Implementing SPHINCS with restricted memory

by Joost Rijneveld

Master Thesis in Computer Science
Supervised by dr. Peter Schwabe
Second supervisor: dr. Andreas Hülsing
Second reader: dr. Lejla Batina
Radboud University Nijmegen
joost@joostrijneveld.nl
May 2015

Abstract

There is an ever-growing chance that quantum computers will become a reality within the near future. As much of today's cryptography relies on the hardness of problems that can be solved orders of magnitudes faster using quantum algorithms, there is a need for new schemes. One of the candidate replacement digital signature schemes is SPHINCS: a stateless hash-based signature scheme with a practical key and signature size (1KB per key and 41KB for the signature), as well as fast performance. In this thesis, SPHINCS-256 is implemented on a Cortex M3-based system with only 16KB of RAM available, running at 32MHz. At 52 seconds, producing a signature takes a significant amount of time but is not entirely impractical. Moreover, these results are in line with what is to be expected of the given platform when comparing it to the settings and results described in the SPHINCS design paper. This thesis describes the relevant cryptographic context, provides an overview of the workings of SPHINCS and details the key implementation changes necessary to make the scheme run on the chosen platform.

Acknowledgements

First and foremost, I would like to thank Peter Schwabe for his continuous support and guidance as my supervisor throughout this project. I greatly appreciate his enthusiasm and dedication, and derived much motivation from this. Thanks go to Andreas Hülsing, who provided helpful background information on hash-based schemes, as well as valuable answers to questions that came up while putting it in practice. Thanks also go to Lejla Batina, who was willing to take the time to act as a second reader for this thesis, and to Harm Berntsen, with whom I had an illuminating discussion on clock configuration. Finally, I would like to thank my friends and family for reading various fragments and drafts of this thesis, and for supporting me during the entire process. Thank you all.

Contents

Introduction

In the highly digitized world of today, cryptography is much more present than many people would suspect. We, as a society, have become highly dependent on it for our everyday business, as much of what we do is being influenced by some form of digital communication. We use cryptography to keep things hidden from prying eyes, but ever so often to protect the integrity or authenticity of data as well. This does not only concern the things we do online, but also many of the computerised systems around us: what to think of public transport or ATM cards, or access control systems? For many applications, knowing the origin and being certain of the integrity of some piece of information is at least as important as concealing it.

These systems typically rely on mathematical 'signatures' that cannot be forged by any computer in existence, or even computers that are to be expected in a few decades. However, it is becoming ever more likely that an entirely new class of computers may be built in the near future: quantum computers. The (envisioned) working of these futuristic machines is not relevant to this thesis, but the crux lies in their computational power. Quantum computers turn out to be excellent at solving a very specific set of previously unsolvable mathematical problems, and the fact that we cannot solve these problems is what a large part of much of modern cryptography relies on. If the quantum computer would become realistic and practical[1], it would all break down.

While research in the area of post-quantum cryptography (i.e., algorithms that can survive quantum computers) has been ongoing, it is still far from practical to use. More work is required in order to find out which algorithms are the best suitable replacements for our current-day crypto. In this thesis, one such algorithm is studied by making it run on a small device: the SPHINCS signature scheme.

This thesis consists of three clearly distinct parts. In the first chapter, an introduction to the relevant cryptography will be given, laying out the basic constructions that will form the building blocks for the rest of the thesis. This background context will be a useful aid in the understanding of the other chapters, and aims to assume no prior knowledge on the subject. Chapter two will use these principles to outline the structure and inner workings of the SPHINCS signature scheme. This will be done in a top-down fashion, starting with a global framework and then progressively filling in the missing pieces. The third chapter discusses the main contribution of this work: making SPHINCS work on a constrained platform. In this chapter, the core concepts that make this implementation possible are addressed.

[1]Note that 'realistic and practical' does not imply for consumers or even academia to have a quantum computer at their disposal. Rather, these machines would first become affordable to the large governmental organisations we have all grown a bit more familiar with over the past few years.

Cryptographic context

In this chapter, some background knowledge on cryptography will be introduced, focussing on the context required to be able to better understand the rest of this work. Readers familiar with the material may wish to skip over it.

1.1 Symmetric- and asymmetric key encryption

Historically, cryptography concerned the encryption and decryption of secret messages, typically between two parties. To encipher and decipher a message, all involved parties would have to use some secret method or system that only they knew or were able to use. Some of these systems date back to as early as ancient Greece, but similar systems were created all throughout history ever since. Whenever there was a need for hidden messages and secrecy, people have been devising ways to protect themselves from prying eyes. Such a system would typically be built around some substitution (i.e., replacing letters with other letters) or transposition (i.e., moving letters around) of which the details were kept secret.

Not only the method was to be kept secret; by introducing some secret parameter to the scheme, the encryption would be more difficult to break. This could be a variation in the letters that were used for replacement, or a change in some starting position. The essence of adding a secret parameter, or *key*, would be that to read a message, someone would need to know both the method and this secret information. Crucially, this meant that the same system could be used between multiple parties without them being able to read each others' messages just as easily, as long as each pair of users used a different key.

Through time, it has become clear that it is unwise to rely on keeping the working of a cryptographic system secret. As early as the late nineteenth century, Auguste Kerckhoffs formulated a set of principles on the design of such systems, one of which has come to be known as Kerckhoffs' principle: a cryptographic system should not require secrecy, and should not be compromised when the system itself falls into enemy hands [1].

The system that remains is a symmetric-key system. To encrypt a message, one can use a certain secret key, making the message unreadable for anyone but others who hold the same key. This key is then also used for decryption.

This is a very useful scheme for many applications. However, having to share a key with another user before being able to securely exchange messages can be highly

impractical. Additionally, in a symmetric system each pair (or group) of users needs to share a different key, of which each of the users needs to keep track. It is not hard to imagine that this can become quite a burden. In 1976, Diffie and Hellman suggested a way to solve this: asymmetric cryptography [2].

Diffie and Hellman describe a system that allows users to share secret information without requiring a shared key, through means of *public keys*. Such a system (coined a *public-key cryptosystem*) is described to consist of pairs of functions to encrypt and decrypt information (E_k and D_k), derived from a secret key (k) only known to one user. The crucial difference with the symmetric system lies in the fact that it should not be possible to derive D_k from E_k without knowing k (Diffie and Hellman say: "for almost every k, D_k is computationally infeasible to derive from E_k"). This would make it possible to publicly share E_k while keeping D_k a secret, enabling others to encrypt information that only the holder of k (and thus D_k) can decrypt. In order to communicate secretly, two users would thus only need to have access to each other's public encryption functions E_k. In practice, this is usually achieved by publishing a *public key*, also derived from the secret k, from which E_k can be constructed.

In the paper where Diffie and Hellman proposed the above framework [2], they also discuss potential pairs of functions that satisfy the required property. It turns out to be non-trivial to find sufficiently practical candidates, but the authors do provide a concrete example of such a public key system. This system has come to be known as the 'Diffie-Hellman key exchange', and can be used to establish a shared key without prior arrangements. In the following years, several others have also published feasible implementations that fill the gaps in the public key framework, basing their functions on a variety of mathematical problems. Notable examples are RSA [3], ElGamal [4] and DSA [5], later followed by elliptic-curve cryptography [6, 7].

1.2 Public-key signatures

It is important to note that, while the above illustrations have focussed on applications that provide confidentiality of messages, public-key cryptography can also be used to provide authentication of messages. One could consider first applying a secret function to some message m, to generate output commonly referred to as a *digital signature*, σ. Afterwards, others can use the matching public function to validate that the secret function was applied correctly, and σ is actually a valid signature for m. As only the holder of the secret function would be able to correctly apply the secret function to generate the signature, this can be used to prove the authenticity of a message. This, too, is proposed in Diffie and Hellman's paper [2].

Similar to the way described above, these functions (typically referred to as the signing function and the validation function) are parametrised by a private and a public key. Most traditional systems, such as RSA and ECC, can be used for both encryption and signing, using the same keys and functions. This is typically done by using the private decryption function to produce a signature, and the encryption function as a validation function. This symmetry does not always work, however – in particular, the scheme that will be described in Chapter 2 is a scheme that can only be used to create and verify signatures, and not for encryption and decryption. Similarly, encryption schemes cannot always be used to create verifiable signatures.

Digital signatures have become a very important primitive in the modern-day digital infrastructure. Besides providing attribution (i.e., being able to prove authorship of a signed message), they are also used extensively to guarantee message (or data) integrity. This enables trusted distribution of software, for example. For an adversary, it is impossible to tamper with signed messages without being detected, as a change in the message would invalidate the signature. In order to create a valid signature without possession of the secret key, an attacker would have to *forge* it. The strength of a signature scheme is strongly related to the effort it costs for an attacker to forge a valid signature. This can be further narrowed down by defining the type of forgery more precisely: is anyone able to create a signature for any message (universal forgery), does that require a prior signature on some chosen message (selective forgery), or can a signature only be forged for some unknown, random message that cannot be chosen (existential forgery)?

While confidentiality is typically not relevant when signing data, it is important that the signatures, besides being unforgeable and fast to create, add only minimal overhead to the message. Especially when transmitting small amounts of data, a large signature can be a serious problem. This will be a recurring theme throughout the rest of this thesis.

1.3 One-way functions

Essentially, one-way functions are functions that are efficient (also: *easy*, *cheap*) to compute, but infeasible to invert. This means that computing $f(x)$ for a large number of values of x should be easy to do, but given any resulting value y, solving $y = f(x)$ should not be feasible. Note that this property is different from non-invertible functions (i.e., non-bijective functions) in mathematics; at least one[1] inverse (or "pre-image") should exist, but it should be difficult to find. Additionally, it should not be feasible to find an *alternative* pre-image x' such that $x \neq x'$, but $f(x) = f(x')$. For applications in cryptography, one-way functions become stronger and more useful as the ratio between the computation effort in each direction grows.

Whether or not true one-way functions actually exist is an open problem, but there are functions that seem to exhibit the desired behaviour (until the contrary is proven). An example of such a one-way function is prime factorisation; given a composite number n (e.g. $n = 91$), it is relatively difficult to efficiently find its prime factors $p_1 \dots p_k$ such that their product is n (in this case, $p_1 = 7$ and $p_2 = 13$), but multiplying prime factors to compute some composite n is comparatively easy ($7 \cdot 13 = 91$).

An important contribution of [2] lies in the idea to use these one-way functions as the building blocks for asymmetric cryptography. All of the asymmetric-key systems mentioned in the previous section share this design. Their robustness is fundamentally coupled with the difficulty to invert the one-way function(s) they build upon.

[1] Depending on the nature of f, there could be infinitely many hard-to-find inverses for any given y.

1.3.1 Cryptographic hash functions

Cryptographic hash functions are a special subset of one-way functions. They combine the aforementioned properties of one-way functions with general-purpose hash functions.

Hash functions are one of the building blocks of a wide range of data structures in computer science. They are popularly assumed to be functions that take arbitrary input (notably: of arbitrary size) and map it to a limited domain of values, but the term can be used in a more general way to describe functions that map data to some table key. This is especially useful for looking up or indexing unsorted data objects [8], and data structures based on these functions are natively available in most modern programming languages. For now, we assume hash functions to be functions that project arbitrary data onto a finite (but significantly large) set of keys. After this section, 'hash function' and 'cryptographic hash function' will be used interchangeably to refer to the latter.

Cryptographic hash functions are one of the fundamental primitives in modern cryptography. They map arbitrary chunks of data to fixed-sized values (*digests*), while also adhering to the following properties (depending on the application, though, not all of these are used). Assume H to be a cryptographic hash function.

Pre-image resistance (sometimes known as one-wayness[2]) Solving $y = H(x)$ for x should be infeasible, for a given y.

Second pre-image resistance Given x_1, it should be infeasible to find x_2 such that $H(x_1) = H(x_2)$, for $x_1 \neq x_2$.

Collision resistance It should be infeasible to find any x_1 and x_2 such that $H(x_1) = H(x_2)$ with $x_1 \neq x_2$.

As we have seen above, the first two properties stem directly from one-way functions. The latter property is strictly stronger than second pre-image resistance, and often (but not always: see Section 2.2.4) a crucial extension.

1.4 Lamport signatures

In [9], Lamport proposes a signature scheme using one-way functions in a very general way; in contrast to the abovementioned schemes (RSA, DSA, ElGamal, elliptic curves), which rely on specific one-way functions such as prime factorisation or modular exponentiation, this scheme can be build upon any function from a broad set of one-way functions. This makes it especially interesting to examine in the context of this thesis and in the light of so-called post-quantum cryptography, where not all one-way functions hold their ground (see Section 1.6).

Lamport introduces his signature scheme as an improvement on Rabin's "Digitalized Signatures" [10], providing improvements that counter some of the fundamental drawbacks in Rabin's scheme (which largely come down to no longer requiring additional private information for signature verification). In addition to the fact that

[2]Note that this can lead to confusion, as one-way functions are typically defined to also satisfy second pre-image resistance and not just one-wayness.

the Lamport scheme can be configured using various one-way functions, it is also interesting because of another property: the Lamport scheme is a one-time signature (OTS) scheme. This means that each pair of public and private 'keys' can only be used to produce a single signature – using them twice would compromise unforgeability. This concept (and the reason why re-use would break the scheme) should become more clear when examining how the Lamport signature scheme works in some detail. Note that the description below is not intended to be as complete and general as in Lamport's paper, but it should suffice as an illustration of the basic principle.

Assume some one-way function $f : K \to V$ where K and V are (sufficiently large) sets of random elements. Then select n pairs of random values $(k_{i_0}, k_{i_1}) \in K$ for $i \in \{1, \ldots, n\}$, and compute $f(k_{i_j}) = v_{i_j}$ for all these values, in order to produce the sequence $(v_{1_0}, v_{1_1}, \ldots, v_{n-1_0}, v_{n-1_1}, v_{n_0}, v_{n_1})$. Lamport refers to this sequence as α, so we shall do the same. This sequence is the de facto 'public key', and should be made widely available, while the original values k_{i_j} need to be kept secret until they are used to produce a signature.

Now assume a binary message M of fixed length[3] n. Note that such a 'message' could be the output of a cryptographic hash function, as explained in Section 1.3.1. Let us refer to the individual bits of M as m_i for $i \in \{1, \ldots, n\}$. Now, in order to produce a signature for this message, we go back to the values k_{i_j}. For each bit m_i of M, we include $k_{i_{m_i}}$ in the signature; that is, we include k_{i_0} if m_i is 0 and k_{i_1} if m_i is 1. The resulting signature is a list of values that, when put through the one-way function f, match half the values in sequence α. Observe that, as the values k_{i_j} were secret and cannot be derived from v_{i_j}, only the original creator of α would be able to reveal the correct values. In order to prevent forgery, however, he or she would need to destroy the remaining unused values k_{i_j}.

How to verify the signature follows quite plainly from the above description. Anyone in possession of α (which is public information, spread as widely as possible) can apply the one-way function f to each of the revealed values k_{i_j} in order to derive the values v_{i_j}, and then confirm that they match the values $v_{i_{m_i}}$ listed in α.

The system described above comes with significant drawbacks, however, limiting its practical use. Next to the prominent usability issue that is inherent to a one-time signature scheme (i.e., only being able to use a certain public α to sign one message), Lamport signatures are often impractical because of their size. In order to prevent others from iterating over all possible values of $k \in K$ and solve $f(k) = v_{i_j}$ for any one of the values v_{i_j}, it is necessary to choose these values k sufficiently large. As each bit of the message requires the publishing of a value k, however, the signature becomes as large as n times the length of k. In its turn, the allowed input length, n, should also be sufficiently large to make it possible to sign a wide range of messages (or, when signing the output of a hash, to prevent collisions). Additionally, each α is of significant size as well, and a new α needs to be stored publicly and indefinitely for each signed message. This causes the scheme to also have a large storage footprint.

[3] In Rabin's paper [10], binary sequences of fixed length are used as input after processing an arbitrary message with an 'encoding function', and Lamport abstracts away from this even further by introducing an additional function that maps messages to a set containing fixed-sized elements. While this works in the general case, in practice fixed-sized binary data is often the relevant instance. More importantly, such values suffice for the purpose of this illustration.

1.4.1 The Winternitz OTS improvement

To reduce the size of the signature, as well as the required storage size for the public value α, Winternitz improved upon the Lamport scheme. Merkle introduces this improvement in a section of [11] – the same example values will be used here.

Instead of signing just a single bit, we can sign a number of bits at a time by applying the function f repeatedly. In order to sign four bits, one first publishes $v = f^{16}(k)$ for some random secret k (where f^{16} indicates for f to be applied sixteen times in succession, using the output of the previous iteration as input for the next). Then, when signing the bits, f is applied repeatedly according to the value of the bits: if we were to sign 1001, we would include $v' = f^9(k)$ in the signature. This results in only having to include one value for each group of four bits, drastically reducing the signature size. Verification is then done by applying another $16 - 9 = 7$ iterations to this value and comparing the resulting $f^7(f^9(k))$ to $v = f^{16}(k)$.

Note that while this method incurs extra computational costs, f is especially designed to be easy to compute. We now have a useful trade-off that allows us to exchange computation time for memory usage, configured by varying the number of bits signed per chain of function applications. This setting is typically referred to as the Winternitz parameter $w \in \mathbb{N}$ [12]. Casual observation quickly reveals that, as w increases linearly, the time required to generate keys, produce a signature or verify an existing signature increases exponentially.

In [11], Merkle makes note of a problem with the above scheme: anyone can easily compute $f^{10}(k)$ using $f^9(k)$. This allows attackers to forge a signature for 1010. He alludes to a solution by including a checksum, however, and this is one of the aspects that is fixed in practical applications of Winternitz chains [12]. Notably, using such a checksum also allows for one less application of f to compute the public key: there is no need to be one step ahead of the highest possible exponent, as the checksum ensures that an adversary would still be unable to forge a signature, even with this extra knowledge. This is used in the application of WOTS+ in SPHINCS in Section 2.4, where the chain only continues for $w - 1$ iterations.

1.5 Merkle trees

In his 1979 patent [13] and in a chapter of 'Advances in Cryptology '89' [11] (note that the latter was written in 1979 as well, but only published a decade later), Merkle introduces an authentication structure which he refers to as an 'authentication tree'. Since then, these structures have come to be known as 'Merkle trees', or simply 'hash trees'. The idea is similar in spirit to the Winternitz improvement described above, but further reduces the large storage requirement of public keys.

Consider a one-time signature scheme, such as the Lamport scheme described above. As was also mentioned above, a major downside of such a scheme is the fact that for each signature, a new pair of 'keys' needs to be generated, the public part of which needs to be stored somewhere accessible. This requires large public-facing resources. Merkle trees resolve this by grouping multiple public keys together in a tree structure, as follows.

Begin by generating $N = 2^n$ (for some $n \in \mathbb{N}$) pairs of OTS keys. Let f be some one-way hashing function, and let α_i be the public key belonging to OTS key pair i. Now, rather than immediately publishing the public keys, we compute one common digest by constructing a binary tree.

For each α_i, we compute $h_j = f(\alpha_i)$ with $j = 2^{(n-1)} + i$ and place each h_j on a leaf node. The value of a parent node is then computed by applying f to the concatenation of the values of its child nodes. We can recursively repeat this until reaching the root of the tree at level n, at which point we will have performed $2^n - 1$ applications of f. See Figure 1.1 for an example where $n = 4$. In this figure, $\|$ signifies concatenation and we use h_x for the hash digests attached to the nodes. After computing the entire tree, only h_1 needs to be made publicly accessible up front.

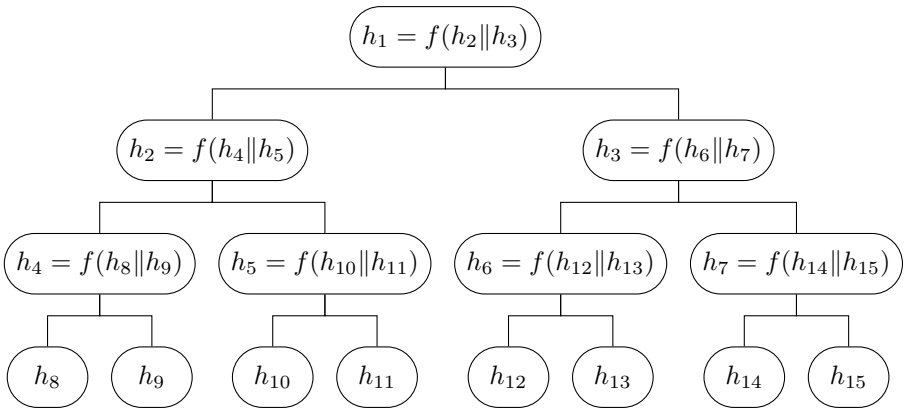

Figure 1.1: A hash-tree of depth 4

Having generated a number of keys and published the root node to authenticate them, one can now sign a message using one of keys from the leaf nodes. As these are still one-time signature keys, care should be taken to only use each leaf node once. This introduces a balance between generating a large tree up front, or creating new trees over time as keys run out. Smaller trees can be easier and lighter to work with throughout the scheme, but frequent renewal can be a serious usability hassle. In general, such a scheme is said to be *stateful*: a state needs to be maintained that describes which keys have already been used. The effect of this is discussed in more detail in Section 2.1.

After producing an OTS signature using one of the leaf node keys, the signer needs to provide sufficient information for the verifier to recreate the relevant parts of the tree. This is done by tracing the *authentication path* from a leaf node to the root node. For all siblings of nodes on this path, we include the node values in the signature. Let us take α_5 as an example. This public key results in digest h_{12}, of which the authentication path is shaded in Figure 1.2 – the greyed out nodes are the relevant siblings.

Note that, in addition to the 'regular' signature of the OTS scheme (which we will refer to as σ_i) and the authentication path, the public key that was used also needs to be included in the signature, as well as the position of the relevant leaf node in the tree. Normally, the public key would be available in publicly accessible storage, but this is precisely what the use of authentication trees has remedied. All in all, the signer in the above example would need to transmit $(\sigma_5, \alpha_5, 5, h_{13}, h_7, h_2)$.

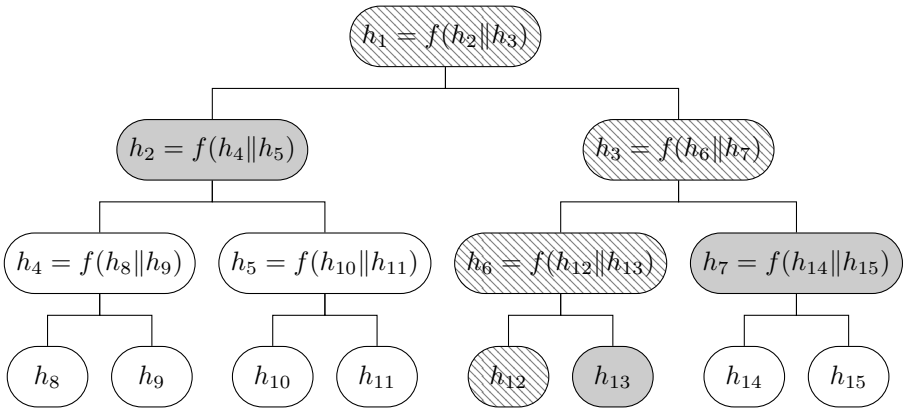

Figure 1.2: Nodes relevant to the authentication path of h_{12}

As mentioned above, another party can verify the signature by recreating the authentication path. First one computes the one-way digest of the OTS public key included in the signature. For the above example, that would be $f(\alpha_5) = h_{12}$. Now, using the sibling nodes included in the signature, all values along the authentication path can be reconstructed: $h_6 = f(h_{12} \| h_{13})$, $h_3 = f(h_6 \| h_7)$ and finally $h_1 = f(h_2 \| h_3)$. If this matches the h_1 that is publicly accessible, the public key α_5 was successfully authenticated. Now the signature of the OTS scheme can be verified in the usual manner (as both α_5 and σ_5 are available).

After proposing the authentication-tree algorithm, Merkle mentions that the current method of authenticating a public key is quite expensive in terms of the required computation, as the entire tree needs to be computed in order to produce the nodes along the authentication path [11]. Especially when large trees are used in order to prevent having to renew the public root node frequently, this can become costly. He goes on to suggest that, when performing multiple signatures sequentially, it can be advantageous to store parts of the authentication path – especially when authenticating multiple public keys *with consecutive indices*, many of these values can be re-used. Additionally, in order to reduce storage for the signing party, Merkle notes that the private keys of the OTS scheme from which the leaf nodes are generated only need to appear random, and can be generated using a pseudorandom number generator. These improvements will resurface in later chapters.

1.6 Post-quantum cryptography

In the previous sections, we have been examining cryptographic primitives first defined several decades ago. In this section, we will see why they may become much more relevant in the time to come. Note that while post-quantum cryptography (and cryptography in general) is much broader, the main focus of this section will again be on public-key cryptography.

1.6.1 What breaks

Over the past years, significant progress has been made in realising a quantum computer, up to the point where it has become not entirely unrealistic to expect a practical implementation within the near future [14]. With quantum computers, a number of algorithms that provide much more efficient solutions to long-standing mathematical problems can suddenly be implemented. However, as we have briefly seen in Sections 1.1 and 1.3, public-key cryptography relies on the fact that some mathematical problems are in fact difficult to solve. As it turns out, especially number-theoretical problems lend themselves to efficient solutions on quantum computers – precisely the type of problems that many of the popular cryptographic algorithms use.

Many of the algorithms currently in use (such as the Diffie-Hellman key exchange and elliptic curve cryptography) rely on the hardness of the discrete logarithm problem (i.e., it is hard to solve $y = g^x$ for an integer x where y and g are elements in a finite group, as long as the values are sufficiently large). Where this problem requires non-polynomial time on a traditional computer, a quantum computer can solve it in polynomial time using Shor's algorithm [15]. The same applies to the problem of prime factorisation (relied upon by RSA, among others), which also serves as a 'hard problem'. In fact, Dattani and Bryans show that 56153 has been factored using a currently available quantum computer [16] (unknowingly, by Xi et al [17], using a different algorithm). While this is still an extremely small prime compared to the ones used in practice, it foreshadows further progress in this direction.

Furthermore, Grover's algorithm [18] significantly reduces the effort required to break a much wider ranger of systems by effectively speeding up searching through a key space. This includes breaking symmetric-key systems and finding pre-images to hash functions. The result of this speed-up is not as dramatic as the ones described above, however. The main principle behind this algorithm is still essentially a brute-force approach, but a quantum computer can reduce the computation needed to find an item of n bits long to only $2^{\frac{1}{2}n}$ operations (instead of the expected 2^n). Essentially, the effort to search a random search space with N items now grows sub-linearly as N increases: instead, it grows only as fast as $\sqrt{N}$. This can be compensated for by doubling the length of the keys used [19], though, as a key of length $2n$ would again require 2^n computations.

As of today, nothing appears to have *really* been broken because of quantum computers in practice yet. As far as the scientific community is aware, no keys have been cracked and no data has been compromised. In the first chapter of [20], though, Bernstein warns that now is the time to start developing systems that will be able to remain secure in a post-quantum era. While there are schemes available that seem to hold their ground against the quantum computer, he illustrates that it takes time for these to be transformed to actual systems that can be used in practice. Additionally, we need to consider the fact that when the current schemes do break, that will put already-encrypted information at risk. As long as the ciphertext is stored somewhere, information that we currently deem well-protected could be compromised. The transition to post-quantum cryptography cannot be instantaneous, but needs to be prepared long before attacks using quantum computers become feasible.

1.6.2 What does not break

Fortunately, not all is lost. In [20], various authors describe a number of (classes of) public-key cryptosystems that do not succumb to Shor's and Grover's algorithms. The first chapter introduces some of these systems – the rest of the book discusses these in more detail. Although the focus of this thesis is on hash-based schemes, the others will be briefly considered here as well. It is important to realise that, even though these schemes are presented as 'post-quantum', all of them do in fact date back to before the turn of the century, and research has been ongoing.

While each of the systems discussed below has potential to become one of the post-quantum standards for public-key cryptography, none of them is flawless. This is a further confirmation of the need for research in this field. By continuing to develop each of these alternative solutions, a suite[4] of practical systems may be available on time.

Hash-based signature schemes

Hash-based signature schemes, such as the Lamport signature scheme described in Section 1.4 and SPHINCS, discussed in Chapter 2, typically rely solely on (the security properties of) the underlying cryptographic hash function. Their construction makes it possible to base the scheme on any arbitrary hash function. This is a result from the fact that the schemes do not rely on some number-theoretical property. This flexibility leads to an important efficiency consideration: if a given platform is more suitable for certain (classes of) hash functions, for example because of hardware support, it is trivial to change the scheme such that it uses precisely the hash function that is practical to use.

Hash-based schemes can definitely be considered to be one of the more conservative choices when it comes to post-quantum cryptography. Hash functions have been extensively studied and attacked in the context of current-day cryptography, and have seen numerous improvements – they have become one of the fundamental building blocks of cryptography. While this does not guarantee the security of any particular scheme built around them, the simplicity instils confidence. More importantly, Rompel showed that the existence of hash functions is a necessary precondition for the existence of *any* secure signature scheme [21]. While individual hash functions may degrade or break altogether under specific attacks, this result guarantees that hash-based signature schemes in general must be among the most solid constructions.

The downside of these kind of schemes, however, is the fact that they tend to produce rather large signatures. Even though the size is still well within reason (consider that SPHINCS produces 41KB signatures) for many applications, this can form a definite problem when space is limited.

[4] All of the systems listed here are either best suitable for encryption or as a signature scheme. Unlike with pre-quantum systems like RSA or ECC, it is unlikely that there will be one system that can fit both uses.

Error-correcting codes

The second class of schemes to discuss here is based on error-correcting codes. In general, these schemes rely on introducing errors dependent on some public key into a codeword, which only someone in possession of the matching private key is able to remove. Using such a private key, the receiver can remove a predefined number of t errors in an efficient manner. An implementation of this idea was introduced by McEliece [22] around the same time as Lamport's scheme, but is also currently not being used in practice. The main reason for this appears to be its large key size.

The security of the system appears to be solid, however, as it has only required a few minor tweaks since its original design in 1978. In a post-quantum world, the McEliece system is a good candidate for public-key encryption. While its relatively large key size makes it a hard sell for some applications, the authors of the relevant chapter in [20] argue for it based on its well-understood and strong security properties, as well as the fact that it is very efficient to compute. Moreover, the McEliece system allows for a conveniently configurable trade-off between security and computational speed.

Lattice-based cryptography

The use of lattice-based constructions in cryptography is a more recent development, dating back to the late nineties. In general terms, these schemes are based on problems relating to vectors in n-dimensional spaces. A common example is a problem where, given a lattice (i.e., a set of points in a vector space with a certain periodic property), one must find the shortest possible vector in that space. Even approximating a solution has not been done in polynomial time.

When used for cryptography, some lattice-based constructions are fairly efficient and practical to use. The more practical kind, however, lacks firm security reductions, and constructions based on problems with very strong security reductions tend to have usability problems (again, most notably large key sizes). From the first category, the NTRU system is able to beat RSA and ECC in terms of speed [23], while the 'learning with errors' problem that typically forms the basis for schemes in the second category has been proven to have strict hardness properties.

Multivariate quadratic equations

The last class to discuss here is multivariate cryptography. The core problem that lies at the basis of these schemes is related to solving quadratic polynomials over a finite field; computing the value of a polynomial for a given input is easy, but there is no known way to invert this efficiently. While some instances benefit from very short signatures and even shorter public keys (especially compared to some of the other systems listed here), several of these have been broken over the past few years. One reason for that is the fact that the problem is so broad that there are many very distinct possible instances, some of which exhibit unforeseen weaknesses.

The short signatures and public keys are a critical advantage over hash-based signature schemes, however, forming the main competition in the field of post-quantum signature schemes. Similar to the situation described above for lattice-based schemes,

the seemingly more robust systems (such as UOV schemes [24]) tend to have much larger key sizes, countering their practical use.

1.6.3 The PQCRYPTO project

As the quantum computer is becoming more of a certainty and less of a vague futuristic concept with each passing month, the interest in cryptographic schemes that will be able to survive is quickly picking up speed. Although not everybody is convinced, more and more research groups around the world are including post-quantum cryptography in their focus. If anything, the fact that the European Union and other large governmental institutions are investing in research projects concerning the development of quantum computers should legitimise putting post-quantum cryptography on the agenda as well.

In March of this year the PQCRYPTO project [25] has kicked off, starting extensive research into practical post-quantum cryptography. As the cryptology group from the Technische Universiteit Eindhoven is spearheading the project, it was covered widely in various Dutch national media in late April and early May of this year. Other participating partners (from Belgium, Denmark, Germany, the Netherlands, Israel and Taiwan) are the Bundesdruckerei, Danmarks Tekniske Universiteit, the Institut National de Recherche en Informatique et en Automatique, the Katholieke Universiteit Leuven, NXP Semiconductors, the Ruhr-Universität Bochum, the Radboud Universiteit Nijmegen, the Technische Universität Darmstadt, the University of Haifa, and Academia Sinica. The PQCRYPTO project, set to last for the coming three years, is funded by the European Union as part of Horizon 2020, the "Framework Programme for Research and Innovation". Through Horizon 2020, the EU aims to stimulate novel research and create a competitive atmosphere.

The purpose of this project is to provide efficient implementations of post-quantum cryptographic algorithms that are ready to be used in practice. In order to achieve this, the project aims to come to a consensus on a small set of public-key schemes that are secure in a post-quantum era and provide high usability. To maximise practical applicability and the likelihood of actual use, one of the goals of the project is to actually standardise this set of algorithms.

As we have seen in the previous section, post-quantum cryptography comes in various flavours, each with their own advantages and disadvantages. We saw that it will be unlikely that there will be a one-size-fits-all for both encryption and signing. However, even this split may not be sufficiently fine-grained. Cryptography is used in a colourful variety of different contexts, and each of these has their own requirements when it comes to properties like size and speed, but also side-channel resistance and defences against widely different attacker models. In order to provide optimal solutions for each of these different platforms, it may very well be necessary to use a number of different schemes in different settings. The PQCRYPTO project addresses these scenarios in three distinct work packages: WP1, aimed at cryptography for small devices; WP2, providing solutions for the use of cryptography on the internet at large; and WP3, for 'the cloud'.

The concerns of WP1 are perhaps the most intuitive to grasp. Small devices are very limited in both memory and CPU capabilities. The registers are typically small, and there is often little hardware support for intricate instructions or instruction sets (such

as vectorisation) from which cryptography can often benefit. Especially memory constraints tend to result in absolute constraints – while time consumption is often a gradual scale, memory can be a real deal-breaker that is hard to recover from. These devices are an important platform to consider, now that the electronics industry has started including microprocessors in all sorts of consumer and enterprise products.

With WP2's focus on cryptography on the internet, it is dealing with data in transit, potentially across many different machines. This introduces concerns when it comes to the size of signatures, ciphertext and keys. The low latency requirements and the frequently high number of parallel connections make computational speed an important metric as well. As the servers are generally powerful machines, the situation can become highly asymmetrical when considering a broad spectrum of client devices.

WP3 deals with 'the cloud'. In the context of PQCRYPTO, this refers to 'data at rest' at a third party storage provider. This scenario brings up considerations in terms of the security of the data as well as the users' privacy, but these constraints are less clear-cut. Cloud storage is typically used to quickly synchronise or search through data – maintaining these features while still being able to rely on future-proof cryptography is a difficult open problem. Time and memory constraints, however, are of lesser importance here.

It will be no surprise that the work presented in this thesis fits snugly into the first category. The implementation of SPHINCS on the Cortex M3, discussed in Chapter 3, provides a proof of concept that the algorithm is feasible and effective on small devices. Suitability for a platform from the first work package is significant, as some of the practical constraints in terms of time and memory consumption are especially tight. These properties carry over to the other work packages. This is not generally applicable for all constraints, though, as the attacker model and required security level vary wildly.

SPHINCS

This chapter will serve to explain the SPHINCS signature scheme proposed in [26]. The chapter is structured in a top-down fashion: after explaining SPHINCS' raison d'être, an overview of the entire scheme is provided and the individual components are discussed. Note that the precise mathematical details are abstracted from where this serves readability and is not required for general understanding. Refer to [26] for the strict definitions.

2.1 Eliminate the state

When using Merkle trees on top of an OTS (as described in Section 1.5), the user should be very careful not to use a certain key twice. This is a fundamental problem, as this implies that, in addition to storing the secret key (i.e., the seed that produces all OTS keys on the leaf nodes) and any cached authentication path nodes, one would have to store some indicator to keep track of the used leaf nodes[1]: the state. While this is not a problem in some applications, key management can quickly become an issue. Consider using the same key on multiple machines (for example on redundant servers) or creating backups – the state of the keys would need to be constantly kept in sync. This makes such a signature scheme highly impractical and incompatible with many of today's systems.

Already in 1986, Goldreich establishes this problem and proposes a solution [27]: create a tree of such depth that, when randomly choosing an OTS key pair for each signature, the chance of accidentally reusing a certain OTS key pair becomes insignificantly small. This way there is no need to keep track of the already used OTS keys. The obvious problem here is actually creating such a tree, but Goldreich is able to avoid this. By not simply hashing nodes together (as we saw in the Merkle tree structure) but instead attaching an OTS key pair to each node and using that to sign the child nodes, it is never necessary to compute the entire tree. This requires that the OTS keys of the nodes along the path from a random leaf to the root can be deterministically generated out of order, but that can easily be done using a random seed and the node index (typically by concatenating the two and applying a hash function).

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

[1] In practice, this could be just the number of messages signed so far, as Merkle showed that using the keys sequentially is preferable [11]

signature. This creates a new hurdle for widespread use, as it makes for tremendously large signatures. In particular, it would be impractical to require signatures significantly larger than the actual data being signed. Consider web pages served to many clients over HTTPS, where adding a signature of several megabytes would drastically hinder performance. Even when taking into account the Winternitz improvement (see Section 1.4.1) using a large w, the signatures remain prohibitively large.

2.2 Overview

As discussed above, two main problems with hash-based signature schemes are the need to maintain a state and the size of the signatures. This has prevented hash-based solutions from being a drop-in replacement for the signature schemes that are currently in use. SPHINCS solves this by combining the approach of Goldreich with traditional Merkle trees in a nested construction. This results in a stateless scheme with signatures of 41KB and private and public keys of 1KB each [26]. At "hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU", it is sufficiently fast for many practical applications.

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

The trees τ_i are binary hash trees, only slightly varying from the original Merkle tree concept. These are elaborated upon in Section 2.2.4. Each of their nodes $\nu_{i,j}$ for $j \in \{1, \ldots, h/d - 1\}$ contains a digest of its child nodes, while the leaf nodes on layer h/d each contain the key of an OTS. For now, let us assume that we have some hash function H that generates these digests. As with Merkle trees, the digest at the root of the tree can be used to authenticate the entire structure by constructing authentication paths.

All these sub-trees are then chained together as in Goldreich's system. Using the OTS keys in the leaf nodes $\nu_{i,h/d}$ of the trees τ_i, the root nodes of the trees τ_{i+1} are signed; a new sub-tree is chained to each of the leaf nodes. See Figure 2.1 for a close-up of this construction. Notes labelled H contain a hash of their child nodes, while OTS nodes include a key pair to authenticate their child node.

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

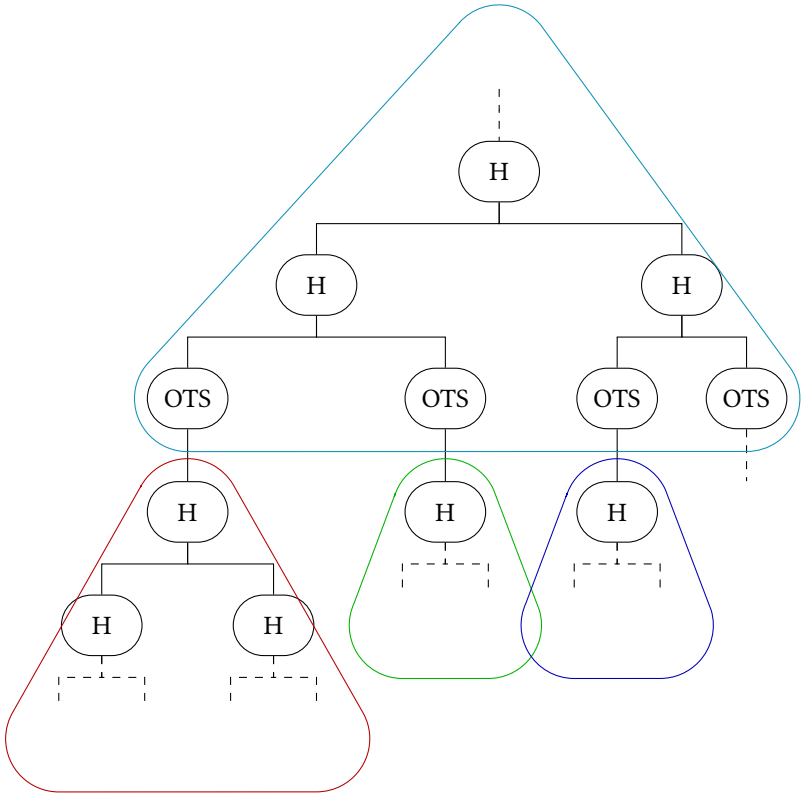

Figure 2.1: Linking sub-trees together

The above describes the basic outline of SPHINCS. This still far from a working algorithm, though, as we have assumed a number of black boxes: some hash function H to use in the sub-trees, an OTS to use between sub-trees and an FTS to sign the actual message at the bottom of the hypertree. In the following sections, we will gradually collect the missing pieces.

2.2.1 Key generation

Because of the Goldreich structure, key generation for SPHINCS is a fairly cheap operation. We start by selecting some random values $SK_1 \in \{0,1\}^n$ and $SK_2 \in \{0,1\}^n$. The first of these is used for key generation, but in the next section we will see why it is convenient to have a long-term random number available for signing as well. Additionally, we generate a set Q of random bitmasks, also from $\{0,1\}^n$,to be used in various places. These masks are used in all the hash trees (in Section 2.2.4), as well as in the OTS and FTS – for now, let us merely acknowledge their existence. Thus $SK = (SK_1, SK_2, Q)$.

In order to generate the public key, we must at least generate the one tree in τ_1: the tree at the top of the structure. This requires generating the OTS keys along the bottom of this tree. Note that these keys need to be generated deterministically; using the address and SK_1 as input to some one-way function we can derive a seed for

this key. Then, a binary hash tree can be built on the public keys of the OTS key pairs, and the root node of this tree is part of the SPHINCS public key: PK_1. As the bitmasks are also needed for verification, they must also be included with the public key: $PK = (PK_1, Q)$.

It is worth noting that, while SK_1, SK_2 and PK_1 are all only n bits in size, Q is significantly larger. The bitmasks thus account for the largest part of the keys. In the SPHINCS-256 configuration discussed in Section 2.5, for instance, PK_1 accounts for only 32 bytes while Q is responsible for $2 \cdot 32 \cdot 16 = 1024$ bytes. In general, the number of bitmasks is determined by the part of the scheme that requires the largest number of them – the FTS, the OTS or the hash trees.

2.2.2 Signing

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

After selecting a particular FTS key pair, it needs to be generated (based on a seed derived from its location and SK_1) and then used to sign D to produce signature σ_{FTS}. Together with the message-specific randomness generated above and the index idx of the selected key pair, this signature forms the first part of the SPHINCS signature. We will refer to the SPHINCS signature as Σ. As SPHINCS uses an OTS and an FTS for which the public keys can be derived from their respective signatures (as we will in Section 2.3 and 2.4), there is no need to include it.

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

Altogether, the SPHINCS signature Σ now contains the message-specific randomness R, the index of the selected FTS key pair, the FTS signature σ_{FTS} and d pairs of

OTS signatures and nodes along the authentication path $(\sigma_{OTS,i}, Auth_i)$. Everything combined, $\Sigma = (idx, R, \sigma_{FTS}, (\sigma_{OTS,1}, Auth_1), \ldots, (\sigma_{OTS,d}, Auth_d))$

2.2.3 Signature verification

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

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

2.2.4 Hash trees

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

In [28], XORing with bitmasks is introduced as part of a linear hashing scheme, and it is employed in [29] in order to construct binary hash trees that do not require the underlying hash function to be collision resistant. Instead, second-preimage resistance is sufficient to attain unforgeability. This hash-tree construction is the one described above, and is used in SPHINCS.

2.3 The FTS: HORST

At the bottom of the hypertree, SPHINCS relies on an few-time signature scheme. For this, a variation of an OTS called HORS [30] is used. The configuration of HORS, called HORST, consists of two parameters that affect the security level, signature size and key sizes: t and k, where $t = 2^\tau$.

As mentioned in the overview of SPHINCS, the key gets seeded based on SK_1 and the location of a particular HORST instance in the hypertree. This seed is expanded to t secret keys sk_i for $i \in \{1, \ldots, t\}$, which are then hashed to create the HORS public keys pk_i. The HORST variation then proceeds to build a hash tree on top, of which the root node is the actual public key pk. For this tree, SPHINCS also makes use of bitmasks as described in Section 2.2.1.

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

Verification is quite similar: first, the revealed pieces of sk are hashed, and the message is split into k parts. These parts are then interpreted as integers and used to place the pieces of sk on the appropriate leafs. Using the nodes supplied in the signature, the path to the root node can then be computed. The verification algorithm then outputs this root as the public key – comparing it to the actual public key will reveal if the signature was valid.

In the original setting, HORS is proposed as a one-time signature scheme. However, by choosing τ such that $t = 2^\tau$ becomes sufficiently large in relation to k, only a small part of the secret key is revealed for each signature, and the chance of a successful forgery after obtaining just a few signatures diminishes. Note that this does require that an adversary cannot control the message hash for which a signature is obtained. In the original HORS scheme the public key would increase linearly with t, but the HORST variation only incurs logarithmic growth in the length of the authentication path, and thus the number of required bitmasks. This makes it feasible to configure HORST as an FTS in SPHINCS.

2.4 The OTS: WOTS+

For the OTS that links the sub-trees together, SPHINCS uses WOTS+. This variation of the Winternitz OTS is proposed in [31], designed to reduce the signature size even further than other WOTS-based schemes. As in WOTS, the Winternitz parameter $w \in \mathbb{N}$ is used to configure the efficiency trade-off. Likewise, one then derives ℓ (consisting of ℓ_1 and ℓ_2) from this parameter and the security setting n as follows.

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

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

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

In order to guarantee deterministic signatures here as well, WOTS+ key pairs are also seeded using SK_1 and their location in the tree. This seed is then expanded to a secret key of ℓ pieces, $sk = (sk_1, \ldots, sk_\ell)$. Generating the key is very similar to traditional WOTS; we simply apply the chaining function c for a total of $w - 1$ times to each part

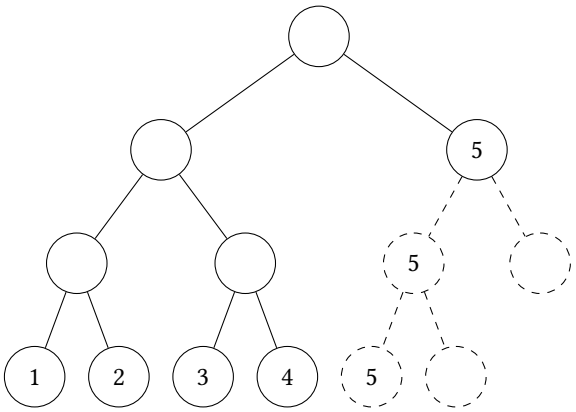

Figure 2.2: An L-Tree with five leaf nodes

of sk to obtain $(pk_1, \ldots, pk_\ell)$. As the reader might be expecting by now, we proceed by building a hash tree on top of these public key parts. These trees make up the third level of trees in the hypertree. However, ℓ is not necessarily a power of two – in fact, as w and n (and thus ℓ_1) typically are a power of two, ℓ is not. This requires the use of a slightly different tree construction: the L-Tree [29]. The structure is entirely identical to binary hash trees, except for the rightmost nodes. Whenever the number of nodes on the current layer is odd, the rightmost node is lifted up to the next layer instead. See Figure 2.2 for an example with five nodes. The root of this tree is the public key pk.

In order to sign a message M, we first interpret it as an integer in binary, and then express it in base w. This effectively distributes M into a sequence of values that can be at most $w-1$. Note how there will be at most ℓ_1 values, as per the construction of ℓ_1. We write $M = (M_1, \ldots, M_{\ell_1})$. Furthermore, a checksum needs to be computed to address the problem mentioned in the last paragraph of Section 1.4.1: $C = \Sigma_{i=1}^{\ell_1}(w - 1 - M_i)$, which is also expressed in base w: $C = (C_1, \ldots, C_{\ell_2})$. The lists M and C are then chained together to form $B = (b_1, \ldots, b_\ell) = M \| C$. These values in B are then used as lengths for the Winternitz chains, producing the signature $(\sigma_1, \ldots, \sigma_\ell) = (c^{b_1}(sk_1), \ldots, c^{b_\ell}(sk_\ell))$.

Verifying a WOTS+ signature is very similar to the regular WOTS scheme, except that one needs to take special care to use the correct bitmasks when applying the chaining function[2]. Let us define the function v to account for this. Like for the original chaining function, the base case is $v^0(x) = x$.

$$v^i(x) = F(v^{i-1}(x) \oplus r_{w-i})$$

We then compute B in the same way as in the signing procedure, and then compute the public key parts $(pk_1, \ldots, pk_\ell) = (v^{w-1-b_1}(\sigma_1), \ldots, v^{w-1-b_\ell}(\sigma_\ell))$. As we did during the key generation step, the last step that remains is computing an L-Tree over these pieces. The root of this tree is then outputted as the result of the verification

[2]Note that this approach differs slightly from the one presented in [31]. In the original definition this is solved by supplying the appropriate set of bitmasks as an argument to the chaining function.

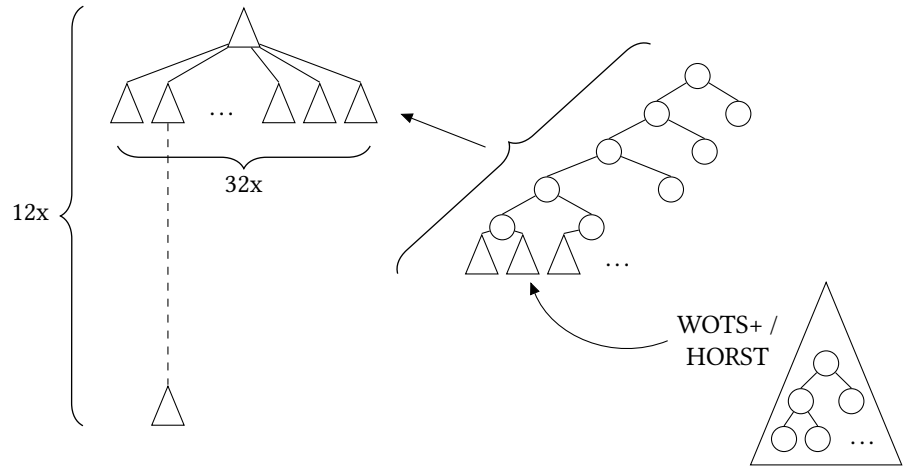

Figure 2.3: The complete hypertree structure

algorithm. Like in HORST, this can then be compared to the actual public key to verify that the signature was valid.

2.5 SPHINCS-256 and ChaCha

The above sections illustrate the general outline of SPHINCS. However, in order to actually use the scheme in practice, there are a numerous parameters and functions that still need to be filled in. In [26], the authors propose a specific configuration: SPHINCS-256. This is the scheme that will be used in Chapter 3. In this section, it will briefly be discussed.

The name of SPHINCS-256 already reveals the value chosen for n: 256. This is the length of the hashes used in the numerous trees, as well as the length of SK_1 and SK_2. The length of the message hash m is 512 bits, however, dictating the product of k and $\log(t)$. The number of leaf nodes on a HORST tree is set to $t = 2^{16}$, requiring $k = 32$ nodes to be revealed for each signature. The Winternitz parameter w is 16, resulting in $\ell = 67$. Finally, the dimensions of the hypertree are set at $h = 60$ and $d = 12$; the complete tree is made up of 60 layers, spread out over 12 layers of subtrees containing 5 hash-layers each. See Figure 2.3 for an overview of the combined tree construction.

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

a permutation-based construction using the permutation from the ChaCha stream cipher family [32].

ChaCha operates on 512-bit input, so in order to use it for F, we need to pad the input with some constant. In SPHINCS, the authors chose the 32-byte ASCII string $C =$"expand 32-byte to 64-byte state!". As the output of ChaCha is also 512 bits, half of the output is truncated to construct F. H is constructed similarly. Let $Chop(M, i)$ be a function that truncates M to i bits, M_i strings of 256 bits and O a string of 256 zero-bits, then:

$$F(M_1) = Chop(\pi_{ChaCha}(M_1 \| C), 256)$$
$$H(M_1 \| M_2) = Chop(\pi_{ChaCha}(\pi_{ChaCha}(M_1 \| C) \oplus (M_2 \| O)), 256)$$

Now only a few minor pieces of the puzzle remain. Creating the message-specific random value R is done by calling BLAKE-512($SK_2 \| M$), and BLAKE-512 is used once more to create the digest D. In order to derive the secret keys for the HORST and WOTS key pairs based on their location and SK_1, the BLAKE-256 function is used as a pseudo-random function. It is worth noting that the BLAKE hash functions [33] also rely on the ChaCha permutation, and in order to expand the seed to a HORST or WOTS key, ChaCha12 is used as well. This way, the SPHINCS-256 construction largely depends on this one permutation (and the way it is used, e.g. in BLAKE) rather than on a number of different hash functions, any of which may prove to be insecure somewhere down the road.

SPHINCS on the Cortex M3

In this chapter, an implementation of the SPHINCS algorithm on a limited platform will be discussed. This forms the main contribution of this thesis. This implementation conforms to the parameter choices made in SPHINCS-256. The focus of this chapter will be on conveying the higher-level design choices, but for some aspects more detailed descriptions of the implementation will be provided. The initial goal of this work was to get the algorithm working on the limited platform, although speed optimisations have been a consideration and influence throughout. The signing times are still somewhat impractical for most applications, and improving this is considered relevant future work (see Section 4.2).

The implementation used here is based on the implementation of SPHINCS-256 presented in [26]. As this implementation targets a much more powerful processor (specifically, an Intel Haswell) than the one used for this work, several components had to be replaced. For one, the Haswell implementation relies on vector instructions provided through the AVX2 instruction set. One can imagine that this is unavailable on a platform as small as the one discussed in the next section. To fill these gaps, implementations of BLAKE-256 and BLAKE-512 [33] by Aumasson as well as an implementation of the ChaCha permutation [32] by Bernstein were retrieved through SUPERCOP [34].

3.1 The platform

The platform of choice for this implementation is the Cortex M3, and development was carried out using the STM32L100C discovery board. The Cortex M3 is a 32-bit microprocessor with sixteen 32-bit registers, thirteen of which are available for general purpose computation (leaving three for the stack pointer, link register and program counter). This processor is commonly found in microcontrollers used in the automotive industry, small industrial systems and (wireless) sensors. As opposed to ARMv6-M found among other Cortex M-series processors, this processor supports the ARMv7-M architecture.

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

it is still possible to produce SPHINCS signatures within these limits in the next sections.

3.2 Trees, treehash and HORST

As we have discussed in Chapter 2, SPHINCS consists of a number of distinct parts, with the HORST trees and WOTS/hash trees as the two most prominent subdivisions. While the memory usage is typically large at the base of a tree, it fans in again as we work our way towards the root. As each tree is stacked onto the one below, it is not necessary to ever store more than one tree in memory at a time before proceeding on to the next.

For the WOTS/hash trees, this is not an immediate problem. Each tree contains only $h/d = 5$ layers of hashing, resulting in a total of $2^6 - 1 = 63$ nodes. Producing a single WOTS signature impacts the available memory for only 67 bytes, which do not need to be maintained afterwards. The WOTS public key is slightly larger at $32 \cdot 67 = 2144$ bytes, but can be quickly reduced to a 32 byte root node by applying the L-trees we have seen in Figure 2.2. The order of performing these steps was not important in the reference implementation, as memory was not a concern, but is crucial for this memory-constrained version; it is important to digest a public key immediately after producing it rather than first computing the public key for each leaf node and then digesting all of them sequentially. This is an easy change without any other impact. This results in 32 leaf nodes of 32 bytes each, which can then be hashed up to compute the root node of the tree. Once the tree has been built, it is trivial to walk from a specific leaf node to the root to collect the nodes along the authentication path. In the end, the constant memory consumption throughout this step of the algorithm comes down to at most one public key of 2144 bytes, and the entire tree at $63 \cdot 32 = 2016$ bytes. Together with some storage space for the seed of the secret key, this makes 4192 bytes, spaciously fitting in the available 16KB. As only the root node needs to be carried on into the next computation, all of this can be freed again before proceeding to the next level.

HORST, on the other hand, is an entirely different beast. With $t = 2^{16}$ and $k = 32$, the trees contain 131071 nodes spread over 16 layers of hashing, making these trees much higher than the hash trees on top of the WOTS signatures. This means that the method of first building the entire tree and then extracting the authentication path, as described above, is not feasible. At 32 bytes per node, the nodes alone would require 4MB of storage.

There is no need to store the entire tree, though, as only a very specific set of nodes is relevant for the signature: the nodes along the authentication path, as well as the root node. As we do require the root node in order to authenticate the tree, there is definitely no escaping having to *compute* the entire tree.

In [26], the authors mention that RAM usage and code size was not one of the concerns – the implementation was optimized for speed on a platform where memory was available in abundance. They remark that, if saving memory is a concern, the treehash algorithm can be used while storing the nodes along the authentication path 'on the fly'.

Treehash is yet another contribution from Merkle's "A Certified Digital Signature" paper [11]. It has since been used in various forms as the basis of tree traversal algorithms. A common approach is expressed by the pseudo-code (based on [35]) in Algorithm 1, and discussed below.

Algorithm 1 One round of the treehash algorithm

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

The core idea is to maintain a collection of the currently relevant nodes: the 'heads' of the different branches. As new nodes are added, these branches are gradually grown to completion, and merged when needed. Any nodes that sit deeper in the tree can safely be forgotten (for the purpose of finding the root node), as each node is only required once to generate its parent. Each round of treehash consists of introducing the next leaf node and updating the heads of the branches until no more new nodes can be computed. For half the leaves (i.e., the 'left neighbours'), their introduction does not allow for the computation of any new parent nodes, while a quarter of the leaves allow us to compute one parent node, etcetera. See Figure 3.1 for an elaborate example.

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

Besides computing the root node of a HORST tree, however, we are also interested in the nodes along the authentication paths from the leafs used to produce the signature to the top of the tree. This was the main reason for storing the entire tree in memory in the first place.

Intuitively, a way to resolve this is by somehow recognising the nodes that need to be included with the signature while performing the treehash rounds. Going through the tree without actually computing the node values is much easier, so we can simply trace the authentication paths from leaf to root and compile a list of the 'interesting' nodes (as well as their required position in the signature[1]). Afterwards, whenever we compute a node while performing treehash, we would then only need to check

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

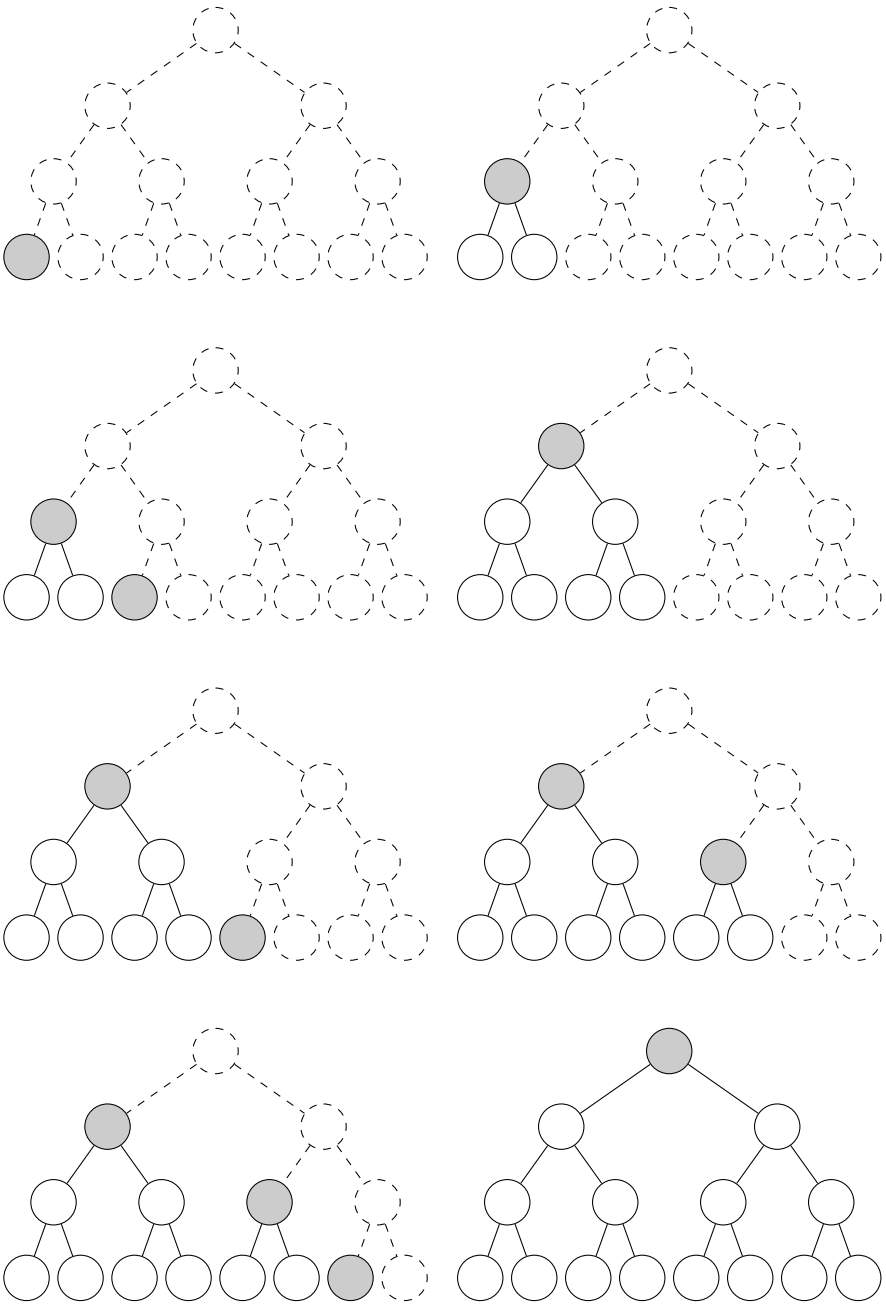

Figure 3.1: Treehash: the ‘relevant nodes’ are marked grey. Each round introduces a leaf node, computes possible parents and updates the set.

to see if it was in the list. If so, it would be stored at the appropriate location in the signature.

While it works in theory, this approach is fairly inefficient in practice. The reason for this is the fact that the computation of every node requires checking the list of interesting nodes. This adds considerable overhead, and only yields a result in a very small number of cases: consider that the tree consists of 131071 nodes, but only 320 nodes[2] need to be found this way, not even accounting for the inevitable duplicates. Going through the list for all nodes only to find out that most of them are not included seems like a waste of cycles.

Instead, when preprocessing the tree to find the interesting nodes, we can immediately compute in which round of treehash they will be produced. This will allow us to recognise rounds that produce interesting nodes, and only evaluate the signature position of this limited set of nodes. When computing the indices of the interesting nodes, their treehash round numbers can be found using the iterative procedure described in pseudo-code in Algorithm 2, below. Here, the input idx is the index of a leaf node at the base of an authentication path.

Algorithm 2 Computing treehash round numbers

Require: idx
Output: treehash round numbers of interesting nodes

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

After marking certain rounds as interesting, it is only a small addition to also keep track of which specific nodes within each round are relevant. To achieve this, a sequence of bits is stored for each round, where each bit indicates whether the node should be included in the signature. These bits are passed along to the treehash rounds, allowing for fast boolean checks.

[2] One would expect to require $32 \cdot 16 = 512$ nodes, as each of the 32 authentication paths results in 16 neighbouring nodes. However, for efficiency reasons, the HORST signature always includes layer 6 in its entirety (so as to prevent including all the duplicate nodes in the top layers) and truncates the authentication paths after 10 nodes. For generality and easier understanding, this optimisation is ignored when not dealing with exact node counts or the precise construction of the HORST signature.

The above approach improves on naive lookups in a list of nodes: the list of rounds is somewhat more limited than the list of nodes, and the number of lookups is smaller. This improvement is small, however, and the process is still quite costly. Moreover, this suffers the same symptom as the per-node lookup: only a very limited number of these rounds will actually be in the list, rendering the majority of these checks unnecessary. This suggests that there is room for further optimisation.

In contrast to the node indices, it is important to realise that the treehash rounds are processed sequentially. We can benefit from this by sorting the list of round numbers and maintaining an iteration pointer that starts at the smallest element in the list. Now, whenever we process a round, we simply check if the round number is equal to the value pointed to by the iterator. If this is not the case, we know that the round is not in the list at all, and we can safely dismiss any nodes computed during this round as uninteresting. If the round number is equal, the indicator bits (discussed above) are retrieved and the respective nodes are stored. Additionally, we increment the indicator to point at the next round number in the list. This way, the leaf nodes and the list of interesting round numbers are iterated over synchronously, greatly diminishing containment checking overhead.

3.3 Signature and key data

The careful reader might have noticed that something is missing from the description above. While computing the SPHINCS signature, we appear not to be actually *storing* the signature. This is correct: as observed before, it is impossible to store the 41KB signature in the mere 16KB that is available. Furthermore, the expanded private keys used in HORST would require $t \cdot n = 2^{16} \cdot 32 = 2097152$ bytes (roughly 2MB) of storage throughout the computation. In this section, we will see how these limits can be overcome by streaming the required data instead.

3.3.1 Streaming the signature output

Instead of trying to squeeze the signature into too little memory, it is streamed out of the board over serial output throughout the computation. For many applications, this is not much different from receiving the entire signature all at once after the entire computation has finished, so this does not immediately pose any usability concerns.

As we have seen in Section 2.2.2, the SPHINCS signature consists of a number of different components. Recall that $\Sigma = (idx, R, \sigma_H, (\sigma_{W,1}, Auth_1), \ldots, (\sigma_{W,d}, Auth_d))$. By now, we can safely use the subscript H for the HORST FTS signature and W for the WOTS OTS signatures for brevity.

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

The HORST signature σ_H is a bit more complicated. It consists of k pairs of secret keys belonging to leaf nodes, and sequences of nodes along the path from each of these leaf nodes towards the top of the tree. As remarked in footnote 2 on page 28, all nodes on layer 6 are always included, so the last 6 nodes of these sequences are truncated. The issue here is the fact that the node sequences are not produced one at a time, but are each grown in an interleaved fashion as more and more of the tree is computed. When storing the hash values in a signature in memory, this does not pose a problem – each node value can be inserted in the right place. When we are streaming the output, however, we cannot go back and insert a node value. Instead, the node values will have to be tagged with what should have been their location in the signature, and rearranged accordingly on the receiving end. For each 32-byte node value, this adds an overhead of two bytes. While this may seem significant, in the end it results in an increase of 832 bytes (640 for the authentication path nodes, 128 for layer 6 and 64 bytes for the secret keys), which is acceptable considering that the entire SPHINCS signature is 41KB.

3.3.2 Streaming the expanded key material

Another aspect of the SPHINCS signing procedure that requires significant memory is the set of secret keys used in HORST. As discussed in Section 2.3, the seed SK_1 and the location in the hypertree are combined to produce t secret keys, one for each leaf node. As mentioned above, the 2MB that this would produce cannot possible be stored in memory. Instead, we can (once again) rely on the fact that treehash rounds consume the leaf nodes sequentially, and only generate the leaf node values when they are required.

It is possible to store the intermediate state of the ChaCha12 stream cipher (initially seeding it in the regular fashion for HORST), and make it perform another iteration based on this stored state whenever more key data is required. This allows the generation of leaf node values on the fly. As ChaCha12 produces an output of 512 bits, every other leaf node requires a new chunk of output to be generated.

3.4 Performance

The previous sections show some of the adjustments required to be able to produce SPHINCS signatures on a platform with only 16KB of RAM. Besides memory usage, however, time is also a relevant metric to consider. While it is not always essential to be able to produce or verify a signature in the absolute minimum possible time, for nearly all applications it is the case that consuming less time is strictly better. In fact, for most applications, the time consumption very much determines the usability. Consider producing or verifying signatures on smart cards, for example when paying for public transport or when using an ATM. When the required cryptographic operations would take more than a few seconds, the system would quickly become frustrating to use.

The implementation presented in this work was optimised for memory consumption, and time consumption was not given priority. Still, it is relevant to briefly discuss

the performance in terms of both time and cycle count. Recall (Section 3.1) that the processor is clocked at 32MHz.

Producing one signature on the Cortex M3 currently takes just over 52 seconds, or 1 681 333 801 cycles, while generating a key pair can be done in 73 986 826 cycles. This is in line with expected performance. For comparison, the speed-optimised implementation for Intel Haswell presented in [26] is able to process "hundreds of messages per second". This benchmark was performed on a quad-core processor at 3.5GHz, parallelising the operations inside ChaCha and making extensive use of vectorisation to compute multiple instances of the functions F and H in parallel as well. The authors report a cycle count of 47 466 005 for one signature and 3 051 562 for key generation. The factor of 35 can be realistically accounted for by a combination of the vector-based 8-way parallelism and more powerful instructions per cycle.

In SPHINCS, practically every part of the signing computation comes down to applying the ChaCha permutation. It is the fundamental building block for both WOTS+ and HORST, and is also used to construct the hypertree around these schemes. On the Cortex M3, the used implementation of ChaCha requires 1615 cycles to perform one permutation. Let us briefly examine the different parts of the algorithm where ChaCha is used, in order to come to an understanding of the scale.

Recall that $t = 2^{16}$. In order to generate a HORST key and produce a signature, ChaCha is used $\frac{1}{2} \cdot t = 32768$ times to expand the seed and generate the secret keys, as the permutation outputs 512 bits and the keys are 256 bits each. These secret keys are then hashed using F to construct the leaf nodes at the cost of another $t = 65536$ permutations. Subsequently, treehash is used to hash together t leaf nodes, at a cost of two ChaCha permutations per parent node (see the function H in Section 2.5), resulting in another $2 \cdot (t - 1) = 131070$ permutations. All in all, this results in 229374 calls for one HORST signature.

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

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

Combining the cost of HORST the WOTS+ trees above it, we come to a grand total of $229374 + 456444 = 685818$ permutations. At 1615 cycles each, this would account for $685818 \cdot 1615 = 1\,107\,596\,070$ cycles of the complete signing operation, or roughly 66%.

Conclusions

In this thesis we have seen how hash functions can be used to create signature schemes that do not succumb to attacks using a quantum computer. We have examined an example of such a scheme, SPHINCS, and discussed the practical implications of its use. Because of the way it is able to provide stateless signing, it has the potential to serve as a drop-in replacement for the signature schemes in use today.

We have seen that it is feasible to implement this scheme on limited platforms where memory is a scarce resource. The resulting time consumption is not yet ideal, but is in line with expectations given the limitations of the platform. Moreover, this shows that the SPHINCS-256 scheme is usable on a wide range of platforms without any insurmountable limitations. While time consumption is something that can be optimised gradually, memory consumption typically has a hard lower bound. This thesis shows that this lower bound is sufficiently low, making SPHINCS a very promising candidate as a future-proof signing scheme.

4.1 Related work

When it comes to hash-based signature schemes, the work of Buchmann, Dahmen and Hülsing on XMSS should be noted [36]. Their (albeit stateful) scheme produces smaller signatures and is provably secure. It was further improved and implemented on a 16-bit smart card in [37], making it the first hash-based signature scheme to be practical to use on such a small device.

In [38], a fast implementation of the Merkle signature scheme on an AVR XMEGA is presented, as well an extensive side-channel analysis. By introducing a new algorithm for the computation of authentication paths in a Merkle tree, side-channel leakage during treehash is significantly reduced. The potential for efficient implementations of the Merkle signature scheme on small devices was demonstrated in [39].

4.2 Future work

As we have seen in Section 3.4, it still takes a significant amount of time to produce a SPHINCS signature. The reason for this appears to be twofold, and improvements can be sought in the same two directions. First and foremost, the implementation of the ChaCha permutation used in this project was not specifically optimised for this

platform. By carefully implementing the permutation in the supported ARMv7-M assembly, it should be possible to decrease the cycle count significantly. As this permutation is the very essence of SPHINCS in terms of cycle consumption, the effect of a reduction of only a limited number of cycles could already be quite significant. This brings us to the second potential for improvement. Currently, ChaCha accounts for 66% of the consumed cycles. While this is already the majority, it may be possible to push this percentage even higher, given that the rest of the algorithm is largely concerned with administration around these permutations. Initial tests reveal that these cycles are distributed fairly evenly, however, suggesting improving this might be easier said than done.

One aspect of SPHINCS that was breezed over perhaps too lightly in Section 3.3 is the fact that SPHINCS should be able to operate on messages M of arbitrary size. The current implementation is constrained by memory, however, as it does not support streaming the message to the device. Notably, the message is used twice: once to generate the message-dependent randomness R, and once to produce the digest D, where the digest also requires part of R as input. This makes it impossible to stream the message only once and derive both values, considering that R is used early on in the computation of D. Potential solutions include streaming the message twice, relying on a different source of randomness or signing a hash of the message, although the latter has implications for the security reduction.

The current implementation supports key generation and signing of messages, but does not yet do verification. Adding this should not prove to present much difficulty, as this operation (as is usually the case with hash-based schemes) makes use of the familiar constructions that are also needed when generating keys and producing a signature. Still, this is a required step towards building an implementation of SPHINCS-256 that is useful in practice. One consideration for development here is that it requires a bidirectional data connection with the board in order to test. When producing a signature, testing can be done unidirectionally by including a fixed message in the flashed code. This is not the case for verification, though, as the signature that needs to be verified is too large to fit in memory all at once.

An additional optimisation was alluded to in Section 1.5, for scenarios where multiple signatures are required sequentially: storing parts of the authentication paths in order to reduce recomputation. The exact feasibility and benefit of the trade-off of memory versus time consumption is not exactly clear, however, as the large hypertree as well as the random leaf node selection may seriously limit the chance of overlap, but it may prove worthwhile to look into.

Bibliography

[1] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, 9: 5–38, 1883. http://www.petitcolas.net/fabien/kerckhoffs/crypto_militaire_1.pdf. 2

[2] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. www-ee.stanford.edu/~hellman/publications/24.pdf. 3, 4

[3] Ronald L. Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2): 120–126, 1978. https://people.csail.mit.edu/rivest/Rsapaper.pdf. 3

[4] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. 3

[5] David W. Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668 https://www.google.com/patents/US5231668. 3

[6] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/. 3

[7] Victor Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO'85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1986. 3

[8] Donald E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Addison Wesley, 2nd edition, 1998. 5

[9] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International, Palo Alto, 1979. 5

[10] Michael O. Rabin. Digitalized signatures. In Richard A. DeMillo, Richard J. Lipton, David P. Dobkin, and Anita K. Jones, editors, *Foundations of secure computation*, volume 78, pages 155–166. Academic Press, 1978. 5, 6

[11] Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1990. www.merkle.com/papers/Certified1979.pdf. 7, 9, 15, 26

[12] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the Winternitz one-time signature scheme. In Abderrahmane Nitaj and David Pointcheval, editors, *Progress in Cryptology – AFRICACRYPT 2011*, volume 6737 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2011. https://huelsing.files.wordpress.com/2013/05/winterleak.pdf. 7

[13] Ralph Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569 https://www.google.com/patents/US4309569. 7

[14] IBM. IBM scientists achieve critical steps to building first practical quantum computer. Press release, 2015. ibm.com/press/us/en/pressrelease/46725.wss, retrieved 2015-05-14. 10

[15] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. http://www.arxiv.org/abs/quant-ph/9508027. 10

[16] Nikesh S. Dattani and Nathaniel Bryans. Quantum factorization of 56153 with only 4 qubits. *arXiv 1411.6758v3*, 2014. http://arxiv.org/pdf/1411.6758v3. 10

[17] Nanyang Xu, Jing Zhu, Dawei Lu, Xianyi Zhou, Xinhua Peng, and Jiangfeng Du. Quantum factorization of 143 on a dipolar-coupling nuclear magnetic resonance system. *Physical review letters*, 108(13):130501, 2012. 10

[18] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996. 10

[19] Daniel J. Bernstein. Grover vs. McEliece. In Nicolas Sendrier, editor, *Post-Quantum Cryptography*, volume 6061 of *Lecture Notes in Computer Science*, pages 73–80. Springer, 2010. Document ID: e2bbcdd82c3e967c7e3487dc945f3e87 cr.yp.to/papers.html#grovercode. 10

[20] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer, 2009. 10, 11, 12

[21] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 387–394. ACM, 1990. http://www.cs.princeton.edu/courses/archive/spr08/cos598D/Rompel.pdf. 11

[22] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978. http://tmo.jpl.nasa.gov/progress_report2/42-44/44N.PDF. 12

[23] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In Josef Pieprzyk, editor, *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2010. http://www.securityinnovation.com/uploads/ntru_speed_benchmark_research.pdf. 12

[24] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999. http://www.goubin.fr/papers/OILLONG.pdf. 13

[25] PQCRYPTO. *EU Horizon 2020, ICT-645622*, 2015. Project partners: Technische Universiteit Eindhoven, Bundesdruckerei, Danmarks Tekniske Universiteit, Institut National de Recherche en Informatique et en Automatique, Katholieke Universiteit Leuven, NXP Semiconductors, Ruhr-Universität Bochum, Radboud Universiteit Nijmegen, Technische Universität Darmstadt, University of Haifa, Academia Sinica. http://pqcrypto.eu.org. 13

[26] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Marc Fischlin and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015. Document ID: 5c2820cfddf4e259cc7ea1eda384c9f9, http://cryptojedi.org/papers/#sphincs. 15, 16, 22, 24, 25, 31

[27] Oded Goldreich. Two remarks concerning the goldwasser-micali-rivest signature scheme. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110. Springer, 1987. http://theory.csail.mit.edu/ftp-data/pub/people/oded/gmr.ps. 15

[28] Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 1997. https://cseweb.ucsd.edu/~mihir/papers/tcr-hash.pdf. 19

[29] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2008. https://www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/DOTV08.pdf. 19, 21

[30] 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*, volume 5299 of *Lecture Notes in Computer Science*, pages 144–153. Springer, 2002. http://www.cs.bu.edu/~reyzin/papers/one-time-sigs.pdf. 19

[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*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013. https://huelsing.files.wordpress.com/2013/05/wotsspr.pdf. 20, 21

[32] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. Document ID: 4027b5256e17b9796842e6d0f68b0b5e, http://cr.yp.to/papers.html#chacha. 23, 24

[33] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. SHA-3 proposal BLAKE. *Submission to NIST*, 2008. https://131002.net/blake/blake.pdf. 23, 24

[34] Daniel J. Bernstein and Tanja Lange, editors. SUPERCOP. In *eBACS: ECRYPT*

benchmarking of cryptographic systems. http://bench.cr.yp.to/supercop.html, accessed 2015-02-08. 24

[35] Andreas Hülsing. *Practical Forward Secure Signatures using Minimal Security Assumptions*. PhD thesis, Technische Universität Darmstadt, 2013. http://tuprints.ulb.tu-darmstadt.de/3651/. 26

[36] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011. https://huelsing.files.wordpress.com/2013/05/mssgesamt.pdf. 32

[37] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward secure signatures on smart cards. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography – SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2013. https://huelsing.files.wordpress.com/2013/05/xmss-smart.pdf. 32

[38] Thomas Eisenbarth, Ingo Von Maurich, and Xin Ye. Faster hash-based signatures with bounded leakage. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2014. http://users.wpi.edu/~teisenbarth/pdf/SignatureswithBoundedLeakageSAC.pdf. 32

[39] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast hash-based signatures on constrained devices. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2008. https://www-old.cdc.informatik.tu-darmstadt.de/reports/reports/REDBP08.pdf. 32