Bachelor thesis
Computing Science

Radboud University

Optimizing the NIST Post Quantum Candidate SPHINCS+ using AVX-512

Author:
Dor Mariel Alter
S1027021
dor.alter@student.ru.nl

First supervisor/assessor:
Prof. dr. Peter Schwabe
peter@cryptojedi.org

Second supervisor:
Prof, dr. Bo-Yin Yang
byyangat@iis.sinica.edu.tw

Second assessor:
Prof. dr. Joan Daemen
j.daemen@cs.ru.nl

August 25, 2021

Abstract

Today's cryptography is not strong enough to ensure the security risks we might face in the future. One of the threats to cryptography is quantum computers, and for that reason, NIST created the NIST Post Quantum Cryptography Standardization (PQC), a standardization process for post-quantum schemes. SPHINCS+ is one of the schemes in the alternative candidates in the third round of this project. The main disadvantage of SPHINCS+ is its speed performance. This thesis analyzes ways to improve its performance using parallelism, more precisely using Intel's AVX-512. Using this approach, we can achieve 1.8× or 1.45× speed up for key generation and signing (for the s and the f options of the scheme) and a near 1.3x time improvement for verification in the case of SHA256, compared to the AVX2 version, which is submitted to the NIST Post Quantum Cryptography Standardization. In the case of SHAKE256, we get 2.7× speed up performance for key generation and signing and 1.9× improvement for verification, comparing to the AVX2 version, which is submitted to the NIST Post Quantum Cryptography Standardization. We also present a comparison to the other NIST candidates who are still part of the third round.

Keywords: AVX2, AVX-512, hash, SHA256, SHAKE256, SPHINCS+, NIST PQC, vectorization.

Contents

Chapter 1

Introduction

Today, cryptographic schemes such as signature schemes can be found in most of our daily used devices and online activities. For instance, when we connect to a website that uses the TLS protocol or run a software update, the browser or the app verifies that the web page or update arrived from the right source using such cryptographic shchemes [33]. Nowadays, most schemes rely on a hard problem to solve, such as discrete logarithm or factorization. However, some of those problems will no longer be hard to solve if quantum computers will be used [35].

At the state of the art, quantum computers are still not able to threaten those schemes, however experts believe that within 15-30 years this will no longer be the case [24]. Therefore, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography Standardization process (PQC), where at the end, NIST will announce their recommendation for post-quantum schemes [29]. This project started with 69 candidates in its first round for both public-key encryption and key-establishment algorithms and digital signature algorithms [30]. This list was narrowed down to seventeen candidates for public-key encryption and key-establishment algorithms and nine candidates for digital signature algorithms in the second round [31]. Moreover, at the time of writing this thesis, this competition is in its third round, and there are four finalists for public-key encryption and key-establishment algorithms and five alternative candidates, and there are three finalists for digital signature algorithms and another three alternative candidates [32].

One of the candidates that is part of the alternative candidates for the digital signature algorithms is the SPHINCS+ scheme. SPHINCS+ is a stateless hash-based signature scheme that is believed to be quantum secure [4]. There are quite a few advantages for using SPHINCS+: minimal security assumptions, state-of-the-art attacks are easily analyzed, small key

sizes and reuse of established building blocks. To sum up the advantages and limitations of SPHINCS+ in one sentence, SPHINCS+ is probably the most conservative design of a post-quantum signature scheme. However, it is relatively inefficient in terms of signature size and speed [4]. Due to this disadvantage, SPHINCS+ is not one of the finalists of the third round but rather on the list of alternative candidates.

This leads us to the purpose of this thesis: to investigate how to speed up SPHINCS+ performance. Different parameters can be used for SPHINCS+, which balance between the speed and the signature size, such that the smaller the signature size is, the slower the scheme becomes. So, boosting the performance in terms of speed might be able to improve both issues. Currently, there are mainly two implementations for SPHINCS+, one of them, the so-called optimized implementation, using the Advanced Vector Extensions 2 (AVX2), [17]. The Advanced Vector Extensions (AVX) is a family of vector extensions used in Intel processors, allowing us to perform more operations simultaneously. In 2017, Intel released a new extension called AVX-512 [11], which allows us to perform even more operations at once compared to AVX2. Therefore, we can further optimize the performance of the SPHINCS+ scheme using this AVX-512.

In this thesis, we will try to answer the question: *To what extent can we improve the speed of the NIST candidate SPHINCS+ using AVX-512?*
For that purpose, we propose a new implementation for SPHINCS+ using AVX-512, which can be found at [1]. We will first explain how SPHINCS+ and its lower level schemes work and the improvements in using AVX-512 (chapter 2). After we understand the scheme, we will talk about other NIST candidates, to which we will compare our implementation and another implementation that was used in our code (chapter 3). Then, we will move on to a higher-level explanation of the scheme's implementation using AVX-512 (chapter 4). Afterward, we will derive a lower bound on the amount of operation for the hashes (chapter 5). Thereupon, we will show the results and compare our implementation to other SPHINCS+ implementation and NIST PQC candidates (chapter 6). Finally, in chapter 7, we will conclude this thesis and discuss further impairments of the scheme as future work.

Chapter 2

Preliminaries

2.1 SPHINCS+

SPHINCS+ is a post-quantum stateless hash-based signature scheme designed and implemented by Bernstein, Dobraunig, Eichlseder, Fluhrer, Gazdag, Hülsing, Kampanakis, Kölbl, Lange, Lauridsen, Mendel, Niederhagen, Rechberger, Rijneveld, and Schwabe [4]. SPHINCS+ is a participant in the NIST Post-Quantum Cryptography Standardization Process (PQC), in which NIST will announce which one of the candidates will become their standard scheme [25]. SPHINCS+ is built on the original SPHINCS scheme [6]. However, it has some improvements such as multi-target attack protection, compressed treeless WOTS+ public key, FORS instead of HORST and verifiable index selection [4]. As SPHINCS+ is a hash-based signature scheme, it relies on a hash function. There are three possible versions of hashes used, SHA-256 (SHA2), SHAKE256 (SHA3) and Haraka. However, for the scope of this thesis, we are going to focus on the two mainstream hashes, which are SHA-256 and SHAKE256. before we explain the SPHINC+ scheme, we will first introduce its dependencies, namely WOTS+, FORC and the hyper trees. In the next two subsections, we will look at how each of them works and later, we will see how they are used to create the final SPHINCS+ scheme.

2.1.1 One Time Signature (OTS) Scheme

One Time Signatures are schemes that can be used to sign only one message. These schemes are the basis of some of the more complicated signature schemes, therefore, it is important first to understand them. The simplest example is the 0-bit message signature, in which the sender does not send a specific message, but the receiver knows that the message received is from the sender. A real-life example of this was the Yo App [27], an app that allowed the users to only send the message "Yo" to each other. It works as follows: the sender uses a secret key and publishes its hash as their public

key. Then, after sending the message, the sender also sends the private key. The receiver hashes the private key, compares it to the sender's public key and if they match, the sender is authenticated. This idea, although rather simple, is the building block for more complex schemes. Let us also consider the 1-bit message scheme, which works as follows: the sender generates two random values (sk_0, sk_1) which are the private key, then hashes each one of them and publishes the result, as their public key. Then, when the sender wants to send a message, they send either sk_0 or sk_1 for the bit to be 0 or 1, respectively. Now, the receiver can hash the message and compare it to the public key, which will tell them if the value is zero or one and that the signature on the 1-bit message is verified. After we understand the simple cases of OTS, we can move on to more complicated schemes used in SPHINCS+.

2.1.2 WOTS+

WOTS+ is a one-time signature (OTS) scheme designed by Hülsing [16], which is an improved version of the famous Winternitz OTS (WOTS) scheme [8]. The main idea of both schemes is to use the so-called chain functions starting from the private key up to the public key. Those chain functions are recursively applied to their output several times, until we get the final value, whence their name as chain functions. Next, we will explain how WOTS works and afterward, we will see how it matured to WOTS+.

WOTS

We start by looking at how WOTS works [8]. WOTS is a one time signature scheme. In this scheme, we sign messages, those messages could be either fixed length messages or the hash values of the messages with a fixed length. In WOTS, there are two keys (public and private), consisting of l pseudo-random values of w bits each, where w is the base used, which is the fixed length we mentioned above.

To compute the length of l, there are two other values that needed to be calculated first. Those being l_1, the amount of blocks we divide the message into and l_2 the length of the checksum. Then l is the sum of l_1 and l_2. Given the length of the message l_m and base w, l_1 and l_2 are computed as follows, $l_1 = \lceil \frac{l_m}{\log_2(w)} \rceil$ and $l_2 = \lfloor \frac{\log_2(l_1 \cdot (w-1))}{\log_2(w)} \rfloor + 1$.

For simplicity, we are going to use base 256 and message length 256 bits as well in our explanation and figures, which gives us $l_1 = 32$ and $l_2 = 2$. Hence the private key is denoted by Prk_0 to Prk_{33} and the public key is denoted by Pub_0 to Pub_{33}, where each value is 256-bits long. The function H is defined as follows $H : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$, this function can be seen as a

non-compressive cryptographic hash function. The public key is generated by applying the hash function H 255 times on each 256-bit value from the private key and then concatenating the results, as shown in Figure 2.1.

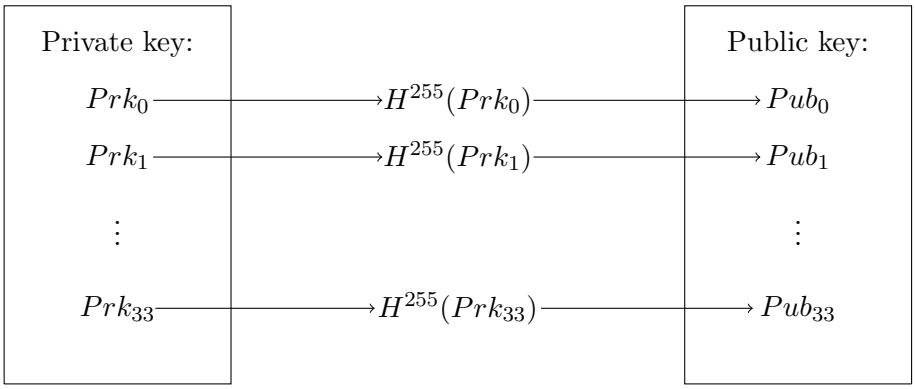

Figure 2.1: WOTS+ generation of the public key using the private key.

To elaborate upon this idea and better understand how to perform key generation for WOTS, a pseudocode is also given for that algorithm, Algorithm 1. Unlike in the case of the figures, the pseudocode does not use a specific base, such as 256, but rather w as the base value.

Algorithm 1 Key generation for WOTS

```
1: function KEYPAIR_WOTS( )
2:     sk ← $(sk_1, sk_2, ..., sk_l)
3:     for i = 0 to l do
4:         pk_i = H^{w-1}(sk_i)
5:     end for
6:     return sk pk
7: end function
```

Notice that the $ represents generating values uniformly at random and H is the non-compressive cryptographic hash function mentioned above.

After we got a notion of how the keys are generated, we look at how to sign a message. Let message M be a message with a length of 256 bits, otherwise, apply a hash function to turn it to 256 bits. We break M into 32 values denoted by m_i, as it is 256 bits, we know that each element will be 8-bits long. Then, for each m_i we compute $H^{255-m_i}(Prk_i)$. Unfortunately, this is not enough for signing the message as it is still prone to forgery. For example, given a message and its signature $m = (6, m_1, \ldots, m_{31})$ an attacker can perform another iteration of the hash resulting in a new valid signature for a different message, $m = (7, m_1, \ldots, m_{31})$. To deal with this, the scheme

introduces a checksum for each m_i. We compute the checksum across all values of m_i as follows: $c = \sum_{i=0}^{32}(256 - 1 - m_i)$. Then we can write c as two values c_0 and c_1 and compute the hash chain function H on them. So, if we increase one of the m_i, the values of the c_i decrease and the signature is no longer valid. An illustration of this procedure is shown in Figure 2.2.

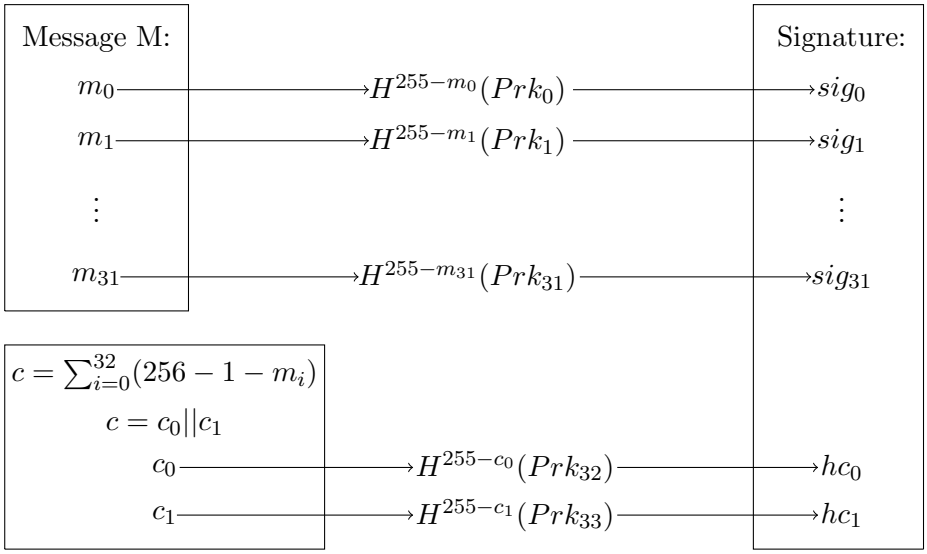

Figure 2.2: WOTS+ singing a message M with 256 bits length.

Again for better understanding, a pseudocode for that algorithm, Algorithm 2 is also provided.

Algorithm 2 Signing a message using WOTS

```
1: function SIGNING_WOTS(M, sk)
2:     CS ← c_0||c_1||...||c_{l_2} = Σ_{i=1}^{l_1}(w − 1 − M_i)
3:     for i = 0 to l_1 do
4:         sig_i ← H^{w−1−m_i}(sk_i)
5:     end for
6:     for i = 0 to l_2 do
7:         hc_i ← H^{w−1−c_i}(sk_{i+32})
8:     end for
9:     return sig, hc
10: end function
```

Note that bars (||) represent concatenation.

This works as we know that M is 256 bits long, so if we break M into 32 values, we will get 8-bit values (as $256/32 = 8$) and we know that $2^8 = 256$. So, 255 is the highest value we can get for m_i, which means that we will

always get a non-negative value for the expression of H^{255-m_i}.

Verifying is as simple as taking each sig_i value and applying H on it m_i times as shown in Figure 2.3 and then comparing it to the public key and verifying the checksum in the same manner.

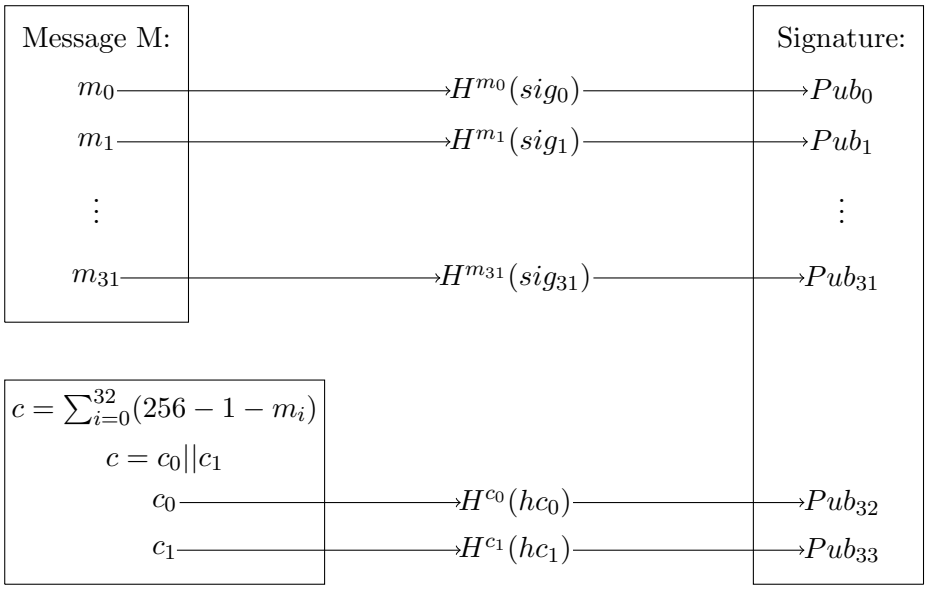

Figure 2.3: WOTS+ verifying a message M with 256 bits length.

The algorithm for verifying will return True if the signature is valid and False otherwise. The following pseudocode for verification, Algorithm 3 is used.

Algorithm 3 Verifying a signature using WOTS

function VERIFYING_WOTS(M, sig, pk, hc)
  for $i = 0$ to l_1 **do**
    $ver_i \leftarrow H^{m_i}(sig_i)$
  end for
  $c_0||c_1||...||c_{l_2} \leftarrow \sum_{i=1}^{l_1}(w - 1 - M_i)$
  for $i = 0$ to l_2 **do**
    $ver_{l_1+i} \leftarrow H^{c_i}(hc_i)$
  end for
  if $pk == ver$ **then**
    return True
  else
    return False
  end if
end function

From WOTS to WOTS+

Since its first introduction, WOTS has been modified and improved upon, while the most noticeable improvement is WOTS+ [16]. Similar to WOTS, WOTS+ also has several versions. Unfortunately, they share the same name, although they have slight differences. As a result, the WOTS+ used in SPHINCS+ is different from the original version. The WOTS+ used in SPHINCS+ introduces two additional parameters which turn the hash function into a tweakable hash function compared to the WOTS discussed above. This notion is beyond the scope of this thesis, but more information can be found in [2]. For this thesis, we can consider WOTS+ as a version of WOTS which takes two extra parameters [5], those being the position of the node in the tree, called address, and a public seed while the signatures have the same procedures as WOTS.

2.1.3 Hypertree and FORS

As the name might suggest, WOTS+ alone is not sufficient for a signature scheme as it is a one-time signature scheme. This means that every time we want to sign a message, we will have to generate a new key pair, which is inconvenient if we want to sign a large number of messages under the same key at the same time. Therefore, we combine WOTS+ with hypertrees in order to address this issue. However, before we can talk about hypertrees, we will first explain how trees can be used to sign messages. For that purpose, we will discuss Merkle trees in the next part.

Merkle tree

A Merkle tree is a type of hash tree used to sign and verify messages, which was designed by dr. Merkle [23]. One of the main disadvantages of schemes that use this method is being stateful, meaning that we need to save the state in which our tree is. The main idea of Merkle tree is to have a tree with a root (gray circle node), nodes (circle nodes) and leaf elements (rectangle node) as can be seen in Figure 2.4. Now, the root of this tree is designated as the public key. The leaf elements are the public key, Y_i, and private key, X_i, of the given signature scheme. The nodes are the hashed value of the concatenation of their children. Notice that when talking about height, the leaves are not taken into consideration because they are not an actual part of the tree, but rather an initializer for the nodes. Therefore, in the tree below Figure 2.4, they are not represented as nodes.

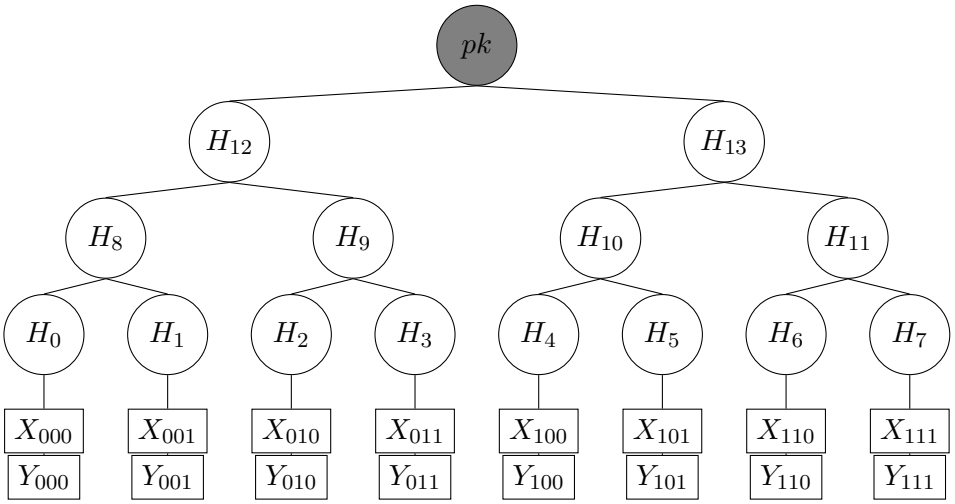

Figure 2.4: Merkle tree of height 3

To sign a message M using the Merkle tree, we will first use the signature scheme given in the leaf, which gives us $sig' = sign'(M, X_i)$. However, unlike in the previous scheme, just publishing the signature, sig', and the public key, whether it is the public key of the leaf (Y_i) or the public key of the scheme (pk), is not enough. In order to get the public key pk from a leaf used to sign it, we need the path from that leaf to the root. Therefore, we will include an additional parameter, $AUTH$ in the signature, being the concatenation of all nodes in the path leading from the leaf to the root. So, the signature will look like this $sig = (i, sign(M, X_i), Y_i, AUTH)$. As an example, assuming we want to sign message M using X_{010}, we first sign M using X_{010} so we have $sig'_{010} = sign'(M, X_{010})$, then we add Y_{010}, the public key of X_{010}, then we add the index of the leaf 010 and finally the authentication path, which gives us $sig = (010, sig'_{010}, Y_{010}, (H_8||H_3||H_{13})$.

To verify sig, we will first use the leaf scheme to verify that $sign'(M, X_i)$ indeed gives Y_i and then use Y_i together with the authentication path to reconstruct the public key. We compare it to the known public key pk and if they match, then we know that the signature is valid.

The idea of stateful here comes from the fact that we also need to update the i for every signature and ensure the same i's do not repeat. If we do not, we risk using the same leaf twice, which might reveal information about the message or might allow attackers to create a forgery, depending on the scheme we use in the leaf. This is due to the fact that the scheme used in the leaves is a one time signature scheme, which means that it must only be used once.

Hypertree

Now, after having a better understanding of the way trees can be used for signing, we can look at hypertrees schemes, which is equivalent to the $XMSS$ scheme [9], where the main idea is to use a tree as the constructor for the scheme. Afterward, we will extend upon this idea to get a multi-tree setting rather than one big tree, which is equivalent with $XMSS^{MT}$ [15].

So, as explained earlier, we do not want to use the same leaf twice, and we want to have enough leaves to sign more messages. The straightforward approach to solve this issue would be to create a signal tree called a hyper tree that is large enough such that there would be sufficient leaves. However, as we increase the height of a tree, we need to consider that we increase the cost of using and constricting this tree. By doing so, we will get 2^h leaves, where h is the height.

In order to improve upon this issue, we use multi hypertree, thus reducing the cost, by using multiple smaller trees instead of a single big tree. This approach will give us $a \times 2^{h'}$ leaves, where a is the number of trees, and h' is the height of those trees. In this case, we can have the same amount of leaves, but as we have multiple unrelated trees, on which we can work in parallel, it increases the efficiency of the scheme.

FORS

Another method that can be used to reduce the impact of using the same leaf twice is to use a few time signature scheme (FTS), rather than an OTS. This way, we will ensure that we will not leak any vital information even if

we use the same leaf twice. After discussing how trees could be used for OTS signing, we will look at their use for FTS signing. The scheme used for that purpose is Forest of Random Subsets (FORS), where the main idea is not to use only one tree, but rather a forest of trees.

Let us look at how FORS works. As in the case of the Merkle tree, we have a public key associated with the sender. However, unlike in the Merkle tree scheme, it is not the root of a single tree, but the concatenation of the roots of all the trees. This public key is computed before sending any message and is published by the sender. To sign a message, we need it to be the same size as the block size, $k \times a$, which means that we need k strings of length a bits. Then, we do the following: we have k trees, and each of them has 2^a leaves, so each value of a string will be interpreted as a leaf in one of the k trees. Afterward, as we have seen in the Merkle tree above, we can sign by using the node representing one string of a together with the authentication path to the root. So, our signature will be the leaf represented by the value of a and the authentication path to the root of that tree, and we can concatenate the results and send it. To verify the signature, we need to reconstruct the k roots of the trees and compare it to the value of the roots given in the public key.

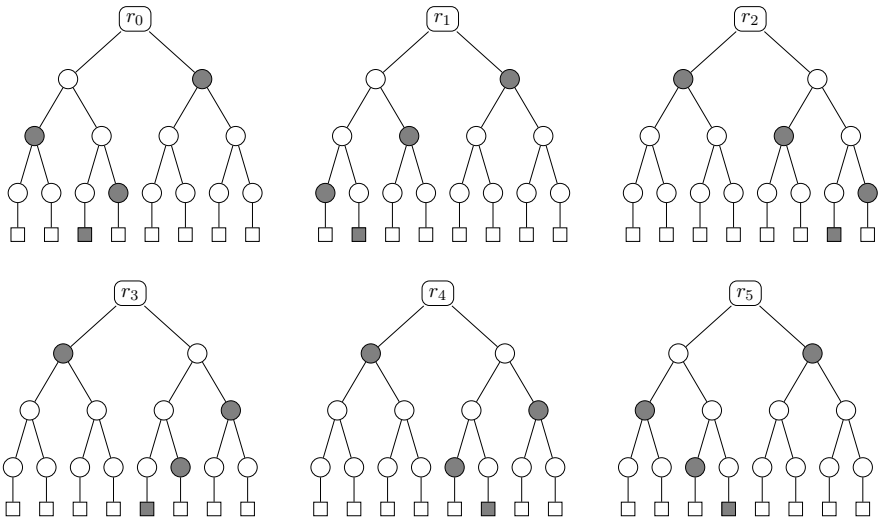

Figure 2.5: An illustration of a FORS signature with $k = 6$ and $a = 3$, for the message 010 001 110 100 101 011.

In Figure 2.5 we can see an example of a signature using FORS. In this case, we see that we have six trees ($k = 6$), and the tree's height is 3 ($a = 3$). Thus, a message should be of length $3 \times 6 = 18$ bits. In this case, we want to sign the following message 010 001 110 100 101 011, which is indeed 18 bits

long. So, we split it into six values of 3 bits each as a is 3. The signature will consist of the gray leaf (representing the leaves corresponding to the six values we split m into) and the gray nodes. As can be seen in Figure 2.5 for each tree, we can construct its root using the gray nodes, thus, verifying the signature. We can also see that the message m is not the value of the leaf node, but rather a pointer towards a specific node, meaning that the values of the roots and leaves are not dependent on the message. But instead, the leaves are chosen by the message, which means that we can compute the leaves when generating the public and private keys.

2.1.4 SPHINCS+

Finally, after discussing the smaller schemes used in SPHINCS+, we can discuss how the scheme works as a whole.

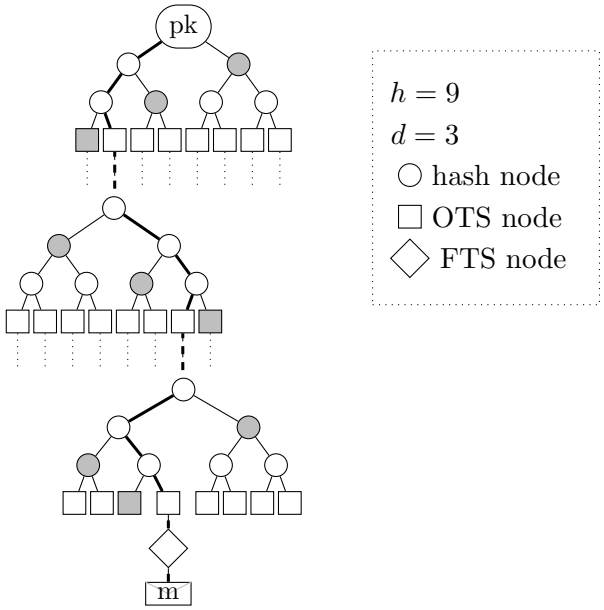

Figure 2.6: Illustration of a small SPHINCS+ structure, Figure reprinted from [2].

In Figure 2.6 we can see an illustration of how we sign a message m. However, before we can sign any message, we need to generate the SPHINCS+ key pair. Like any other public-key scheme, SPHINCS+ consists of a public key and a private key. The public key is composed of two n-bit values: the root node of the top trees in the scheme and a random public seed. The private key is also composed of two n-bit values, two random seeds, one for WOTS+ and FORS and another for the randomized message digest. As we discussed in FORS, we do not generate the leaves when signing a message,

but rather we create them in the key generation process. After generating these keys and understanding how the tree is created, we can discuss how we sign a message. We sign a message using FORS, and then we sign the FORS pubic key using WOTS+, which gives us a node in the lowest tree. Additionally, we add the authentication path leading to the top root from that node, giving us the SPHINCS+ signature. In order to verify the signature, we use the authentication path, and we keep going up the trees until we get to the top root node, which is the public key. Then, we simply compare if the top root we got is identical to the known public key and if so, the message is authenticated. There are two important aspects to notice here: first, the whole tree is not present in the verification process, but only the nodes needed to get from the signed node to the root (so, for instance, if the node is on the right branch of the tree, considering its first two children, we only need the left child and not the whole tree resulting from that child). Secondly, the FORS public key and WOTS public keys are never explicitly verified, but rather used to construct the top root of the tree. Those are important aspects that reduce the time and space complexity of the scheme.

2.1.5 SPHINCS+ Instantiation

We proceed to discuss the different parameter sets and implementations we use within it. The first two implementations we can talk about are the simple and robust versions. Each focuses on a different aspect of the scheme. While the simple version is more about speed, the robust version is more about security, in the sense that we have fewer security assumptions, so those two are two versions of the SPHINCS+, which balance the scheme's security and speed. Next, we have six options for the scheme, which uses different parameters to provide different security strengths as required by NIST. We are going to represent those options in the following table Table 2.1 where we have the following abbreviations, n - hash output length, h - the height of the hypertree, d - number of subtree layers, $\log(t)$ - FORS tree height, k - Number of FORS Trees and w - Winternitz parameter.

	n	h	d	$\log(t)$	k	w	security bits	pk bytes	sk bytes	signature bytes
SPHINCS+-128s	16	63	7	12	14	16	133	32	64	7,856
SPHINCS+-128f	16	66	22	6	33	16	128	32	64	17,088
SPHINCS+-192s	24	63	7	14	17	16	193	48	96	16,224
SPHINCS+-192f	24	66	22	8	33	16	194	48	96	35,664
SPHINCS+-256s	32	64	8	4	22	16	255	64	128	29,792
SPHINCS+-256f	32	68	17	9	35	16	255	64	128	49,856

Table 2.1: SPHINCS+ Instantiation.

2.2 Hash Functions

As mentioned above, SPHINCS+ is a hash-based signature scheme, which means that the main operations in this scheme are hashing. Therefore, it is important to explain how the hash functions used in this scheme work. In this thesis, we will consider the implementations of SPHINCS+ using both SHA2 and SHA3, more specifically SHA256 and SHAKE256, so in the next part, we will explain how each of those hashes works.

2.2.1 SHA256

The first hash function we will discuss is SHA256, which is part of the SHA2 hash function family. As the name suggests, it is a more secure scheme comparing to its previous version of SHA1. This function is a standard by the National Institute of Standards and Technology (NIST) and was published in FIPS 180-2 [26], and since then, it was and still is widely used.

Now, we discuss how this function works. Given an input message, we divide it into values of 512-bit length and hash each block separately. Therefore, we are going to focus on the operations we perform on a single block. If we do not have 512 bits, we will pad the message to end up with a 512-bit value. The first thing we need to consider is the 8 constants that will initialize the hash values, let us note them by h_i where $i \in \mathbb{N} \wedge i \in \{0, 1, ..., 7\}$. We obtain those constants by taking the 32 first bits of the fractional parts of the square roots of the first eight primes. Those values are usually hardcoded in the code. The values, represented in hexadecimal, are as follow:

$h_0 := 0x6a09e667$
$h_1 := 0xbb67ae85$
$h_2 := 0x3c6ef372$
$h_3 := 0xa54ff53a$
$h_4 := 0x510e527f$
$h_5 := 0x9b05688c$
$h_6 := 0x1f83d9ab$
$h_7 := 0x5be0cd19$

In addition to those constants, there are another 64 constants consisting of 32 bits each. We are going to place them in an array K, which has 64 entries of 32 bits each. Those constants are the first 32 bits of the fractional parts of the cube roots of the first 64 primes. Again, those constants are hardcoded into the code. The values of the constants, represented in hexadecimal, are:

$0x428a2f98$ $0x71374491$ $0xb5c0fbcf$ $0xe9b5dba5$ $0x3956c25b$ $0x59f111f1$
$0x923f82a4$ $0xab1c5ed5$ $0xd807aa98$ $0x12835b01$ $0x243185be$ $0x550c7dc3$
$0x72be5d74$ $0x80deb1fe$ $0x9bdc06a7$ $0xc19bf174$ $0xe49b69c1$ $0xefbe4786$
$0x0fc19dc6$ $0x240ca1cc$ $0x2de92c6f$ $0x4a7484aa$ $0x5cb0a9dc$ $0x76f988da$
$0x983e5152$ $0xa831c66d$ $0xb00327c8$ $0xbf597fc7$ $0xc6e00bf3$ $0xd5a79147$
$0x06ca6351$ $0x14292967$ $0x27b70a85$ $0x2e1b2138$ $0x4d2c6dfc$ $0x53380d13$
$0x650a7354$ $0x766a0abb$ $0x81c2c92e$ $0x92722c85$ $0xa2bfe8a1$ $0xa81a664b$
$0xc24b8b70$ $0xc76c51a3$ $0xd192e819$ $0xd6990624$ $0xf40e3585$ $0x106aa070$
$0x19a4c116$ $0x1e376c08$ $0x2748774c$ $0x34b0bcb5$ $0x391c0cb3$ $0x4ed8aa4a$
$0x5b9cca4f$ $0x682e6ff3$ $0x748f82ee$ $0x78a5636f$ $0x84c87814$ $0x8cc70208$
$0x90befffa$ $0xa4506ceb$ $0xbef9a3f7$ $0xc67178f2$

Now that we have established all the constants, it is time to start operating on the input. However, before that, let us define the following functions and their notation: *32-bit add* ($+$), which adds two values of 32 bits, *bitwise flip* (*not*), which flips the 32-bit value given to it, *bitwise or* ($\vee$), which takes two 32-bit values and returns the bitwise or between them *bitwise and* ($\wedge$), which takes two 32-bit values and returns the bitwise and between them, *rotate right* ($ROTR^n(x) = (x \gg n) \vee (x \ll w - n)$, where w is the size of the word in bits), which takes a 32-bit value, x, and an integer n and rotates it to the right n times, *rotate left* ($ROTL^n(x) = (x \ll n) \vee (x \gg w - n)$, where w is the size of the word in bits), which takes a 32-bit value, x, and an integer n and rotates it to the left n times, *shift right* ($x \gg n$), which takes a 32-bit value and an integer n and shifts it to the right n times and *shift left* ($x \ll n$), which takes a 32-bit value and an integer n and shifts it to the left n times.

The following steps happen for each chunk of 512 bits. We start by initializing an array W with 64 entries, each being 32 bits. The first 16 elements ($W[0..15]$) are our 512 bits input and the rest, for now, will be 0. Then, we perform the following:

Algorithm 4 Sigmas computation, initializing W

function SIGMAS(W)
  for $i = 16$ to 63 **do**
    $x_1 \leftarrow i - 15$
    $x_2 \leftarrow i - 2$
    $s_0 \leftarrow ROTR^7(W[x_1]) \oplus ROTR^{18}(W[x_1]) \oplus (W[x_1] >> 3)$
    $s_1 \leftarrow ROTR^{17}(W[x_2]) \oplus ROTR^{19}(W[x_2]) \oplus (W[x_2] >> 10)$
    $W[i] \leftarrow W[i-16] + s_0 + w[i-7] + s_1$
  end for
end function

Next, we are going to copy the values of the constants of h_i to new variable denoted from a to h such that $a = h_0,\ b = h_1$ and so on. Then we are going to perform the following:

Algorithm 5 SHA256 rounds

function ROUNDS($a, b, c, d, e, f, g, h, W, K$)
  for $i = 0$ to 63 **do**
    $S_0 \leftarrow ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$
    $S_0 \leftarrow ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$
    $S_1 \leftarrow ROTR^6(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e)$
    $ch \leftarrow (e \wedge f) \oplus (not(e) \wedge g)$
    $maj \leftarrow (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
    $temp_1 \leftarrow h + S_1 + ch + K[i] + W[i]$
    $temp_2 \leftarrow S_0 + maj$
    $h \leftarrow g$
    $g \leftarrow f$
    $f \leftarrow e$
    $e \leftarrow d + temp_1$
    $d \leftarrow c$
    $c \leftarrow b$
    $b \leftarrow a$
    $a \leftarrow temp_1 + temp_2$
  end for
end function

Next, we are going to add the 8 constants to the values we obtained for a, b, c, d, e, f and g one last time:

$h0 = h_0 + a$
$h1 = h_1 + b$
$h2 = h_2 + c$
$h3 = h_3 + d$

$h4 = h_4 + e$
$h5 = h_5 + f$
$h6 = h_6 + g$
$h7 = h_7 + h$

Finally, we concatenate all the h_i and we got the hashed value.

2.2.2 SHAKE256

The second hash function that we will discuss is SHAKE256, a hash function from the function family SHA3, or more precisely, the Keccak family. This function is an improvement to the SHA2 family of functions in terms of security, but not only. Unlike the previous versions of SHA, which were fixed digest length, this version allows us to modify the output size to any number of bits. This function became a standard by the National Institute of Standards and Technology (NIST), published in FIPS 202 in 2015 [28] and it is the latest version of the SHA family of functions.

The first aspect we will discuss while talking about SHAKE256 is the parameters. In order to decide on the size of the state, we have parameter $b = 25 \times 2^l = \{25, 50, 100, 200, 400, 800, 1600\}$. In the case of SHAKE256, the state size is 1600. After we have chosen a state size, we determine the number of rounds, $Rounds = 12 + 2 \times l$. So, for instance, for a state size of 1600 like in SHAKE256, we have $l = 6$, so we have 24 rounds. Another important parameter is the w parameter, which is equal to $w = b/25$. We will use this parameter when writing the dimensions of the state, which are 5 by 5 by w. Next, we can talk about the rate denoted by r and the capacity denoted by c, the values are such that $r + c = b$. Those two variables are different for each member of the SHA3 family and in our case, for SHAKE256, the values are $r = 1088$ and $c = 512$. As we can see, adding those two will give us 1600, which is our state size.

Before we dive deeper into understanding how SHAKE256 works, we will see a more general diagram that shows us how Keccak works. Keccak is also known as a sponge function, this is due to the way we can split it into two parts, the absorbing phase and the squeezing phase. In Figure 2.7 below, we are going to look at how the absorbing phase works.

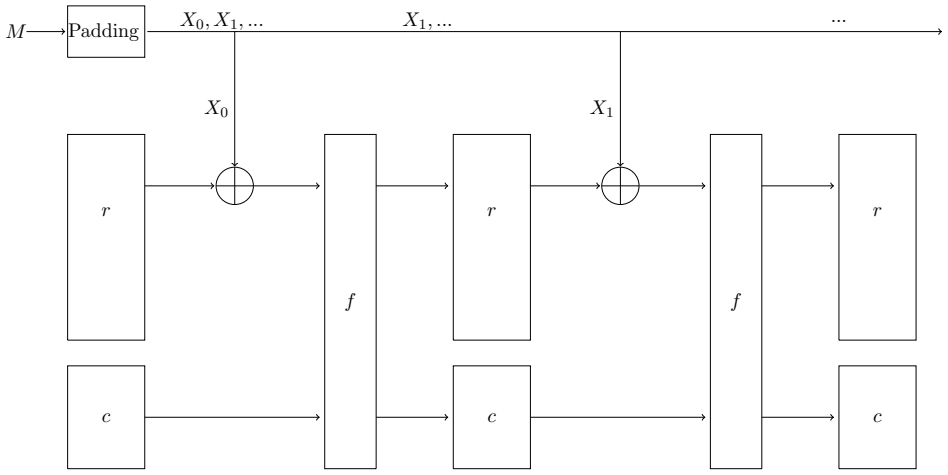

Figure 2.7: Illustration of the absorbing phase of Keccak.

In Figure 2.7 we can see the absorbing phase of a message M, which is the first phase in its hashing. The first step is to add padding, such that the length is divisible by r. Then we XOR r bits from the message with the values of r. Notice that the initial values of r and c are 0's. Then we add to those r bits the additional c and perform the permutation operations on it, which are called f, representing the rounds discussed earlier. We will perform these operations until we have covered all the input from message M. This is known as the absorbing phase, because using the input M we create, "absorbed", our state and we repeat this, until we have "absorbed" the whole input. Afterward, we will "squeeze" the state into output. Next, we are going to look at how the squeezing phase works.

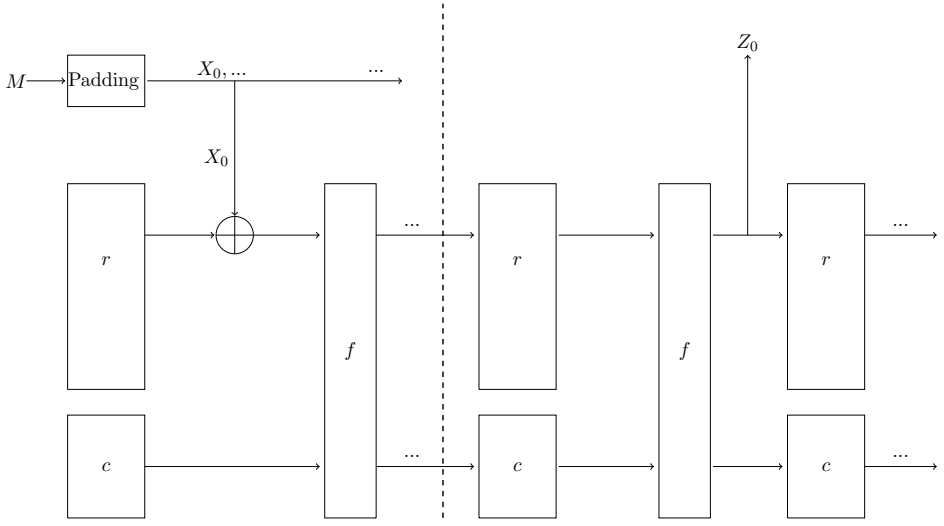

Figure 2.8: Illustration of the squeezing phase of Keccak.

On the left side, we can see the absorbing phase we talked about above and on the right hand, we can see the squeezing phase. In the squeezing phase, we take the state (r and c) that we got from the absorbing phase, run the f function on it again, and take the resulted r to be the output. As mentioned above, we can get the output size to be as we wish. This is done by squeezing until we get the amount of Z's needed. If we want less output than the length of r, we will take the first x-bits out of the Z_0, where x is the length needed. The idea for the name squeezing is because after we absorbed the input, we can "squeeze" as much output as we want from the state.

After understanding the absorption and squeezing phases, we need to comprehend how the f function from above works. As we can see from the Figures 2.7 and 2.8 above, it takes as input a state which in the case of SHAKE256 is 1600 bits, and it returns a state of the same length. This state is an array of b bits with dimension 5 by 5 by w and it is going to be denoted by A, while the output state will be denoted as A'. Now, the f function is built up out of 5 smaller functions denoted by θ, ρ, π, χ and ι. Next, we will explain how each of them works and then see how we can combine them to create one round of function f, as explained in [28]. Recall from above that we mentioned that depending on the chosen value of l, we have different number of rounds in each call to f. In the next definitions we are going to denote the array as an array of x by y by z, where we know that $x \in \{0, 1, 2, 3, 4\}, y \in \{0, 1, 2, 3, 4\}$ and $z \in \{0, 1, 2, ..., w - 1\}$. Another term we will use throughout those definitions is the lane, lane is a sub-array of w bits with constant x and y coordinates. Now that we have covered all the terms and notations, we can explain how the smaller functions work.

First is the θ function, the idea is to XOR each bit in the state with the parities of two columns in the state. We do that as follows:

Algorithm 6 The θ function, SHAKE256

```
1: function THETA(A, w)
2:     for x = 0 to 5 do
3:         for z = 0 to w do
4:             C[x, z] ← A[x, 0, z] ⊕ A[x, 1, z] ⊕ A[x, 2, z] ⊕ A[x, 3, z] ⊕ A[x, 4, z]
5:         end for
6:     end for
7:     for x = 0 to 5 do
8:         for z = 0 to w do
9:             D[x, z] ← C[(x − 1) mod 5, z] ⊕ C[(x + 1) mod 5, (z − 1) mod w]
10:        end for
11:    end for
12:    for x = 0 to 5 do
13:        for y = 0 to 5 do
14:            for z = 0 to w do
15:                A′[x, y, z] ← A[x, y, z] ⊕ D[x, z]
16:            end for
17:        end for
18:    end for
19:    return A'
20: end function
```

Next, we are going to talk about ρ, the idea behind this function is to rotate the bits of each lane by a length, called the offset, which depends on the fixed x and y coordinates of the lane. We perform that as follows:

Algorithm 7 The ρ function, SHAKE256

```
1: function RHO(A, w)
2:     for z = 0 to w do
3:         A′[0, 0, z] ← A[0, 0, z]
4:     end for
5:     (x, y) ← (1, 0)
6:     for t = 0 to 23 do
7:         for z = 0 to w do
8:             A′[x, y, z] ← A[x, y, (z − (t + 1)(t + 2)/2) mod w]
9:         end for
10:        (x, y) ← (y, (2x + 3y) mod 5)
11:    end for
12:    return A'
13: end function
```

Now, we are going to talk about the π function. In this function, we are going to rearrange the positions of the lanes. We are performing this as follows:

Algorithm 8 The π function, SHAKE256

```
1: function PI(A, w)
2:     for x = 0 to 5 do
3:         for y = 0 to 5 do
4:             for z = 0 to w do
5:                 A'[x, y, z] ← A[(x + 3y) mod 5, x, z]
6:             end for
7:         end for
8:     end for
9:     return A'
10: end function
```

Next on the list is the χ function, the effect of this function is to XOR each bit with a nonlinear function of two other bits in its row. We perform this as follows:

Algorithm 9 The χ function, SHAKE256

```
1: function CHI(A, w)
2:     for x = 0 to 5 do
3:         for y = 0 to 5 do
4:             for z = 0 to w do
5:                 A'[x, y, z] ← A[x, y, z] ⊕
6: ((A[(x + 1) mod 5, y, z] ⊕ 1) · A[(x + 2) mod 5, y, z])
7:             end for
8:         end for
9:     end for
10:    return A'
11: end function
```

Note that the dot $(\cdot)$ represents integer multiplication.

Finally, we got to the last function ι, but before we can define this function we are going to define another function which we will use in the definition of ι. Let us call this function rc. rc receives an integer denoted by t and returns a one-bit value. The function rc is defined as follows:

Algorithm 10 The rc function, SHAKE256

```
1: function RC(t)
2:     if t mod 255 == 0 then
3:         return 1
4:     end if
5:     R ← [1,0,0,0,0,0,0,0]
6:     for i = 1 to t mod 255 do
7:         R ← 0 || R
8:         R[0] ← R[0] ⊕ R[8].
9:         R[4] ← R[4] ⊕ R[8].
10:        R[5] ← R[5] ⊕ R[8].
11:        R[6] ← R[6] ⊕ R[8].
12:        R = Trunc_8[R]
13:    end for
14:    return R[0]
15: end function
```

Note that the bars (||) represent concatenation and $Trunc_i(x)$ is a function that returns the first and the $i-1$ (last) bits of x.

Now, we can define the ι function, the idea of this function is to modify some of the bits of lane $(0,0)$ in a manner that depends on the round index i_r. This means that, unlike the previous function, this function apart from the state also gets the round index. We define the function as follows:

Algorithm 11 The ι function, SHAKE256

```
1: function IOTA(A, i_r)
2:     for x = 0 to 5 do
3:         for y = 0 to 5 do
4:             for z = 0 to w do
5:                 A'[x,y,z] ← A[x,y,z]
6:             end for
7:         end for
8:     end for
9:     RC ← [0,0,..,0] (w times)
10:    for j = 0 to len(i_r) − 1 do
11:        RC[2^j − 1] ← RC(j + 7 ∗ i_r)
12:    end for
13:    for z = 0 to w do
14:        A'[0,0,z] ← A'[0,0,z] ⊕ RC[z]
15:    end for
16:    return A'
17: end function
```

After we have covered all the functions that make up a round, we can properly define a round. Each round is given a state array A and a round index i_r, so we define a round as:
$round(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r)$.

Now, after we understand how a round works and how the absorption and squeezing work, we have a complete image of how SHAKE256 works. We first absorb the input as explained above, then perform the round function 24 times and squeeze the output to a 256-bit value.

2.3 Intel Advanced Vector Extensions 512

Today, we can find different applications in non-computer-science-related fields which require the use of supercomputers [34] [38]. As a result, scientists are looking at different methods to increase the performance of computers, which will allow us to do more operations per unit of time (let it be minute, second or cycle). In order to achieve such improvements, it is a common practice to use parallelism, meaning executing multiple tasks at the same time. There are three main methods of parallelism in today's architecture and software: multi-core [14], multi-threading [37] and single instruction multiple data vectorization. Each of those methods works on a different CPU level and allows it to perform multiple tasks at once. Before we start discussing Intel Advanced Vector Extensions 512, we first discuss the parallel approach it uses.

2.3.1 Single Instruction Multiple Data

AVX-512 uses the Single instruction multiple data (SIMD) vectorization parallel approach. In this case, the idea is to increase the size of the registers such that instead of containing a single value, it will contain a vector consisting of multiple values. Operations on this vector will be performed the same way as operations on one register, as all the values are in one register. For instance, let us say we have eight floats, each of 32 bits and we want to add four of them to the other four. If we do that in sequence, having two registers, each with a size of 32 bits, it will take us four operations to complete the task. However, if instead of two registers of 32 bits, we will have two registers with 128 bits, then we can fill each register with four floats values and add those two registers. So, we will only need one operation in order to add all the floats, instead of four. This way, we can work in parallel within the same thread, using just the registers available.

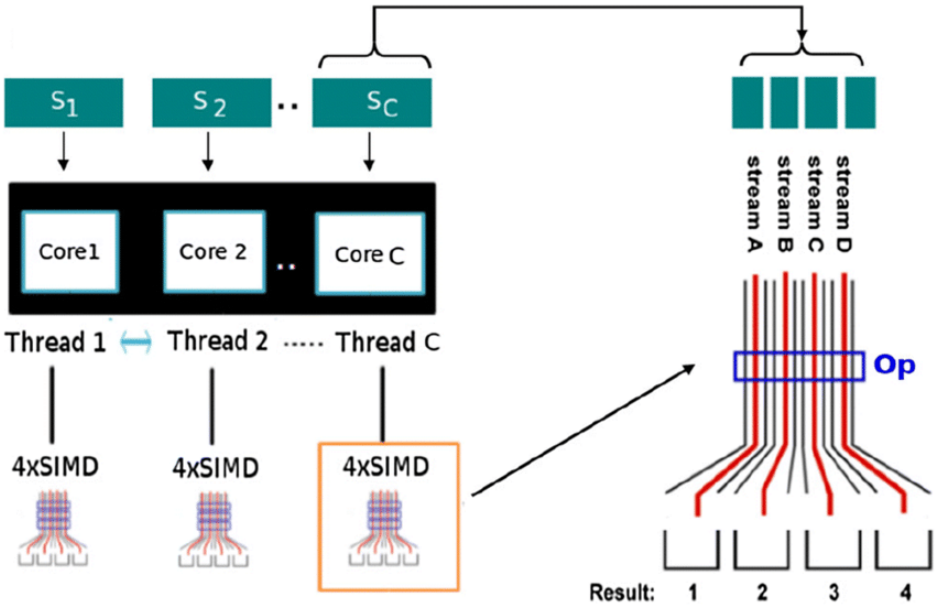

Figure 2.9: An illustration of different parallel approaches, Figure reprinted from [20].

For a better understanding of the differences between the approaches the following example is given. In Figure 2.9 we can see an illustration of all three approaches on one task. Let us say we have Task T which can be broken into C different tasks denoted by S_i, where $i \in \mathbb{N} \wedge i \in \{1, 2, ..., C\}$ such that each core, $Core_i$ is assigned one task to perform. After that, we can further break down each task S_i into another C sub-tasks $Thread_i$, which can be done by each thread and communicated within the same core to result in the final task S_i. Then, each thread needs to perform different operations on different variables, which can again be done in parallel. As we can see in Figure 2.9 on the right, each operation is carried out on four different streams, which is the same idea as operating on the floats, as discussed above.

2.3.2 Intel Advanced Vector Extensions 512

Now, that we know the different parallelism methods, we can talk about Intel Advanced Vector Extensions 512, also known as AVX-512. AVX-512 is an extension that implements the SIMD vectorization approach. However, before we can understand the benefits of AVX-512, we will first look at the previous implementation of AVX. The first implementation in this series is AVX, which was an improvement for the known SSE, AVX has a register size of 256 bits and limited Intel operations set [21]. Afterward, Intel introduced AVX-2, which has the same register size of 256 bits but additional

operations, which allows us to not only work on 256 bits register but also run more sophisticated operations [17]. Lastly, AVX-512 extended the register size again to 512 bits, extended the number of registers from 16 to 32 and added additional instruction sets for more complex operations [11].

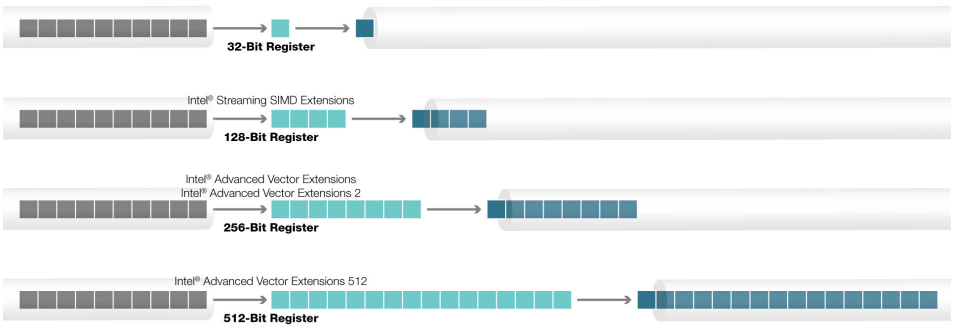

Figure 2.10: Comparison of different version of AVX and traditional register size, Figure reprinted from [36].

In order to see the differences in scale, we can look at Figure 2.10 which illustrates the differences in register size between the different versions of AVX and the traditional register size.

As mentioned above, in each version of AVX, Intel introduces more powerful instruction sets. When discussing the hashes in Section 2.2, we have seen that we need some logic and arithmetical operations to hash values. When comparing AVX2 and AVX-512, we can see that new operations, such as the rotate intrinsic, can help us speed up the performance of hashing. Also, we can use the ternary logic operation to boost the performance of logic operations on three values. More about those benefits will be discussed in Section 6.2 where we explain the motivation for the results we got.

Another benefit of AVX-512 comparing to previous versions is the increase in the number of registers. While in the previous version, AVX2, there were only 16 registers in AVX-512, there are 32 registers, allowing us to keep more data in memory. Later in Section 6.2 we will see how this helps us to improve the performance of our scheme.

To conclude this section, it can be seen that the new AVX-512 has some advantages over its predecessors. Those advantages can help boost the speed of a program, which will allow us to perform more tasks in a given unit of time.

Chapter 3

Related Work

In this paper, we propose a new implementation of SPHINCS+ using AVX-512. By using AVX-512, we hope to optimize the performance in terms of speed for the SPHINCS+. In order to better understand our implementation and the possible optimization, we compare it to other implementations and analyze the results.

The first comparison that comes to mind is to compare it to the REF implementation proposed in [4]. However, we need to consider that this version is not optimized in terms of efficiency and is instead used as proof of concept for the SPHINCS+ scheme.

Another implementation of SPHINCS+ to which we can compare our results is the AVX2 implementation [39], which is also part of the code submitted to the third round of the NIST post-quantum competition [5]. By comparing those three implementations, we will better understand the different operations of the scheme and the benefit of using AVX-512. In addition, we will be able to grasp and understand which parts we are getting a meaningful improvement. For instance, if we expect AVX-512 to be twice as fast as AVX2 and observe that we only get an improvement of 1.5× for some parts, we might think the implementation is not optimal. However, suppose we also see that we get an improvement of 1.6× for the same parts comparing to the REF and AVX2, it will become clear that it is not about the implementation not being optimal enough or due to overhead, but rather due to parts that cannot benefit from the use of this type of parallelism.

As we mentioned above, SPHINCS+ is a candidate in the NIST post-quantum competition, [25]. It is one of the alternative schemes of the third round for the Digital Signature Algorithms competition. Therefore, we are going to end the last section with a comparison to the other five NIST

candidates. It is important to mention that, unlike our implementation, the other implementations are not optimized for AVX-512. However, as they are all part of the NIST post-quantum competition, it is still essential to see the different performances for each candidate and compare them to our implementation. For this comparison, we are going to use SUPERCOP [3] which has the implementation of the other candidates. SUPERCOP is toolkit used to measure the performance of cryptographic software. Also, we are going to integrate our implementation into SUPERCOP, in order to make the comparison as fair as possible. Next, we are going to introduce each one of the candidates.

There are two lattice-based post-quantum signature schemes, CRYSTALS-DILITHIUM [22] and FALCON [13], two multivariate based post-quantum signature schemes, Rainbow [19] and GeMSS (A Great Multivariate Short Signature) [10] and one other scheme, Picnic [41]. Unlike the other schemes, Picnic is not based on number-theoretic or structured hardness assumptions, but rather on zero-knowledge proofs, where Alice convinces Bob that she knows a secret without revealing any information about the secret. It uses this idea together with symmetric cryptography, hash functions, and block ciphers.

Apart from those schemes, which we will compare to our implementation, there is another implementation we should acknowledge. As mentioned above, SPHINCS+ uses SHAKE256 as one of its hash functions. For that purpose, we used the eXtended Keccak Code Package for AVX-512, which was written by the Keccak team [40] and further discussed in [7]. It is an optimized version of Keccak for AVX-512, and we used its implementation of the 24 rounds (the f function) on the state array in our proposed implementation.

Chapter 4

Optimizing SPHINCS+

As we have seen in Section 2.1, where we explain how the SPHINCS+ scheme works, the main operations we are performing are creating the keys, signing a message, and verifying a signature. It should not be a surprise that the main operation in a hash-based scheme is hashing. Therefore, if we want to optimize the speed of the scheme, we will have to focus on improving the performance of hashing, and later on, combine this new version of hashing in the calls of the scheme. That is why it is essential to understand how the hashes work, as we explained in Section 2.2. As we have seen in Section 2.1, we have $a \cdot 2^{h'}$ leaves in our tree, and each one of them is using hashes that are independent of each other, meaning that the hash value of one of the leaves is not dependent on the hash value of another leaf. Therefore, we can parallelize them by hashing different leaves simultaneously, which would speed up the performance. It is important to mention that this approach is already present in another implementation of the SPHINCS+ code [39]. However, in that case, they used an older version of the AVX family, AVX2, while in this thesis, we are using the newer version AVX-512. This proposed implementation works the same way as the original scheme, which implies it does not affect its security, but only improves upon its speed by performing a similar operation on the hash values, using the AVX-512 as we saw in Section 2.3.1, when we talked about SIMD vectorization. Below, we will explain this idea further and see how each hash is implemented, using AVX-512 in our implementation of SPHINCS+. As we have seen in Section 2.1.5, SPHINCS+ has two types of implementation: the simple version, which focuses on speed, and the robust version, which provides a more secure version. However, for simplicity, in this section, we do not go into details about either of them, but rather explain how we improved their main operations. For more details, see the code [1].

4.1 SHA256 Implementation

Let us start by explaining how we implemented SHA2 (SHA256) using AVX-512. Similar to what we saw in Section 2.2.1 we will follow those steps, but instead of hashing one value at the time, we are going to hash 16 values at the time. Since we are working with 32-bit values for SHA256, and as we know that AVX-512 has a register size of 512 bits, we can work on $512/32 = 16$ values at once. Therefore, in the code, [1] we can see the *x16* added to some of the function and file names to emphasize that their idea is the same, but they perform 16 hash values at one call. Now, we are working with 32-bit values and as we have seen in Section 2.2.1 we have eight values we are working with $(a, b, \ldots, h)$. This means that we have $32 \cdot 8 = 256$-bit values which we work with for one hash value. In order to work with the 512-bit registers, two of those 256-bit values are placed within one register. This is done as follows: hash values 0 through 7 are placed in the first part of the register and hash values 8 through 15 are placed in the second part of the register, as shown in Figure 4.1. Next, we are going to explain how we perform the hashing on 16 values at once.

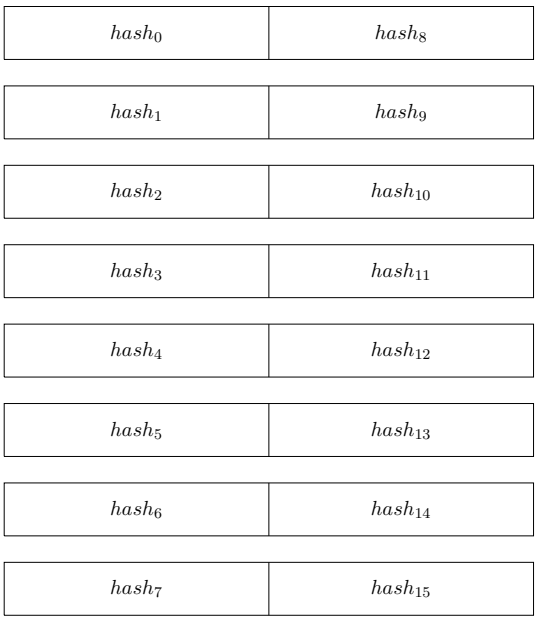

Figure 4.1: Illustration of the ordering of the 16 hash values in the registers.

As explained above, we are going to store the 16 values in 8 registers, for convenience, an array, called s, of eight entries each of 512 bits is used. As expected, the first step in hashing is initializing the array with the SHA256 constants. Since there are 16 values hashed at once, there are sixteen values of each constant. As each constant is 32 bits and we have a 512-bit

register, this fits perfectly. For an illustration of the code, the initialization part is given in Figure 4.2, where *_m512_set1_epi32* is an Intel intrinsic for broadcasting the value given to the register as a 32-bit value.

```
void sha256_init16x(sha256ctx *ctx) {
    ctx->s[0] = _mm512_set1_epi32(0x6a09e667);
    ctx->s[1] = _mm512_set1_epi32(0xbb67ae85);
    ctx->s[2] = _mm512_set1_epi32(0x3c6ef372);
    ctx->s[3] = _mm512_set1_epi32(0xa54ff53a);
    ctx->s[4] = _mm512_set1_epi32(0x510e527f);
    ctx->s[5] = _mm512_set1_epi32(0x9b05688c);
    ctx->s[6] = _mm512_set1_epi32(0x1f83d9ab);
    ctx->s[7] = _mm512_set1_epi32(0x5be0cd19);

    ctx->datalen = 0;
    ctx->msglen = 0;
}
```

Figure 4.2: The initialization of array s.

For a better illustration of how the *_m512_set1_epi32* intrinsic works in Figure 4.2 we are also given an example of how the registers would look like in Figure 4.3 where cs_i refers to the constants such that $cs_0 = 0x6a09e667$, $cs_1 = 0xbb67ae85$, ... , $cs_7 = 0x5be0cd19$.

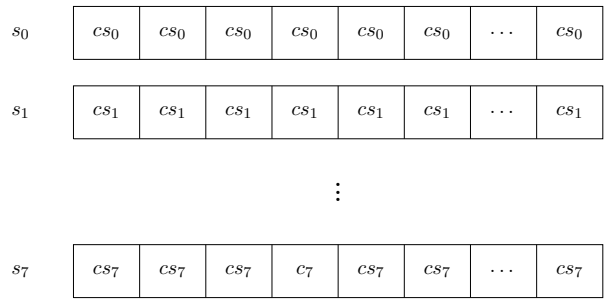

Figure 4.3: Illustration of the ordering of the constants in s, where we have 16 values within one entry.

The next step is to upload the values we want to hash into an array of 512-bit values, called W, and transpose the data to correspond with the constants we filled in our s array. Then, we create functions that will perform the operations used in SHA256. We start with computing the values for W using the sigmas, as we saw in 2.2.1. In the code, these sigmas are called *WSIGMA1_AVX* and *WSIGMA0_AVX*. Below, in Figure 4.4 the

most significant functions used in this part are given.

```
#define SHIFTR32(x, y) _mm512_srli_epi32(x, y)
#define SHIFTL32(x, y) _mm512_slli_epi32(x, y)
#define ROTR32(x, y) _mm512_ror_epi32(x,y)
#define ROTL32(x, y) _mm512_rol_epi32(x,y)

#define WSIGMA1_AVX(x) XOR3(ROTR32(x, 17), ROTR32(x, 19),
↪  SHIFTR32(x, 10))
#define WSIGMA0_AVX(x) XOR3(ROTR32(x, 7), ROTR32(x, 18),
↪  SHIFTR32(x, 3))
```

Figure 4.4: Definition of s's functions for SHA256.

Then we have another function called *SHA256ROUND_AVX*, where we perform the S's , ch, maj, $temp_1$, $temp_2$ and the $a - h$ updates, which can be seen in Figure 4.5. Again, for readability, we only show the most relevant parts of the code.

```
#define XOR3(a,b,c) _mm512_ternarylogic_epi32(a,b,c,0x96)

#define MAJ_AVX(a, b, c)
↪  _mm512_ternarylogic_epi32(a,b,c,0xE8)
#define CH_AVX(a, b, c)
↪  _mm512_ternarylogic_epi32(a,b,c,0xCA)

#define SIGMA1_AVX(x) XOR3(ROTR32(x, 6), ROTR32(x, 11),
↪  ROTR32(x, 25))
#define SIGMA0_AVX(x) XOR3(ROTR32(x, 2), ROTR32(x, 13),
↪  ROTR32(x, 22))

#define SHA256ROUND_AVX(a, b, c, d, e, f, g, h, rc, w) \
T0 = ADD5_32(h, SIGMA1_AVX(e), CH_AVX(e, f, g),
↪  _mm512_set1_epi32(RC[rc]), w); \
d = ADD32(d, T0); \
T1 = ADD32(SIGMA0_AVX(a), MAJ_AVX(a, b, c)); \
h = ADD32(T0, T1);
```

Figure 4.5: Definition of SHA256ROUND_AVX functions for SHA256.

For the *MAJ* and *CH* functions, we use a new intrinsic that allows us to perform fast logical operations on three values at once. To achieve that, we pass a constant as the last element, which specifies how to interpret the

logical operation. For instance, for the *XOR3*, we pass the hexadecimal value $0x96$, which in binary is 10010110, this value being the result of the truth table of the expression $a \oplus b \oplus c$. The same values were calculated to give the equivalent results for the *MAJ* and *CH*.

After executing those functions, we add the constants one last time, and we get the hashed values in s. However, before extracting the values, we transpose them again in order to get them in the order needed for extraction. Then we simply extract them.

4.2 SHAKE256 Implementation

Moving on, we will explain how SHA3, SHAKE256, is implemented using AVX-512. Similar to the case of SHA256, we follow the steps from Section 2.2.2, but instead of performing hashing on one value at the time, we are going to hash eight values at once. The idea here is that we are working on 1600 bits state that is split into 25 lanes, so our w is 64, which means that we have 25 lanes of size 64 bits. So, using the AVX-512, we can perform $512/64 = 8$ hashes simultaneously, hence the eight values at once.

For this part, we used an existing Keccak implementation for AVX-512, [40] which was optimized by the Keccak team. This idea is similar to the SHA256 version above: we pass eight values and we run the functions on them simultaneously. As expected, we start with a zero state, then we initialize the state array with the values such that we place eight values of 64 bits in a 512 bits entry of the state. So, the state will look like Figure 4.6.

h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7

Figure 4.6: Illustration of the ordering of the initial values in one register of 512 bits where h represents the initial 64-bit values.

We have the state, which is represented as an array of 25 entries of 512-bit values called state, and in the initialization, we upload the values in the same way as shown in Figure 4.6. The code for this part can be seen in Figure 4.7.

```
for (unsigned int i = 0; i < (SPX_N/8) * inblocks; i++) {
        state[SPX_N/8+4+i] = _mm512_set_epi64(
                    ((int64_t*)in7)[i],
                    ((int64_t*)in6)[i],
                    ((int64_t*)in5)[i],
                    ((int64_t*)in4)[i],
                    ((int64_t*)in3)[i],
                    ((int64_t*)in2)[i],
                    ((int64_t*)in1)[i],
                    ((int64_t*)in0)[i]
                );
    }
```

Figure 4.7: Initialization of the state of SHAKE256.

After we initialize the state with our input and fill the state array with the corresponding values, we can run the f functions. This means that we are going to run the 24 rounds on the state array. For that, we are using the function *KeccakP1600times8_PermuteAll_24rounds*, which is taken from the Keccak implementation for AVX-512 [40]. In that function, we perform all the smaller functions we discussed in Section 2.2.2. Depending on the scenario, we either run this function once or until we are done with the absorption and squeezing phases. After the hashing, we extract the first 256 bits of the values, as shown in Figure 4.8.

```
__m256i_u take1, take2;
for (int i = 0; i < 8; i++) {
    (take1) = _mm512_extracti64x4_epi64(state[i],0);
    (take2) = _mm512_extracti64x4_epi64(state[i],1);

    ((int64_t*)out0)[i] = _mm256_extract_epi64(take1, 0);
    ((int64_t*)out1)[i] = _mm256_extract_epi64(take1, 1);
    ((int64_t*)out2)[i] = _mm256_extract_epi64(take1, 2);
    ((int64_t*)out3)[i] = _mm256_extract_epi64(take1, 3);
    ((int64_t*)out4)[i] = _mm256_extract_epi64(take2, 0);
    ((int64_t*)out5)[i] = _mm256_extract_epi64(take2, 1);
    ((int64_t*)out6)[i] = _mm256_extract_epi64(take2, 2);
    ((int64_t*)out7)[i] = _mm256_extract_epi64(take2, 3);
}
```

Figure 4.8: Extracting the hashed values from the state array.

Chapter 5

A Lower Bound on the Amount of Cycles per Hash

As mentioned above, we are going to use AVX-512 in order to optimize the implementation of SPHINCS+. The idea will be to measure the execution time using clock cycles and compare the implementation proposed in this thesis to other implementations. However, before talking about optimization and asking to what extent we can improve the speed of the scheme, we will first look at the theoretical improvement that can be gained from using the AVX-512. In order to do so, we are going to derive a lower bound on the amount of cycles per one hash. As SPHINCS+ is a hash-based scheme, most of the execution time is spent on hashing. Therefore, it is important to see how many cycles it takes to perform one hash. By doing so, we will better understand the improvement and see the limitations of using the AVX-512. It is also important to distinguish between the theoretically expected speed up and the practical one. In theory, we should be able to derive the amount of cycle it takes to perform a hash. However, this is not always the case in practice, as running the code will give us a different amount of cycles compared to the theoretical one. This is due to the differences in communication and execution of the CPU, which is not taken into consideration when calculating the theoretical amount of cycles and due to different optimizations done by the CPU. Therefore, we are going to name the derived amount of cycles the lower bound on the amount of cycles rather than the expected amount of cycles. Notice that we will only calculate the amount of cycles that it takes to compute the hash of a value and not the amount of cycles of other operations, such as loading the values to the registers, copying them to variables, etc.

After we have discussed the meaning and motivation behind deriving the lower amount of cycles, we are going to calculate it. To do so, we will

use our implementation together with the fourth Software Optimization Resources manual [12] and Intel intrinsics guide [18], which will provide us with the throughput of the operations we perform in order to hash. We are using the throughput and ports given for the Skylake microarchitecture.

In order to compute the cost of a hash, we need to consider the smaller operations we perform while hashing. One might think that we can simply calculate the number of operations we perform and then multiply it by the execution time of each operation, add them all together, and get the lower bound. However, this approach is too pessimistic, as the CPU can execute some of the operations simultaneously. There are a few execution units for each execution core of a microprocessor, on which different categories of operations can be performed. Those execution units are clustered around one or more execution ports. The idea is that each operation passes through an execution port to get to the execution unit in which it will be performed. This might result in a bottleneck if multiple operations request the same port, as only one operation can be performed at a time. However, this also means that different operations which are performed on different ports can be done simultaneously as long as they are independent of each other's output and there are enough registers. Thus, it is not enough to count the number of operations and multiply them by the operation's execution time, but we also need to think about how the CPU schedules them on different ports. Hence, before we can compute the cost of those functions, we give the throughput of AVX-512 intrinsics used and the port in which the instruction is executed. Next, we are going to derive the lower bound for SHA256 and SHAKE256.

5.1 SHA256

Let us start with driving a lower bound for SHA256. The main operations we have in SHA256 are the round, consisting of $S_0, S_1, ch, maj, temp_1$ and $temp_2$, the sigmas for w and the eight adds at the end, as we saw in Figure 4.4 and Figure 4.5 in section 4.1. The first step we need to perform in order to compute the cost is to specify the cost of each intrinsic and the port it is executed on. For the logical operations *AND* and *XOR*, we have the cost of 0.5 cycles on port 5. For the ternary logical operations *MAJ*, *CH* and *XOR three times*, we have the cost of 0.5 cycles on port 5 again. For *shift right* and *rotate right* we have the cost of 1 cycle on port 0 and for broadcast 32-bit value to 512-bit value we have the same cost, but instead of port 0 we have port 5. Finally, for a 512-bit value added as 32-bit values, we have the cost of 0.5 cycles on port 5.

Now, let us start with computing the cost of a round. For that, we have

seven *32 bits add*, six *rotate right*, two *ternary XORs*, one *CH*, one *MAJ* and one *set*. Some of the operations are dependent on each other's output, so they cannot all be run simultaneously. For this reason, the expected order of execution is given in Figure 5.1, where each line represents 0.5 cycles. So, if we want to execute an operation of 1 cycle, we will have to leave the next row empty as this port is still busy.

port 0	port 5
rotate	CH
	add
rotate	set
rotate	add
	add
rotate	XOR3
	add
rotate	add
	MAJ
rotate	
	XOR3
	add
	add

Figure 5.1: Order of execution of the operations in one round of SHA256 where each row represents 0.5 cycles.

As we can see, we have 15 rows, which means that it takes us 7.5 cycles to compute one round. We perform the round while $i \in \{0, 1, ..., 63\}$ thus, we perform it sixty four times which gives us $7.5 \cdot 64 = 480$ cycles. Next, we will compute the amount of cycles for the sigmas. For the sigmas, we have three *32 bits add*, four *rotate to the right*, two *shift to the right* and two *ternary XORs*. Similarly to above, the order of execution is given in Figure 5.2 where each row represents 0.5 cycles.

port 0	port 5
rotate	add
rotate	
shift	
rotate	XOR
	add
rotate	
shift	
	XOR
	add

Figure 5.2: Order of execution of the operations performed in the sigmas of SHA256 where each row represents 0.5 cycles.

As we can see, we have 14 rows which means that it takes us 7 cycles to compute one round. We perform the sigmas while $i \in \{16, 17, ..., 63\}$ which means we perform it forty eight times, which gives us $7 \cdot 48 = 336$ cycles. Finally, in the end, we add the 8 constants, which takes 4 cycles. So, in total we get $480 + 336 + 4 = 820$ cycles for one hash.

5.2 SHAKE256

Now, let us take a look at SHAKE256 and derive a lower bound for it as well. In this case, the most important part is the 24 rounds function. As in the other parts, we simply initialize and extract, while in the rounds, we perform operations on the state. Again, before we move to the computation part, we provide a list of throughputs for the AVX-512 intrinsics used and the ports on which they are executed. For the logical operation *XOR* and *OR*, we have 0.5 cycles on port 5. For *rotate left*, we have 1 cycle on port 0. Finally, for the ternary functions *XOR3* and *CHI*, we have 0.5 cycles on port 5. Now, let us start with computing the throughput of one run of the 24 rounds function. The implementation of the 24 rounds is split into 6 blocks of 4 rounds and in every block, we perform the function we discussed in section 2.2.1 four times. In each round of the f function, we perform the following operations *KeccakP_ThetaRhoPiChiIota0*, *KeccakP_ThetaRhoPiChi1*,

KeccakP_ThetaRhoPiChi2, *KeccakP_ThetaRhoPiChi3* and *KeccakP_ThetaRhoPiChi4*. However, as we can see in the last four functions, we perform the same operations (call to *KeccakP_ThetaRhoPiChi*) but with different values. So, we only need to calculate the cost for the first one and for *KeccakP_ThetaRhoPiChi*.

For *KeccakP_ThetaRhoPiChiIota0* we have 5 *XOR5s*, 9 *rotates to the left*, 5 *CHIs* and 11 *XORs*. For *XOR5*, we can simply perform *XOR3* two times, which will take us 1 cycle. Next, we are going to provide the order in which the operations are executed and based on that, we will compute the throughput for this part. Again each line represents a 0.5 cycle.

port 0	port 5
	XOR5
	XOR5
rotate	XOR5
rotate	XOR5
rotate	XOR5
rotate	XOR
	XOR
rotate	XOR
	XOR
	XOR
	XOR
rotate	XOR
	XOR
rotate	XOR
	XOR
rotate	
rotate	
rotate	
	Chi
	Chi
	Chi
	Chi
	Chi
	XOR

Figure 5.3: Order of execution of the operations in *KeccakP_ThetaRhoPiChiIota0* where every row takes 0.5 cycles.

As we can see, we have 32 rows which means that it takes us 16 cycles to compute *KeccakP_ThetaRhoPiChiIota0*. Next, we will perform the same process to compute the amount of cycles it takes to execute *KeccakP_ThetaRhoPiChi*.

For *KeccakP_ThetaRhoPiChi* we have 5 *XORs*, 5 *rotates to the left* and 5 *CHI*. The order of execution of that can be seen in Figure 5.4.

port 0	port 5
	XOR
	XOR
rotate	XOR
	XOR
rotate	XOR
rotate	
rotate	
	Chi
	Chi
	Chi
	Chi
	Chi

Figure 5.4: Order of execution of the operations in *KeccakP_ThetaRhoPiChi* where every row takes 0.5 cycles.

As we can see, we have 15 rows which means that it takes us 7.5 cycles to compute *KeccakP_ThetaRhoPiChi*. As explained above, we perform this 4 times together with *KeccakP_ThetaRhoPiChiIota0* and get one round. So in total, we know that one round takes $16 + 7.5 \cdot 4 = 46$ and as we have 24 rounds, we know that the total throughput of performing all the rounds takes $46 \cdot 24 = 1104$ cycles.

Chapter 6

Results and Comparison

In this section, we are going to take a look at the performance of the proposed implementation. First, we will compare the results we got for our implemented version of the hashes to the lower bound we computed in Chapter 5. Then, we move on to comparing the run time of hashing a single value between our version and the AVX2 implementation. Afterwards, we compare our implementation to the REF [4] and AVX2 [39] implementations of SPHINCS+. Finally, we will compare it to other third-round NIST post-quantum competition schemes. Before we start with the results, we should mention that all the executions of the different implementations are done in a benchmark environment, where performance boosts are turned off. The processor used for those measures was Cascade Lake processor, Xeon Silver 4215R, which has eight cores per socket and 16 CPUs. Although this is a slightly different microarchitecture than the one used in Section 5 it is the closest documented microarchitecture to Cascade.

6.1 Comparison of Lower Bound to Run Time

As discussed in Chapter 5 we derived a lower bound for the amount of cycles expected from the execution of the code. The next step will be to measure the execution of our implementation for the hashes and compare it to the lower bound we computed. By doing so, we will see how close our implementation is to the theoretical execution time. This comparison was carried out several times during the implementation of the code until we got a close enough result to the lower bound. However, here we only include the final results.

As we have discussed in section 5.1 for SHA256, we got a lower bound of 820 cycles for the computation of one hash. We got an execution time of 880 cycles in our implementation, which is relatively close to our expectations.

For SHAKE256, as mentioned in Section 5.2, we got a lower bound of 1104 cycles, while in our implementation, we got a run time of 1245 cycles. As we can see, both implementations are within 12% from our theoretical lower bound. It was according to our expectations. Therefore, we can move on to the next step, linking them with the code of SPHINCS+ and later comparing the execution time to the other implementation of SPHINCS+ and the other NIST PQC candidates. However, before moving to the result of the comparison of the scheme as a whole, we will look at the improvement of the hash functions in our AVX-512 implementation comparing to the AVX2 implementation and try to understand the motivation behind it.

6.2 Comparison of Single Hash in AVX2 and AVX512

This section will compare the execution of hashing a single time using AVX2 and AVX-512 implementations. Same as for the measures for the lower bound, we only look at the operations of the hashing and not other side operations, such as initialization and extraction. This means that for SHA256, we are only looking at the round function and the computation of w and for SHAKE, we are only looking at the 24 rounds function.

In the case of SHA256, we got the following results: for the AVX2 implementation, it takes 1,252 cycles to run one hash value, while for the AVX-512 implementation, it only takes 880 cycles to perform one hash. This means that we get a $1,42\times$ speed-up improvement in our implementation using AVX-512. The main reason for this is that we can run faster different operations using the new Intel intrinsics. The main improvements achieved from using faster intrinsics are:

- In order to perform a rotate operation, we needed to shift twice and then XOR in AVX2. In AVX-512, we can simply use a rotation instruction that takes one cycle.
- Intel introduced a new ternary logic function that allows us to perform any logic operation on three registers at the cost of one logic operation. This was used in simple functions such as *XOR three times*, which reduces the cost of XORing, but also in more complicated functions such as *MAJ* and *CH*. In AVX2, in order to perform the *MAJ* or *CH* function, we needed to perform 3 ANDs and 2 XORs or 2 XORs, 2 ANDs and 1 set, respectively. Now we can simply call the ternary function, which has the same throughput as any of those logic operations.
- We broadcast the same 32-bit value to 512 bits instead of internalizing it 32 bits by 32 bits while initializing the constants.

In the case of SHAKE256, we got the following results for the AVX2 implementation: hashing one time takes 1790 cycles, while for the AVX-512, performing one hash takes 1252 cycles. This means that we get a $1,43\times$ speed-up improvement in the implementation of SHAKE256 with AVX-512. It is important to mention that this implementation was written by the Keccak team and further discussed in [7]. In order to understand the motivation behind these speed-ups, we need to consider the benefit of the AVX-512 and consider how SHAKE256 works. As we have discussed in section 2.3, one of the improvements comparing to AVX2 is the addition of more registers. While in AVX2, there were only 16 registers, in AVX-512, there are 32 registers. Now, we know that for the SHAKE256, there is a state of 1600 bits split into 25 lanes. In the case of AVX2, since we only have 16 registers, we will have to swap lanes between the memory and the registers, which is a time-consuming operation. However, in AVX-512, since we have 32 registers, we can simply keep all lanes in the register and thus perform the operations faster. Also, we can benefit from the ternary logic operation and rotate operation mentioned above. All of those improvements result in the speed up that we have seen.

6.3 Comparision of different implementations of SPHINCS+

In the following sections, we will compare the proposed implementation of SPHINCS+ to the two other versions: the AVX2 implementation and the REF implementation. We will give the number of cycles for the most common operations of the scheme, as well as for the two smaller schemes used in SPHINCS+, which were discussed in Section 2.1. Those operations generate the public and private keys for the whole scheme, WOTS+ public key, signing a message (using SPHINCS+), signing a message using FORS, and verifying a signature.

6.3.1 SHA256

Let us start by analyzing the results of SHA256. Before we present the results, it is important to mention that we expect two improvement values, depending on the option used for the SPHINCS+, as there are two versions in that implementation. In order to understand the motivation for that, we need to consider the parameters of the different versions and the idea of the improvement. We are basing our improvement of SHA256 on two benefits, the bigger size of the registers and the new faster AVX-512 intrinsics. Now, for the advantage of the register, as explained in Section 4.1 we are performing 16 hashes at once, meaning we work on 16 nodes at once. However, in the case of SPHINCS+ using the f option with sizes 128 and

192, the hypertree has a height of 66 and the number of subtree layers is 22. This means that the height of a tree will be $\frac{66}{22} = 3$, so we will have $2^3 = 8$ leaves in one tree. Since the code is implemented in such a way that we work on a single tree at a time and compute its root, we face the problem that we only have 8 leaves in a tree, which will not allow us to perform 16 operations at once simply because we do not have 16 leaves to work on. Therefore, we cannot benefit from the improvement of performing twice fewer operations. For that reason, these options have a slightly different implementation, while still using the new AVX-512 intrinsics, we only perform 8 hash at once, instead of 16. This means that the improvement we expect will be smaller for those options. This can be seen in Table 6.1 where we give the results of different operations for the three implementations of the SPHINCS+ using SHA256, our proposed implementation AVX-512, the AVX2 implementation, and the REF implementation.

	Keypair Generation (cycles)	WOTS pk generation (cycles)	Signing (cycles)	FORS signing (cycles)	Verifying (cycles)
128f robust AVX-512	1329235	1307109	31477151	2260854	3161335
128f robust AVX2	1962009	1936334	46478171	3275073	4236203
128f robust REF	10772523	1344012	249889292	12870059	15652196
128f simple AVX-512	732076	724326	17240880	1170396	1622396
128f simple AVX2	1057992	1067508	24903369	1653298	2114719
128f simple REF	5432123	678936	127283704	7644523	7619587
192f robust AVX-512	2049619	2011244	54712829	9509437	5271305
192f robust AVX2	2980925	2956470	79648436	13926095	6770038
192f robust REF	16080880	2003874	424844865	71212878	24408338
192f simple AVX-512	1108198	1097462	29712256	5355827	2588770
192f simple AVX2	1594595	1582247	42957518	7859900	3226355
192f simple REF	8017572	996445	215379398	39256192	11522757
256f robust AVX-512	8058598	8020842	162544195	25427339	9570043
256f robust AVX2	16781379	8385915	336779564	51380624	13882777
256f robust REF	60672290	3784410	1240958967	209446921	35849994
256f simple AVX-512	2439356	2403821	50828565	9349677	2599902
256f simple AVX2	4199701	2092771	87999113	16413113	3371802
256f simple REF	20921322	1304590	438061836	82966534	11930640
128s robust AVX-512	65157755	2027415	497899597	41773867	1233761
128s robust AVX2	124449739	1933960	948725240	77513432	1663939
128s robust REF	688486498	1342298	5171599328	350399739	5279514
128s simple AVX-512	37368240	1185413	284785923	23215968	671904
128s simple AVX2	67259588	1049217	511255919	40423932	789884
128s simple REF	345259975	672793	2624259822	206965533	2557810
192s robust AVX-512	102369620	3181245	953234172	236646319	2312743
192s robust AVX2	191639633	2958018	1782224754	440805869	2862057
192s robust REF	1025097181	1997362	9520139802	2344050584	8822422
192s simple AVX-512	57191635	1780133	535278766	134943588	1097686
192s simple AVX2	101020935	1574326	956109801	248924122	1365384
192s simple REF	514163879	1005737	4892530688	1294632267	4284338
256s robust AVX-512	131420674	8142236	1542643320	491153345	5477365
256s robust AVX2	268349824	8366201	3163901068	1017087661	7492251
256s robust REF	972569496	3791256	11994415382	4205064722	18029912
256s simple AVX-512	38580865	2426475	487479308	178897003	1552991
256s simple AVX2	67312150	2098441	864543353	326158773	1960065
256s simple REF	334958126	1304665	4340941609	1660977800	5896725

Table 6.1: Comparison of the different versions of SPHINCS+ using SHA256 in terms of amount of cycles taken to perform hash operations for AVX-512, AVX2 and REF implementation.

For a better comparison, we normalized the values of the experiment by dividing the AVX2 measured values by the AVX-512 measured values, the REF measured values by the AVX2 measured values and the REF measured values by the AVX-512 measured values and took the average of all the results. We put the results in two figures based on whether we hash 8 times at once or 16 times at once. The results are in the following figures Table 6.2 and Table 6.3.

	Keypair Generation	WOTS pk genration	Signing	FORS signing	Verifying
AVX2/AVX-512	1.454	1.467	1.456	1.448	1.294
REF/AVX2	5.262	0.659	5.201	4.665	3.619
REF/AVX-512	7.651	0.967	7.584	6.761	4.682

Table 6.2: Average of the improvement factors of the results in the case of performing 8 hashes at once from Table 6.1.

	Keypair Generation	WOTS pk genration	Signing	FORS signing	Verifying
AVX2/AVX-512	1.868	0.933	1.873	1.872	1.298
REF/AVX2	4.788	0.600	4.815	4.815	3.020
REF/AVX-512	8.875	0.555	8.958	8.970	3.903

Table 6.3: Average of the improvement factors of the results in the case of performing 16 hashes at once from Table 6.1.

As we expected, we get different results for the two versions we have. For the 8 times with only the new intrinsics, we get a speed-up of around 1.45× for key pair generation, WOTS+ public key generation signing, and FORS signing and 1.29× improvements for verification of a signature. For the 16 times optimization, we get around 1.87× improvements for key pair generation, signing, and FORS signing, around 1.3× for verification of a signature, however, we get a slower implementation for the WOTS+ public key generation by 0.93×. In order to understand the reason behind these results, we need to consider the benefit of the AVX-512 and consider the implementation of SHA256.

Let us start by analyzing the results of the 8 times version. In that case, we simply use the new faster intrinsics, which improve the implementation of the hash. As we can see, the improvement we get for the key generation, WOTS+ public key generation, signing and FORS signing is almost the same as the improvement we get for the hash values, 1.45× compared to 1.42×. The idea here is that we are mainly hashing in those operations, so it is expected that if we improve the hash, we will get the same improvement for the whole operation. In the case of verifying, we have the same limitations, as we need to wait for some nodes in order to compute the root using the authentication path. Therefore the time improvement is slightly slower.

Now, let us look at the 16-time version. In this case, we see that we get a slower implementation for the WOTS+ public key generation, the reason for that being due to the overheads we have as we are working on two values within a single register. For instance, in the initialization function, we need to make sure we correctly place every two hashes in one register and the same should be done for input hash values we receive when placing

them in w. Later on, in the transpose function, we again need to make sure we transposed correctly and this is done slightly differently in AVX-512 comparing to AVX2. Due to the new shuffling intrinsic in AVX-512, we need to pass a 512-bit integer value instead of an int in AVX2, which is slower. Also, we again need to make sure we correctly take the hashes from each register in the extract. While simply hashing is done faster, those operations create some overhead, which results in a slightly slower implementation for the WOTS+ public key generation. Nevertheless, we need to remember that we will perform half the amount of hashing in the signature operations by performing those operations. So, we will still get some improvement while performing those operations. In addition, as we can see, we also get a slow down for WOTS+ public key generation when comparing AVX2 and REF. This is again due to the overhead in those functions.

Next, for key pair generation, signing and FORS signing, we get an improvement of around 1.87×. The reason for the improvement is that as we perform 16 values at once instead of 8, we can perform half the amount of operations. In those operations, we mainly perform WOTS+ public key generation and hashing and, as we saw, we get a 0.933× slow down moving from AVX2 to AVX-512, but we perform twice less. So, we should expect an improvement of around two times this value, which gives $2 \cdot 0.933 = 1.866$. This is similar to the improvement we get for those operations.

Finally, in the last operation, verification, we can see that the improvement is lower. Similarly, it is also the case when comparing the REF implementation to the AVX2. We see a 3.02 improvement for the verification, comparing to around 4.8 in the other operations. The motivation behind it is that while verifying, we need to wait for values to hash and recall that we are given a leaf and a path. So, unlike in the generation, where we can just run each leaf and path separately, here we need to wait until we computed the lower path, in order to continue with the higher path. For instance, let us take an elementary example, where we have eight leaves: we use leaf 1 as a node, in the authentication path, we will get leaf 0, a hash of a combination of leaves 2 and 3, and a hash of a combination of leaves 4, 5, 6 and 7. Now, to perform the operations on the hash of leaves 4, 5, 6 and 7 we need the hash of leaves 0, 1, 2 and 3. For that, we need the hash of leaves 0 and 1, and the hash of leaves 2 and 3, and so on. So, in this case, we have some dependencies in the tree, which cause us to perform a little slower, as we cannot simply run everything twice faster.

6.3.2 SHAKE256

Moving on to SHAKE256, in Table 6.4 we give the results of different operations for the three implementations of the SPHINCS+, using SHAKE256, our proposed implementation AVX-512, the AVX2 implementation, and the REF implementation.

	Keypair Generation (cycles)	WOTS pk generation (cycles)	Signing (cycles)	FORS signing (cycles)	Verifying (cycles)
128f robust AVX-512	1508149	1496761	35217789	2054807	3638377
128f robust AVX2	4292217	2138524	99846834	5367302	7536495
128f robust REF	18160305	2264230	421328445	21973966	25570140
128f simple AVX-512	792018	785041	18663577	1259390	1872484
128f simple AVX2	2234613	1115702	52842798	3255549	3803755
128f simple REF	9343874	1165200	218744119	13143815	12835936
192f robust AVX-512	2220205	2208578	56329421	7504515	5148846
192f robust AVX2	6190683	3086233	157065137	20884821	10737256
192f robust REF	26559796	3313270	676114412	88391932	37777982
192f simple AVX-512	1169080	1161554	30344265	4638726	2634774
192f simple AVX2	3252835	1610788	83985470	12574724	5420898
192f simple REF	13691035	1706685	353631291	52736331	19238198
256f robust AVX-512	5894607	2940407	115938201	15777837	5332546
256f robust AVX2	16047940	4006880	316833904	43770581	10629669
256f robust REF	70695400	4414168	1390519744	189055237	38598677
256f simple AVX-512	3119372	1569917	62793283	9736976	2787534
256f simple AVX2	8477199	2118907	170791517	26497230	5493487
256f simple REF	36070434	2250485	724238430	111610011	19331306
128s robust AVX-512	95970218	1496075	720732817	48980504	1677370
128s robust AVX2	274304451	2138256	2060509426	140487765	2923322
128s robust REF	1165534526	2264543	8731958180	597812954	8948336
128s simple AVX-512	50666487	785086	384737484	30075364	833056
128s simple AVX2	143087646	1115186	1086418236	85222909	1471744
128s simple REF	598867558	1164928	4549473219	356213066	4589397
192s robust AVX-512	141505690	2206663	1229584977	239050035	2307832
192s robust AVX2	398579702	3101247	3471482061	681555553	4114201
192s robust REF	1708169043	3329677	14878053675	2924080166	13198490
192s simple AVX-512	74521099	1161942	668348451	146852908	1172624
192s simple AVX2	207028321	1614763	1859915777	410688165	2086606
192s simple REF	874233197	1704000	7860638078	1734564453	6336907
256s robust AVX-512	94471316	2938885	1067937038	312182793	3203126
256s robust AVX2	256954790	4010175	2932974894	876938321	5802809
256s robust REF	1132391265	4418712	12871693747	3809296695	19343712
256s simple AVX-512	49911841	1567552	591915343	192665326	1634790
256s simple AVX2	136715548	2133558	1623211830	530350369	2977290
256s simple REF	576428862	2248780	6861202003	2246967862	9451665

Table 6.4: Comparison of the different versions of SPHINCS+ using SHAKE256 in terms of amount of cycles taken to perform hash operations for AVX-512, AVX2 and REF implementations.

For a better comparison, we normalized the values of the experiment by dividing the AVX2 measured values by the AVX-512 measured values, the REF measured values by the AVX2 measured values and the REF measured values by the AVX-512 measured values and took the average of all the results. We put the results in the following figure, Table 6.5.

	Keypair Generation	WOTS pk genration	Signing	FORS signing	Verifying
AVX2/AVX-512	2.784	1.393	2.788	2.758	1.909
REF/AVX2	4.261	1.066	4.255	4.218	3.326
REF/AVX-512	11.864	1.484	11.859	11.640	6.373

Table 6.5: Average of the improvement factors of the results from Table 6.4.

Now, we see a speed-up of around 2.78× for key pair generation, signing, and FORS signing, while we only get 1.39× improvement for WOTS+ public key generation and 1.9× improvement for verification of a signature. In order to understand the motivation behind these speed-ups, we need to consider the benefit of the AVX-512 and consider how SHAKE256 works.

As we have discussed in Section 2.3 the AVX-512 increases the size of the register from 256 bits (present in AVX2) to 512 bits, so one would expect an improvement of at most 2, considering the fact that we might create overheads by using this type of parallelism. However, as we saw in Section 6.2 when we were talking about SHAKE256, not only do we not have overheads, but we get an improvement, as we do not have to perform too many operations on the values, as in SHA256. It can be seen in the improvement of WOTS+ public key generation when comparing AVX2 to REF, we only gain 1.06× improvement, which is mainly due to the fact that REF was not focusing on performance, so there were some minor improvements possible. However, when comparing AVX512 to AVX2, we see a speed-up of 1.39× as we take advantage of faster implementation of the hash. We can see that the speed-up of WOTS+ key pair generation is almost the same as the improvement we get for the hash 1.39× compared to 1.43×, which is mainly due to the fact that the main operation in WOTS+ public key generation is hashing.

When performing key pair generation, signing, and verification, we are mainly performing WOTS+ signing and verification (in which the main operation is hashing) in these operations. Now, since each hashing computation is faster, and since we have to perform most of the operations twice less (as we perform eight values at once instead of four), we get an improvement of more than two. It can be seen as the improvement we get for WOTS+ public key generation times two, meaning $1.39 \cdot 2 = 2.78$, which is around the same speed up we get for those operations.

Finally, in the last operation, verification, we can see that the improvement is smaller. However, it is also the case when comparing the REF implementation to the AVX2. We see a 3.32× improvement for the verification, comparing to around 4.2× in the other operations. Similar to what was explained in verification for the SHA256 implementation, the motivation behind it is that while verifying, we need to wait for values to hash, recall that we are given a leaf and a path. So, unlike in the generation, where we can just run each leaf and path separately, here we need to wait until we computed the lower path, in order to continue with the higher path. In this case, we have some dependencies in the tree, which cause us to perform a little slower, as we cannot simply run everything twice as fast.

6.4 Comparison to Other NIST PQC Candidates

In this section, we are going to compare the speed performance of the six remaining candidates in the third round of the NIST PQC. In this project, NIST defined five levels of security [29], therefore each scheme specifies to which security level it belongs. In this case, we have schemes that comply with four out of those five levels, level 1, level 2, level 3 and level 5. However, not each scheme has an implementation for each level, and most schemes have only three levels of security. In order to compare the schemes, we are going to divide them into three tables, where we compare levels 1 and 2, level 3 and level 5. Since levels 1 and 2 are relatively close to each other security-wise, they can be compared to each other and put in the same table.

In order to optimize the benchmarking, we use SUPERCOP [3], more precisely supercop-20210423, and we use the scheme versions that were present in this version. We also follow the SUPERCOP convention and report the results for the operation on 59 bytes message value.

The results of benchmarking using SUPERCOP on the remaining six NIST PQC can be seen in Tables 6.6 to 6.8 and as explained above, they are divided based on their security level.

scheme	key generation (cycles)	signing (cycles)	verifying (cycles)
dilithium2/avx2	84775	214945	96191
dilithium2aes/avx2	53004	225865	67616
falcon512dyn/avx2	18435111	757966	94450
falcon512tree/avx2	18294471	407077	78340
rainbow1a/avx2	971022592	55819	45061
rainbow1aclassic363232/avx2	9180336	65821	31706
rainbow1acompres363232/avx2	10468189	6688416	3335531
rainbow1acyclicc363232/ssse3	9888685	111910	3107618
rainbow1b/avx2	145158902	193658	154181
rainbow1c/avx2	169424620	102299	80653
bluegemss128/skylake	62636362	124960374	259912
bluegemss128v2/skylake	68260284	133496044	231920
gemss128/skylake	62299388	786229802	271956
gemss128v2/skylake	66942770	693450196	238082
redgemss128/skylake	60081930	4705046	251520
redgemss128v2/skylake	69223506	3941038	240952
picnic2l1fs/optimizedct/avx2	5112	154319440	68282072
picnic3l1/optimizedct/avx2	2934	18650008	13884468
picnicl1fs/optimizedct/avx2	5094	4956484	3932056
picnicl1full/optimizedct/avx2	2938	3650194	2873242
picnicl1ur/optimizedct/avx2	5082	6182712	5024920
sphincsf128harakarobust/aesni	884628	32662282	1772100
sphincsf128harakasimple/aesni	765752	26227775	1086770
sphincsf128sha256robust/avx512	1529710	36586954	3573674
sphincsf128sha256simple/avx512	704568	16357292	1590924
sphincsf128shake256robust/avx512	1437342	33723304	4674484
sphincsf128shake256simple/avx512	786524	18632486	2111900
sphincss128harakarobust/aesni	28010379	546829396	763554
sphincss128harakasimple/aesni	21078955	397786584	470838
sphincss128sha256robust/avx512	80278446	612061142	1405248
sphincss128sha256simple/avx512	34627046	262987792	619820
sphincss128shake256robust/avx512	91769352	690458876	2384902
sphincss128shake256simple/avx512	49965472	379889980	993144

Table 6.6: Results of SUPERCOP of the NIST candidates for NIST security level 1 and 2.

scheme	key generation (cycles)	signing (cycles)	verifying (cycles)
dilithium3/avx2	153241	251128	156909
dilithium3aes/avx2	83469	162561	96869
rainbow3b/avx2	1032425599	600159	524856
rainbow3c/avx2	1451911935	390035	320645
rainbow3cclassic683248/avx2	54411786	399186	199347
rainbow3ccompres683248/avx2	58100021	38807771	19957470
rainbow3ccyclicc683248/ssse3	63646352	504659	20300479
bluegemss192/skylake	311558044	369765050	671846
bluegemss192v2/skylake	349824664	431083596	592532
gemss192/skylake	310017862	2544269142	647096
gemss192v2/skylake	345919488	2535517698	575744
redgemss192/opt	310632024	9055040	665196
redgemss192v2/skylake	352277284	12084184	600970
picnic2l3fs/optimizedct/avx2	8576	466578866	152273198
picnic3l3/optimizedct/avx2	4052	38034828	29144700
picnicl3fs/optimizedct/avx2	8238	11261426	9256502
picnicl3full/optimizedct/avx2	4066	6958596	5615246
picnicl3ur/optimizedct/avx2	8278	14681382	12100836
sphincsf192harakarobust/aesni	1347431	40769290	2730824
sphincsf192harakasimple/aesni	973898	28510255	1750182
sphincsf192sha256robust/avx512	2432422	63533558	6075968
sphincsf192sha256simple/avx512	1023124	27686240	2469838
sphincsf192shake256robust/avx512	2106784	53924202	6471390
sphincsf192shake256simple/avx512	1151354	30121236	2892488
sphincss192harakarobust/aesni	42607808	1291958816	1124046
sphincss192harakasimple/aesni	37067199	966429552	726042
sphincss192sha256robust/avx512	119938826	1111945958	3167410
sphincss192sha256simple/avx512	49573904	472637426	1363204
sphincss192shake256robust/avx512	134521832	1172473256	3258638
sphincss192shake256simple/avx512	73611512	663084066	1690528

Table 6.7: Results of SUPERCOP of the NIST candidates for NIST security level 3.

scheme	key generation (cycles)	signing (cycles)	verifying (cycles)
dilithium5/avx2	234415	445832	244385
dilithium5aes/avx2	127564	235418	144751
falcon1024dyn/avx2	60583627	1510942	189071
falcon1024tree/avx2	59836769	798813	154490
rainbow5c/avx2	4883226585	710293	843879
rainbow5cclassic963664/avx2	217096914	1054799	482671
rainbow5ccompres963664/ssse3	222968004	107506972	44998749
rainbow5ccyclicc963664/avx2	224725909	1054523	45980787
bluegemss256/opt	844345150	577432038	1519252
bluegemss256v2/skylake	1060940250	562046960	1500592
gemss256/skylake	828494972	3534176918	1421404
gemss256v2/skylake	835931232	4208805286	1447230
redgemss256/skylake	849336260	15678678	1517624
redgemss256v2/skylake	1054998820	14620022	1389520
picnic2l5fs/optimizedct/avx2	11830	981924130	266902408
picnic3l5/optimizedct/avx2	5242	64679238	46383352
picnicl5fs/optimizedct/avx2	11032	19159952	15867934
picnicl5full/optimizedct/avx2	5364	11360782	9205348
picnicl5ur/optimizedct/avx2	11138	23972024	19986934
sphincsf256harakarobust/aesni	3212160	83547013	3093406
sphincsf256harakasimple/aesni	2884315	72502328	1929650
sphincsf256sha256robust/avx512	14912084	180516682	10563458
sphincsf256sha256simple/avx512	2047930	42931234	2536494
sphincsf256shake256robust/avx512	5599888	110778770	7193552
sphincsf256shake256simple/avx512	3129072	63184706	3212818
sphincss256harakarobust/aesni	55273977	770314352	1638852
sphincss256harakasimple/aesni	41093470	571192516	994564
sphincss256sha256robust/avx512	142816070	1690745092	6086916
sphincss256sha256simple/avx512	32097622	415021020	1554960
sphincss256shake256robust/avx512	89918536	1021923746	4563864
sphincss256shake256simple/avx512	49497880	589175936	2341038

Table 6.8: Results of SUPERCOP of the NIST candidates for NIST security level 5.

As expected, the improvement does not reach the performance of the lattice-based schemes (Dilithium and Falcon) or multivariate-based schemes (Rainbow and GeMSS). However, the target of SPHINCS+ is different

comparing to those other schemes. SPHINCS+ is used for cases where strong latency requirements are not an issue, like offline code signing or certificate signing. In those scenarios, the signature size (and occasionally public-key size, signing speed, and verification speed) are the most important optimization targets [2] and therefore, SPHINCS+ optimizes based on them rather than on other parameters.

As we can see, there is another symmetric-crypto-based scheme among the NIST PQC scheme, being Picnic. Therefore, we are going to mainly discuss the comparison between those two schemes, as Picnic is the closest scheme to SPHINCS+ and thus, making the comparison more appropriate.

As explained in [2], Picnic has three variants based on different transformations. Without going into too many details, two are based on Fiat-Shamir transform ("Picnic1" and "Picnic2") and another is based on Unruh transform ("Picnic3") (notice that understanding the differences between them is out of scope and the only reason we mentioned them are for naming convention). In the case of Unruh transform, "Picnic3", the signature is 4 times bigger than the ones for SPHINCS+ for similar security levels. In the case of "Picnic2", we get signatures with sizes in a similar ballpark as SPHINCS+. Therefore, to get a balanced comparison, we will compare "Picnic2" to our implementation. In order to highlight the results, we extract the instances we compared from the bigger tables above and present them in Table 6.9.

scheme	key generation (cycles)	signing (cycles)	verifying (cycles)
Security Level 1			
picnic2l1fs/optimizedct/avx2	**5112**	**154319440**	**68282072**
sphincsf128sha256simple/avx512	704568	16357292	1590924
sphincsf128shake256simple/avx512	786524	18632486	2111900
sphincss128sha256simple/avx512	**34627046**	**262987792**	**619820**
sphincss128shake256simple/avx512	**49965472**	**379889980**	**993144**
Security Level 3			
picnic2l3fs/optimizedct/avx2	**8576**	**466578866**	**152273198**
sphincsf192sha256simple/avx512	1023124	27686240	2469838
sphincsf192shake256simple/avx512	1151354	30121236	2892488
sphincss192sha256simple/avx512	**49573904**	**472637426**	**1363204**
sphincss192shake256simple/avx512	**73611512**	**663084066**	**1690528**
Security Level 5			
picnic2l5fs/optimizedct/avx2	**11830**	**981924130**	**266902408**
sphincsf256sha256simple/avx512	2047930	42931234	2536494
sphincsf256shake256simple/avx512	3129072	63184706	3212818
sphincss256sha256simple/avx512	**32097622**	**415021020**	**1554960**
sphincss256shake256simple/avx512	**49497880**	**589175936**	**2341038**

Table 6.9: Comparison of Picnic and SPHINCS+ from the results of Tables 6.6 - 6.8, where the values discussed below are highlighted.

For our comparison, we are going to consider the s option for SPHINCS+ as it relies on fewer security assumptions (therefore considered more secure). However, for completeness, we included the f option as well. Now, as can be seen in Table 6.9 for security level 1, we get a slower signing time by 70% for SHA256 and a slower signature time of 0.4× for SHAKE256 for SPHINCS+ comparing to Picnic. For the same security level, we also get much slower key generation, but verification is more than 110× faster for SHA256 and more than 68× for SHAKE256 for SPHINCS+ comparing to Picnic. In the case of security level 3, we only get a slower signing time of 1%, which means that they take approximately the same amount of time for signing using SHA256 and a slow down of 42% using SHAKE256. Again, generation is much slower, but verification is more than 111× faster for SHA256 and more than 90× faster for SHAKE256. For security level 5, we also get an improvement for signing, where for SHA256, we get more than 2× faster and for SHAKE256, we get more than 66% faster. Again key generation is much slower, but verification is 171× faster for SHA256 and 114× faster for SHAKE256.

Chapter 7

Conclusions and Future Work

In this thesis, we discussed how SPHINCS+ algorithm works and we have seen an approach to improve its performance using the AVX-512 vector extensions. We also understood how well the proposed implementation is compared to the previous AVX2 implementation and compared to other NIST PQC candidates.

The results look promising, but there are still some improvements that can be done in order to optimize the speed performance of SPHINCS+ further using AVX-512. Such improvements could involve optimizing small functions implemented in assembly faster or more complicated implementation features on the scheme itself. Next, we are going to discuss those ideas which could be seen as ideas for future work on improving the performance of SPHINCS+.

Examples of optimizations using assembly might entail for instance a function that takes 64 bits integer from 512 bits integer value. As it stands now, we first need to extract 256 integer values and only then 64 int value. By creating such a function, we can reduce a cycle for each time we extract a 64-bit value, which is mainly done in the squeezing part of SHAKE256.

Another function that can be written in assembly is a concatenation function of two 256 bits integers to one 512 bits integer. Currently, this is done using permutation but can be more efficiently done.

A more efficient implementation of the shuffling instruction *_mm512_permutex2var_epi64*, used in the transpose part in the SHA256 implementation, could also improve the space and speed performance of the code. In this function, we pass a 512-bit integer value to specify the

shuffling order. However, there is no use for all the bits and a simple 64-bit integer would be sufficient to cover all cases. Furthermore, in the code this function is used to concatenate two 256-bit values, so if implemented in assembly, we will no longer need to pass any additional values but only the two values we wish to concatenate. This might increase the speed performance as well since there is no longer a need to create a 512-bit value to pass to this function.

Apart from those smaller functions, which could provide a small improvement for the proposed version in this thesis, other ideas might result in more significant changes to the code. One approach could be to parallelize independent vector operations and integer operations in order to increase performance. While the integer operations are relatively bigger, compared to vector operations, they leave space in the CPU that can still be used. Whereas this space is too small for another integer operation, it might be enough for vector operation. Thus, we will create better parallelism by executing those operations at the same time.

While working on the f option for SHA256, we saw that due to the fact that we work on one tree at a time, we do not have enough leaves and therefore, we cannot perform 16 hashes at once. An idea can be to modify the code to work on multiple trees at once. This way, we will have more leaves and we will be able to perform the 16 times hash for this option as well. This might also be useful when and if Intel will release a new version for the AVX, which works with 1028 bits registers.

All in all, we have seen an approach for optimizing the NIST PQC candidate SPHINCS+. The results we got are a significant first step towards optimal implementation, but, as it stands for optimizations, there is always a place for improvement and we intend to explore a number of different ideas to such end in the future.

Bibliography

[1] D.M. Alter. sphincsplus AVX-512 implementation. *GitHub*, 2021. https://github.com/DorAlter/sphincsplus/tree/master.

[2] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe. The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security,*. New York : ACM, 2019. https://dl.acm.org/doi/10.1145/3319535.3363229.

[3] D. J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, accessed on May 2021. https://bench.cr.yp.to.

[4] D.J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M.M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, and P. Schwabe. SPHINCS+ - Submission to the NIST post-quantum cryptography project, 2017. Submission available at https://sphincs.org.

[5] D.J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M.M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, and P. Schwabe. SPHINCS+ – Submission to the 3rd round of the NIST post-quantum project, 2020. Submission available at https://sphincs.org.

[6] D.J. Bernstein, D. Hopwood, A.T. Hülsing, T. Lange, R.F. Niederhagen, L. Papachristodoulou, P. Schwabe, and Z. Wilcox O'Hearn. *SPHINCS: Practical Stateless Hash-Based Signatures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. https://link.springer.com/chapter/10.1007/978-3-662-46800-5_15.

[7] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier. KangarooTwelve: fast hashing based on Keccak-p. Cryptology ePrint Archive, Report 2016/770, 2016. https://eprint.iacr.org/2016/770.

[8] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the Security of the Winternitz One-Time Signature Scheme. In *International Journal of Applied Cryptography*, volume 3, pages 363–378, 06 2011. https://link.springer.com/chapter/10.1007/978-3-642-21969-6_23.

[9] J. Buchmann, E. Dahmen, and A. Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions, 2011. https://eprint.iacr.org/2011/484.

[10] A. Casanova, J. C. Faugere, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. GeMSS: A Great Multivariate Short Signature, 2020. https://www-polsys.lip6.fr/Links/NIST/GeMSS_specification_round2.pdf.

[11] A.V. Cueva. The Intel Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length Extensions Feature on Intel® Xeon® Scalable Processors. *Intel*, 2018 accessed on May 2021. https://software.intel.com/content/www/us/en/develop/articles/the-intel-advanced-vector-extensions-512-feature-on-intel-xeon-scalable.html?wapkw=advanced%20vector%20extensions.

[12] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *anger.org*, 2021 (last update) accessed on May 2021. https://agner.org/optimize/instruction_tables.pdf.

[13] P. A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, Pornin T, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU, 2020. https://falcon-sign.info/falcon.pdf.

[14] P. Gepner. Multi-Core Processors: New Way to Achieve High System Performance. In *PARELEC 2006 - Proceedings: International Symposium on Parallel Computing in Electrical Engineering*, pages 9–13, 01 2006. https://ieeexplore.ieee.org/document/1698630?arnumber=1698630.

[15] A. Hülsing, L. Rausch, and J. Buchmann. Optimal Parameters for $XMSS^{MT}$. Cryptology ePrint Archive, Report 2017/966, 2017. https://eprint.iacr.org/2017/966.

[16] A. Hülsing. W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes. In *International Conference on Cryptology in Africa*, volume 7918, pages 173–188, 06 2013. https://link.springer.com/chapter/10.1007/978-3-642-38553-7_10.

[17] Intel. Intrinsics for Intel Advanced Vector Extensions 2. *Intrinsics for Intel Advanced Vector Extensions 2*, 2013 accessed on April 2021. https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/composer_xe_2013/Documentation/en_US/compiler_c/main_cls/index.htm#GUID-9E84F9C5-1711-4F59-8742-8F9DF283A472.htm.

[18] Intel. Intel Intrinsics Guide. *Intel official website*, accessed on June 2021. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[19] J.Ding, M. S. Chen, A. Petzoldt, D. Schmidt, B. Y. Yang, M. Kannwischer, and J. Patarin. RAINBOW, 2020. https://www.pqcrainbow.org/.

[20] M. Lhoussein, H. Sylvain, H. Dominique, B. Said, Abdelkrim H., and Z. Yahya. Efficient adaptive load balancing approach for compressive background subtraction algorithm on heterogeneous CPU–GPU platforms. *Journal of Real-Time Image Processing*, 17, 2020. https://link.springer.com/article/10.1007/s11554-019-00916-4.

[21] C. Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011, accessed on May 2021. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html.

[22] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehle, and S. Bai. CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation, 2020. https://pq-crystals.org/dilithium/data/dilithium-specification-round3.pdf.

[23] R. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer Berlin Heidelberg, 1988. https://link.springer.com/chapter/10.1007/3-540-48184-2_32.

[24] M. Mosca and M. Piani. Quantum Threat Timeline Report 2020, 2021. https://globalriskinstitute.org/publications/quantum-threat-timeline-report-2020/.

[25] G. Alagic (NIST), J. Alperin-Sheriff (NIST), D. Apon (NIST), D. Cooper (NIST), Q. Dang (NIST), J. Kelsey (NIST), Y. Liu (NIST), C. Miller (NIST), D. Moody (NIST), R. Peralta (NIST), R. Perlner (NIST), A. Robinson (NIST), and D. Smith-Tone (NIST). Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. *NISTIR 8309*, 2020. https://csrc.nist.gov/publications/detail/nistir/8309/final.

[26] National Institute of Standards and Technology. SECURE HASH STANDARD. *Federal Information Processing Standards Publication (FIPS) 180-2*, 2002. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf.

[27] National Institute of Standards and Technology. What is Yo? (Yo App documentation). *Yo App website*, 2014, accessed on April 2021. https://docs.justyo.co/.

[28] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Federal Information Processing Standards Publication (FIPS) PUB 202*, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[29] National Institute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. *NIST official website*, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[30] National Institute of Standards and Technology. Round 1 Submissions. *NIST official website*, 2019. https://csrc.nist.gov/Projects/post-quantum-cryptography/Round-1-Submissions.

[31] National Institute of Standards and Technology. Round 2 Submissions. *NIST official website*, 2020. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions.

[32] National Institute of Standards and Technology. Round 3 Submissions. *NIST official website*, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[33] E. Rescorla and T. Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008. https://datatracker.ietf.org/doc/html/rfc5246.

[34] V.A. Shargatov, A.S. Pecherkin, A.S. Sofin, A.A. Agapov, S.V. Gorkunov, S.I. Sumskoi, Y.A. Bogdanova, and A.V. Karabulin. Modeling of Shock Wave Propagation Over the Obstacles Using Supercomputers. In *Journal of Physics*, volume Conference Series 1099 (1), 2014. https://iopscience.iop.org/article/10.1088/1742-6596/1099/1/012014.

[35] P. 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://dx.doi.org/10.1137/S0097539795293172.

[36] R. Sole. AVX-512: qué es y para qué sirve estas instrucciones que debes tener en cuenta. *Professional review*, 2021, accessed on June 2021. https://www.profesionalreview.com/2021/06/12/intel-avx-512/.

[37] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture*, pages 248–252, 2005. https://ieeexplore.ieee.org/document/1385946.

[38] B.T. Sutcliffe, J. Tennyson, and S. Miller. The Use of Supercomputers for the Variational Calculation of Ro-Vibrationally Excited States of Floppy Molecules. In *Theoretica Chimica*, volume 72(4), pages 265–276, 1987. https://www.semanticscholar.org/paper/The-use-of-supercomputers-for-the-variational-of-of-Sutcliffe-Tennyson/747a55d6457a977006937048dc376378030c2bc9.

[39] SPHINCS+ Team. sphincsplus. *GitHub*, accessed on November 2020. https://github.com/sphincs/sphincsplus.

[40] Keccake team XKCP. eXtended Keccak Code Package. *GitHub*, accessed on November 2020. https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600-times8/AVX512.

[41] G. Zaverucha, M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, J. Katz, X. Wang, V. Kolesnikov, and D. Kales. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. Cryptology ePrint Archive, Report 2017/279, 2017. https://ia.cr/2017/279.