BACHELOR

A theoretical comparison between SPHINCS+ and Gravity-SPHINCS

Boschman, W.M.

Award date:
2019

A theoretical comparison between SPHINCS$^+$ and Gravity-SPHINCS

Eindhoven University of Technology

Bachelor Thesis

Author:
W.M. Boschman, 0898103

Supervisor:
Dr. A. Hülsing

November 4, 2019

Abstract

To resist quantum cryptoanalysis, a stateless hash-based signature scheme has been constructed called SPHINCS. To improve SPHINCS's security and performance, two new signature schemes SPHINCS$^+$ and Gravity-SPHINCS were proposed as part of the NIST post-quantum cryptography competition in 2017. The aim of this report is to compare both SPHINCS$^+$ and Gravity-SPHINCS on their performance and security. As both schemes are based on their predecessor, SPHINCS, first a detailed explanation of SPHINCS is provided. We identified three different potential security improvements for SPHINCS: multi-target attack protection, the use of a verifiable index and prevention of colliding indices in HORST. SPHINCS$^+$ dealt with all these improvements by introducing tweakable hash functions, FORS and a different method to compute the message digest, whereas Gravity-SPHINCS only solved the unverifiable index and colliding indices issue by the introduction of PORST. However, Gravity-SPHINCS introduced a lot of options to reduce computation time, i.e. secret key caching, mask-less hashing and octopus authentication. Based on the comparison in this thesis, some additions and changes for SPHINCS$^+$ and Gravity-SPHINCS were identified that could improve both scheme's performance and security level. In future research, more insight for the optimization of the schemes could be provided by a more practical comparison, in which SPHINCS$^+$ and Gravity-SPHINCS are compared for different sets of parameters.

Contents

1 Introduction

It is thought that quantum computers will be available in the near future, putting most of the nowadays indispensable cryptosystems used for our modern communication in danger, for instance, RSA and DSA [1]. In order to find new algorithms that would be less susceptible to a quantum computer's attack than the current algorithms available, NIST (The National Institute of Standards and Technology) organized a competition for post-quantum cryptography in 2017. Among the submissions were SPHINCS$^+$ [2] and Gravity-SPHINCS [3]. These algorithms were based on SPHINCS, a stateless hash-based signature scheme, which was the first signature scheme to propose parameters to resist quantum cryptoanalysis [2]. SPHINCS$^+$ and Gravity-SPHINCS were constructed independently of each other and therefore have different approaches when it comes to improving SPHINCS 's security and performance.
Up till now, no direct comparison has been made between SPHINCS$^+$ and Gravity-SPHINCS, hence, it is unknown which algorithm would be the most attractive answer to the post-quantum cryptography challenge. The aim of the NIST competition was to design a signature scheme that is able to generate 2^{64} signatures, which all can be verified with the same public key and the signature scheme should be composed in such way that signing a message is done within a reasonable amount of time [4]. SPHINCS is not able to meet these requirements, since SPHINCS is only able to compute 2^{50} signatures, while still keeping a reasonable signing speed. It is possible for SPHINCS to reach 2^{64} signature, but then the scheme will run into performance issues.
The aim of this report is to compare the stateless hash-bases signature schemes SPHINCS$^+$ and Gravity-SPHINCS on their performance in a theoretical way. First of all, it is important to gain a better understanding of these schemes. Therefore, a closer look is taken at their common origin, SPHINCS, which is elaborately discussed in section 3. Explanation is provided by taking a look at the different components that SPHINCS consists of and it is discussed how key generation, signature formation and verification work. Section 4 discusses potential improvements in SPHINCS, as there are certain aspects of SPHINCS that can be used as an advantage for a possible attack. In section 5, it is explained what changes and additions have been made in SPHINCS$^+$ and Gravity-SPHINCS in comparison to SPHINCS. Last but not least, a comparison between these improvements is made in section 6. However, before discussing these advanced schemes it is essential to understand why signature schemes are important and what is required to construct a secure signature scheme. This is the focus of the next section.

2 Preliminary

Currently, a lot of cryptosystems are available. There exist systems that encrypt a message, so that only the receiver of the message is able to read it, but there are also systems that simply create a signature. SPHINCS$^+$ and Gravity-SPHINCS are both hash-based signature schemes. The hash-based part will be explained further on in this chapter. For now let us focus on what a signature scheme is and what it is used for. In general, a signature scheme allows the user to sign a message (document, contract, bill, etc). It follows the same principle as it would in real-life: a person writes a signature as a proof of identity or to provide evidence of deliberation and informed consent. A digital signature

helps with the authenticity of the digital message. Adding such a signature will give a certainty that the message indeed came from the sender. It provides a way to verify the identity of the sender and the integrity of the message.
A signature scheme consists of three steps, which are shown in Figure 1. The first step is called key generation, during which a key pair is generated. It starts with a randomly generated secret key. With use of this secret key a public key is created. The second step of the process is the signing algorithm. The message (document, contract, bill, etc) together with the secret key are used as input for the signing algorithm. This algorithm will give as output a signature. The last step is the verification of the signature. The verification algorithm uses the message, the public key and the signature and either accepts or rejects the input. If the verification algorithm accepts the input, then the signature is correct.

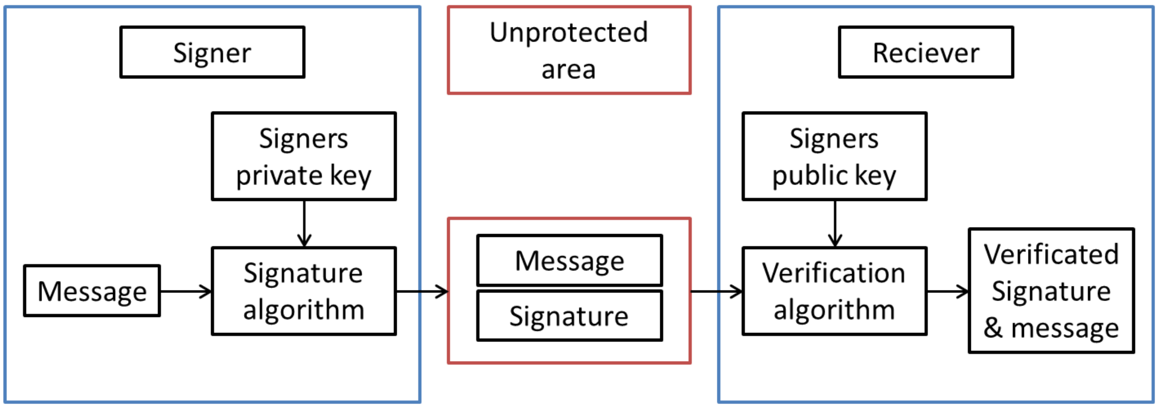

Figure 1: The signature process

The following example illustrates the importance of using a secure signature scheme. Imagine that Alice wants to send a bill to Bob, because Bob still owes Alice some money. On this bill Alice wrote her bank account details, so Bob knows where to send the money to. It is possible that a third-party, Eve, intercepts the bill. If Alice decided not to add a signature to the bill, then Eve is able to change Alice's bank account details into her own bank account details. When Bob receives the bill, he expects the bill to come from Alice. Since Alice did not add a signature to the bill, there is no verification available that the bill is actually from Alice. Thus Bob sends the money to the bank account that was included in the bill; resulting in Eve receiving the money, instead of Alice. Because Alice does not want this to happen, she adds a signature by means of a signature scheme. The bill with the bank account details acts as the message that is signed. If Bob was to receive this message now, he can actually verify that it came from Alice. Eve can still intercept the message and alter the bank account details. However, doing so will change the message, resulting in a signature that does not correspond to the message anymore. Through verification Bob can now tell that either the message, signature or public key does not correspond anymore. If Eve wishes to receive money from Bob, she can either remove the signature (which results in a message of which Bob cannot verify its sender

and therefore Bob may decide not to send money to the given bank account) or she can break the signature scheme that Alice used. If Eve breaks the signature scheme, she can copy Alice's signature and sign anything pretending to be Alice. To prevent this, it is of great importance to use a secure signature scheme.
There exist many different signature schemes and SPHINCS$^+$ and Gravity-SPHINCS are based on hash functions. So before going into detail about these schemes and their predecessor SPHINCS, it is important to explain what hash functions are.

2.1 Hash functions

Hash functions are functions that map a bit string of arbitrary length to a bit string of a fixed length. These functions are designed as one-way functions. This means that the function is easy to compute in one way, but computing the inverse is much more difficult. Besides the one-wayness property, hash functions should have a few more security properties.
First of all, the function has to be collision resistant. Consider a hash function H, then H is collision resistant if there exists no bit string $A \neq B$ that satisfies $H(A) = H(B)$. In other words, there can not exist two different bit strings that hash to the same bit string. Unfortunately, in case there is a hash function $\{H : \{0,1\}^{m(n)} \rightarrow \{0,1\}^n\}$ with more input than output, so $m(n) \geq n$, there is no way to avoid collision [5]. Consider a bit string with n bits that are either 0 or 1, so 2^n bit strings are possible. For an attacker to find a collision with a high probability, he only has to compute $2^{\frac{n}{2}}$ different hashes. This latter amount of different hashes can be derived in a similar way as the method used in the- birthday paradox, where the chances that two people in a group of 23 people have the same birthday is a little more than 50% [6].
The second security property is that the hash function should be preimage resistant. This means that for a given $H(A)$, with A unknown, it is difficult to find a distinct A' such that $H(A) = H(A')$
Last of all, the hash function has to be second-preimage resistant. This means that for a given random input A it should be difficult to find a second preimage $A' \neq A$ such that $H(A) = H(A')$ [5].
In conclusion, a hash function is believed to be secure if it is collision resistant, preimage resistant and second preimage resistant. Examples for currently used hash functions are SHAKE256 and SHA-256. Although the main part of a hash based signature scheme is based on these hash functions, this thesis will not focus so much on these functions, but rather focus on how the schemes work and on how something is signed.

2.2 Security

As illustrated in the Eve example in section 2, it is of great importance that a signature scheme is secure. In order to make a signature scheme secure, there should be no possibility for forgeries. In general, these forgeries can be categorized into four different types:

- *A total break.* An attacker knows the secret key.
- *Universal forgery.* An attacker is able to forge a signature for any given message.

- *Selective forgery.* An attacker is able to forge a signature on a message of their choice.
- *Existential forgery.* An attacker is able to forge a signature for an arbitrary message.

Note that these types of forgeries are listed from most difficult to obtain to least difficult to obtain: there is less information required to construct an existential forgery than to totally break the scheme. In order for an attacker to obtain this kind of information, the attacker has to perform an attack. There are two basic types of attacks:

- *Key-only attacks.* The attacker only knows the public key.
- *Message attacks.* The attacker knows some signatures corresponding to either known or chosen messages.

The message attacks are divided into four different types of message attacks. These are the following:

- *Known message attack.* The attacker knows a set of messages with their corresponding signatures, however these messages are not chosen by the attacker.
- *Generic chosen message attack.* The attacker is able to choose a set of messages and obtain their signatures. However the attacker has to choose the set of messages before seeing any signatures. The set of messages is also fixed and independent of the public key.
- *Directed chosen message attack.* The attacker can again choose a set of messages, but this time the public key is known and the set of messages is dependent on the public key. Also the attacker has to choose the set of messages before seeing any signatures.
- *Adaptive chosen message attack.* The attacker can again choose a set of messages. This time the attacker can be adaptive by changing the set of messages after seeing some signatures. The attacker can request signatures for messages depending on the public key and on previously obtained signatures.

The list is ordered from the least severe message attack to the most severe message attack. It is important that the signature scheme is secure enough against all these types of attack, so no valuable information is obtained by the attacker. An attack on a security system is successful if one of the aimed for forgeries is reached with a non-negligible probability and in probabilistic polynomial time. There are ways for the attacker to reach the same goal by trying every single possible option, also known as a brute force attack. Brute forcing will always break the signature scheme, it is however not defined as a successful attack, because it requires a lot of computation time [7].
In conclusion, in order to classify a signature scheme as secure, it should take a long time for an attacker to obtain valuable information, that can be used to construct a forgery.

3 SPHINCS

SPHINCS$^+$ and Gravity-SPHINCS are both signature schemes that can be used to sign multiple messages. They are both based on the structure of their predecessor SPHINCS. SPHINCS is quite a complex signature scheme, since it consists of multiple signature schemes. Each of the components of SPHINCS will be explained in subsection 3.2. In subsection 3.3 is described how all these components work together in the key generation, signing and verification processes. In order to get a better understanding of how these components function, we first have to take a look at one of the first hash-based signature schemes, Lamport.

3.1 Lamport

One of the very first hash-based signature schemes is the Lamport one-time signature (OTS) scheme. The scheme works in the following way: first, a secret key is randomly generated, that consists of $2m$ bit strings of length n. This secret key can be divided into $2m$ bit strings : $x_{1,0}, x_{1,1}, \ldots, x_{m,0}, x_{m,1}$, where each bit string has length n. To create the public key each bit string is hashed individually, using a hash function H, $H : \{0,1\}^n \to \{0,1\}^n$, such that:

$$H(x_{1,0}), H(x_{1,1}), \ldots, H(x_{m,0}), H(x_{m,1}) = y_{1,0}, y_{1,1} \ldots, y_{m,0}, y_{m,1}$$

Consider a message M you want to sign and divide this message up into m bits: $b_1, b_2, \ldots, b_m$. For each bit in $b_1, b_2, \ldots, b_m$ one of the corresponding secret key parts is chosen. Take b_1 for instance: if $b_1 = 0$, then $x_{1,0}$ is used, and if $b_1 = 1$, then $x_{1,1}$ is used. This way you end up with a signature σ consisting of secret key parts, e.g. $\sigma = x_{1,0}, x_{2,1} \ldots, x_{m,0}$. In order to verify that the message that was sent came from the person who signed it, the receiver has to check the signature. The receiver knows the following: the signature σ, the hash function H, the message M and the public key of the sender, $y_{1,0}, y_{1,1} \ldots, y_{m,0}, y_{m,1}$. In order to check the signature, the receiver also divides the message M into m bits, which will give him $b_1, b_2, \ldots, b_m$. Using these bits the receiver can check which parts of the public key were used, for instance b_1 was either 0 or 1 corresponding to $y_{1,0} = H(x_{1,0})$ or $y_{1,1} = H(x_{1,1})$ respectively. Doing this for each b_i of the message will give him a key of length m that contains the public key parts corresponding to the b_i. The receiver now hashes the individual parts of the signature and checks if the result matches the key that the receiver just computed [8].
However, as implied by the name of the scheme, the signature should only be used once. If a sender would use the same secret key more often, then anyone who intercepts these messages, Eve for instance, could figure out the full secret key based on the given signatures. Eve would then be able to sign messages pretending to be the sender. To prevent this from happening, a new secret key should be generated for every message you want to send, which means that for each message also a new public key is computed, resulting in a large number of public keys. Hence, this signature scheme could be easily improved if the same public key can be used for multiple secret keys and their corresponding signatures.

3.2 Components of SPHINCS

As stated in the introduction, SPHINCS is a stateless hash-based signature scheme that consists of multiple components [9]. In this section each of these components is explained and the last subsection we discuss how all these components interact in a full hypertree.

3.2.1 Merkle trees

Ralph Merkle created an algorithm that produces multiple signatures corresponding to a single public key. In order to do so, he used multiple OTS instances and created a tree out of this, which is also known as a hash tree or Merkle tree. The set-up is a full complete binary tree (see Figure 2), where the leaves of the tree correspond to the OTS public keys used. Note that the amount of signatures the Merkle tree can produce is 2^h, where h is the height of the tree.

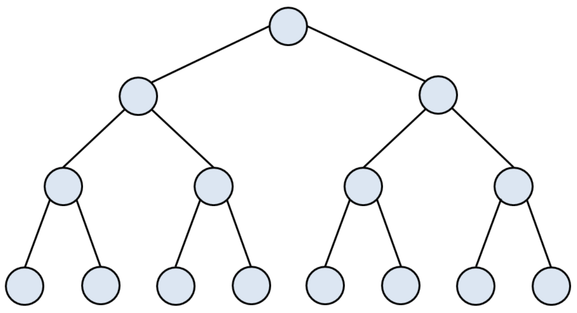

Figure 2: Full complete binary tree

In order to obtain the public key of the scheme, the root node of the tree has to be computed. Consider the full complete binary tree depicted in Figure 2: this Merkle tree has eight leaf nodes (the nodes depicted at the bottom layer). Each of these leaf nodes corresponds to a public key of a one-time signature. Consider $pk_1, \dots, pk_8$ as our OTS public keys and H as our hash function. To compute the nodes of the layer above the leaf node layer, two of the adjacent leaf nodes (starting from the left side) are hashed together to form the new node. In the example illustrated in Figure 2, the first node that will be computed is $pk_{1,2} = H(pk_1 \| pk_2)$. When we continue this for the other leaf nodes, then the layer above will look as follows: $pk_{1,2}, pk_{3,4}, pk_{5,6}, pk_{7,8}$. Continuing on to the next layer, again hashing together the previous two nodes, will gives us the following two nodes: $pk_{1,2,3,4}, pk_{5,6,7,8}$. Hashing these two nodes together will give us the root node of the tree, also known as the public key of the scheme. In this case the public key is $pk_{1,2,3,4,5,6,7,8}$
In order to generate a signature, the message M is signed using one of the one-time signatures in the Merkle tree. This requires the use of one of the OTS secret keys. Recall that these should only be used once, hence we have to keep track of the secret keys used by using an index i. For example, Alice wants to send message M to Bob. She uses the secret key sk_3 for this, so $i = 3$ (see Figure 3). Firstly, she creates a signature from

M using the OTS: we call this signature $\sigma(M)$. Bob receives the message M, signature $\sigma(M)$ and the public key $pk_{1,2,3,4,5,6,7,8}$ from Alice. With use of the message M, signature $\sigma(M)$ and the OTS verification algorithm, Bob is now able to compute pk_3. In order to verify the signature, Bob needs a way to compute the public key $pk_{1,2,3,4,5,6,7,8}$ with use of pk_3. To do so, Alice has to send the authentication path. The authentication path contains the adjacent nodes of the nodes that are used in the tree. Using the nodes contained in the authentication path, Bob can compute the root node of the tree, $pk_{1,2,3,4,5,6,7,8}$. The authentication path is included in the signature Alice has sent and is as follows: $\sigma = (i = 3, \sigma(M), pk_4, pk_{1,2}, pk_{5,6,7,8})$. This signature will allow Bob to recompute the public key in the following way: again he uses the verification algorithm of OTS to compute pk_3. After obtaining pk_3, Bob computes $H(pk_3 \| pk_4) = pk_{3,4}$, then $(pk_{1,2} \| pk_{3,4}) = pk_{1,2,3,4}$ and at last $H(pk_{1,2,3,4} \| pk_{5,6,7,8}) = pk_{1,2,3,4,5,6,7,8}$. Bob has now obtained a public key and can compare his result with the public key of Alice. If these public keys match, then Bob has verified that the signature is correct [10].

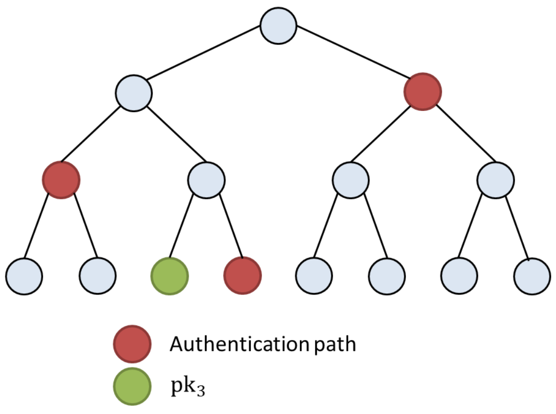

Figure 3: Authentication path

There exist a few variations of Merkle trees. In SPHINCS one of these variations has been incorporated in the scheme: SPHINCS makes use of extended Merkle trees (XMSS). XMSS uses Winternitz OTS (WOTS) as OTS [11]. In order to explain XMSS, we first have to discuss how WOTS works.

3.2.2 WOTS

WOTS works similar to the Lamport one-time signature scheme. However, in WOTS the secret key is hashed more often by a length-preserving hash function. This latter means that a function $H : \{0,1\}^n \rightarrow \{0,1\}^n$ will hash a bit string of length n to another bit string of length n. In addition, WOTS makes use of the Winternitz parameter $w \in \mathbb{N}$, with $w > 1$. This Winternitz parameter determines how many times a bit string is hashed to form the public key. Thus, for a larger value of w, it takes longer, or better said, more calls for the hash function are required, to compute the public key. For a message of length m, the length l of the secret key is computed as follows:

$$l_1 = \lceil \frac{m}{\log_2(w)} \rceil, \qquad l_2 = \lfloor \frac{\log_2(l_1(w-1))}{\log_2(w)} \rfloor + 1, \qquad l = l_1 + l_2$$

Each part of the secret key is a bit string of n bits and the full secret key is then $sk = (sk_1, \ldots, sk_l)$. So the total length of the secret key is $n \cdot l$. To generate the public key, each of the secret key parts, $sk_1, \ldots, sk_l$, is hashed $w-1$ times. So $pk_1 = H(\ldots H(H(sk_1)) \ldots) = H^{w-1}(sk_1)$. This gives the following public key:

$$pk = (pk_1, \ldots, pk_l) = (H^{w-1}(sk_1), \ldots, H^{w-1}(sk_l))$$

For the signing part of the algorithm, the message M is divided into l_1 bit strings: $M = (M_1, \ldots, M_{l_1})$. Each part has length $\frac{m}{l_1} = \frac{m}{\lceil \frac{m}{\log_2(w)} \rceil} = \log_2(w)$. Because each fragment of M has the length of $\log_2(w)$, the value of each fragment is between 0 and $w-1$. Next, we define a checksum $C = \sum_{i=1}^{l_1} w - 1 - M_i$, with $i \in 1, \ldots, l_1$. The maximum value of C is reached if $M_i = 0$ for all i: this means that $C \leq l_1(w-1)$. C is a value and corresponds to a bit string which can be divided into l_2 parts of length $\log_2(w)$. This gives us $C = (C_1, \ldots, C_{l_2})$. C and M can be used to define a new bit string B: $B = (B_1, \ldots, B_l) = (M_1, \ldots, M_{l_1}, C_1, \ldots, C_{l_2})$. Every $B = (B_1, \ldots, B_l)$ is a bit string of length $\log_2(w)$, which corresponds to a value between 0 and $w-1$. Using B we can compute our signature as:

$$\sigma = (\sigma_1, \ldots, \sigma_l) = (H^{B_1}(sk_1), \ldots, H^{B_l}(sk_l))$$

In order to verify the signature, each part of the signature has to be hashed a certain amount of times, so the total amount of times it is hashed adds up to $w-1$. The receiver knows the following: the message M, the signature σ, the public key and the Winternitz parameter w. For the verification of the signature, the receiver computes l_1, l_2 and l, using w and the length of the message m. Using l_1 and l_2, the receiver can determine $M = (M_1, \ldots, M_{l_1})$ and $C = (C_1, \ldots, C_{l_2})$, which allows him to find $B = (B_1, \ldots, B_l)$. The received signature is $\sigma = (\sigma_1, \ldots, \sigma_l) = (H^{B_1}(sk_1), \ldots, H^{B_l}(sk_l))$, so if the receiver would compute $H^{w-1-B_1}(\sigma_1), \ldots, H^{w-1-B_l}(\sigma_l)$, then this should equal the public key. Note that the signature parts $(\sigma_1, \ldots, \sigma_l)$ are hashed, this results in $sk_1, \ldots, sk_l$ being hashed $w-1$ times, giving us the public key [12]:

$$pk = (pk_1, \ldots, pk_l) = (H^{w-1}(sk_1), \ldots, H^{w-1}(sk_l))$$

WOTS+ is an extended version of this scheme, which works in a similar way as WOTS only here a bitmask is added to the iteration. A bitmask is a string of bits that is bitwise (OR, AND, XOR) added to another string of bits of the same length, to create a new bit string of the same length. In order to add these bitmasks, WOTS+ uses a set $\mathbf{Q} = (r_1, \ldots, r_j) \in \{0,1\}^{n \times j}$ as bitmasks. These bitmasks are used every time a secret key part is hashed. First a secret key part is operated through an gate (OR, XOR or AND) using a random bit string from $\mathbf{Q} = (r_1, \ldots, r_j)$, and then the new bit string is hashed. Since this occurs $w-1$ times for each secret key part, the bitmasks ensure that more randomness is added to the public key. For the receiver, it is necessary that the bitmasks are also send with the signature, as these are essential in the verification process [13].

3.2.3 XMSS

As stated in subsubsection 3.2.1 on Merkle trees, SPHINCS makes use of the extended version of Merkle trees (XMSS) instead of the standard Merkle trees. XMSS uses WOTS+ as OTS and adds an L-tree at the leaves of the tree to compress the WOTS+ public key. In addition to the use of L-trees, the way two nodes are combined together in XMSS is different from the process in a standard Merkle tree: the process namely includes the use of a bitmask. Before two nodes are hashed together using a hash function $H : \{0,1\}^{2n} \rightarrow \{0,1\}^n$, a bitmask is added bitwise using a XOR gate to both of these nodes. Thus, in general it holds that $N_{i,j} = H((N_{2i,j-1} || N_{2i+1,j-1}) \oplus \mathbf{Q}_j)$. Where N is the node, i, j describe the location with $0 < j \leq h$ and $0 \leq i < 2^{h-j}$ and $\mathbf{Q}_j$ is the set of bitmasks. See Figure 4.

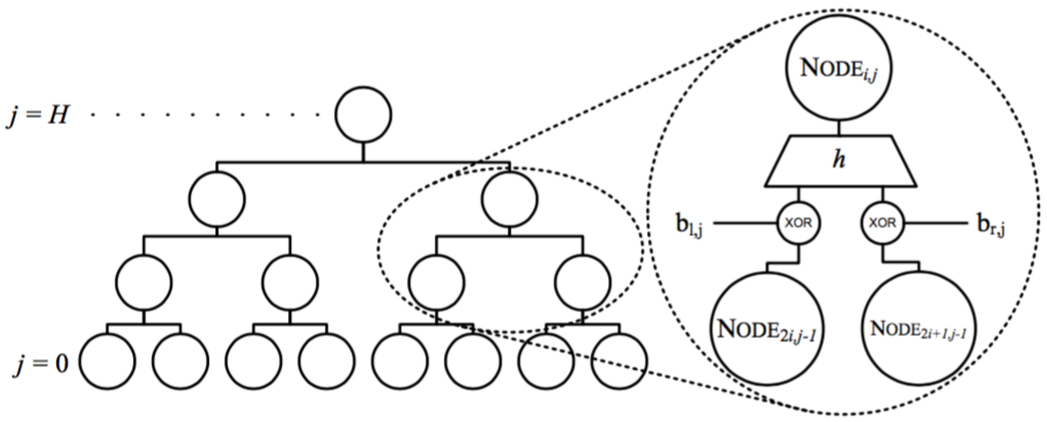

Figure 4: XMSS tree [9]

The leaf nodes of the XMSS tree are the different WOTS+ public keys. The size of each WOTS+ public key is $n \cdot l$ bits, hence the bitmasks on the leaf node layer also have this size. In order to reduce the size of the WOTS+ public key and the bitmasks, L-trees are incorporated. These L-trees take the WOTS+ public key and compress it. A WOTS+ public key consists of l parts, each of them is n bits long. These parts become the leaves of the L-tree, which uses the same principle as an XMSS tree. However, l is often not a power of 2, which means it is not possible to form a full binary tree. The L-tree is therefore modified. It is called an L-tree, because it always uses the left nodes. That means if a left node in the tree does not have a right sibling node then this node is pushed to the next layer. The root node of the L-tree is a public key that is n bits long instead of $n \cdot l$ [11]. Take for example $l = 7$ (see Figure 5), 7 is not a power of 2, so the L-tree is not a full binary tree. Instead 6 out of the 7 nodes have a sibling node, which can then be hashed together to become the node for the next layer. The leftover node that does not have a sibling node, skips this layer and is added to the layer above. So the next layer contains 3 nodes that were hashed and the leftover node, which is in total 4 nodes. Since 4 is a power of 2, the tree acts again like a normal binary tree.

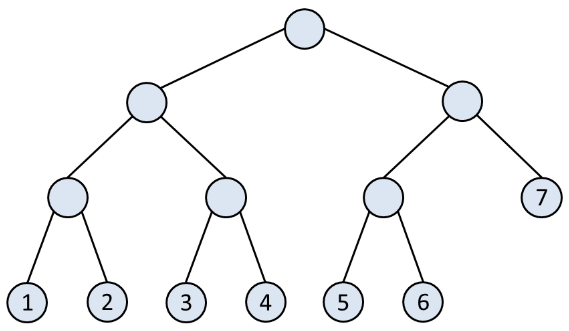

Figure 5: L-tree with 7 starting nodes

3.2.4 HORST

SPHINCS is defined as a stateless hash-based signature scheme. In the previous subsections the basics of the SPHINCS signature scheme are discussed, however so far none of these basics make SPHINCS stateless. Stateful means that you have to keep track of all the keys that have been used. Thus, if we consider a system that involves a large number of keys (e.g. 2^{64} keys, the aims of the NIST competition), it would be much more convenient if the system is stateless.
To achieve statelessness in SPHINCS, HORST is used. HORST (Hash to Obtain Random Subset Tree) is a variant of HORS (Hash to Obtain Random Subset), but then build up into a tree, as the name already implied. HORS is not a one-time signature but a few-times signature. In comparison to using a one-time signature more than once, a few-time signature has a lower loss in security when it is used more than once.
The key generation using HORS works as follows. First of all, a secret key of t bit strings with length n is generated. To compute the corresponding public key, each of the t bit strings has to be hashed individually using a length preserving hash function, $F : \{0,1\}^n \to \{0,1\}^n$.
To generate a HORS signature, a message of arbitrary length, $M \in \{0,1\}^*$, is hashed using a different hash function than F, $H : \{0,1\}^* \to \{0,1\}^{k\cdot\log_2(t)}$, which will hash the message into a bit string that has length $k \cdot \log_2(t)$. This allows the hashed message, $H(M)$, to be divided into k parts of length $\log_2(t)$, so $H(M) = (h_0, h_1, \ldots, h_{k-1})$. Because the length of each part is $\log_2(t)$, the integer h_i is a value between 0 and $t-1$. Using the values of h_i, the following signature can be created: $\sigma = (sk_{h_0}, sk_{h_1}, \ldots, sk_{h_{k-1}})$. This signature has length $k \cdot n$. Note that the signature only uses a selection of the available secret key parts and it is possible that a particular secret key part is used multiple times.
In order to verify the signature, all the receiver has to do is compute $H(M) = (h_0, h_1, \ldots, h_{k-1})$ and $(F(sk_{h_0}), F(sk_{h_1}), \ldots, F(sk_{h_{k-1}})) = (pk_{h_0}, pk_{h_1}, \ldots, pk_{h_{k-1}})$. If these public key parts match the correct parts of the public key, then the signature is correct.
In conclusion, HORS generates a public key and signature of size $k \cdot n$ bits. To avoid having to deal with a public key and a signature of a size this large, HORST sacrifices runtime and thereby reduces the public key size and signature [9]. HORST uses a binary tree, a message length m, a parameter k and a amount of bit strings $t = 2^\tau$ with $k \cdot \tau = m$. Firstly, a secret key is generated and similar to HORS this secret key consists of t bit strings, so $sk = (sk_1 \ldots, sk_t)$. These are random bit strings of length n. Similar

to HORS, each of these secret key parts is hashed individually and form the leaves of the tree. To construct the rest of the binary tree, bitmasks $\mathbf{Q} \in \{0,1\}^{2n \times \log(t)}$ are used. The bitmasks are used in a similar way as in an XMSS tree.
To sign a message M, the message is divided in k parts, so $M = (M_0, \ldots, M_{k-1})$. Since the length of the message equals $k \cdot \tau = m$, each part of the message has a length of τ bits. This implies that each message part is a value between 0 and $t - 1 = 2^\tau - 1$, which in its turn corresponds to one of the leaf nodes. Recall that for HORS the signature is given by $\sigma = (sk_{M_0}, sk_{M_1}, \ldots, sk_{M_{k-1}})$. However, HORST uses a binary tree and because of that, k authentication paths, $Auth_{M_i}$, have to be added to the signature. Otherwise it would not be possible to verify the signature for a given public key. Thus, for HORST the signature is given by $\sigma = (\sigma_0, \ldots, \sigma_{k-1})$, where $\sigma_i = (sk_{M_i}, Auth_{M_i})$. In order to reduce the signing time, a σ_k is added to the signature. σ_k contains 2^x nodes of the tree. These are the nodes on the top part of the HORST tree, so $Auth_{M_i}$ only has to provide information about the path up till these nodes. σ_k is added because these top nodes are almost always being used to sign and verify the signature. In short, σ_k contains the top nodes $(N_{0,\tau-x}, \ldots, N_{2^x-1,\tau-x})$, hence the authentication path is defined as $Auth_{M_i} = (A_0, \ldots, A_{\tau-1-x})$.
To verify the signature the receiver computes $M = (M_0, \ldots, M_{k-1})$. Next, each of these message parts is checked using the authentication path in order to verify whether the obtained signature matches with the given public key [9].

3.2.5 Hypertree

As stated before, one of the aims of the NIST competition was to have a signature scheme that could generate 2^{64} signatures that can be verified with the same public key. If we would use an XMSS tree for this purpose, then a tree of 64 layers is needed and each node has to be computed in order to obtain the root node, public key. This would require a huge amount of computation time, hence one big XMSS tree would not be ideal.
In order to obtain such a large tree without running into performance issues, the tree can be split into multiple trees, a tree of trees also known as a hypertree. SPHINCS is a hypertree of height h consisting of d layers of XMSS trees, so each XMSS tree will have a height corresponding to $\frac{h}{d}$. Starting at the top of the hypertree, at layer $d-1$, is a tree of height $\frac{h}{d}$. Therefore, the top XMSS tree contains $\frac{h}{d}$ leaf nodes. Each of these leaf nodes is connected to a tree on the layer below. That results in a layer that consists of $\frac{h}{d}$ trees. If this process is repeated until the last layer (layer 0), then you will end up with $2^{\frac{(d-1)\cdot h}{d}}$ XMSS trees on this last layer. Each of these XMSS trees contains $2^{\frac{h}{d}}$ leaf nodes, so in total there are $2^{\frac{(d-1)\cdot h}{d}} \cdot 2^{\frac{h}{d}} = 2^h$ leaf nodes at the bottom layer. However, note that we have now described a hypertree consisting of XMSS trees only. In order to make the hypertree stateless, each of these 2^h leaf nodes on the bottom layer is connected to the few-time signature scheme, HORST. The full hypertree is depicted in Figure 6:

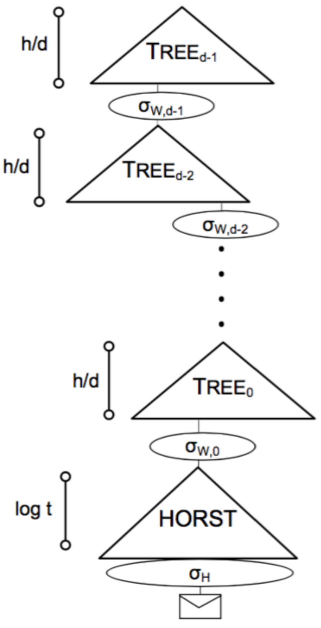

Figure 6: Structure of SPHINCS [9]

Furthermore, the hypertree is never fully computed. In the next section we will see that only one route through the hypertree has to be computed for the signing and verification processes. To obtain the public key (as part of the key generation process), only the top tree has to be computed, since the hypertree is split in multiple trees and each tree is independent of each other. However, this does require the hypertree to be consistent, meaning that recomputing the hypertree will always give the same outcome. In order to make the hypertree consistent, SPHINCS uses a simple addressing scheme for pseudorandom key generation. This way the key-pairs will be pseudorandomly generated and appointed to the correct position in the hypertree. An address is a bit string of length $A = \lceil \log(d+1) \rceil + (d-1)(h/d) + (h/d) = \lceil \log(d-1) \rceil + h$. The first $\lceil \log(d+1) \rceil$ bits will indicate which layer the tree is on, during the signing process this will be a counter, keeping track of the layers (note that this also includes the HORST trees). The next $(d-1)(h/d)$ will indicate which tree on the layer is used, numbering the tree from left to right. The last (h/d) bits indicate the position of the WOTS+ key-pair in the XMSS tree (again numbering from left to right). The address, A, is just the indication of the position, to actually obtain the right WOTS+ or HORST key-pair, both the address and the SPHINCS secret key are inserted in a pseudorandom function to compute a seed. This seed, $\mathcal{S} \in \{0,1\}^n$, can then be used to compute the corresponding WOTS+ or HORST secret key [9].

3.3 SPHINCS: key generation, signing and verification

In subsubsection 3.2.5 only a short introduction and overview of the hypertree used in SPHINCS is given. In order to fully comprehend the use of this hyper tree in SPHINCS, we have to look at the key generation, signing and verification processes of SPHINCS. In this section these processes are explained step-by-step.

3.3.1 Key generation

The key generation process starts by sampling two secret key values (SK_1, SK_2) from $\{0,1\}^n \times \{0,1\}^n$, where SK_1 is used for pseudorandom key generation and SK_2 is used to compute an unpredictability factor for the signing process. Besides these values also p uniformly random n-bit values are sampled, $\mathbf{Q} \rightarrow \{0,1\}^{p\times n}$, that will be used as bitmasks. For SPHINCS p is given as $p = \max\{w-1, 2(h+\lceil \log_2 l \rceil), 2\log t\}$. Note that $\{w-1, 2(h+\lceil \log_2 l \rceil), 2\log_2 t\}$ is used, because it corresponds to the different parts of the scheme where bitmasks are used. The WOTS+ parts of the scheme use the first $w-1$ bitmasks that are sampled: $\{0,1\}^{(w-1)\times n}$. The L-trees use the first $2(h+\lceil \log_2 l \rceil)$ bitmasks ($\{0,1\}^{(2(h+\lceil \log_2 l \rceil))\times n}$) and the HORST instances use the first $2\log_2 t$ bitmasks ($\{0,1\}^{(2\log_2 t)\times n}$). Hence, we may denote $\mathbf{Q}$ as $\mathbf{Q} = (\mathbf{Q}_{HORST}, \mathbf{Q}_{WOTS+}, \mathbf{Q}_{L-tree})$.
The last part of the key generation process consists of the computation of the public key. As mentioned before, only the top tree of the hypertree needs to be computed for this. To do so, first the WOTS+ key pairs that correspond to the top tree are generated by obtaining the correct seeds from the addresses and SK_1. This gives us the WOTS+ secret keys and with the WOTS+ bitmasks we can compute the WOTS+ public keys. The WOTS+ public keys are then compressed with an L-tree to generate the leaf nodes of the top XMSS tree. Next, the top XMSS tree is built and its root node becomes PK_1. The public key is $PK = (PK_1, \mathbf{Q})$ and the secret key is $SK = (SK_1, SK_2, \mathbf{Q})$. A brief overview of the key generation process in SPHINCS is given below.

SPHINCS key generation

1. Generate $SK = (SK_1, SK_2) \in \{0,1\}^n \times \{0,1\}^n$.
2. Choose p random bitmasks $\mathbf{Q} \rightarrow \{0,1\}^{p\times n}$.
3. Obtain the WOTS+ secret keys from the seeds
4. Compute the WOTS+ public keys from the WOTS+ secret keys and the WOTS+ bitmasks.
5. Compress each WOTS+ public keys with an L-tree.
6. Compute the root node of the top XMSS tree.
7. Output:
   - Secret key: $SK = (SK_1, SK_2, \mathbf{Q})$.
   - Public key: $PK = (PK_1, \mathbf{Q})$.

3.3.2 Signing process

Once the secret and public key are generated, the signing process can commence. Consider a message of arbitrary length $M \in \{0,1\}^*$. This message is transformed into a message digest D. The main reason for this is to create an input of a fixed amount of bits. To compute the message digest an unpredictability factor $R = (R_1, R_2) \in \{0,1\}^n \times \{0,1\}^n$ is required. Recall that during the key generation process SK_2 was sampled from $\{0,1\}^n$. The unpredictability factor R is computed by putting the message M and SK_2 together in a pseudorandom function, PRF for short. Next, R_1 is hashed together with the message M to create a message digest $D \in \{0,1\}^m$. Using the other part of R, R_2, an index i is created. The index i and R_1 are saved as the first two parts of the overall SPHINCS signature Σ. i is the first h bits of R_2, where h is the height of the full hypertree. That means i is a number between 0 and 2^h, therefore it can function as an indication for the addresses. With index i the HORST address is given as $A_{HORST} = (d||i)$, where d is the amount of tree layers in the hypertree. The address of the HORST secret key corresponds to the bottom layer of the lowest XMSS tree layer (layer 0 in Figure 6). The first $(d-1)(h/d)$ bits of i are used to indicate which tree on the bottom layer will be used and the remaining h/d bits are used to indicate the position in the tree. Using the HORST address and SK_1 in a PRF yields the corresponding seed and via this seed the HORST secret key sk_H is obtained.

For the HORST part of the signing process, we have as input D, the HORST bitmasks $\mathbf{Q}_{HORST}$ and the HORST secret key sk_H, that we obtained in the previous step. The output of HORST is σ_H and pk_H, where σ_H is the HORST signature and pk_H the HORST public key. σ_H is part of the overall SPHINCS signature and pk_H is used in layer 0 of the hypertree.

The first WOTS+ signature is created using pk_H as input, which acts as a message that has to be signed (see subsubsection 3.2.2 on WOTS+). The address of the WOTS+ secret key on the bottom layer is $A_0 = (0||i)$ and is used (in combination with $SK1$, to compute the corresponding seed first) to obtain the WOTS+ secret key $sk_{W,0}$. Using $sk_{W,0}$, the WOTS+ bitmasks $\mathbf{Q}_{WOTS+}$ and pk_H we can compute the first WOTS+ signature $\sigma_{W,0}$. To obtain the WOTS+ public key, $pk_{W,0}$, we can either hash the WOTS+ secret key w times or use the WOTS+ verification algorithm. The latter option requires less calls for the hash function, since we already hashed the WOTS+ secret key a few times to compute the signature $\sigma_{W,0}$. Thus, by means of $\sigma_{W,0}$, pk_H and $\mathbf{Q}_{WOTS+}$ we can compute $pk_{W,0}$ using the verification algorithm of WOTS+. This WOTS+ public key is first compressed using the L-tree and then used as leaf node in the XMSS tree. From this XMSS tree we can compute the root of the tree, the XMSS public key pk_0, and obtain the authentication path $Auth_{A_0}$. $\sigma_{W,0}$ and $Auth_{A_0}$ are added to Σ and pk_0 is used as new message in the tree above.

The previous steps are repeated for the trees that form a path to the top node, layer 1 to $d-1$ of the hypertree. On each layer $1 \leq j < d$ a WOTS+ secret key signs the root node, pk_{j-1}, of the previous layer. The addresses to obtain the correct WOTS+ secret keys is $A_j = (j||i)$, where the first $(d-1-j)(h/d)$ bits of i indicate which tree on the layer is used and the last h/d bits of i indicate the position inside the tree. For each layer the WOTS+ signature and the authentication path are added to Σ. In the end the following overall SPHINCS signature is obtained: $\Sigma = (i, R_1, \sigma_H, \sigma_{W,0}, Auth_{A_0}, \ldots, \sigma_{W,d-1}, Auth_{A_{d-1}})$. A brief overview of the signature process in SPHINCS is given below.

SPHINCS Signing process

1. Input: Message M and secret key $SK = (SK_1, SK_2, \mathbf{Q})$.
2. Compute unpredictability factor $R = (R_1, R_2) \in \{0,1\}^n \times \{0,1\}^n$ from M and SK_2.
3. Compute message digest $D \in \{0,1\}^m$ from M and R_1.
4. Obtain index i from R_2. Add i and R_1 to the overall signature Σ.
5. Obtain the HORST secret key sk_H using SK_1 and the HORST address, $A_{HORST} = (d||i)$.
6. Create the HORST signature σ_H and the HORST public key pk_H using D, sk_H and the HORST bitmasks $\mathbf{Q}_{HORST}$.
7. Obtain the WOTS+ secret key of the first layer $sk_{W,0}$ using the address $A_0 = (0||i)$ and SK_1.
8. Compute WOTS+ signature $\sigma_{W,0}$ using pk_H as message, $sk_{W,0}$ and the WOTS+ bitmasks $\mathbf{Q}_{WOTS+}$. Add $\sigma_{W,0}$ to Σ.
9. With $\sigma_{W,0}$ compute the WOTS+ public key $pk_{W,0}$ using the verification algorithm.
10. Compress $pk_{W,0}$ with the L-tree and use this as leaf node in the XMSS tree.
11. Compute the XMSS tree to obtain the root node, pk_0, and the authentication path $Auth_{A_0}$. Add $Auth_{A_0}$ to Σ.
12. Repeat steps 7 - 11 for the other XMSS layers $j \in \{1, d-1\}$. For each layer j, use pk_{j-1} as message and add $\sigma_{W,j}$ and $Auth_{A_j}$ to Σ.
13. Output: signature $\Sigma = (i, R_1, \sigma_H, \sigma_{W,0}, Auth_{A_0}, \ldots, \sigma_{W,d-1}, Auth_{A_{d-1}})$.

3.3.3 Verification process

To verify that a SPHINCS signature actually matches the public key, the receiver follows the following steps. The receiver knows the message M, the signature Σ and the public key PK. The receiver starts by computing the message digest D. Again, from the signature R_1 is obtained and together with the message the message digest is computed. Note that from the public key we know the bitmasks $\mathbf{Q}$. The message digest D, the HORST signature σ_H and the HORST bitmasks $\mathbf{Q}_{HORST}$ are then used in the HORST verification algorithm, which gives the HORST public key pk_H as output. Similar to the signing process, pk_H, the first WOTS+ signature $\sigma_{W,0}$ and the WOTS+ bitmasks are used for the WOTS+ verification, in order to compute the first WOTS+ public key, $pk_{W,0}$. Now that $pk_{W,0}$ is obtained, it can be compressed with an L-tree to obtain the corresponding leaf node. Using $Auth_{A_0}$ and the compressed $pk_{W,0}$, we are able to compute the root node

pk_0. Again this process is repeated for each layer in remaining layers of the hypertree. The last node that will be computed is the root node of the tree on the top layer. If this root node value equals the public key PK, we can conclude that the signature is correct and corresponds to the message. A overview of this verification process is given below.

SPHINCS Verification process

1. Input: Message M, public key PK and signature $\Sigma = (i, R_1, \sigma_H, \sigma_{W,0}, Auth_{A_0}, \ldots, \sigma_{W,d-1}, Auth_{A_{d-1}})$.
2. Compute message digest $D \in \{0,1\}^m$ from M and R_1.
3. Compute the HORST public key pk_H using D, the HORST signature σ_H and the HORST bitmasks $\mathbf{Q}_{HORST}$ in the HORST verification algorithm.
4. Compute the first WOTS+ public key $pk_{W,0}$ using pk_H and the first WOTS+ signature $\sigma_{W,0}$.
5. Compress $pk_{W,0}$ to obtain the XMSS leaf node.
6. Use the compressed $pk_{W,0}$ and authentication path $Auth_{A_0}$ to obtain the XMSS root node pk_0.
7. Repeat steps 4 - 6 for layers $j \in \{1, d-1\}$ to obtain the root node of the tree in the top layer of the hypertree.
8. Compare the obtained root node to the PK, if equal then the signature is correct.

4 Potential improvements in SPHINCS

Two main types of potential improvements in SPHINCS can be identified: improving its performance and improving its security. Both of these types of improvements can be reached by changing the parameters of SPHINCS. However, there is a balance between performance and security. In order to decrease the amount of computation time (a measure of performance) the parameters have to be smaller (e.g. a smaller d means less layers and therefore less WOTS+ instances), but this will interfere with the security level of the scheme. On the other hand, increasing the security of a scheme can cause issues with regard to the performance. In addition, SPHINCS has some properties that an attacker could use to his or her advantage. This section will explain these properties. In subsection 6.3 we will look at these properties again and see how SPHINCS$^+$ and Gravity-SPHINCS have dealt with them.

4.1 Multi-target attack

SPHINCS is designed to be collision-resilient by introducing bitmasks. This means that there are no bit strings $A \neq B$ with $H(A) = H(B)$. However, the used hash functions leave the possibility open to find a (second-)preimage. Since SPHINCS is such a large signature scheme, it uses a lot of calls for a hash function. In SPHINCS the same hash function is used whenever a call for a hash function is required. The potential danger in this is that an attacker can use this as an advantage. With regard to preimage resistance, an attacker will succeed when it finds an x' such that $H(x) = H(x')$. Since SPHINCS uses the same hash function in the entire hypertree, there are a lot of possibilities for the attacker to actually find a preimage. One preimage is already sufficient for the attacker to break the property. Therefore, SPHINCS should increase its security, for instance by using a different hash function each time one is required [2].

4.2 Unverifiable index

Recall that in SPHINCS the index i is generated by taking the first h bits of R_2. This index is used to obtain the HORST secret key from an address. i is fully dependent on R_2. Therefore i is added to the signature, because a receiver is not able to recalculate R_2, since he or she does not have the secret key SK. Therefore, even though index i is pseudorandomly selected, the receiver of the message is not able to verify this. Seeing that HORST is a few time signature scheme, there is an opportunity for the attacker to uncover necessary secret key parts. If an attacker is able to find an index that creates an address of the HORST secret key of which the necessary secret key parts are already uncovered, then it is possible for the attacker to forge a signature using this index. Since the receiver cannot verify the index, the attacker can use the same index each time to get the most favourable HORST secret key [2].

4.3 Colliding indices in HORST

There is a property in HORST that can be troublesome in SPHINCS when certain parameters are chosen too small. In HORST the message digest is split up in k indices. Each of these k indices selects a secret key part from the generated HORST secret key. There is however nothing that prevents the k indices from selecting the same secret key part (also known as the collision of indices). In the worst case each of the k indices selects the same secret key part. This means that only one secret value has to be known in order to sign a message. The likelihood that this happens is relatively small for a large enough k and when enough different secret key parts are available (a large value for t). However, an attacker can use this property as an advantage. Choosing larger parameters is an option to lower the advantage for an attacker, but this will also increase the computation time. The better option would be to prevent the k indices from selecting the same secret key parts [2].

5 SPHINCS$^+$ & Gravity-SPHINCS

SPHINCS$^+$ and Gravity-SPHINCS are both adaptations of SPHINCS. These adaptations meet the standards NIST asked for, as well as make the scheme more secure and increase its performance. SPHINCS$^+$ and Gravity-SPHINCS can be defined in terms of changes that have been made with respect to SPHINCS and also include some additions. In Table 1 an overview is given of these changes and additions. This section discusses each of these changes and additions of SPHINCS$^+$ and Gravity-SPHINCS.

	SPHINCS$^+$	**Gravity-SPHINCS**
Changes	Tree-less WOTS+ public key compression FORS Message digest	Mask-less hashing PORST
Additions	Tweakable hash function	Secret key caching Octopus authentication Batch signing

Table 1: Changes and additions in SPHINCS$^+$ and Gravity-SPHINCS with regard to their predecessor SPHINCS

5.1 SPHINCS$^+$

First, we will discuss the changes and additions made to SPHINCS by SPHINCS$^+$. The overall structure of the scheme remains similar to that of SPHINCS. However, as stated before, SPHINCS should be more optimized and still contains certain vulnerabilities. The main additions and changes in SPHINCS$^+$ are: tweakable hash functions, tree-less WOTS+ public key compression, FORS and changes in the message digest. Each of these changes and additions will be explained in this section, including the rationale for the choices that were made.

5.1.1 Tweakable hash functions

A major addition in SPHINCS$^+$ is the introduction of tweakable hash functions. That means for each call for an hash function in SPHINCS$^+$ a different key and a different bitmask is used. This sounds like a lot of keys and bitmasks, however, these are pseudo-randomly generated from an address and a public seed. Note that the hash function itself does not change, instead each time a different public seed and an address are added to the input. The result of this procedure is that each call is independent of each other. The reason for making each hash function call independent of each other is to add protection against multi-target attacks. As explained in subsection 4.1, an attacker wants to invert the hash function or find a second preimage for a given target value. Ideally, this would require brute force. However, for SPHINCS there are numerous calls for the same hash function, giving the attacker many targets. Note that only one second preimage has to be found for an attacker to succeed. So making each hash function call different from the other hash function calls will mitigate this type of attack.

5.1.2 Tree-less WOTS+ public key compression

With the introduction of tweakable hash function in SPHINCS$^+$, the bitmasks are pseudo-randomly generated and not stored in the public key (which was the case in SPHINCS). This way, it is possible to replace the L-tree by a single call for a hash function. In SPHINCS, the main reason for using L-trees is to compress the WOTS+ public keys and thereby reduce the amount of bitmasks that are needed. With L-trees the WOTS+ public keys went from $n \times l$ bits long to n bits long. The advantage of this is that the

length of the bitmasks is also smaller. Note that without the use of L-trees bitmasks of $2n \times l$ are needed for the first layer of the XMSS tree. Using L-trees this is reduced to $2n$. However, the L-trees also require bitmasks. This is fortunately only $\log(l)$ bitmasks instead of l. In conclusion, in total less bits for bitmasks are required by introducing L-trees. In SPHINCS these bitmasks were stored in the public key. So the more bits used for bitmasks, the larger the size of the public key. The disadvantage of adding L-trees is that the computation time becomes larger. Thus, in SPHINCS a balance exists between the public key size and extra computation time.
In SPHINCS$^+$ the bitmasks are not stored in the public key anymore and are generated separately. This make the whole purpose of the L-trees redundant, only resulting in additional computation time. So instead of L-trees SPHINCS$^+$ uses a singular call to a tweakable hash function in order to reduce the size of the WOTS+ public keys.

5.1.3 FORS

In SPHINCS$^+$, HORST is changed into FORS (Forest Of Random Subsets). As stated before in subsection 4.3, in HORST there is a likelihood that the k indices, created by the message digest, collide. FORS uses, instead of just one tree, a collection of trees that together form a forest. It works as follows: there are k trees of height a. The height is a so each tree has $t = 2^a$ leaf nodes. The secret key used for FORS consists of $k \cdot t$ random n-bit strings, in other words k sets of t n-bit strings. Each of these n-bit strings can be appointed to one of the leaf nodes. Similar to HORST in SPHINCS, FORS signs the message digest and the FORS public key is used further on in the hypertree. The message digest in SPHINCS$^+$ is also changed with respect to SPHINCS, as will be discussed in subsubsection 5.1.4. For now lets call the message digest md. For FORS md is divided in k strings, m_i with $i \in [0, k-1]$. Each string corresponds to one tree in the forest. To decide which secret key element is to be used for each tree, each m_i corresponds with an integer ranging from 0 to $t-1$. To create the FORS signature, each of the strings is signed with the corresponding secret key element. This means each signature of the selected secret key elements will contain an authentication path of length a.
Once the FORS signature is obtained, the FORS public key has to be computed, in order to use it as input in the first layer of the hypertree (layer 0). This public key is computed by the verification algorithm of the FORS signature. In order to do this we need to compute the root nodes of the k trees, using the authentication paths and signatures, and hash each root node together to produce the FORS public key.
The main difference between HORST and FORS is that instead of one tree, multiple trees are used in FORS. This ensures that a different secret key part is used each time. So the problem discussed in subsection 4.3 about the same secret key parts that are selected, is solved. However, a useful property of HORST is that each of the secret key parts belongs to the same tree to generate the HORST public key. This means that a lot of authentication paths overlap. Hence, the top nodes of the tree are used more often. In HORST these top nodes are saved in order to reduce the length of the authentication paths, thereby reducing its size and computation time. Regarding performance, the same parameters k and t will yield a lower computation time in HORST than in FORS. Fortunately for FORS, because the tree is split into multiple trees, the sizes of the trees do not have to be as large as the size of a single HORST tree. Hence, the values for k

and t can be much smaller resulting in smaller signature sizes [2].

5.1.4 Message digest

In SPHINCS$^+$ the way the message digest is computed is different from SPHINCS. Recall that in SPHINCS R is computed by putting the message M and SK_2 into a pseudo random function. In SPHINCS$^+$ an additional 256 bit value, $OptRand$, is added to the pseudo random function, so $R = PRF(SK, OptRand, M)$. The purpose of computing R this way is to add extra bits to R and avoid the signing to be deterministic. The message digest and the index are computed together by hashing R, the public key and the message. In SPHINCS the index i and R_1 are added to the signature, because this was required for the verification of the signature. However, in SPHINCS$^+$ the index is computed together with the message digest. This means that the index no longer needs to be added to the overall signature, because it can be recomputed anyway using R, the public key and the message, all of which are already given in the overall signature. The main reason for including the index in the message digest computation is to prevent an attacker from choosing an index. It makes the index verifiable. As explained in subsection 4.2, an unverifiable index can be used as an advantage for an attacker and it would be better if it was verifiable.

5.2 Gravity-SPHINCS

Gravity-SPHINCS is also based on SPHINCS and includes different changes and additions in comparison to SPHINCS$^+$. Although a lot of these changes look similar to those of SPHINCS$^+$, they were approached in a different way. The hypertree of Gravity-SPHINCS has more or less the same structure as SPHINCS. However, bitmasks are not used and therefore the hypertree consists of Merkle trees with WOTS public keys as leaf nodes. Similar to SPHINCS, these WOTS instances are compressed by L-trees. The bottom of the hypertree consists of PORST public key compression trees. As stated before, HORST has some collision problems (subsection 4.3) and in Gravity-SPHINCS HORST is substituted by PORST, which will be explained in more detail later in this section.
Apart from these changes to the overall structure of the hypertree, also other changes and additions, for instance secret key caching and batch signing, were made in order to reduce the key size and speed up the signing process. In summary, Gravity-SPHINCS implemented the following changes and additions: secret key caching, mask-less hashing, octopus authentication, PORST and batch signing.

5.2.1 Secret key caching

One of the additions in Gravity-SPHINCS is secret key caching. This means that the nodes of the first layers of the top Merkle tree are saved and put into a cache. The main reason for this is because these nodes are used in almost every single signature and saving them in a cache will remove the need to compute these nodes again. Consider for example caching the top c layers of the first Merkle tree. Now that these nodes are saved, they do not require anymore calls for a hash function. This will safe time in the signing process. Not only does this speed up the signing time, but it is also possible to decrease the signature size. In Gravity-SPHINCS between each Merkle tree is a WOTS instance.

If the top Merkle tree would be twice as large as the rest of the Merkle trees in the scheme, then for signature scheme of the same size it is possible to remove a set of Merkle trees from the bottom layer and thereby also remove some WOTS instances. If there are less WOTS instances per signature, then the size of the signature is smaller and requires less computation time. A disadvantage of secret key caching is that the top tree is larger and therefore this tree takes longer to compute during the key generation process. In conclusion, the advantages of secret key caching outweigh the latter disadvantage as the key generation process only happens once.

5.2.2 Mask-less hashing

In order to simplify the scheme and to reduce the key size, no bitmasks are used in Gravity-SPHINCS. In short, this means that instead of XMSS and WOTS+ in SPHINCS, normal Merkle trees and WOTS are used in Gravity-SPHINCS. Recall that the reason bitmasks are used in SPHINCS is that it will reduce the security risk to second preimage resistance instead of collision resistance[2]. Classical attacks against collision resistance are of $\mathcal{O}(2^{\frac{n}{2}})$, while attacks against second preimage resistance are of $\mathcal{O}(2^n)$ [14]. However, in a post-quantum world attacks against collision resistance and second preimage resistance are both of $\mathcal{O}(2^{\frac{n}{2}})$. This makes it arguable to leave out the bitmasks, because it makes the scheme more complicated and in a post-quantum world this advantage would be negligible. However, it is also arguable that bitmasks should be implemented, because we are not yet in a post-quantum world and the difference in security is quite significant, when you compare $\mathcal{O}(2^{\frac{n}{2}})$ to $\mathcal{O}(2^n)$.
Besides the simplification of the scheme, the public keys are smaller. Originally in SPHINCS, the bitmasks were stored in the public key. Now that the scheme is simplified by leaving the bitmasks out, the size of the public key is smaller.

5.2.3 Octopus authentication

Another addition in Gravity-SPHINCS is octopus authentication. Octopus authentication is implemented to remove redundant authentication nodes from authentication paths in binary trees. A huge part of the signature size is the result of the authentication paths in a HORST tree. If a message digest is divided, into k pieces, then the signature also contains k authentication paths. Since the tree is binary and k authentication paths are used, some nodes are stored multiple times in these authentication paths. Octopus authentication basically merges as many of these paths as possible and if one of the nodes is already stored in one of the previous paths, then this node is not stored an additional time in the signature. In SPHINCS this was already partly done by adding the top nodes to the signature, making the authentication paths shorter. Octopus authentication is basically an improvement to this principle. The effectiveness of octopus authentication increases when k and the tree are larger. For smaller trees just saving the top x layers will result in the same efficiency.

5.2.4 PORST

The issues mentioned in subsection 4.3, Gravity-SPHINCS solved by introducing PORS, which is the abbreviation of PRNG (pseudo random number generator) to obtain a random subset. In Gravity-SPHINCS PORS becomes PORST, similar to HORS becoming HORST in SPHINCS, because it is used in a tree. PRNG is a pseudorandom number generator, taking as input, similar to HORS, the message digest. The problem PORST solves is that in HORST the same secret key parts could be selected. In PORST, selecting the secret key parts is done in such a way that the message digest will continue creating a subset until each of the k indices in the subset is different from each other. This will slightly increase the computation time, but as compensation PORST solved the problem of selecting the same key parts.

5.2.5 Batch signing

Batch signing is an implementation that takes multiple messages together and signs them all at once. With certain batching methods it is possible to reduce the signing time and even reduce the signature size. However, it is not always the case that multiple message will be send at once. The implementation of this is rather specific and will change the setup of the signature scheme. For these reasons, this implementation will not be explained more thoroughly. For a more in depth description see [15].

6 SPHINCS$^+$ vs. Gravity-SPHINCS

In this section SPHINCS$^+$ will be compared to Gravity-SPHINCS. Both schemes use different approaches to increase the security and performance of SPHINCS. To meet the requirements of the NIST competition, i.e. a signature scheme that is able to create 2^{64} different signatures for one public key, while still keeping a reasonable computation time for the signing process, SPHINCS needed some optimization. Regarding its security, it was important that certain properties, which an attacker could use as an advantage, were changed.
The comparison is divided into three sections. The first section is a comparison between both schemes' improvement of HORST. For this FORS and PORS in combination with octopus authentication are compared. The second section compares the hypertree of Gravity-SPHINCS to the hypertree of SPHINCS$^+$. It focuses mainly on the structural changes in the hypertree, for instance Gravity-SPHINCS leaving out the bitmasks. The last part of this section focuses on whether both schemes have dealt with the potential improvements identified in section 4.

6.1 Improvement of HORST

Both schemes had a way to improve HORST, because of the property explained in subsection 4.3. In order to remove this property, Gravity-SPHINCS implemented PORST, where the message digest will continue creating a subset until each of the k indices in the subset is different from each other. This approach is simple and does not add too much extra computation time in comparison to HORST. Considering SPHINCS$^+$ on the other hand, HORST was changed into FORS, where instead of one tree multiple trees are used. FORS prevents indices from colliding by this change in the structure, which causes each selected secret key part to be independent of the other selected secret key parts. Unfortunately, the usage of multiple trees also means that each of these trees has

to be computed in order to generate the root nodes. To be more precise: k trees of $\log_2(t)$ have to be computed instead of just one.
At first glance FORS seems to differ a lot from PORST, but it is actually quite similar: the only thing is that the secret key values for FORS are spread over multiple trees instead of belonging to one tree. However, having the secret key values in the same tree, which is the case in PORST, gives the advantage that all authentication paths are from the same tree. Hence, octopus authentication could be incorporated into Gravity-SPHINCS. Octopus authentication allows to remove the redundant nodes from the authentication paths. By removing these redundant nodes the signature size will be reduced. This addition cannot be incorporated in FORS, since the authentication paths do not collide. However, something that can be done for FORS and not for PORS, without weakening the scheme too much, is choosing different values for the parameter t and k. Choosing a lower value for t makes the trees smaller and thereby also the length of the authentication paths, which will reduce the signature size. If k is lowered, then there are less trees, so less computation time and less authentication paths. Note that FORS is only able to sign a $k2^a$ bit message digest. So lowering k implies that the size of the message digest has to be reduced as well. However, this is no issue, because the message digest is created by hashing the message together with a random factor, which results in a fixed size of the message digest.
To further compare both schemes on their HORST improvements, we consider the estimated costs provided by both papers [2][3]. The estimated cost is based on the amount of calls for hash functions and operations. PORST and FORS are both used during the signing process and the verification process. Table 2 shows the estimated costs for both signature schemes.

	SPHINCS$^+$	**Gravity-SPHINCS**
Signing	$k \cdot t$ calls to PRF and F $k(t-1)$ calls to H one call to T_k	2 calls to H and a few calls to G t calls to G $t-1$ calls to H
Verification	$k \cdot t$ calls to PRF and F $k(\log_2 t - 1)$ calls to H one call to T_k	1 call to H and a few calls to G k calls to F $\leq k(\log_2 t - \lfloor \log_2 k \rfloor)$ calls to H

Table 2: Estimated costs for FORS and PORST

In FORS, H and F are hash functions, PRF is a pseudorandom function to generate the bitmasks and T_k is the hash to combine all the roots of the trees. In PORST, H and F are hashfunctions as well and G is a pseudorandom function to make sure each of the k indices is different. Most of the costs are approximately the same, but it is important to notice that for the verification process in PORST octopus authentication can reduce the amount of calls needed. If we take the same values for t and k in FORS and PORST, then we can conclude that FORS requires a factor k more calls for operations during the signing process. However, in FORS the parameters t and k can be chosen smaller, resulting in less calls for operations than in PORST.
For the same security level of 128 bits (security level according to the NIST requirements

[4]), PORST and FORS provided the following parameters. In order to reach 2^{64} signatures, SPHINCS$^+$ uses $t = 2^{15}$ and $k = 10$ and Gravity-SPHINCS uses $t = 2^{16}$ and $k = 28$. For these chosen parameters Gravity-SPHINCS costs less with regard to both signing and verification. However, SPHINCS$^+$ provides an option with a different set of parameters where $t = 2^{10}$ and $k = 30$. From comparing these parameters to the ones from PORST, we may conclude that FORS requires less calls for the hash function, because $30(2^{10} - 1) < 2^{16} - 1$. As mentioned before, it is easier in FORS to change the parameters in such a way that the level of security stays the same, while reducing the estimated cost. For PORST, in order to increase the security, both t and k have to be increased. Increasing t will result in a larger amount of operations. The security level of SPHINCS$^+$ with $t = 2^{10}$ and $k = 30$ is 256 bits and gives a signature size of 49216 bytes. Of course this option takes a while to compute and the overall outcome of the signature scheme does not only depend on the parameters t and k [2][3].
In conclusion, comparing these estimated costs really depends on what values are chosen for parameters t and k. For the same set of parameters PORST will be the better option in terms of estimated costs. However, if we consider the level of security, then FORS would be the better option for the same set of t and k. A comparison based on different sets of parameters for FORS and PORST concludes that FORS performs better in terms of estimated costs, whilst both provide the same level of security.

6.2 Hypertree comparison

When comparing the hypertree the main differences between SPHINCS$^+$ and Gravity-SPHINCS schemes is that in Gravity-SPHINCS the bitmasks are no longer used. In SPHINCS$^+$ they are still used and they even have their own address to be called from. Bitmasks are added to a signature scheme to make sure that the hash functions are collission resistant. The reasons why Gravity-SPHINCS does no longer use the bitmasks is because it makes the scheme less complex and in a post-quantum world attacks against collission resistance and second preimage resistance are both of $\mathcal{O}(2^{\frac{n}{2}})$. This means that in the current situation, in the pre-quantum world, Gravity-SPHINCS is susceptible to attacks of order $\mathcal{O}(2^{\frac{n}{2}})$, while SPHINCS$^+$ is susceptible to attacks of $\mathcal{O}(2^n)$, which is a massive difference. The amount of operations needed for Gravity-SPHINCS on the other hand is lower than that of SPHINCS$^+$, because the bitmasks are not used in Gravity-SPHINCS. The additional operations needed for SPHINCS$^+$ to add the bitmasks is: $2^{\frac{h}{d}}l$ for the key generation, $kt + d(2^{\frac{h}{d}})l + 1$ for the signing process and kt for the verification process, where h is the height of the hypertree, d the amount of subtrees, l the number of n-bit string element in WOTS+, k amount of FORS trees and t the number of leaves of a FORS tree. The total estimated costs for the bitmasks would thus be an additional $2kt + (d+1)(2^{\frac{h}{d}}l) + 1$ operations, since each time a hashfunction is used also a bitmask is applied. Even though this would require additional computation time, the security level bitmasks provide in SPHINCS$^+$ is significantly higher.
In SPHINCS, bitmasks were stored in the public key. For SPHINCS$^+$ this no longer the case, because they are pseudorandomly generated. That means the public keys are smaller. The reason L-trees were implemented in SPHINCS was to compress the WOTS+ public keys, so that the root nodes of the XMSS tree would require $2^(h/d) \times \log l$ bitmasks of length $2n$ bits instead of $2^(h/d)$ bitmasks of length $2n \times l$ bits. In conclusion, compres-

sion successfully decreases the total amount of bits used for bitmasks. The only problem is that the compression using L-trees results in additional computation time. As the bitmasks for SPHINCS$^+$ are generated anyway, the trade-off is made to remove the L-trees and use a tweakable hash function instead. In Gravity-SPHINCS however, the L-trees are still included, whereas no bitmasks are present in the scheme. There is no additional advantage in using L-trees to compress the size of the WOTS public keys in comparison to using a standard hash function. In conclusion, the L-trees in Gravity-SPHINCS do no have any effect and only result in extra computation time.
One of the last things to compare is the implementation of secret key caching. Gravity-SPHINCS implemented this addition so the top of the hypertree would be cached. This increases the key generation time, but in exchange it will reduce the signing time and the signature size. In section 8.2 of the SPHINCS$^+$ paper about discarded changes[2] the addition of secret key caching is discussed. The provided reason for discarding the idea of secret key caching is that the computation time of the key generation will increase a lot and the top tree has to be handled in a different way. However, the key generation has to be performed once, whereas a reduction in signing time and signature size takes place every time a signature is made. Therefore, implementing secret key caching in SPHINCS$^+$ is advantageous in the long run, under the condition that potential problems in handling the top tree can be solved [2][3]. However, before making any conclusive statements about this, both schemes have to be tested on their runtime with and without the secret key caching. This could be part of a more practical comparison between SPHINCS$^+$ and Gravity-SPHINCS, which goes beyond the aims of this thesis.

6.3 Potential improvements in SPHINCS

In section 4 potential improvements SPHINCS were discussed. These improvements concern some properties of SPHINCS that could be exploited and were identified as follows: multi-target attack, an unverifiable index and the colliding indices in HORST.
The improvements for HORST were FORS and PORST. These two have already been compared in subsection 6.1 and both systems provided a way to solve the potential problem of HORST.
Multi-target attacks have to do with the fact that in SPHINCS every time the same hash function was used when generating a signature. Because the hash function was used this much, it left an attacker with the possibility to use it as an advantage when trying to break one of the hash function properties. SPHINCS$^+$ came with the improvement to tweak the used hash function each time it is used. This means each hash function and bitmask is called from a public seed and is called upon with a different key. Gravity-SPHINCS on the other hand uses the same hash function each time. An improvement for Gravity-SPHINCS is to do the same as SPHINCS$^+$ and change the hash function into a tweakable hash function in order to mitigate the multi-target attacks opportunities.
The last potential improvement concerns the unverifiable index. In SPHINCS$^+$ the index and the message digest are computed by hashing a randomization factor, the public key and the message together. This way it is possible to verify the index and the index also does not have to be included in the overall signature anymore. In Gravity-SPHINCS the index is computed during the PORST process. It starts by hashing the message digest to-

gether with salt (some randomly generated bits) to create the public salt. The salt in this case works as unpredictability factor. To obtain the actual index in Gravity-SPHINCS, the public salt is hashed with the message digest to compute a factor g. This factor will be used in a pseudorandom number generator together with an address $A(0, 0)$ to compute a factor b, of which the last h (the height of the tree) bits from the index λ. Important to note is that this way of computing the index allows the receiver to verify the index.

7 Conclusion

In this thesis two stateless hash-based signature schemes SPHINCS$^+$ and Gravity-SPHINCS, submitted for NIST's post-quantum cryptography competition, are compared and evaluated in a theoretical way. As both schemes are based on their predecessor, SPHINCS, first a detailed explanation of SPHINCS and its components was given. We described the key generation, signing and verification processes of SPHINCS to get a better understanding of the importance and function of all its elements. We identified three different potential improvements for SPHINCS: multi-target attack protection, the use of a verifiable index and prevention of colliding indices in HORST. To compare and evaluate both systems, these security aspects and the overall performance (e.g. computation time, amount of signatures that can be generated for a single public key and the size of the signature) are taken into account.
Aiming to improve SPHINCS, SPHINCS$^+$ changed HORST into FORS, changed the L-trees into a single hash function call (also known as tree-less WOTS+ public key compression) uses a different approach in computing the message digest and added tweakable hash functions. Gravity-SPHINCS on the other hand, changed HORST into PORST and removed the bitmasks from the scheme. In addition, Gravity-SPHINCS implemented octopus authentication, secret key caching and batch signing.

The focus of the structural changes and additions in SPHINCS$^+$ lies mostly on improving its security, whereas Gravity-SPHINCS's additions and changes seem more related to improving its computation time and overall performance.
Considering security, SPHINCS$^+$ dealt with all the potential security improvements identified for SPHINCS. Tweakable hash-functions were added to mitigate multi-target attacks, since they ensure that each call for a hash function will independent of each other. To prevent an attacker from choosing the index, the index is computed together with the message digest from the message, making the index verifiable. As solution to the colliding indices in HORST, SPHINCS$^+$ implemented FORS, which is basically HORST, but now the tree is split up in multiple trees, thereby removing the property of colliding indices, since they are now chosen from different trees.
In Gravity-SPHINCS, HORST was changed into PORST to remove the possibility of indices colliding, because in PORST the message digest will continue creating a subset until all the indices in the subset are different from each other. During the PORST process also a verifiable index for the hypertree is generated, making it not possible for an attacker to choose an index. Unfortunately, Gravity-SPHINCS did not include any changes or additions that could be useful in mitigating the multi-target attack. Further-

more, to improve its performance bitmasks are left out of Gravity-SPHINCS's scheme, as it makes the scheme more simple. In a post-quantum world the effect of bitmasks on the scheme's security is negligible, as attacks against (second) preimage resistance and collision resistance are both of $\mathcal{O}(2^{\frac{n}{2}})$. However, in the current, pre-quantum world this results in a drastic loss in security, because bitmasks add collision resistance and (second) preimage resistance is of $\mathcal{O}(2^n)$. So, regarding security SPHINCS$^+$ would be the better option in the current world, as the use of bitmasks allows a higher level of security.
Considering performance, SPHINCS$^+$ replaced the L-trees by a single call for a tweakable hash function, resulting in a more efficient compression of the WOTS+ public keys. In addition, the extra level of security introduced by FORS makes it possible to lower certain parameters (e.g. the amount of trees and the size of the trees), thereby increasing performance. Gravity-SPHINCS's PORST could outperform FORS theoretically in computation time, but it is highly dependent of the choice of parameters which one in the end will perform better, as this choice also heavily affects the level of security. To make PORST perform even better, redundant nodes in authentication paths were removed using octopus authentication. Furthermore, Gravity-SPHINCS removed the bitmasks to increase its performance. However, this made the use of L-trees redundant, yet they are still present in Gravity-SPHINCS and contribute to the total computation time. Nevertheless, Gravity-SPHINCS's method for secret key caching ensures that the signing time and the signature size is reduced every time a signature is produced at the expense of a one-time increase in key generation computation time. Lastly, batch signing in Gravity-SPHINCS is supposed to further reduce the signing time and signature size, yet the implementation is rather specific. To compare the exact effect of batch signing on the schemes, it should be implemented in both Gravity-SPHINCS and SPHINCS$^+$.

In conclusion, considering security SPHINCS$^+$ seems the better option, yet its performance could still be improved. Based on the comparison some additions and changes that were identified could help with the optimization of both schemes. Gravity-SPHINCS brought the addition of secret key caching to the table. Theoretically, implementing secret key caching in SPHINCS$^+$ as well is beneficial for its computation time, under the condition that potential practical problems in handling the top tree can be solved. Gravity-SPHINCS could easily be improved by removing the L-trees, for instance by replacing them with a simple hash function, resulting in less computation time. However, an even better approach would be to replace the L-trees and all the other hash functions used by a tweakable hash function, giving protection against multi-target attacks. Even with these changes Gravity-SPHINCS would still underperform in security in comparison to SPHINCS$^+$, because it does not use any bitmasks.
However, to make definite statements on the performance and security level of SPHINCS$^+$ and Gravity-SPHINCS a theoretical comparison is not sufficient. Component-wise, we have shown that it is possible to look at estimated costs: in this thesis we discussed the influence of PORST versus FORS, as well as including versus excluding bitmasks, on the approximate amount of operations. However, a full comparison should be based on the system as a whole, not by looking at the estimated costs of every single component. Nevertheless, from the component-wise comparison we could conclude that there is a tight balance between the computational costs and the security profits. Therefore, a practical comparison, in which SPHINCS$^+$ and Gravity-SPHINCS are compared for different sets

of parameters, could provide quantitative results on these costs and profits and hence provide more insight in the most optimized form of the system or help to identify further possible bottlenecks.

References

[1] Johannes A Buchmann, Denis Butin, Florian Göpfert, and Albrecht Petzoldt. Post-quantum cryptography: state of the art. In *The New Codebreakers*, pages 88–108. Springer, 2016.

[2] Daniel J Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-lukas Gazdag, Panos Kampanakis, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. Submission to the NIST post-quantum project. pages 1–55, 2017.

[3] Jean-Phillippe Aumasson and Guillaume Endignoux. Gravity-sphincs. Technical report, Tech. rep., National Institute of Standards and Technology, 2017.

[4] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. http://www.nist.gov/pqcrypto.

[5] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3017(Fse):371–388, 2004.

[6] Igor Pletnev, Andrey Erin, Alan McNaught, Kirill Blinov, Dmitrii Tchekhovskoi, and Steve Heller. Inchikey collision resistance: an experimental testing. *Journal of cheminformatics*, 4(1):39, 2012.

[7] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[8] Leslie Lamport. Constructing Digital Signatures from a One-Way Function. *SRI International Computer Science Laboratory*, 94025(October):1–8, 1979.

[9] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 368–397, 2015.

[10] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep*, 2008.

[11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011.

[12] Johannes A. Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the winternitz one-time signature scheme. *IJACT*, 3(1):84–96, 2013.

[13] Andreas Hülsing. W-ots+–shorter signatures for hash-based signature schemes. In *International Conference on Cryptology in Africa*, pages 173–188. Springer, 2013.

[14] Guillaume Endignoux. Design and implementation of a post-quantum hash-based cryptographic signature scheme. *Master's thesis, École polytechnique fédérale de Lausanne*, 2017.

[15] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In *Cryptographers' Track at the RSA Conference*, pages 219–242. Springer, 2018.