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

Andreas Hülsing	Mikhail Kudinov	Eyal Ronen	Eylon Yogev
andreas@huelsing.net	m.kudinov@tue.nl	eyal.ronen@cs.tau.ac.il	eylon.yogev@biu.ac.il
TU Eindhoven	TU Eindhoven	TAU	Bar-Ilan University

September 14, 2022

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 large) variant and a small (but costly) 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 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

1 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 [Beu22]. 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 [Lam79] and improved by Winternitz [Mer89]. In 2015, Bernstein et al. presented a stateless hash-based signature scheme called SPHINCS [Ber+15]. 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 [Gol86]. 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 [RR02a]. 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 [RR02b; GM17; AE17; Hül17; AE18; BHRV20; PZCMP21; ZCY22]. Most notable is the suggested proposal SPHINCS+ [BHKNRS19a], 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 [HK22] 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. The proposal includes a fast (but larger) variant and a small (but slower) variant for each security level. For example, for 192 bits of security, it has signatures of size $\approx$ 16KB for the small variant and $\approx$ 35KB for the fast variant. Unfortunately, further reduction in the signature size is not considered practical due to prohibitively high computational cost for the signer.

1.1 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. Our goal is to reduce the signature size while maintaining (and in some cases even improving) the speed of signature generation. Furthermore, the security of our scheme 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 3.

- 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 4.

- 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 6.

We provide a security proof of our scheme, a reference implementation based on the code of SPHINCS+,[1] and propose several parameters 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 the signing speed. We note that users may want to look for other trade-offs depending on the use case. Table 1 provides a comparison for the signature sizes of the SPHINCS+ variants submitted to

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

Table 1: 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 running-times of the signers are (approximately) the same in both scheme for all parameter settings.

	Small Signature Size		Fast Signature Size	
Security 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%)

round 3 of the NIST competition [Aum+22], 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 5).

Paper organization

The organization of the rest of the paper is as follows. In Section 2, we give the relevant background on the SPHINCS+ signature scheme and its components. In Section 3, we describe WOTS+C, our new variant of WOTS+, along with its theoretical analysis. In Section 4, we describe our FORS+C scheme, along with its theoretical analysis. In Section 5, we introduce SPHINCS+C and prove it secure. We then discuss parameter selection and propose concrete parameter sets in Section 6. In Section 7, we describe our implementation and provide a comparison with the SPHINCS+ scheme (in terms of speed and signature size). In Section 8 we provide a general discussion of further considerations that are relevant to our scheme. Finally, Section 9 suggests future work.

2 Background

In this subsection, we provide relevant background on the SPHINCS+ signature scheme, covering its main building blocks. Readers familiar with SPHINCS+ may safely skip this subsection.

2.1 WOTS

The Winternitz signature scheme (WOTS) and its variants (e.g., WOTS+) are one-time signature schemes [Hül13]. The core idea of WOTS (and its variants) is to use len function chains starting from random inputs. These random inputs together act as the secret key. The public key consists of all of the ends of each chain. 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 the basic WOTS scheme to present our ideas. Later in the paper, we discuss that with more complex proposals such as WOTS+ or WOTS-TW our optimizations work in the same way.

In more detail, WOTS 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. Let

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

The secret key consists of len n-bit random values: $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{len}})$, where $\mathsf{len} = \mathsf{len}_1 + \mathsf{len}2$. To compute public key one need to apply a hash function F iteratively $w-1$ times to each of the secret key elements: $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}}) = (F^{w-1}(\mathsf{sk}_1), \ldots, F^{w-1}(\mathsf{sk}_{\mathsf{len}}))$, where $F^i(x) = \underbrace{F(F(\ldots F(x)))}_{\text{F is applied } i \text{ times}}$. This way we obtain len chains.

To sign a message m, we interpret m as len_1 integers a_i each between 0 and $w-1$. We compute a checksum $C = \sum_{i=1}^{\mathsf{len}_1}(w-1-a_i)$, represented as string of len_2 base-w values $C = (C_1, \ldots, C_{\mathsf{len}_2})$. This checksum is also signed and is what provides security of the scheme. Let us denote $(b_1, \ldots, b_{\mathsf{len}}) = (a_1, \ldots, a_{\mathsf{len}_1}, C_1, \ldots, C_{\mathsf{len}_2})$. Using these len integers as chain lengths, the chaining hash function F is applied to the private key elements. The result is a list of len values, each is a n-bit string that consist of the signature: $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}}) = (F^{b_1}(\mathsf{sk}_1), \ldots, F^{b_{\mathsf{len}}}(\mathsf{sk}_{\mathsf{len}}))$. The verifier recomputes the checksum, and chain lengths, and then applies F to complete each chain: $\mathsf{pk}' = (F^{w-1-b_1}(\sigma_1), \ldots, F^{w-1-b_{\mathsf{len}}}(\sigma_{\mathsf{len}}))$. Then the verifier checks whether $\mathsf{pk}' = \mathsf{pk}$.

A lot of different variants were introduced in the literature, for example WOTS+ [Hül17], WOTS-TW [HK22] and several others. The main difference in all of them is how the chain is computed. As an illustration in WOTS+ instead of iterating over the same hash function, a new chaining function which involves random masking values r and a family of hash functions f_k is defined. In every iteration, the function first takes the bitwise xor of the intermediate value and bitmask r and evaluates f_k on the result afterwards. This enables them to get a tight security analysis using weak (inversion) properties of the hash functions. WOTS-TW is built on tweakable hash functions and introduces different security notions for such functions.

2.2 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+ [BHKNRS19a], introduced as an improvement to HORST [Ber+15]. The FORS scheme is defined in terms of integers k and $t = 2^b$, and can be used to sign strings 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, resulting in k roots. The public-key consists of the hash of all these roots together. Given a message of $k \cdot b$ bits, the signature algorithm extracts k strings of b bits. Each of these bit strings is interpreted as the index of a leaf in each of the k FORS trees. The signature contains the authentication path for each leaf. As each index is used in a different tree, FORS solves the problem of weak messages in HORST [AE17], that is due to a collision of two or more indices in the single tree used by HORST.

2.3 SPHINCS+

The SPHINCS+ [BHKNRS19a] scheme is a stateless hash-based signature improving upon the previous version called SPHINCS [Ber+15]. It uses a hyper-tree structure where each tree is a Merkle tree over the public keys of WOTS+ instances. The WOTS+ instances in the leaves of the lowest layer of the hyper-tree are used to sign the public keys of FORS instances.

When signing a message, one part of the digest of the message determines the exact leaf of the hyper-tree to be used. The FORS key pair that corresponds to that leaf is used to sign the second part of the digest. The FORS public key is signed by the WOTS+ key pair of that leaf. The root of each Merkle on the path from that leave to the root of the hyper-tree is then signed by a WOTS+ key pair in the leaves of the tree in the layer above it. This continues until we reach the root of the top-level tree, which is also the public key. The SPHINCS+ signature includes all of the used FORS and WOTS+ signatures and the authentication paths of the Merkle trees that are required for the verifier to compute the root of the hyper-tree. See [BHKNRS19a] for more details and security proofs. A brief description can also be found in Appendix A.

3 WOTS+C

In this section, we present a variant of the WOTS [Mer90] construction (which can also be applied to other variants). Our construction reduced the size of the signature, *without making the chains longer* and without increasing the running time of the 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 defined 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 reduced 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. See an illustration in Figure 1.

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

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

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 $\mathsf{pk}_i = F^{w-1}(\mathsf{sk}_i)$.

Algorithm 2: $\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$

1. Sample a salt $s \in \{0,1\}^n$ at random, until $d = H(s\|\mathsf{m})$ satisfies the above two conditions (if non 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 $\sigma = (\sigma_1, \ldots, \sigma_\ell, s, r)$.

Algorithm 3: $\mathbf{V}(1^n, \mathsf{m}, \sigma, \mathsf{pk})$

1. Parse σ as $(\sigma_1, \ldots, \sigma_\ell, s, r)$.
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 usually called chaining functions. Note that our modification behave the same regardless of the way these chains are computed. For now the reader can focus on a simple construction with one hash function. Latter in the paper we will show a construction with a more complex chaining function. The analysis of such construction is conceptually the same and differs only in the way we handle the compression

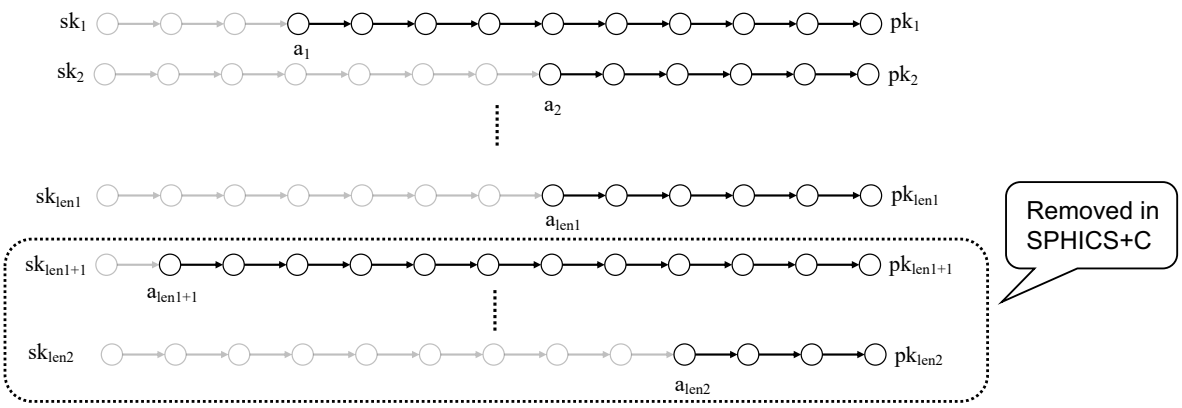

Figure 1: An illustration of the WOTS+ chains that are removed in the WOTS+C scheme.

hash function (in our example hash function 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 with the WOTS+ 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). The signing time has the additional step of finding the right counter value, but this is compensated with the smaller number of chains it needs to compute. 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 increase the overall signature generation time. However, as we discuss in Section 8.3, it can decrease the overall time for SPHINCS+C signature generation.

3.1 Proof of security

We tightly relate the EU-CMA security of our WOTS+C scheme to that of the WOTS+ [Hül17] 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 3.1. *Let $n, t, q \in \mathbb{N}$, $t, q = \text{poly}(n)$, $\mathsf{Dss}(1^n)$ 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 $\mathsf{Succ}^{EU\text{-}CMA}_{\mathsf{Dss}(1^n)}(\mathcal{A})$ 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+.

3.2 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 [BHRV20]. 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.

3.3 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 [Abr76] (Section 3, equation E) this number can be computed as:[2]

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

Then the probability of hitting a good hash is

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

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 (Appendix H). 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 6.2, we show how to bound the variance in the signature generation time.

[2]We use $(S_{w,n} + \mathsf{len})$ to compensate for the fact that [Abr76] assumes values range of $[1, w]$.

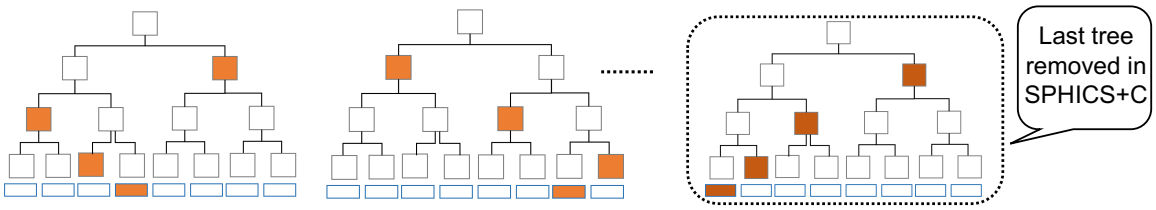

Figure 2: An illustration of the FORS tree that is removed in the FORS+C scheme.

4 FORS+C

In this section, we present the FORS+C scheme, as an improved variant to the FORS few-time signature scheme (see Section 2.2 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 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.

4.1 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 2.

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).

4.1.1 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 more detail, the security analysis of FORS shows that the overall success probability of the attacker 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 $t = 2^b$. We will now describe the security analysis.

We begin by describing the security analysis of FORS. Assume that the adversary has seen γ signatures. We use the analysis from [BHKNRS19a] 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 [BHKNRS19a], 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 [BHKNRS19a], 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

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

This improved security probability is later used in the SPHINCS+C schemes for the overall security analysis.

4.2 Interleaving trees

The main advantage of the FORS scheme compared to schemes with a single tree (HORS, HORST) is the simple and tight security analysis. The message is mapped to k leaves, each leaf in a *different tree*. Thus, we know these leaves will never collide: a signature will always contain k distinct leaves. This is important as an attacker can use duplicate leaves to gain an advantage [AE17].[3] Thus, in the FORS scheme, it suffices to analyze the security of a single tree and then conclude with overall security that is exponential in k, as the trees are independent.

While the FORS is simple and straightforward to analyze, one downside is that it does not enjoy the "path pruning" method to reduce the signature size that was suggested in [AE18] as "Octopus". In this method, one can significantly save in the authentication paths for opening leaves close to each other. For example, if one is to open the first leaf and the 4th leaf, then after two layers, their authentication paths are aligned and can be added to the signature only once. In HORS(T), there is one large tree and openings of k leaves that share some of the authentication paths, depending on the (random) location of the leaves. In FORS, all the paths are *always disjoint*, as each path in a different leaf.

We introduce a variant of FORS, which is able to enjoy both worlds at no computational or security cost. It enjoys the simple analysis of the FORS scheme and the path pruning method of the HORS(T) scheme. The idea is to take the k trees of the FORS, each with t leaves, and arrange them in an interleaving order before computing the Merkle tree. That is, in our variant, we still have k independent arrays of leaves, each of size

[3] Note that we can use a similar approach with a counter to both compress HORST and make sure there are no duplicate leaves. However, we choose to base our solution on the FORS signature to minimize the changes from SPHINCS+.

t. Each message is mapped to a single leaf in each array. However, before we compute the Merkle tree, we permute the leaves order as follows: the i-th node of the j-th FORS tree is mapped to location $(i-1) \cdot k + j$. For example, in the new order, the first k leaves correspond to the first leaves of each of the k trees.

As currently described, this variant suffers from the same limitation as previous suggestions, such as Octopus [AE18]. It will reduce the size of the FORS signature *on average*, but not by much. Furthermore, this will result in a variable signature size, and we still need to support the worst-case signature size, which is the same as without "path pruning".

We overcome the above limitations. We can again employ the same counter technique we use for tree removal. That is, the signer will use incrementing counter values until it finds a digest where the resulting FORS signature after "pruning" is under some pre-defined size. For example, for the parameter set $\log(t) = 15, k = 13$ used in our proposed small variant for 192-bit security (see Section 6), after 2^5 hashes, we expect to reduce the size of the FORS signature from $208 * 24 = 4992$ bytes to $194 * 24 = 4656$ bytes. A reduction of about 7% in the FORS signature size or 2.4% of the total signature size. Again, as we provide the counter to the verifier, the added run time cost is only in the signature generation phase. With this modification, we take the event of a having a relatively small signature, which happens with small probability, and make sure it always happens. This results with a signature with a fixed pre-defined (small) size.

The cost of using both the tree removal and path pruning is the product of the costs. As in our small variant for 192-bit security, by removing a tree of size 2^{12}, the total cost is $2^{12} \cdot 2^5 = 2^{17}$. This is still very small compared to the total signature time of this variant. Note that although calculating the authentication paths pruning is more costly than hash calculation, we only need to do it if all the bits of the tree removal are indeed zeros. This means that in our example, it only needs to be checked 2^5 times which is negligible compared to the total signature generation time.

Practical consideration. We note that this variant gives a reduced signature size over the original FORS construction. However, when we compare it against our variant with the removed tree (and with WOTS+C), for the specific trade-off point we use for our parameter sets, the savings are less significant (approximately 1% extra reduction in signature size). Thus, our implementation does not currently support this improvement, and all experiments shown in this paper are without it. In other contexts, this improvement could be significant, which is why we described it here and hope it will be helpful for others.

4.2.1 Security

The security analysis of this variant is exactly the same as in the original FORS scheme. The security relies on analyzing the probability of the adversary hitting all opened leaves and is not affected by the order of the leaves. The main advantage is that now we can exploit the path pruning technique to reduce the signature size. For example, suppose a signature contains the first leaf from the first two trees. In the original FORS scheme, this would lead to two disjoint paths. In our new scheme, these leaves sit adjacent to each other, and thus their combined authentication path is the same as a *single* authentication path.

4.3 Complexity analysis of FORS+C

The number of hash calls required for FORS+C signature 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 appendix H. If the interleaved trees optimization is used, the probability of finding a good hash is multiplied by the probability of finding a digest that results in a FORS signature that can be "pruned" under the pre-defined size.

In Section 6.2, we show how to bound the variance in the signature generation time.

5 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 [HK22]. In this section we discuss WOTS+C in the context of SPHINCS+ and show the security bound for SPHINCS+C.

5.1 WOTS+C in the context of SPHINCS+

In section 3 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 how 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 underlying WOTS scheme and on the complexity of finding a collision under H. In the 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 [BHKNRS19a], 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 give a tight reduction. After defining THFs, we give a construction of the WOTS+C concept with a THF for message hashing. Such modification naturally fits SPHINCS+ without degradation of the security level (see Section 5.2).

Definition 5.1 (Tweakable hash function; [HK22, 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 simplicity, we denote $\mathsf{Th}_{P,T}(M) = \mathsf{Th}(P,T,M)$.

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. Additionally, we have a unique tweak to compute the hash of the message with the counter. The tweak associated with the j-th function call in the i-th chain is defined as $T_{i,j}$. We also add a special tweak T^*_{addr} to denote the tweak for hashing the message for the WOTS instance that corresponds for the address $addr$, and we require that the hash with the counter has the desired fixed sum. If we use only one instance of WOTS+C then we omit the $addr$.

Chaining function $c^{j,k}(\mathsf{m}, i, \mathsf{Seed})$: The chaining function takes as inputs a message $\mathsf{m} \in \{0,1\}^n$, 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

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

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 procedure have access to the context information. A 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:

Algorithm 4: $\mathsf{KeyGen}(1^n)$

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 $\mathsf{pk}_i = c^{0,w-1}(\mathsf{sk}_i, i, \mathsf{Seed})$.

Algorithm 5: $\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$

1 Find $\mathsf{count} \in \{0,1\}^r$ such that $d = \mathsf{Th}(\mathsf{Seed}, T^*, \mathsf{m} \| \mathsf{count})$ 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 $\sigma_i = c^{0,a_i}(\mathsf{sk}_i, i, \mathsf{Seed})$.
4 Output $\sigma = (\sigma_1, \ldots, \sigma_\ell, \mathsf{count})$.

Algorithm 6: $\mathbf{V}(1^n, \mathsf{m}, \sigma, \mathsf{pk})$

1 Parse σ as $(\sigma_1, \ldots, \sigma_\ell, \mathsf{count})$.
2 Parse $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell)$.
3 Compute $d = \mathsf{Th}(\mathsf{Seed}, T^*, \mathsf{m} \| \mathsf{count})$.
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 $\mathsf{pk}_i = c^{a_i, w-1-a_i}(\sigma_i, i, \mathsf{Seed})$.

Security of WOTS-TW 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 [HK22] 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. In [HK22] the tweakable WOTS (WOTS-TW) variant used in SPHINCS+ is proven secure under this notion.

It is straight forward 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 so, also our reduction only needs the public key when the signature query is answered.

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 proofs we will also utilize 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.

In the security reduction we also follow the ideas from [HK22] and make use of a collection of tweakable hash functions which we call Th_λ. 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 specifying 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 in side the scheme structure. These challenges may depend on previous invocations of tweakable hash functions. If you don't have access to the public parameter that is used throughout the whole scheme at the moment of challenge placement you may struggle to obtain a proof. To address this problem the Th_λ oracle is introduced. It 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. 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.

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

5.2 SPHINCS+C security

To evaluate the security of SPHINCS+C one need to analyze an extension on the EU-naCMA model which is introduced in [HK22] and called $d-$EU-naCMA. This extension uses several instances of WOTS+C under one public parameter but with different tweaks. This is exactly the way WOTS-TW is used in SPHINCS+. A more detailed discussion and a sketch of the proof of WOTS+C in the $d-$EU-naCMA model can be found in Appendix D. By obtaining a $\mathsf{d}-$EU-naCMA security proof for WOTS+C one can just substitute WOTS-TW with our modification. This results in adding a S-TCR(+C) term to the security of SPHINCS+.

Combining with the result from [HK22] we get the following bound:

Theorem 5.2. *For parameters n, w, h, d, m, t, k as described in [BHKNRS19b] and l be the number of chains in WOTS-TW instances which is converted to WOTS+C the following bound can be obtained:*

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

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 $\mathcal{A}$.

The definitions of the properties can be found in Appendix F and functions are defined as follows:
$\mathbf{F} := \mathsf{Th}_1 : \mathcal{P} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$; $\mathbf{H} := \mathsf{Th}_2 : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{2n} \to \{0,1\}^n$;
$\mathsf{Th}_l : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{ln} \to \{0,1\}^n$; $\mathsf{Th}_k : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{kn} \to \{0,1\}^n$;
$\mathbf{PRF} : \{0,1\}^n \times \{0,1\}^{256} \to \{0,1\}^n$; $\mathsf{Th}^{+C} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^n \times \{0,1\}^r \to \{0,1\}^n$
$\mathbf{H_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^m$.
$\mathbf{PRF_{msg}} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$;

6 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 a matter 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 rational, 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 signing speed and security level of the original variants proposed in the NIST submission of SPHINCS+ [Aum+22]. Note that this is an arbitrary choice to allow for a simple comparison with SPHINCS+. In Section 8, we discuss the effect of the parameters on different optimizations, such as minimizing verification time or reducing the variance in signing speed.

6.1 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[4] and modified it to support SPHINCS+C.

[4] `http://sphincs.org/data/spx_parameter_exploration.sage`

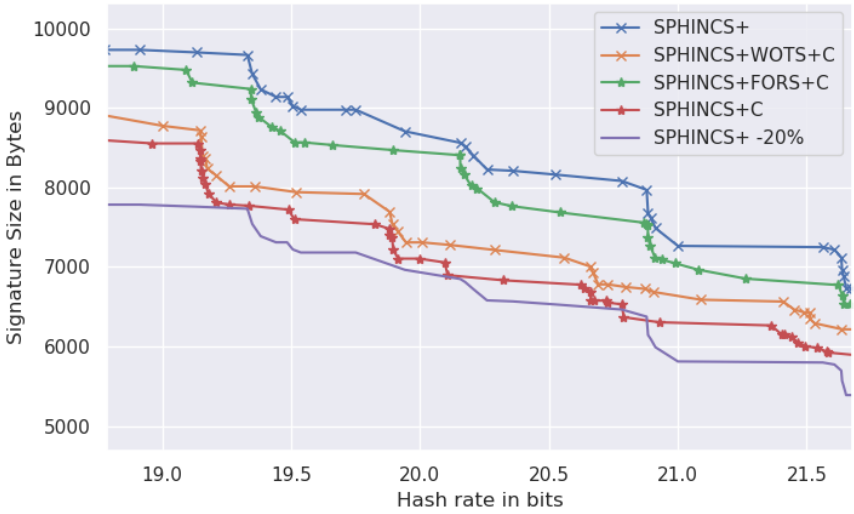

Figure 3: Hash rate (computational cost) to signature size in bytes trade-off curves for SPHINCS+ and our various compression options for (WOTS+C, FORS+C, and the full SPHINCS+C).

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.

Figure 3 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 over 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 Appendix G.To simplify the script and implementation, we did not incorporate the relatively complex interleaving trees optimization.

In Table 2 we provide a summary of the concert parameters of our proposed variants. For each variant, the signature generation time is similar to its counterpart in the SPHINCS+ NIST submission, while resulting in a smaller signature size.

6.2 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 3.3, 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

	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 2: 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.

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 4.3, 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 3 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}.

	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 3: 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.

7 Implementation

We based our implementation on the latest official version of the SPHINCS+ code.[5] 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.

7.1 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 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.

7.2 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+ [BHKNRS19a] 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.

[5] `https://github.com/sphincs/sphincsplus/`

	Key Generation		Signature		Verification		Size	
	SPHINCS+	Compressed	SPHINCS+	Compressed	SPHINCS+	Compressed	SPHINCS+	Compressed
SHAKE-128s	721.4	649.8 (−10%)	5398.0	4964.5 (−8%)	5.0	4.9 (−1%)	7856	7344 (−7%)
SHAKE-128f	10.8	9.7 (−11%)	256.3	232.4 (−9%)	16.4	14.2 (−13%)	17088	16012 (−7%)
SHAKE-192s	1068.6	962.3 (−10%)	9133.3	8283.6 (−9%)	8.5	7.5 (−12%)	16224	15392 (−6%)
SHAKE-192f	15.1	13.4 (−12%)	380.6	347.6 (−9%)	22.0	19.5 (−12%)	35664	33956 (−5%)
SHAKE-256s	652.6	648.2 (−1%)	7573.0	7367.8 (−3%)	11.4	11.3 (−1%)	29792	28580 (−5%)
SHAKE-256f	44.4	38.5 (−13%)	860.3	763.4 (−11%)	24.1	21.1 (−12%)	49856	47976 (−4%)

Table 4: Comparison of average running time in millions of cycles 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.

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, this will cause the signature to fail.

7.3 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. SPHINCS+ supports 3 different underlining hash constructions, we chose to benchmark the reference code for the "robust" SHAKE [Dwo15] variant based on the Keccak permutation [BDPVA09]. Our main goal is to compare the running time of SPHINCS+C with SPHINCS+ [BHKNRS19a]. For each parameter set we ran the key generation, 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 4 shows the average 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 5 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 4 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%

	Key Generation		Signature		Verification		Size	
	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C
SHAKE-128s	721.4	341.1 (−53%)	5398.0	4602.4 (−15%)	5.0	30.8 (+518%)	7856	6304 (−20%)
SHAKE-128f	10.8	8.6 (−20%)	256.3	237.6 (−7%)	16.4	12.0 (−27%)	17088	14904 (−13%)
SHAKE-192s	1068.6	501.2 (−53%)	9133.3	8107.8 (−11%)	8.5	45.0 (+429%)	16224	13776 (−16%)
SHAKE-192f	15.1	12.9 (−15%)	380.6	379.4 (−0%)	22.0	18.6 (−15%)	35664	33016 (−8%)
SHAKE-256s	652.6	432.0 (−34%)	7573.0	7339.4 (−3%)	11.4	36.1 (+218%)	29792	26096 (−13%)
SHAKE-256f	44.4	37.7 (−15%)	860.3	810.5 (−6%)	24.1	19.9 (−18%)	49856	46884 (−6%)

Table 5: Comparison of average running time in millions of cycles between the original SPHINCS+ NIST submission and and our new SPHINCS+C for different variants. Here the used SPHINCS+C parameters *differ* from those of SPHINCS+ and are chosen with a focus on size reduction.

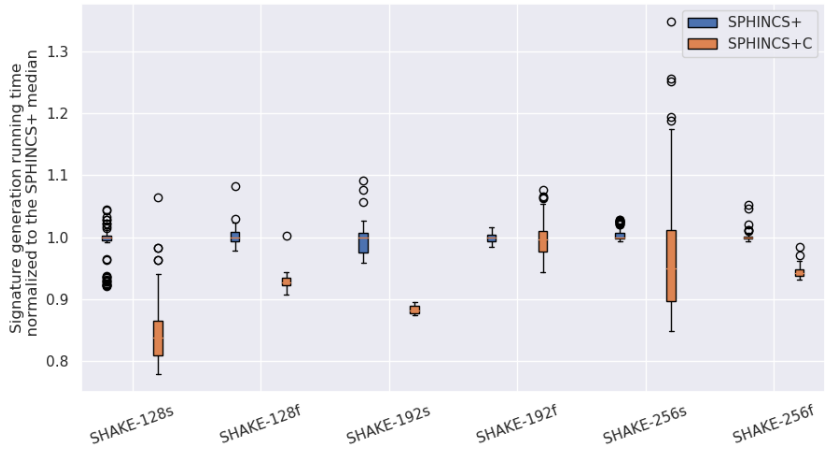

Figure 4: Comparison of variability in signature generation time between different SPHINCS+ and SPHINCS+C variants. The results of each variant include 100 repeated runs of randomized signatures and are normalized to the median of the SPHINCS+ variant.

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 5).

8 Discussion

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

8.1 Variance in signature generation time

The signing speed is not constant due to the search for good counters. In Section 6.2, 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.

8.2 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, only $w = 16$ is used for all the parameter sets.

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.

8.3 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.

8.4 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 [AE18], 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.

There are 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.

8.5 Constant verification time

Similar to [PZCMP21; ZCY22] 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 for every signature.

9 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.

As an example 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.

9.1 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.

9.2 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. This reduces the total verification time.

References

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. "Clarifying the subset-resilience problem". In: *IACR Cryptol. ePrint Arch.* (2017), p. 909 (cit. on pp. 2, 4, 9).

[AE18] Jean-Philippe Aumasson and Guillaume Endignoux. "Improving stateless hash-based signatures". In: *Cryptographers' Track at the RSA Conference*. Springer. 2018, pp. 219–242 (cit. on pp. 2, 9, 10, 19).

[Abr76] Morton Abramson. "Restricted Combinations and Compositions". In: *Fibonacci Quart* 14.5 (1976), pp. 439–452 (cit. on p. 7).

[Aum+22] Jean-Philippe Aumasson et al. "SPHINCS+ Submission to the NIST post-quantum project, v.3.1". In: (2022). URL: `http://sphincs.org/data/sphincs+-r3.1-specification.pdf` (cit. on pp. 3, 13, 28).

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak sponge function family main document". In: *Submission to NIST (Round 2)* 3.30 (2009), pp. 320–337 (cit. on p. 17).

[BHKNRS19a] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS$^+$ Signature Framework". In: *CCS*. ACM, 2019, pp. 2129–2146 (cit. on pp. 2, 4, 9, 11, 16, 17).

[BHKNRS19b] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS$^+$ Signature Framework". In: 2019, pp. 2129–2146. DOI: `10.1145/3319535.3363229` (cit. on pp. 13, 23, 33).

[BHRV20] Joppe W. Bos, Andreas Hülsing, Joost Renes, and Christine van Vredendaal. "Rapidly Verifiable XMSS Signatures". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2020), 137–168. DOI: `10.46586/tches.v2021.i1.137-168`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/8730` (cit. on pp. 2, 7).

[Ber+15] Daniel J. Bernstein et al. "SPHINCS: Practical Stateless Hash-Based Signatures". In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 368–397 (cit. on pp. 2, 4).

[Beu22] Ward Beullens. "Breaking Rainbow Takes a Weekend on a Laptop". In: *IACR Cryptol. ePrint Arch.* (2022), p. 214. URL: `https://eprint.iacr.org/2022/214` (cit. on p. 1).

[Dwo15] Morris J Dworkin. "SHA-3 standard: Permutation-based hash and extendable-output functions". In: (2015) (cit. on p. 17).

[GM17] Shay Gueron and Nicky Mouha. "SPHINCS-Simpira: Fast Stateless Hash-based Signatures with Post-quantum Security". In: *IACR Cryptol. ePrint Arch.* (2017), p. 645 (cit. on p. 2).

[Gol86] Oded Goldreich. "Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. 1986, pp. 104–110 (cit. on p. 2).

[HK22] Andreas Hülsing and Mikhail A. Kudinov. "Recovering the tight security proof of SPHINCS$^+$". In: *IACR Cryptol. ePrint Arch.* (2022), p. 346 (cit. on pp. 2, 4, 11–13, 23, 27–31).

[HRS16] Andreas Hülsing, Joost Rijneveld, and Fang Song. "Mitigating Multi-target Attacks in Hash-Based Signatures". In: 2016, pp. 387–416. DOI: `10.1007/978-3-662-49384-7_15` (cit. on pp. 24, 30).

[Hül13] 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*. 2013, pp. 173–188 (cit. on p. 3).

[Hül17] Andreas Hülsing. "WOTS+ - Shorter Signatures for Hash-Based Signature Schemes". In: *IACR Cryptol. ePrint Arch.* (2017), p. 965 (cit. on pp. 2, 4, 6).

[Lam79] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. 1979 (cit. on p. 2).

[Mer89] 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*. 1989, pp. 218–238 (cit. on p. 2).

[Mer90] Ralph C. Merkle. "A Certified Digital Signature". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. New York, NY: Springer New York, 1990, pp. 218–238. ISBN: 978-0-387-34805-6 (cit. on p. 5).

[PZCMP21] Lucas Pandolfo Perin, Gustavo Zambonin, Ricardo Custódio, Lucia Moura, and Daniel Panario. "Improved constant-sum encodings for hash-based signatures". In: *Journal of Cryptographic Engineering* 11.4 (2021), pp. 329–351 (cit. on pp. 2, 19).

[RR02a] 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*. 2002, pp. 144–153 (cit. on p. 2).

[RR02b] Leonid Reyzin and Natan Reyzin. "Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying". In: *ACISP*. Vol. 2384. Lecture Notes in Computer Science. Springer, 2002, pp. 144–153 (cit. on p. 2).

[ZCY22] Kaiyi Zhang, Hongrui Cui, and Yu Yu. "SPHINCS-α: A Compact Stateless Hash-Based Signature Scheme". In: *IACR Cryptol. ePrint Arch.* (2022), p. 59 (cit. on pp. 2, 19).

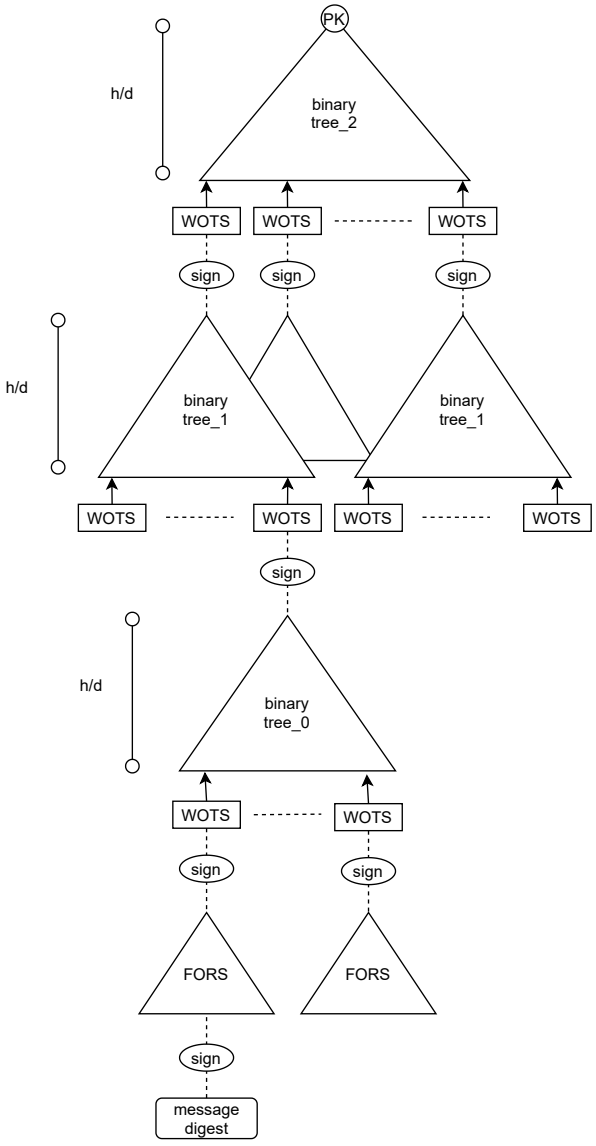

Figure 5: Example of a SPHINCS$^+$ structure

A Brief description of SPHINCS+

Here we give a brief description of the SPHINCS$^+$ signature scheme from [HK22]. This description is the same as in the referred paper and presented here just to make the paper self-sufficient.

An example of the SPHINCS$^+$ structure is shown in fig. 5.

A detailed description can be found in [BHKNRS19b]. The public key consists of two n-bit values: a random public seed **PK**.seed and the root of the top tree in the hypertree structure. **PK**.seed is used as a first argument for all of the tweakable hash functions calls. The private key contains two more n-bit values **SK**.seed and **SK**.prf. We discuss the main parts of SPHINCS$^+$. First we describe the addressing scheme. As SPHINCS$^+$ uses Tweakable Hash functions, different tweaks are required for all calls to Tweakable Hash functions. The tweaks are instantiated by the addresses. The address is a 32 byte value. Address coding can be done in any convenient way. Each address has a prefix that denotes to which part of the SPHINCS$^+$

structure it belongs. We denoted this prefix as **ADRS** in previous sections.

Then we need to discuss 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 the internal nodes of the tree are calculated from the children of that node using a Tweakable Hash function. A leaf of a binary tree is the output of a Tweakable Hash function that takes the elements of a WOTS-TW public key as input.

Binary trees and WOTS-TW signature schemes are used to construct a hypertree structure. WOTS-TW instances are 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 instance. 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) which consist 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 Tweakable Hash function. The resulting value will be the FORS public key.

FORS signature. A message of ka bits is divided into k lines of a bits. Each of these lines 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 there is a match, it is said that the signature was verified. 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 $\mathbf{SK}.\mathsf{prf}$ and the message. This function can be made non-deterministic initializing the value $\mathsf{OptRand}$ 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: $(\mathsf{MD}||\mathsf{idx}) = \mathbf{H_{msg}}(\mathbf{R}, \mathbf{PK}.\mathsf{seed}, \mathbf{PK}.\mathsf{root}, M)$.

The signature consists of the randomness $\mathbf{R}$, the FORS signature (under idx from $\mathbf{H_{msg}}$) 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 test this chain, the verifier iteratively reconstructs the public keys and tree roots until it gets the root of the top tree. If this matches the root given in the SPHINCS$^+$ public key, the signature is accepted.

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) [HRS16]. 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 B.1 (Multi-target extended target collision resistance (m-eTCR) [HRS16])**.** *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

Algorithm 7: M-ETCR reduction

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

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

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

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

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 7. 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

Algorithm 8: 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** $d^*, \tilde{\sigma}$

challenge. Assume that the probability of eventually hitting line 11 in algorithm 7 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 8 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 3.3. This concludes the proof. □

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

In this section we give a full proof for theorem C.2. To do we introduce several notions. First is a special target collision resistance for tweakable hash functions.

Definition C.1 (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.*

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

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 a tweak. In S-TCR(Prop) this is handled by $\mathcal{O}^{\mathsf{Prop}}$. In appendix E, 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 use 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. 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 $k = 2^30$ and $p_\nu \approx 0.015$ 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 one 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 can not depend its behavior on the existence of a fitting counter.

Theorem C.2. *Let WOTS-TW be a signature scheme as defined in [HK22]. Let* Th *be a tweakable hash function. Then the insecurity of WOTS+C using* Th *against one-time EU-naCMA attacks is bounded by*

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

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 9. One can see that similar to algorithm 7 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:

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

Now we limit the probability of the adversary in succeeding in GAME.1. To do so we construct another algorithm. In algorithm 10 we see that if $d^* \neq d$ than the forgery for WOTS+C results in the 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 3.3. This concludes the proof. □

In [HK22], it was shown that

$$\mathrm{InSec}^{\text{EU-naCMA}}(\text{WOTS-TW}; t, 1) \leq$$

Algorithm 9: 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 $\perp$

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

with $\widetilde{t} = t + \mathsf{len} \cdot w$, where time is given in number of Th evaluations. See Appendix F 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:

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

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

D d-EU-naCMA of WOTS+C

To analyze SPHINCS+C we have to analyze an extension of EU-naCMA to multiple instances with the same public seed. The same approach was used in the [HK22] paper. In that scenario we have multiple instances of WOTS+C under the same public seed. Each instance uses different tweaks for their hash function calls. To distinguish different instances of WOTS+C we use **ADRS**, which defines a unique prefix for tweaks used in a particular instance of WOTS+C (see [Aum+22; HK22] for more details).

We use the d-EU-naCMA security model introduced in [HK22].

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 $j \in [1, d] \wedge [\mathrm{Vf}(\mathsf{PK}_j, \sigma^\star, M^\star) = 1] \wedge [M^\star \neq M_j] \wedge$ $[\mathbf{DIST}(\{\mathbf{ADRS}_i\}_{i=1}^d)] \wedge [\forall \mathbf{ADRS}_i \in Q, \mathbf{ADRS}_i \notin T' = \{\mathsf{adrs}(\mathsf{T_i})\}_{i=1}^p]$,

Algorithm 10: WOTS-TW reduction

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

1 Obtain a signature query m from the adversary $\mathcal{A}$
2 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
3 Send d as a signature query for C
4 Obtain a signature for d: $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$
5 Set $\sigma' = (\sigma_1, \ldots, \sigma_{\mathsf{len}_1 - z}, seed)$
6 Send σ' to the adversary $\mathcal{A}$
7 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
8 Obtain a forgery (m^*, σ^*) from $\mathcal{A}$
9 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
10 **return** $d^*, \tilde{\sigma}$

where $\mathbf{DIST}(\{\mathbf{ADRS}_i\}_{i=1}^d)$ outputs 1 iff all arguments are distinct and 0 otherwise, $\mathsf{adrs}(\cdot)$ returns a prefix of a tweak.

The following theorem can be proven by generalization of the proof of theorem C.2.

Theorem D.1. *Let n, $w \in \mathbb{N}$ and $w = poly(n)$. Then the following inequality holds:*

$$\mathrm{InSec}^{\mathsf{d\text{-}EU\text{-}naCMA}}(\mathsf{WOTS{+}C}; t, d) < \mathsf{InSec}^{\text{S-TCR}(+\text{C})}(\mathsf{Th} \in \mathsf{Th}_\lambda; t', d) + \mathrm{InSec}^{\mathsf{d\text{-}EU\text{-}naCMA}}(\mathsf{WOTS\text{-}TW}; t', d) \quad (1)$$

with $\widetilde{t} = t + d \cdot \mathsf{len} \cdot w$, where time is given in number of Th evaluations and $t \approx t'$.

Proof sketch. Assume GAME.0 is the original $\mathsf{d}-$EU-naCMA game. GAME.1 differs from GAME.0 in that the game is considered lost if one can extract a collision under Th from a forgery and signature responses. This collision must be for $T^*_{\mathbf{ADRS}}$, where $\mathbf{ADRS}$ is an address of one of the WOTS+C instances. The differences in these two games should be bounded by the S-TCR(+C) property of $\mathsf{Th} \in \mathsf{Th}_\lambda$. To do so a signing query is handled the following way: the message and the tweak are sent to the S-TCR(+C) challenger via $\mathcal{O}^{+C}$ to obtain a good hash for WOTS+C. Then using Th_λ one should calculate a signature and a local public key for this specific WOTS+C instance. If there is a forgery that contains a collision under Th at position $T^*_{\mathbf{ADRS}}$ then this is a solution for S-TCR(+C). Here we can not use Public seed to construct a WOTS+C signature as in theorem C.2, because there might be more signature queries, so we need to use our S-TCR(+C) challenger several times. We obtain public seed only after all the signing queries are done, until that we use Th_λ.

To limit the success probability of the adversary in GAME.1 one simply utilizes a $\mathsf{d}-$WOTS-TW challenger. Every signing query from WOTS+C adversary is handled the following way: first the message and the tweak are used to search for a good WOTS+C hash using Th_λ. That hash is then transferred to the signing oracle of the $\mathsf{d}-$WOTS-TW challenger. On obtaining a signature from WOTS-TW challenger we can extract the signature for the WOTS+C. By the same reasoning as in theorem C.2 if there occurs a forgery for WOTS+C without a collision under T^* we can extract a signature forgery for WOTS-TW.

This concludes the sketch of the proof. □

E S-TCR(+C) security

In this section we analyze the complexity of generic attacks against the S-TCR(+C) property. We do this, proving a lower bound on the query complexity of a quantum adversary against the S-TCR(+C) property for a random function. To do so we reprogram a quantum-accessible oracle. The analysis here makes heavy use of the tools used in [HK22]. Consider the following games for a two staged adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

Game 0: $P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(\cdot,\cdot,\cdot),\ \mathcal{O}^{+C}(P,\cdot,\cdot)}(\ ); b \leftarrow \mathcal{A}_2^{\mathsf{Th}(\cdot,\cdot,\cdot)}(Q,S,P),\ b \in \{0,1\}$, where $Q = \{(T_i, M_i)\}_{i=1}^{p}$ - are $\mathcal{A}$'s queries to $\mathcal{O}^{+C}(P,\cdot,\cdot)$. Here $\mathcal{A}$ gets quantum access to Th and classical access to $\mathcal{O}^{+C}(P,\cdot,\cdot)$.

Game 1: Adversary gets access to the tweakable hash function. It makes q queries to the tweakable hash function Th and p queries $\{T_i, M_i\}_{i=1}^{p}$ to the oracle $\mathcal{O}^{+C}(P,\cdot,\cdot)$, which specify the challenges. Such queries are handled differently from Game 0. In this case, instead of responding with $\mathcal{O}^{+C}(P, T_i, M_i)$ a value $g(T_i, M_i, ||j_i) = y_i$ is returned, where $g : \mathcal{T} \times \{0,1\}^{\alpha} \to \{0,1\}^{n}$ is a random function and $j_i \in \{0,1\}^{r}$ is a value that leads to y_i which satisfies the WOTS+C properties. We call this new oracle $\mathcal{O}_g^{+C}$ After all p challenge queries are made the Th is reprogrammed into Th' using a random function g the following way:

$$\mathsf{Th}'(pp,t,x)\begin{cases}\text{if } (pp = P): \text{ Return } g(t,x)\\ \text{Return } \mathsf{Th}(pp,t,x)\end{cases}$$

So we have the following game:

$P \leftarrow_{\$} \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(\cdot,\cdot,\cdot),\ \mathcal{O}_g^{+C}(\cdot,\cdot)}(\ ); b \leftarrow \mathcal{A}_2^{\mathsf{Th}'(\cdot,\cdot,\cdot)}(Q,S,P),\ b \in \{0,1\}$, where $Q = \{(T_i, M_i)\}_{i=1}^{p}$ - queries to $\mathsf{Th}(P,\cdot,\cdot)$. Where $\mathcal{A}$ gets quantum access to Th and Th' and classical access to $\mathcal{O}_g^{+C}(\cdot,\cdot)$.

According to Lemma 4 from [HK22]:

Lemma E.1. $|\Pr[\mathcal{A}(G_0) = 1] - \Pr[\mathcal{A}(G_1) = 1]| \leq \delta$,
where $\delta = 4q^2/|\mathcal{P}|$

Let us recall some definitions that are useful in the analysis of S-TCR(+C).

Definition E.2 ([HRS16])**.** *Let* $\mathcal{F} := \{f : \{0,1\}^m \to \{0,1\}\}$ *be the collection of all boolean functions on* $\{0,1\}^m$*. Let* $\lambda \in [0,1]$ *and* $\varepsilon > 0$*. Define a family of distributions* D_λ *on* $\mathcal{F}$ *such that* $f \leftarrow_{\$} D_\lambda$ *satisfies*

$$f : x \to \begin{cases} 1 & \textit{with prob. } \lambda \\ 0 & \textit{with prob. } 1-\lambda \end{cases}$$

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

According to [HRS16] we define $\mathsf{Avg} - \mathsf{Search}_\lambda$ as a problem of finding an x such that $f(x) = 1$, where $f \leftarrow D_\lambda$. For any quantum algorithm $\mathcal{A}$ that makes q queries, we define

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

Theorem E.3 ([HRS16])**.** $\mathrm{Succ}_\lambda^q(\mathcal{A}) \leq 8\lambda(q+1)^2$ *holds for any quantum algorithm* $\mathcal{A}$ *with* q *queries.*

Now we are ready to prove the following lemma.

Lemma E.4. *Let* Th *be a tweakable hash function. Any quantum adversary* $\mathcal{A}$ *that solves* S-TCR(+C) *making* q *quantum queries to* Th *can be used to construct a quantum adversary* $\mathcal{B}$ *such that* $Succ^{\mathrm{Avg-Search}_{1/2^n}}(B) + 16q^2/|\mathcal{P}| \geq \mathrm{Succ}_{\mathsf{Th},p}^{\text{SM-TCR}}(\mathcal{A})$*, where* $\mathcal{B}$ *is a* $2q$*-query adversary. So we get*

$$\mathrm{Succ}_{\mathsf{Th},p}^{\text{S-TCR(+C)}}(\mathcal{A}) \leq \frac{32(q+1/2)^2}{2^n} + 16q^2/|\mathcal{P}|$$

Proof. For this proof we define two games:

Game 0: Original S-TCR(+C) game.

Game 1: The modifications defined in lemma E.1.

Algorithm 11: AvgSearch to S-TCR

Input : $f \leftarrow D_\lambda : [p] \times \{0,1\}^\alpha \to \{0,1\}$, SM-TCR adversary $\mathcal{A}$
Output : $b' \in \{0,1\}^n$

1 Generate a random tweakable hash function Th.
2 Give oracle access to Th for $\mathcal{A}_1$, handle the queries $\{T_i, M_i\}$ to the challenge oracle $\mathcal{O}^{+C}(P,\cdot,\cdot)$ by generating random values until one of them satisfies the WOTS+C restrictions. Call this value MD_i and let the iteration on which this value have been generated be j_i for i-th query. Respond using $\{MD_i, j_i\}$.
3 Generate random functions $g_i : \{0,1\}^\alpha \to \{0,1\}^n / MD_i,\ i \in [1,p]$.
4 After $\mathcal{A}_1$ stops construct $g : \mathcal{T} \times \{0,1\}^\alpha \to \{0,1\}^n$ using a random function $g' : \mathcal{T} \times \{0,1\}^\alpha \to \{0,1\}^n$ the following way: $g(t,m) : \begin{cases} \text{if } (t = T_i \wedge m = M_i || j_i) & \text{Return } MD_i \\ \text{if } (t = T_i \wedge f(i,m) = 1) & \text{Return } MD_i \\ \text{if } (t = T_i) & \text{Return } g_i(m) \\ \text{else} & \text{Return } g'(t,m) \end{cases}$
5 and a tweakable hash function Th' as $\mathsf{Th}'(pp,t,x) : \begin{cases} \text{if } (pp = P) : \text{ Return } g(t,x) \\ \text{Return } \mathsf{Th}(pp,t,x) \end{cases}$
6 Give oracle access to Th' for $\mathcal{A}_2$.
7 Output $\mathcal{A}_2^{\mathsf{Th}'}(Q,S,P)$

The difference in success probabilities of $\mathcal{A}$ in these two games can be limited by lemma E.1.

Now we have to limit the success probability in Game 1. This is where $\mathrm{Avg-Search}_{1/2^n}$ problem comes in handy. Let $\mathcal{B}$ be defined as algorithm 11. Whenever $\mathcal{A}$ succeeds in finding a collision the result is actually a solution for the $\mathsf{Avg-Search}_{1/2^n}$ problem. Hence, we get

$$\begin{aligned} &\Pr[P \leftarrow_\$ \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(\cdot,\cdot,\cdot),\mathcal{O}_g^{+C}(\cdot,\cdot)}(\,); \\ &(j,M) \leftarrow \mathcal{A}_2^{\mathsf{Th}'}(Q,S,P) : \\ &\mathsf{Th}'(P,T_i,M_i||i_j) = \mathsf{Th}'(P,T_j,M||counter) \wedge M \neq M_j \\ &\wedge \mathbf{DIST}(\{T_i\}_{i=1}^p)] \leq \mathrm{Succ}_{1/2^n}^{2q}(\mathcal{B}) \leq \frac{32(q+1/2)^2}{2^n} \end{aligned}$$

Combining the obtained result we get the desired bound.

□

F Properties definitions

In this section we present the security definitions of Th from [HK22]. We add them just for the paper to be self-contained.

Definition F.1 (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.*

$$\mathsf{Succ}_{\mathsf{Th},p}^{\text{SM-TCR}}(\mathcal{A}) = \Pr[P \leftarrow_\$ \mathcal{P}; S \leftarrow \mathcal{A}_1^{\mathsf{Th}(P,\cdot,\cdot)}(\,);$$

$$(j, M) \leftarrow \mathcal{A}_2(Q, S, P) : \mathsf{Th}(P, T_j, M_j) = \mathsf{Th}(P, T_j, M)$$
$$\wedge M \neq M_j \wedge \mathbf{DIST}(\{T_i\}_{i=1}^p)]$$

Definition F.2 (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.*

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

Definition F.3 (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.*

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

Definition F.4 (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 F.5 (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*

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

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 F.6 ($\mathsf{SP}_{P,T}$)**.** *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:*

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

where $\mathsf{Th}_{P,T}^{-1}$ *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 [BHKNRS19b] for a tweakable hash function. The intuition behind this notion is that the adversary should be unable to find a preimage such that doesn't have a second preimage.

Definition F.7 (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.*

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

where

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

G Parameter-evaluation Sage script

```
#This script is based on the paramter finding script
#of SPHINCS+ from http://sphincs.org/data/spx_parameter_exploration.sage

import collections
tsec = 128
#tsec = 192
#tsec = 256
maxsigs = 2 ^ 64 # at most 2^72
if tsec == 128:
    maxsigbytes = 18000 # Don't print parameters if signature size is larger
else:
    maxsigbytes = 64000
MaxHash = 23
CounterBytes = 4

USE_WOTS_CHECKSUM_COMPRESS = False
USE_WOTS_DIGITS_COMPRESS = False
USE_FORS_COMPRESS = False

#### Don't edit below this line ####
#### Generic caching layer to save time

def get_file_name():
    file_str = 'Results'
    if USE_WOTS_CHECKSUM_COMPRESS:
        file_str += 'ChecksumWOTS'
    if USE_WOTS_DIGITS_COMPRESS:
        file_str += 'DigitsWOTS'
    if USE_FORS_COMPRESS:
        file_str += 'CompressFORS'
    return file_str

class memoized(object):
  def __init__(self, func):
    self.func = func
    self.cache = {}
    self.__name__ = 'memoized:' + func.__name__

  def __call__(self, *args):
    if not isinstance(args, collections.Hashable):
      return self.func(*args)
    if not args in self.cache:
      self.cache[args] = self.func(*args)
    return self.cache[args]

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

# Pr[FORS forgery given that exactly r sigs hit the leaf] = (1-(1-1/F(2^b))^r)^k
```

```
@memoized
def forgeryprob(b, r, k):
    if k == 1:
        return 1-(1-1/F(2 ^ b)) ^ r
    return forgeryprob(b, r, 1)*forgeryprob(b, r, k-1)
    #return min(1,F((r/F(2**b)))**k)

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

def cost_wots_zero_chain(w):
    return log(w, 2)

# https://www.fq.math.ca/Scanned/14-5/abramson.pdf
#[1] Morton Abramson. "Restricted Combinations and Compositions." In: Fibonacci Quart. $14.5$
    (1976), pp. 439-452. issn: 0015-0517.
@memoized
def restricted_compositions(S, l, w):
    res = 0
    #Fix sum as we count from 0 to k-1, and the results if from 1 to k
    S = S + l
    for j in range(l + 1):
        t = S - j * w - 1
        if t < 0:
            continue
        res += binomial(l, j) * binomial(t, l - 1) * (-1) ** j
    return res

def findMaxS(l, w):
    return l * (w - 1) / 2

@memoized
def get_cost_wots_cs(hashbytes, w):
    l = int(hashbytes * 8 / log(w, 2)) # casting to int returns nearest integer towards zero
        (floor for positve numbers)
    zero_bits = hashbytes * 8 - l * log(w, 2)
    S = int(findMaxS(l, w))
    e_sum_cost = F(F(restricted_compositions(S, l, w)) / F(w ** l))
    e_wotsc_p = e_sum_cost * ((1 / 2) ** zero_bits)
    return (float(-log(e_wotsc_p,2)), l, float(log(l*w, 2)), S)


def ld(r):
    return F(log(r)/log2)

def run_script():
    hashbytes = ceil(tsec/8) # length of hashes in bytes
    w_list = [16, 256]
    if USE_WOTS_CHECKSUM_COMPRESS:
        w_list = [16, 32, 64, 128, 256]
    filename = get_file_name() + '%d' % hashbytes
    f = open(filename, 'wt')
    global F
```

```
global FDef
F = RealIntervalField(tsec+100)
FDef = RealField()
if USE_FORS_COMPRESS:
    sigmalimit = F(2 ^ (-tsec+MaxHash))
    donelimit = 1-sigmalimit/2 ^ (20+MaxHash)
else:
    sigmalimit = F(2 ^ (-tsec))
    donelimit = 1-sigmalimit/2 ^ 20
sigmaLowerLimit = F(2 ^ (-tsec-MaxHash))

s = log(maxsigs, 2)
# Iterate over total tree height
for h in range(s-8, s+20):
    print('#starting-H %d' % (h))
    leaves = 2 ^ h
    # Iterate over height of FORS trees
    for b in range(3, 24):
        # Iterate over number of FORS trees
        for k in range(1, 64):
            sigma = 0
            r = 1
            done = qhitprob(leaves, maxsigs, 0)
            while done < donelimit:
                t = qhitprob(leaves, maxsigs, r)
                sigma += t*forgeryprob(b, r, k)
                if sigma > sigmalimit:
                    break
                done += t
                r += 1
            sigma += min(0, 1-done)
            if sigma > sigmalimit:
                continue
            # means we have more efficent options with smaller tree
            if sigma < sigmaLowerLimit: continue
            sec = ceil(log(sigma, 2))
            remove_tree_fors_bit = FDef(max(tsec+log(sigma, 2), 0))
            fors_work = (k*2 ^ (b+1)) # cost of FORS
            if fors_work > 2 ^ MaxHash:
                break
            for d in range(3, floor(h/2)):
                if (h) % d == 0 and h <= 64+(h/d):
                    wots_tree_levels = (h)/d
                    for w in w_list:       # Try different Winternitz parameters
                            compress_wots_cs_cost, wots_digits, wots_work_bit, target_sum =
                                get_cost_wots_cs(hashbytes, w)
                            max_wots_zero_chain = ceil(
                                (MaxHash-compress_wots_cs_cost)/cost_wots_zero_chain(w))
                            wots_work = 2 ^ wots_work_bit
                            if not USE_WOTS_CHECKSUM_COMPRESS:
                                wots_digits = wotschains(8*hashbytes, w)
                                wots_work = (wots_digits*w)
                                compress_wots_cs_cost = 0
                                max_wots_zero_chain = 0
                            if not USE_WOTS_DIGITS_COMPRESS:
                                max_wots_zero_chain = 0
```

```
                            for wots_zero_chain in range(max_wots_zero_chain + 1):
                                wots_compress_cost_bit = compress_wots_cs_cost + \
                                    wots_zero_chain * \
                                        cost_wots_zero_chain(w)
                                wots = wots_digits-wots_zero_chain
                                sigsize = ((b+1)*k+h+wots*d+1)*hashbytes
                                if USE_WOTS_CHECKSUM_COMPRESS:
                                    sigsize += CounterBytes*(d)
                                if USE_FORS_COMPRESS:
                                    sigsize += CounterBytes
                                # Rough speed estimate based on #hashes
                                speed = fors_work + d * \
                                    (2 ^ (wots_tree_levels)*(wots_work+1)) + d * \
                                     2**wots_compress_cost_bit + 2**remove_tree_fors_bit
                                hash_count = ld(speed)
                                if(sigsize < maxsigbytes and hash_count < MaxHash):
                                    f.write('h: %d, d: %d, log(t): %d, k: %d, w: %d, \
                                        sigsize: %d, hashbit: %f, sigma %f,
                                            wots_compress_cost_bit: %d, \
                                           remove_tree_fors_bit: %f, fors_bit: %f\n' % (
                                        h, d, b, k, w, sigsize, hash_count, \
                                            FDef(ld(sigma))-remove_tree_fors_bit,
                                                wots_compress_cost_bit, \
                                               remove_tree_fors_bit, ld(fors_work)))
    f.close()

for tsec in [128, 192, 256]:
    if tsec == 128:
        maxsigbytes = 18000 # Don't print parameters if signature size is larger
    else:
        maxsigbytes = 64000
    MaxHash = 23
    for USE_WOTS_CHECKSUM_COMPRESS in [False, True]:
        for USE_WOTS_DIGITS_COMPRESS in [False, True]:
            for USE_FORS_COMPRESS in [False, True]:
                    if USE_WOTS_DIGITS_COMPRESS and not USE_WOTS_CHECKSUM_COMPRESS:
                        continue
                    print('Start ', tsec, USE_WOTS_CHECKSUM_COMPRESS,
                          USE_WOTS_DIGITS_COMPRESS, USE_FORS_COMPRESS)
                    run_script()
```

H Time estimations for WOTS+C signature evaluation

```
from sage.all import RealField, binomial, ceil, log

F = RealField(128)

# https://www.fq.math.ca/Scanned/14-5/abramson.pdf

#[1] Morton Abramson. "Restricted Combinations and Compositions." In: Fibonacci Quart. $14.5$
    (1976), pp. 439-452. issn: 0015-0517.
def restricted_compositions(S, l, w):
    res = 0
    #Fix sum as we count from 0 to k-1, and the results if from 1 to k
    S = S + l
    for j in range(l + 1):
        t = S - j * w - 1
        if t < 0:
            continue
        res += binomial(l, j) * binomial(t, l - 1) * (-1) ** j
    return res

def findMaxS(l, w):
    return l * (w - 1) / 2

def get_cost_wots_cs(hashbytes, w):
    l = int(hashbytes * 8 / log(w, 2)) # casting to int returns nearest integer towards zero
        (floor for positve numbers)
    zero_bits = hashbytes * 8 - l * log(w, 2)
    S = int(findMaxS(l, w))
    e_sum_cost = F(F(restricted_compositions(S, l, w)) / F(w ** l))
    e_wotsc_p = e_sum_cost * ((1 / 2) ** zero_bits)
    return (float(-log(e_wotsc_p,2)), l, float(log(l*w, 2)), S)

def bound_tries(p_trial, p_bound):
    p_trial = F(p_trial)
    num_trials = log(F(p_bound))/log(F(1-p_trial))
    return num_trials

def bound_tries_d(p_trial, p_bound, d):
    i = int(d) - 1
    d_succ_in_n = F(0)
    p_bound = F(p_bound)
    p_succ_bound = 1- p_bound
    p_trial = F(p_trial)
    while p_succ_bound > d_succ_in_n:
        d_succ_in_n += binomial(i, d - 1) * (p_trial) ** d * (1 - p_trial) ** (i + 1 - d)
        i += 1
    return i

def get_sig_gen_factor(p_bound, hashbytes, w, d, total_hash_bit, fors_tp):
    n = hashbytes * 8
    l = int(n / log(w, 2))
    zero_bits = n - l * int(log(w, 2))
    S = findMaxS(l, w)
    probability = F(F(restricted_compositions(S, l, w)) / F(w ** l)) * ((1 / 2) ** zero_bits)
```

```
    d_expected = F(d / probability) # expected number of trials until d successes
    d_max_runs = bound_tries_d(probability, p_bound, d)
    max_extra_wots = d_max_runs - d_expected
    total_hash = 2 ** total_hash_bit
    t_p = F(2 ** fors_tp)
    F_probability = F(1 / t_p)
    F_expected = F(1 / F_probability)
    max_extra_fors = bound_tries(F_probability, p_bound) - F_expected
    max_run_factor = (max_extra_wots + max_extra_fors + total_hash) / total_hash
    return max_run_factor
```