Radboud University Nijmegen

Faculty of Science

Optimizing SPHINCS$^+$ on the Raspberry Pi 4

Parallelising SHA-256 using NEON

Thesis BSc Computing Science

Author:
Martin Lehmann
s4577353

First supervisor:
Prof Dr Peter Schwabe
peter@cryptojedi.org

Second supervisor:
Dr Bas Westerbaan
bas@westerbaan.name

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

March 5, 2022

Contents

List of Figures

List of Tables

List of Codes

List of Names and Abbreviations

A Alice

ADD Bitwise add operation

AES Advanced Encryption Standard

AND Bitwise and operation

ARM Acorn RISC Machine

ARMv8 Eigth version of the ARM processor architecture

ASIMD Advanced Single Instruction Multiple Data

B Bob

Cortex-A72 Processor implementing ARMv8 architecture

CPU Central Processing Unit

CTR Cycle counter register

Ed25519 Elliptic curve-based asymmetric cryptographic algorithm

FORS Forest Of Random Trees

MSS Merkle Signature Scheme

NEON ASIMD coprocessor on ARM processors

NIST National Institute of Standards and Technology

NOT Bitwise not operation

ROTR Rotate Right

SHA-256 Secure Hash Algorithm with 256-bit output length

SPHINCS Stateless Practical Hash-based Incredibly Nice Cryptographic Signatures

SPHINCS$^+$ Improved Stateless Practical Hash-based Incredibly Nice Cryptographic Signatures

SUPERCOP System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives

WOTS Winternitz One-Time Signature

WOTS$^+$ The improved Winternitz One-Time Signature

XOR Exclusive Or

Abstract

SPHINCS$^+$ is a contender in the current NIST competition for a post quantum secure cryptographic standard. Compared to currently used standards, such as Ed25519, SPHINCS$^+$ is very slow. Being a hash-based signature scheme utilising a hypertree of WOTS$^+$, FORS and Merkle trees, SPHINCS$^+$ is very parallelisable. The aim of this work was to determine the effect of parallelising one of the underlying hash functions, namely SHA-256, on the performance of SPHINCS$^+$, specifically on the Raspberry Pi 4. This was done using the NEON coprocessor of the Cortex-A72. Using NEON SHA-256 digests were computed up to 27.2% faster. This did not carry over into SPHINCS$^+$ which was not significantly faster but up to 50.8% slower than with regular SHA-256. This is unlikely to be representative for all ARMv8 CPUs.

Chapter 1

Introduction

Computers run on ones and zeros and they connect the world via the internet. Quantum computers may change both of these facts. Unlike regular computers, a quantum computer does not work with bits holding ones and zeros, but qubits that hold the superposition state of two classical states [1].

Currently, large efforts in quantum research are made to build a European quantum network [2]. While this research may enables new ways of communicating, quantum computers also pose a threat for our current communication. On a sophisticated quantum computer, specific algorithms such as Shor's algorithm will be able to break factorization based cryptographic schemes used in today's cryptographic algorithms making them obsolete [3, 4]. With their "Call for Proposals for Post-Quantum Cryptography Standardization" the National Institute of Standards and Technology (NIST), a major authority on cryptographic standards based in the USA, launched a competition to find a new, quantum secure cryptographic standard [5].

One of the candidates is SPHINCS, or Stateless Practical Hash-based Incredibly Nice Cryptographic Signatures, which is being developed by a team led by Andres Hülsing of the Eindhoven University [6, 7]. SPHINCS is a signature scheme utilising hash functions to generate signatures without needing to keep track of a state. Its security is based on the second preimage resistance of the underlying hash function. Grover's algorithm is a possible attack vector against this security, but the threat of Shor's algorithm for cryptography based on the hardness of factorization is more relevant [8].

However, SPHINCS$^+$ has a major disadvantage in its performance and signature size. While SPHINCS$^+$ may be a fast algorithm in its class of stateless hash-based signature schemes, it is extremely slow compared to the current, non-post quantum standard Ed25519. On the Raspberry Pi 4 which implements the widely used Cortex-A72 architecture, the computation of a single Ed25519 signature takes around 130 thousand CPU cycles. For a single SPHICNS$^+$ signature 140 million CPU cycles are needed even when using the fastest implementation `sphincsf128sha256simple` [9].

Currently, SPHINCS$^+$ is specified to use one of three underlying hash functions: SHA-256, SHAKE256, and Haraka. The presented thesis will focus on SHA-256 instances of SPHINCS$^+$ exclusively. The goal of this study was to determine the effect of parallelising SHA-256 on the performance of SPHINCS$^+$. This was to be achieved by using NEON, an Advanced Single Instruction Multiple Data (ASIMD) extension present on the Cortex-A72 of the Raspberry Pi 4. Popular chips such as the Snapdragon 652 processor also implement the Cortex-A72 [10]. Any achieved performance increase should carry over to the many devices using it.

In **Chapter 2** aspects of SPHINCS$^+$, SHA-256, and the Raspberry Pi 4 relevant to this thesis are presented. The research of this thesis is put into the context of the state of the art in the field in **Chapter 3**. **Chapter 4** focuses on the implementation in the C programming language as well as the theoretical lower bound of the implementation. The results are presented in **Chapter 5** and discussed in **Chapter 6**. Finally, a conclusion is drawn and an outlook is given in **Chapter 7**. The most important features of the implementation can be found in **Appendix** A. **Appendix** B contains the steps necessary to reproduce the raw data displayed in **Appendix** C.

Chapter 2

Preliminaries

For the cryptographic actions in SPHINCS$^+$ such as key generation, signing and verifying a hyper tree, a tree of trees, is used. This hyper tree combines Merkle trees, the Winternitz one-time signature scheme and the forest of random trees. The main components of SPHINCS+ are highly parallelisable implying potential performance increases of the overall algorithm. The most relevant to this thesis are the Merkle trees and the Winternitz one-time signature scheme which will be shown to be parallelisable. Further, hash-based signatures, SHA-256, SPHINCS$^+$ itself and key characteristics of the Raspberry Pi 4 are explained.

2.1 Hash-Based Signatures

In this section the history and development of hash-based signatures relevant for the understanding of SPHINCS$^+$ are discussed. Hash-based signature schemes use hash functions to generate signatures of messages reducing the security requirements to those of the hash functions used. The Lamport one-time signature scheme was the first hash-based signature scheme. It was immediately improved upon by Merkle using a binary tree approach [11]. Winternitz proposed a different approach to Merkle which became the Winternitz One-Time Signature (WOTS) scheme. These schemes share several concepts such as

- the one-way function **F**
- the client secrets $s_i = x_1^i, ..., x_{|s_i|}^i$
- the private keys k_i
- public keys $Y_i = \{y_1, ..., y_{|Y_i|}\}$

The notation $|x|$ is used to represent the length of x either in bits or elements. In the context of SPHINCS$^+$, SHA-256 serves as the one-way

function **F** and will be explained in a later section. In his 1989 paper "A Certified Digital Signature" Merkle uses the example of phone brokerage [11]. Here, Alice (A) seeks to authenticate herself towards her broker Bob (B) in order to conduct business on her account. We will return to this example repeatedly throughout this section.

2.1.1 Lamport One-Time Signature Scheme

Lamport described the first iteration of hash-based signatures in 1979 [12]. Being the first of its kind, the scheme was still rather simple and vulnerable to attacks.

Consider the classical, ever communicating parties Alice (A) and Bob (B). Using the example of phone brokerage say A holds some stock and B is her broker. When A wants to sell stock, she wishes to not have to go through a lengthy process of authentication. Instead, A chooses a secret s and generates a private key k_i for each bit of the secret. She then computes the public key $Y = \{y_1, ..., y_{|s|}\}$ where $y_i = F(k_i)$. B receives Y from A and stores it. To authenticate herself, A sends the keys k_1 through $k_{|s|}$ to B. B then computes Y' from the keys to then check whether $Y = Y'$. If so, A is authenticated and business may commence.

Lamport explained that such a function **F** must have two properties. It must be preimage and second-preimage resistant [12]. Thus, given a digest d it must be infeasible to find a message m such that $F(m) = h$. Also, given a message m_1 it must be infeasible to find a second message m_2 such that $F(m_1) = F(m_2)$. Secrets are not digested in whole, but bit by bit, meaning a secret s of length l yields a vector of public keys of length l making signatures very long and memory intensive [11].

2.1.2 Merkle Signature Scheme

Ten years after Lamport published their signature scheme, Merkle described his approach to hash-based signatures. He aimed to improve on the problem of Lamport's scheme being memory intensive [11].

Merkle ran the following thought experiment: Say we have a one-way function **F** that outputs 100-bit message digests. Returning to the example of phone brokerage, if our broker B had to hold 1000 signatures of messages of length $l = 1000$ they would need around 2.5 megabytes of storage per client. As a broker usually is not limited to a single client, we further assume they hold signatures for 1000 clients. The broker B would then have to hold on to 2.5 gigabytes of data.

In 1989, this "hardly seem[ed] economical" [11]. Instead, Merkle suggested constructing a binary tree that has the secrets in the leaves and hashes these up towards the root. Here, hashing up means that any node that is not a leaf node concatenates the values stored in its children and applies **F** to the concatenation. The value in the root is a hash of hashes that

depends on all values of the leaf nodes and forms the public key that the broker from earlier would hold [11]. The authentication path of the secret which the fifth leaf holds can be seen in Figure 2.1 in yellow.

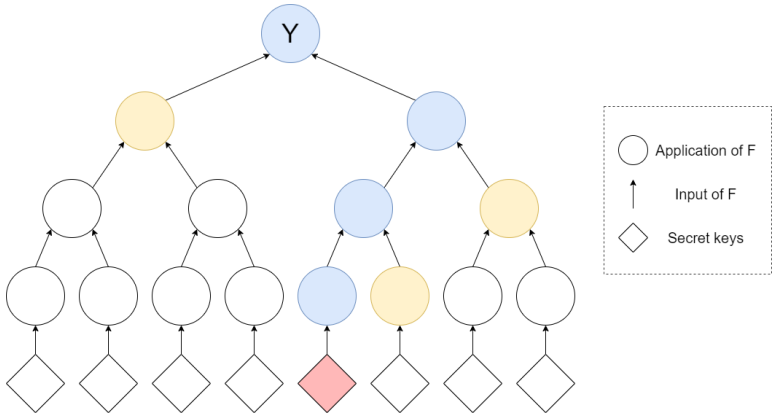


Figure 2.1: Authentication path (yellow) for secret 5 (red) and values computed (blue) to verify secret 5. The root node holds public key (Y).

Again, A generates secret keys for each bit of the message and applies **F** to them. The outputs of **F** form the leaves of the tree. A computes the root, which serves as public key Y, from the leaves and sends it to B. For authentication, A sends the first secret key and its authentication path to B. B computes Y' from the secrets they received from A. If the $Y' = Y$, the first secret is authenticated. This process is repeated for all leaves of the tree. If the root value can be correctly computed from each secret and authentication path, A is authenticated.

Using the Merkle signature scheme the signature of arbitrarily long messages is always the same. The public key is no longer a vector but a single value. However, as Merkle mentions, it can be improved further using the approach proposed by Winternitz [11].

2.1.3 Winternitz One-Time Signature Scheme

Shortly before Merkle published his paper he was contacted by Winternitz who suggested to sign several bits at once in the bottom layer of the tree [11]. Merkle prognosed that this would reduce the memory required to store public key even further [11].

In Winternitz' approach, a message m is split into n blocks $\{m_1, ..., m_n\}$ of length l from which the Wintznitz parameter $w = 2^l$ is computed. Additionally n keys $\{k_1, ..., k_n\}$ are generated from a seed. Then, for all message blocks the according public key is computed by applying **F** w times to the blocks private key: $y_i = F^w(k_i)$.

In the phone broker example, A will share these with B. A holds on to the intermediate values, the secrets $s_1, ..., s_n$ which are $s_i = F^{m_i}(k_i)$. In

order to authenticate herself to B, she will then send these secrets to B who computes $y'_i = F^{w-m_i}(s_i)$. If $y_i = y'_i$ for all secrets, A is authenticated.

During the transmission of the secrets, an adversary may be able to produce a forgery of the signature. In order to do so the adversary simply applies **F** again and send $F(s_i)$ claiming the original message block was indeed $m_i + 1$. The verifier will compute $y'_i = F^{w-m_i-1}(F(s_i))$ and find that $y = y'$. The adversary has successfully forged a part of the signature.

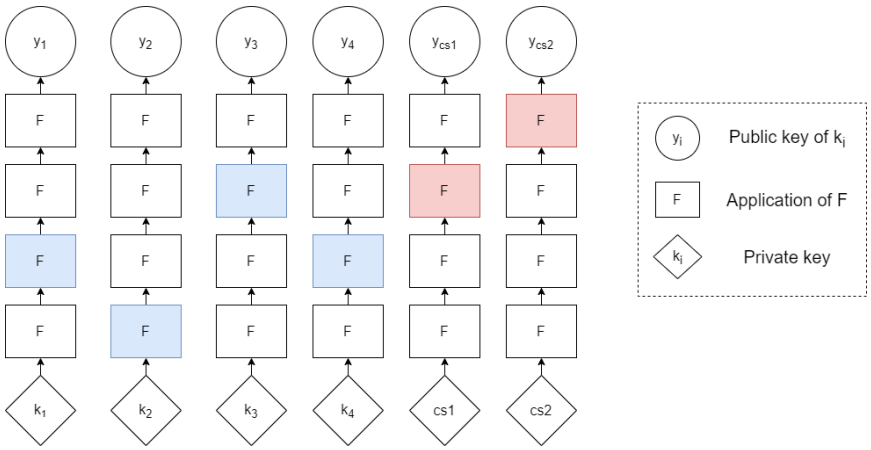


Figure 2.2: Small instance of WOTS. In blue the secrets computed for the message $m = 01|00|10|01$, in red the secret computed for its checksum blocks $cs_1 = 01$ and $cs_2 = 00$.

To prevent such attacks a check sum cs is introduced. The checksum is the sum of the message blocks and is also split into blocks such that $cs_i < w$. However, in order for the checksum not to suffer from the same flaw, it is signed in an inverted manner with $y_{cs_i} = F^{w-cs_i}(cs_i)$ as can be seen in Figure 2.2. Now, to accomplish the previous attack, the attacker has two equally infeasible approaches: Apply **F** to one of the secrets and compute a preimage of the check sum or apply **F** to the checksum and compute the preimage of one of the message blocks. As **F** is preimage resistant, this is not a realistic attack.

2.2 SPHINCS$^+$

As previously mentioned, SPHINCS$^+$ utilises Merkle trees together with WOTS$^+$, a modified version of WOTS, and FORS. WOTS$^+$ reduces the security to second preimage resistance. By doing this collision resistance is not a requirement any more making the scheme overall collision resilient [13].

The inner workings and even the direct effects of the FORS structure at the bottom of the hyper tree are not relevant to this thesis. Indeed relevant

to this thesis is that the function **F** applied in the WOTS and Merkle tree is called several thousand times depending on the instance of SPHINCS$^+$.

A small example of SPHINCS$^+$ can be seen in Figure 2.3. This instance features Merkle trees with a height of three. With this height they have eight leaves. As three of these Merkle trees feed into another, a total of $8 * 8 * 8 = 512$ or 2^9 FORS instances are at the bottom. The squares below the Merkle Trees represent the instances of the WOTS scheme, the rhombus at the bottom represents the instances of the FORS few-times signature scheme. Further, marked in grey are the nodes of the authentication path.

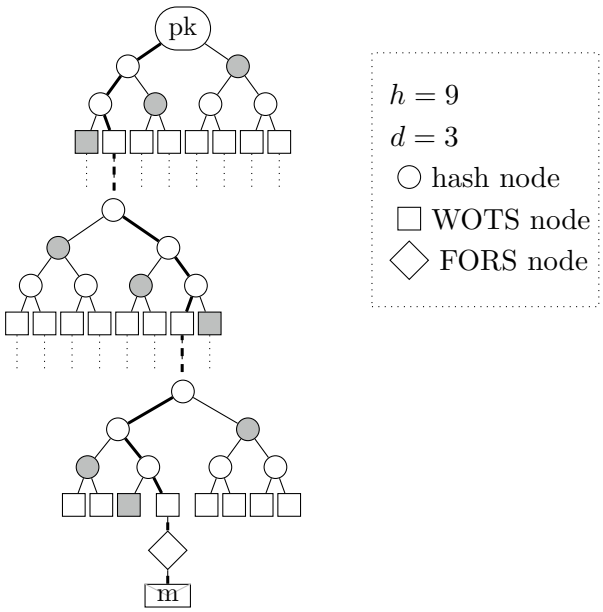


Figure 2.3: Small instance of SPHINCS$^+$. Grey nodes represent the authentication path for message m. Figure adapted from [13].

SPHINCS$^+$ comes in two variants, simple and robust. The simple variant applies **F** directly to a given messages while the robust variant first uses **F** to generate a bit mask, applies the bit mask to the message and then applies **F** on the masked message [14]. Instances of SPHINCS$^+$ are further described by the parameters associated with them. The parameters of SPHINCS$^+$ are:

- w - The Winternitz parameter
- n - The output length of **F**
- d - The number of layers in the hypertree
- h - The height of the hypertree

- b, k - Parameters for FORS

These parameters are determined by choosing the number of messages to be signed with an instance of SPHINCS$^+$ and the degree of security the scheme should have. Additionally, it must be chosen whether the instance should be optimized for speed or size. A database of parameter sets is then searched for the optimal combination of parameters to achieve the desired security degree and optimization goal [13]. The instances are named by placing an `f` for fast of `s` for size after `sphincs`. This is followed by target bit security, the underlying function **F** and the variant. A speed optimized, simple instance of SPHINCS$^+$ using SHA-256 with a target security of 128 bits is referred to as `sphincsf128sha256simple`.

2.3 SHA-256

SHA-256 is a hash algorithm of the SHA2 family [15]. It takes inputs of arbitrary length and splits them into 512-bit message blocks to digest them. The output is a 256-bit hash or digest of the input message [16].

The 2048-bit key k and 256-bit initialisation vector IV used in SHA-256 are fixed. In addition to the key and IV, a message schedule w is created for each 512-bit block. The message schedule w consists of sixty-four 32-bit words. The first 16 words of w, i.e. $w_0, ..., w_{15}$, contain the entire message block in big-endian fashion. Each subsequent word in w depends on the previous words. Define s_{0_i}, s_{1_i} to be

$$s_{0_i} = (w_{i-15} \ggg 7) \oplus (w_{i-15} \ggg 18) \oplus (w_{i-15} \gg 3)$$

and

$$s_{1_i} = (w_{i-2} \ggg 17) \oplus (w_{i-2} \ggg 19) \oplus (w_{i-2} \gg 10)$$

then

$$w_i = w_{i-16} + s_{0_i} + w_{i-7} + s_{1_i}$$

for $16 \leq i \leq 63$.

SHA-256 uses eight 32-bit registers a–h. These are initially set to the IV and then used to compute the variables *choose, majority*, Σ_0, and Σ_1:

$$choose = (e \cdot f) \oplus (\neg e \cdot g)$$

$$majority = (a \cdot b) \oplus (a \cdot c) \oplus (b \cdot c)$$

$$\Sigma_0 = (a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)$$

$$\Sigma_1 = (e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)$$

Further, the intermediate values t_1 and t_2 are used:

$$t_{1_i} = h + choose_i + w_i + k_i + \Sigma_{1_i}$$

$$t_{2_i} = \Sigma_{0_i} + majority_i$$

where $1 \leq i \leq 64$.

In the main body of SHA-256, called the update function, these variables are used in a loop of 64 iterations as depicted in Code 2.1.

In each iteration, the values of *choose, majority*, Σ_0, Σ_1, t_{1_i} and t_{2_i} are computed anew. Finally, after the last iteration is finished, the digest is updated to the sum of its previous contents and the registers *a-h*. Once the final message block has been digested, the overall digest is returned as the concatenation of the values of the digest data structure *dig*.

First the content of the *IV* is copied into the data structure holding the digest *dig* during initialization. The *init()* function is called once per message while the update function is called for each message block.

```
1  void init(){
2      for (int i = 0; i < 8; i++){
3          dig[i] = IV[i];
4      }
5  }
6
7  void update(){
8      compute message schedule w0,...,w63
9      compute t1i, t2i, Sigma0 and Sigma1
10
11     for (int i = 0; i < 64; i++){
12         h = g;
13         g = f;
14         f = e;
15         e = d + t1_i;
16         d = c;
17         c = b;
18         b = a;
19         a = t1_i + t2_i;
20     }
21
22     dig[0] = a + dig[0];
23     ...
24     dig[7] = h + dig[7];
25 }
```

Code 2.1: Initialisation of the digest data structure and the update function of the SHA-256 hash algorithm.

2.4 Raspberry Pi 4

The Raspberry Pi 4 is a small form factor, low-cost computer. The most important feature of the Raspberry Pi 4 for this thesis is the Broadcom BCM2711 chip. This chip is equipped with the Cortex-A72 processor which is built on the ARMv8 instruction set. It has four cores running at a base clock speed of 1.5 gigahertz. It further includes the 32-bit and 64-bit ARM

instruction sets as well as a large subset of the NEON ASIMD instruction set [17]. ARMv8 features NEON instructions for the cryptographic primitives of AES, SHA-1 and SHA-2 [18]. While these instructions are part of the specification of the Cortex-A72, they are not available on the Broadcom BCM2711 chip [19].

2.4.1 Architecture

Important features of the ARM Cortex-A72 include its registers, pipelining, the NEON coprocessor and its cycle counter register CTR. Unless stated differently, the following information has been summarised from the documentation [18].

2.4.1.1 Registers

The Cortex-A72 has thirty-two 64-bit general-purpose registers as well as seven status registers of also 64 bits. When in ARM state, 16 data registers are accessible at any given point in time. r0 through r13 are general-purpose while r14 serves as link register and r15 as program counter.

The NEON coprocessor has its separate register bank with sixteen 128-bit registers Q0 through Q15. These can also be interpreted as thirty-two 64-bit double registers then called D0 through D31.

2.4.1.2 Pipeline

As the Cortex-A72 is a pipelined and out-of-order processor, it allows for multiple instructions to be processed simultaneously. As long as two instructions are independent of each other, meaning the result of one does not influence that of the other, they may be executed directly after one another. Otherwise, the first instructions latency has to be accounted for.

The pipeline is divided into four high-level phases which are part of one of two categories: in-order or out-of-order. These phases and categories can be seen in Figure 2.4. The in-order phases include the fetch phase and the decode-rename-dispatch phase. In the fetch phase instructions are fetched from memory and provided to the next phase which decodes it into possibly smaller operations, renames registers accordingly and then dispatches them to the next phase.

The issue phase, the first of the out of order phases, issues the received operation to one of eight execution pipelines (see Figure 2.4). Each of the execution pipelines has the capacity to accept and complete one operation of its specific operation set per cycle.

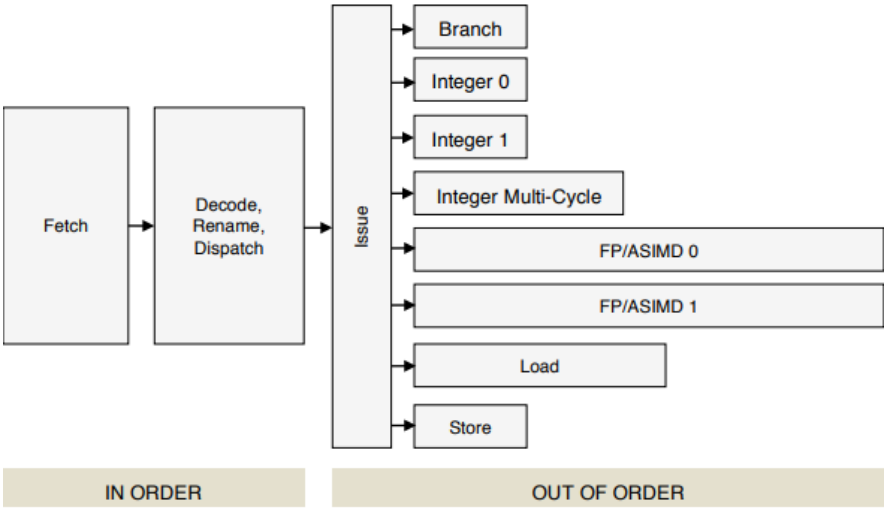


Figure 2.4: Pipeline of Cortex-A72. Figure taken from [18].

2.4.1.3 NEON

The NEON coprocessor present on the Cortex-A72 enables the use of ASIMD instructions. Unlike regular ARM instructions which take general-purpose registers as arguments, NEON instructions take NEON registers which may hold multiple values at once. The NEON instructions then apply a given operation on all or one of the values in the registers at once. For example, the instruction in Code 2.2 takes each of the four 32-bit integers in the register V2, multiplies them by the second 32-bit integer in V3 and stores the resulting values in register V0 as visualised in Figure 2.5.

```
1    MUL V0.4S, V2.4S, V3.S[1]
```

Code 2.2: Simple example of an ASIMD instruction.

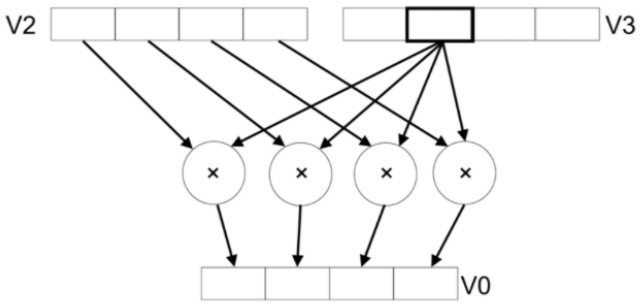


Figure 2.5: SIMD visualisation for Code 2.2. Figure taken from [20].

The Broadcom BCM2711 does not implement the cryptographic extension for NEON. SHA-256 must hence be implemented using the bitwise operations of NEON rather than the hardware implementation of SHA-256 present on other Cortex-A72 CPUs [19]. Since SHA-256 operates on 32-bit words with 32-bit operations four SHA-256 digests can be computed in parallel with NEONs 128-bit vector registers. This is the major challenge of the presented thesis.

2.4.2 Dual Issue

The NEON coprocessor supports a feature called dual issue. The dual issue feature allows for two independent instructions to be dispatched to the arithmetic unit and load/store unit at the same time. The two instructions are then processed in parallel. This feature can greatly improve code performance by smartly interleaving load/store and arithmetic instructions.

2.4.2.1 Cycle Counter

As many processors do, the Cortex-A72 has a register that stores the number of cycles that have passed, the CTR. In order to access it, however, it must be enabled in kernel mode. The kernel module written by GitHub user jerinjacobk accessible at www.github.com has been used to make it available in user mode [21].

Chapter 3

Related Work

Van de Linde implemented parallelised versions of SHA-256 and SHA-512 using NEON in 2016. Their implementation was made in assembly on the Cortex-A8 processor, a processor built on the ARMv7 32-bit architecture. Significant performance improvements of 33.5% and 27.5% respectively were achieved with their implementation [22].

In 2020 Fujii *et al.* published their work on NEON parallelised (AES) for ARMv8 CPUs that do not feature the cryptographic NEON extension such as the Raspberry Pi 4 [23]. This year, 2021, Nguyen and Gaj were able to substantially improve the performances of CRYSTALS-Kyber, NTRU and Saber [24]. They worked with the recent Apple M1 "Firestorm" chip as well as the Raspberry Pi 4's Cortex-A72 using NEON on both. Their implementation was furhter improved by Becker *et al.*. Combining methods for fast division of large numbers and modular multiplication, a speed up by a factor of 1.7 was achieved [25].

Oliveira and López improved the performance of the hash-based Merkle Signature Scheme (MSS) on an Intel Haswell processor using AVX2 instructions in 2015. This was achieved with vectorized versions of SHA-256 and SHA-384 speeding up key generation, signing, and verifying [26]. Dor Alter is currently implementing SPHINCS$^+$ using SHA-256 and Keccak on an Intel processor using AVX512 instructions under the supervision of Prof Peter Schwabe at the Radboud University.

Our work aims to provide improvements for SHA-256 based SPHINCS$^+$ implementation on processors of the ARMv8 architecture. We use the C programming language on the Cortex-A72 of the Raspberry Pi 4 with the goal of determining the performance improvement of SPHINCS$^+$ when using a NEON parallelised version of SHA-256 that does not use the cryptographic extension of the Cortex-A72.

Chapter 4

Implementation

This chapter will explain the approach taken for the implementation and provide a lower bound performance analysis. The code structure and aspects of the NEON instructions generated by our implementation are central to this computation.

4.1 Code Structure

The SHA-256 algorithm had to be implemented using bitwise operations as the cryptography extension of NEON is not available on the Raspberry Pi 4. Our implementation is based on the SHA-256 implementation by Bernstein contained in the SUPERCOP benchmark suite. The SUPERCOP implementation is also the default SHA-256 implementation within SPHINCS$^+$. Further, the SUPERCOP implementation was used for benchmarking in Chapter 5.

For our implementation we adapted core functions to take four inputs and produce four outputs instead of one input and one output to facilitate the use of NEON intrinsics. A major difference to the SUPERCOP implementation is the way messages are loaded into the message schedule. The final structure of our code can be found in Appendix A.

4.2 Performance Analysis

A mere count of cycles per execution does not provide sufficient insight into the performance of the implemented code. Rather, a theoretical lower bound must be computed beforehand to be able to draw a meaningful conclusion. For this purpose, the number of instructions per iteration in the main body and message schedule creation of SHA-256 has to be calculated. Next, the latency and throughput of these functions are determined in order to find the total theoretical lower bound of digesting one message block with SHA-256 on the Cortex-A72.

Load and store operations are ignored as with ARMv8's previously mentioned dual issuing feature they come at reduced cost. The latency of the used instructions is also ignored, as in most cases it can be compensated by interleaving as described in Section 4.2.3.1. This makes the lower bound computed in this chapter an underestimation of the actual lower bound.

4.2.1 Message Schedule

The message schedule is created from the 512-bit message block which the update function receives. First the messages are copied into w_0 through w_{15}. Copying the slices of the messages into the message schedule as shown in Code 4.1 is highly inefficient as for each w_i four load/store operations need to be executed.

```
1  uint32x4_t w0 = {m0[0], m1[0], m2[0], m3[0]};
2  uint32x4_t w1 = {m0[1], m1[1], m2[1], m3[1]};
3  uint32x4_t w2 = {m0[2], m1[2], m2[2], m3[2]};
4  uint32x4_t w3 = {m0[3], m1[3], m2[3], m3[3]};
```

Code 4.1: Slow method of copying data into NEON registers.

While copying the message as seen in Code 4.2 is significantly faster, the resulting structures need to be transposed in order to further process them. With NEON's hardware implementation of transpositions this way of copying the message slices into w_0 through w_{15} is faster than Code 4.1 despite seeming more complex. The transposition is done with four instructions.

```
1  uint32x4_t w0 = {m0[0], m0[1], m0[2], m0[3]};
2  uint32x4_t w1 = {m1[0], m1[1], m1[2], m1[3]};
3  uint32x4_t w2 = {m2[0], m2[1], m2[2], m2[3]};
4  uint32x4_t w3 = {m3[0], m3[1], m3[2], m3[3]};
```

Code 4.2: Fast method of copying data into NEON registers.

The Cortex-A72 operates in big-endian mode while the message is stored little-endianly meaning that the messages cannot be copied into registers as they are. The 32-bit slices of the messages would first need to be converted to big-endian mode while loading them into the message schedule:

$$m_{i+3}|m_{i+2} << 8|m_{i+1} << 16|m_i << 24$$

NEON also offers an efficient way of reversing the bytes within the 32-bit words of its 128-bit registers via its `zip` and `rev` instructions. Each w_i needs to be reversed individually.

As the loading using transpositions and reversing is done for 128-bit message slices at a time, four iterations are necessary to fill w_0 through w_{15}. The copying process in total requires $4*4 = 16$ load operations, $4*4*4 = 64$ transpositions, $4*4*4 = 64$ zips and $4*4*4 = 64$ reversals (see Table 4.2).

The copying of the message blocks is followed by 48 iterations of the previously mentioned operations

$$s_{0_i} = (w_{i-15} \ggg 7) \oplus (w_{i-15} \ggg 18) \oplus (w_{i-15} \gg 3)$$

and

$$s_{1_i} = (w_{i-2} \ggg 17) \oplus (w_{i-2} \ggg 19) \oplus (w_{i-2} \gg 10)$$

to compute

$$w_i = w_{i-16} + s_{0i} + w_{i-7} + s_{1_i}$$

These variables are computed in parallel for all four messages with NEON instructions yielding a total of $48 * 2 = 96$ right shift operations, $48 * 2 * 2 = 192$ exclusive or (XOR) operations, $48 * 2 = 96$ rotate right (ROTR) operations and $48*3 = 144$ add operations (ADD). Due to the fact that there is no ROTR NEON instruction on the Cortex-A72, it has to be replaced by an additional left and right shift operation as well as an XOR operation:

$$a \ggg b = (a \ll (32 - b)) \oplus (a \gg b)$$

This increases the overall lower bound. The new total for the message schedule is hence $4 * 4 = 16$ load operations, $4 * 4 * 4 = 64$ transpositions, $4 * 4 * 4 = 64$ zips and $4 * 4 * 4 = 64$ reversals for copying the message blocks and $48 * 2 + 48 * 4 = 288$ right shift operations, $48 * 2 = 96$ left shift operations, $48 * 2 * 2 + 48 * 2 = 288$ XOR operations and $48 * 3 = 144$ ADD operations for computing w_{16} through w_{63} (see Table 4.1). The subtractions for the ROTR replacement can be ignored as in all use cases in SHA-256 all instances of b are fixed and the value of $32 - b$ is precomputed by the compiler.

4.2.2 SHA-256 Update

The update function is the core of the SHA-256 hash algorithm. As discussed in Chapter 2.3 the variables *choose, majority*, Σ_0, Σ_1, t_{1_i} and t_{2_i} are computed and used 64 times. To reiterate, the variables being computed are defined as follows:

$$choose = (e \cdot f) \oplus (\neg e \cdot g)$$

$$majority = (a \cdot b) \oplus (a \cdot c) \oplus (b \cdot c)$$

$$\Sigma_0 = (a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)$$

$$\Sigma_1 = (e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)$$

$$t_{1_i} = h + choose + w_i + k_i + \Sigma_{1_i}$$

$$t_{2_i} = \Sigma_{0_i} + majority_i$$

For *choose* $64 * 2 = 128$ logical and (AND) operations, 64 XOR operations and 64 logical not (NOT) operations are needed. However, much like the ROTR operation needed for the message schedule, there is no native NOT operation on NEON. Instead the inverse of e is computed as $e \oplus 0xffffffff$, adding another 64 XOR operations instead of the 64 NOT operations.

Further, for *majority* $64 * 3 = 192$ AND operations and $64 * 2 = 128$ XOR operations are needed. Additionally, Σ_0 and Σ_1 are identical with regards to the instructions used and we can handle them at one. For them in total $64 * 3 * 2 = 384$ ROTR operations and $64 * 2 * 2 = 256$ XOR operations are needed. Again, as was the case in the message schedule creation, the ROTR operation is in fact replaced by a right shift, a left shift, and an XOR operation. This nets the Σ_0 and Σ_1 computations overall $64 * 2 * 2 + 64 * 3 * 2 = 640$ XOR operations as well as $64 * 2 * 2 = 256$ left shifts and $64 * 2 * 2 = 256$ right shifts. Finally, for the helper t_{1_i} $64 * 4 = 256$ ADD operations are needed and another 64 ADD operations are required for the helper t_{2_i}.

	$\oplus$	$+$	$\cdot$	$\ll$	$\gg$	$(\ggg)$	$(\neg)$
w_i $(i > 15)$	288	144	-	288	96	(96)	-
choose	128	-	128	-	-	-	(64)
majority	128	-	192	-	-	-	-
Σ_0	320	-	-	192	192	(192)	-
Σ_1	320	-	-	192	192	(192)	-
t_{1_i}	-	256	-	-	-	-	-
t_{2_i}	-	64	-	-	-	-	-
Total	**1184**	**464**	**320**	**672**	**480**	**(480)**	**(64)**

Table 4.1: Operations per variable of message schedule and update. Grey indicates operations translated into other operations.

	Transpose	Zip	Reverse
w_i $(i < 15)$	64	64	64
Total	**64**	**64**	**64**

Table 4.2: Operations required to compute w_0 through w_{15}.

Table 4.1 lists the total number of operations per variable computation and the grand total of these operations. The computations for w_0 through w_{15} are listed separately in Table 4.2 as they cover operations. The overall lower bound, however, is not found by simply adding up all instructions. For

this it is important to take the used instructions characteristics, the pipeline they utilise, and their order of execution into account.

4.2.3 Lower Bound

Other factors need to be taken into account in order to correctly compute our implementations lower bound. Characteristics of the processor it is run on are as important as the inner structure of the algorithm. This section will introduce latency, throughput and utilised pipelines of the individual instructions and how these aspects impact the overall lower bound.

4.2.3.1 Latency, Throughput and Pipeline

The Cortex-A72 Software Optimization Guide provides data on every single instruction available on the Cortex-A72 [18]. This data includes the latency and throughput of these instructions as well as which pipeline is used to execute them.

An instructions latency describes the number of cycles a dependent operation must wait for the result of the operation it depends on. This is measured from the point the previous instruction arrives in the execution pipeline. A latency of three means a dependent instruction can neither be executed parallel to nor directly after the instruction it depends on, but must idle for two additional cycles. The key word in this is 'dependent' as for instructions independent of each other, latency is irrelevant.

The throughput of an instruction describes how many instructions of the same family can be executed within the same cycle over the entire processor. Where only one branch instruction can be executed every cycle, two shift operations of the arithmetic logic unit can be executed in the same cycle.

Operation op	Latency l	Throughput t	Pipelines
$\oplus$	3	2	F0/F1
$+$	3	2	F0/F1
$\cdot$	3	2	F0/F1
$\ll$	3	1	F1
$\gg$	3	1	F1
Transpose	3	2	F0/F1
Zip	3	2	F0/F1
Reverse	3	2	F0/F1

Table 4.3: Latency, throughput and used pipeline per used instruction.

As discussed in Chapter 2.4.1 and displayed in Figure 2.4, not all instructions are handled in the same pipeline. Instead every instruction has

specific pipelines they will be executed in. This also is not limited to just one pipeline per instruction. Important for SHA-256 as implemented for this thesis, the ASIMD instructions for left shift, right shift, XOR, ADD, and AND. The characteristics of the instructions used in our implementation vary widely as can be seen in Table 4.3.

4.2.3.2 Computing the Lower Bound

The latency and especially the throughput of the individual instructions are important when computing the lower bound as we may not simply add up all instruction instances as stated earlier. While not all computations in our implementations are independent of each other, cleverly moving around instructions can still eliminate the need to take latency into account. Looking at the way *choose* is computed, the XOR operation has to wait for the results of the two ADD operations. As the NEON ADD instruction has a latency of three the XOR operation cannot be executed for another three cycles. This restriction does not apply to other operations that do not depend on the result of the ADD operation. The computation of the *choose* and *majority* variables can be interleaved in such a manner that no instruction causes the CPU to idle. There are only a few edge cases in our implementation where this cannot be done. Hence, the latencies are not taken into account for the computation of the lower bound.

Operation op	t_{op}	x_{op}	l_{op}
$\oplus$	2	1184	592
$+$	2	464	232
$\cdot$	2	320	160
$\ll$	1	672	672
$\gg$	1	480	480
Transpose	2	64	32
Zip	2	64	32
Reverse	2	64	32
L			2232

Table 4.4: Throughputs, individual and total lower bound.

An instruction's throughput describes how many instances of this instruction can be computed simultaneously. Where a high latency would have been a disadvantage, a high throughput impacts the lower bound positively. For any given operation op, we can compute the total number of cycles used on that operation, l_{op}, with

$$l_{op} = \frac{x_{op}}{t_{op}}$$

where x_{op} is the number of times the operation is executed in our implementation and t_{op} is the operations throughput.

The total lower bound, L, may be defined as

$$L = \sum_{op \in OP} l_{op}$$

where OP is the set of all used operations. For a message that can be digested with a single block, we find that $L = 2232$ cycles as gathered from Table 4.4.

Chapter 5

Results

This chapter will provide a twofold performance analysis. After describing the benchmark environment, our parallelised implementation of SHA-256, `shax4-neon`, will be compared to a reference implementation of SHA-256, referred to as `sha`. The reference implementation of SHA-256 was written by Bernstein and is available at bench.cr.yp.to [27]. This is the same implementation of SHA-256 that is used in regular SPHINCS$^+$. Then, the performance differences of three implementations of SPHINCS$^+$ are compared to one another. The first implementation is a base implementation of SPHINCS$^+$, referred to as `sphincs`, as it is available in the SPHINCS$^+$ repository [28]. The second implementation, that serves as an additional reference, is a modified version of SPHINCS$^+$, `sphincsx4`, that uses four calls to `sha`. The final implementation uses our parallelised version of SHA-256 made for this thesis and will be referred to as `sphincsx4-neon`. Both `sphincsx4` and `sphincsx4-neon` will be compared to `sphincs` as well as to one another.

5.1 Benchmark Environment

All benchmarks were run under Debian 10, kernel version `5.10.17-v8+`, on the Raspberry Pi 4 Model B Rev 1.4. This Raspberry Pi is built with the Broadcom BCM2711. The four-core CPU on this chip has a base clock of 1.5 gigahertz and 1.8 gigabytes of system memory.

The cycle counter mentioned in Chapter 2.4.2.1 was used for the benchmarks of both SHA-256 and SPHINCS$^+$. All implementations were compiled with `gcc v8.3.0` and the flags `-O3`, `-std=c99`, `-march=native` and `-flax-vector-conversions`. For further information see Appendix B.

5.2 Parallelised SHA-256

In the following the two implementations `sha` and `shax4-neon` will be compared. For a consistent comparison, the reference implementation digested

the same message four times while the parallelised implementation digested four copies of the same message. This was repeated 1000 times each to minimise measuring errors due to the processor preempting in order to execute other programs. To further reduce measuring errors, the implementations were loaded in to the L1 cache by executing them once before entering the loop. The messages were chosen such that they are digested within a specific number of blocks. The first message was digested in a single block, the second in two, the third in three, the fourth in ten and the fifth and final message in 32 blocks.

Figure 5.1 shows the improvements our parallelised version of SHA-256, `shax4-neon`, offered over the non-parallel reference implementation, `sha`, as a function of the number of blocks to digest. For single-block messages, the performance increase was the highest with 27.2%. For messages of digested in two and three blocks, the advantages declined significantly to 20.0% and 18.1% respectively. For messages filling ten blocks the advantage further decreased to 15.3%. Finally, for 32-block messages the digestion was improved by 13.9%. Overall, with longer messages to digest the performance increase offered by `shax4-neon` declined. The blue line serves as a guide to the eye as non-integer values on the x-axis have no real world meaning. It emphasises that the performance offered by `shax4-neon` over `sha` declines logarithmically with respect to the input length (see Figure 5.1).

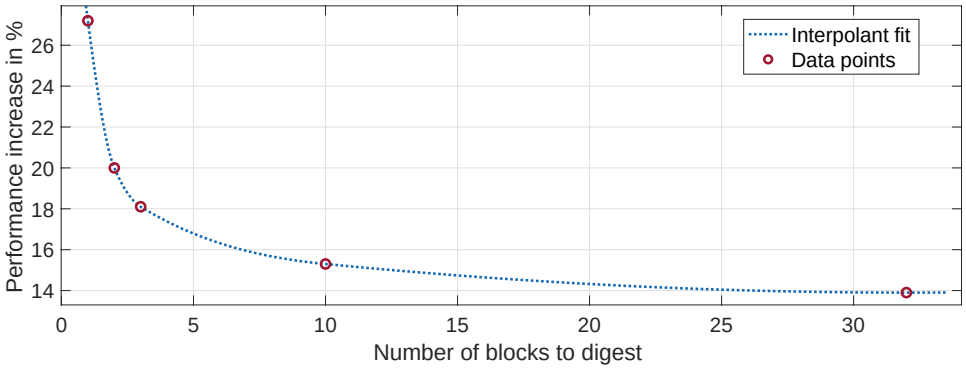


Figure 5.1: Performance increase of `shax4-neon` over `sha` per number of blocks the input is split into. The fit (dotted blue line) is only a guide to the eye as non-integer values on the x-axis are not real-world applicable.

Though the results show improvements over the reference implementation, the experimentally determined cycles count differs from the computed lower bound. In practice, our implementation took 1649 cycles longer for a single block message than expected. The average cycles count is 74% higher than the lower bound. As key generation, signing and verifying within SPHINCS$^+$ require the computation of a substantial amount of SHA-256 digests, this performance increase should carry over into SPHINCS$^+$.

5.3 Parallelised SHA-256 in SPHINCS$^+$

The iteration of the SPHINCS$^+$ repository at the time of writing this thesis contains tools for testing and benchmarking [28]. The tests have been used to ensure our implementation produced correct results. The benchmark data computed by the benchmark script will be used to determine the overall performance benefit of using `shax4-neon` within SPHINCS$^+$. The benchmarking script provides an even playing field for different implementations of SPHINCS$^+$. An instance of SPHICNS$^+$ using our implementation of SHA-256 was prepared and added to the benchmarking script.

With SPHINCS$^+$ instances being optimised for size, `s`, or speed, `f`, using the robust or simple hash function and meeting a target bit security of 128, 192 or 256 bits a total of twelve instances were created. The according SPHINCS$^+$ parameter sets are listed in Appendix C, Table C.1. Benchmark data was generated for each of these twelve instances per implementation. In the following a comparison of the performance benefits of `sphincsx4` and `sphincsx4-neon` over `sphnics` is conducted on the basis of their individual benchmark data. The data discussed below was derived from the raw data that can be viewed in Appendix C, Table C.3.

The performances of `sphincsx4` and `sphincsx4-neon` relative to `sphincs` for the twelve instances can be seen in Figure 5.2. The results are split into key generation (Figure 5.2a), signing (Figure 5.2b), and verification (Figure 5.2c). A positive performance increase means an implementation is faster than `sphincs` which is favourable.

5.3.1 Key Generation

In Figure 5.2a the performance of key generation of both `sphincsx4` and `sphincsx4-neon` relative to `sphincs` is displayed. For most simple instances `sphincsx4` is slower than `sphincs` offering an increase of 0.1% in the case of `128f` and neither an increase nor a penalty for `192f` with 0.0%. All robust instances of `sphincsx4` are faster than the reference `sphincs`.

In most cases the instances of `sphincsx4-neon` were slightly faster than `sphincs`. Exceptions to this are the robust instance of `128f` with a 2.6% decrease and the simple instances of `192f` and `256f` with 4.3% and 4.8% decreases in performance, respectively. Lastly, the robust instances with a target security of 256 bits were substantially slower than the reference with a performance penalty greater than 45%. The `sphincsx4-neon` implementation offered performance increases between 0.4% and 1.2% in all other instances.

Generally, the instances of `sphincsx4-neon` were slightly faster than `sphincsx4`. The robust instances of `128f` and `192f` were exceptions to this rule. Here `sphincsx4` offered a slight improvement over `sphincs` while `sphincsx4-neon` caused a performance penalty. A great performance penalty

in general and compared to sphincsx4 was observed in robust instances of 256s and 256f.

5.3.2 Signing

Similar behaviour can be observed for computing signatures in Figure 5.2b. All simple instances of sphincsx4 were slower than or as fast as the sphincs instances except for 256f which was 0.2% faster than sphincs. Signing in all robust instances of sphincsx4 was faster than sphincs by 0.7% to 1.4%.

The simple sphincsx4-neon instances of 128s, 128f, 256s are minimally slower than sphincs while the other simple instances of sphincsx4-neon are slightly faster with improvemnts between 0.3% and 1.5%. Robust instances of sphincsx4-neon are generally faster than the reference by 0.6% to 2.5%. An exception is 128f being 0.1% slower and the instances with 256-bit target security which are more than 40% slower.

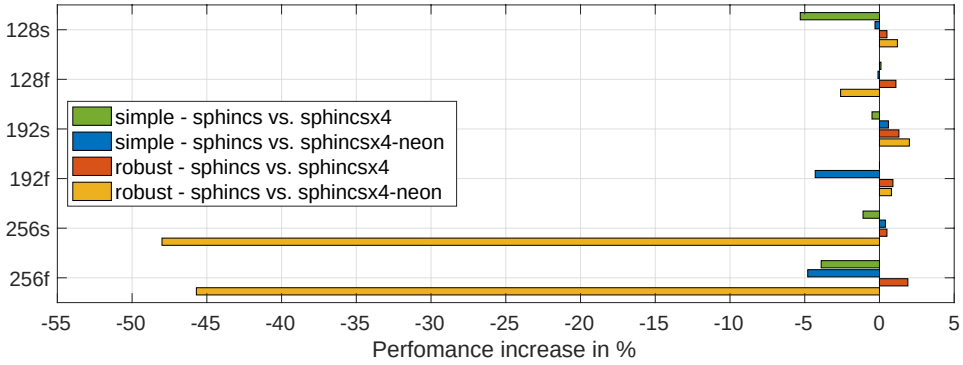
Simple instances of sphincsx4-neon are nearly always faster than the equivalent sphincsx4 instances. Exceptions are 128f and 256f where the sphincsx4-neon implementation is slightly slower than both sphincsx4 and sphincs. In most robust instance excluding 256s and 256f sphincsx4-neon is slightly faster or slightly slower than sphincsx4. The robust instances of sphincsx4-neon with a target security of 256 bits are notably slower than sphincsx4.

5.3.3 Verification

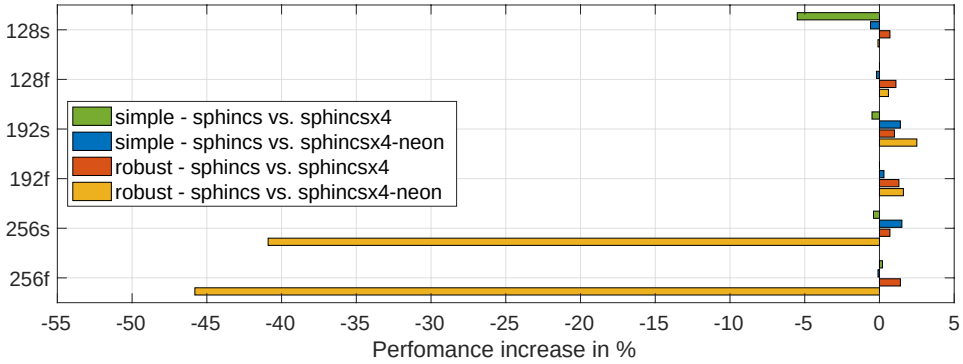
During verification sphincsx4 was observed to be faster than sphincs only in size optimized instances with a target security of 192 bits. Here, robust sphincsx4 was 0.1% faster than sphincs while the simple instance was 1.4% faster. In all other cases sphincsx4 was between 3.3% and 8.5% slower than sphincs except for simple 128s which was 32.4% slower than the sphincs equivalent (see Figure 5.2c).

The instances of sphincsx4-neon behaved very similarly. The only performance benefits were observed for instances of 192f where the simple instance was 4.3% faster than sphincs and the robust variant was 3.0% faster. Other instances of sphincsx4-neon were between 2.9% and 11.4% slower than sphincs. Outliers are, again, instances of 256s and 256s. Here, sphincsx4-neon was 43.2% to 50.8% slower than sphincs.

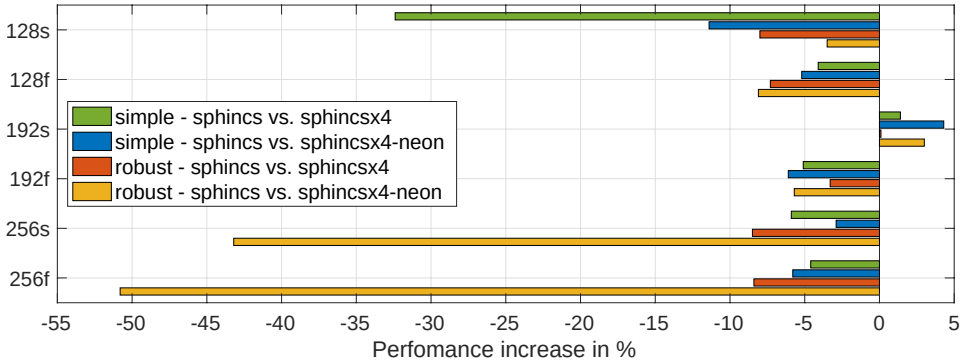
Generally, size optimized instances of sphincsx4 were slower than the sphincsx4-neon equivalent while in speed optimized instances sphincsx4 was mostly faster. Again, sphincsx4-neon is outstandlingly slower than both sphincs and sphincsx4 in robust instances with a 256-bit target security.



(a) Performance comparison for key generation.



(b) Performance comparison for signing.



(c) Performance comparison for verifying.

Figure 5.2: Relative performances of pseudo-parallelised and NEON parallelised instances of SPHINCS$^+$ compared to base SPHINCS$^+$ instances.

Chapter 6

Discussion

The results described in the previous chapter are discussed in the following while taking the calculated lower bound and aspects of the inner structure of SPHICNS$^+$ into account.

The results show that the initial assumption that NEON can offer increased performance held true for SHA-256. Here, the NEON-parallelised implementation, `shax4-neon`, was up to 27.2% faster than the reference implementation `sha` (see Figure 5.1). This increased performance did not translate into a performance increase within SPHINCS$^+$. The implementation of SPHINCS$^+$ using `shax4-neon`, `sphincsx4-neon`, was up to 50.8% slower than the base implementation of SPHINCS$^+$ (`sphincs`).

Insignificant benefits in performance were observed for a small number of instances. Only in two of the twelve benchmarked cases a performance benefit greater than 2.5% could be observed (see Figure 5.2). Compared to the associated performance penalties in other instances this is a negligible performance increase.

6.1 Parallelisation of SHA-256

The parallelisation of SHA-256 overall was a success. Performance increases of 13.8% up to 27.2% were observed where the benefit declined with increasing message length. While successful, the experimental data is off from the computed lower bound.

To compute the digest of a single block the NEON implementation of SHA-256, `shax4-neon`, took 3881 cycles. This is 1649 cycles more than the computed lower bound. As load and store operations as well as some conversion from little-endian to big-endian were not taken into account, it is expected that the lower bound is indeed an underestimation of the actual cycles needed.

However, the theoretical and practical data differ by 74% which is unexpected. This difference may be the reason the performance benefit decreases with increasing message length. The data implies that an aspect of our im-

plementation has linear complexity when it should be constant. It was not possible to identify this aspect within the frame of the thesis.

It is to ecpect that an implementation using NEON's cryptographic functions yields much higher performance benefits [29].

6.2 Parallelised Hash Functions within SHPINCS$^+$

There is a general lack of performance increases associated with using our NEON parallelised implementation of SHA-256 within SPHICNS$^+$. It may be attributed to two aspects of SPHINCS$^+$ discussed in the following. Firstly, parallelised SPHINCS$^+$ comes at a fix cost and secondly, only selected building blocks of SPHINCS$^+$ that can benefit from parallelisation were indeed parallelised.

6.2.1 Overhead within SPHINCS$^+$

During key generation and signing, SPHINCS$^+$ using four separate calls to SHA-256, `sphincsx4`, was often slightly faster than the base implementation. Here, improvements from using parallelisation are most visible. However, verification in `sphincsx4` was noticeably slower than the reference, `sphincs`. This indicates that a general overhead is associated with instances of SPHINCS$^+$ that compute multiple hashes in a single function call. Any implementation of SPHINCS$^+$ that aims to achieve a better performance via parallelisation must be able to compensate this overhead.

6.2.2 Parallelisation within SPHINCS$^+$

Merkle trees as well as instances of WOTS and FORS are highly parallelisable. However, within SPHINCS$^+$ parallelised implementations of hash functions are only used within WOTS and FORS. Instances of Merkle trees are not optimized in this manner. As one instance of WOTS is used at each leave of a Merkle tree, parallelising WOTS has a significantly higher impact on the overall performance than computing nodes on the Merkle trees would. Also, while parallelising the Merkle trees might offer a minor performance increase it would also make the code less comprehensible.

6.3 Possible Improvements

Within the scope of the thesis no explanation for the vast performance penalties in robust instances of `256s` and `256f` could be found.

Calls to `malloc()` are relatively expensive. Within our implementation, all calls to `malloc()` were removed and replaced by regular pointer arithmetic yielding minor but significant performance benefits. The implementation of SHA-256 in base SPHINCS$^+$ uses `malloc()` with each call to the

wrapper function `sha256`. Removing and remplacing `malloc()` here may have beneficial impacts on the performance of base SPHINCS$^+$.

Chapter 7

Conclusion and Outlook

Two implementations of SHA-256 were compared in this thesis: a reference implementation and a four-way NEON-parallelised implementation based on the reference implementation. The parallelised implementation offered a great performance increase for short messages that declined logarithmically with growing message length. Using the NEON-parallelised implementation within SPHINCS$^+$ generally performance increases of minor significance. The performance penalties in the case of robust 256-bit target security instances make the NEON-parallelised implementation overall worse than the base implementation.

ASIMD is a relevant tool for highly parallelisable tasks. SPHINCS$^+$ should benefit from using it. It is likely that the limited benefit of NEON in SHA-256 and negative effects it had within SPHINCS$^+$ are specific to the processor used. Better results may be achieved on different hardware such as the latest generation of Apple MacBook Pros with their new M1 chips. These implement an architecture with four NEON pipelines as opposed to the two NEON pipelines on the Cortex-A72 [18, 30].

The `bench.cr.yp.to` benchmarks imply that a massive performance increase can be achieved by using the ARMv8 cryptographic extension [29]. This should also have a noticeable effect of the performance on SPHINCS$^+$. Further, older or more simple devices such as the Raspberry Pi 4 that are not upgraded regularly and may benefit from an optimised version of our implementation. Therefore, it is relevant to further improve our implementation including research into why robust 256-bit security instances perform so poorly.

Bibliography

[1] Benjamin Schumacher. Quantum coding. *Physical Review A*, 51:2738–2747, April 1995.

[2] Thales et al. European industry white paper on the european quantum communication infrastructure. Technical report, QTSpace, Fall 2019. Online; accessed 16th of August 2021. URL: http://www.qtspace.eu/sites/testqtspace.eu/files/other_files/IndustryWhitePaper_V3.pdf.

[3] Daniel J. Bernstein, Johannes Buchmann, and Erik Dhamén. *Post-Quantum Cryptography*. Springer, 2019.

[4] Lieven M. K. Vandersypen et al. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866):883–887, December 2001. URL: http://dx.doi.org/10.1038/414883a, doi:10.1038/414883a.

[5] Kevin Kimball. Announcing request for nominations for public-key post-quantum cryptographic algorithms. Online; accessed 12th of August 2021. URL: https://www.federalregister.gov/documents/2016/12/20/2016-30615/announcing-request-for-nominations-for-public-key-post-quantum-cryptographic-algorithms.

[6] Daniel J. Bernstein et al. SPHINCS$^+$ - submission to the NIST post-quantum project, November 2017. Online; accessed 18th of June 2021. URL: https://sphincs.org/data/sphincs+-specification.pdf.

[7] Peter Schwabe. Hash-based signatures. Online; accessed 6th of July 2021. URL: https://cryptojedi.org/peter/data/taipei-20180628a.pdf.

[8] Sam Jaques. Consequences of Grover's algorithm. Online; accessed 6th of July 2021. URL: https://quantumcomputing.stackexchange.com/questions/5803/consequences-of-grovers-algorithm.

[9] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. Online; accessed 31st of May 2021. URL: http://bench.cr.yp.to/results-sign.html.

[10] Qualcomm Atheros Inc. Snapdragon 652 processor product brief. "Online; accessed 10th of August 2021". URL: https://www.qualcomm.com/media/documents/files/snapdragon-652-processor-product-brief.pdf.

[11] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238, New York, NY, 1989. Springer New York.

[12] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, October 1979. Online; accessed 1st of August 2021. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[13] Daniel J. Bernstein et al. The SPHINCS$^+$ signature framework, September 2019. "Online; accessed 27th of May 2021". URL: https://sphincs.org/data/sphincs+-paper.pdf.

[14] Jean-Philippe Aumasson et al. SPHINCS$^+$ - submission to the NIST post-quantum project, v.3. "Online; accessed 6th of August 2021". URL: https://sphincs.org/data/sphincs+-round3-specification.pdf.

[15] National Institute of Standards and Technology. Hash functions. Online; accessed 15th of August 2021. URL: https://csrc.nist.gov/projects/hash-functions.

[16] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011. Online; accessed 3rd of July 2021. URL: https://rfc-editor.org/rfc/rfc6234.txt, doi:10.17487/RFC6234.

[17] JuliusPC and James Hughes. BCM2711. Online; accessed 10th of August 2021. URL: https://github.com/raspberrypi/documentation/blob/master/hardware/raspberrypi/bcm2711/README.md.

[18] ARM Limited. Cortex®-A72 software optimization guide. Online; accessed 16th of June 2021. URL: https://developer.arm.com/documentation/uan0016/a/.

[19] Raspberry Pi Community , timg236. RPi4 BCM2711 and ARMv8 Crypto Extensions? "Online; accessed 7th of August 2021". URL: https://www.raspberrypi.org/forums/viewtopic.php?t=243410.

[20] ARM Limited. Introducing Neon for Armv8-A. Online; accessed 29th of April 2021. URL: https://developer.arm.com/documentation/102474/0100/Fundamentals-of-Armv8-Neon-technology.

[21] Jerin (jerinjacobk). ARMv8 performance counters management module. "Online; accessed 7th of August 2021". URL: https://github.com/jerinjacobk/armv8_pmu_cycle_counter_el0.

[22] Wouter van de Linde. Parallel SHA-256 in NEON for use in hash-based signatures. Bachelor Thesis at Radboud University Nijmegen, July 2016.

[23] Hayato Fujii, Félix C. Rodrigues, and Julio López. Fast AES implementation using ARMv8 ASIMD without cryptography extension. In *Information Security and Cryptology – ICISC 2019*, pages 84–101, Cham, 2020. Springer International Publishing.

[24] Kris Gaj Duc Tri Nguyen. Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8. *3rd PQC Standardization Conference*, 2021.

[25] Hanno Becker et al. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. Cryptology ePrint Archive, Report 2021/986, 2021. Preprint at https://ia.cr/2021/986.

[26] Ana Karina D. S. de Oliveira and Julio López. An efficient software implementation of the hash-based signature scheme MSS and its variants. In *Progress in Cryptology – LATINCRYPT 2015*, pages 366–383, Cham, 2015. Springer International Publishing.

[27] VAMPIRE. eBACS: ECRYPT benchmarking of cryptographic systems - SUPERCOP. "Online; accessed 7th of August 2021". URL: https://bench.cr.yp.to/supercop.html.

[28] sphincs.org. SPHINCS$^+$. "Online; accessed 9th of August 2021". URL: https://github.com/sphincs/sphincsplus.

[29] Daniel J. Bernstein. Implementation comparison: crypto_hash/sha256. Online; accessed 14th of August 2021. URL: https://bench.cr.yp.to/impl-hash/sha256.html.

[30] Andrei Frumusanu. Apple announces the Apple Silicon M1: Ditching x86 - what to expect, based on A14, November 2020. Online; accessed 15th of August 2021. URL: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2.

[31] Bas Westerbaan Martin Lehmann. Sphincscortexa-72. Online; accessed 19th of August 2021. URL: https://github.com/tharrry/SphincsCortexA-72.

Appendix A

C-code Structure

Here the most important functions of our implementation are given in a C-like syntax. Padding and other repetitive parts are omitted to keep the code as brief and legible as possible while also keeping it unambiguous.

```
1
2  #define SHR(x, c) (vshrq_n_u32((x), (c)))
3  #define ROTR_32(x, c) (veorq_u32(vshlq_n_u32((x), 32 - (c)),
       vshrq_n_u32((x), (c))))
4
5  #define Ch(x, y, z) (veorq_u32(vandq_u32((x), (y)),vandq_u32(
       veorq_u32(vdupq_n_u32(0xffffffff), (x)),(z))))
6  #define Maj(x, y, z) (veorq_u32(veorq_u32(vandq_u32((x),(y)),
       vandq_u32((x),(z))),vandq_u32((y),(z))))
7
8  #define Sigma0_32(x) (veorq_u32(ROTR_32(x, 2), veorq_u32(ROTR_32
       (x,13), ROTR_32(x,22))))
9  #define Sigma1_32(x) (veorq_u32(ROTR_32(x, 6), veorq_u32(ROTR_32
       (x,11), ROTR_32(x,25))))
10 #define sigma0_32(x) (veorq_u32(ROTR_32(x, 7), veorq_u32(ROTR_32
       (x,18), SHR(x, 3))))
11 #define sigma1_32(x) (veorq_u32(ROTR_32(x, 17), veorq_u32(
       ROTR_32(x,19), SHR(x, 10))))
12
13 #define M_32(w0, w14, w9, w1) w0 = vaddq_u32(vaddq_u32((
       sigma1_32(w14)), (w9)), vaddq_u32((sigma0_32(w1)), (w0)));
14
15
16 #define EXPAND_32                \
17     M_32(w0, w14, w9, w1)    \
18     M_32(w1, w15, w10, w2)   \
19     M_32(w2, w0, w11, w3)    \
20     M_32(w3, w1, w12, w4)    \
21     M_32(w4, w2, w13, w5)    \
22     M_32(w5, w3, w14, w6)    \
23     M_32(w6, w4, w15, w7)    \
24     M_32(w7, w5, w0, w8)     \
25     M_32(w8, w6, w1, w9)     \
26     M_32(w9, w7, w2, w10)    \
27     M_32(w10, w8, w3, w11)   \
```

```
28      M_32(w11, w9, w4, w12)  \
29      M_32(w12, w10, w5, w13) \
30      M_32(w13, w11, w6, w14) \
31      M_32(w14, w12, w7, w15) \
32      M_32(w15, w13, w8, w0)
33
34  static size_t crypto_hashblocks_sha256(uint8_t *statebytes,
35                                         const uint8_t *in0,
36                                         const uint8_t *in1,
37                                         const uint8_t *in2,
38                                         const uint8_t *in3,
39                                         size_t inlen,
40                                         int padded) {
41
42      size_t offset = 0;
43      uint32x4_t state[8];
44      uint32x4_t a;
45      ...
46      uint32x4_t h;
47      uint32x4_t T1;
48      uint32x4_t T2;
49
50      (void)padded;
51
52      uint32x4_t w0;
53      ...
54      uint32x4_t w15;
55
56      uint32x4_t i,  j,  k,  l, t;
57      uint64x2_t i2, j2, k2, l2;
58
59      uint32_t *in32_0 = (uint32_t *)in0;
60      uint32_t *in32_1 = (uint32_t *)in1;
61      uint32_t *in32_2 = (uint32_t *)in2;
62      uint32_t *in32_3 = (uint32_t *)in3;
63
64
65
66      a = load_bigendian_state_32(statebytes + 0);
67      state[0] = a;
68      ...
69      h = load_bigendian_state_32(statebytes + 28);
70      state[7] = h;
71
72      while (inlen - offset*4 >= 64) {
73
74          i = vld1q_u32(in32_0 +  0 + offset);
75          j = vld1q_u32(in32_1 +  0 + offset);
76          k = vld1q_u32(in32_2 +  0 + offset);
77          l = vld1q_u32(in32_3 +  0 + offset);
78
79          t = vtrn1q_u32(i, j);
80          j = vtrn2q_u32(i, j);
81          i = t;
```

```
82
83          t = vtrn1q_u32(k, l);
84          l = vtrn2q_u32(k, l);
85          k = t;
86
87          i2 = vreinterpretq_u64_u32(i);
88          j2 = vreinterpretq_u64_u32(j);
89          k2 = vreinterpretq_u64_u32(k);
90          l2 = vreinterpretq_u64_u32(l);
91
92          w0 = vrev32q_u8(vreinterpretq_u32_u64(vtrn1q_u64(i2, k2)
    ));
93          w1 = vrev32q_u8(vreinterpretq_u32_u64(vtrn1q_u64(j2, l2)
    ));
94          w2 = vrev32q_u8(vreinterpretq_u32_u64(vtrn2q_u64(i2, k2)
    ));
95          w3 = vrev32q_u8(vreinterpretq_u32_u64(vtrn2q_u64(j2, l2)
    ));
96
97
98          ...
99
100
101         i = vld1q_u32(in32_0 + 12 + offset);
102         j = vld1q_u32(in32_1 + 12 + offset);
103         k = vld1q_u32(in32_2 + 12 + offset);
104         l = vld1q_u32(in32_3 + 12 + offset);
105
106         t = vtrn1q_u32(i, j);
107         j = vtrn2q_u32(i, j);
108         i = t;
109
110         t = vtrn1q_u32(k, l);
111         l = vtrn2q_u32(k, l);
112         k = t;
113
114
115         i2 = vreinterpretq_u64_u32(i);
116         j2 = vreinterpretq_u64_u32(j);
117         k2 = vreinterpretq_u64_u32(k);
118         l2 = vreinterpretq_u64_u32(l);
119
120         w12= vrev32q_u8(vreinterpretq_u32_u64(vtrn1q_u64(i2, k2)
    ));
121         w13= vrev32q_u8(vreinterpretq_u32_u64(vtrn1q_u64(j2, l2)
    ));
122         w14= vrev32q_u8(vreinterpretq_u32_u64(vtrn2q_u64(i2, k2)
    ));
123         w15= vrev32q_u8(vreinterpretq_u32_u64(vtrn2q_u64(j2, l2)
    ));
124
125         F_32(w0, vdupq_n_u32(k[0]))
126         ...
127         F_32(w15, vdupq_n_u32(k[15]))
```

```
128
129         EXPAND_32
130
131         F_32(w0,  vdupq_n_u32(k[16]))
132         ...
133         F_32(w15, vdupq_n_u32(k[31]))
134
135         EXPAND_32
136
137         F_32(w0,  vdupq_n_u32(k[32]))
138         ...
139         F_32(w15, vdupq_n_u32(k[47]))
140
141         EXPAND_32
142
143         F_32(w0,  vdupq_n_u32(48))
144         F_32(w15, vdupq_n_u32(63))
145
146         a += state[0]; ... h += state[7];
147
148         state[0] = a; ... state[7] = h;
149
150         offset += 16;
151     }
152
153     store_bigendian_32(statebytes + 0, state[0]);
154     ...
155     store_bigendian_32(statebytes + 28, state[7]);
156
157     return inlen - (offset*4);
158 }
159
160 void sha256x4_inc_finalize(
161                 uint8_t *out0, uint8_t *out1,
162                 uint8_t *out2, uint8_t *out3,
163                 uint8_t *state,
164                 const uint8_t *in0, const uint8_t *in1,
165                 const uint8_t *in2, const uint8_t *in3,
166                 size_t inlen) {
167     uint8_t padded0[128];
168     ...
169     uint8_t padded3[128];
170     uint64_t mlen = (uint64_t)state[4*32] + inlen;
171
172     // Digest full message blocks
173     crypto_hashblocks_sha2563(state, in0, in1, in2, in3, inlen,
    0);
174
175     // Copy remaining message bytes
176     ...
177
178     // Padding as described in Section 2.3
179     ...
180
```

```
181     // Compute digest of last block
182     crypto_hashblocks_sha2563(state, padded0, padded1, padded2,
        padded3, 64, 1);
183
184     return;
185 }
186
187 void sha256x4(
188                 uint8_t *out0, uint8_t *out1,
189                 uint8_t *out2, uint8_t *out3,
190                 const uint8_t *in0, const uint8_t *in1,
191                 const uint8_t *in2, const uint8_t *in3,
192                 size_t inlen) {
193     sha256ctx2 state;
194
195     sha256x4_inc_init3(&state);
196     sha256x4_inc_finalize3(out0, out1, out2, out3, state.ctx,
        in0, in1, in2, in3, inlen);
197 }
```

Code A.1: All important functions and macros of the NEON-parallelised SHA-256 implementation.

Appendix B

Running our Implementation

Our implementation is available at github.com [31]. Included in the repository is the benchmarking program used to obtain the numbers for Section 5.1. To be able to run the code, the NEON extension must be activated in `binutils` and the cycle counter made available in user mode. The latter can be done using jerinjacobks kernel extension accessible at www.github.com [21].

To use our implementation on its own, include the `sha256_neon.c` file and `sha256_neon.h` header in your program. The API exposes functions to compute both unseeded and seeded digests. For unseeded hashes use the function `sha256x4` taking four output and input buffers as well as the length of the messages. Please note that all four messages must be of equal length. To compute seeded messages use `sha256x4_inc_finalize` with a character buffer containing the concatenation of all four states and the length that has been digested so far.

Additionally, our implementation provides the mask generating function `mgf1x4` for generating bit masks of a desired length in accordance with RFC2437. Like `sha256x4` it takes four output and input buffers as well as the length of the messages. Additionally, the desired length of the produced bit masks is required.

Appendix C

Parameters Raw Data

The tables below contains the parameter sets of all benchmarked instances of SPHINCS$^+$ and the the raw benchmark data of the SHA-256 and SPHINCS$^+$ benchmarks.

Target bit security	**Variant**	w	n	d	h	b	k
128	f	16	16	22	66	6	33
128	s	16	16	7	63	12	14
192	f	16	24	22	66	8	33
192	s	16	24	7	63	14	17
256	f	16	32	17	68	9	35
256	s	16	32	8	64	14	22

Table C.1: The benchmarked instances of SPHINCS$^+$ and their associated parameters.

Message Length		**Cycles**		**Improvement**
Bytes	Blocks	`sha`	`shax4-neon`	
55	1	5332	3881	27.2%
119	2	8148	6521	20.0%
183	3	11356	9301	18.1%
631	10	34168	28945	15.3%
2039	32	104920	90284	13.9%

Table C.2: Average cycles and improvements of parallelised SHA-256, `shax4-neon`, over non-parallelised SHA-256, `sha` for different message lengths.

Variant		Performance Increase					
	Action	128s	128f	192s	192f	256s	256f
sphincs							
Robust	Key Gen	555,168,309	8,723,435	829,624,282	12,934,689	779,317,929	49,779,755
	Signing	4,162,068,986	202,340,830	7,683,365,976	342,208,862	9,576,636,616	994,728,438
	Verifying	4,379,817	12,622,846	7,230,574	19,345,865	14,492,381	28,032,057
Simple	Key Gen	2,733,95,754	4,309,140	401,371,880	6,313,926	265,597,704	16,664,437
	Signing	2,069,569,548	100,211,109	3,827,112,118	169,326,395	3,448,154,481	34,9910,936
	Verifying	1,907,059	6,161,825	3,429,466	9,097,654	4,748,683	9,292,108
sphincsx4							
Robust	Key Gen	552,615,109	8,625,548	819,070,585	12,824,482	775,289,520	48,855,288
	Signing	4,134,562,449	200,052,774	7,607,347,598	356,104,439	9,507,486,367	980,529,480
	Verifying	4,732,108	13,549,864	7,224,194	19,993,036	15,720,062	30,378,738
Simple	Key Gen	287,930,512	4,305,635	403,479,027	6,316,367	268,540,067	17,309,190
	Signing	2,182,398,820	100,183,535	3,846,194,158	169,260,766	3,463,472,297	349,247,531
	Verifying	2,524,513	6,415,628	3,380,751	9,564,234	5,031,044	9,721,896
sphincsx4-neon							
Robust	Key Gen	548,530,192	8,948,003	813,096,085	12,837,677	1,153,044,523	72,549,998
	Signing	4,167,904,977	201,107,970	7,492,065,253	336,847,748	13,495,468,639	1,450,404,142
	Verifying	4,534,390	13,644,883	7,013,260	20,448,631	20,760,047	42,260,270
Simple	Key Gen	274,332,968	4,312,666	398,808,380	6,584,104	264,472,855	17,470,188
	Signing	2,082,951,527	100,381,406	3,775,261,836	168,804,871	3,395,989,914	350,415,883
	Verifying	2,124,053	6,480,163	3,281,065	9,653,046	4,888,327	9,827,715

Table C.3: Raw benchmark data of `sphincs`, `sphincsx4` and `sphincsx4-neon` in absolute cycles.

Acknowledgement

I would like to thank my supervisor, Prof Dr Peter Schwabe, for giving me the opportunity to pursue my bachelor thesis on a topic that is highly relevant to current research. Further, I would like to thank Prof Dr Joan Daemen for being my second assessor. Bas Westerbaan, I would like to thank for being my second supervisor and for always being there to answer my questions and assisting me in finding some less obvious bugs in my code. His ability to differentiate between situations where it was appropriate to fully explain concepts and when a pointer in the right direction sufficed was helpful, to say the least. Finally, I would like to thank my family and my friend Hannah Willenberg as well as her family for proofreading and supporting me throughout with kind words and cake.