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

Eyal Ronen
eyal.ronen@cs.tau.ac.il
TAU

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

June 16, 2022

Abstract

The SPHINCS+ [CCS '19] proposal is one of the alternate candidates for digital signatures in 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 SPHICS+ supports a tradeoff between signature size and the computational cost of the signature, further reducing the signature size (below the small variants) results in a prohibitively high computational cost for the signer (as well as the verification cost).

This paper presents several novel methods for further compressing the signature size while requiring negligible added computational costs for the signer and faster verification time. Moreover, our approach enables a much more efficient tradeoff 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.

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 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 and benchmark results.

Keywords: hash based signatures;post-quantum security

Contents

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. Other signature schemes are either not post-quantum secure (e.g., discrete-log based), have large signatures (e.g., lattice-based), or rely on assumptions that have not been studied and are often prone to new attacks [Beu22]. Thus, as digital signatures are a crucial cryptographic tool in practice, reducing the size of hash-based signatures is of the utmost importance.

Hash-based signature has a long history starting from 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 [PZCMP21; ZCY22; AE18; GM17; RR02b; AE17; Hül17]. Most notable is the suggested proposal SPHINCS+ [BHKNRS19], which is one of the finalist (as an alternative candidate) in NIST's post-quantum standardization process for digital signatures. The SPHINCS+ scheme introduces a new few-time signature scheme called FORS, uses WOTS+ [Hül17] as the one-time signature component, and a new security analysis framework that uses "tweakable hash function". The SPHINCS+ scheme is generally considered the current state-of-the-art of stateless hash-based signatures. It supports a tradeoff 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 consider 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 running-time of the signature generation. Furthermore, our scheme does not introduce complicated security analysis, and the security reduces back to the security of each component of the original SPHINCS+ scheme.

Our main contributions and new techniques are summarized in the following fronts:

- Improved one-time signatures (WOTS): We introduce a new variant of the Winternitz one-time signature scheme, which we refer to as WOTS+C. The scheme reduces the number of chains in

	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%)

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.

the 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 given in Section 3.

- Improved few-time signatures (FORS): We introduce a new variant of the FORS scheme, called FORS+C. As in the WOTS case, our variant reduces the signature size and (slightly) reduces the verification time while maintaining the same signing time (and key-generation time). The scheme is given in Section 4.
- Improved state-less hash signature scheme: We propose a novel variant of SPHINCS+, called SPHINCS+C that uses WOTS+C and FORS+C. The new one-time and few-time signatures we use (along with additional improvements) allow for reducing the signature size of each component and a more comprehensive range of tradeoffs for the whole signature scheme. This enable us to find for better tradeoffs and reduce the overall size of the signature, while maintaining the signature generation time. This is described in Section 5.

We provide a security proof of our scheme, a reference implementation based on the code of SPHINCS+,[1] and propose several variants of recommended parameters sets that we benchmarked. Table 1 provides a comparison for the signature sizes of between the SPHINCS+ variants submitted to round 3 of the NIST competition [Aum+22], and our proposed variants that maintain a comparable (and sometimes slightly better) signature generation time.

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 the signature generation time (see benchmark at Table 3).

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+, a one-time signature scheme, along with its security analysis. In Section 4, we describe our FORS+C scheme, a new variant of the FORS few-time signature, along with its security analysis. In Section 5, we describe SPHINCS+C, which assembles the former two building blocks into the SPHINCS+ scheme while finding the optimal tradeoff of parameters. In Section 6, we describe our implementation and

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

provide a comparison with the SPHINCS+ scheme (in terms of running-time and signature size). In Section 7 we provide a general discussion of further considerations that are relevant to our scheme, and directions for future work.

The Sage script we used for parameter-evaluation for the SPHINCS+C variants is provided in Appendix A.

2 Background

In this subsection, we provide relevant background on the SPHINCS+ signature scheme, before we describe our improvements.

2.1 WOTS and WOTS+

The Winternitz signature scheme (WOTS) and its variants (e.g., WOTS+) are one-time signature scheme [Hül13]. The core idea of WOTS (and its variants) is to use w 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. The WOTS+ scheme use a special mode of iteration which enables tight security proof (without the need of collision resistant).

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 defined the length of the chains, and is usually set to 4, 16 or 256. Define

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

Moreover, let $\mathsf{len} = \mathsf{len}_1 + \mathsf{len2}$, which represents the number of n-bit values in an uncompressed WOTS+ private key, public key, and signature.

To sign a message m, we interpret m as len_1 integers a_i each between 0 and $w-1$. We compute a checksum $C = \sum_{i=1}^{\mathsf{len}_1}(w-1-a_1)$, represented as string of len_2 base-w values $C = (C_1, \dots, C_{\mathsf{len}_2})$. This checksum is also signed and is what provides security of the scheme. Using these len integers as chain lengths, the chaining hash function H is applied to the private key elements. The result is a len values, each is a n-bit string that consist of the signature. The verifier recomputes the checksum, and chain lengths, and then applies H to complete each chain. This leads to the chain heads, which are hashed together to provide a single n bit string serving as the public-key.

The WOTS+ has a similar structure to WOTS, where the main difference is in how the chain is computed. Instead of iterating over the same hash function, WOTS+ defines a new chaining function which involves random masking values r and a family of hash functions f_k. In every iteration, the function first takes the bitwise xor of the intermediate value and bitmask r and evaluates f_k on the result afterwards. This enables them to get tight security analysis using weak (inversion) properties of the hash functions. See [Hül17] for more details and the security proofs.

2.2 FORS

While one-time signature schemes lose all security after more than one signature, few-time signatures (FTS) can maintain some level of security even after a small number of signatures are signed. The security of the 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 SPHINS+[BHKNRS19], 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 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 collision of two or more indices in the single tree used by HORST.

2.3 SPHINCS+

The SPHINCS+ [BHKNRS19] 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 of the public keys of WOTS+ signature. The WOTS+ signatures in the leaves of the lowest layer of the trees are used to sign the roots of FORS signature.

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 signature that corresponds to that leaf is used to sign the second part of the digest. The root of the FORS signature is signed by the WOTS+ signature of that leaf. The root of each Merkle tree in the bottom layer is then signed by one of the WOTS+ signatures 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. Then 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 [BHKNRS19] for more details and security proofs.

3 WOTS+C

In this section, we present a variant of the WOTS+ [Hül13] constructions (which can be applied also 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).

Our modification works for any WOTS-like scheme that has the chaining structure, and does not depend on how the chain is computed and which functions are used for chaining. Thus, for simplicity of presentation here, we describe our modification as a variant of the simpler WOTS scheme. However, we stress that our implementation and experimental results are with the WOTS+ scheme. The description below uses WOTS solely for ease of presentation.

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 parameter $S_{w,n}, z \in \mathbb{N}$. Instead of signing the message m, we sign $d = H(\mathsf{m} \| r)$ where r is a short random bit string (a salt). We choose r 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 r. This is done by enumerating over r until the two conditions holds. We analyze the cost of process in the next subsection, but first we describe the benefits we gain:

- The first conditions means the 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 signing the first z. 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 $\ell = \mathsf{len}_1 - z$. Then, our scheme is implemented as follows:

- $\mathsf{KeyGen}(1^n)$: The secret key is random strings $\mathsf{sk} = (\mathsf{sk}_1, \dots, \mathsf{sk}_\ell)$. The public key is $\mathsf{pk} = (\mathsf{pk}_1, \dots, \mathsf{pk}_\ell)$, where $\mathsf{pk}_i = H^w(\mathsf{sk}_i)$.
- $\mathsf{Sign}(\mathsf{m}, \mathsf{sk})$: Find r that satisfies the two conditions. Compute $d = H(\mathsf{m}\|r)$. Map d to len_1 chain locations $a_1, \dots, a_{\mathsf{len}_1} \in [w]$. For $i \in [\ell]$ compute $\sigma_i = H^{a_i}(\lambda_i)$. Output $\sigma = (\sigma_1, \dots, \sigma_\ell, r)$.
- $\mathbf{V}(1^n, \mathsf{m}, \sigma, \mathsf{pk})$: Parse σ as $(\sigma_1, \dots, \sigma_\ell, r)$. Parse $\mathsf{pk} = (\mathsf{pk}_1, \dots, \mathsf{pk}_\ell)$. Compute $d = H(\mathsf{m}\|r)$. Map d to len_1 chain locations $a_1, \dots, a_{\mathsf{len}_1} \in [w]$. Verify that for all $i \in [\ell]$, it holds that $\mathsf{pk}_i = H^{w-a_i}(\sigma_i)$.

Once we remove the checksum chains, the running time of the verification becomes much faster (as it does not need to computes 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 7.3, it can decrease the overall time for SPHINCS+C signature generation.

3.1 Proof of security

We will now show a tight reduction from the EU-CMA security of our WOTS+C scheme to the WOTS+ [Hül17] scheme. The EU-CMA security is defined using the following experiment (where $D_{SS}(1^n)$) denote denote 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 the following way.

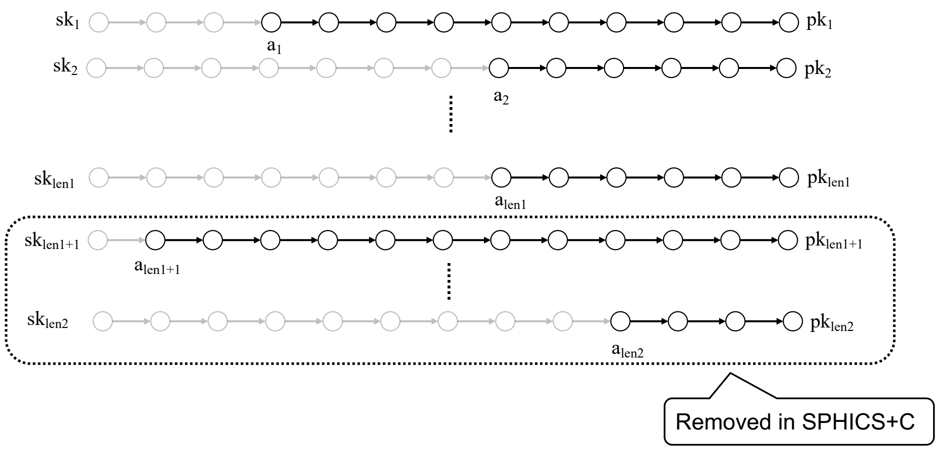

Figure 1: An illustration of the WOTS+ chains that are removed in the WOTS+C scheme.

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 with WOTS and WOTS+.

Our security reduction. To prove the security of our new scheme, we show how to reduce its security to the original security of WOTS. Denote the WOTS+ scheme by $(\mathsf{KeyGen}, \mathsf{Sign}, \mathbf{V})$ and our scheme by $(\mathsf{KeyGen}', \mathsf{Sign}', \mathbf{V}')$. Let $\mathcal{A}'$ be an adversary for our scheme. Assume, without loss of generality, that $\mathcal{A}'$ requests to see at least one signature. We build $\mathcal{A}$, an adversary for WOTS, as follows.

1. Given $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^n)$ (generated by the WOTS+ scheme), $\mathcal{A}'$ takes $\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}})$ and creates a public key $\mathsf{pk}' = (\mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{len}_1 - z})$ for our scheme by removing the len_2 words encoding the sum and to the first z words.
2. The adversary $\mathcal{A}$ runs $\mathcal{A}'$ with pk'.
3. For each request to sign a message m:
   (a) Find a random bitstring r such that $d = H(\mathsf{m} \| r)$ is valid for our signature (satisfies that two requirements).
   (b) Send d to the WOTS signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ to get $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$ (and store the signature).
   (c) Set $\sigma' = (\sigma_1, \ldots, \sigma_{\mathsf{len}_1 - z}, r)$ send the signature σ' to the adversary $\mathcal{A}'$.
4. Finally, $\mathcal{A}'$ outputs (m^*, σ').
5. Take $\sigma' = (\sigma_1, \ldots, \sigma_{\mathsf{len}_1 - z}, r)$ and create $\sigma = (\sigma_1, \ldots, \sigma_{\mathsf{len}})$ by adding the truncated parts from the signature we received from the WOTS challenger. Note that as the sum is always fixed and the first z words are always zero, the corresponding parts in the WOTS signature will always be the same. Output (d, σ) to the WOTS challneges.

The success probability of our adversary $\mathcal{A}$ is identical to that of $\mathcal{A}'$. Thus, the only cost of the reduction is the running of the adversary. We note that during our reduction, we need to find a suitable r value which require work. However, this is only a fixed additive amount of work that not related to the security parameter. Therefore, for reasonable security parameter and signature costs, this cost is negligible compare to expected running time of the adversary so our reduction is tight.

The tweakable WOTS (WOTS-TW) variant used in SPHINCS+ was formally proved in [HK22] in the non-adaptive chosen message attacks (EU-naCMA) where the adversary receives the public key only after it made its signature query. The process of creating our tweakable variant of WOTS+C from WOTS-TW and proving the reduction to the EU-naCMA of WOTS-TW is identical as the adversary in our reduction does not require the public-key for the signature oracle.

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 sign 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 get further gains, we can make the last tree bigger before we remove 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. When considering different trade-offs of parameters (in the overall hyper-tree of the SPHINCS+C scheme), this will be useful.

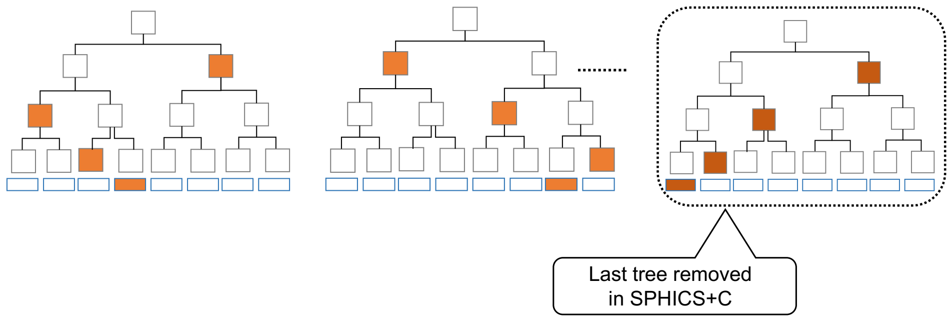

Figure 2: An illustration of the FORS tree that is removed in the FORS+C scheme.

4.1.1 Security

The security reduction is the same as the security analysis of FORS. One can (virtually) imagine that all k trees exists, 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. To 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 which leaves are opened. Thus, the security of our scheme follows verbatim.

Moreover, the analysis remains the same if not all trees are the same size. This mean 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 [BHKNRS19] 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 [BHKNRS19], 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 [BHKNRS19], we have that

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

The probability of being covered in all k trees is:

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

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

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

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].[2] 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 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 5), after 2^5 hashes, we expect to reduce the size of the FORS signature from $208 * 24 = 4992$

[2]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+.

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.

5 SPHINCS+C

At its simplest form, our proposed SPHINCS+C is the SPHINCS+ where we use WOTS+C instead of WOTS+ and FORS+C instead of FORS (later we describe additional optimizations). As we can reduce the security of our compressed variants to the original ones, the original security proof of SPHINCS+ holds also 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 on low-resources devices with one set, and to use the other set when the signer computational power is larger, but we want to reduce the size of the signature. 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.

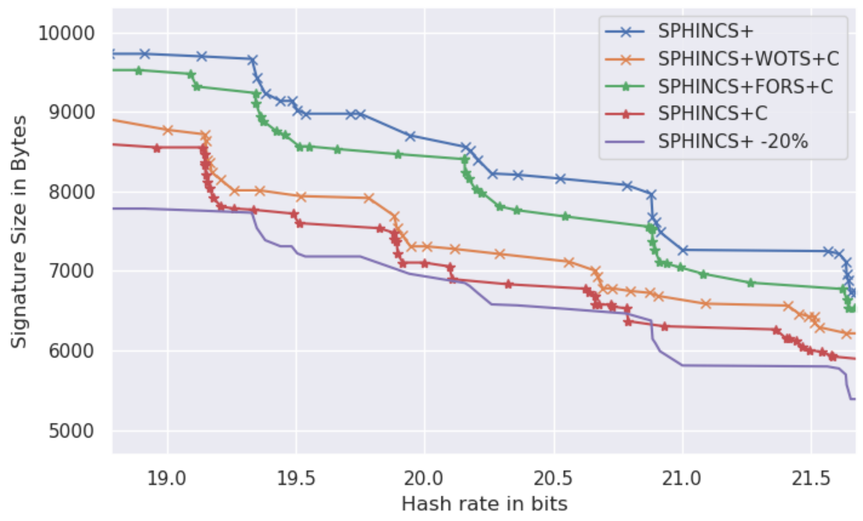

Figure 3: Hash rate (computational cost) to signature size in bytes trade-off curves for SPHINCS+ and our various compression options for (WOTS+C, FORS+C, and the full SPHINCS+C).

5.1 Parameters choice

For our concrete parameter choices, we searched for parameters that will minimize the signature size, while maintaining the signer running time and security level of the original variants proposed in the NIST submission of SPHINCS+ [Aum+22]. We note that in our "small" variants the verification time is (slightly) longer than in SPHINCS+. Nevertheless, we consider this to be acceptable, as in these "small" variants the major bottleneck is the run time of the signer which is more than two orders of magnitude larger than the verification time. Moreover, further optimizations that are not yet implemented will plausibly remedy the situation.

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

We recall the parameters notation from SPHINCS+:

n:	the security parameter in bytes.
w:	the Winternitz parameter.
h:	the height of the hypertree as defined in Section.
d:	the number of layers in the hypertree.
k:	the number of trees in FORS.
t:	the number of leaves of a FORS tree.

For our scheme we add a parameter:

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

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

	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.

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 A. 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.

5.2 Cost of finding valid counter for WOTS+C

In WOTS+C, we search for a counter where the resulting checksum of the digest is the expected value $S_{w,n}$. Moreover, when the number of bits $(N * 8)$ is not dividable by $\log(w)$, the last chain encodes less than $\log(w)$-bits, it is more effective to require it to have a zero value instead of including it in the signature.

We use an auxiliary script that for each possible values of n and w, tries a large number of random messages, and checks what is the probability of getting the expected average checksum (and the last zero chain when applicable). It stores the expected number of trials, the number of digits, the required number of hash calls for signature, and signature size. These values are then used in our script for finding the best parameters for the overall SPHINCS+C scheme.

We note that our WOTS+C algorithm ensures a constant time verification process, while incurring (small) variability in the cost of signing.

5.3 Using different values of w

The SPHINCS+ scheme uses only 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 $\log(w)$. We note that 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 last chain.

As a further optimization, we can use multiple values of w in out signature (e.g., 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.

6 Implementation

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

6.1 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.

6.2 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 virtual tree structure independent from each other. We refer the reader to the SPHINCS+ [BHKNRS19] 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 this we now add a 32-bit counter field and the remaining 64-bits are padded with zeros.

To save computation time, the unique bitmask used in the tweaked hash function is generated once with a counter value of zero. After that we try incremented values of the counter, until the resulting digest of the addresses concatenated with the masked message have the required checksum

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

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 validate 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 at 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.

6.3 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+ support 3 different underlining hash constructions, we chose to benchmarked the reference code for the “robust” SHAKE [Dwo15] variant that is based on the Keccak permutation [BDPVA09]. Our main goal is to compare the running time of SPHINCS+C with SPHINCS+ [BHKNRS19]. For each parameter set we ran the key generation, randomized signature generation and verification procedure, for 100 times.

Table 3 shows the average results of our benchmark. 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.

	Key Generation		Signature		Verification	
	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C	SPHINCS+	SPHINCS+C
SHAKE-128s	721.4	360.1 (−50%)	5398.0	4839.0 (−10%)	5.0	31.4 (+530%)
SHAKE-128f	10.8	9.5 (−12%)	256.3	262.2 (+2%)	16.4	13.2 (−20%)
SHAKE-192s	1068.6	599.4 (−44%)	9133.3	9339.0 (+2%)	8.5	51.1 (+502%)
SHAKE-192f	15.1	13.6 (−10%)	380.6	401.2 (+5%)	22.0	19.6 (−11%)
SHAKE-256s	652.6	396.1 (−39%)	7573.0	7323.2 (−3%)	11.4	37.9 (+234%)
SHAKE-256f	44.4	43.2 (−3%)	860.3	919.5 (+7%)	24.1	22.2 (−8%)

Table 3: Comparison of average running time in millions of cycles for the various variants between the original SPHINCS+ NIST submission and variants and our new SPHINCS+C variants.

7 Discussion

We discuss several properties of our schemes that are somewhat different that in SPHINCS+, and suggest future work.

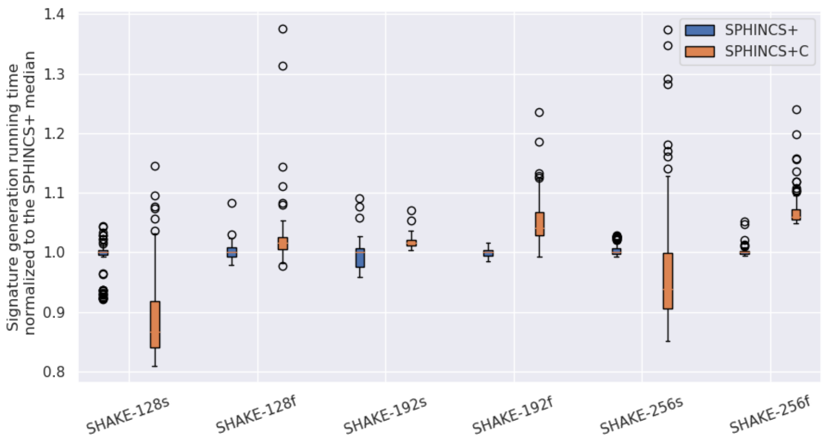

Figure 4: Comparison of variability in signature generation time between different SPHINCS+ and SPHINCS+C variants. The results of each variant include 100 repeated runs of randomized signatures and are normalized to the median of the SPHINCS+ variant.

7.1 Non-constant signing time

Our proposed signature scheme does not run in constant-time, which may raise questions regarding possible vulnerabilities to side-channel attacks. However, we note that the vulnerability depends only on the message and public values (i.e., the public key and public roots of the Merkle trees). This means that any variability in the running time of the signer is completely independent of any secret values and does not leak any information about them.

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 the average to find a suitable counter. Although theoretically possible, this is not a genuine concern for our parameter sets in practice. The reason is that the expected amount of work for finding a suitable counter is usually several orders of magnitude less than the total amount of work. This means that even when we are very unlucky, the total change in the run time is minimal. This is because running 10 times the expected running time happens with a probability of $e^{-10} \approx 2^{-15}$. For most parameter sets, even such an order of magnitude increase in the running time of the valid counter search phase is negligible compared to the total running time of the signature generation.

We 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% 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 3).

7.2 Constant verification time

Similar to [PZCMP21; ZCY22] our WOTS+C variant also 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 the running time for the verifier is constant for a fixed set of parameters for every signature.

7.3 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.

7.4 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 tradeoff can be especially useful for CAs, that sign a relatively small number of certificates, but their signatures are a part of a very large number of certificates and are verified in a large number of 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 tradeoffs and parameter sets for these use cases may help to facilitate the deployment of hash-based signature schemes.

7.5 Proof-of-work

Part of our optimizations require enumerating over and index i, in order to find a value with a digest of a specific format. For example, in our FORS+C scheme, the signer search for adigest that begins with b zeros. Only when such a digest is found is the signer allowed to continue creating the signature. This optimization can be interpreted as a *proof-of-work* [DN92].

We replaced the security that stems from the last tree with an equivalent proof-of-work. Any adversary has first to solve the proof-of-work and then can continue to try to attack the scheme. Without a solution to the proof-of-work, the verification algorithm will reject the signature. The advantage is that the solution for the proof-of-work is tiny (roughly b bits), much smaller than an entire authentication path (which is roughly $b \cdot n$ bits).

7.6 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.

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.

Acknowledgments

We thank Ofek Bransky for his contribution to the implementation and benchmark of the code.

Eyal Ronen is supported in part by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik ICRC, and Robert Bosch Technologies Israel Ltd. He is a member of CPIIS.

Eylon Yogev is supported in part by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the Alter Family Foundation.

References

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. "Clarifying the subset-resilience problem". In: *IACR Cryptol. ePrint Arch.* (2017), p. 909 (cit. on pp. 1, 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. 1, 9, 16).

[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. 2, 11).

[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. 14).

[BHKNRS19] 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. 1, 4, 8, 13, 14).

[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. 1, 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).

[DN92] Cynthia Dwork and Moni Naor. "Pricing via Processing or Combatting Junk Mail". In: *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*. 1992, pp. 139–147 (cit. on p. 16).

[Dwo15] Morris J Dworkin. "SHA-3 standard: Permutation-based hash and extendable-output functions". In: (2015) (cit. on p. 14).

[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. 1).

[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. 1).

[HK22] Andreas Hülsing and Mikhail A. Kudinov. "Recovering the tight security proof of SPHINCS$^+$". In: *IACR Cryptol. ePrint Arch.* (2022), p. 346 (cit. on p. 7).

[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 pp. 3, 4).

[Hül17] Andreas Hülsing. "WOTS+ - Shorter Signatures for Hash-Based Signature Schemes". In: *IACR Cryptol. ePrint Arch.* (2017), p. 965 (cit. on pp. 1, 3, 5).

[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. 1).

[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. 1).

[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. 1, 16).

[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. 1).

[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. 1).

[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. 1, 16).

A 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]

#Precomputed cost of work to find the expected sum and zeroized remaining bits
# if log(w) is not 4 or 8, number of chains, work to compute PK, and the expected sum
cost_wots_cs = {}
cost_wots_cs[(16, 4)] = (4.500985, 64, 8.000000, 96)
```

```
cost_wots_cs[(16, 8)] = (7.228422, 42, 8.392317, 147)
cost_wots_cs[(16, 16)] = (6.041304, 32, 9.000000, 240)
cost_wots_cs[(16, 32)] = (9.871210, 25, 9.643856, 387)
cost_wots_cs[(16, 64)] = (9.722725, 21, 10.392317, 661)
cost_wots_cs[(16, 128)] = (10.602894, 18, 11.169925, 1143)
cost_wots_cs[(16, 256)] = (9.580629, 16, 12.000000, 2040)
cost_wots_cs[(24, 4)] = (4.775446, 96, 8.584963, 144)
cost_wots_cs[(24, 8)] = (5.550769, 64, 9.000000, 224)
cost_wots_cs[(24, 16)] = (6.347193, 48, 9.584963, 360)
cost_wots_cs[(24, 32)] = (9.137762, 38, 10.247928, 589)
cost_wots_cs[(24, 64)] = (8.038267, 32, 11.000000, 1008)
cost_wots_cs[(24, 128)] = (11.947862, 27, 11.754888, 1714)
cost_wots_cs[(24, 256)] = (9.828281, 24, 12.584963, 3060)
cost_wots_cs[(32, 4)] = (4.978009, 128, 9.000000, 192)
cost_wots_cs[(32, 8)] = (6.737043, 85, 9.409391, 297)
cost_wots_cs[(32, 16)] = (6.533493, 64, 10.000000, 480)
cost_wots_cs[(32, 32)] = (8.367181, 51, 10.672425, 790)
cost_wots_cs[(32, 64)] = (12.252528, 42, 11.392317, 1323)
cost_wots_cs[(32, 128)] = (13.118589, 36, 12.169925, 2286)
cost_wots_cs[(32, 256)] = (10.027687, 32, 13.000000, 4080)

#### 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)

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 = \
                                    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()
```