAME.1 is lost if $H(s, m) = H(s^*, m^*)$.

We now show that the difference in winning the two games can be bounded by the M-ETCR security. Consider Algorithm 1. In the algorithm we view a hash function H for message hashing as a keyed hash function. But instead of a key it accepts a random seed. The seed (which can be viewed as a key) is then published as part of the signature. Note that every time the algorithm does not abort, the forgery contains a collision for a previous query under H. The occurrence of this event is exactly the difference between GAME.0 and GAME.1 (conditioned on that it finds a proper hash with q queries in the first place). At the same time, if the reduction does not abort, it outputs a collision under H which is a valid solution for the M-ETCR challenge. Assume that the probability of eventually hitting line 11 in Algorithm 1 is ϵ then we obtain the following inequality:

$$|\text{GAME.0} - \text{GAME.1}| \leq 1/\epsilon \cdot \mathsf{InSec}^{\text{M-ETCR}}(H; \tilde{t}, q)$$

We are left to bound the probability of the adversary in succeeding in GAME.1. To do so we construct another algorithm. In Algorithm 2 we see that if $d^* \neq d$ than any forgery for WOTS+C results in a forgery for WOTS*. Since the case where $d^* = d$ is excluded in GAME.1 we conclude that

$$\text{GAME.1} \leq \mathsf{InSec}^{\text{EU-CMA}}(\text{WOTS*}; t', 1)$$

Note here that t' depends on how many iterations are done to find d. We give bounds for this in Section III-C. This concludes the proof. □

C. Security of WOTS-TW in the EU-naCMA model

In this section we give a full proof for Theorem A.5. For this we introduce two more security notions. First we formally define EU-naCMA security [16]. Next we introduce a special target-collision resistance notion for tweakable hash functions. In the proofs we utilize an oracle $\mathcal{O}^{+C}(P, \cdot, \cdot)$ which on a fixed public key accepts a tweak and a message (T, M) and outputs $\mathsf{Th}(P, T, M \| i)$ where i is chosen so that the hash satisfies the WOTS+C requirements.

EU-naCMA security is defined using the following experiment where S is a shared state of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$.

Experiment $\mathsf{Exp}^{\mathsf{EU-naCMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$**:**
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Kg}(1^n)$.
$(\{M_1, \ldots, M_q\}, S) \leftarrow \mathcal{A}_1()$.
Compute $\{(M_i, \sigma_i)\}_{i=1}^q$ using $\mathsf{Sign}(\cdot, \mathsf{sk})$.
$(M^\star, \sigma^\star) \leftarrow \mathcal{A}_2(S, \{(M_i, \sigma_i)\}_{i=1}^q, pk)$
Return 1 iff $\mathrm{V}(M^\star, \sigma^\star, \mathsf{pk}) = 1$ and $M^\star \notin \{M_i\}_{i=1}^q$.

Definition A.3 (EU-naCMA [16])**.** *Let* Dss *be a digital signature scheme. We define the success probability of an*

adversary $\mathcal{A}$ against the EU-naCMA *security of* Dss *as the probability that the above experiment outputs 1:*

$$\mathrm{Succ}^{\mathsf{EU-naCMA}}_{\mathsf{Dss}(1^n),q}(\mathcal{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{EU-naCMA}}_{\mathsf{Dss}(1^n)}(\mathcal{A}) = 1\right],$$

where q denotes the number of messages that $\mathcal{A}_1$ asks the game to sign.

When we limit the number of queries q to the signing oracle to 1 we call the model one-time EU-naCMA.

We now turn to special target-collision resistance.

Definition A.4 (S-TCR(Prop))**.** *In the following let* 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* S-TCR(Prop) *security of* 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 of the form $(\mathcal{T} \times \mathcal{M})$ to an oracle $\mathcal{O}^{\mathsf{Prop}}(P,\cdot,\cdot)$, which works the following way: $\mathcal{O}^{\mathsf{Prop}}(P,T,M) = \{\mathsf{Th}(P,T,M||i), i\}$, where $i \in \mathbb{N}$ such that $\mathsf{Prop}(\mathsf{Th}(P,T,M||i)) = 1$. We denote the set of $\mathcal{A}_1$'s queries and responses by $Q = \{(T_i, M_i), (y_i, j_i)\}_{i\in[p]}$ and define the predicate $\mathsf{DIST}(\{T_i\}_{i\in[p]}) = (\forall i,k \in [1,p], T_i \neq T_k)$, i.e., $\mathsf{DIST}(\{T_i\}_{i\in[p]})$ outputs 1 iff all tweaks are distinct.*

$$\begin{aligned}\mathsf{Succ}^{\text{S-TCR(Prop)}}_{\mathsf{Th},p}(\mathcal{A}) = \Pr[&P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathcal{O}^{\mathsf{Prop}}(P,\cdot,\cdot)}();\\ &(i, M, counter) \leftarrow \mathcal{A}_2(Q, S, P, \mathsf{Th}) :\\ &\quad \mathsf{Th}(P,T_i,M_i||j_i) = \mathsf{Th}(P,T_i,M||counter)\\ &\quad \wedge\ M \neq M_i \wedge \mathsf{DIST}(\{T_i\}_{i\in[p]})]\ .\end{aligned}$$

This notion is a variant of the notion of single-function multi-target target-collision resistance (SM-TCR) that is used in the analysis of SPHINCS+. The new notion is necessary for a technical reason: In SM-TCR the adversary is only allowed to query its challenge oracle once per tweak. However, we need targets that fulfill Prop. The search for these requires to query the oracle with different counter values for the same tweak. In S-TCR(Prop) this is handled by $\mathcal{O}^{\mathsf{Prop}}$. In the full version of the paper, we show that generic attacks against S-TCR(Prop) have the same complexity as those against SM-TCR.

In the rest of the paper we instantiate Prop with the predicate $+C$. This predicate is modelling the requirements of WOTS+C, i.e. on input d it returns true if the len_1 values $a_1, \ldots, a_{\mathsf{len}_1} \in [w]$ representing the base-w encoding of d satisfy:

1) $\sum_{i=1}^{\mathsf{len}_1} a_i = S_{w,n}$.
2) $\forall i \in [z] : a_i = 0$.

Another thing to discuss is the counter. Assume the probability of hitting a good hash is p_ν. The probability of not succeeding after k tries is $(1-p_\nu)^k$. In case we have d instances of WOTS the probability of not being able to find a good counter after k tries for each of them is $P = 1-(1-(1-p_\nu)^k)^d$. For example if we set $k = 2^{30}$, $p_\nu \approx 0.015$, and $d = 16$ (example of actual parameters for $w = 16$ and $\ell = 32$) we get approximately $1-(1-2^{-23412264})^{16}$. Note that $(1-2^{-23412264})^{16}$ is extremely close to 1. Hence, the resulting probability P is extremely close to 0. So in our analysis we assume that it is always possible to find a good counter and the adversary cannot make its behavior depend on the existence or nonexistence of a fitting counter.

As we mentioned before, we also utilize collections of tweakable hash functions and a $\mathsf{Th}_\lambda(P,\cdot,\cdot)$ oracle. This idea was introduced in [16]. More complex constructions such as SPHINCS+ utilize collections of tweakable hash functions. The main difference between the tweakable hash functions in the collection is the input length. THFs for one instance of the scheme are united by using the same public parameter for all the calls. On the other hand each call is separated from another by using different tweaks. To obtain a security proof one may want to put challenges inside the scheme structure. These challenges may depend on previous invocations of the collection of THFs. If the reduction does not have access to the public parameter that is used throughout the whole scheme at the moment of challenge placement it will not be able to place create a suitable challenge. Making a Th_λ oracle available solves this problem. The oracle shares the public parameter with the challenger. The adversary should be able to prepare for a challenge query by using Th_λ. To maintain security the adversary is allowed to make queries to Th_λ only with tweaks that are different from the tweaks used for challenge queries. If we consider a random tweakable function, then tweak defines an independent random function, which gives no information to the adversary about the challenges, hence such oracle does not affect the security of the primitive.

We now got everything to prove the following theorem.

Theorem A.5. *Let WOTS-TW be a signature scheme as defined in [16]. Let* Th *be a tweakable hash function. Then the insecurity of WOTS+C using* Th *against one-time EU-naCMA attacks is bounded by*

$$\begin{aligned}\mathsf{InSec}^{\textit{EU-naCMA}}(\textit{WOTS+C}; t; 1) \leq\ &\mathsf{InSec}^{\text{S-TCR(+C)}}(\mathsf{Th}; \tilde{t}, 1)\\ &+ \mathsf{InSec}^{\textit{EU-naCMA}}(\textit{WOTS-TW} \in \mathsf{Th}_\lambda; \tilde{t}, 1),\end{aligned}$$

where $\widetilde{t} \approx t$ is the time needed to find a proper counter value to obtain a hash that satisfies the requirements of WOTS+C.

Proof. We follow ideas from the EU-CMA proof. We define GAME.0 as the original game. GAME.1 differs from GAME.0 in that we consider the game lost if one can extract a collision under Th from a signature query and the forgery. Assume the message sent for the signature query was m. Assume the response was $\sigma = (\sigma_1, \ldots, \sigma_{len_1-z}, i)$ and the valid forgery for message m^* is $\sigma^* = (\sigma^*_1, \ldots, \sigma^*_{len_1-z}, j)$. We consider GAME.1 lost if $\mathsf{Th}(P,T^*,m||i) = \mathsf{Th}(P,T^*,m^*||j)$.

To prove a bound for this game-hop we utilize the S-TCR(+C) property. Consider Algorithm 3. One can see that similar to Algorithm 1 every time the algorithm does not abort the forgery contains a collision under Th which is on the one hand the difference between GAME.0 and GAME.1 and on the other hand a solution for the S-TCR(+C) challenge. Hence, we obtained the following inequality:

$$|\mathrm{GAME.0} - \mathrm{GAME.1}| \leq \mathsf{InSec}^{\text{S-TCR(+C)}}(\mathsf{Th}; \tilde{t}, 1)$$

Now we limit the probability of the adversary in succeeding in GAME.1. To do so we construct another algorithm. In

Algorithm 3: S-TCR(+C) reduction

Input : S-TCR(+C) challenger C, adversary $\mathcal{A}$ against WOTS+C
Output : (M^*, j) or $\perp$

Generate the context information for WOTS+C instance T^*
Receive a signing query m from $\mathcal{A}$
Query $\mathcal{O}^{+C}$ from the S-TCR(+C) challenger C with (T^*, m)
Obtain a response $\{(T^*, m), i, \mathsf{Th}(P, T^*, m||i)\}$
Obtain public value P from C
Run KeyGen of WOTS+C with public seed P and context information T^*
Obtain a signature σ on the requested message by mapping $d = \mathsf{Th}(P, T^*, m||i)$ to len_1 chain locations $a_1, \ldots, a_{\mathsf{len}_1}$ and proceeding as in the Sign algorithm
Return the signature to the adversary
Give the public key Pub to the adversary $\mathcal{A}$
Obtain a forgery $\sigma^* = [\sigma^* = (\sigma_1^*, \ldots, \sigma_\ell^*, j), m^*]$
if $m^* \neq m \wedge \mathsf{Verify}(m, \sigma^*, Pub) \wedge \mathsf{Th}(P, T^*, m^*||j) = \mathsf{Th}(P, T^*, m||i)$ **then**
  return j, m^*, i
else
  return $\perp$

Algorithm 4: WOTS-TW reduction

Input : WOTS-TW $\in \mathsf{Th}_\lambda$ challenger C, adversary $\mathcal{A}$ against WOTS+C
Output : message, signature

Obtain a signature query m from the adversary $\mathcal{A}$
Query Th_λ with $T^*, \mathsf{m}\|seed'$ for $seed' \in \{0,1\}^r$ until a value $seed$ is found such that $d = \mathsf{Th}(P, T^*, \mathsf{m}\|seed)$ satisfies the properties of WOTS+C
Send d as a signature query for C
Obtain a signature for d: $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$
Set $\sigma' = (\sigma_1, \ldots, \sigma_{\mathsf{len}_1 - z}, seed)$
Send σ' to the adversary $\mathcal{A}$
Obtain $\mathsf{pk} \leftarrow \mathsf{KeyGen}(1^n)$ (generated by the WOTS-TW scheme), take $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}}, P)$ and create a public key $\mathsf{pk}' = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}_1 - z}, P)$ for $\mathcal{A}$ by removing the len_2 words encoding the sum and to the first z words
Obtain a forgery (m^*, σ^*) from $\mathcal{A}$
Compute $d^* = \mathsf{Th}(P, T^*, \mathsf{m}^*||seed^*)$. Set $\tilde{\sigma} = (\sigma_1^*, \ldots, \sigma_{\mathsf{len}_1 - z}^*, \sigma_{\mathsf{len}_1 - z + 1}, \ldots, \sigma_{\mathsf{len}})$ by adding the truncated parts from the signature we received from the WOTS-TW challenger
return $d^*, \tilde{\sigma}$

Algorithm 4 we see that if $d^* \neq d$ then a forgery for WOTS+C results in a forgery for WOTS-TW. Since we excluded the case where $d^* = d$ we conclude that

$$\text{GAME.1} \leq \mathsf{InSec}^{\text{EU-naCMA}}(\text{WOTS-TW} \in \mathsf{Th}_\lambda; t', 1)$$

Note here that $\tilde{t}$ depends on how much time it took the oracle $\mathcal{O}^{+C}$ to find d. We give the calculations for that in Section III-C. This concludes the proof. □

In [16], it was shown that

$$\mathrm{InSec}^{\text{EU-naCMA}}(\text{WOTS-TW}; t, 1) \leq \mathrm{InSec}^{\text{PRF}}(\mathbb{PRF}; \widetilde{t}, \mathsf{len}) + \mathrm{InSec}^{\text{SM-TCR}}(\mathsf{Th}; \widetilde{t}, \mathsf{len} \cdot w) + \mathrm{InSec}^{\text{SM-PRE}}(\mathsf{Th}; \widetilde{t}, \mathsf{len}) + w \cdot \mathrm{InSec}^{\text{SM-UD}}(\mathsf{Th}; \widetilde{t}, \mathsf{len})$$

with $\widetilde{t} = t + \mathsf{len} \cdot w$, where time is given in number of Th evaluations. See Appendix D for the precise definition of PRF, SM-PRE, and SM-UD, which we do not provide here.

Our proof only adds a term for the S-TCR(+C) notion. Thus, we conclude with our security bound:

$$\begin{aligned}\mathrm{InSec}^{\text{EU-naCMA}}(\text{WOTS+C}; t, 1) &\leq \mathrm{InSec}^{\text{PRF}}(\mathbb{PRF} \in \mathsf{Th}_\lambda; \widetilde{t}, \mathsf{len}) \\ &+ \mathrm{InSec}^{\text{SM-TCR}}(\mathsf{Th} \in \mathsf{Th}_\lambda; \widetilde{t}, \mathsf{len} \cdot w) \\ &+ \mathrm{InSec}^{\text{SM-PRE}}(\mathsf{Th} \in \mathsf{Th}_\lambda; \widetilde{t}, \mathsf{len}) \\ &+ w \cdot \mathrm{InSec}^{\text{SM-UD}}(\mathsf{Th} \in \mathsf{Th}_\lambda; \widetilde{t}, \mathsf{len}) + \mathsf{InSec}^{\text{S-TCR(+C)}}(\mathsf{Th}; t', 1)\end{aligned}$$

with $\widetilde{t} = t + \mathsf{len} \cdot w$ and $t' \approx t$.

D. Properties definitions

In this section we recall the remaining security definitions that we require from Th as given in [16], and the definition of d-EU-naCMA security. These are not novel but included for the paper to be self-contained.

We start with d-EU-naCMA security which is defined via an experiment. In the experiment the adversary is allowed to make signature queries for different instances of WOTS+C. The set of queries $Q = \{(M_i, \mathbf{ADRS}_i)\}_{i=1}^d$ contains messages M_i to be signed and addresses $\mathbf{ADRS}_i$ that reference the different instances of WOTS+C used to sign the messages. We require that no more than one signing query for each WOTS+C instance can be made, so every address should be unique. Additionally, the adversary is allowed to query the collection oracle Th_λ defined above. The oracle Th_λ can be used to generate dependent signing queries and should not provide additional information about the WOTS+C instances. This is modeled via the limitation that the addresses used as tweaks in queries to Th_λ must be different from the ones used in the signing queries. The resulting experiment is as follows:

Experiment $\mathsf{Exp}^{\text{d-EU-naCMA}}_{\text{WOTS+C}}(\mathcal{A})$

- $Seed \leftarrow_\$ \{0,1\}^n$
- $\mathcal{S} \leftarrow_\$ \{0,1\}^n$
- $state \leftarrow \mathcal{A}_1^{\text{WOTS+C.sign}(\cdot, Seed, \cdot, \mathcal{S}), \mathsf{Th}_\lambda(Seed, \cdot, \cdot)}(\,)$
- $(M^\star, \sigma^\star, j) \leftarrow \mathcal{A}_2(state, Seed)$
- Return 1 iff $j \in [1, d] \wedge [\mathrm{Vf}(\mathsf{PK}_j, \sigma^\star, M^\star, \mathbf{ADRS}_j) = 1] \wedge [M^\star \neq M_j] \wedge [\mathbf{DIST}(\{\mathbf{ADRS}_i\}_{i=1}^d)] \wedge [\forall \mathbf{ADRS}_i \in Q, \mathbf{ADRS}_i \notin T' = \{\mathsf{adrs}(\mathsf{T_i})\}_{i=1}^p]$,

where $\mathbf{DIST}(\{\mathbf{ADRS}_i\}_{i=1}^d)$ outputs 1 iff all arguments are distinct and 0 otherwise, T' denotes a set of tweaks used for Th_λ queries and $\mathsf{adrs}(\cdot)$ returns a prefix of a tweak.

The success probability of an adversary $\mathcal{A}$ in the described experiment with d instances is defined as $\mathsf{Succ}^{\text{d-EU-naCMA}}_{\text{WOTS+C},d}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr\left[\mathsf{Exp}^{\text{d-EU-naCMA}}_{\text{WOTS+C}}(\mathcal{A}) = 1\right]$.

Next we cover the remaining security properties for THFs.

Definition A.6 (SM-TCR)**.** *In the following let* 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* 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 classical queries to an oracle $\mathsf{Th}(P,\cdot,\cdot)$. 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.

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

Definition A.7 (SM-PRE)**.** *In the following let* 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-PRE *security of* 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 classical queries to an oracle $\mathsf{Th}(P,\cdot,x_i)$, where x_i is chosen uniformly at random for the query i (the value of x_i stays hidden from $\mathcal{A}$). We denote the set of $\mathcal{A}_1$'s queries by $Q = \{T_i\}_{i=1}^p$ and define the predicate $\mathbf{DIST}(\{T_i\}_{i=1}^p)$ as we did in the definition above.*

$$\mathsf{Succ}^{\text{SM-PRE}}_{\mathsf{Th},p}(\mathcal{A}) = \Pr[P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(P,\cdot,x_i)}(\,); (j,M) \leftarrow \mathcal{A}_2(Q,S,P) : \mathsf{Th}(P,T_j,M) = \mathsf{Th}(P,T_j,x_j) \wedge \mathbf{DIST}(\{T_i\}_{i=1}^p)]$$

Definition A.8 (SM-UD)**.** *In the following let* 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-UD *security of* Th*. The definition is parameterized by the number of targets p for which it must hold that $p \leq |\mathcal{T}|$. First the challenger flips a fair coin b and chooses a public parameter $P \leftarrow_{\$} \mathcal{P}$. Next consider an oracle $\mathcal{O}_P(\mathcal{T}, \{0,1\})$, which works the following way: $\mathcal{O}_P(T,0)$ returns $\mathsf{Th}(P,T,x_i)$, where x_i is chosen uniformly at random for the query i; $\mathcal{O}_P(T,1)$ returns y_i, where y_i is chosen uniformly at random for the query i. In the definition, $\mathcal{A}_1$ is allowed to make p classical queries to an oracle $\mathcal{O}_P(\cdot,b)$. The goal of $\mathcal{A}$ is to distinguish whether the oracle is $\mathcal{O}_P(\mathcal{T},0)$ or $\mathcal{O}_P(\mathcal{T},1)$. We denote the set of $\mathcal{A}_1$'s queries by $Q = \{T_i\}_{i=1}^p$ and define the predicate $\mathbf{DIST}(\{T_i\}_{i=1}^p)$ as we did above.*

$$\mathsf{Adv}^{\text{SM-UD}}_{\mathsf{Th},p}(\mathcal{A}) = |\Pr[P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathcal{O}_P(\cdot,0)}(\,); 1 \leftarrow \mathcal{A}_2(Q,S,P) \wedge \mathbf{DIST}(\{T_i\}_{i=1}^p)] - \Pr[P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathcal{O}_P(\cdot,1)}(\,); 1 \leftarrow \mathcal{A}_2(Q,S,P) \wedge \mathbf{DIST}(\{T_i\}_{i=1}^p)]|$$

Definition A.9 (Keyed hash function)**.** *Let $\mathcal{K}$ be the key space, $\mathcal{M}$ the message space, and $\mathcal{N}$ the output space. A keyed hash function is an efficient function $F : \mathcal{K} \times \mathcal{M} \to \mathcal{N}$ generating an n-bit value out of a key and a message.*

In the following we give the definition for PRF security of a keyed hash function $F : \mathcal{K} \times \mathcal{M} \to \mathcal{N}$. In the definition of the PRF distinguishing advantage, the adversary $\mathcal{A}$ gets (classical) oracle access to either $F(S,\cdot)$ for a uniformly random secret key $S \in \mathcal{K}$ or to a function G drawn from the uniform distribution over the set $\mathcal{G}(\mathcal{M},\mathcal{N})$ of all functions with domain $\mathcal{M}$ and range $\mathcal{N}$. The goal of $\mathcal{A}$ is to distinguish both cases.

Definition A.10 (PRF)**.** *Let F be defined as above. We define the* PRF *distinguishing advantage of an adversary $\mathcal{A}$ making q queries to its oracle as*

$$\mathsf{Adv}^{\text{PRF}}_{F,q}(\mathcal{A}) = \left| \Pr_{S \leftarrow_{\$} \mathcal{K}}[\mathcal{A}^{F(S,\cdot)} = 1] - \Pr_{G \leftarrow_{\$} \mathcal{G}(\mathcal{M},\mathcal{N})}[\mathcal{A}^{G(\cdot)} = 1] \right|.$$

Here we present a multi-target version of DSPR which is denoted as SM-DSPR. To do so, we need a second-preimage exists predicate for tweakable hash functions.

Definition A.11 ($\mathsf{SP}_{P,T}$)**.** *A second preimage exists predicate of tweakable hash function $\mathsf{Th} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n$ with a fixed $P \in \mathcal{P}$, $T \in \mathcal{T}$ is the function $\mathsf{SP}_{P,T} : \{0,1\}^m \to \{0,1\}$ defined as follows:*

$$\mathsf{SP}_{P,T}(x) \stackrel{def}{=} \begin{cases} 1 \text{ if } |\mathsf{Th}^{-1}_{P,T}(\mathsf{Th}_{P,T}(x))| \geq 2 \\ 0 \text{ otherwise} \end{cases},$$

where $\mathsf{Th}^{-1}_{P,T}$ refers to the inverse of the tweakable hash function with fixed public parameter and a tweak.

Now we present the definition of SM-DSPR from [7] for a tweakable hash function. The intuition behind this notion is that the adversary should be unable to find a preimage that doesn't have a second preimage.

Definition A.12 (SM-DSPR)**.** *Let* Th *be a tweakable hash function. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be a two stage adversary. The number of targets is denoted with p, where the following inequality must hold: $p \leq |\mathcal{T}|$. $\mathcal{A}_1$ is allowed to make p classical queries to an oracle $\mathsf{Th}(P,\cdot,\cdot)$. We denote the query set $Q = \{(T_i, M_i)\}_{i=1}^p$ and predicate $\mathsf{DIST}(\{T_i\}_1^p)$ as in previous definitions.*

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

where

$$\mathsf{succ} = \Pr[P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(P,\cdot,\cdot)}(); (j,b) \leftarrow \mathcal{A}_2(Q,S,P) : \mathsf{SP}_{P,T_j}(M_j) = b \wedge \mathsf{DIST}(\{T_i\}_1^p)].$$

$$\mathsf{triv} = \Pr[P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(P,\cdot,\cdot)}(); (j,b) \leftarrow \mathcal{A}_2(Q,S,P) : \mathsf{SP}_{P,T_j}(M_j) = 1 \wedge \mathsf{DIST}(\{T_i\}_1^p)].$$