Short-term state in SPHINCS

by Moritz Neikes

February 2020

Master Thesis in Computer Science
Supervised by dr. Peter Schwabe
Second reader: dr. Andreas Hülsing
Radboud University Nijmegen

moritz@post-apocalyptic-crypto.org

Preface

The work on this thesis started in early 2016, and was finished around September 2017, aside from minor issues. However, before I was able to complete and publish this thesis, I had to take a break for mental health reasons. In December 2019, I resumed my work, and wrapped up any outstanding issues by early February 2020.

Because of that, the research in this thesis is, for the most part, based on the current state of research around hash-based signatures as it was in 2017, and newer developments, including the work on SPHINCS$^+$ (see `https://sphincs.org`), have not been taken into account.

In particular, there is significant overlap between the work in "Improving Stateless Hash-based Signatures" by Aumasson and Endignoux [2] and my thesis, but by the time their paper was published, on September 25, 2017, the vast majority of my research was already completed, reaching similar conclusions.

In this context, I would like to refer to the git histories of the two projects I created during my work on this thesis, which date back to April 2016. The histories can be found here:

- For my work on SPHINCS short-term states:
  `https://github.com/25A0/sts-sphincs/commits/master`
- For my work on authentication sets:
  `https://github.com/25A0/authentication-sets/commits/master`

I add this preface for the sake of transparency, and to avoid confusion about the overlap between my work and the work of Aumasson and Endignoux.

Abstract

There is an ongoing and well-funded effort to build practical quantum computers. Whether and when this effort will bear fruit is difficult to predict, but if it does, then many widely deployed cryptographic signature schemes will break. Practical post-quantum cryptography is still rare, but not unheard of. SPHINCS is a practical post-quantum stateless hash-based signature scheme that is built on well-understood cryptographic primitives. Its stateless character makes it easy to use in practice, but also introduces some inefficiency.

This thesis explores two ways in which a short-term state can be added to SPHINCS. As a middle ground between statefulness and statelessness, a short-term state can combine a significant performance boost with the ease of use that comes with stateless signature schemes. Furthermore this thesis introduces a novel way to combine authentication paths in binary hash trees. This has a practical application in HORST, a hash-based signature scheme that is used as a building block in SPHINCS.

Acknowledgments

I want to thank Peter Schwabe for his continuous support and trust, not only for this thesis, but throughout the entirety of the past years. I greatly appreciate his guidance, which has always been a valuable and reassuring anchor for me. I also want to thank Andreas Hülsing for his helpful explanations and valuable feedback, and for his willingness to remain part of the project despite some serious delays along the way.

I would like to thank Meike for her invaluable support and encouragement, and for being so understanding throughout the stressful final weeks of the project. Finally, I want to thank my friends and family for their valuable support in various ways, and in particular Maurice Knoop, for his support and encouragement, and for always being ready to provide constructive feedback.

Contents

1 Introduction **3**
1.1 Cryptographic hash functions 5
1.2 Hash-based signature schemes 6
1.2.1 Stateful vs stateless signature schemes 6
1.3 Contributions and overview 7

2 SPHINCS **9**
2.1 Hash trees 10
2.1.1 Authentication paths 10
2.1.2 L-trees 12
2.2 The WOTS signature scheme 12
2.2.1 WOTS Key generation 14
2.2.2 WOTS Signing 14
2.2.3 WOTS Verification 14
2.3 The HORST signature scheme 15
2.3.1 HORST Key generation 15
2.3.2 HORST Signing 16
2.3.3 HORST Verification 16
2.3.4 The vagueness of “few” 16
2.4 simple-SPHINCS 17
2.4.1 The SPHINCS hypertree 17
2.4.2 Key generation 20
2.4.3 Signing 20
2.4.4 Verifying 21
2.4.5 simple-SPHINCS-256 21
2.4.6 Differences to SPHINCS 22
2.5 Multi-target attacks in SPHINCS 22
2.5.1 Multi-target attack surfaces in SPHINCS 23
2.5.2 Mitigating multi-target attacks 25

3 Reducing HORST signature size **27**
3.1 HORST signature size in SPHINCS 27
3.2 HORST Authentication sets 28
3.2.1 Combining authentication paths 28
3.2.2 Authentication-set size 30
3.2.3 Best-case and worst-case scenarios 32
3.3 Implementing HORST with authentication sets 36
3.3.1 Naïve implementation 36

3.3.2 Efficient implementation 37
3.3.3 Using authentication sets in HORST 38
3.4 Practical aspects of HORST authentication sets 40

4 Two approaches to short-term state in SPHINCS 43
4.1 The benefit of state in SPHINCS 43
4.2 Challenges of stateful signature schemes 44
4.3 Short-term state in SPHINCS 45
4.4 Approach 1: Iterate through lowest subtree 46
4.4.1 Impact on security assumptions 47
4.5 Approach 2: Add a short-term subtree below HORST 51
4.5.1 Impact on security assumptions 52
4.6 The two approaches in comparison 53
4.7 Parameter discussion 53
4.7.1 Sequential batch signing 54
4.7.2 Subtree batch signing 54
4.8 Creating successive short-term states 56

5 Implementation and results 58
5.1 Benefits of batch signing 58
5.2 Results 59
5.2.1 Utilization 60
5.3 Conclusion 61

A Fast and flexible heterogeneous buffers in C 69

Chapter 1

Introduction

Cryptographic signatures have become an essential building block of digital information exchange today. They are used to authenticate websites, software and updates, and sometimes e-mails. But many of the signature schemes that are in use today will be broken with the dawn of sufficiently powerful quantum computers.

On the one hand, Shor's algorithm [47] can be used on quantum computers to factor integers, and to compute discrete logarithms in polynomial time. On classical computers, these problems can be solved in exponential time at best. Because of that, Shor's algorithm breaks many commonly used cryptographic signature schemes [30], most prominently RSA [46] and elliptic curve cryptography, but also Diffie-Hellman key exchange [18].

On the other hand, Grover's algorithm [28] can be used to find a specific element in an unordered database of 2^n elements in $\mathcal{O}(\sqrt{2^n}) = \mathcal{O}(2^{n/2})$ steps with a $> 50\%$ success rate. Determining whether an element satisfies the search condition should be possible in a single step. To some degree, this problem is similar to symmetric encryption: An n bit key is just one entry in a database of 2^n elements containing all possible keys. An attacker wants to find the one element that deciphers a given ciphertext. Under Grover's algorithm, an n bit key would then only deliver $n/2$ bits of security against a quantum-aided attack. Because of that, it is argued that doubling the key size should be sufficient to outweigh the advantage of a quantum-aided attack [7].

While the impact of Grover's algorithm is not as devastating as the impact of Shor's algorithm, both need to be taken into consideration when designing cryptographic algorithms for a world in which sufficiently powerful quantum computers exist. All publicly known advances in actually building quantum computers are still far off the number of qubits that are required to break commonly used cryptography today, but we continue to see significant improvements [11, 1]. At the same time, new algorithms are discovered that might lower the number of qubits necessary to break currently used signature schemes [8, 43].

This is not the first time that widely adapted cryptographic algorithms are at the risk of being broken. Many early hash algorithms have been broken, and

have been replaced by better ones, but this process takes time.

Furthermore, someone in possession of a powerful quantum computer can not only tinker with today's signatures, but they could also try to recover secret keys from pre-quantum signatures used in the past. This might allow them to distribute malicious or modified firmware, documents, or mobile applications with forged signatures. The sooner data is signed with post-quantum signature schemes, the sooner will these attack vectors disappear.

Thus, even though quantum computers might not be able to break cryptographic algorithms today, there is plenty of motivation to move towards post-quantum cryptographic algorithms sooner rather than later. There are, however, not many options for post-quantum algorithms that are widely adapted at this point. NIST makes an effort to change this with a competition for post-quantum public-key algorithms[1].

SPHINCS [9], a stateless hash-based signature scheme, fits the goal of this competition. This thesis attempts to make it even more appealing to transition from traditional signature schemes to a post-quantum signature scheme, by speeding up successive signatures creation in SPHINCS. This is especially useful for server environments to generate many signatures with little latency. The solution presented in this thesis can speed up the signature creation by a factor of 30, and reduces the size of public and private keys.

In addition to hash-based signature schemes, there are other approaches to post-quantum signature schemes.

Lattice-based signature schemes. Given n linearly independent basis vectors $b_1, \ldots, b_n \in \mathbb{R}^m$, an n-dimensional lattice L is the set of all integer linear combinations of these vectors: $L(b_1, \ldots, b_n) = \{\sum_{i=1}^{n} x_i b_i, x \in \mathbb{Z}\}$. For example, $\mathbb{Z}^3$ is a three-dimensional lattice that can be formed from the basis vectors $(1, 0, 0), (0, 1, 0), (0, 0, 1)$, but there are many other basis vectors from which $\mathbb{Z}^3$ could be formed, too.

Amongst the various lattice problems, there are a couple that are of particular interest for cryptography: In the *Shortest Vector Problem* (SVP), the task is to find the shortest non-zero vector in a lattice L, given a basis of a vector space. In the *Closest Vector Problem* (CVP), given a lattice L, and a vector v in $\mathbb{R}^n$ but not necessarily in L, the task is to find a vector in L that is closest to v. Finally, in the *Shortest Independent Vector Problem* (SIVP), given a lattice L of dimension n, the task is to find n linearly independent vectors that are shorter than all basis vectors.

The hardness of these problems, and many variants, has been studied extensively (see [49] for an overview). Lattices are well-studied in general, and for their use in cryptographic constructions in particular [23, 15], including the impact that quantum computers may have on their security [36].

The first lattice-based signature scheme by Goldreich et al. [26] has since been broken [41], but new and yet unbroken signature schemes have been proposed [24, 38, 29, 21].

[1] `https://csrc.nist.gov/projects/post-quantum-cryptography`

Multivariate signature schemes. Multivariate cryptography is centered around the problem of finding a vector $\mathbf{x} \in \mathbb{F}^n$ that solves a set of m polynomial equations in n variables each. If the polynomials are of degree two, then these systems are called $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic ($\mathcal{MQ}$) equations, a problem that is known to be NP-complete for non-trivial n and m [50].

The first multivariate signature scheme was $C*$, designed by Imai and Matsumoto [34], and later broken by Patarin [42]. Patarin later published different designs himself, and some modern multivariate signature schemes are still based on some of Patarin's concepts.

Multivariate cryptography is well-suited for devices with tight resource constraints [10], and has also been successfully implemented in hardware [3, 51]. While signature sizes in $\mathcal{MQ}$ signature schemes tend to be very small (in the range of a few hundred bytes), public keys tend to be rather big [50].

Code-based signature schemes. Code-based public key encryption was keyed by McEliece in 1978 [39], and is built atop the Syndrome Decoding problem, a problem that is known to be NP-complete [6].

McEliece-type public key encryption systems have been studied extensively over the last forty years, the theory around code-based cryptography is generally well understood, and "no significant quantum algorithmic developments appear to be directly relevant to these decoding problems" [19, p. 762]. This makes code-based public key encryption a promising method for post-quantum cryptography, and variants of McEliece-type public key encryption systems exist for signature schemes [16, 4].

1.1 Cryptographic hash functions

Hash functions typically map data of arbitrary size to a fixed-size output. They are used in countless applications, ranging from data structures like hash tables to version control systems like git.

Cryptographic hash functions are fundamental building blocks of digital signatures and message authentication codes. Cryptographic hash functions are hash functions that meet specific requirements. These requirements typically include a subset of the following properties:

Collision resistance. It should be infeasible to find x, x' such that $x \neq x'$ but $H(x) = H(x')$.

Preimage resistance. Given $H(x)$, it should be infeasible to find x.

Second preimage resistance. Given x, it should be infeasible to find $x', x' \neq x$, such that $H(x) = H(x')$.

Undetectable. Let $H : D \rightarrow \{0, 1\}^n$ be a hash function. H is undetectable if an adversary cannot distinguish $H(x)$ from an element of $\{0, 1\}^n$ chosen uniformly at random.

The exact requirements depend on the applications, but most applications require the cryptographic hash function to be (second) preimage resistant.

1.2 Hash-based signature schemes

Hash-based signature schemes are constructed around properties of cryptographic hash functions. Hash-based signature schemes are often agnostic of the used hash function. If one particular hash function is broken, it can usually be replaced by another hash function.

In [37], Lamport presented the first hash-based signature scheme. Take two values, x_0, x_1, and a cryptographic hash function H, and compute $y_0 = H(x_0), y_1 = H(x_1)$. Treat x_0, x_1 as the secret key, and y_0, y_1 as the public key. To sign a single bit b, publish x_b. The verifier can then check $H(x_b) \stackrel{?}{=} y_b$. Given that the used hash function offers (second) preimage resistance, it is infeasible for an attacker to find x'_b such that $H(x'_b) = y_b$.

Lamport's scheme is as secure as the underlying hash function, but leads to impractical key and signature sizes. Winternitz improved Lamport's scheme by repeatedly applying the hash function to sign more bits at once (as mentioned in [40, p. 227]). With Winternitz' improvement, the scheme can offer more reasonable signature and key sizes. However, both of these schemes have a limitation that is unusual compared to signature schemes commonly used today: Every key pair in the Lamport or Winternitz signature scheme can be used for exactly one signature, since the signature reveals part of the secret key. This limitation makes these so-called one-time signature (OTS) schemes hard to use in practice by themselves.

In [40], Merkle presented a way to combine an arbitrary number of OTS key pairs into a single key pair. In this Merkle signature scheme (MSS), a fixed number of messages can then be signed with each key pair. Over the years, many different improvements of MSS have been published [12, 33, 32, 14].

1.2.1 Stateful vs stateless signature schemes

Signature schemes that are commonly used today (RSA [46], ECDSA [35], Ed25519) are stateless: signing a new message does not depend on messages previously signed with the same key pair. In contrast to that, many hash-based signature schemes are stateful, in that previously created signatures change how a new signature is created.

One-time signature schemes (OTSS) like the Lamport OTSS, or the Winternitz OTSS, are inherently stateful, as they can only be used once.

Other signature schemes offer a limited number of signatures with each key pair. For example, the Merkle signature scheme [40] offers n signatures that can be generated with each key pair, by associating n OTS public keys with the public key of a Merkle signature scheme key pair, and using a previously unused OTS

for each new signature. This signature scheme is stateful since the signer has to distinguish used from unused OTS key pairs.

Using stateful signature schemes introduces some challenges, compared to stateless signature schemes: The state of a stateful signature scheme has to be handled with care. If it is ever not properly updated after creating a signature, or if it is restored from a backup, previously used key material might be re-used, potentially breaking the scheme. Furthermore, a single key pair of a stateful signature scheme cannot be used on multiple CPUs or threads without synchronizing the state between them, which might harm performance. These factors do not need to be considered when using stateless signature schemes.

However, in some cases the stateful nature of these signature schemes also allows specific optimizations: The BDS tree traversal algorithm (based on [13]) is used in many implementations of stateful hash-based signature schemes [12, 32], and speeds up signature creation by caching the results of some intermediate computations.

In [25], Goldreich proposes a stateless ("memoryless") variation of the stateful Goldwasser-Micali-Rivest signature scheme [27]. In the GMR scheme, a large set of OTS key pairs is associated with each key pair. The GMR scheme is stateful since a new signature would depend on the number of messages that were previously signed with the key pair.

When signing a message with Goldreich's scheme, however, the digest of that message determines which of those OTS key pair is used in the signature. The security of this scheme is limited by the collision resistance of the hash function that is used to choose the OTS key pair. As collision resistance underlies the birthday paradox, using a hash function with n bits of output will deliver at most $2^{n/2}$ bits of security. This limitation leads to impractical signature sizes [9, p. 4]. SPHINCS [9] significantly improves Goldreich's design with greatly reduced signature size and signing speed, while keeping the stateless nature of the original scheme.

In both SPHINCS and Goldreich's scheme, signing speed could be improved if there was a way to store the results of reoccurring intermediate computations. That, however, is not possible without introducing some sort of state, which would re-introduce all the difficulties related to using stateful signature schemes.

A hybrid model, in which the results of some computations are stored in a short-term state, could benefit from both, the ease-of-use of a stateless signature scheme, as well as the improved signing speed from the ability to cache the results of some reoccurring computations. This is the central idea that is explored in this thesis.

1.3 Contributions and overview

This work explains and compares two ways in which a short-term state can be added to SPHINCS to significantly speed up successive signatures. Both approaches to short-term state have been implemented in C and are available for download. See `https://github.com/25a0/sts-sphincs` for code, in-

structions, and examples. This work also introduces a novel way to reduce the size of HORST signatures without compromising security. For this, see `https://github.com/25a0/authentication-sets` for code and instructions.

Chapter 2 explains SPHINCS, the stateless hash-based signature scheme introduced in [9]. Chapter 3 introduces reduced signature sizes for HORST. Chapter 4 explains two approaches to add short-term state to SPHINCS, and draws a comparison between the two. Chapter 5 discusses practical results.

Chapter 2

SPHINCS

This chapter explains a simplified version of the hash-based signature scheme SPHINCS, the complete version of which is introduced in [9]. SPHINCS uses two other hash-based signature schemes as building blocks: WOTS and HORST. Section 2.1 explains hash trees, a structure that is widely used throughout SPHINCS. Section 2.2 explains the one-time signature scheme WOTS, and Section 2.3 explains a simplified version of the HORST few-times signature scheme. Finally, Section 2.4 explains how all these building blocks can be combined to form a simplified version of SPHINCS, *simple-SPHINCS*. The crucial differences between the full version of SPHINCS proposed in [9], and the simplified version are listed in Section 2.4.6.

Notation. Throughout this chapter, binary trees – trees where each parent node has at most two child nodes – are used in multiple places. When addressing the nodes in these trees, the following notation will be used. The single node that has no parent node is called the root node of the tree. The nodes that have no child nodes are called leaf nodes. The height h of a tree is the maximum number of edges that form the shortest path from any leaf node to the root node of the tree. A perfect binary tree is a binary tree where each node has either two or no child node, and where the length of the shortest path from a leaf node to the root of the tree is the same for all leaf nodes. A perfect binary tree of height h has $2^{h+1} - 1$ nodes, of which 2^h are leaf nodes. Nodes can be divided into layers. Layer 0 contains the leaf nodes of the tree, layer 1 contains all parent nodes of those leaf nodes, and so on. Layer h contains the root node. Note that a binary tree of height h has thus $h + 1$ layers.

In a perfect binary tree, layer $l, 0 \leq l \leq h$ contains 2^{h-l} nodes. On each layer, the nodes can be indexed from left to right, having node 0 at the very left, and node $2^{h-l} - 1$ at the very right end of the layer. Combining these two notations, each node can be uniquely addressed as $N_{l,i}$, where l is the layer of the node, and i is the index of the node within that layer.

If a node $N_{l,i}$ has a parent node, then it can be addressed as $N_{l+1,\lfloor i/2 \rfloor}$. If a node $N_{l,i}$ has child nodes, then they can be addressed as $N_{l-1,2i}$ and $N_{l-1,2i+1}$.

2.1 Hash trees

Hash trees were first mentioned in a 1990 paper by Merkle [40]. They are binary trees, in which each parent node is formed by computing the hash of the concatenation of its two child nodes. Contrary to intuition, these trees are thus *grown* from their leaf nodes to the root node.

Given 2^h inputs $i_0, \ldots, i_{2^h-1}$ of size k, where $h, k \in \mathbb{N}$, and a cryptographic hash function F, the inputs $i_0, \ldots, i_{2^h-1}$ form the 2^h leaf nodes of a binary hash tree of height h. Starting with layer 1, the value of each node $N_{l,i}$ can be computed as $F(N_{l-1,2i}||N_{l-1,2i+1})$, where $||$ denotes concatenation.

Because of the preimage resistance of the used cryptographic hash function, it is infeasible to obtain the value of the child nodes, given the value of the parent nodes.

2.1.1 Authentication paths

In the context of SPHINCS, a common task is to compute the root node of a hash tree, given one of its leaf nodes. Of course this can be done by publishing all other leaf nodes of that hash tree. However, if the sole goal is to compute the root node, then it is sufficient to reveal a smaller set of nodes: A given leaf node, together with its sibling node, can be used to compute the parent node of these two leaves. With the sibling of this newly restored node, in turn, their parent node can then be restored. With this pattern, all nodes can be restored that are on the shortest path from the given leaf node to the root of the tree. The set of sibling nodes that are required for this is called the *authentication path* of the given leaf node.

Figure 2.1 shows the authentication path of a leaf node in a binary tree of height 3.

To compute the authentication path of a leaf node $n_{0,i}$, one can first enumerate the first h nodes along the shortest path from the leaf node to the root of the tree: $n_{0,i}, n_{1,\lfloor i/2 \rfloor}, n_{2,\lfloor i/2^2 \rfloor}, \ldots, n_{h-1,\lfloor i/2^{h-1} \rfloor}$. Note that this excludes the root node itself. The siblings of all these nodes form the authentication path of leaf $n_{0,i}$. The sibling of a node $n_{l,i}$ is the node $n_{l,i\oplus 1}$, where $\oplus$ denotes binary XOR. The authentication path is thus the set $\{n_{0,i\oplus 1}, n_{1,\lfloor i/2 \rfloor \oplus 1}, n_{2,\lfloor i/2^2 \rfloor \oplus 1}, \ldots, n_{h-1,\lfloor i/2^{h-1} \rfloor \oplus 1}\}$.

To summarize, the authentication path for any leaf node of a binary tree of height h contains h nodes – one for each layer, except for layer h which contains the root node. This means that the number of elements in the authentication path grows linearly as the height of the hash tree grows, while the number of nodes in the tree grows exponentially. Thus, while the root node can also be restored from a given leaf node by revealing all other leaf nodes, it is more efficient to only reveal the nodes along the authentication path of a leaf node.

While not relevant in the context of hash-based signatures, it is interesting to note how using an authentication path shifts around the workload between the party that reveals the node values and the party that wishes to recompute the

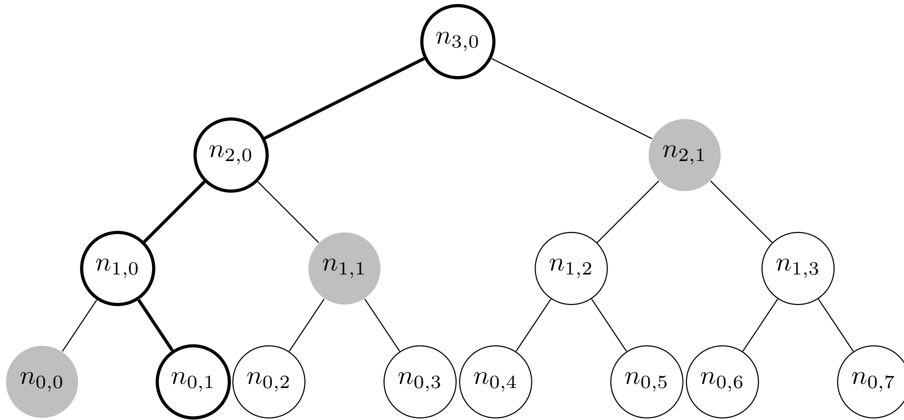

Figure 2.1: The outlined nodes and edges show the path from node $n_{0,1}$ to the root of the tree. The light-gray nodes show the authentication path of node $n_{0,1}$.

root node from the given leaf node. First, note that it requires $2^h - 1$ hashes to compute the root of a binary hash tree from its 2^h leaf nodes, since a binary tree of height h has $2^h - 1$ non-leaf nodes. The number of hashes that are required to compute the elements of the authentication path, on the other hand, is not as intuitive. Note that in Figure 2.1 there is one node on each layer $l, 1 \leq l \leq h$ of the tree that does not need to be computed in order to obtain all the nodes of the authentication path of $n_{0,1}$. Thus, while it takes $2^h - 1$ hashes to compute all nodes of a binary hash tree, it only takes $2^h - 1 - h$ hashes to only compute the authentication path of any given leaf node. It takes then another h hashes to compute the root of the binary tree from a given leaf node and the h elements of its authentication path. Since $2^h - 1 - h + h = 2^h - 1$, the total number of hashes that need to be computed stays the same, no matter if an authentication path is used or all leaf nodes are revealed.

Even though using authentication paths does not reduce the total number of hashes that need to be computed, it does make a difference for the workload of the two involved parties. When revealing the 2^h leaf nodes, there is no additional work for the party that reveals the node values, but the verifying party needs to compute $2^h - 1$ hashes to restore the root node. With authentication paths, the revealing party needs to produce the node values along the authentication path, which requires the computation of $2^h - 1 - h$ hashes to construct almost the entire hash tree. The verifying party, however, then only needs to compute h hashes to restore the root node from the leaf node and the authentication path.

Furthermore, using authentication paths reduces the amount of data that needs to be communicated between the two parties. Instead of sending out the values of the 2^h leaf values, only h values need to be communicated.

Thus, the total number of hashes that need to be computed is the same in both scenarios, but the workload is distributed differently, and the number of node values that need to be communicated between two parties is vastly smaller when using authentication paths rather than providing all leaf nodes.

2.1.2 L-trees

Perfect binary trees are simple, but only work in cases where the number of leaves is a power of two. L-trees can be used in all other cases [17]. These trees are similar to perfect binary trees, but nodes that do not have a sibling are raised to the next layer. For example, seven leaf nodes could be fitted in a binary tree of height 3 like the one shown in Figure 2.1, but Node $n_{0,7}$ would be missing. In that case, Node $n_{0,6}$ would not have a sibling, and would be raised to the next layer to replace Node $n_{1,3}$. With this transition, each non-leaf node still has exactly two child nodes.

To build an L-tree for k leaf nodes, start with a binary tree of height $h = \lceil \log_2(k) \rceil$, and place down the k leaf nodes $n_{0,0}, \ldots, n_{0,k}$. Missing leaf nodes or nodes that have no child nodes are omitted.

Initially, $k / \lceil 2^l \rceil$ nodes remain on layer l. To determine which of the remaining nodes need to be raised, compute $d = 2^h - k$. The binary representation of d indicates which nodes need to be raised. A `1` at bit i indicates that the rightmost node on layer i has no sibling and needs to be raised. Here, bit 0 is the least significant bit, and bit h is the most significant bit. If k is a power of two, then this construction will produce a binary hash tree.

2.2 The WOTS signature scheme

This section will give a high-level overview of the Winternitz One-Time Signature scheme (WOTS) [31]. The section is divided into two parts. The first part shows the basic building blocks of WOTS and how a naïve signature scheme can be built from that.

The second part points out the flaw that breaks this naïve approach, explains how it can be fixed using a checksum, and finally describes how key generation, signing, and verification works in WOTS.

At its core, WOTS is built atop a cryptographic hash function F, which is one-way, undetectable, and collision-resistant [20, p. 101]. Given a cryptographic hash function $F : \{0,1\}^k \rightarrow \{0,1\}^k$, which fulfills these requirements, and input x, it is trivial to obtain $F(x)$, given x, but it should be infeasible to obtain x given $F(x)$. Based on this, WOTS uses a chaining function $C(x, i)$, where $C(x, 0) = x$, and $C(x, i) = F(C(x, i-1))$ where $i > 0$. It is trivial to obtain $C(x, i+1)$ given $C(x, i)$, but it should be infeasible to obtain $C(x, i-1)$ given $C(x, i), i > 0$.

From this, one could build a simple one-time signature scheme for inputs of size n. Choose a secret key $s \in \{0,1\}^k$, and publish public key $p = C(s, 2^n)$. To sign a message $m \in \{0,1\}^n$, publish $c = C(s, m)$. One can verify this signature by checking that $C(c, 2^n - m) = p$.

In practice, this signature scheme would be unreasonably slow for reasonable message sizes. A message size of 32 bits would require computing up to 2^{32} hashes. To fix this, the scheme can be adapted to use multiple hash chains. Using t chains instead of one, the signature scheme

would look like this: The signer chooses t secret-key components of k bits each to form secret key $s = (s_0, \ldots, s_{t-1})$, and publishes public key $p = (C(s_0, 2^{n/t}), \ldots, C(s_{t-1}, 2^{n/t}))$. To sign a message m, the signer splits the message into t components $m = (m_0, \ldots, m_{t-1})$ of n/t bits each, and publishes $c = (C(s_0, m_0), \ldots, C(s_{t-1}, m_{t-1}))$. This signature can then be verified by checking that $p = (C(c_0, 2^{n/t} - m_0), \ldots, C(c_{t-1}, 2^{n/t} - m_{t-1}))$. Note that the number of hashes that need to be computed shrinks exponentially when t increases, while the signature size only grows linearly.

However, both of these naïve schemes have a devastating flaw: Given a signature of message m, an attacker can easily forge valid signatures for some messages other than m. For the simple scheme, where only one chain is used, an attacker can forge a signature for any message $m' > m$. Given signature $c = C(s, m)$, the attacker computes $c' = C(c, m' - m)$, which is a valid signature for m'. Signatures for any message $m'' < m$ cannot be forged, since the attacker cannot feasibly compute $c' = C(c, m'' - m)$, since $m'' - m$ is negative for any $m'' < m$.

For the more complex scheme, where t chains are used, an attacker can also forge a signature for some messages. Let $m' \gg m$ denote that for all message components $m'_i, m_i, i \in [0, t-1]$ it holds that $m'_i \geq m_i$, and that there exists some $j \in [0, t-1]$ such that $m'_j > m_j$. Given signature $c = (c_0, \ldots, c_{t-1})$ of message $m = (m_0, \ldots, m_{t-1})$, an attacker can then forge a signature for any message $m' \gg m$. For these messages, all $m'_i - m_i$ are non-negative, allowing the attacker to compute $c' = (C(c_0, m'_0 - m_0), \ldots, C(c_{t-1}, m'_{t-1} - m_{t-1}))$, which forms a valid signature of m'.

Signatures for any other messages cannot be forged with this attack; any message m'' which has at least one message component $m''_i < m_i$ would require the attacker to compute $C(c_i, m''_i - m_i)$, where $m''_i - m_i < 0$, which they cannot feasibly do.

To prevent the attack described above, a checksum is introduced. Given message m, the checksum H is computed as $H(m = (m_0, \ldots, m_{t-1})) = \sum_{i=0}^{t-1} h(m_i)$, where $h(m_i) = (2^{n/t} - 1 - m_i)$. Note that for any m_i, m'_i it is the case that $m'_i \geq m_i \rightarrow h(m'_i) \leq h(m_i)$, and that $m'_i > m_i \rightarrow h(m'_i) < h(m_i)$.

With this checksum, the signature scheme can be fixed by signing not the message itself, but the concatenation of the message and its checksum. For this, the signer needs an additional secret key component s_h, of k bits. Signatures are then created as so: $c = (c_0, \ldots, c_{t-1}, c_h) = (C(s_0, m_0), \ldots, C(s_{t-1}, m_{t-1}), C(s_h, H(m)))$.

An attacker is still able to forge valid signature components for the message components of a message $m' \gg m$ given the signature of message m, but the attacker will fail to also sign the checksum $H(m')$: Recall that $m' \gg m$ implies two properties:

1. For all message components, it holds that $m'_i \geq m_i$.
2. There is a message component $j \in [0, t-1]$ such that $m'_j > m_j$.

The first property implies that $\forall i \in [0, t-1], h(m'_i) \leq h(m_i)$. Looking at the sum of these values, we see that $\sum_{i=0}^{t-1} h(m'_i) \leq \sum_{i=0}^{t-1} h(m_i)$, and thus $H(m') \leq H(m)$.

With the second property, however, we know that $\exists j \in [0, t-1]$ such that $m'_j > m_j$, and thus $h(m'_j) < h(m_j)$. But in that case, $\sum_{i=0}^{t-1} h(m'_i) < \sum_{i=0}^{t-1} h(m_i)$, and thus $H(m') < H(m)$ and $H(m') - H(m) < 0$.

Since it is not feasible for the attacker to produce $C(c_h, H(m') - H(m))$, they cannot forge a signature for any message $m' \gg m$, given the signature of m.

The remainder of this section will describe how these building blocks are used to form the one-time signature scheme WOTS.

Given a message length of n bits, choose a chain length w as a trade-off between signing speed and signature size. Reducing the chain length increases the signature size linearly, but also increases the signing speed exponentially. Typically, w is a power of two, so it can be written as $w = 2^b, b \in \mathbb{N}$. Furthermore, define $l_1 = \lceil n/b \rceil$, $l_2 = \lceil \lceil \log_2(l_1(w-1)) \rceil / b \rceil$, and $l = l_1 + l_2$. As before, let $F : \{0,1\}^k \rightarrow \{0,1\}^k$ be a cryptographic hash function that is one-way, collision-resistant, and undetectable. Let $C(x, i)$ be a chaining function, where $C(x, 0) = x$, and $C(x, i) = F(C(x, i-1))$ for $i > 0$

2.2.1 WOTS Key generation

Use a PRNG, seeded with a seed S, to generate $l \cdot k$ bits of the secret key, and split them into l secret key components of k bits each, to form secret key $s = (s_0, \ldots, s_{l-1})$. Using chaining function C, generate public key components $(C(s_0, 2^b), \ldots, C(s_{l-1}, 2^b))$. On top of these l key components, build a hash tree of height $\lceil \log_2(l) \rceil$. Since l is not necessarily a power of two, an L-tree, as described in Section 2.1.2, is used for this. The root of this tree forms the WOTS public key.

In practice, neither the secret key nor the public key are ever stored permanently. Instead, they are regenerated from the seed S on demand.

2.2.2 WOTS Signing

Given message m of n bits, and secret key (S), split m into l_1 message components $m_0, \ldots, m_{l_1-1}$ of b bits each. Then, compute checksum $H(m = (m_0, \ldots, m_{l_1-1})) = \sum_{i=0}^{l_1-1}(2^b - 1 - m_i)$, and split $H(m)$ into l_2 components $h_0, \ldots, h_{l_2-1}$ of b bits each. Concatenate those elements to form input $I = (m_0, \ldots, m_{l_1-1}, h_0, \ldots, h_{l_2-1})$. This input has l components of b bits each, which will be addressed as $I_i, 0 \leq i < l$.

For each input component I_i, compute $c_i = C(s_i, I_i)$ to form signature $c = (c_0, \ldots, c_{l-1})$.

2.2.3 WOTS Verification

To verify the signature $c = (c_0, \ldots, c_{l-1})$ on message m, given public key p, compute checksum $H(m)$, and split both the message and the checksum into a total of l components of b bits each, forming input $I = (m_0, \ldots, m_{l_1-1}, h_0, \ldots, h_{l_2-1})$.

Then, compute the public key components $(C(c_0, I_0), \ldots, C(c_{l-1}, I_{l-1}))$ and build an L-tree on top of these public key components. The signature is valid if and only if the root of this L-tree equals the public key p.

2.3 The HORST signature scheme

This section will give a high-level overview of HORST, a few-times signature scheme based on the HORS signature scheme introduced in [44]. The security of a few-times signature scheme degrades the more messages are signed with a key pair.

A HORST secret key consists of a number of secrets. The HORST public key is a digest of the hashes of all those secrets. When signing a message, it is crucial that the signer first creates a digest of the message, rather than signing the plain message. Based on this digest, the signer then reveals a few of these secrets, and provides a way for the verifier to check that these secrets were indeed part of their public key.

A single key pair can be used for a few signatures, since only a fraction of all secret key components are revealed with each signature. However, as more messages are signed, the fraction of revealed secret key components grows. Eventually, there will be a non-negligible chance that an attacker can sign the digest of a new message using the revealed secret key elements of previous signatures. This is why the security of a key pair degrades as more messages are signed.

HORST can be configured to balance signature size against signing speed and re-usability. The parameter $t = 2^b, b \in \mathbb{N}$ determines the number of secret- and public-key components. The parameter $k \in \mathbb{N}$ determines in how many pieces the signed message digest will be divided. Both parameters influence the signature size, signing speed, and how often a key pair can be used to sign messages. Increasing either t or k increases the signature size, decreases the signing speed, but also increases the re-usability of each key pair.

This HORST key pair can then be used to sign a message digest of size $k \log_2 t$ bits. For practical choices for k and t, this usually results in a digest size of under $1kbit$. If necessary, this restriction can be overcome by hashing an arbitrary-size message using a cryptographic hash function with output length $k \log_2 t$ bits or more, and truncating this hash to $k \log_2 t$ bits.

2.3.1 HORST Key generation

Let $\mathcal{H} : \{0,1\}^n \rightarrow \{0,1\}^n$ be a cryptographic hash function that is (second) preimage resistant and collision resistant[1]. Given a secret seed S, use a PRNG to expand this seed to t secret-key components of n bits to obtain secret key $sk = (sk_0, sk_1, \ldots, sk_{t-1})$.

[1] With the changes that will be introduced in Section 2.5.2, this hash function will only need to be (second) preimage resistant

The public key is generated in two steps. First, each secret key component sk_i is hashed with $\mathcal{H}$, giving $pk_i = \mathcal{H}(sk_i)$ for $0 \leq i < t$. Now, a binary tree of height $\log_2 t$ is built from the t hashes $pk_0, pk_1, \ldots, pk_{t-1}$. The root of this tree forms the HORST public key pk.

2.3.2 HORST Signing

Given message M and some hash function $\mathcal{F}$ with an output length of $k \cdot b$, calculate $m = \mathcal{F}(M)$. Now, split m into k message components $m_1, m_2, \ldots, m_k$, each of size b. For each message component m_i, reveal the corresponding secret-key component sk_{m_i}. That is, reveal the secret-key component at the *index* that is equal to message component m_i. For example, if the message component has the value 25, then the 25th secret-key component is revealed.

Finally, for each revealed secret-key component sk_i, reveal the authentication path a_i that is needed to reconstruct the tree root pk from $\mathcal{H}(sk_i)$. The entire signature is then $((sk_{m_1}, a_{m_1}), \ldots, (sk_{m_k}, a_{m_k}))$.

2.3.3 HORST Verification

To verify a given message M, given signature $((sk_{m_1}, a_{m_1}), \ldots, (sk_{m_k}, a_{m_k}))$, compute $m = \mathcal{F}(M)$, and split m into k message components $m_1, m_2, \ldots, m_k$. For each message component m_i, compute $pk_i = \mathcal{H}(sk_{m_i})$ for all message components, and verify that public key pk can be restored from pk_i combined with authentication path a_i. The signature is valid if and only if this verification succeeds for all message components.

2.3.4 The vagueness of "few"

As seen above, each HORST signature reveals parts of the secret key. With each new signature, more parts of the secret key may be revealed. An attacker can assemble these secret-key components to forge signatures of a few unseen messages. For example, consider a HORST key pair where $k = 2$ and $t = 2$. If the attacker is in possession of a signature of the message digest `0110`, then the attacker can also forge a signature of message digest `1001`, since the secret key component $s_{\texttt{10}}$ and $s_{\texttt{01}}$ were part of the genuine signature.

The more messages are signed with a particular key pair, the more elements of the secret key are revealed, and it becomes increasingly likely that the k elements of a new message digest are a subset of all the secret key components that were already revealed in previous signatures.

As analyzed in [44, p. 6], HORS offers $k(\log_2 t - \log_2 k - \log_2 r)$ bits of security, where r is the number of signatures that have already been revealed. Using a larger value for t decreases signing speed and increases signature size, but improves the re-usability of a key pair. Increasing k reduces the re-usability of a key pair, but increases the message size.

2.4 simple-SPHINCS

SPHINCS combines WOTS and HORST into a stateless signature scheme. This section explains a simplified version of the original signature scheme presented in [9]. Key differences to the original version are outlined in Section 2.4.6.

2.4.1 The SPHINCS hypertree

SPHINCS uses a large tree structure called a hypertree. The hypertree is a hash tree that is divided into multiple layers of subtrees. Figure 2.2 shows an example of such a hypertree. While the size of the hypertree in SPHINCS is much larger than shown in Figure 2.2, the structure itself is otherwise identical. The hypertree in Figure 2.2 has a total height h of 4, and contains $d = 2$ layers of subtrees. Each subtree has a height of $h/d = 2$.

There exists a WOTS key pair for each leaf of every subtree, and the *public key* of this key pair forms the leaf node, shown in Figure 2.2 as ⓦ. With the exception of the lowest layer of the hypertree, each WOTS key pair is used to sign the root of the subtree below.

In addition to that, there exists a HORST key pair for each leaf of the hypertree. As shown in Figure 2.2, the public key of each HORST key pair, shown as Ⓗ, is signed with the WOTS key pair that forms the corresponding leaf of the hypertree.

Finally, the root of the hypertree forms the SPHINCS public key.

In practice, this hypertree is very large. SPHINCS-256, for example, uses a hypertree with a total height of 60. Due to the enormous size of the hypertree, it might be possible, but it is certainly not practical to ever calculate the entire tree. For the same reason, the WOTS and HORST key pairs are never permanently stored, but are instead generated deterministically from the SPHINCS secret key, and their position in the hypertree. This makes it possible to calculate any node of the hypertree with reasonable effort. For example, to calculate the value of node $n_{1,1}$ in subtree $s_{1,0}$, it is sufficient to generate the WOTS public keys that form the leaf nodes $n_{0,2}$ and $n_{0,3}$ in that subtree, and hash them. In a traditional hash tree that is not divided into subtrees, all the 8 leaves in subtree $s_{0,2}$ and $s_{0,3}$ would be required to produce this value. This demonstrates how a hypertree allows to compute any node in the tree from a small number of nodes.

When a message is signed with SPHINCS, the signer first picks a leaf node of the hypertree, and signs the message with the HORST key pair that corresponds to that leaf node. On its own, the resulting HORST signature is worthless, since there is no way for the verifier to check whether the used HORST key pair was actually part of the signer's hypertree. And since there are 2^{60} HORST key pairs in a SPHINCS hypertree, it is impractical to include all potential HORST public keys in the SPHINCS public key. Instead, the signer provides a series of WOTS signatures and authentication paths that allow the verifier to restore the root of the hypertree from the signature. This way the verifier can compare the restored root to the root that is stored in the signer's public key.

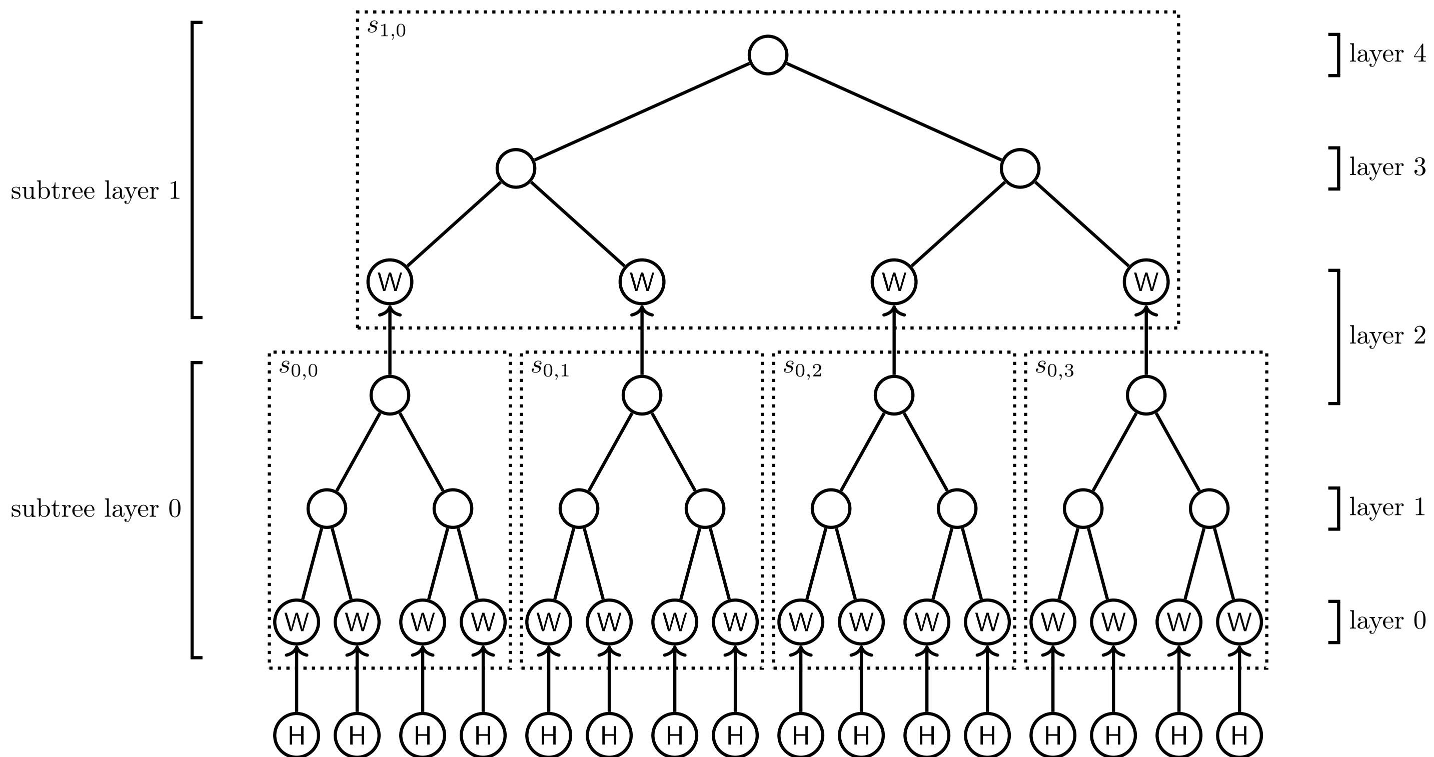

Figure 2.2: SPHINCS hypertree with a total height of 4 and subtrees of height 2. Dashed outlines show individual subtrees.

Figure 2.2 shows which elements of the hypertree the signer needs to reveal, so that the verifier can restore the root of the hypertree. After producing a HORST signature as described above, the signer signs the public key of the used HORST key pair, using the WOTS key pair that corresponds to the chosen leaf node of the hypertree. Then, the signer calculates all nodes along the authentication path of that leaf, which are needed to restore the root of the lowest subtree. The signer then signs that subtree root, using the WOTS key pair in the subtree above, and again calculates all nodes along the authentication path that are needed to restore the root of that subtree. The signer continues this process layer by layer, until they reach the root of the hypertree. With these WOTS signatures and authentication paths, the verifier can restore the root of the hypertree, and thereby verify whether the used HORST key pair was indeed part of the signer's hypertree.

Note that the SPHINCS signature only contains the WOTS and HORST *signatures*, but not the corresponding public keys. SPHINCS utilizes a property shared by WOTS and HORST: When verifying a WOTS or HORST signature, the verifier ends up with the public key of the used key pair if and only if the signature was valid, and otherwise ends up with a different value. Since the WOTS public keys form the leave nodes of the subtrees, an invalid WOTS signature will also change the nodes (and in particular the root) of the subtree.

Thus, if the HORST signature or any of the WOTS signatures are invalid, then the verifier will arrive at a different HORST or WOTS public key. Different public keys lead to different subtrees, and this change propagates through the hypertree, all the way to its root node. Any invalid signature will thus lead to a different root node. Since the expected root node is part of the SPHINCS public key, the verifier can then simply check whether the restored root node matches the expected root node. The SPHINCS signature is valid if and only if they match.

The various elements of the hypertree can be addressed in the following way:

- The hypertree has a total height of h.
- There are 2^h HORST key pairs at the bottom of the hypertree, $H_0, \ldots, H_{2^h-1}$.
- The hypertree consists of d layers of subtrees, with $d \mod h = 0$. Each subtree has height $h_s = h/d$. The 0 th layer is at the very bottom of the hypertree, containing 2^{h-h_s} subtrees. The subtrees of this layer have 2^h leaves in total.
- Each node within a subtree can be uniquely addressed by assigning indices 0 through $2^{h_s} - 1$ to the leaves of the subtree, 2^{h_s} through $2^{h_s} + 2^{h_s-1} - 1$ to the nodes of the second layer of the subtree, and so on, so that the root of the subtree has index $2^{h_s+1} - 2$.
- Each subtree in the hypertree can be uniquely addressed by the combination of its layer in the hypertree, and its index within that layer.
- By combining these addressing schemes, each node within the hypertree can be uniquely addressed by a tuple (i_l, i_s, i_n), where i_n is the index of

the node within its subtree, i_s is the index of the subtree within its layer, and i_l is the index of the layer that contains the subtree.

2.4.2 Key generation

A SPHINCS private key contains:

- A secret seed from which the the WOTS and HORST key pairs will be generated, and
- a fixed message hash salt S, which is used to induce pseudo-randomness when signing messages.

All these elements are chosen pseudo-randomly. The corresponding public key contains the root of the hypertree.

The root of the hypertree is calculated by generating the WOTS key pairs at the leaves of the topmost subtree, and building the hash tree on top of the public keys of those key pairs.

2.4.3 Signing

To sign a message m with a secret key sk, the signer first chooses an index i of a hypertree leaf deterministically from the message and the message hash salt.

The signer then hashes the message and the message hash salt. This hash is signed using the HORST key pair H_i at index i, yielding HORST signature σ_H. The public key of this key pair is then signed using the WOTS key pair at leaf node $(0, i/h_s, i \mod 2^{h_s})$, yielding s_0. The signer then computes the root r_0 of subtree i/h_s, as well as the authentication path a_0, containing the h_s nodes that are necessary to restore r_0 from the leaf node $i \mod 2^{h_s}$.

As Figure 2.2 shows, each subtree layer has as many subtrees as the layer above has leaf nodes, so that each subtree root can be mapped to a subtree leaf of the layer above (with the exception of the subtree at the top of the hypertree). The signer finds the WOTS key pair that corresponds to the root of subtree i/h_s on layer 0, and uses this key pair to sign r_0, yielding WOTS signature σ_1. As before, the signer then computes the root r_1 of this subtree, as well as the authentication path a_1 that is needed to restore r_1 from the public key of the used WOTS key pair.

The signer continues this for the remaining layers, signing the root of the last subtree with the corresponding WOTS key pair, and computing the root and authentication path of the new subtree, until they reach the root of the hypertree.

Finally, they produce signature $(S, i, \sigma_H, \sigma_0, \dots \sigma_{d-1}, A)$, where

- S is the message hash salt,
- i is the index of the HORST key pair that was used to sign the message,

- σ_H is the HORST signature of the hash of the message and the message hash salt,
- σ_0 is the WOTS signature that signs the public key of the used HORST key pair,
- $\sigma_1 \ldots \sigma_{d-1}$ are the $d-1$ WOTS signatures that sign the roots of the subtrees, and
- A is the set of h nodes that form the authentication paths a_0 through a_{d-1}.

As explained above, the signature does not need to contain the WOTS and HORST public keys of the used key pairs, as the verifier will compute the public keys as part of the signature verification.

2.4.4 Verifying

Given message m, public key (M, r), and signature $(S, i, \sigma_H, \sigma_0, \ldots \sigma_{d-1}, A)$, the verifier hashes m and S. The resulting hash, together with HORST signature σ_H restores the public key pk_H.

From the HORST public key pk_H and the WOTS signature σ_0, the verifier can restore WOTS public key pk_0. This, combined with the first h_s nodes of the authentication path A restores the root r_0 of the subtree on layer 0. From here, the verifier works their way up through the hypertree. Each time the root of a subtree is restored, the verifier uses the next WOTS signature to restore the WOTS public key of the key pair that was used to sign that root node. With this new public key and the next h_s authentication nodes, the verifier can then restore the next subtree root, up until they restored the root of the hypertree.

Finally, the verifier checks whether the restored root of the hypertree matches the root that is stored in the signer's public key. The signature is valid if and only if these nodes match.

2.4.5 simple-SPHINCS-256

This section will define simple-SPHINCS-256, a specific instantiation of the parameters in the general SPHINCS construction described above. SPHINCS-256 was defined in [9, p. 21f] as a trade-off between signature size and speed. Simple-SPHINCS-256 uses the same parameters where applicable for this simplified version. In simple-SPHINCS-256, the hypertree has a total height of $h = 60$, with $d = 12$ subtree layers, implying a subtree height of $h_S = 5$. The security parameter $n = 256$ is chosen to provide 2^{128} security against attackers with access to quantum computers, and also sets the output length of the hashes used in WOTS and HORST to 256 bits. Furthermore, the output bit length of the message hash that is signed using HORST is set to $m = 512$. Each HORST key pair has $t = 2^{16}$ secret key elements, and $k = 32$ HORST secret key elements are revealed with each signature. Finally, SPHINCS-256 sets the chain length in WOTS signatures to $w = 16$, implying $l = 67$.

This configuration leads to a signature size of $41,000$ bytes, a public key size of 1056 bytes, and a secret key size of 1088 bytes.

The number of messages that can be safely signed with a single SPHINCS key pair is finite, since the number of leaf nodes in the SPHINCS hypertree is finite, and since HORST is a few-times signature scheme. However, with the SPHINCS-256 configuration, 2^{50} messages can be signed comfortably without compromising the 2^{128} security [9, p. 20].

2.4.6 Differences to SPHINCS

Simple-SPHINCS, the signature scheme described above, is a simplified version of SPHINCS, the signature scheme presented in [9]. Note that the version presented here cannot be used as a simple drop-in replacement of the more complex version, since some details are omitted in this simplified version which are crucial to the security of the signature scheme.

A key difference is that SPHINCS heavily uses bit masks throughout the hypertree, as well as in the HORST tree construction. By using these bit masks, the underlying hash function only needs to be second-preimage resistant, rather than collision resistant. This idea was first proposed in [17], based on work in [5].

The details around the bit masks are omitted in simple-SPHINCS, since they will be replaced by a different method in Section 2.5.2.

2.5 Multi-target attacks in SPHINCS

This section explains multi-target attacks, how they apply to SPHINCS, how they can be mitigated, and which advantage this mitigation has on the key size of SPHINCS. The attack vector itself was first described in [33], along with a mitigation technique. The same technique can also be adapted to mitigate multi-target attacks in SPHINCS.

Multi-target attacks come into play when the same hash function is used many times in a cryptographic construction. A simple classical brute-force attack on the (second-)preimage resistance of a hash function with output length n has a success chance of roughly $1/2^n$, since that is the chance that the produced output matches the favorable outcome. However, if the same hash function is used d times, and if the attacker benefits from finding (second-)preimages to any of those outcomes, then the success chance of a brute-force attack becomes $d/2^n$ [33, p. 8]. For a generic quantum attack based on Grover's algorithm, the success probability increases by $\sqrt{d}$ [33, p. 8]. Depending on the value of d, this can undermine not only formal security arguments, but also affects the security of the scheme in practice.

2.5.1 Multi-target attack surfaces in SPHINCS

In SPHINCS, hashes are used in numerous scenarios, but not all of them are affected by multi-target attacks in the same way. It follows an overview of the various ways in which hashes are utilized throughout SPHINCS, along with an explanation how the hash function used in each scenario is affected by multi-target attacks.

To get a better intuition on the actual impact on SPHINCS in practice, the configuration of SPHINCS-256 is used to produce specific numbers.

WOTS. There are $\sum_{d=1}^{12} 2^{5d} \approx 2^{60}$ WOTS key pairs associated with each SPHINCS-256 key pair, the public keys of which are part of the hypertree. This is a prime example for multi-target attacks. For all WOTS key pairs, the same hash function is used to build a binary tree, the root of which forms the public key of that key pair. An attacker can generate an arbitrary WOTS key pair, and when the public key of this WOTS key pair matches any of the 2^{60} public keys in the hypertree, then the attacker can easily generate genuine signatures.

In practice, the attacker will need to consider that generating WOTS key pairs is a non-negligible effort by itself; for SPHINCS-256, generating a WOTS key pair involves computing roughly 2^{10} hashes. The generated key pairs, however, do not need to be specially crafted when attacking a specific SPHINCS key pair – they can be re-used in future attacks.

The public key is not the only attack surface in each WOTS key pair. Each WOTS signature uses 67 hash chains of length 16. Each hash chain, and each individual hash is calculated using the same hash function. If an attacker were to find a preimage for any one of the $67 \cdot 16$ chain elements in any WOTS signature, then the attacker is able to produce a signature for a slightly altered message than the one that was signed with that signature.

Recall, however, that the WOTS key pairs are used to sign tree roots and HORST public keys. When an attacker finds a preimage in one of the WOTS chain, they might be able to forge certain signatures, but that leaves the problem of finding either, a HORST public key, or a suitable tree that can be signed with the help of the found preimage.

HORST. There are 2^{60} HORST key pairs associated with each SPHINCS-256 key pair, the public keys of which are part of the hypertree. If an attacker generates a HORST key pair with a public key that matches the public key of any HORST key pair in the hypertree, then the attacker is able to forge signatures of arbitrary messages.

Also for HORST there is a second attack surface. Each HORST key pair has 2^{16} secret-key components. The hashes of these secret-key components form the corresponding public key components. If an attacker was able to find a preimage for any of those public-key components, they could use the found secret-key component to forge a signature.

For this attack in particular it is important to understand when components of the HORST key pair are revealed. With each signature, 32 secret-key components are revealed as part of the signature itself, along with an additional 32 public key components which are revealed as part of the authentication paths. If an attacker was able to find the preimage of any of those public key components, it would allow them to forge a signature of a set of previously unsignable messages. However, since the HORST signatures never sign plain-text messages directly but a randomized digest, the attacker would still have to find a way to produce a message with a digest that is now signable, thanks to the found preimage.

Hypertree. $2^{60}-1$ hashes of the form $\{0,1\}^{2n} \rightarrow \{0,1\}^n$ are used to construct the hypertree. It might not be immediately clear why these hashes are affected by multi-target attacks. After all, if an attacker managed to find a (second) preimage of any of the nodes in the hypertree, then they would merely learn the concatenation of the two child nodes.

Recall that the leaf nodes of subtrees contain WOTS public keys. An attacker could generate two WOTS key pairs, treat their public keys as leaf nodes, and produce their parent node. If the value of this parent node matches the value of any node on layer 1 of a subtree, then the attacker can integrate their own WOTS key pairs into this subtree, and in turn forge a signature. Each subtree has 2^4 nodes on layer 1, so that there are 16 favorable outcomes for the attacker in each subtree.

But the attacker is not limited to generating two WOTS key pairs to launch a multi-target attack. They can generate any $2^i, 1 \leq i \leq 5$ key pairs, build a binary tree on top of them, and check if the root node of this tree equals any of the 2^{5-i} nodes on layer $5-i$ of that subtree. Furthermore, the attacker can even re-shuffle the order of the WOTS key pairs, or re-use some of the WOTS key pairs, leading to different binary trees, which have a new chance of matching any of the favorable outcomes. Generating x WOTS key pairs allows the attacker to arrange them in $x^{(2^5)} = x^{32}$ different ways at the leaves of a subtree of height 5. Building the corresponding subtree is comparably cheap, and yields $2^5 - 1 = 31$ nodes that might match one of the favorable outcomes.

The attacks above can be executed on all subtrees within the hypertree. There are $\sum_{l=0}^{11} 2^{5l}$, or roughly 2^{55} subtrees in the SPHINCS hypertree. For each layer i within those subtrees any forged binary tree has thus $2^{55} \cdot 2^{5-i}$ potential matches.

WOTS and HORST public keys Hash trees are also used in the construction of both, the WOTS and the HORST public keys. Similar to the previous attack on binary hash trees in the hypertree, an attacker can construct an arbitrary part of those hash trees, and compare the root node of that partial tree to a number of favorable outcomes. Depending on which part of the tree is attacked, there are up to 2^{60+15} favorable outcomes for hashes in the HORST public keys, and up to $2^{60} \cdot (2^5 + 1)$ favorable outcomes for hashes in the WOTS

public keys[2].

However, both of these attacks have a small impact compared to the impact of a successful attack on the binary tree that forms the hypertree. If successful, an attacker could insert a WOTS or HORST key pair into the hypertree that is partially different from one of the genuine key pairs. However, this still leaves the problem of generating or finding a suitable key pair in the first place.

It should be noted that only a small fraction of those hash images are revealed with each signature, so that the attack surface grows the more signatures are created with any given key pair. In addition to that, these attacks can also be carried out across different key pairs.

2.5.2 Mitigating multi-target attacks

One option to mitigate these attacks is to simply use hash functions with a larger output size. However, the key size and more importantly the signature size in SPHINCS and its underlying hash-based signature schemes scales linearly with the output size of the used hash function. This was also pointed out in [33] for XMSS and its variants, but applies to SPHINCS in the same way.

Another solution is to use a different hash function in each distinct scenario in which the hash function family is used. In practice, this can be done by keying the hash function with a specific value that uniquely identifies the scenario in which this hash function is used. This method is described in [33, p.17], and there it is proven that this mitigation technique makes it just as hard to find a (second) preimage of a hash value as if the hash function was used only once in the entire cryptographic construction.

In SPHINCS, this can be implemented with two changes. To make sure that multi-target attacks cannot be launched across different key pairs, a random public seed is added to every key pair.

In addition to that, each hash function call is uniquely identified with an addressing scheme. In [33], this addressing scheme is constructed recursively by first identifying the *structure* that is being addressed, followed by an index that identifies the specific hash function call within that structure. This can be adapted for SPHINCS. Addresses can be grouped into four categories.

- A `HORST` address addresses all hash function calls that are involved in key generation, signing, and verification with HORST. The hash function calls can be indexed as follows. The first 2^b indices address the hash calls that generate public key components from secret key components. The following $2^b - 1$ indices are used for the hash calls that construct the binary tree of height b on top of the public key components.
- A `WOTS` address covers the hash function calls in the WOTS chaining function. Each call can be identified by first addressing in which of the

[2] The L-tree used to construct WOTS public keys has height 7, but nearly half of the nodes are omitted, which reduces the attack surface.

chains the call is made, and which chain link is being calculated.

- A `WOTS_L` address covers the hash calls that construct the L-tree on top of the WOTS public key components. For this, the nodes are simply indexed one by one, starting with 0 at the left-most node on layer 0.

- Finally, a `SPHINCS` address uniquely identifies a node within the SPHINCS hypertree through the combination of the subtree layer, the index of the subtree within that layer, and finally the index of the node within that subtree.

  This address type is used to address the hash function calls that construct the hypertree. In addition to that, all other address types are always coupled with a SPHINCS address.

In SPHINCS, each hash function call was masked with bit masks that were determined during key generation. With the addressing scheme described above we no longer need pre-generated bit masks. Each hash function call is then keyed with the public seed, and masked with a bit mask derived ad-hoc from the address that uniquely identifies the hash function call.

In SPHINCS, the bit masks were determined during key generation, and contributed to the size of the public key. Replacing the bit masks reduces the public key size significantly. For instance, SPHINCS-256 has a public key of 1056 bytes. By replacing the bit masks with a single public seed of 32 bytes, the public key shrinks down to a total of 64 bytes.

Chapter 3

Reducing HORST signature size

This chapter introduces a new method to reduce the size of HORST signatures. Section 3.1 first explains an optimization already present in SPHINCS that greatly reduces the size of the HORST signatures used in SPHINCS. Section 3.2 then presents a new method that can reduce the size of HORST signatures even further, proves that this method is always beneficial over HORST signatures with no optimizations, and discusses its efficiency in worst-case and best-case scenarios. Section 3.3 discusses implementation details, and Section 3.4 concludes with statistical measurements of the efficiency of the new approach in practice.

3.1 HORST signature size in SPHINCS

In SPHINCS-256, each HORST signature consists of 32 leaves of a binary tree of height 16. To reconstruct the root of this tree from any one of these leaves, an authentication path of 16 nodes is required. Simply storing all of these authentication paths separately leads to a total of $32 \cdot 16 = 512$ nodes.

SPHINCS stores the authentication paths for the 32 leaves in a more efficient way. Since all leaves are part of the same tree, the authentication paths for the leaves must necessarily share many nodes – especially on higher layers of the tree. To not waste space with these duplicates, SPHINCS stores layer 10 of the binary tree in its entirety inside the signature. This means that nodes on layers 10 through 15 can be removed from the authentication paths. Across the authentication paths for all the 32 different leaf nodes, this reduces the number of nodes by $32 \cdot 6 = 192$ nodes in total. Layer 10 itself contains 64 nodes. Thus, overall this optimization reduces the signature size from 512 hashes to $32 \cdot 10 + 64 = 384$ hashes.

3.2 HORST Authentication sets

This section presents a novel approach to verify multiple leaves of a binary tree, in which the authentication paths of all leaves are combined into an *authentication set*, instead of storing them separately. This approach reduces the HORST signature to 352 hashes in the worst-case scenario, and to 11 hashes in the (admittedly unlikely) best-case scenario. Authentication sets are not only small, they can also be calculated efficiently, and with a small memory footprint.

Given a set of leaf nodes L, the authentication set A_L is the minimal set of nodes that is required to recompute the root of a binary hash tree from the leaf nodes in L. Section 3.2.1 shows that authentication sets are always smaller than the combination of separate authentication paths. Section 3.2.2 then produces a general formula to calculate the size of an authentication set, and Section 3.2.3 analyses best-case and worst-case scenarios. Section 3.3 covers ways to implement authentication sets, and Section 3.4 discusses practical aspects of using authentication sets in SPHINCS, including their efficiency on average.

When referring to nodes in binary trees, the same notation will be used that was introduced at the beginning of Chapter 2.

3.2.1 Combining authentication paths

In SPHINCS, the HORST signature contains the nodes of the authentication path of each leaf separately. An authentication set benefits from the fact that separate authentication paths might contain duplicate nodes, and that the authentication path of one leaf might allow one to generate a node on the authentication path of a different leaf.

In general, if more than one leaf of the binary tree is given, an authentication set will always be strictly smaller than the combination of separate authentication paths. If a single leaf is given, the authentication set is the same as the authentication path. We can prove that for any set of given leaves L, $|L| \geq 1$ of a tree of height h, there is a corresponding authentication set A_L such that $|A_L| \leq (h-1)(|L|-1)+h$.

This can be proven by induction on the size of L. In the base case, only a single leaf is given, so that $|L| = 1$. If only a single leaf is given, then the authentication set equals the normal authentication path of that leaf node, which contains h nodes, so that $|A_L| = h$. So, for $|L| = 1$ we have

$$
\begin{aligned}
|A_L| &\leq (h-1)(|L|-1)+h \\
|A_L| &\leq (h-1)(1-1)+h \\
|A_L| &\leq h,
\end{aligned}
$$

which holds since $|A_L| = h$.

For the inductive case, recall that in a binary tree of height h with $h > 0$, layer $h-1$ always contains exactly two nodes, $n_{h-1,0}$ and $n_{h-1,1}$. Consequentially, all leaves are either a child of $n_{h-1,0}$ or $n_{h-1,1}$.

Assume that there is a set L with a corresponding authentication set A_L for which it holds that $|A_L| \leq (h-1)(|L|-1)+h$. Given a leaf l, $l \notin L$, we want to find the maximum size of authentication set $A_{L\cup\{l\}}$ that authenticates all nodes in $L \cup \{l\}$.

Assume that l is a child of $n_{h-1,0}$. Then there are two possible cases:

1. There are no other leaf nodes in L that are children of $n_{h-1,0}$.

   If we were to construct the full authentication path of all leaves independently, then $n_{h-1,1}$ would be part of the authentication path of l, while $n_{h-1,0}$ would be part of the authentication paths of all nodes in L.

   We can reconstruct $n_{h-1,0}$ from the first $h-1$ nodes of the authentication path of l. We can also reconstruct $n_{h-1,1}$ from the nodes in A_L, since $n_{h-1,1}$ lies on the paths of all nodes in L. We therefore do not need either node of layer $h-1$ to authenticate $L \cup \{l\}$.

   Thus, adding l makes one node in A_L redundant, and only $h-1$ new nodes are necessary to authenticate l itself. Therefore, $|A_L|-1+(h-1) = |A_{L\cup\{l\}}| \leq |A_L| + (h-1)$.

2. There are other leaf nodes in L that are children of $n_{h-1,0}$. In this case, node $n_{h-1,1}$ is on the authentication path of other nodes in L. Since A_L contains all nodes required to authenticate all leaf nodes in L, we know that either $n_{h-1,1}$ is part of A_L, or $n_{h-1,1}$ can be reconstructed from other nodes of A_L and L.

   Either way, node $n_{h-1,1}$ does not need to be included in $A_{L\cup\{l\}}$, so that at most $h-1$ nodes are required to authenticate l, and $|A_{L\cup\{l\}}| \leq |A_L| + (h-1)$.

   In practice, the size of the authentication set can then be reduced further by applying these rules recursively on the tree underneath node $n_{h-1,0}$ until $h = 0$.

If l was a child of $n_{h-1,1}$ instead, then a similar proof can be made by swapping $n_{h-1,0}$ and $n_{h-1,1}$.

Thus, adding any new leaf l adds at most $h-1$ nodes to the authentication set A_L, so that for the new set $L \cup \{l\}$ of size $|L|+1$ there is a new authentication set $A_{L\cup\{l\}}$ of maximum size $|A_L| + (h-1)$ so that

$$\begin{aligned}
|A_{L\cup\{l\}}| &\leq |A_L| + (h-1) \\
&\leq (h-1)(|L|-1) + h + (h-1) \\
&\leq (h-1)((|L|+1)-1) + h. \\
&\leq (h-1)(|L\cup\{l\}|-1) + h.
\end{aligned}$$

Note that this proof does not capture the full benefit of using authentication sets; it is simply meant to prove that authentication sets are never larger than the combined size of authentication path, and that they are even smaller than the combined size of authentication paths if the number of leaves is larger than 1.

Leaf	Authentication path
$n_{0,1}$	$(n_{0,0}, n_{1,1}, n_{2,1})$
$n_{0,2}$	$(n_{0,3}, n_{1,0}, n_{2,1})$
$n_{0,3}$	$(n_{0,2}, n_{1,0}, n_{2,1})$
$n_{0,4}$	$(n_{0,5}, n_{1,3}, n_{2,0})$

Table 3.1: Examples for authentication paths of leaves of the binary tree in Figure 3.1.

3.2.2 Authentication-set size

This section analyzes which characteristics of the given leaves influence the size of the minimal authentication set, and presents a function that determines the size of the minimal authentication set from a set of given leaves.

As described above, the authentication set is smaller than the sum of separate authentication paths since duplicate and redundant nodes are removed. Thus, the more nodes the various authentication paths share, the smaller the authentication set is. Unfortunately it is not straight-forward to determine the exact number of duplicate and redundant nodes.

Table 3.1 shows the authentication paths of different leaves of the binary tree in Figure 3.1. Even though all four leaves are right next to one another, the number of shared nodes in their authentication paths is different for each pair of leaves. For example, leaves $n_{0,2}$ and $n_{0,3}$ share two nodes of their authentication paths, while the authentication paths of leaves $n_{0,3}$ and $n_{0,4}$ do not have a single node in common. Thus, the number of shared nodes (and therefore also the size of the authentication set) depends not only on the number of given leaves, but also on their relative position to one another.

How many nodes the authentication paths of two leaves share depends on the layer on which the paths between the root and leaves diverge. Let $n_{l,i}$ be the node below which the paths of the two leaves diverge. Below layer l, the authentication paths of the two leaves are different. But on and above layer l the authentication paths are identical. The examples in Table 3.1 show this, too. The paths of $n_{0,1}$ and $n_{0,2}$ diverge on layer 2, and the nodes on layer 0 and 1 on their authentication paths are indeed different, while the node on layer 2 is the same for both leaves, which confirms the pattern described above. The paths of $n_{0,2}$ and $n_{0,3}$ diverge on layer 1, and indeed, from layer 1 upwards the authentication paths of these leaves are identical.

To compute the layer on which the paths of two leaves diverge, or the *distance to divergence*, we can look at the binary representation of their indices.

The paths of leaves $n_{0,2}$ and $n_{0,3}$ diverge on layer 1. Looking at the binary representation of their indices (10_2 and 11_2, respectively), we can note that they only differ at the least significant bit.

The path of leaves $n_{0,3}$ and $n_{0,4}$ diverge on layer 3. Accordingly, the binary representation of their indices (011_2 and 100_2) differ up to the third bit, counting from the least significant bit.

In general, the layer on which the paths of two leaf nodes $n_{0,i}$ and $n_{0,j}$ diverge

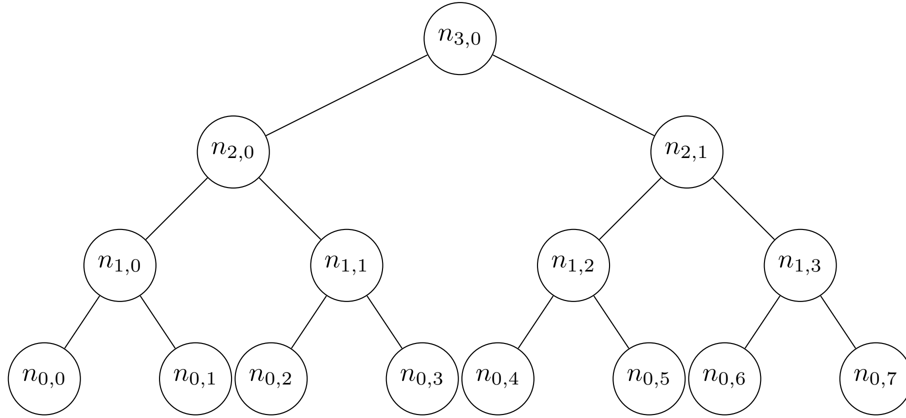

Figure 3.1: Binary tree of height 3.

is equal to the position of the left-most bit at which the binary representations of i and j differ. The following function $d(i, j)$ returns that position.

$$d(i,j) = \begin{cases} 0 & \text{if } i = j \\ 1 + d(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor) & \text{otherwise} \end{cases}$$

We can also explain this function with the geometry of binary trees. Recall that the parent node of $n_{l,i}$ is $n_{l+1,\lfloor i/2 \rfloor}$. The distance to divergence between two nodes $n_{0,i}$ and $n_{0,j}$ is 0 if $i = j$. Otherwise, it is one bigger than the distance to divergence of their parent nodes.

There are $2 \cdot d(i, j)$ non-shared nodes on the authentication paths of two leaves $n_{0,i}$, $n_{0,j}$, but two of these nodes do not need to be added to the authentication set, namely the ones on layer $d(i, j) - 1$. If the paths of the two leaves diverge in a node $n_{d(i,j),v}$ on layer $d(i, j)$, then one of its child node is on the path from $n_{0,i}$ to the root node, and on the authentication path of $n_{0,j}$, and vice versa for the other child node. Because of that, both of those nodes can be constructed from the first $d(i, j) - 1$ nodes of the authentication paths of the two leaves, and therefore neither of those nodes need to be included in the authentication set.

For example, leaves $n_{0,1}$ and $n_{0,2}$ have in total $2 \cdot d(1, 2) = 4$ non-shared nodes in their authentication paths, as shown in Table 3.1. But node $n_{1,1}$ can be constructed from $n_{0,0}$ and the given leaf $n_{0,1}$, while node $n_{1,0}$ can be constructed from $n_{0,3}$ and the given leaf $n_{0,2}$, which means that neither of the leaves on layer $d(1, 2) - 1 = 1$ need to be included in the authentication set.

With this metric we can compute the size of the authentication set for two given leaves $n_{0,i}$, $n_{0,j}$ in a tree of height h:

$$\begin{aligned} s((n_{0,i}, n_{0,j}), h) &= (d(i,j) - 1) \cdot 2 + (h - d(i,j)) \\ &= h + d(i,j) - 2. \end{aligned}$$

This is simply the number of non-shared nodes in the authentication paths per leaf, minus the two redundant nodes, plus the number of shared nodes in the upper layers of the tree.

If more than two leaves are given, the overlap in the authentication paths will differ for each pair of leaves. However, this does not mean that the overlap needs to be calculated for each pair of given leaves. Instead, by iterating through the given leaves from left to right, the leaves can be enumerated in such a way that for each leaf l_i it holds that no l_j, $j > i$ is closer to l_i than l_{i+1}. Here, *closer* refers to the distance as determined by d.

Given three unique leaves l_i, l_j, l_k, sorted from left to right, the total size of the authentication set can then be computed like so:

$$s((n_{0,i}, n_{0,j}, n_{0,k}), h) = d(i, j) + d(j, k) + h - 2 \cdot (3 - 1).$$

This can be disassembled into three parts:

1. For each given leaf and its closest given neighbor leaf to the right, add the $d(j, k)$ nodes that are required to restore a subtree up to the layer where the paths of these two leaves diverge.
2. For the last leaf add the size of its entire authentication path.
3. For each leaf but the last leaf, subtract the two nodes that have become redundant as described above.

This equation can be generalized to compute the size of the authentication set for an arbitrary number n of given leaves:

$$s(L = (n_{0,l_1}, n_{0,l_2}, \dots, n_{0,l_n}), h) = h - 2 \cdot (n - 1) + \sum_{i=1}^{n-1} d(l_i, l_{i+1}) \tag{3.1}$$

Here, L is the tuple containing the given leaves, sorted by their index, and with duplicate leaves removed, and h is the height of the binary tree.

3.2.3 Best-case and worst-case scenarios

This section analyses these best-case and worst-case scenarios for authentication sets, and produces two formulas that can calculate the size of the authentication set in the best and worst case, given the tree height and the number of given leaf nodes. Average cases are not discussed theoretically, but Section 3.4 presents experimental results for the average size of HORST authentication sets in SPHINCS.

With Equation 3.1 alone, it is difficult to reason about the efficiency of authentication sets in general, since the exact size of the authentication set depends on the individual indices of the given leaves. It is, however, possible to find formulas for the authentication-set size in the best and worst case: For a fixed number of unique given leaves, the size of the authentication set only depends on the geodesic distance between neighbor leaves. In graph theory, the geodesic distance is the number of edges in the shortest path from one node to the other. Note that the geodesic distance of two leaf nodes $n_{0,i}$, $n_{0,j}$ is $2d(n_{0,i}, n_{0,j})$. Minimizing or maximizing the geodesic distance between all leaves will also minimize or maximize the size of the minimal authentication set, respectively.

The sum of the geodesic distances between n unique leaves is minimal if all leaf nodes are children or descendants of $\lceil n/2^k \rceil$ node(s) on layer k, for all $k \in [1, \lceil \log_2(n) \rceil]$. In this case, the given leaf nodes are distributed across the tree as little as possible, leading to maximal overlap between their authentication paths, which minimizes the size of the authentication set.

For example, consider the binary tree in Figure 3.1. The following leaf nodes are given: $n_{0,0}, n_{0,2}, n_{0,3}, n_{0,6}, n_{0,7}$. On layer $\lceil \log_2(5) \rceil = 3$, all these leaf nodes are descendants of a single node $n_{3,0}$. On layer 2, all the leaf nodes are descendants of the $\lceil 5/2^2 \rceil = 2$ nodes $n_{2,0}$ and $n_{2,1}$. Finally, on layer 1, all the leaf nodes are children of the $\lceil 5/2^1 \rceil = 3$ nodes $n_{1,0}$, $n_{1,1}$, and $n_{1,3}$. The authentication set for these leaf nodes consists of only two nodes: $n_{0,1}$ and $n_{1,2}$.

To give another example, consider that the following leaf nodes are given: $n_{0,0}, n_{0,2}, n_{0,3}, n_{0,5}, n_{0,6}$. As before, these leaf nodes are descendants of one node on layer 3, and two nodes on layer 2. On layer 1, however, the leaf nodes are children of four different nodes $(n_{1,0}, n_{1,1}, n_{1,2}, n_{1,3})$, while $\lceil 5/2^1 \rceil = 3$. Thus, this set of leaf nodes does not fall into the best-case scenario. Indeed, the authentication set for these leaf nodes has three elements $(n_{0,1}, n_{0,4}, n_{0,7})$.

When a set of leaf nodes falls into the best-case scenario, it is possible to compute $\sum_{i=1}^{n-1} d(l_i, l_{i+1})$ without knowing the exact indices of the given leaf nodes. Starting at layer 0, we know that amongst the n leaf nodes are $\lfloor n/2 \rfloor$ pairs of sibling nodes, which share the same parent node on layer 1. Naturally, there cannot be more than $\lfloor n/2 \rfloor$ pairs of sibling nodes in a binary tree. But there cannot be less than $\lfloor n/2 \rfloor$ pairs of sibling nodes, either: We know that, in the best-case scenario, all leaf nodes are distributed across $\lceil n/2^1 \rceil$ nodes on layer 1. If n is even, then the $n = 2k, k \in \mathbb{N}$ leaf nodes are distributed across $\lceil 2k/2 \rceil = k$ nodes on layer 1, which necessarily leads to $n/2 = k$ pairs of sibling nodes. If n is odd, then the $n = 2k+1, k \in \mathbb{N}$ leaf nodes are distributed across $\lceil (2k+1)/2 \rceil = k+1$ nodes on layer 1. But those $k+1$ nodes have precisely $2k+2$ child nodes. No matter how the $2k+1$ revealed leaf nodes are distributed amongst those $k+1$ parent nodes, at least $k = \lfloor (2k+1)/2 \rfloor$ pairs of siblings will be amongst them.

When there are $\lfloor n/2 \rfloor$ pairs of sibling nodes amongst the given leaf nodes, then for each pair l_i, l_{i+1}, we know that $d(l_i, l_{i+1}) = 1$, since their paths to the root of the tree diverge on layer 1.

Moving on to layer 1, we know that there are $c_1 = \lceil n/2^1 \rceil$ nodes on layer 1 which are parents to the revealed leaf nodes, and, as before, there are $\lfloor c_1/2 \rfloor$ pairs of sibling nodes amongst them. In each pair we can identify a left node $n_{1,l}$ and a right node $n_{1,r}$. Amongst the children of $n_{1,l}$ there is one leaf node, $n_{0,i}$, which is the furthest to the right between all revealed leaf nodes that are descendants of $n_{1,l}$. Similarly, $n_{1,r}$ is the parent node of some leaf node $n_{0,j}$, which is the furthest to the left between all revealed leaf nodes that are descendants of $n_{1,r}$. Then, $n_{0,j}$ is a direct successor of $n_{0,i}$, when ordering all revealed leaf nodes from left to right. Furthermore, we know that the paths of $n_{0,i}$ and $n_{0,j}$ diverge at layer 2, or more precisely, in the parent node of nodes $n_{1,l}$ and $n_{1,r}$. Therefore, $d(n_{0,i}, n_{0,j}) = 2$.

This pattern is generalized for all layers in this formula:

$$\sum_{i=1}^{n-1} d(l_i, l_{i+1}) = \sum_{l=0}^{h-1} \left\lfloor \frac{\lceil n/2^l \rceil}{2} \right\rfloor \cdot (l+1).$$

Table 3.2 lists $\sum_{i=1}^{n-1} d(l_i, l_{i+1})$ for various numbers of leaves, computed with this method. Looking up this sequence of integers in the On-Line Encyclopedia of Integer Sequences[1] yields a connection between the exact sequence (A005187[2]) and the Hamming weight (A000120[3]), or the number of ones in the binary representation of a number. The link between those two sequences was pointed out by OEIS user Paul Barry. This leads to the following compact formula for the best case scenario:

$$\sum_{i=1}^{n-1} d(l_i, l_{i+1}) = 2(n-1) - H(n-1),$$

where $H(n)$ is the Hamming weight of n. Table 3.2 shows $\sum_{i=1}^{n-1} d(l_i, l_{i+1})$ for various numbers of leaves, along with $H(n-1)$ and $2(n-1) - H(n-1)$.

# of given leaves	1	2	3	4	5	6	7	8	9	10	11	12
$\sum_{i=1}^{n-1}(d(l_i, l_{i+1}))$	0	1	3	4	7	8	10	11	15	16	18	19
$2(n-1)$	0	2	4	6	8	10	12	14	16	18	20	22
$H(n-1)$	0	1	1	2	1	2	2	3	1	2	2	3
$2(n-1) - H(n-1)$	0	1	3	4	7	8	10	11	15	16	18	19

Table 3.2: Examples to illustrate the connection between the geodesic distances and the Hamming weight.

Then, the best-case overall size of the authentication set is

$$\begin{aligned} s_B(n,h) &= h - 2(n-1) + 2(n-1) - H(n-1) \\ &= h - H(n-1). \end{aligned}$$

In the worst-case scenario, on the other hand, all nodes are spaced out as far as possible across the tree structure, which maximizes the geodesic distance between them. Let L be the set of given leaf nodes, and let $n = |L|$ be the number of given leaf nodes. Let $h_l = \lceil \log_2(n) \rceil$ be the minimum height of a binary tree big enough to hold n leaf nodes.

In the worst-case scenario, the paths from all n given leaf nodes to the root pass through n *different* nodes on layer $l = h - h_l$, while forming only $n - 2^{h_l - 1}$ pairs of siblings.

First, observe that, if the paths of all leaf nodes pass through different nodes on layer l, then there is no overlap between their authentication paths on the lowest l layers. Therefore, the authentication set will need to contain $n \cdot l$ nodes that are necessary to restore the nodes on layer l.

[1] `https://oeis.org`
[2] `https://oeis.org/A005187`
[3] `https://oeis.org/A000123`

Next, we will turn our attention to the pairs of siblings on layer l. If two nodes $n_{l,i}, n_{l,j}$ are siblings, then the paths to the root of the tree meet in their common parent node. Otherwise, their paths to the root of the tree will meet further up in the tree, causing less overlap in the authentication paths of the two nodes. Let N_L be the set of nodes on layer l that are on any of the paths from the leaf nodes in L to the root of the tree. The fewer pairs of sibling nodes there are in N_L, the less overlap is there in the authentication paths of the given leaf nodes, and the bigger is the size of the authentication set.

There have to be at least $n - 2^{h_l - 1}$ pairs of siblings amongst N_L on layer l. Since layer $l + 1$ has $2^{h_l - 1}$ nodes, there are also $2^{h_l - 1}$ pairs of sibling nodes on layer l in total. From our definition of h_l, we know that $n > 2^{h_l - 1}$. It is possible to arrange the first $2^{h_l - 1}$ elements of N_L such that none of them are siblings of each other. However, the remaining $n - 2^{h_l - 1}$ elements of N_L will necessarily be nodes that are siblings of nodes which are also in N_L. Those form $n - 2^{h_l - 1}$ pairs of sibling nodes, neither of which need to be included in the authentication set.

That leaves the remaining $2^{h_l} - n$ nodes in N_L which do not have a sibling node that is also in N_L. The siblings of those nodes do have to be included in the authentication set.

At this point, all nodes on layer $l + 1$ can then be restored. Therefore, none of the nodes on layers $l + 1$ and above need to be included in the authentication set.

This leaves us with the following formula for the worst-case scenario:

$$s_W(n, h) = n \cdot l + 2^{h_l} - n, h_l = \lceil \log_2(n) \rceil, l = h - h_l.$$

For SPHINCS-256, in the best-case scenario all 32 given leaves are children of the same node on layer 5. In that case, this node and its subtree can be completely restored just from the given leaves, since a binary tree of height 5 has exactly 32 leaves. Thus, only the authentication path on layer 5 through 15 is required to restore the root of the tree. Since all leaves have the same authentication path from layer 5 upwards, the entire authentication set only consists of the 11 nodes on layer 5 through 15. Verifying this with the equation above yields indeed

$$s_B((l_1, l_2, \ldots, l_32), 16) = 16 - H(32 - 1) = 16 - 5 = 11.$$

The worst-case scenario in SPHINCS would be that the 32 given leaves are spread across the 32 nodes on layer 11. Up to layer 11, none of the authentication paths of the given leaves overlap. This accounts for $11 \cdot 32 = 352$ nodes in the authentication set. No additional nodes are required from layer 11 upwards however, since all nodes on layer 11 can be restored from those 352 nodes. Using $h_l = \lceil \log_2(32) \rceil = 5$ and $l = 16 - 5 = 11$, we can verify this with the equation above:

$$s_W((l_1, l_2, \ldots, l_32), 16) = 32 \cdot 11 + 2^5 - 32 = 352.$$

3.3 Implementing HORST with authentication sets

This section will explain how HORST signature creation and verification with authentication sets can be implemented efficiently. The initial focus will be on the computation of the authentication set itself. After that we explore how the authentication sets can be used in the HORST signing and verification algorithms.

Section 3.3.1 will walk through a naïve Python implementation to compute the authentication set. Section 3.3.2 will then introduce a more efficient implementation, both in terms of execution time and memory requirement.

3.3.1 Naïve implementation

The function `path(index, height)` returns the path from the leaf at index `index` to the root of a binary hash tree of height `height`. Note that the returned path of a given leaf does include the leaf itself. The function `auth_path(index, height)` returns the authentication path for the leaf at index `index` in a binary hash tree of height `height`.

Nodes are uniquely identified with a tuple that contains their layer and their index (both in line with the definitions introduced at the beginning of Chapter 2). With this indexing method, the index of the left child node is always even, and the index of a right child node is always odd. Since the nodes are indexed continuously from left to right, starting with 0, the sibling of a node at index i is always at index $i \oplus 1$, where $\oplus$ denotes binary XOR.

```
1  def path(index, height):
2    layer = 0
3    path = []
4    while layer < height:
5      path.append((layer, index))
6      layer += 1
7      index >>= 1
8    return path
9
10 def auth_path(index, height):
11   layer = 0
12   authpath = []
13   while layer < height:
14     authpath.append((layer, index ^ 1))
15     layer += 1
16     index >>= 1
17   return authpath
```

The function `auth_set_addresses` computes the addresses of the nodes in the authentication set of a set of `leaves` in a binary hash tree of height `height`. First, for each given leaf the authentication path of this leaf is computed, as well as the path from that leaf to the root of the binary hash tree. These authentication paths and paths are accumulated in two lists. Next, the Python built-in function `set` transforms the accumulated authentication paths and accumulated

paths into unordered collections of unique nodes. The difference of these two sets form the minimal authentication set for the given leaves.

```
18 def auth_set_addresses(leaves, height):
19   accum_path = []
20   accum_auth = []
21   for leaf in leaves:
22     accum_path += path(leaf, height)
23     accum_auth += auth_path(leaf, height)
24   return set(accum_auth) - set(accum_path)
```

It is easy to show that all remaining nodes are strictly necessary to restore the binary tree's root from the set of given leaves. Suppose that the produced set was not minimal. In that case, there has to be at least one superfluous node that could be removed for one of the following reasons:

1. The node is not on the authentication path of any of the given leaves. This is not possible since the authentication set only ever contains nodes that are on the authentication path of at least one of the given leaves.
2. The node is not unique. This is not possible since duplicates have been removed by the `set` function.
3. The value of the node can be computed from other given leaves. This is not possible either, since hash trees can inherently only be computed from the bottom up. Thus, a value of any node can only be computed if it is on the path of a given leaf to the tree's root. Since all such nodes have been removed, none of the remaining nodes can be computed from other leaves.

Thus, removing any one of the remaining nodes from the authentication set is guaranteed to remove a node from the authentication path of at least one of the given leaves which cannot be computed from any of the other given leaves, making it impossible to restore the root of the binary tree.

Therefore this function does indeed produce the minimal authentication set for the given leaves.

3.3.2 Efficient implementation

The naïve implementation is easy to read, and should help to get an idea of how to compute authentication sets, but it is not particularly efficient. It computes the path and authentication path of all leaves and holds them in memory temporarily, and it compares all nodes of the authentication paths of all given leaves to all the nodes of the paths of all given leaves.

Below is a more efficient implementation in Python. This implementation, and a C implementation of the same algorithm are available at `https://github.com/25A0/authentication-sets`.

```
1 def auth_set_addresses(leaves, height):
2   stack = []
3   auth_set = []
4   sorted_unique_leaves = unique(sorted(leaves))
5
6   for i in range(0, len(sorted_unique_leaves)):
```

```
7      if i < len(sorted_unique_leaves) - 1:
8        d = half_dist(sorted_unique_leaves[i],
9                sorted_unique_leaves[i+1])
10     else:
11       d = height + 1
12
13     leaf_index = sorted_unique_leaves[i]
14     for h in range(0, d - 1):
15       if len(stack) > 0 and stack[-1] == h:
16         stack.pop()
17       else:
18         # Add that element to the authentication set
19         auth_set.append((h,leaf_index ^ 1))
20       leaf_index >>= 1
21     # Add that height to the stack
22     stack.append(d-1)
23   return auth_set
```

The Python built-in function `sorted(list)` returns a new list containing all elements of `list` sorted in ascending order, while the function `unique(list)` returns a list of the unique elements of the given, sorted `list`. As long as `list` is sorted, `unique` can be implemented very efficiently.

The remainder of the function computes the addresses of the nodes in the authentication set, using the patterns explained in the previous sections.

The stack never exceeds size $h-1$, and the size of the authentication set can be computed in advance using the equations given in previous sections. It might seem faster to simply compute the size of the authentication set for the worst-case scenario, since here the individual geodesic distances between the given leaves do not need to be computed. However, these distances can simply be cached in an array of size n, and then re-used when the actual authentication set is computed.

3.3.3 Using authentication sets in HORST

The efficient implementation above helps to explain the initial claim that it is easy to implement HORST signatures this way in a memory-efficient manner. When creating a traditional HORST signature, there are three options how the authentication paths can be constructed:

- The authentication path is constructed separately for each leaf. This is very memory efficient since the tree can be computed by caching at most h nodes at a time. However, this requires the entire tree to be computed k times.
- The entire tree is cached. This is reasonable for tree sizes like the ones used in SPHINCS, but can be difficult in situations with tight memory restrictions. The obvious benefit is that the tree only needs to be computed once.
- The authentication paths are all constructed during a single iteration through the tree. This combines the best of the other two options: It avoids iterating through the tree more than once, but does not require the verifier to cache the entire tree. This option was used in [45, p. 15f] to

construct a HORST signature with very tight memory restrictions without compromising speed.

When memory restrictions are not too tight, the fastest method to collect the hashes of the authentication set is to build the entire tree in memory, and simply pick out the nodes of the authentication set.

For tight memory restrictions this approach is not an option. Ideally the signer would use the tree-hash algorithm to build the tree with a minimal memory footprint, and copy the nodes of the authentication set as they are computed. Unfortunately this is not trivial when the authentication set was computed with the efficient algorithm given above. The order of nodes in the authentication set is different from the order in which they are encountered in the tree-hash algorithm. For example, the authentication set might start with nodes $n_{0,2}$ and $n_{1,0}$, if leaf $n_{0,3}$ is given. However, in the tree-hash algorithm the verifier will calculate $n_{1,0}$ before they come across $n_{0,2}$. In general the verifier would need to check for each encountered node whether it is part of the authentication set, which is very inefficient. One solution to this is to sort the nodes of the authentication set, so that they appear in the order in which they are encountered in the tree-hash algorithm. With this change the verifier only ever needs to compare the encountered node to the next non-encountered node in the authentication set.

Verifying a signature that includes an authentication set is not trivial, unfortunately. When a separate authentication path is given for each given leaf, then the tree root can be restored simply by computing the hash of the given leaf and the first item of the authentication path, taking that hash and hashing it together with the second item of the authentication path, and continuing this way until the root is reached.

With authentication sets the process gets more complicated. Since there are no more redundant nodes in the authentication set, the verifier can only reconstruct the root node by combining all given leaves with all nodes in the authentication set. The order in which the nodes need to be combined is not as straightforward as it is with traditional HORST signatures. In each step, the verifier has the choice between using one of the items from the authentication path to produce the next hash, or pushing the current hash to a stack and continuing with the next given leaf. Luckily though this decision is always easy when the efficient algorithm above is used to produce the authentication set. The following function produces the root of the hash tree this way.

```
1  def verify(hashed_leaves, auth_set, auth_set_addresses, height, hashfun):
2    stack = []
3    i_auth = 0
4    i_leaf = 0
5    hash_stage = [None, None]
6    while i_auth < len(auth_set_addresses) or i_leaf < len(hashed_leaves):
7      # Pick the next given leaf
8      height = 0
9      index, current_hash = hashed_leaves[i_leaf]
10     i_leaf += 1
11     while True:
12       hash_stage[index % 2] = current_hash
13       needed_node = (height, index ^ 1)
14       # Consume as many nodes from the stack and
```

```
15        # the auth set as possible
16        if len(stack) > 0 and needed_node == stack[-1][0]:
17          _, hash_stage[(index % 2) ^ 1] = stack.pop()
18        elif i_auth < len(auth_set_addresses) and \
19           needed_node == auth_set_addresses[i_auth]:
20          hash_stage[(index % 2) ^ 1] = auth_set[i_auth]
21          i_auth += 1
22        else: break
23        current_hash = hashfun(hash_stage[0], hash_stage[1])
24        height += 1
25        index >>= 1
26      stack.append(((height, index), current_hash))
27
28    assert(len(stack) == 1 and (height, 0) == stack[0][0])
29    # Return the root's hash
30    return stack[0][1]
```

Here, `auth_set_addresses` contains the addresses of the nodes in the authentication set (as produced by the algorithm in Section 3.3.2), while `auth_set` contains the hashes of the authentication set in the order dictated by `auth_set_addresses`; i.e. the hash of the node with address `auth_set_addresses[i]` can be found at `auth_set[i]`.

During verification, the verifier calculates one hash for each node that is provided by the signer. This means that using authentications sets also slightly reduces the number of hashes that need to be calculated by the verifier. Similar to the size of the authentication set, in SPHINCS-256 the verifier will need to calculate 352 hashes in the worst case, as opposed to 384 hashes.

3.4 Practical aspects of HORST authentication sets

HORST authentication sets differ from traditional HORST signatures in that their size varies even when tree size and the number of given leaves are fixed. When it comes to using authentication sets within SPHINCS, there are two options how this variable signature size can be handled:

- The whole SPHINCS signature could be changed to be of variable length, in which case the overall signature size would benefit from each byte that is saved with the HORST signature set.
- The SPHINCS signature could reserve enough room for the HORST signature in the worst case. With this approach the size of the SPHINCS signature would still benefit from the fact that even in the worst case the authentication set requires 32 nodes less than the HORST signature in the SPHINCS implementation.

This section will inspect whether the reduced signature size would justify a variable signature size for SPHINCS as a whole.

Figure 3.2 shows the results of a simulation that measures how the size of the authentication set is distributed. The parameters in this simulation are identical to those used in SPHINCS-256: 32 leaves of a binary tree of height

16 are chosen at random. The plot shows the result of 2^{16} samples. Table 3.3 contains a statistical summary of the findings.

mean	324.265
standard deviation	7.168
minimum	286.0
25th percentile	320.0
50th percentile	325.0
75th percentile	329.0
maximum	346.0

Table 3.3: Statistical summary of the authentication set size across 2^{16} samples.

Recall that the size of the authentication set in the best-case and worst-case scenario was 11 and 352, respectively. Even though the authentication set could contain as little as 11 nodes in the best case, these results show that in reality most of the signatures will contain 320 to 340 nodes. Since the average authentication-set size is much closer to the worst case than to the best case, it would appear that the API complications that come along with variable size signatures are not justified.

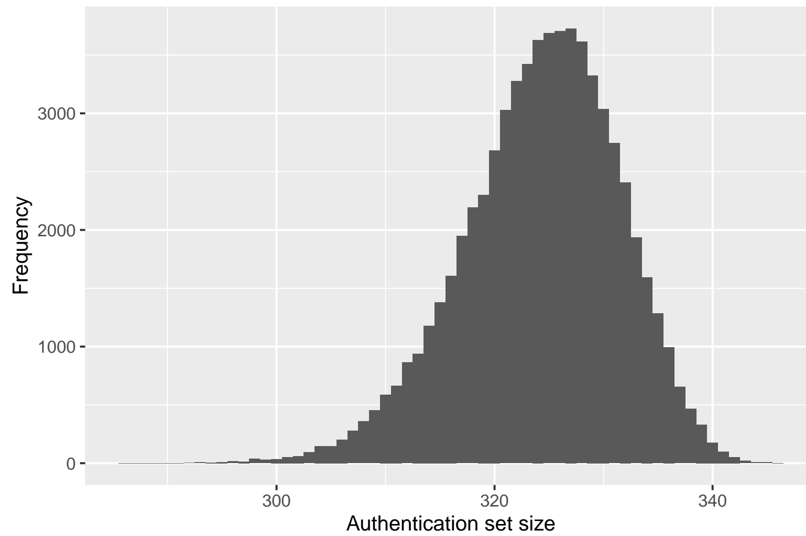

Figure 3.2: Distribution of authentication set size

Note that these results change when the number of given leaves changes; if more leaves are revealed, then the benefit of using authentication sets increases. In the context of the HORST signature scheme, revealing more leaf nodes would impact the security of the scheme. However, authentication sets can be applied whenever leaf nodes of a binary hash tree need to be authenticated, not just in the context of HORST signatures.

Figure 3.3 shows how the size of the authentication set compares to the cumulative size of separate authentication paths, for different numbers of revealed leaf

nodes. Here, a value of 0.3 means that the authentication set is 30% smaller than the cumulative size of the separate authentication paths. The more leaves are given, the more does the signature size benefit from the use of authentication sets. Thus, while variable size signatures might not be justified for SPHINCS, they might very well be justified in a different scheme.

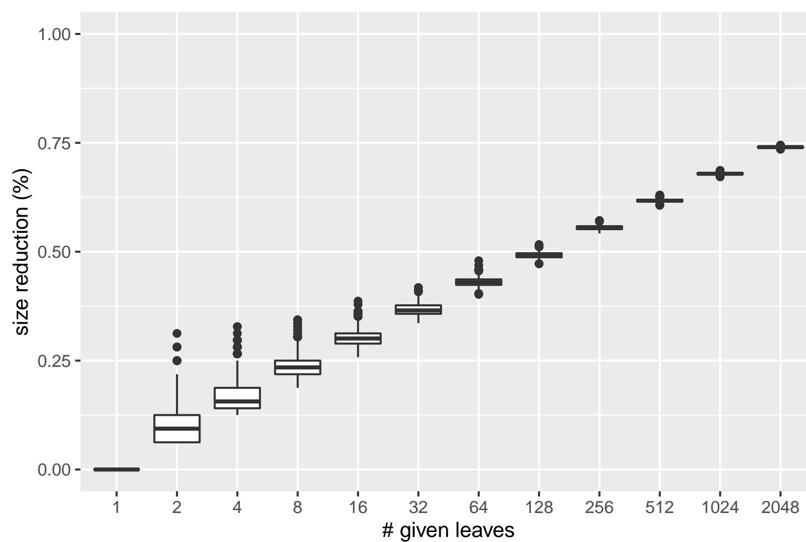

Figure 3.3: Benefit of authentication sets compared to cumulative size of authentication paths, in a binary hash tree of height 16.

Chapter 4

Two approaches to short-term state in SPHINCS

This chapter introduces two ways in which a short-term state can be used to speed up SPHINCS signature creation. Section 4.1 explains why adding some sort of state can speed up signature creation in non-standard SPHINCS. Section 4.2 discusses general risks of stateful signature schemes, and the remainder of this chapter introduces and compares two approaches to utilize short-term state.

4.1 The benefit of state in SPHINCS

The reason why signature creation can be sped up by introducing a short-term state is rooted in two properties of SPHINCS:

- A large part of the SPHINCS signature does not depend on the content of the signed message. This allows the signer to compute parts of the signature without knowing the message that will be signed.
- Parts of the signature can be cached and re-used to reduce the amount of work per signature.

This section explains these properties in more detail.

Signing a message in SPHINCS involves multiple signature schemes. The verifier needs to produce a HORST signature of the message itself, multiple WOTS signatures, as well as the authentication path that allows the verifier to reproduce the root of the hypertree. But only the computation of the HORST signature involves the message itself; to calculate the remainder of the signature it is sufficient to know the index of the leaf that is being used to sign the message. Specifically, out of the $41,000$ bytes in a SPHINCS signature, the calculation of $13,352$ bytes involve the message itself, and the remaining $27,648$ bytes only depend on the index of the used leaf. In the following, the terms *message-dependent* and *leaf-dependent* will be used to refer to those two chunks of the

signature.

Thus, just from knowing which leaf node is used to sign a message, the signer can compute a significant part of the signature. In classic SPHINCS however, the leaf node is chosen deterministically, based on the message. If the leaf node was chosen independently of the message, then that would open up new ways to distribute the workload of signature creation; in particular, the leaf-dependent part of a signature could be computed before the message is known. This idea is also explored in On-line/Off-line digital signatures by Even et al. [22].

Secondly, the leaf-dependent parts of two signatures will be partially identical, depending on the geodesic distance between the two leaves. The closer two leaves are to each other in terms of their geodesic distance (the number of edges between two nodes in a graph), the smaller is the difference between the leaf-dependent parts of the signatures. If two messages were signed with the HORST key pair of the same leaf, then the leaf-dependent parts of those signatures would be completely identical.

Now, combining these properties, one could construct a stateful variant of the traditional SPHINCS signature scheme where the leaves of the hypertree are used sequentially from left to right to sign messages, and where the leaf-dependent parts of the signature are cached and re-used. This will minimize the geodesic distance between the leaves that are used for two successive signatures. In turn, this maximizes how much of the previous signature can be re-used. But it does require the signer to keep track of the latest used hypertree leaf index, and to cache parts of the signature. Since traditional SPHINCS is a stateless signature scheme, there is no way to keep track of this information without abandoning the stateless characteristic of traditional SPHINCS. At the same time, stateful signature schemes are difficult to manage securely, as the next section will explain.

A *short-term state* forms a hybrid solution that retains the stateless nature of the signature scheme as a whole, but caches and re-uses the leaf-dependent parts of the SPHINCS signature for a limited time. This allows the signer to quickly sign messages, since only a fraction of the SPHINCS signature needs to be recomputed for each new message. At the same time, this approach avoids the usual downsides of a fully stateful signature schemes.

4.2 Challenges of stateful signature schemes

Stateful signature schemes come with many challenges. This section will discuss difficulties related to backups and scalability.

With one-time signature schemes, where secret keys can only be used to sign a single message, the verifier needs to keep track of which keys have already been used. If this record resides in permanent storage (e.g., a hard disk), then the signer must always use the latest version of this record to prevent accidental key re-use. If this record is ever lost, then the key pair cannot be used any longer, since the signer has no guarantee that any of the one-time signatures have not been used in the past. Similarly, if this record is part of a backup, then restoring

this backup could lead to accidental key re-use.

Scaling a stateful signature scheme is also challenging. If the same key pair should be used to sign messages on different threads or CPUs, then the state needs to be synchronized between the threads or CPUs to prevent accidental re-use of already used key material.

4.3 Short-term state in SPHINCS

A short-term state can add some of the benefits of a stateful signature scheme without introducing the risks that come with it.

short-term states can be initialized independently, which makes SPHINCS easily scalable. The same SPHINCS key pair can be used on different threads, CPUs, VM instances or physical machines by simply initializing a fresh short-term state for each instance, without any need for state synchronization.

As long as the short-term state is never written to disk, there is no risk that restoring a backup could lead to re-use of already used key material. Of course, as soon as a short-term state is captured in a backup, the same problems arise that affect stateful signature schemes. For example, if a short-term state is included in a VM-snapshot, then restoring that same snapshot multiple times can lead to re-use of used key material.

The following sections present two ways to add a short-term state to SPHINCS. For each approach, the following actions will be defined:

State initialization. The signer initializes a short-term state based on their SPHINCS key pair and some chosen or pseudo-randomly produced information. This short-term state can then be used to sign a fixed number of messages.

State incrementation. The signer updates the short-term state after signing a message.

Signing. The signer signs a message, using the short-term state and their own SPHINCS key pair.

Verifying. The verifier verifies a signature on a given message.

Querying. The signer queries the number of messages that can be signed with a given short-term state.

Key generation is not different from key generation in traditional SPHINCS.

The following notation will be used:

- h The total height of the hypertree
- d The number of subtree layers in the hypertree
- $h_s = h/d$ The height of the subtrees in the hypertree

4.4 Approach 1: Iterate through lowest subtree

The first approach comes with minimal changes to the signature scheme. Rather than picking a random leaf of the hypertree for each signed message, the signer could pick one of the subtrees at the bottom of the hypertree randomly, and use the leaf nodes of this subtree one by one to sign messages.

This allows the signer to re-use a large portion of the signature, since most of the message-independent parts of the SPHINCS signature are identical for each leaf in the chosen subtree.

This approach will be referred to as *sequential* batch signing, since the signer iterates over the leaves of a subtree in sequence.

State initialization. Given a SPHINCS secret key, choose a subtree on subtree layer 0 by (randomly) choosing an index $i_s \in [0, 2^{h-h_s} - 1]$. Then choose the index of a HORST key pair within that subtree, by (randomly) choosing an index $i_h \in [0, 2^{h_s} - 1]$. Finally, compute the root r of subtree i_s.

Next, compute the WOTS signature s_1 by signing r with the corresponding WOTS key pair on subtree layer 1. Compute the authentication path a_1 for the used key pair, as well as the root of the enclosing subtree. Repeat this procedure for each subtree layer, up to the root of the hypertree.

This produces $d-1$ WOTS signatures, as well as $h-h_s$ nodes along the authentication path. This produces the short-term state $(i_s, i_h, n, (s_1, a_1), \ldots, (s_d, a_d))$, where n is the number of messages that have been signed with the short-term state, and initialized to 0. Furthermore, s_k is the WOTS signature of layer k of the hypertree, and a_k is the authentication path through layer k of the hypertree.

Signing. Given message m, short-term state $(i_s, i_h, n, (s_1, a_1), \ldots, (s_d, a_d))$, and secret key sk, check if $n \geq 2^{h_s}$, in which case the signing capacity of the short-term state has been exhausted. In that case, abort with an error.

Otherwise, compute $i_l = i_h + n \mod 2^{h_s}$. This is the index of the HORST key pair within subtree i_s that will be used for the signature. Then compute $i = i_s \cdot 2^{h_s} + i_l$, the index of the hypertree leaf node that will be used for this signature. Here, $i_s \cdot 2^{h_s}$ produces the index of the left-most leaf node of subtree i_s.

Then produce a message hash m_h and HORST signature sig_H, using the HORST key pair at leaf node i. Then produce a WOTS signature of the HORST public key s_0, and the authentication path a_0 that allows the verifier to restore the root of subtree containing i. The signature is then $(m_h, i, \mathrm{sig}_H, (s_0, a_0), \ldots, (s_d, a_d))$.

Note that this signature is structured the same way a traditional SPHINCS signature is structured.

State incrementation. Whenever a message was signed using the short-term state $(i_s, i_h, n, (s_1, a_1), \dots, (s_d, a_d))$, increment the number of signed messages, n.

Verifying. For this approach, verification works exactly the same way it works with traditional SPHINCS signatures.

Querying. Given short-term state $(i_s, i_h, n, (s_1, a_1), \dots, (s_d, a_d))$, return $2^{h_s} - n$.

4.4.1 Impact on security assumptions

With this approach, 2^{h_s} messages can be signed with a single short-term state – one for each HORST key pair in the subtree.

Note that, while all leaf nodes of the chosen subtree are used throughout the lifetime of a short-term state, they are not necessarily used from left to right; the left-most leaf node of the chosen subtree is not necessarily used for the first message signed with the short-term state. When the short-term state is generated, a random leaf node of the hypertree is chosen. This leaf node can be anywhere within its subtree. After that, the leaf nodes of the subtree are used from left to right, wrapping around to the left-most node once the right-most node was used.

This detail is crucial in cases where not each short-term state is used to its full potential. If each short-term state started with the left-most leaf of a subtree, and if the short-term states were not fully utilized, then the leaf nodes of the hypertree would not be evenly used: A leaf node further left in its subtree would be more likely to be used than a leaf node further right in its subtree. The above approach removes this bias.

Next, we will analyze how this approach affects the collision probability amongst the HORST key pairs. More specifically, we are interested in the probability that a HORST key pair is used γ times or more over the lifetime of a SPHINCS key pair. Here, γ depends on the chosen HORST parameters. HORST is a few-times signature scheme, and its security degrades the more messages are signed with the same key pair.

For example, SPHINCS-256 chose HORST parameters that provide 256 bit pre-quantum security or more, as long as no HORST key pair is used to sign $\gamma = 9$ messages or more.

The size of the hypertree was then chosen accordingly so that it becomes sufficiently unlikely that any HORST key pair is used 9 times or more, while as many as $q = 2^{50}$ messages can be signed over the lifetime of a SPHINCS key pair. But how exactly can we determine how likely it is that a HORST key pair is used a certain number of times?

The work in [48] on multi-collisions is essential for this analysis: A *γ-collision* is the event that, when sampling q times from a set of n elements, the same

element is chosen γ or more times. In [48], the exact probability is given as a recursive function:

$$Pr[C(n,q,\gamma)] = \frac{1}{n^{\gamma-1}} \cdot \sum_{i=\gamma}^{q} \binom{i-1}{\gamma-1} \cdot \left(1 - \frac{1}{n}\right)^{i-\gamma} \cdot (1 - Pr[C(n, i-\gamma, \gamma)]) \tag{4.1}$$

Using this function is rather impractical for our context, but the following upper bound is given, too:

$$Pr[C(n,q,\gamma)] \leq \frac{1}{n^{\gamma-1}} \binom{q}{\gamma} \tag{4.2}$$

In traditional SPHINCS, these formulas can be used to compute the probability that any HORST key pair is used γ times or more. Here, $n = 2^h$ is the number of HORST key pairs in the hypertree. With the introduction of short-term states, however, these formulas do not apply any more. Now, rather than choosing a HORST key pair pseudo-randomly for each message, a subtree on the lowest hypertree layer is chosen pseudo-randomly, and used to sign a number of messages.

Let $k, 1 \leq k \leq 2^{h_s}$ be the number of messages that are signed with each short-term state on average. Then q/k short-term states need to be initialized to sign q messages. With each short-term state initialization, one of the 2^{h-h_s} subtrees is chosen pseudo-randomly.

Following the approach in [48, p. 31], we can model the probability of a γ-collision with the following experiment: We throw a total of q balls at 2^h buckets. The buckets are arranged in groups of size 2^{h_s}, forming 2^{h-h_s} groups. Instead of throwing the balls one by one, they are thrown in volleys of k balls, with $1 \leq k \leq 2^{h_s}$. Each of the $v = q/k$ volleys is aimed at one group, chosen at random. Within each group, the 2^{h_s} buckets are arranged in a circle. When a volley hits a group, the first ball of the volley hits one of the 2^{h_s} buckets, chosen at random. The remaining $k-1$ balls of the volley hit the following $k-1$ successors of that bucket, moving clockwise through the circle of buckets. Note that in a single volley, no bucket can be hit more than once.

Let us first discuss the probability that a bucket B is hit in a volley. Let $H_B(k,h,h_s)$ denote the event that bucket B is hit in a volley of k balls, with 2^h buckets in total, and bucket groups of size 2^{h_s}. Let $H_{group(B)}(h,h_s)$ denote the event that the group of bucket B is hit in a volley, with 2^h buckets in total, and groups of size 2^{h_s}. We know that $Pr[H_{group(B)}(h,h_s)] = \frac{1}{2^{h-h_s}}$ since there are 2^{h-h_s} groups in total, and a volley hits exactly one of them.

$Pr[H_B(k,h,h_s)|H_{group(B)}(h,h_s)]$ is the probability that B is hit, given that the group containing B is hit. We know that $Pr[H_B(k,h,h_s)|H_{group(B)}(h,h_s)] = \frac{k}{2^{h_s}}$, because B is hit if the first ball of the volley either hits B itself, or any of the $k-1$ predecessors of B, moving counter-clockwise. Thus,

$$\begin{aligned} Pr[H_B(k,h,h_s)] &= Pr[H_B(k,h,h_s)|H_{group(B)}(h,h_s)] \cdot Pr[H_{group(B)}(h,h_s)] \\ &= \frac{1}{2^{h-h_s}} \cdot \frac{k}{2^{h_s}} = \frac{k}{2^h}. \end{aligned}$$

Let $C_{STS}(h, h_s, q, k, \gamma)$ denote the event that after throwing q balls, in volleys of k balls each, at 2^h buckets grouped into groups of size 2^{h_s}, there is at least one bucket containing at least γ balls.

Let $C_{STS-i}(h, h_s, v, k, \gamma, i)$ denote the event that a γ-collision occurs in the ith volley – that is, when the ith volley is thrown, one of the buckets will contain γ balls. Note that a γ-collision cannot occur before γ volleys have been thrown, since a bucket can be hit at most once per volley. Then

$$Pr[C_{STS}(h, h_s, q, k, \gamma)] = \sum_{i=\gamma}^{q/k} Pr[C_{STS-i}(h, h_s, q/k, k, \gamma, i)].$$

We can find $Pr[C_{STS-i}(h, h_s, v, k, \gamma, i)]$ as follows:

- One bucket B can be selected from 2^h buckets in 2^h ways.
- $\gamma - 1$ volleys that hit B can be selected from the previous $i-1$ volleys in $\binom{i-1}{\gamma-1}$ ways.
- The probability that the γ selected volleys hit B is $\left(\frac{1}{2^{h-h_s}} \cdot \frac{k}{2^{h_s}}\right)^{\gamma}$.
- The probability that none of the other $i - \gamma$ volleys hit B is $\left(1 - \frac{1}{2^{h-h_s}} \cdot \frac{k}{2^{h_s}}\right)^{i-\gamma} \cdot (1 - Pr[C_{STS}(h, h_s, i-\gamma, k, \gamma)])$.

Then

$$\begin{aligned}
Pr[C_{STS-i}(h, h_s, q, k, \gamma, i)] &= 2^h \cdot \binom{i-1}{\gamma-1} \cdot \left(\frac{1}{2^{h-h_s}} \cdot \frac{k}{2^{h_s}}\right)^{\gamma} \cdot \\
&\quad \left(1 - \frac{1}{2^{h-h_s}} \cdot \frac{k}{2^{h_s}}\right)^{i-\gamma} \cdot \\
&\quad (1 - Pr[C_{STS}(h, h_s, i-\gamma, k, \gamma)]) \\
&= 2^h \cdot \binom{i-1}{\gamma-1} \cdot \left(\frac{k}{2^h}\right)^{\gamma} \cdot \left(1 - \frac{k}{2^h}\right)^{i-\gamma} \cdot \\
&\quad (1 - Pr[C_{STS}(h, h_s, i-\gamma, k, \gamma)]) \\
&= \frac{k^{\gamma}}{(2^h)^{\gamma-1}} \cdot \binom{i-1}{\gamma-1} \cdot \left(1 - \frac{k}{2^h}\right)^{i-\gamma} \cdot \\
&\quad (1 - Pr[C_{STS}(h, h_s, i-\gamma, k, \gamma)])\,.
\end{aligned}$$

Thus we have

$$\begin{aligned}
Pr[C_{STS}(h, h_s, q, k, \gamma)] &= \sum_{i=\gamma}^{q/k} Pr[C_{STS-i}(h, h_s, q/k, k, \gamma, i)] \qquad (4.3) \\
&= \frac{k^{\gamma}}{(2^h)^{\gamma-1}} \cdot \sum_{i=\gamma}^{q/k} \binom{i-1}{\gamma-1} \cdot \left(1 - \frac{k}{2^h}\right)^{i-\gamma} \cdot \\
&\quad (1 - Pr[C_{STS}(h, h_s, i-\gamma, k, \gamma)])\,.
\end{aligned}$$

Using [48, Lemma 1(1)], we can derive an upper bound for this formula:

$$Pr[C_{STS}(h, h_s, q, k, \gamma)] \leq \frac{k^\gamma}{(2^h)^{\gamma-1}} \cdot \binom{q/k}{\gamma} \tag{4.4}$$

Note that for $k = 1$, Formula 4.3 is equal to Formula 4.1:

$$\begin{aligned} Pr[C_{STS}(h, h_s, q, 1, \gamma)] = &\frac{1^\gamma}{(2^h)^{\gamma-1}} \cdot \sum_{i=\gamma}^{q} \binom{i-1}{\gamma-1} \cdot \left(1 - \frac{1}{2^h}\right)^{i-\gamma} \cdot \\ &(1 - Pr[C_{STS}(h, h_s, i-\gamma, 1, \gamma)]) \\ = &Pr[C(2^h, q, \gamma)]. \end{aligned}$$

The formulas are not equal for other values of k, though. For example with $k \geq 2$, we can compare what can happen when signing two messages with classic SPHINCS on the one hand, or with sequential batch signing on the other hand. In classic SPHINCS, while highly unlikely, it is possible that the same HORST key pair is picked twice for signing two consecutive messages. When using sequential batch signing, however, we only pick a subtree once during short-term state initialization. After that, two different HORST key pairs will be used to sign the two messages. Thus, the probability of a 2-collision after signing just two messages is zero when using sequential batch signing, but it is non-zero in classic SPHINCS.

To better understand the difference between the two exact formulas, we can compare the two upper bound formulas 4.4 and 4.2:

$$\begin{aligned} \frac{k^\gamma}{(2^h)^{\gamma-1}} \cdot \binom{q/k}{\gamma} &\stackrel{?}{=} \frac{1}{(2^h)^{\gamma-1}} \cdot \binom{q}{\gamma} \\ \frac{k^\gamma}{(2^h)^{\gamma-1}} \cdot \frac{(q/k)!}{\gamma!(q/k-\gamma)!} &\stackrel{?}{=} \frac{1}{(2^h)^{\gamma-1}} \cdot \frac{q!}{\gamma!(q-\gamma)!} \\ k^\gamma \cdot \frac{(q/k)!}{(q/k-\gamma)!} &\stackrel{?}{=} \frac{q!}{(q-\gamma)!} \end{aligned}$$

Note that, in the context of SPHINCS, $q \gg k$ and $q \gg \gamma$, so that k and γ become negligible, leaving us with

$$\frac{q!}{q!} = \frac{q!}{q!}.$$

Thus, the probability that the same HORST key pair is used γ or more times has roughly the same upper bound in classic SPHINCS and in SPHINCS with sequential batch signing.

We will return to these formulas in Section 4.7 when discussing parameters, and show that the probability of a γ-collision for SPHINCS-256 parameters is equally unlikely in classic SPHINCS and SPHINCS with sequential batch signing.

4.5 Approach 2: Add a short-term subtree below HORST

In this approach, batch signing is implemented by adding a short-term subtree of height h_b underneath a randomly chosen leaf i_l of the hypertree, and using the WOTS key pairs at the leaves of this subtree to sign a sequence of messages. The HORST key pair at leaf i_l of the hypertree is then used to sign the two nodes on layer $h_b - 1$ of the short-term subtree. Section 4.5.1 will explain why HORST is used to sign two nodes, rather than simply signing the root of the short-term state subtree.

This approach will be referred to as *subtree* batch signing, since it essentially adds a dedicated subtree to the hypertree.

State initialization. Given a SPHINCS secret key, choose a leaf of the hypertree by (randomly) choosing an index $i_l \in [0, 2^h - 1]$.

Now, randomly produce a seed seed_W to generate the WOTS key pairs of the short-term subtree. Produce the two nodes $n_{h_b-1,0}, n_{h_b-1,1}$ of the short-term subtree, and sign $n_{h_b-1,0}||n_{h_b-1,1}$ with the HORST key pair at index i_l, producing HORST signature s_H. Then produce the rest of the SPHINCS signature that allows the verifier to restore the root of the hypertree. This signature is stored in the state so that the signer does not need to recompute it later. The short-term state is then $(0, i_l, \text{seed}_W, s_H, (s_0, a_0), \ldots, (s_d, a_d))$. Here, the first element is the index of the leaf in the short-term subtree that should be used to sign the next message, and is initialized to 0.

Signing. Given message m, SPHINCS secret key sk and short-term state $(i_s, i_l, \text{seed}, s_H, (s_0, a_0), \ldots, (s_d, a_d))$, check if $i_s \geq 2^{h_b}$. In that case, abort with an error, since the signing capacity of the short-term state has been exhausted.

Otherwise, produce the WOTS key pair at index i_s of the short-term subtree from the seed in the short-term state. Produce a message hash from the message m, the seed in sk, and the indices i_l and i_s, yielding m_h. Sign the message hash m_h using this key pair, producing WOTS signature s_W. Compute the authentication path a_s that allows the verifier to restore the two nodes $n_{h_b-1,0}, n_{h_b-1,1}$ of the short-term subtree. The signature is then $(m_h, i_s, i_l, s_W, a_s, s_H, (s_0, a_0), \ldots, (s_d, a_d)$.

State incrementation. Whenever a message was signed using the short-term state $(i_s, i_l, \text{seed}, s_H, (s_0, a_0), \ldots, (s_d, a_d))$, increment i_s.

Verifying. Given message m, SPHINCS public key pk and signature $(m_h, i_s, i_l, s_W, a_s, s_H, (s_0, a_0), \ldots, (s_d, a_d)$, the verifier restores the nodes $n_{h_b-1,0}, n_{h_b-1,1}$ of the short-term subtree using m, m_h, s_W, i_s and a_s. They then restore the public key pk_H of the used HORST key pair from the two nodes and s_H. After that, the verifier works their way up through the hypertree using the WOTS signature s_i and authentication path a_i on each hypertree layer i. This produces

the root r_h of the hypertree. The signature is valid if this root matches the root stored in the SPHINCS public key pk.

Querying. Given short-term state $(i_s, i_l, \text{seed}, s_H, (s_0, a_0), \ldots, (s_d, a_d))$, return $i_s - 2^{h_b}$.

4.5.1 Impact on security assumptions

In SPHINCS-256, HORST is configured to sign a message hash of 64 bytes. In this approach, HORST signs the two nodes on layer $h_s - 1$ of the short-term subtree. This construction might seem rather arbitrary. This section will analyze why HORST cannot be used to simply sign the root of the short-term subtree, which has only 32 bytes, while providing 256 bits of pre-quantum security.

The size of the signed message has an impact on the security of the scheme. Let us assume that HORST was used to sign just the 32 bytes root of the subtree. With $t = 2^{16}$, each signature would reveal $k = 256/16 = 16$ elements to sign the $32 \cdot 8 = 256$ bits of the subtree root. As outlined in [44], HORS provides $k(\log t - \log k - \log r)$ bits of pre-quantum security after r messages have been signed. With $t = 16$ and $k = 16$, the scheme only offers 176 bits of security after the first message was signed, and that quickly deteriorates as r grows.

For comparison, SPHINCS-256 uses $t = 2^{16}$ and $k = 32$, yielding 352 bits of pre-quantum security after the first message was signed, and 256 bits of security after 8 messages have been signed.

Of course the 32 bytes could be expanded to 64 bytes using the same function that maps a message to a message hash. But since there are only $2^{32 \cdot 8} = 2^{256}$ possible values for the root of the short-term state subtree, these values can only ever map to 2^{256} values in the co-domain of the message hash, since that hash function is deterministic. This does not line up with the model that is used in [44], which assumes that the hash function is a random oracle.

Figure 4.1 plots $f_b(t) = k(\log_2(t) - \log_2(k) - 1), k = b/\log_2(t)$ for $b = 512$ (upper curve), and $b = 256$ (lower curve), or how many bits of security are provided by HORS when signing 512-bit or 256-bit messages, respectively, after one message has been signed.

The graph clearly shows that no practical parameter choice can reach 256-bits of pre-quantum security when signing 256-bit messages. Increasing t increases the security of HORS, but only logarithmically. In addition to that, it would harm the performance of HORS as the number of key elements would grow exponentially. Decreasing t would increase k, but HORS offers poor security as $\log_2 k$ approaches $\log_2 t$.

In conclusion, it is not practical to achieve 256-bits of pre-quantum security in HORS when signing 256-bit messages. This is why HORST is used to sign the two nodes on layer $h_b - 1$ instead.

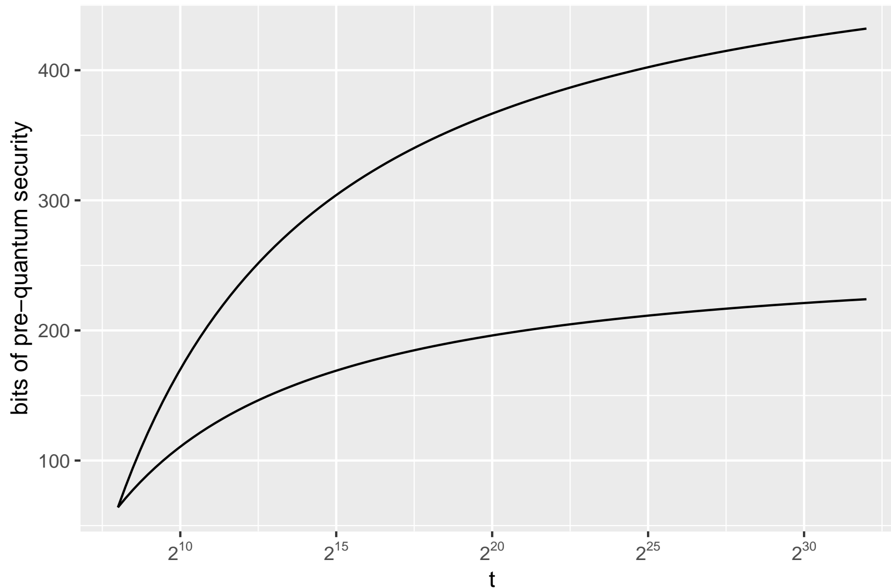

Figure 4.1: The security provided by HORS when signing 512 bit messages (upper curve) and 256 bit messages (lower curve), after one message has been signed.

4.6 The two approaches in comparison

This section gives only a brief, theoretical comparison of the two approaches. A more in-depth comparison follows in Chapter 5, along with speed measurements.

Signatures produced with sequential batch signing are identical to the signatures produced by SPHINCS. This has the advantage that the verifier does not need to know whether the signer used batch signing to produce this signature. The signer can thus freely decide to use batch signing as needed. This is not the case for subtree batch signing, since those signatures are different from traditional SPHINCS signatures.

Signatures produced with subtree batch signing do not need to recompute the HORST signature. Combined with the fact that WOTS public keys can be cached in the short-term state, this reduces the work per signature significantly for subtree batch signing, as Chapter 5 will show.

4.7 Parameter discussion

The two approaches were designed with different goals in mind. While sequential batch signing serves as a drop-in replacement for traditional SPHINCS-256, subtree batch signing is aimed at situations that can fully utilize batch signatures. This section discusses the decision process behind the parameters that were chosen for the two variants.

4.7.1 Sequential batch signing

Sequential batch signing uses the same parameters as SPHINCS-256, so that it can be used in any context where SPHINCS-256 is used today: For individual signatures, it can be as fast as SPHINCS-256. But if the short-term state is used to its full potential, signatures are about four times faster thanks to the cached signature elements. Finally, just like with SPHINCS-256, up to 2^{50} messages can safely be signed with a single key pair, no matter if the signer uses a short-term state just for a single signature, or to its full potential.

SPHINCS-256 parameters were chosen so that each HORST key pair can be reused up to eight times before security drops below 256 bit. With a hypertree height of $h = 60$, the probability that any HORST key pair is used nine times or more when signing $q = 2^{50}$ messages is at most

$$Pr[C(2^{60}, 2^{50}, 9)] \leq \frac{1}{(2^{60})^8} \binom{2^{50}}{9} \approx 2^{-48},$$

using the upper bound formula in [48, p. 32].

Recall that the probability of such a multi-collision in the case of sequential batch signing needs to be computed with Formula 4.4, instead. This formula is parameterized with the number of messages signed with each short-term state, but we find that for any value of k for $1 \leq k \leq 32$,

$$Pr[C_{STS}(60, 5, 2^{50}, k, 9)] \leq \frac{k^9}{(2^{60})^8} \cdot \binom{2^{50}/k}{9} \approx 2^{-48}, 1 \leq k \leq 32.$$

Thus, when using sequential batch signing, it remains sufficiently unlikely that any HORST key pair is used nine times or more, no matter how many messages are signed with each short-term state.

4.7.2 Subtree batch signing

Subtree batch signing offers more flexibility in its parameter choices. Its design goal is to offer a significant speed-up compared to simple SPHINCS-256 signatures, without a significant increase in signature size, or reduced security. Furthermore, the height of the short-term state subtree can be different from the height of the other subtrees in the hypertree. This adds some flexibility to balance short-term state initialization time with the number of signatures that can be created with each short-term state.

Hypertree height. The total height of the hypertree determines the number of HORST key pairs at the leaf nodes of the hypertree. For subtree batch signing, each short-term state creates a subtree that is signed by one such HORST key pair. The number of HORST key pairs affects the number of short-term states that can be initialized with the same key pair, since each HORST key pair is only safe to use for a limited number of signatures.

The hypertree height is set to $h = 60$, matching the hypertree height of SPHINCS-256. With this, up to 2^{50} short-term states can be initialized, while maintaining 256 bits of pre-quantum security.

Subtree height. The large hypertree in SPHINCS consists of many layers of smaller subtrees. This way, the hypertree can have as many as 60 layers, while the tree never needs to be computed in its entirety.

The height of the subtrees has no effect on the security of the scheme, but is chosen as a trade-off between signature size and signing speed. In classic SPHINCS, the subtree height h_s and the total height $h = 60$ of the hypertree determine the number of subtree layers $d = 60/h_s$. For each subtree layer, one WOTS signature is included in the SPHINCS signature. This makes the subtree height roughly inverse proportional to the signature size. At the same time, changing the subtree height changes the signing speed exponentially: Increasing the subtree height by 1 doubles the number of WOTS key pairs at the leaf nodes of every subtree.

Subtree batch signing introduces a second tree height, h_b, for the height of the short-term state subtree. It is important to understand the difference between the roles of these two parameters. Changing h_b changes the signature size only marginally via the number of nodes in the short-term state subtree authentication path.

The choice of h_b has an exponential effect on the initialization time of a new short-term state; increasing h_b by one doubles the number of WOTS key pairs that need to be generated to build the short-term state subtree. However, this cost is averaged over all signatures created with that short-term state.

More importantly, we can pick h_b to compensate larger choices of h_s: Incrementing h_s doubles the work necessary to produce all WOTS key pairs in the hypertree. However, with subtree batch signing, that work is averaged over all signatures created with a single short-term state. Classic SPHINCS had to be conservative with h_s to balance signature size with signing speed. With subtree batch signatures, we can increase h_s further to reach smaller signatures, and increase h_b to average the additional work over more signatures.

The choice of h_s and h_b also affects the size of the short-term state, but since the short-term state is only stored locally, we did not consider its size when fixing parameters.

Figures 4.2, 4.3, and 4.4 illustrate the relations outlined above. On that basis, we define $h_s = 10$, and $h_b = 12$, implying $d = 6$.

With this configuration, up to 2^{62} messages can be signed with one key pair, if all short-term states are used to their full potential. Initializing the short-term state takes as long as 32 classic SPHINCS signatures, but once the short-term state is created, up to 4096 messages can be signed with it. We justify the high initialization time of the short-term state with the design goal of this approach: These parameters are chosen for cases where the short-term state is usually used to sign as many messages as possible. For other cases, sequential batch signatures might be a better choice.

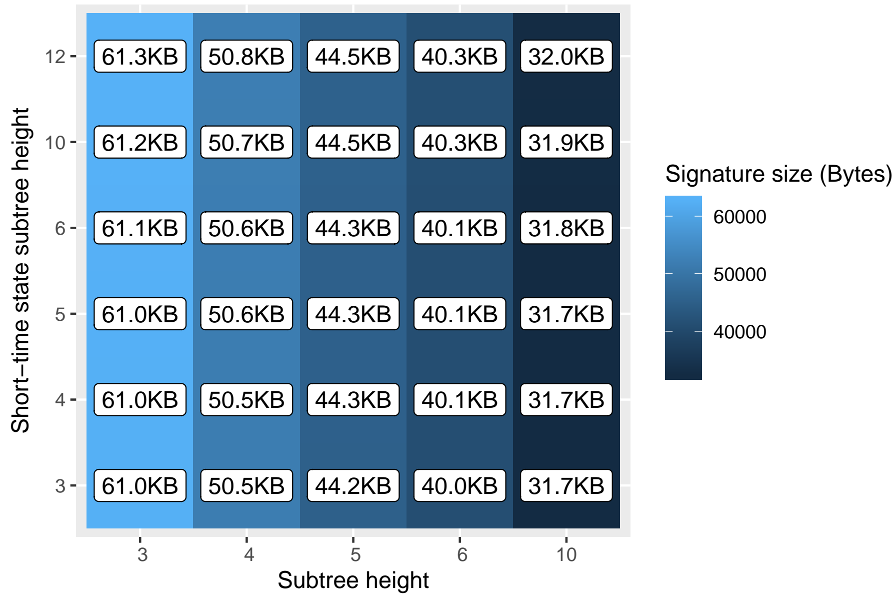

Figure 4.2: The signature size for various choices of subtree height h_s and short-term state subtree height h_b.

On average, these parameters speed up signing by a factor of 128, compared to classic SPHINCS, while reducing the size of the signature from $41,000$ bytes to $32,720$ bytes.

4.8 Creating successive short-term states

The sequential batch signing approach utilizes the overlap between signatures created with neighboring leaf nodes. This approach can be taken a step further to speed up the creation of successive short-term states, for both sequential batch signing and subtree batch signing. This idea has not been implemented for this thesis, but the potential benefit is promising.

Utilizing the BDS algorithm from [13], it should be possible to cache a number of nodes throughout the hypertree to quickly produce a new short-term state. The parameters of the BDS algorithm can be tweaked to balance the number of cached nodes against the computational cost to create a new short-term state.

The creation of the next short-term state could even be off-loaded to a different thread, core, or computer, while another short-term state is used to sign messages.

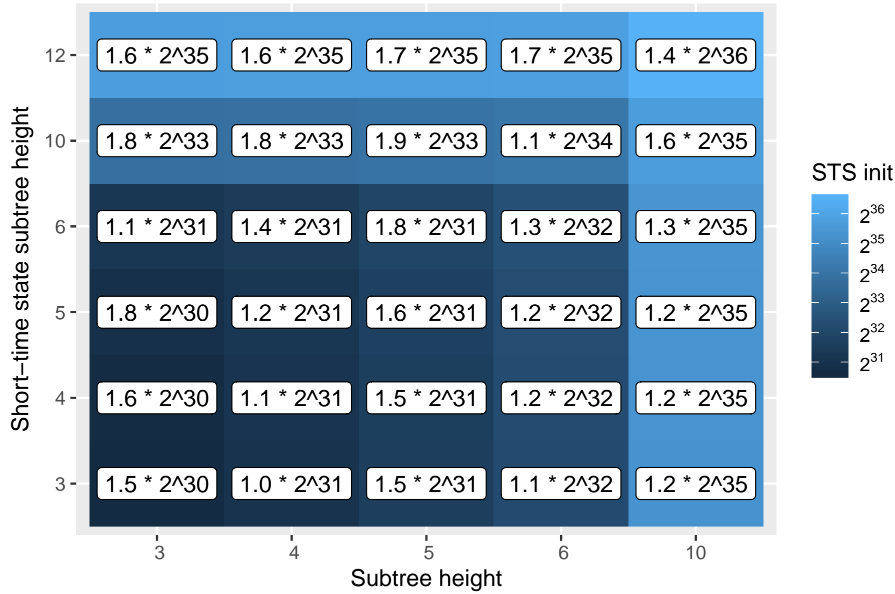

Figure 4.3: The cost, in cycles, to initialize a new short-term state, for various choices of subtree height h_s and short-term state subtree height h_b.

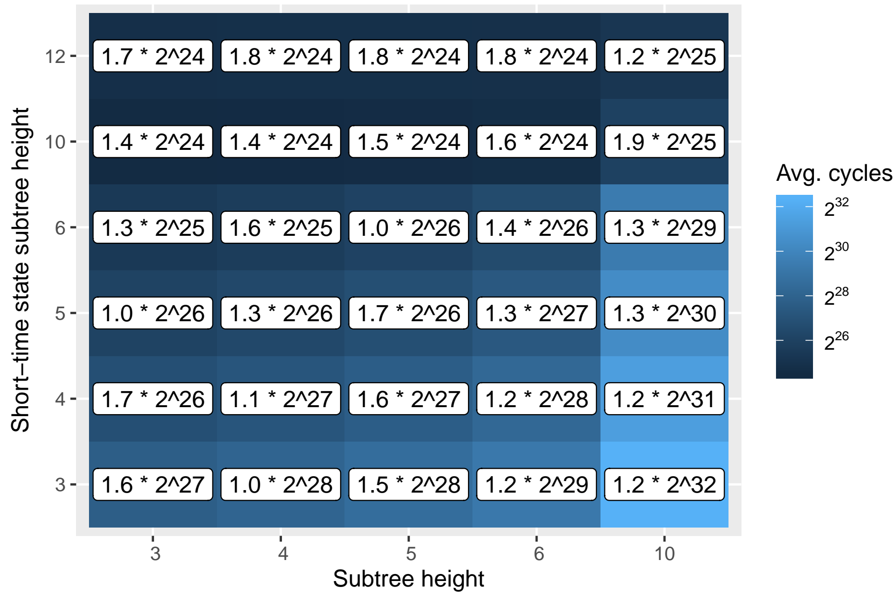

Figure 4.4: The average cost, in cycles, of one signature, for various choices of subtree height h_s and short-term state subtree height h_b. The cycle count includes the cost to initialize the short-term state, averaged over all signatures that can be created with one short-term state.

Chapter 5

Implementation and results

This chapter shows how batch signing speeds up signature creation in practice. The first part of this chapter will explain in general how signing speed benefits from batch signing. The second part will then show specific measurements to compare traditional SPHINCS signatures to the two batch signing variants.

Measurements in this chapter are based on the cycle count reported by the CPU. So far, the short-term state variants of SPHINCS have not been heavily optimized, and are based on the reference implementation of SPHINCS, available in the SUPERCOP benchmarking suite[1].

5.1 Benefits of batch signing

There are two major effects that cause SPHINCS batch signatures to be much faster than traditional SPHINCS signatures. Firstly, batch signing utilizes the fact that two signatures are very similar if their used leaf nodes are close to each other. In traditional SPHINCS, when two messages are signed with two leaves that are part of the same subtree on layer 0, then 25 of the 41kB of those signatures will be identical. Batch signing benefits from this fact and computes the unchanged parts of the signature only once.

Secondly, batch signing can further reduce the amount of data that is computed for each signature by caching the WOTS public keys of the lowest subtree. Recall that the SPHINCS signature contains authentication paths for each subtree. The authentication paths of all subtrees on layer 1 and above are only computed once, and are stored in the short-term state. The authentication path for the lowest subtree, however, changes with each signature, since each signature uses a different leaf of the subtree.

Computing the authentication path of a subtree requires the WOTS public keys that form the leaf nodes of that subtree. Since the WOTS key pairs are usually generated on demand, obtaining these leaf nodes is a serious effort. For SPHINCS-256 with WOTS parameters $l = 67$ and $w = 16$, this requires

[1] `http://bench.cr.yp.to/supercop.html`

$16 \cdot 67 + 2^7 - 1 = 1199$ hashes. For subtree batch signing, where $l = 131$, even 2223 hashes need to be computed.

Thus, caching the WOTS public keys in the short-term state saves a lot of hashes. This gives another significant boost to the speed of batch signatures. Caching these values does have an impact on the memory footprint, though, as the 4096 WOTS public keys account for $2^{12} \cdot 32 = 131,072$ bytes of the short-term state size.

If it is not an option to store all WOTS public keys in the short-term state, then techniques similar to those used in [32], based on the work in [13], can be applied as a trade-off between the number of WOTS signatures stored in the short-term state and the number of WOTS signatures that have to be re-computed for each new authentication path. More specifically, with the algorithm presented in [13], one can choose parameter $K, K \geq 2$, such that $12 - K$ is even, as a trade-off between the number of nodes that need to be cached in the short-term state, and the number of leaf nodes and inner nodes that need to be computed for each new signature [13, Theorem 2, with $H = 12$]. For example, with $K = 2$, the short-term state would need to store 32 nodes, and 6 leaf nodes would need to be computed for each new signature. With $K = 10$, the short-term state would need to store 1004 nodes, but only 2 leaf nodes would need to be computed for each new signature.

5.2 Results

This section will first show how the batch-signing variants compare to traditional SPHINCS signatures in general. It will then present the benefits of batch signatures as compared to traditional signatures. Finally, it will show how the subtree height affects the performance. In this section, the *cost* of a signature refers to the measured number of cycles that it took to generate this signature.

Note that the performance of the batch signing variants is compared to the reference implementation of SPHINCS, not the optimized implementation of SPHINCS. The implementations were benchmarked on an Intel Haswell $i5 - 4690K$ CPU running at 4.3 GHz, with Turbo Boost and hyper-threading disabled. The following code was used to query the current cycle count:

```
1  static __inline__ unsigned long GetCC(void)
2  {
3    unsigned a, d;
4    asm volatile("rdtsc" : "=a" (a), "=d" (d));
5    return ((unsigned long)a) | (((unsigned long)d) << 32);
6  }
```

Table 5.1 shows the sizes of signatures and key pairs for traditional SPHINCS signatures as well as for the two batch signature variants. Signatures produced by either of the two batch signing variants are not larger than SPHINCS signatures; in fact, when using the *subtree* approach, the produced signature will even be over $8KB$ smaller than traditional SPHINCS signatures, due to fewer subtree layers in the hypertree.

	signature	pk	sk	short-term state
SPHINCS	$41,000B$	$64B$	$96B$	n/a
subtree	$32,720B$	$64B$	$96B$	$159,280B$
sequential	$41,000B$	$64B$	$96B$	$26,384B$

Table 5.1: Size of signature, key pairs and short-term state.

Figure 5.1 shows various performance measurements to illustrate the differences between classic SPHINCS and the two batch signing variants. The data shows that there is no performance penalty if a user was to use sequential batch signing for just a single signature: Initializing a short-term state for sequential batch signing, and then signing a message takes just as long as a classic SPHINCS signature. Similarly, creating a key pair for sequential batch signing is just as fast as it is for classic SPHINCS.

When sequential batch signing is used to sign the full 32 messages that can be signed with a short-term state, then on average a message can be signed about 4.7 times faster than with classic SPHINCS.

These results confirm that sequential batch signatures are a suitable drop-in replacement for classic SPHINCS: Signatures are just as fast when the short-term state is used to sign just one message, and the better the short-term state is utilized, the faster is the signing process. At the same time, the number of messages that can be signed with one key pair is not affected by the use of the short-term state.

Figure 5.1 also shows that the results for subtree batch signing look noticeably different. Creating a key pair for subtree batch signing takes considerably longer than for the other two variants, due to the taller subtrees in the hypertree. Similarly, initializing a short-term state is a significant effort at over 2^{36} cycles. Because of that, subtree batch signing performs poorly if a short-term state was only used to sign a single message.

However, when the short-term state is used to its full potential, subtree batch signing outperforms both classic SPHINCS as well as sequential batch signing by a large margin. Averaged over all 4096 signatures that can be created with a single short-term state, each signature only costs slightly over 2^{25} cycles. Compared to the reference implementation of classic SPHINCS, that is a speed-up by a factor of 86.

Finally, verifying is slightly faster for subtree batch signing, due to the reduced number of WOTS signatures that have to be verified.

5.2.1 Utilization

After a short-term state has been initialized, it can be used to produce a fixed number of signatures. In practice, it might be difficult to make sure that each short-term state is perfectly utilized. Also, especially if signatures are created infrequently, it might be tempting to use SPHINCS without the batch signing capabilities introduced in this thesis, so that the short-term state does not need to be cached in between signatures.

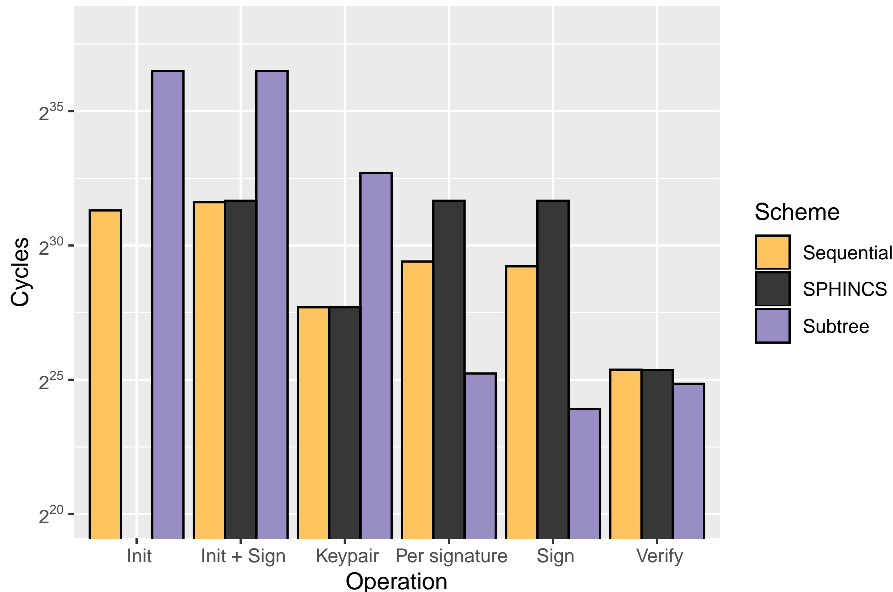

Figure 5.1: The cost of various operations for the different schemes. Here, *Keypair* is the cost for creating a key pair, *Init* is the cost for initializing a short-term state, *Sign* and *Verify* are the costs for one signature and verification, respectively. *Init + Sign* shows the cost of the first signature, ignoring all benefits of batch signing. *Per signature* shows the benefit of batch signing, as the cost to initialize a short-term state is averaged over the 32 or 4096 signatures that can be created from one short-term state, using sequential or subtree batch signing, respectively.

Figure 5.2 shows how the cost per signature changes as utilization changes. These measurements show that short-term states do not need to be fully utilized in order to benefit from the performance boost. In fact, the performance boost is significant even if as little as 25% of the signatures are actually used.

Furthermore, we saw before that it is a significant computational effort to create short-term states for subtree batch signatures. The results in Figure 5.2 show that this effort pays off long before a significant portion of the short-term state is utilized. As expected, the cost per signature with subtree batch signatures is higher than the cost per signature for the other schemes, if the short-term state is only used for a small number of signatures. However, as long as more than 32 messages are signed with each short-term state, it is faster to create and use a subtree short-term state than to use either of the other methods.

5.3 Conclusion

This thesis introduced two approaches to batch signatures in SPHINCS. Sequential batch signatures are a suitable drop-in replacement for classic SPHINCS, with no performance penalty when it is used just for a single signature, and

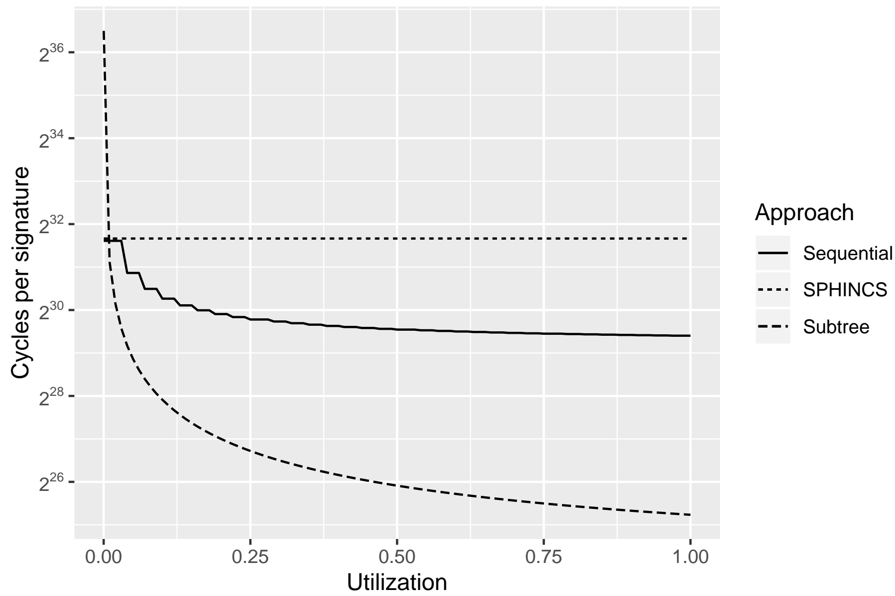

Figure 5.2: The cost per signature relative to the utilization of the short-term state. Here, a utilization of p means that $100 \cdot p\%$ of the signatures in the short-term state have been used.

$4\times$ faster signatures when utilizing at least half of the signing capacity of each short-term state. Subtree batch signatures are a powerful solution for situations in which batch signing can be fully utilized, offering smaller signatures, $16\times$ faster signatures at 25% utilization, and $80\times$ faster signatures at 100% utilization.

Future work should explore options to speed up the creation of successive short-term states to further speed up signature creation.

Bibliography

[1] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. https://authors.library.caltech.edu/99516/2/41586_2019_1666_MOESM1_ESM.pdf.

[2] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, pages 219–242, Cham, 2018. Springer International Publishing. https://eprint.iacr.org/2017/933.pdf.

[3] Sundar Balasubramanian, Harold W Carter, Andrey Bogdanov, Andy Rupp, and Jintai Ding. Fast multivariate signature generation in hardware: The case of rainbow. In *2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 25–30. IEEE, 2008. https://sci-hub.se/10.1109/asap.2008.4580149.

[4] Paulo SLM. Barreto, Rafael Misoczki, and Marcos A. Simplicio Jr. One-time signature scheme from syndrome decoding over generic error-correcting codes. *Journal of Systems and Software*, 84(2):198–204, 2011. https://sci-hub.se/10.1016/j.jss.2010.09.016.

[5] Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 470–484. Springer Berlin Heidelberg, 1997. https://cseweb.ucsd.edu/~mihir/papers/tcr-hash.pdf.

[6] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978. https://authors.library.caltech.edu/5607/1/BERieeetit78.pdf.

[7] Daniel J. Bernstein. Grover vs. McEliece. In Nicolas Sendrier, editor, *Post-Quantum Cryptography*, pages 73–80, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://cr.yp.to/codes/grovercode-20100303.pdf.

[8] Daniel J. Bernstein, Jean-François Biasse, and Michele Mosca. A low-resource quantum factoring algorithm. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography*, pages 330–346, Cham, 2017. Springer International Publishing. https://eprint.iacr.org/2017/352.

[9] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Marc Fischlin and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer Berlin Heidelberg, 2015. Document ID: 5c2820cfddf4e259cc7ea1eda384c9f9, `https://cryptojedi.org/papers/#sphincs`.

[10] Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, and Christopher Wolf. Time-area optimized public-key engines: $\mathcal{MQ}$-cryptosystems as replacement for elliptic curves? In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, pages 45–61, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. `https://link.springer.com/content/pdf/10.1007/978-3-540-85053-3_4.pdf`.

[11] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, 2017. `https://arxiv.org/pdf/1608.00263.pdf`.

[12] 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*, pages 117–129. Springer Berlin Heidelberg, 2011. `https://eprint.iacr.org/2011/484.pdf`.

[13] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 63–78. Springer Berlin Heidelberg, 2008. `https://www-old.cdc.informatik.tu-darmstadt.de/reports/reports/BDS08.pdf`.

[14] Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS – an improved Merkle signature scheme. In Rana Barua and Tanja Lange, editors, *Progress in Cryptology - INDOCRYPT 2006*, pages 349–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. `https://eprint.iacr.org/2006/320.pdf`.

[15] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. `https://link.springer.com/content/pdf/10.1007/978-3-642-25385-0_1.pdf`.

[16] Nicolas T. Courtois, Matthieu Finiasz, and Nicolas Sendrier. How to achieve a McEliece-based digital signature scheme. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 157–174, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. `https://link.springer.com/content/pdf/10.1007/3-540-45682-1_10.pdf`.

[17] 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: Second International Workshop, PQCrypto 2008 Cincinnati, OH, USA, October 17-19, 2008 Proceedings*, pages 109–123, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://whereisscihub.now.sh/go/10.1007/978-3-540-88403-3_8.

[18] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976. https://caislab.kaist.ac.kr/lecture/2010/spring/cs548/basic/B08.pdf.

[19] Hang Dinh, Cristopher Moore, and Alexander Russell. McEliece and Niederreiter cryptosystems that resist quantum Fourier sampling attacks. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 761–779, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-22792-9_43.pdf.

[20] Chris Dods, Nigel P. Smart, and Martijn Stam. Hash based digital signature schemes. In Nigel P. Smart, editor, *Cryptography and Coding*, pages 96–115, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. https://whereisscihub.now.sh/go/10.1007/11586821_8.

[21] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 40–56, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-40041-4_3.pdf.

[22] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 263–275, New York, NY, 1990. Springer New York. https://sci-hub.se/10.1007/0-387-34805-0_24.

[23] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 31–51, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-540-78967-3_3.pdf.

[24] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 197–206, New York, NY, USA, 2008. Association for Computing Machinery. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.4269&rep=rep1&type=pdf.

[25] Oded Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 104–110, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-47721-7_8.pdf.

[26] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 112–131, Berlin, Heidelberg,

1997. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/BFb0052231.pdf.

[27] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A "paradoxical" solution to the signature problem. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 467–467. Springer Berlin Heidelberg, 1985. https://people.csail.mit.edu/rivest/pubs/GMR84b.pdf.

[28] Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, 79(2):325, 1997. https://arxiv.org/pdf/quant-ph/9706033.pdf.

[29] 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*, pages 530–547, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-33027-8_31.pdf.

[30] Sean Hallgren and Ulrich Vollmer. *Quantum computing*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. http://www.ic.unicamp.br/~rdahab/cursos/mo421-mc889/2014-1s/Welcome_files/Post%20Quantum%20Cryptography%20(Bernstein,%20Buchmann,%20Dahmen,%202009).pdf#page=23.

[31] 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*, pages 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://eprint.iacr.org/2017/965.pdf.

[32] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSSMT. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, pages 194–208, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://eprint.iacr.org/2017/966.pdf.

[33] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography – PKC 2016*, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. https://eprint.iacr.org/2015/1256.pdf.

[34] Hideki Imai and Tsutomu Matsumoto. Algebraic methods for constructing asymmetric cryptosystems. In Jacques Calmet, editor, *Algebraic Algorithms and Error-Correcting Codes*, pages 108–119, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. https://sci-hub.se/10.1007%2F3-540-16776-5_713.

[35] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1(1):36–63, 2001. https://pdfs.semanticscholar.org/c26e/3c42c2a85e2479c1316ccc8c20754533e406.pdf.

[36] Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2):375–400, Dec 2015. https://link.springer.com/content/pdf/10.1007/s10623-015-0067-5.pdf.

[37] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979. http://lamport.azurewebsites.net/pubs/dig-sig.pdf.

[38] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 738–755, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-29011-4_43.pdf.

[39] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Coding Thv*, 4244:114–116, 1978. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19780016269.pdf#page=123.

[40] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY, 1990. Springer New York. https://link.springer.com/content/pdf/10.1007%2F0-387-34805-0_21.pdf.

[41] Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 271–288, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/11761679_17.pdf.

[42] Jacques Patarin. Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt'88. In Don Coppersmith, editor, *Advances in Cryptology — CRYPT0' 95*, pages 248–261, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-44750-4_20.pdf.

[43] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. Leveraging secondary storage to simulate deep 54-qubit sycamore circuits, 2019. https://arxiv.org/abs/1910.09534.

[44] 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*, pages 144–153, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://eprint.iacr.org/2002/014.pdf.

[45] Joost Rijneveld. Implementing SPHINCS with restricted memory. Master's thesis, Radboud University, May 2015. https://joostrijneveld.nl/theses/sphincs_cortexm3/20150528_msc_sphincs_cortexm3.pdf.

[46] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. https://apps.dtic.mil/dtic/tr/fulltext/u2/a606588.pdf.

[47] Peter W. Shor. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. pages 289–289, 1994. https://adsabs.harvard.edu/abs/1999SIAMR..41..303S.

[48] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006*, pages 29–40, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://sci-hub.se/10.1007/11927587_5.

[49] Joop van de Pol. Lattice-based cryptography. Master's thesis, Eindhoven Univ. of Technology, 2011. https://pure.tue.nl/ws/portalfiles/portal/47023806/719274-1.pdf.

[50] Christopher Wolf and Bart Preneel. Taxonomy of public key schemes based on the problem of multivariate quadratic equations. *IACR Cryptology ePrint Archive*, 2005:77, 2005. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.2940&rep=rep1&type=pdf.

[51] 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*, pages 73–88, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. http://precision.moscito.org/by-publ/recent/39340073.pdf.

Appendix A

Fast and flexible heterogeneous buffers in C

In a few places across the SPHINCS code base, a single byte buffer is used to store heterogeneous data. For example, the API returns a SPHINCS signature in a single byte buffer, but this buffer contains different, distinct elements like the index of the used hypertree leaf, or the d WOTS signatures.

During development this pattern can become difficult to maintain, since the positioning of each distinct element within this buffer depends on the order and size of other elements. In practice, changing the order of elements in the buffer required changes to the code in numerous places.

Using `struct`s and a multitude of buffers instead of a single buffer would of course make it much easier to access the various elements, but only at the cost of not having the data stored in a continuous chunk of memory.

This section introduces a hybrid solution that can be useful in these situations. The data can still be stored in a single buffer, but is easily accessible, as if it was stored in a `struct`.

To show how this solution works in practice, consider a case where data about cookie recipes should be stored in buffers. This data consists of four distinct elements: The name, stored as a 32 bytes character array, the number of servings and the calories, both stored as an `int`, and finally the URL that points to the recipe, stored as a 2000 bytes character array.

First, a `struct` is defined that contains pointers to each element.

```
1  struct cookie_data {
2    unsigned char* name;
3    int* servings;
4    int* calories;
5    unsigned char* url;
6  };
```

Then, a function is defined that can populate such a struct.

```
1  #define NAME_BYTES 32
```

```c
#define URL_BYTES 2000
#define COOKIE_BYTES (NAME_BYTES + sizeof(int) + \
                      sizeof(int) + URL_BYTES)

const struct cookie_data
init_cookie_data(unsigned char* buffer)
{
  struct cookie_data data;
  int offset = 0;

  data.name = buffer + offset;
  offset += NAME_BYTES;

  data.servings = (int*) (buffer + offset);
  offset += sizeof(int);

  data.calories = (int*) (buffer + offset);
  offset += sizeof(int);

  data.url = buffer + offset;
  offset += URL_BYTES;

  assert(offset == COOKIE_BYTES);

  return data;
};
```

Now elements of this buffer can be accessed like so:

```c
int main(int nargs, const char** args)
{
  unsigned char buffer[COOKIE_BYTES];

  struct cookie_data data = init_cookie_data(buffer);

  strncpy(data.name, "Hazelnut␣butter␣cookies", NAME_BYTES);
  data.name[NAME_BYTES - 1] = 0;
  *data.servings = 6;
  *data.calories = 320;
  strncpy(data.url,
          "https://www.recipes.com/hazelnut-butter-cookies",
          URL_BYTES);
  data.url[URL_BYTES - 1] = 0;

  printf("%s\n", data.name);
  printf("%d\n", *data.servings);
  printf("%d\n", *data.calories);
  printf("%s\n", data.url);

  return 0;
}
```

Adding elements, or changing the order of elements is as easy as changing the offsets inside the function `init_cookie_data`.

An alternative solution is to have cascaded constants for the offsets of all elements:

```c
  #define OFFSET_NAME 0
  #define OFFSET_SERVINGS (OFFSET_FLAVOR + NAME_BYTES)
  #define OFFSET_CALORIES (OFFSET_SERVINGS + sizeof(int))
  #define OFFSET_URL (OFFSET_CALORIES + sizeof(int))
```

This approach has the advantage that it does not require the initialization of a struct to access elements of the buffer. However, it does not offer the convenience that elements can be accessed by their type without explicit casting.