ARMed SPHINCS

Computing a 41 KB signature in 16 KB of RAM

Andreas Hülsing[1], Joost Rijneveld[2], and Peter Schwabe[2]

[1] Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
andreas.huelsing@googlemail.com
[2] Radboud University, Digital Security Group,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
joost@joostrijneveld.nl, peter@cryptojedi.org

Abstract. This paper shows that it is feasible to implement the stateless hash-based signature scheme SPHINCS-256 on an embedded microprocessor with memory even smaller than a signature and limited computing power. We demonstrate that it is possible to generate and verify the 41 KB signature on an ARM Cortex M3 that only has 16 KB of memory available. We provide benchmarks for our implementation which show that this can be used in practice. To analyze the costs of using the stateless SPHINCS scheme instead of its stateful alternatives, we also implement XMSS^{MT} on this platform and give a comparison.

Keywords: post-quantum cryptography, hash-based signature schemes, microcontroller, resource-constrained devices, ARM Cortex M3, SPHINCS-256, XMSS^{MT}

1 Introduction

It is difficult to precisely predict the future of computing, but once large-scale quantum computers become feasible in practice, all of the asymmetric cryptography that is widely deployed today will be broken. Over the past few years, several schemes have been proposed that address this issue. One of the most promising classes of schemes that provide post-quantum secure digital signatures are hash-based schemes [21].

Hash-based schemes come with well-understood security guarantees, building only on the assumption of a secure cryptographic hash function. This makes them a very attractive, confidence inspiring choice. The keys they use and the signatures they produce are of practical sizes, and signing is reasonably fast. Additionally, signature verification is very fast. Until recently, however, all practical hash-based schemes required a state that must be constantly kept up to date.

This work was supported by the Netherlands Organisation for Scientific Research (NWO) under Veni 2013 project 13114 and by European Commission through the ICT program under contract ICT-645622 (PQCRYPTO). Permanent ID of this document: c7ea17f606835ab4368235a464e1f9f6. Date: 2016-02-03

In many real-world scenarios, this is a far reach from the signature schemes that are currently in use.

The introduction of SPHINCS [4] at Eurocrypt 2015 demonstrated that it is not strictly necessary to maintain a state for a hash-based scheme to be practical. Lifting this constraint does not come for free; it is paid for by an increase in signing time as well as signature size. Still, SPHINCS-256 remains fairly efficient in terms of these dimensions, with signatures of 41 KB and a signing rate of *"hundreds of messages per second on a modern 4-core CPU"* [4].

This shows that SPHINCS is a feasible solution on high-end servers and desktops, but the question remains, whether SPHINCS is also a feasible solution for small embedded "Internet-of-Things" devices. This is not merely a question of performance. It is a question of whether it is even possible to compute a 41 KB SPHINCS signature on a device with only little RAM. In this paper we demonstrate that it possible to compute and verify SPHINCS-256 signatures on an ARM Cortex M3 microcontroller with only 16 KB of RAM, but the performance results indicate that practical applications are limited to non-interactive contexts (such as sensor nodes sending signed data several times a day).

To illustrate the cost of eliminating the state in hash-based signatures, we furthermore implement the state-of-the-art stateful hash-based signature scheme XMSS^{MT} as described in a recent Internet draft [16]. To provide a fair comparison, we replaced all the used hash functions with similar functions as SPHINCS-256.

Availability of software. We place all software described in this paper into the public domain. It is available online at `https://joostrijneveld.nl/papers/armedsphincs`.

1.1 Related work

In [24], the potential for hash-based signature schemes on constrained microprocessors was first demonstrated. The authors establish that it is possible, to implement GMSS [6], an improvement of Merkle's original hash-based signature scheme, on an 8-bit AVR microprocessor at a speed comparable to RSA and ECDSA, although without key generation. The described platform offers 8 KB of program memory and 4 KB of SRAM.

A variant of XMSS was implemented on a 16-bit smart card [8]. The authors show that key generation can be done on the device and get even faster speeds than [24], further demonstrating practicality of (stateful) hash-based signature schemes on constrained devices.

Extensive side-channel analysis of a fast Merkle signature scheme implementation on an AVR ATxmega is presented in [10]. This paper introduces a new algorithm for the computation of authentication paths in a Merkle tree to significantly reduce (and actually bound) side-channel leakage during this computation.

Other post-quantum schemes also show promising results on embedded systems. In [12], a lattice-based signature scheme is shown to produce signatures of

9 KB, with keys of 2 KB and 12 KB in size, beating RSA in terms of speed, on a Xilinx Spartan-6 FPGA. While memory usage is slightly higher compared to hash-based schemes, [22] shows that lattice-based signatures can be very fast by providing an implementation on a Cortex-M4F. Multivariate-quadratic systems have also been implemented and proven to be practical on low-resource devices as well as ASICs, with keys of practical size [25].

The software presented in this paper is the first to describe (stateless) signature software achieving 128 bits of security against quantum attackers on an embedded microcontroller, which naturally makes comparison to previous results hard. On the one hand, none of the previous papers targets 128 bits of post-quantum security, and, unlike our software, both [10] and [8] use hardware accelerators for fast hashing. On the other hand, 8-bit AVR microcontrollers used in [6] and [10] and the 16-bit Infineon SLE 78 used in [8] are less powerful (and offer less RAM and ROM) than the more recent Cortex-M3 used in this paper. For many applications there is a trend to move from 8-bit and 16-bit microcontrollers towards more powerful 32-bit processors like the Cortex-M; mainly towards the low-end Cortex-M0, which is explicitely advertised to "achieve 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices" [20]. The Cortex-M3 is quite a bit more powerful than the Cortex-M0 and most importantly offers more RAM and ROM than the M0. Our memory usage suggests that it might be feasible to bring SPHINCS also to the M0 with 8 KB of RAM, but this would not leave any space for other applications and would thus be a mere academic challenge.

2 The SPHINCS-256 signature scheme

This section explains the SPHINCS-256 signature scheme as proposed in [4]. The section follows a top-down approach: we first explain why there is a need for SPHINCS and present a high-level overview. Then, we fill in details on each of the components. Rather than discussing the SPHINCS scheme in general, we will directly and explicitly adhere to the parameters and functions proposed as SPHINCS-256 in the original paper. Refer to [4] for a more general description.

2.1 Eliminate the state

In [17], Lamport describes a one-time signature scheme (OTS) that forms the foundation for hash-based signatures. This scheme has later been used and improved by Merkle [21] to build a many-time signature scheme. This is done constructing a binary hash tree on top of a series of OTS key pairs, effectively joining them together under a single long-term public key. When a sequence of authentication nodes is supplied as part of the signature together with an OTS public key, a verifier is able to reconstruct the long-term public key at the root of the authentication tree. See Figure 1 for an illustration of this. This approach is still the main construction used in modern hash-based signature schemes (such as XMSS [5]).

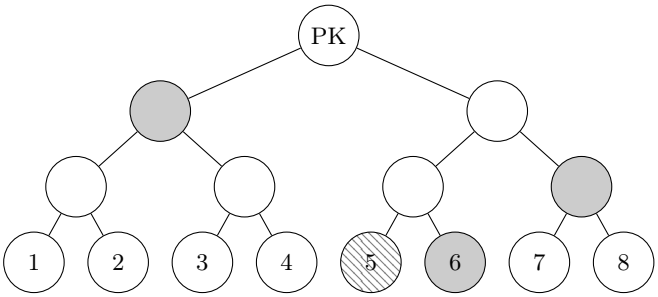

Fig. 1. The authentication nodes needed to authenticate leaf node 5 are coloured grey.

However, these constructions suffer from a fundamental problem. When using Merkle trees on top of OTS key pairs, the user should be very careful not to use a OTS key twice as this would undermine security. This implies that, in addition to storing the secret key (i.e., the seed that produces all OTS keys on the leaf nodes), one needs to store some indicator to keep track of the leaf nodes that have already been used[3]: the state. While this is not a problem in some applications, key management can quickly become an issue. In scenarios where multiple instances of the key are stored in different places (for example, backups, different machines used for load balancing, or different devices owned by the same user), the state needs to be constantly kept in sync among those copies. This makes such a signature scheme highly impractical and incompatible with many of today's systems.

Already in 1986, Goldreich recognized this problem and proposed a solution [11]: create a tree of such depth that, when randomly choosing an OTS key pair for each signature, the chance of accidentally reusing a certain OTS key pair becomes insignificantly small. This way, there is no need to keep track of the already used OTS keys. The obvious problem here is actually creating such a tree in the first place, but Goldreich is able to avoid this. By not simply hashing nodes together (as is the typical Merkle tree construction) but instead attaching an OTS key pair to each tree node and using that to sign the child nodes, it is never necessary to compute the entire tree. This requires that the OTS keys of the nodes along the path from a random leaf to the root node are deterministically generated out of order, but this can easily be done using pseudorandom function with a secret seed and the node index.

While Goldreich's system solves the issue of having to maintain a state, it introduces a new problem. As it replaces hashing with signing throughout the tree, it also replaces hash digests with OTS signatures for the authentication-path nodes included in each signature. This creates a new hurdle for practical

[3] In practice, this could be just the number of messages signed so far, as Merkle showed that using the keys sequentially is often preferable [21]

use, as it results in tremendously large signatures (more than 1 MB for reasonable parameters).

2.2 Overview of SPHINCS-256

As discussed above, the main problem with hash-based signature schemes is either the need to maintain a state or the size of the signatures. This has prevented hash-based solutions from being a drop-in replacement for the signature schemes that are currently in use. SPHINCS solves this by combining the approach of Goldreich with traditional Merkle trees in a nested construction and few-time signatures. This results in a stateless scheme with signatures of 41 KB and private and public keys of 1 KB each [4]. At "hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU", it is shown to be sufficiently fast for many practical applications. In later sections of this paper, we demonstrate that it is also practical on low-end devices with highly limited resources.

The nested trees construction forms the base of SPHINCS. The complete structure consists of a total of $h = 60$ layers, divided over $d = 12$ layers of sub-trees. This can be viewed as a hypertree of two levels of abstractions, where each node in the global tree represents a sub-tree. Each of these sub-trees then consists of $h/d = 5$ layers of nodes themselves. Let us refer to the sub-trees on layer i of the global tree as τ_i, where $i \in \{1, \ldots, d\}$. We refer to the nodes in a sub-tree as $\nu_{i,j}$, where $j \in \{1, \ldots, h/d\}$ is their level in the sub-tree and $i \in \{0, 2^j - 1\}$ the index within that level. There is no need to diversify between nodes or trees in the same layer at this point, as they each serve an identical purpose.

The trees τ_i are binary hash trees, only slightly varying from the original Merkle tree concept. Each of their nodes $\nu_{i,j}$ for $j \in \{1, \ldots, h/d - 1\}$ contains a digest of its child nodes, while the leaf nodes on layer h/d each contain the key of an OTS. For now, let us assume that we have some hash function H that generates these digests. As with Merkle trees, the digest at the root of the tree is used to authenticate the entire structure by constructing authentication paths.

All the sub-trees are then chained together as in Goldreich's system. Using the OTS keys in the leaf nodes $\nu_{i,h/d}$ of the trees τ_i, the root nodes of the trees τ_{i+1} are signed; a new sub-tree is chained to each of the leaf nodes. See Figure 2 for a close-up of this construction. Nodes labeled H contain a hash of their child nodes; nodes labeled OTS include a key pair to authenticate their child node.

The OTS key pairs of the leaves of the trees on the bottom layer are not used to authenticate more sub-trees. Instead, they are used to authenticate the public key of a *few-time* signature scheme (FTS). An FTS behaves similarly to an OTS, but can be used several times before revealing too much of the secret key. By using an FTS rather than an OTS, SPHINCS does not require as many leaf nodes to maintain the same security level: the required maximal probability of selecting the same node repeatedly can be much higher without breaking the system. These FTS keys are used to sign the actual messages.

The above describes the basic outline of SPHINCS. This still far from a working algorithm, though, as we have assumed a number of black boxes: some

hash function H to use in the sub-trees, an OTS to use between sub-trees and an FTS to sign the actual message at the bottom of the hypertree. In the following subsections, we will gradually collect the missing pieces.

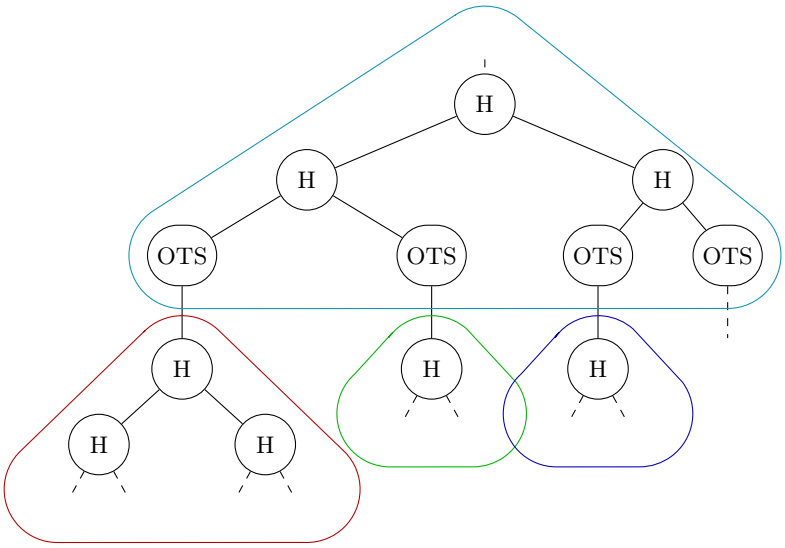

Fig. 2. Linking sub-trees together

2.3 Key generation

Because of the hypertree structure, key generation for SPHINCS is a fairly cheap operation. We start by selecting some random values $SK_1 \in \{0,1\}^n$ and $SK_2 \in \{0,1\}^n$. For SPHINCS-256, $n = 256$. The first of these values is used for key generation, while the second is required for signing. This will be illustrated in the next subsection. Additionally, we generate a tuple Q of random bitmasks, each one also from $\{0,1\}^n$. These masks are used in the hash trees (as described in Section 2.6), as well as in the OTS and FTS – for now, let us merely acknowledge their existence. Thus $SK = (SK_1, SK_2, Q)$.

In order to generate the public key, we only need to generate the single tree in τ_1: the tree at the top of the structure. This requires generating the OTS keys along the bottom of this tree. Note that these keys need to be generated deterministically; using their address and SK_1 as input to some pseudorandom function we can derive a seed for this key. Then, a binary hash tree can be built on the public keys of the OTS key pairs, and the root node of this tree is part of the SPHINCS public key: PK_1. As the bitmasks are also needed for verification, they must also be included in the public key: $PK = (PK_1, Q)$.

It is worth noting that, while SK_1, SK_2 and PK_1 are all only 256 bits (or 32 bytes) in size, Q is significantly larger. SPHINCS-256 uses 32 bitmasks in Q, which add up to a total size of $32 \cdot 32 = 1024$ bytes. Bitmasks thus account for the largest part of the keys. In general, the number of bitmasks is determined by the part of the scheme that requires the largest number of them – the FTS, the OTS or the hash trees.

2.4 Signing

Public-key signature schemes typically compute a hash of the message that is to be signed, and then sign that hash. This ensures that the input is of a constant, relatively small length. In a stateless scheme like Goldreich's, a random key pair at the bottom of the tree would then be selected to sign the hash. In SPHINCS, however, the key pair is selected based on the message hash itself. In order to prevent attackers from specifically targeting certain key pairs, some random or unknown factor still needs to be included – this is what SK_2 is for. We first compute a bitstring $(idx \| R)$ using a pseudorandom function that takes SK_2 and the message as input, and use idx to select an FTS key pair. The second part R is used to compute a randomized digest D of the message. This digest is what we will be signing. As a practical result of all this, the selection of an FTS key pair is completely deterministic with respect to a secret key SK_2 and a message M.

After selecting a particular FTS key pair, the secret key of this key pair needs to be generated (based on a seed derived from its location and SK_1) and is then used to sign D and produce the signature σ_{FTS}. Together with the message-specific randomness R generated above and the index idx of the selected key pair, this signature forms the first part of the SPHINCS signature Σ. As SPHINCS uses an OTS and an FTS for which the public keys can be derived from their respective signatures (as we will see in Section 2.7 and 2.8), there is no need to include the FTS public key here.

We then generate the OTS key pair for its parent node in $\nu_{d,h/d}$ in the relevant sub-tree in τ_d (again using its position in the tree in combination with SK_1), and use it to sign the FTS public key. Let us refer to the produced signature as $\sigma_{OTS,d}$. This signature is also added to Σ. The public key of this OTS needs to be authenticated, so we compute all nodes along its authentication path throughout the tree in τ_d and include those in Σ as well. We refer to the nodes along the authentication path in the selected tree on layer d as $Auth_d$. Upon reaching the root of the tree in this fashion, we generate the OTS key pair that belongs to its parent node in $\nu_{d-1,h/d}$ and use that to sign the root. This procedure continues all the way up to the root of the one tree in τ_1, which is, by construction, included in PK. While progressing up the hypertree, all OTS signatures and nodes along the authentication paths need to be added to Σ.

Altogether, the SPHINCS signature Σ now contains the message-specific randomness R, the index idx of the selected FTS key pair, the FTS signature σ_{FTS} and d pairs of OTS signatures and sequences of nodes along the authentica-

tion path ($\sigma_{OTS,i}$, $Auth_i$). Everything combined, $\Sigma = (idx, R, \sigma_{FTS}, (\sigma_{OTS,1}, Auth_1), \ldots, (\sigma_{OTS,d}, Auth_d))$.

2.5 Signature verification

The procedure for verifying a signature on M is very similar to signing. As we have seen above, the signature Σ contains the message-specific randomness R, the FTS signature and the OTS signatures and authentication paths. After computing D (using R and M), the FTS signature is verified. As mentioned above, the verification function of the OTS and FTS used in SPHINCS output the respective public key. Hence, the OTS signature on the FTS public key can now be verified, resulting in the respective OTS public key. As the authentication path is also given in Σ, the root node of the tree in τ_d can now be computed. Similar to the way the signature was generated, we now continue up the tree along the authentication paths while verifying the signatures on the root nodes of each sub-tree.

In the end, the verification eventually arravies at the root node of the single tree in τ_1. This root node should be equal to PK_1, included in PK. If this is the case, the signature is valid.

2.6 Hash trees

At its core, SPHINCS heavily relies on hash trees. The construction of these trees is slightly different from the classical binary Merkle trees. After concatenating the values of the two child nodes, they are not immediately fed to a hash function to produce the parent node. Instead, two *bitmasks* are applied first; let Q_i, Q_{i+1} be such bitmasks and h_2, h_3 child nodes, then $h_1 = H((h_2 \| h_3) \oplus (Q_i \| Q_{i+1}))$.

In [2], XORing with bitmasks is introduced as part of a linear hashing scheme, and it is employed in [9] in order to construct binary hash trees that do not require the underlying hash function to be collision resistant. Instead, second-preimage resistance is sufficient to attain unforgeability. This hash-tree construction is the one described above, and is used in SPHINCS.

2.7 The FTS: HORST

At the bottom of the hypertree, SPHINCS relies on an few-time signature scheme. For this, a variation of an FTS called HORS [23] is used. This variant, referred to as HORST, adds a tree construction to plain HORS [4]. The configuration of HORST consists of two parameters that control the security level, signature size, and key sizes: t and k, where t is a power of 2. For SPHINCS-256, we have $t = 2^{16}$ and $k = 32$.

As mentioned in the overview of SPHINCS, the key is seeded based on SK_1 and the location of a particular HORST instance in the hypertree. This seed is expanded to t secret-key components to form $sk = (sk_0, \ldots, sk_t)$, which are then hashed to create the public-key components pk_i for $i \in \{0, \ldots, t\}$. In SPHINCS-256, each sk_i and pk_i has 32 bytes. The HORST variation then proceeds to

build a hash tree on top of the public-key components. The root of this tree is the actual HORST public key pk. For this tree, SPHINCS also makes use of bitmasks as described in Section 2.6.

Signing a message M using a HORS key pair is done by splitting the message into k pieces of length $\log_2 t$. Each of these pieces M_i is then used as an index to address a piece of the secret key, sk_{M_i}, which is subsequently revealed. For HORST, the nodes along the authentication path from these hashes to the root of the tree are also required as part of the signature.

Verification is quite similar: first, the revealed pieces of sk are hashed, and the message is split into k parts. These parts are then interpreted as integers and used to place the pieces of sk on the appropriate leaves. Using the nodes supplied in the signature, the path to the root node can then be computed. This is done for all nodes and authentication paths checking that all agree on the same root. If this is not the case, verification fails. The verification algorithm then outputs this root as the public key – comparing it to the actual public key will reveal if the signature was valid.

What makes HORS usable as a few-time signature scheme (as opposed to an OTS) is the choice of a sufficiently large t in relation to k. This implies that only a small part of the secret key is revealed for each signature, and the chance of a successful forgery after obtaining just a few signatures diminishes. Note that this does require that an adversary cannot control the message hash for which a signature is obtained. As we have seen above, this requirement is satisfied in SPHINCS. In the original HORS scheme the combined size of public key and signature would increase linearly with t, but the HORST variation only incurs logarithmic growth in t, caused by the length of the authentication paths. This makes it possible to use HORST as the FTS in SPHINCS without dramatically increasing the key length.

2.8 The OTS: WOTS+

For the OTS that links the sub-trees together, SPHINCS uses WOTS+. This variation of the Winternitz OTS is proposed in [14], designed to reduce the signature size even further than other WOTS-based schemes. As in WOTS, the Winternitz parameter $w = 16$ is used to configure the efficiency trade-off. Likewise, one then derives ℓ (consisting of ℓ_1 and ℓ_2) from this parameter and the security setting $n = 256$ as follows.

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

For the SPHINCS-256 configuration, it can be readily computed that $\ell_1 = 64$ and $\ell_2 = 3$, thus $\ell = 67$.

In the plain WOTS scheme, a function F is applied to the secret key several times to produce a hash chain. In WOTS+, however, we take into account the bitmasks. In each iteration, before applying F, the input is XORed with a round-specific bitmask Q_i. The chaining function then looks as follows (where the base case is $c^0(x) = x$):

$$c^i(x) = F(c^{i-1}(x) \oplus Q_i)$$

In order to guarantee deterministic signatures here as well, WOTS+ key pairs are also seeded using SK_1 and their location in the tree. This seed is then expanded to a secret key of $\ell = 67$ pieces, $sk = (sk_1, \ldots, sk_\ell)$. Generating the key is very similar to traditional WOTS; we simply apply the chaining function c for a total of $w-1$ times to each part of sk to obtain $(pk_1, \ldots, pk_\ell)$. In SPHINCS-256, each of the sk_i and pk_i has again 32 bytes. As the reader might be expecting by now, we proceed by building a hash tree on top of these public-key parts. These trees make up the third abstraction level of trees in the hypertree. However, ℓ is not necessarily a power of two – in fact, as w and n (and thus ℓ_1) typically are a power of two, ℓ is not. This requires the use of a slightly different tree construction: the L-Tree [9]. The structure is entirely identical to binary hash trees, except for the rightmost nodes. Whenever the number of nodes on the current layer is odd, the rightmost node is lifted up to the next layer instead. See Figure 3 for an example with five nodes. The root of this tree is the public key pk.

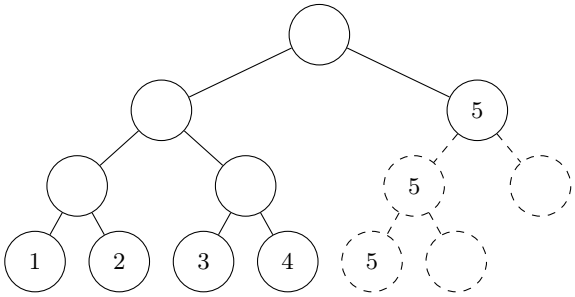

Fig. 3. An L-Tree with five leaf nodes

In order to sign a message M, we first interpret it as an integer in binary, and then express it in base w. This effectively splits M into a sequence of values that can be at most $w-1$, each. Note how there will be at most ℓ_1 values, as per the construction of ℓ_1. We write $M = (M_1, \ldots, M_{\ell_1})$. Furthermore, a checksum needs to be computed to prevent an attacker from being able to forge a signature: $C = \Sigma_{i=1}^{\ell_1}(w - 1 - M_i)$, which is also expressed in base w: $C = (C_1, \ldots, C_{\ell_2})$. The lists M and C are then chained together to form $B = (b_1, \ldots, b_\ell) = M \| C$. These values in B are then used as lengths for the Winternitz chains, producing the signature $(\sigma_1, \ldots, \sigma_\ell) = (c^{b_1}(sk_1), \ldots, c^{b_\ell}(sk_\ell))$.

Verifying a WOTS+ signature is very similar to the regular WOTS scheme, except that one needs to take special care to use the correct bitmasks when

applying the chaining function[4]. Let us define the function v to account for this as

$$v^{i,j}(x) = F(v^{i,j-1}(x) \oplus Q_{w-i+j-1}).$$

Like for the original chaining function, the base case is $v^{i,0}(x) = x$ and we abbreviate $v^{i,i}(x) = v^i(x)$.

We then compute B in the same way as in the signing procedure, and then compute the public key parts $(pk_1, \ldots, pk_\ell) = (v^{w-1-b_1}(\sigma_1), \ldots, v^{w-1-b_\ell}(\sigma_\ell))$. As was the case during the key generation step, the last step that remains is computing an L-Tree over these pieces. The verification algorithm then outputs the root of this tree. Like in HORST, this can then be compared to the actual public key to verify that the signature was valid.

2.9 *H* and *F*: ChaCha

Two key elements of SPHINCS have not yet been discussed. Practically the entire scheme consists of computing hashes. In WOTS+, some hash function $F : \{0,1\}^n \to \{0,1\}^n$ is used to construct the chaining function, and throughout the entire hypertree, $H : \{0,1\}^{2n} \to \{0,1\}^n$ is used to construct binary hash trees. The function F is also used to compute the HORST leaf nodes based on the secret key. For the performance of the scheme, it is crucial that these functions are sufficiently fast. An important characteristic of both of these functions is that they do not need to be able to take arbitrarily long input. This makes it unnecessary (and wasteful) to select a hash function that can. As it turns out, being able to accept arbitrarily long input is one of the properties that typically slow down hash functions. Instead, SPHINCS uses a permutation-based construction following the sponge design using the permutation from the ChaCha stream-cipher family [3].

The core of ChaCha is a 512-bit permutation, so in order to use it for F, the input needs to be padded to extend it. In SPHINCS, the authors chose the 32-byte ASCII string $C =$ "expand 32-byte to 64-byte state!". The output of F is obtained by truncating the output of the ChaCha permutation to 256 bits. H is constructed similarly. Let $Chop(M, i)$ be a function that truncates M to i bits, M_1 and M_2 be strings of 256 bits, and O be a string of 256 zero-bits, then

$$\begin{aligned} F(M) &= Chop(\pi_{ChaCha}(M\|C), 256), \text{ and} \\ H(M_1\|M_2) &= Chop(\pi_{ChaCha}(\pi_{ChaCha}(M_1\|C) \oplus (M_2\|O)), 256). \end{aligned}$$

Now only a few minor pieces of the puzzle remain. Creating the message-specific random value R is done by calling BLAKE-512$(SK_2\|M)$ [1], and BLAKE-512 is used once more to create the digest D. In order to derive the secret keys for the HORST and WOTS+ key pairs based on their location and SK_1, the

[4] Note that this approach differs slightly from the one presented in [14]. In the original definition this is solved by supplying the appropriate set of bitmasks as an argument to the chaining function.

BLAKE-256 function is used as a pseudo-random function. Along the bottom of each of the trees, the ChaCha12 stream cipher is used to generate HORST and WOTS+ keys. An complete overview of the SPHINCS-256 hypertree is shown in Figure 4.

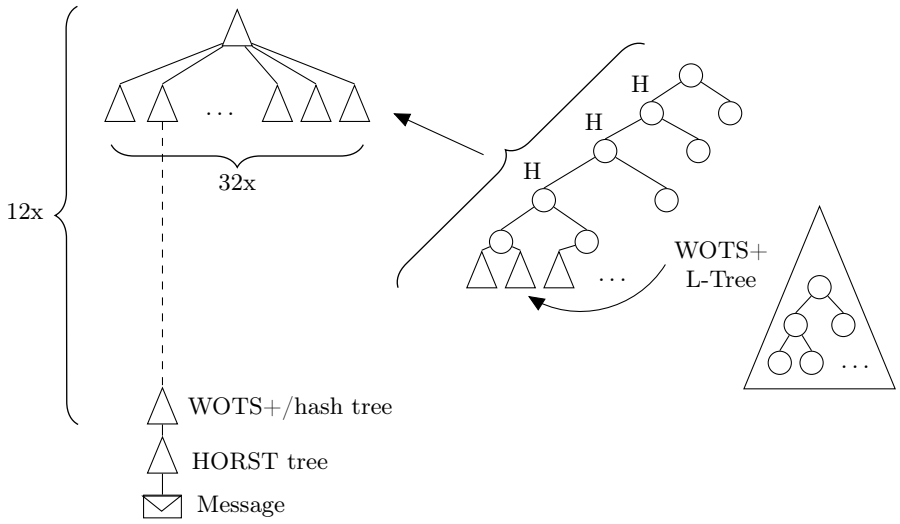

Fig. 4. The complete hypertree structure

3 The ARM Cortex-M3 microcontroller

The platform of choice for this implementation is the ARM Cortex M3, and development was carried out using the STM32L100C discovery board. The Cortex M3 is a 32-bit microprocessor with sixteen 32-bit registers, thirteen of which are available for general-purpose computation (leaving three for the stack pointer, link register and program counter). This microcontroller is commonly found in embedded systems used in the automotive industry, small industrial systems and (wireless) sensors, but as we mentioned in Section 1.1, the low-end 32-bit Cortex-M processors are also starting to replace 8-bit and 16-bit microprocessors (given the similarly low production costs and efficiency, but larger computational power). As opposed to the ARMv6-M [18] architecture found among the smaller processors in the Cortex M-series, this processor supports the ARMv7-M [19] architecture.

The ARMv7-M architecture is aimed at microcontrollers, and is highly optimized for low-cost devices. It only supports the Thumb-2 instruction set (as opposed to the full ARM instruction set) and its register scheme is very straightforward – there are no SIMD instructions or extended register banks. This is

somewhat compensated for by the efficient STM and LDM instructions. While these cannot be pipelined with other instructions, they provide internal pipelining when transferring more than one register to or from memory, making it not so costly to swap out the register contents. In Thumb-32 instructions, there are some limitations to their use on the SP, PC and LR registers, but none of this is a direct obstacle in our use-case.

The STM32L100C is part of the STM32 ultra-low-power series. The processor runs at a clock speed of 32 MHz, and the board offers 16 KB of RAM. This makes it a highly constrained platform for the implementation of SPHINCS, given that the stack usage of the Haswell implementation [4] runs into several megabytes. Notably, this means that the available RAM is insufficient to store the signature, which weighs in at 41 KB.

In order to communicate with the device at runtime, we make use of serial communication over USART. This can be done efficiently using the direct-memory-access (DMA) controller, to prevent blocking the computation while waiting for the communication interface. Doing this, we are able to communicate reliably at a baud rate of 921 600 Bd. To be able to configure and use this from a more abstract level, we made use of the open-source libopencm3 firmware library.

4 Implementing SPHINCS-256 on the Cortex-M3

The main contribution of this work is to show that SPHINCS-256 can be implemented on resource-constrained devices. The Cortex M3 is quite constrained both in terms of available volatile memory as well as processor speed, serving as a proof of concept that this does not render SPHINCS unusable. In this section, we describe implementation-specific design choices and present the achieved speed results. It should further be noted that this implementation makes use of code from the SPHINCS reference implementation [4] as well as (parts of) implementations of BLAKE-256 and BLAKE-512 [1] and the ChaCha12 stream cipher [3].

4.1 Signing within 16 KB

In a hash-based signature scheme, signing and verification are very similar in design, but they differ significantly when actually being performed. For verification, much of the work has already been done when producing the signature, and there is no need to construct entire trees from the ground up. The same is the case for SPHINCS. While verification is fairly straight-forward in terms of memory use, signing requires more effort to get right.

The general approach is as follows. In order to reduce the memory consumption of the signing operation, we split the computation into disjunct parts, and process the output of each part before continuing with the next part. This makes sure that we only have to account for the memory requirements of each such part at a time, instead of the consumption of the entire operation. The remaining task

is now to find suitable points at which to split the computation such that memory use is sufficiently low in each of the parts, without introducing too much performance overhead.

Tree storage. The SPHINCS scheme consists of a number of clearly distinct components, with the HORST trees and WOTS+/hash trees as the two most prominent subdivisions. While the memory usage is typically large at the base of a tree, it fans in again as one progresses towards the root. As each tree is stacked on top of the one below, it is not necessary to ever store more than one tree in memory at a time before proceeding on to the next – this progression is highly sequential.

For the WOTS+/hash trees, the available memory is not an immediate problem. Producing a single WOTS+ signature impacts the available memory for only 67 bytes. At $32 \cdot 67 = 2144$ bytes, the WOTS+ public key itself is slightly larger, but this can be quickly reduced to a 32 byte root node by applying the L-Trees we have seen in Figure 3. This does imply that one really has to perform this reduction before continuing with the next WOTS+ leaf node, but in the current context this has no negative impact. After processing all leaf nodes in this fashion, one is left with 32 leaf nodes of 32 bytes each. Each authentication tree contains only $h/d = 5$ layers of hashing, resulting in a total of $2^6 - 1 = 63$ nodes, which can be stored in memory all at once. After computing the entire tree, the nodes along the authentication path can be conveniently selected.

HORST, on the other hand, is a different beast entirely. With $t = 2^{16}$ and $k = 32$, the trees contain 131071 nodes spread over 16 layers of hashing, making these trees much higher than the hash trees on top of the WOTS+ signatures. This means that the method of first building the entire tree and then extracting the authentication path is not feasible. At 32 bytes per node, the nodes alone would require 4 MB of storage. There is no need to store the entire tree, though, as only a very specific set of nodes is relevant for the signature: the nodes along the 32 authentication paths, as well as the root node. As we do require the root node to authenticate the tree, there is definitely no escaping having to *compute* the entire tree.

Treehash. In [4], the authors mention that RAM usage and code size was not one of the concerns when writing the optimized implementation – the implementation was optimized for speed on a platform where memory was available in abundance. They remark that, if saving memory is a concern, the treehash algorithm could be used. This is what we will now apply in order to compute the HORST authentication path.

The treehash algorithm is another contribution by Merkle [21, Section 7]. It has since been used in various forms as the basis of tree-traversal algorithms. A common approach is expressed by the pseudo-code (based on [13]) in Algorithm 1, and discussed below.

The core idea is to grow a tree, using its leaves in subsequent order and only maintaining a collection of the currently relevant nodes: the 'heads' of the different branches. As new nodes are added, these branches are gradually grown to completion, and merged when needed. Any nodes that occur deeper in the tree

Algorithm 1 One round of the treehash algorithm

Require: STACK, next leaf node N
Ensure: STACK is updated
1: **while** STACK.peek() is on same level as N **do**
2: neighbour ← STACK.pop()
3: N ← H(neighbour ‖ N)
4: **end while**
5: STACK.push(N)

can safely be forgotten (for the purpose of finding the root node), as each node is only required once to generate its parent. Each round of treehash consists of introducing the next leaf node and updating the heads of the branches until no more new nodes can be computed. For half of the leaves (i.e., the 'left neighbors'), their introduction does not allow for the computation of any new parent nodes, while a quarter of the leaves allows us to compute one parent node, etcetera.

When examining how the set of relevant nodes evolves, there is a strict ordering in when these nodes become relevant again, based on their level in the tree. It can be easily observed that nodes are always consumed in a last-in-first-out manner – the set is really a stack. After introducing all leaf nodes and completing the last round, the root node will be the only node left on the stack. Another important observation here is the fact that there are never two nodes of the same tree level on the stack at the same time. The nodes on the stack are inherently ordered by their tree level (nodes that occur higher in the tree are stored deeper down the stack). As leaves are consumed in subsequent order, two nodes at the same level have to be neighboring nodes that can immediately be used to produce their parent node. This allows us to conclude that using treehash for HORST requires a stack that can hold $log(t) = 16$ nodes. At 32 bytes per node, this easily fits in the available memory.

Besides computing the root node of a HORST tree, however, we are particularly interested in the nodes along the authentication paths from the leaves used to produce the signature, to the top of the tree. The position of these nodes on the stack is less easy to predict, but we do not want to compute parts of the tree more than once in order to gather all required nodes. Intuitively, a way to resolve this is by somehow recognizing the nodes that need to be included in the signature while performing the treehash rounds. Navigating through the tree without actually computing the node values is cheap, allowing us to trace the authentication paths from leaf to root and observe which nodes will need to be output. Rather than compiling a list of these nodes and performing costly lookups, we can compute and store in which treehash round they will be produced, as well as their position in the signature[5]. Algorithm 2 shows how to compute the round numbers of all nodes along the authentication path for a given leaf-node index.

[5] As nodes of the various authentication paths will be generated interleaved, it is necessary to rearrange them accordingly.

Algorithm 2 Computing treehash round numbers

Require: idx
Output: treehash round numbers of authentication nodes

$roundno \leftarrow idx + t$
$roundnumbers \leftarrow [\,]$
for $i \in \{1, \ldots, \log(t)\}$ **do**
  ▷ Find the neighbour node's round number..
  if $idx \bmod 2 = 1$ **then** ▷ (idx is a 'right-node')
    $roundno \leftarrow roundno - 2^{i-1}$
    $idx \leftarrow idx - 1$
  else ▷ (idx is a 'left-node')
    $roundno \leftarrow roundno + 2^{i-1}$
  end if
  $roundnumbers$.APPEND($roundno$)
  ▷ ..and move up to the parent node.
  if $idx \bmod 2 = 0$ **then**
    $roundno \leftarrow roundno + 2^{i-1}$
  end if
  $idx \leftarrow idx/2$
end for
return $roundnumbers$

Consider that the tree consists of $2^{17} - 1 = 131071$ nodes, but only 320 nodes[6] are relevant. Because of this, only a small subset of all treehash rounds contains relevant nodes. This makes it especially important to optimize recognizing relevant treehash rounds.

An efficient way to recognize which nodes need to be included in the signature while performing treehash is by storing bitmasks for each of the relevant rounds. By sorting these bitmasks by their round index, one can iterate over the mask-index pairs while processing each of the leaf nodes. Pointing an iterator at the current mask-index pair and only incrementing it when the index is equal to the index of the current leaf node will result in an overhead of only one comparison for each non-relevant round.

Streaming out signature data. In the previous section, we have glossed over an important aspect of the signing process: constructing the signature. Where an implementation with an abundance of memory available would simply allocate 41 KB of memory and insert the different pieces of the signature in the right place as they are computed, this is not possible on our device. Instead, the signature is streamed out of the board over the serial port throughout the computation. For many applications, this is not much different from receiving the entire signature

[6] One might expect to require $32 \cdot 16 = 512$ nodes, as each of the 32 authentication paths results in 16 neighboring nodes. However, in order to prevent needless duplication in the top layers, the HORST signature always includes layer 6 in its entirety and truncates the authentication paths after 10 nodes, leaving it to the verifier to reconstruct the paths.

all at once after the entire computation has finished, so we believe this should not pose any immediate usability concerns.

As was discussed in Section 2.4, the SPHINCS signature consists of a number of different components. Recall that $\Sigma = (idx, R, \sigma_H, (\sigma_{W,1}, Auth_1), \ldots, (\sigma_{W,d}, Auth_d))$, where, for brevity, H and W now denote the HORST FTS signature and the WOTS+ OTS signatures, respectively.

The values idx and R are generated at the start of the signing procedure, and can be written to the output stream immediately. The WOTS+ signatures $\sigma_{W,i}$ and sequences of nodes $Auth_i$ are generated in the same order as the order in which they are supposed to be arranged in Σ, so this does not lead to any difficulty, either – instead of storing them in memory, we simply write these values to the output stream as they are computed.

The HORST signature σ_H is a bit more complicated. It consists of k pairs of secret keys belonging to leaf nodes, and sequences of nodes along the path from each of these leaf nodes towards the top of the tree. As remarked in footnote 6 on page 16, all nodes on layer 6 are always included, so the last 6 nodes of these sequences are truncated. The issue here is the fact that the node sequences are not produced one at a time, but are each grown in an interleaved fashion as more and more of the tree is computed. When storing the hash values in a signature in memory, this does not pose a problem – each node value can be inserted in the right place. When streaming the output, however, one cannot go back and insert a node value. Instead, the node values will have to be tagged with what should have been their location in the signature, and rearranged accordingly on the receiving end of the communication. For each 32-byte node value, this adds an overhead of two bytes. While this may seem significant, in the end it results in an increase of 832 bytes (640 for the authentication path nodes, 128 for the nodes on layer 6 and 64 bytes for the secret keys). Considering that the entire SPHINCS signature is 41 KB, this can be considered acceptable.

HORST key material. Similarly, generating a HORST secret key (based on the seed SK_1 and its location in the hypertree) results in too much key material to fit in memory. With 2^{16} leaf nodes of 32 bytes each, this would amount to 2 MB. Instead, we can once more rely on the fact that treehash rounds consume the leaf nodes sequentially, and only generate the leaf node values when they are required. To achieve this, we briefly store the intermediate state of the ChaCha12 stream cipher instead, initially seeding it in the regular fashion for HORST. We then perform the next iteration based on the stored state whenever more key data is required. This allows for the generation of leaf node values on the fly. As ChaCha12 produces output blocks of 512 bits, every other leaf node requires a new chunk of output to be generated.

Streaming the message. On the subject of streaming data, it should also be remarked that for the Cortex M3 to be able to sign messages of a length larger than the available memory, it is necessary to process the message in a streamed fashion as well. This is possible, but requires the message to be streamed twice. In Section 2.4, we described that the message is first used together with the secret key to generate a message-specific random value R, after which the message

digest is generated. As this digest is computed as the hash of the concatenation of a part of R and the message (in that order), the message needs to be available twice. As storing it on the device is not an option for large messages, it needs to be streamed twice. The additional overhead is minimal as it can be streamed in block by block while performing the BLAKE512 hash using direct memory access.

4.2 Performance

The previous subsections show some of the adjustments required to be able to generate SPHINCS signatures on a platform with only 16 KB of volatile memory. Besides memory usage, time is also a relevant metric to consider. In fact, the running time very much determines usability in practice.

ChaCha permutation. When considering SPHINCS-256, one of the key observations here is the repeated use of the ChaCha permutation. It is the fundamental building block in both WOTS+ and HORST, as well as the hash trees that make up the rest of the hypertree.

Recall that $t = 2^{16}$. In order to generate a HORST key and produce a signature, ChaCha is used $\frac{1}{2} \cdot t = 32768$ times to expand the seed and generate the secret keys, as the permutation outputs 512 bits and the keys are 256 bits each. These secret keys are then hashed using F to construct the leaf nodes at the cost of another $t = 65536$ permutations. Subsequently, treehash is used to hash together t leaf nodes, at a cost of two ChaCha permutations per parent node (as follows from the construction of the function H in Section 2.9), resulting in another $2 \cdot (t - 1) = 131070$ permutations. All in all, this results in 229374 calls for one HORST signature.

WOTS+ is significantly cheaper. Recall that $\ell = 67$ and $w = 16$. Generating a WOTS+ key pair requires ℓ secret keys, which costs $\lceil \frac{1}{2} \cdot \ell \rceil = 34$ permutations to expand the seed, $\ell \cdot (w - 1) = 1005$ invocations of F at one permutation each for the chaining function and 66 invocations of H to build the L-tree, totaling $34 + 1006 + 2 \cdot 66 = 1171$ permutations. Each of the trees in the hypertree has 32 WOTS+ leaf nodes, costing a total of $32 \cdot 1171 = 37472$ permutations per tree.

Constructing a tree with WOTS+ key pairs on the leaf nodes costs an additional 31 invocations of H. One of the WOTS+ nodes is used to produce a signature on the sub-tree below, at the average cost of $\lceil \frac{1}{2} \cdot \ell \cdot (w - 1) \rceil = 503$ more invocations of F. As there are trees 12 in the hypertree, this leads to a total of $12 \cdot (37472 + 2 \cdot 31 + 503) = 456444$. Summing the cost of HORST and the WOTS+ trees, we arrive at a grand total of $229374 + 456444 = 685818$ permutations.

Because we perform so many ChaCha permutations, it is worthwhile to optimize this in ARMv7-M assembly. Internally, the ChaCha permutation operates on sixteen words of 32 bits each. These fit precisely in the 32-bit registers that are available to us on this platform, and the arithmetic in ChaCha is very simple to perform once the words are accessible. Additionally, many of the arithmetic operations come for free, as the ARMv7-M instruction set provides instructions

that take rotated registers as arguments. This enables us to perform nearly all of the rotation operations implicitly. There are not enough registers available for all of these words, though, as register 13, 14 and 15 are reserved for the stack pointer, link register and program counter, respectively. This would imply that three of the sixteen words would need to be saved in memory at all times, at the cost of a load and a store whenever one of these is needed. While we need the program counter and stack pointer for the code to run properly, we are not making any function calls that require the link register – the extra cost of having to pop it from the stack in the end is easily compensated by the benefit of an extra general purpose register. Now that we have fourteen registers to work with, we can arrange the order of the round internals of the ChaCha permutations such that we only need to switch out the two words on the stack once every round, on average. Doing so, we arrive at 542 cycles for one permutation; in the context of the ChaCha12 stream cipher, this corresponds to around 17 cycles per byte.

Key generation. Generating a SPHINCS-256 key on the device takes 28 205 671 cycles. At 32 MHz, this amounts to just below a second. As one would expect, virtually all of these cycles can be attributed to WOTS+ key generation. When it comes to key generation, it should be noted that the STM32L100C discovery board that we used for these benchmarks is not equipped with a random number generator. Instead, we used a hard-coded 32-byte value that is included when we flash the device. While in practice, using this board, key generation would have to be done off the board, our results show that for similar boards with a TRNG on-board key generation is not only feasible but practical.

Signing. Producing a signature takes 589 018 151 cycles, or approximately 18.41 seconds. As described above, we cannot store the signature on the board – this requires communication to a host outside of the board. Using direct memory access, we can efficiently interleave control of this communication with computations. If we disable communication and instead discard the signature as it is being produced, the signing procedure requires 584 384 791 cycles (for messages of small length, so as to focus the benchmark on penalty of signature output). This shows that the overhead is noticeable but not significant. In practice, this is a factor that may vary slightly depending on the specific context and interfaces available.

In terms of RAM usage, the signing procedure ends up using 8 755 bytes of stack space. Note that some of this stack usage is the result of function inlining by the compiler, to prevent having to perform function calls. When disabling this behavior, the stack space consumption is reduced to 6 619 bytes. Furthermore, we observe that the current implementation requires 25 KB of flash memory (or 19 KB, without inlining). These results show that there is a sufficient amount of memory left on the device (in terms of both RAM and ROM) for other applications, but also indicate that moving to even smaller devices (such as the Cortex M0) would be quite challenging.

Verification. Verification is much more straight-forward. The memory limit does not necessitate any significant changes like it did for signature generation,

as the verification procedure never requires the construction of a full tree. The signature needs to be streamed to the device, but this does not complicate processing, as the node values arrive in the order in which they are to be consumed. At 16 414 251 cycles, verification takes roughly 513 milliseconds. When ignoring the communication and operating on bogus data instead, verification requires 5 991 643 cycles. The communication penalty is in the same ballpark as the one incurred when signing, but still noticeably different. This can be accounted for by the way in which communication and computation can be interleaved in the two procedures: for verification, the windows in which communication can be performed are much smaller, making it more difficult to schedule the computation and communication efficiently.

5 The cost of eliminating the state

As has been discussed earlier, being able to compute digital signatures in a stateless configuration has definite advantages over a scheme that has to maintain a state. The advantages are generally focused around practical applicability. In order to be able to get rid of the state, however, SPHINCS pays a significant price. In this section, we will review just how much it costs to get rid of the state. We do this by comparing the performance of SPHINCS on the Cortex M3 to that of Multi Tree XMSS [16] on the same platform, configured in such a way that both schemes offer a similar security level using similar primitives.

5.1 XMSS^{MT}

XMSS^{MT} [16] uses the XMSS construction [5] in such a way that it is possible to sign a much larger number of messages before having to generate a new key. Depending on the specific parameters and practical application, this limit is virtually non-existent.

In essence, XMSS (and, by extension, XMSS^{MT}) is the stateful counterpart of SPHINCS. The high-level design is very similar, using WOTS+ leaf nodes to sign messages and including the authentication path in the signature, as well as adding bitmasks to the hash tree layers. The differences originate from the fact that XMSS uses the leaf nodes sequentially to guarantee that they are only used once, while SPHINCS selects them at random with a negligible chance of duplication. As we have discussed earlier, SPHINCS reduces this chance by adding a layer of HORST nodes underneath the WOTS+ leaves and by greatly increasing the tree size. In order to feasibly operate on such a large tree, SPHINCS includes layers of WOTS+ signatures to link different subtrees together. These linked subtrees are precisely how XMSS^{MT} is also able to increase its tree size (and thus the number of available leaf nodes).

Besides being able to work with a smaller, more efficient tree, going through the leaf nodes sequentially also allows us to re-use parts of the previous authentication path when generating a new signature. By storing the authentication path and the WOTS+ signature, only very few new nodes need to be computed

when the next signature is generated. The amount of work that needs to be done to update the authentication path varies wildly for the different leaf nodes, however, making the signing cost very diverse. In order to be able to efficiently compute each signature at the same costs, the authors of XMSS^{MT} suggest the use of the BDS traversal algorithm [7] with a distributed signature generation method[7].

BDS traversal. While this is not the right place to go into the precise details of the BDS algorithm [7], it is relevant and necessary to have a basic intuition. The goal of this algorithm is to have all the nodes for a certain authentication path available right when it is required, while still keeping the storage requirement to a minimum. This is done by maintaining an elaborate state and allocating 'updates' to each round (i.e. to be performed whenever an authentication path is returned). The state consists, among other structures, of instances of the treehash algorithm progressing through the current subtree, but also of work-in-progress instances of the next subtree, for each layer in the hypertree. The allocated updates are assigned to the treehash instances that have the most work to do relative to their deadline, guaranteeing that each node is produced when needed. Additionally, the algorithm configuration allows for a trade-off between the amount of work per update and the storage requirement by caching particularly expensive nodes high up in the trees.

For now, the main detail we need to make note of is the fact that in order to initialize the state and initial authentication path, it is necessary to compute the first full subtree on every layer. Additionally, it is relevant to remark that, for our implementation, changing from one subtree to the next is a more costly operation, as this requires a new WOTS+ signature to effectively link the new tree to the existing parent tree.

5.2 Parameters

XMSS^{MT} offers a diverse set of parameters to make different trade-offs between the runtime and storage requirements. For the parameters selection, we tried to conform to the settings proposed in the XMSS^{MT} Internet-Draft [15]. This let to the choice of $m = 32$ and $n = 32$ for the function output sizes, a tree with a total height of $h = 20$, $d = 2$ subtree layers and a Winternitz parameter $w = 16$ (resulting in a length of $\ell = 67$).

In terms of running time, the performance would have benefited significantly from a larger number of subtree layers, d. However, each layer d implies the need to store an additional WOTS+ signature, quickly exceeding our memory constraint. Moreover, a signature contains one WOTS+ signature per layer, increasing the signature size significantly. For the BDS algorithm, we choose $k = 6$. This allows us to cache a fairly large number of expensive nodes in the limited memory that is available.

[7] Where the costs for signature generation are equally distributed among all signature generations

In order to be able fairly compare XMSS^{MT} to SPHINCS-256, we do not use SHA-256 and SHA-512 to compute the message digest or the parent nodes in the hash trees. Instead, we rely on the BLAKE hash functions [1] for the message digest, and use a construction based on the ChaCha permutation similar to the ones described in Section 2.9 for the functions H and F, listed below. As in SPHINCS, let $C =$ "expand 32-byte to 64-byte state!", let $Chop(M, i)$ be a function that truncates M to i bits, π the ChaCha permutation, M_i strings of 256 bits and O a string of 256 zero-bits, then:

$$F(K, M) = Chop(\pi(\pi(K \| C) \oplus (M \| O)), 256)$$
$$H(K, M_1, M_2) = Chop(\pi(\pi(\pi(K \| C) \oplus (M_1 \| O)) \oplus (M_2 \| O)), 256)$$

For pseudo-random number generation, we replace ChaCha20 with ChaCha12, as this matches the choice for SPHINCS-256. All of this implies that we can use the same ARMv7-M assembly implementation of the ChaCha permutation that we used for SPHINCS.

5.3 Performance

The difficulty with an accurate performance estimate for XMSS^{MT} is that it highly depends on the practicalities of the platform it is deployed on, as well as the precise use-case. This is a result of the extra administration that comes with dealing with the state. As suggested above, part of the state is crucial for the security of the scheme (namely the index of the last processed leaf node), and while the structures that need to be stored for BDS traversal are needed for signing time optimization purposes. Writing persistent data is a relatively costly operation on most platforms, so different decisions will need to be made depending on use case specific requirements. On the STM32L100C, writing a well-aligned 4-byte word to non-volatile memory costs roughly 216 500 cycles on average, and scales linearly with the number of words written.

For our experiments, we assume that the device is powered on for a longer period of time, and is being queried for multiple signatures over this interval. This is an especially relevant scenario for XMSS^{MT}, as this is where the benefit of the BDS state comes into play most prominently.

Before outputting each new signature, it is necessary to write the updated secret key to persistent memory. This prevents re-use of a leaf node (and thus compromise of the key) when the power gets cut. As the BDS state is much larger and thus more expensive to store, it is only written to persistent memory when a graceful power-off occurs. In case this state is lost, it can be reinitialized based on the secret key seed and leaf node index. For the purpose of this comparison, this is considered out of scope.

Key generation and initialization. Compared to SPHINCS, the key generation phase for XMSS^{MT} is much more expensive, especially in the setting described here. The main reason for this is the fact that the two trees consist of 10 levels each, resulting in the computation of 2048 WOTS+ leaves (1024 on

each level). As mentioned above, the generation of two trees is necessary to initialize the BDS state. Additionally, a WOTS+ signature needs to be computed for the bottom tree. For the specified parameters, the initialization phase takes 8 857 708 189 cycles. Each WOTS+ leaf computation costs 4 299 598 cycles, and the WOTS+ signature costs 1 079 936 cycles. This accounts for most of the work, leaving only a small fraction for the hash trees.

Signing and verification. For signing, the cycle count is not precisely identical for each signature. The BDS algorithm tries to distribute costs equally among signature generations by running a fixed amount of treehash 'updates' for each signature. However, for the first few signatures not all these updates are needed as all structures are initialized during key generation and only few values have to be computed during each signature generation. It turns out that during this "start-up phase" it is slightly more costly to update the state for 'right' leaf nodes than for their 'left' neighbors, signatures using a left leaf node come in at 21 551 730 cycles, while right nodes cost 17 308 759 cycles. For our implementation, transitioning from one tree to the next does cost significantly more cycles than a regular signature: Signatures that require renewing the WOTS+ signature that binds the subtrees together cost 28 344 774 cycles. Overall, the average signing time is 19 441 021 cycles.

As one would expect of a hash-based signature scheme, verification is a much cheaper operation. At only 4 961 447 cycles, the relative gain in comparison to SPHINCS is not as dramatic as it is for the signing procedure, but it is still a significant difference.

6 Conclusions

Having to maintain a state for digital signature schemes can have several negative consequences. With this work, we have shown that it is feasible to run stateless hash-based signatures on a microcontroller, both in terms of performance and memory use. The fact that a SPHINCS signature itself does not fit in memory has proven to be a surmountable obstacle. This could make SPHINCS-256 well-suited as a cross-platform post-quantum signature scheme, especially when keeping a state is not a viable option.

However, we have also shown that eliminating the state does not come for free. While verification is fast for stateful and stateless schemes, signing with stateful XMSS is about 30 times faster than signing with stateless SPHINCS. The 589 018 151 cycles used by SPHINCs may be acceptable for non-interactive applications that need long term security and cannot maintain a state, but we believe that further algorithmic improvements to stateless hash-based signatures will be needed to enable deployment on a broader scale.

References

1. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. SHA-3 proposal BLAKE. *Submission to NIST*, 2008. https://131002.net/blake/

blake.pdf.
2. Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *LNCS*, pages 470–484. Springer, 1997. https://cseweb.ucsd.edu/~mihir/papers/tcr-hash.pdf.
3. Daniel J. Bernstein. ChaCha, a variant of Salsa20. SASC 2008: The State of the Art of Stream Ciphers, 2008. Document ID: 4027b5256e17b9796842e6d0f68b0b5e, http://cr.yp.to/papers.html#chacha.
4. Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Marc Fischlin and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015. Document ID: 5c2820cfddf4e259cc7ea1eda384c9f9, http://cryptojedi.org/papers/#sphincs.
5. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011. https://huelsing.files.wordpress.com/2013/05/mssgesamt.pdf.
6. Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 4521 of *LNCS*, pages 31–45. Springer, 2007.
7. Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *LNCS*, pages 63–78. Springer. https://www.cdc.informatik.tu-darmstadt.de/reports/reports/AuthPath.pdf.
8. Christoph Busold, Johannes Buchmann, and Andreas Hülsing. Forward secure signatures on smart cards. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography – SAC 2012*, volume 7707 of *LNCS*, pages 66–80. Springer, 2013. https://huelsing.files.wordpress.com/2013/05/xmss-smart.pdf.
9. Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *LNCS*, pages 109–123. Springer, 2008. https://www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/DOTV08.pdf.
10. Thomas Eisenbarth, Ingo Von Maurich, and Xin Ye. Faster hash-based signatures with bounded leakage. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 223–243. Springer, 2014. http://users.wpi.edu/~teisenbarth/pdf/SignatureswithBoundedLeakageSAC.pdf.
11. Oded Goldreich. Two remarks concerning the goldwasser-micali-rivest signature scheme. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *LNCS*, pages 104–110. Springer, 1987. http://theory.csail.mit.edu/ftp-data/pub/people/oded/gmr.ps.
12. Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: a signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 530–547. Springer, 2012. https://www.sha.rub.de/media/sh/veroeffentlichungen/2014/06/12/lattice_signature.pdf.

13. Andreas Hülsing. *Practical Forward Secure Signatures using Minimal Security Assumptions*. PhD thesis, TU Darmstadt, 2013. http://tuprints.ulb.tu-darmstadt.de/3651/.
14. Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013. https://huelsing.files.wordpress.com/2013/05/wotsspr.pdf.
15. Andreas Hülsing, D. Butin, S. Gazdag, and A. Mohaisen. Xmss: Extended hash-based signatures draft-irtf-cfrg-xmss-hash-based-signatures-01. Crypto Forum Research Group Internet-Draft, 2015. https://tools.ietf.org/html/draft-irtf-cfrg-xmss-hash-based-signatures-01.
16. Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128 of *LNCS*, pages 194–208. Springer, 2013. https://huelsing.files.wordpress.com/2013/04/xmss-optimal.pdf.
17. Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.
18. ARM Limited. *ARMv6-M Architecture Reference Manual. Document ID: ARM DDI 0419C*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419c/.
19. ARM Limited. *ARMv7-M Architecture Reference Manual. Document ID: ARM DDI 0403E.B*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.b/.
20. ARM Limited. Cortex-m0 processor – ARM. http://www.arm.com/products/processors/cortex-m/cortex-m0.php.
21. Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1990. www.merkle.com/papers/Certified1979.pdf.
22. Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In *Design Automation Conference – DAC 2014*, pages 1–6. ACM, 2014. https://www.sha.rub.de/media/attachments/files/2014/06/bliss_arm.pdf.
23. Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Batten and Jennifer Seberry, editors, *Information Security and Privacy 2002*, volume 2384 of *LNCS*, pages 1–47. Springer, 2002. http://www.cs.bu.edu/~reyzin/papers/one-time-sigs.pdf.
24. Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast hash-based signatures on constrained devices. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *LNCS*, pages 104–117. Springer, 2008. https://www-old.cdc.informatik.tu-darmstadt.de/reports/reports/REDBP08.pdf.
25. Bo-Yin Yang, Chen-Mou Cheng, Bor-Rong Chen, and Jiun-Ming Chen. Implementing minimized multivariate pkc on low-resource embedded systems. In John A. Clark, Richard F. Paige, Fiona A. C. Polack, and Phillip J. Brooke, editors, *Security in Pervasive Computing*, volume 3934 of *LNCS*, pages 73–88. Springer, 2006. http://precision.moscito.org/by-publ/recent/39340073.pdf.