Putting Wings on SPHINCS

Stefan Kölbl
stek@dtu.dk

DTU Compute, Technical University of Denmark, Denmark

Abstract. SPHINCS is a recently proposed stateless hash-based signature scheme and promising candidate for a post-quantum secure digital signature scheme. In this work we provide a comparison of the performance when instantiating SPHINCS with different cryptographic hash functions on both recent Intel and AMD platforms found in personal computers and the ARMv8-A platform which is prevalent in mobile phones.
In particular, we provide a broad comparison of the performance of cryptographic hash functions utilizing the cryptographic extensions and vector instruction set extensions available on modern microprocessors. This comes with several new implementations optimized towards the specific use case of hash-based signature schemes.
Further, we instantiate SPHINCS with these primitives and provide benchmarks for the costs of generating keys, signing messages and verifying signatures with SPHINCS on Intel Haswell, Intel Skylake, AMD Ryzen, ARM Cortex A57 and Cortex A72.

Keywords: hash-based signature schemes, implementation, post-quantum cryptography, SPHINCS, ARM

1 Introduction

Digital signature schemes are one of the fundamental cryptographic algorithms and are typically used to provide authenticity, integrity and non-repudiation for a message. They have found several applications in information security, e.g. certification of public keys, code signing or as an electronic signature. One of the major threats to the currently widely used digital signature schemes like DSA/ECDSA is that they are not secure if an attacker can build a large enough quantum computer. The security of these schemes relies on difficult number theoretic problems, which can be solved in polynomial time on a quantum computer [33].

There are various solutions for *post-quantum* secure digital signature schemes, namely lattice-based, multivariate-quadratic, code-based and hash-based signatures. One of the main advantage of hash-based signature schemes is that the security reduces to properties of the underlying cryptographic hash function. As every digital signature scheme requires a one-way function [32] these can be seen as the minimal assumptions necessary to construct a secure signature scheme. All the other previously mentioned signature schemes require further assumptions by relying on the difficulty of *hard* problems for which the asymptotic difficulty might not always hold for the concrete instances used in a cryptographic systems and they require carefully choosing the parameters.

Hash-based digital signature schemes are therefore a very attractive choice. However most schemes, like XMSS [9] and LMS [14], are *stateful*, this means that one has to update the secret key with every signature. This may sound quite innocent, however it can be a severe difficulty in practice. For instance when sharing a private key on different computers one has to synchronize all of them or security can be void. For some applications this might be acceptable, however in general we desire to have a stateless signature scheme.

Goldreich proposed the first stateless hash-based signature scheme [18], however the parameters required for this construction to provide a sufficient security level and reasonable number of signatures per key pair resulting in a fairly large signature above 1 MB. SPHINCS [5] improves upon this construction in several aspects and first demonstrates that stateless schemes can be practical and provide a reasonable signature size (41 KB) while computing hundreds of signatures per second on a modern CPU.

The performance of SPHINCS directly relates to the underlying cryptographic hash function and therefore the performance of this function is critical, which will be the main focus of this work. The requirements for this function also differ from the classic use cases for cryptographic hash functions, as we do not require *collision resistance* and the inputs for most calls are rather short, typically 256 or 512 bits.

Contributions. The main goal of this work is to provide a comparison of performance when instantiating SPHINCS with different hash functions on modern high-end processors found in personal computers and mobile phones. In order to achieve this we provide several implementations, for modern Intel, AMD and ARM CPUs, optimized towards the requirements of SPHINCS. This includes implementations of SHA256, Keccak, Simpira, Haraka and ChaCha optimized for hashing short inputs in parallel utilizing vector instructions and cryptographic extensions available on these microprocessors.

We further instantiate SPHINCS with these implementations and provide a broad comparison of the costs of generating key pairs, signing messages and verifying signature on Intel Haswell, Intel Skylake, AMD Ryzen, ARM Cortex A57 and A72. These are also the first optimized implementations for the ARMv8-A platform for SPHINCS and improve the understanding of the costs of stateless hash-based signature schemes. This performance results also indicate that SPHINCS is practical on the architecture used in a growing number of mobile phones.

Software. The implementations are put in the public domain and are available under https://github.com/kste/sphincs.

Related Work. So far there is only a limited amount of benchmarks for SPHINCS available. The original paper proposing SPHINCS [5] provides a reference implementation and an optimized implementation which utilizes the AVX2 vector extensions for speeding up the underlying ChaCha permutation. In [29] the authors propose a dedicated short-input hash function Haraka, which utilizes the AES instruction set to speed-up hash-based signature schemes and also provide some benchmarks for SPHINCS on the recent Intel platforms. The AES-based permutation design Simpira has recently also been proposed to instantiate SPHINCS [21] and its performance on Intel Skylake was evaluated. The first implementation on low-end platforms was provided in [25]. Here the authors demonstrate that SPHINCS can also be implemented on a 32-bit microcontroller based on the ARM Cortex M3 with very limited RAM available.

2 The SPHINCS Signature Scheme

In this section we give an overview of the SPHINCS digital signature scheme. Throughout the paper we will use the same parameters as suggested in [5], which will give a signature size of 41KB, public-key size of 1056 bytes and a private-key size of 1088 bytes. These parameters target a security level of 128 bits against an adversary who has access to a large enough computer and allow up 2^{50} signatures for a key pair. For more details we refer the reader to [5].

First, we will give a brief description of the main components used in SPHINCS and provide some insights on how much impact the performance of the underlying primitives has on the performance of SPHINCS. In particular, we are interested in two functions

$$\begin{aligned} \mathbf{F} &: \{0,1\}^{256} \rightarrow \{0,1\}^{256} \\ \mathbf{H} &: \{0,1\}^{512} \rightarrow \{0,1\}^{256}. \end{aligned} \tag{1}$$

which, as we will see later, are responsible for most of the computations in SPHINCS.

2.1 Hash Trees

At various points in the construction, SPHINCS uses a hash tree (also known as Merkle tree). A hash tree is a full binary tree of height h. We denote the ith node at level j of this tree as $N_{i,j}$, hence the root corresponds to $N_{0,h}$. Each node, which is not a leaf, gets labeled with the hash of its child nodes $N_{i,j} = \mathbf{H}(N_{2i,j-1}||N_{2i+1,j-1})$. Note that in order to drop the requirement for a collision resistant hash function [13], the inputs to $\mathbf{H}$ are further masked in all hash trees used in SPHINCS.

An important term related with hash trees is the *authentication path*. The authentication path Auth_i serves as a proof that the node $N_{i,j}$ is part of the hash tree with root $N_{0,h}$. It contains the minimal number of nodes which are required to recompute the root of a hash tree given $N_{i,j}$. This newly computed root can then be compared with the previously commited one to verifiy that $N_{i,j}$ is indeed part of the original tree.

2.2 One-time Signature: WOTS$^+$

As a one-time signature SPHINCS uses WOTS$^+$ [24], which has a parameter w allowing a trade-off between signature size and number of computations. Further, we derive the following parameters

$$l_1 = \left\lceil \frac{n}{\log w} \right\rceil,\ l_2 = \left\lfloor \frac{\log\left(l_1(w-1)\right)}{\log w} \right\rfloor + 1,\ l = l_1 + l_2. \tag{2}$$

In the case of SPHINCS $w = 16$, thus $l = 67$. Additionally, we use $\mathbf{F}$ to construct the chaining function

$$c^i(x) = \mathbf{F}(c^{i-1}(x) \oplus Q_i) \tag{3}$$

where Q_i is a round specific bitmask and $c^0(x) = x$.

Key Generation. The keys are derived from an initial secret key $\mathcal{S}$ which is expanded with a pseudo-random generator to obtain a secret key $\mathrm{sk} = (\mathrm{sk}_1, \ldots, \mathrm{sk}_{67})$ for WOTS$^+$. The public key pk is then computed by applying the chaining function on each part of the secret key

$$(\mathrm{pk}_1, \ldots, \mathrm{pk}_{67}) = (c^{w-1}(\mathrm{sk}_1), \ldots, c^{w-1}(\mathrm{sk}_{67})). \tag{4}$$

In order to reduce the size of this public key we build a hash tree on top of it to obtain pk. As l is usually not a power of two the L-tree [13] construction is used. This structure is similar to a binary tree, however if there is an odd number of nodes on a level the rightmost node is lifted up one level (see Figure 1). The root of the resulting tree is then used as the public key pk.

Signing. A message m is signed by first computing the base w representation of the message $M = (M_1, \ldots M_{l_1})$. The next step is to compute a checksum $\sum_{i=1}^{l_1}(w - 1 - M_i)$ and also its base w representation $C = (C_1, \ldots, C_{l_2})$. We concatenate these values and obtain $B = (B_1, \ldots, B_l) = M||C$. The signature for M is then given by

$$\sigma = (\sigma_1, \ldots, \sigma_l) = (c^{B_1}(\mathrm{sk}_1), \ldots, c^{B_l}(\mathrm{sk}_l)). \tag{5}$$

Verification. The process of verifying a signature σ of a message m with the public key pk is done in a similar way. First, we have to recompute B and then compute

$$(\mathrm{pk}'_1, \ldots, \mathrm{pk}'_l) = (c^{w-1-B_1}(\sigma_1), \ldots, c^{w-1-B_l}(\sigma_l)) \tag{6}$$

Note that the correct bitmasks have to be used in each step of the chaining function to get the correct results. The final step is to recompute the root of the L-tree and check if $\mathrm{pk}' = \mathrm{pk}$.

2.3 Few-time Signature: HORST

The second important component of SPHINCS is a few-time signature scheme. SPHINCS uses HORST, which is a variant of HORS [31] with an additional tree structure. HORST has two parameters t and k, which are $t = 2^{16}$ and $k = 32$ in the case of SPHINCS.

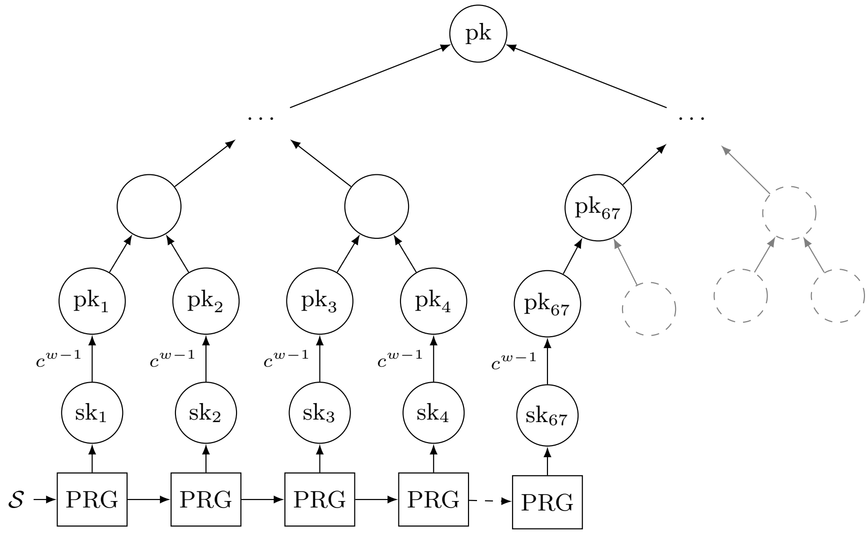

Fig. 1. WOTS$^+$ key generation using an L-tree for computing the public key.

Key Generation. In order to generate the secret key we expand a secret $\mathcal{S}$ to obtain $\mathrm{sk} = (\mathrm{sk}_1, \ldots, \mathrm{sk}_t)$, similar to the WOTS$^+$ key generation. The elements of this list are used to generate the leaves of a binary tree by computing $\mathbf{F}(\mathrm{sk}_i)$. We then compute a hash tree on top of these leaves and the public key is the root node.

Signing. For signing, the message m is split into k pieces of length $\log t$ giving us $M = (M_1, \ldots, M_k)$. Next, we interpret each M_i as an integer and compute the signature as $\sigma = (\sigma_1, \ldots, \sigma_k, \sigma_{k+1})$. Each block $\sigma_i = (\mathrm{sk}_{M_i}, \mathrm{Auth}_{M_i})$ for all $i \leq k$. This corresponds to the M_ith element in the secret key and Auth_{M_i} are the elements required for computing the authentication path up to level 10 (see Figure 2). Finally, σ_{k+1} contains all nodes at level 10 of the tree.

Verification. The verification process is very similar. First, the received parts of the secret key are hashed using $\mathbf{F}$. Together with the authentication paths this allows us to recompute the nodes at level 10 for each sk_i. These can then be verified with the values given in σ_{k+1}. Finally, the nodes in σ_{k+1} are used to recompute the root of the tree which has to be equal to pk.

2.4 Putting everything together

SPHINCS uses a nested tree structure consisting of 12 layers of trees of height 5 (see Figure 3). Each tree is a binary tree where the leaves are the public key of a WOTS$^+$ key pair. The top layer consists of a single tree and each key pair in the leaves is used to sign the root of another tree. Hence, on the second layer we will

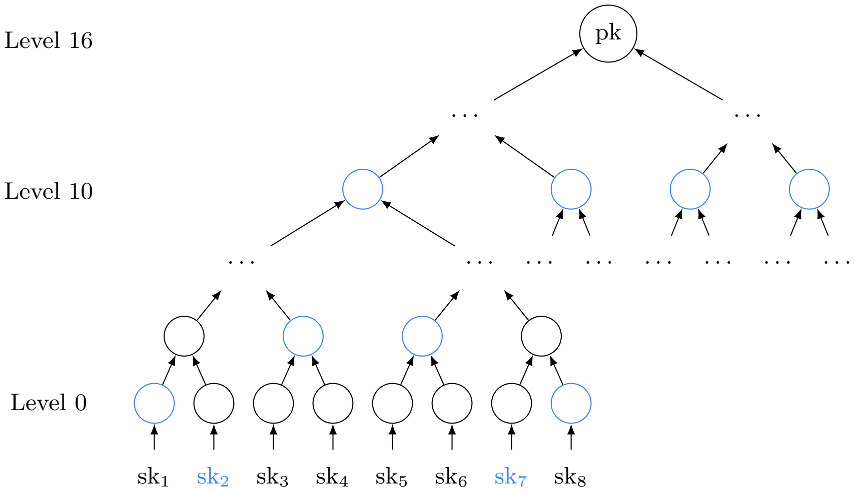

Fig. 2. Signing process in the HORST few-time signature scheme. In this case sk_2 and sk_7 are chosen by m and all the blue nodes are part of the authentication path and therefore part of the signature.

have 32 trees. This process is repeated until we reach the bottom layer. On the bottom layer we use the final WOTS$^+$ keys to sign a HORST public key, which is then used to sign the message.

Key Generation. For generating the keys in SPHINCS we choose two random 256-bit values $\mathcal{S}, \mathcal{S}'$. The first value is used during the key generation and the second one for signing. Furthermore, we need to generate all the bitmasks Q for WOTS$^+$, HORST and the binary hash trees. For the public key pk we only need to compute the root of the tree at the top and therefore have to generate the 32 WOTS$^+$ key pairs. The secret key is then $(\mathcal{S}, \mathcal{S}', Q)$ and the public key (pk, Q).

Signing. The first step is to select a HORST key to sign the message. We use a pseudorandom function (which involves $\mathcal{S}'$) to compute the index *idx* of the HORST key pair which we then use to sign a *randomized* digest R derived from m giving us the signature σ_{HORST}. Note that the HORST key pair is fully determined by this *idx* and the secret key $\mathcal{S}$.

The next step is to generate the WOTS$^+$ key pair which signs the HORST public key used when computing σ_{HORST}. This again depends entirely on $\mathcal{S}$ and the position in the tree and gives us the WOTS$^+$ signature $\sigma_{w,1}$. The public key for this WOTS$^+$ signature is part of another tree and needs to be authenticated again. We therefore compute the authentication path $\mathrm{Auth}_{w,1}$ for $\mathrm{pk}_{w,1}$.

This procedure of signing the root with a WOTS$^+$ key pair and computing the authentication path is repeated until we reach the top layer. The full signature

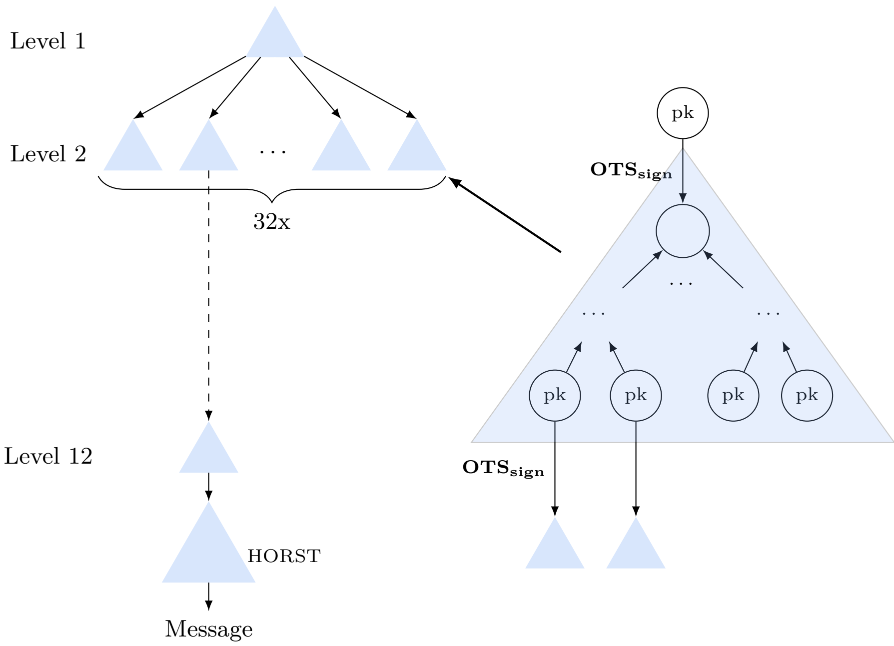

Fig. 3. Virtual tree structure used in SPHINCS.

then consists of

$$\sigma = (\text{idx}, R, \sigma_{HORST}, \sigma_{w,1}, \text{Auth}_{w,1}, \ldots, \sigma_{w,12}, \text{Auth}_{w,12}). \tag{7}$$

Verification. The verification process consists of recomputing the randomized digest for the message and first verifying σ_{HORST}. If this is successful we continue with the verification of $\sigma_{w,1}$ and all further signature $\sigma_{w,i}$ until we reach the root of our tree. If all verifications succeed and the root of the top tree equals pk the signature is accepted.

3 How to instantiate SPHINCS?

The performance of SPHINCS strongly correlates with the performance of two functions **F** and **H** which have the following security requirements

- **Preimage Resistance**: For a given output y it should be computationally infeasible to find an input x' such that $y = f(x')$.
- **Second-Preimage Resistance**: For a given x and $y = \mathcal{H}(x)$ it should be computationally infeasible to find $x' \neq x$ such that $f(x') = y$.
- **Undetectability**: It should be computationally infeasible for an adversary to predict the output.

For **F** we require *preimage resistance*, *second-preimage resistance* and *undetectability*, while **H** has to be *second-preimage resistant*. The best generic attacks against an ideal function with an output size of n bits require 2^n calls to the function respectively $2^{n/2}$ on a quantum computer using Grover's algorithm. In the case of SPHINCS an attacker with access to a quantum computer should not be able to succeed in violating any of these properties with less than 2^{128} calls to the underlying function.

Contrary to a generic cryptographic hash function these requirements are very different. For instance we do not require those functions to be collision resistant, which in general is a much stronger requirement. Various cryptographic hash functions in the past have been broken in this setting like MD4 [35], MD5 [37] or SHA-1 [36] and while one can construct collisions in practice for all these functions, finding a preimage is still very costly, even for MD4 [30,22]. The second difference is that these functions have a fixed input size. Most hash functions only reach their best performance for longer messages and several attacks are also only applicable for long messages.

Before, we discuss the different choices we first take a closer look at how many calls to these functions are required for *key generation*, *signing* and *verification* (also see Table 1). For generating the key in SPHINCS we need to do 32 WOTS$^+$ key generations (and the corresponding L-tree) and construct the hash tree. In total this amounts to $32 \cdot (67 \cdot 15) = 32160$ computations of **F** and $(32 \cdot 66) + 31 = 2143$ computations of **H**.

Table 1. Costs in term of **F** and **H** for the operations in SPHINCS.

Operation	Calls to **F**	Calls to **H**
Key Generation	32160	2143
Signing	451456	93406
Verification	$\leq$ 12092	1235

For signing we need to compute one HORST signature and 12 trees which include the costs for one WOTS$^+$ key generation each. Note that the WOTS$^+$ signature can already be extracted while generating the WOTS$^+$ key pairs. This means that one signature requires at least $65536 + (12 \cdot 32160) = 451456$ calls to **F** and $65535 + 12 \cdot 2144 + 2143 = 93406$ calls to **H**.

For verification we need one HORST verification and 12 WOTS$^+$ verifications (including the L-tree) which corresponds to at most $12 \cdot (67 \cdot 15) + 32 = 12092$ calls to **F** and $(12 \cdot (66 + 5)) + 383 = 1235$ calls to **H**.

3.1 ChaCha

ChaCha is a family of stream ciphers [4]. In the original SPHINCS design both **F** and **H** are constructed from the 512-bit permutation π_{ChaCha}. If π_{ChaCha}

represents 12 rounds of the ChaCha permutation then

$$\mathbf{F}(M_1) = \text{Trunc}(\pi_{\text{ChaCha}}(M_1||C))$$
$$\mathbf{H}(M_1||M_2) = \text{Trunc}(\pi_{\text{ChaCha}}(\pi_{\text{ChaCha}}(M_1||C) \oplus (M_2||0^{256}))$$

where M_1, M_2 are 256-bit messages and C is a 256-bit constant. Trunc is a function which truncates the output to 256 bits.

The best attack on the ChaCha stream cipher can recover a secret key for 7 rounds [2], however no concrete analysis exists in the construction used here. The building block used for the SHA-3 candidate Blake [3] shares a lot of similarities with the permutation used for ChaCha and it is likely that similar attack strategies can be applied. The best (second)-preimage attacks on Blake only cover 2.75 rounds and a (pseudo) preimage attack on 6.75 rounds of the compression function exists [16].

3.2 SHA256

SHA256 is one of the most widely used cryptographic hash functions. It was published in 2001 and designed by the NSA. The compression function processes blocks of 512-bit using the Davies-Meyer construction and can be directly used to build both **F** and **H**[1]. We denote these functions as SHA256-**F** and SHA256-**H**. The best preimage attacks on SHA256 reach 45 out of 64 rounds [28] and are only slightly faster than bruteforce. In [1], the costs of finding a preimage using Grover's quantum algorithm [19] for SHA-256 have been estimated at around 2^{166} basic operations.

3.3 Keccak

Keccak is a family of cryptographic hash functions based on the Sponge construction and has been standardized as SHA-3 (FIPS PUB 202). It offers a range of permutations of size $b = 25 \cdot 2^l$ for $l = 0, \ldots, 6$. For an output size of 256-bit the SHA-3 standard specifies to use Keccak$[b = 1600, c = 512]$. This would allow us to instantiate **F** and **H** with a single call to the permutation, as we can process up to 1088 bits. However, this seems quite an inefficient use and it might be beneficial to use a smaller permutation. Recently, two versions of Keccak with a reduced number of rounds have been proposed [8]. KangarooTwelve for 128-bit security and MarsupilamiFourteen for 256-bit security.

The *capacity* c in a sponge directly relates to the security level and in the classical setting a Sponge requires $c = 512$, to have 256-bit second-preimage resistance. However, it is not clear whether we need a capacity of 512 bits if we only require 2^{128} security against a quantum adversary.

[1] To separate the domains of the two functions one could use a different IV or round constants.

In order to evaluate the potential of using Keccak in SPHINCS we choose both a smaller permutation and reduce the number of rounds

$$\begin{aligned} \text{Keccak-}\mathbf{F}(M) &= \text{Trunc}(\text{Keccak}[b = 800, \text{rounds} = 12, c = 256](M)) \\ \text{Keccak-}\mathbf{H}(M) &= \text{Trunc}(\text{Keccak}[b = 800, \text{rounds} = 12, c = 256](M)). \end{aligned} \tag{8}$$

The best preimage attacks on Keccak with an output size of 256-bit can cover 4 rounds of Keccak [23], apart from a slight improvement over brute force with huge memory for 8 rounds [10]. The costs of applying Grover's quantum algorithm to find a preimage for SHA3-256 have also been estimated at around 2^{166} in [1]. Overall, taking into account the restricted setting a reduced-round version of Keccak seems reasonable for this use case.

3.4 Haraka

Haraka is a short-input hash function, specifically designed for the use in hash-based signature schemes [29]. The construction uses an efficient 256-bit (resp. 512-bit) permutation based on the AES with a simple mode (see Figure 4) to build the two functions **F** and **H**.

The best preimage attacks by the authors can find a preimage for 3.5 respectively 4 out of 5 rounds. For an earlier version of Haraka-**H** there also exists an attack exploiting weak round constants which can find a preimage in 2^{192} evaluations [26], however this attack is not applicable to the current version.

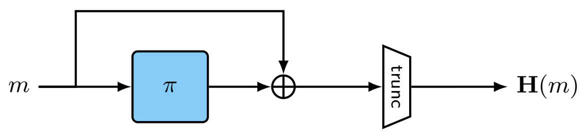

Fig. 4. Using a permutation π to construct a short-input hash function.

3.5 Simpira

Simpira is a family of cryptographic permutations [20] that supports an input size of $b \cdot 128$. The design is based on generalized Feistel networks and uses the AES round function for updating the branches. The variants with $b = 2$ and $b = 4$ can be used in the same mode as Haraka to construct

$$\begin{aligned} \text{Simpira-}\mathbf{F}(M) &= \text{Simpira}[b = 2](M) \oplus M \\ \text{Simpira-}\mathbf{H}(M) &= \text{Trunc}(\text{Simpira}[b = 4](M) \oplus M). \end{aligned} \tag{9}$$

The security claim for Simpira is that no distinguisher with costs $< 2^{128}$ exists, but so far no concrete preimage attacks have been published.

4 Efficient Implementations for F and H

The target platforms for our implementations are on one hand the recent x86 CPUs by Intel (Haswell and Skylake), AMD (Ryzen) and on the other hand the ARMv8-A architecture, which has a large share in the mobile phone market. In order to understand how to efficiently implement our primitives on these platforms we give a quick overview of the most important features we utilize.

4.1 Instruction Pipeline

Modern CPUs have an instruction pipeline, which allows some form of parallelism on a single CPU core. This is realized by splitting up an instruction into different stages which can be executed in the same cycle. In order to assess the performance of an instructions we use two notions, the *latency* and the *inverse throughput*. Latency corresponds to the number of clock cycles we have to wait until we get the result of an instruction, while the inverse throughput is the number of clock cycles we have to wait until we can issue the same instruction again.

Utilizing the pipeline is an important performance consideration and can especially be useful for instructions with a high latency and low inverse throughput. This has previously been studied in various AES-based designs [27,20,29] to increase the performance of cryptographic operations. In the case of SPHINCS it is particularly easy to keep the pipeline filled up, as one has multiple independent inputs available for most operations. For instance, the WOTS^+ chains can be computed in parallel and most levels of a hash tree allow a high degree of parallelism.

4.2 Vector Instructions

Another important feature of modern microprocessors are vector units which provide parallelism through *single instruction, multiple data* (SIMD) instructions. These instructions allow to apply the same operation to multiple values stored in a *vector register* and can significantly increase the throughput. For many cryptographic primitives the fastest implementations utilize SIMD instructions. While we often have to pack the data in a specific format, these costs are compensated by processing multiple messages/blocks in parallel. Especially in the case of hash-based signature where multiple independent inputs are almost constantly available it allows us to fully utilize this feature for a very efficient implementation.

On the current Intel and AMD platforms[2] the vector extensions is called AVX2, which features 16 registers of 256-bit. This will be further extended to AVX-512[3], allowing to operate on 512-bit vectors which will likely speed-up all vector implementations through the higher degree of parallelism.

[2] AVX2 is available since Intel Haswell, for older platforms the predecessor AVX can be used which supports 128-bit vectors.

[3] AVX-512 can already be found in Xeon Phi (Knights Landing) and Skylake-X processors.

The ARMv8-A architecture offers the NEON instruction set, which allows to operate on 128-bit vectors. Future ARM platforms [34] will come with a scalable vector extension (SVE), supporting vectors up to a size of 2048 bits and hence allowing 16 times the parallelism compared to the current ARM processors.

4.3 Crypto Extensions

An increasing number of platforms provide instructions carrying out cryptographic operations, which provide a significant speed-up for the supported primitives while also providing a constant running time and protection against cache-timing attacks. All recent Intel platforms provide instructions for the round function of the AES and a similar extensions is available on ARMv8-A. Additionally, the ARM crypto extensions support SHA-1 and SHA256. On the newest AMD platform Ryzen these instructions are also available and support for them is also planned for the next generation of Intel processors. An overview of these instructions and their performance characteristics is given in Table 3.

4.4 ChaCha-F and -H

The ChaCha permutation is very fast in software and benefits strongly from the SIMD features on modern CPUs, which is also one of the main motivations why the SPHINCS designers use it for instantiating SPHINCS. As the design is based on 32-bit words, AVX2 can be utilized to process up to 8 blocks in parallel. Similar, using ARM NEON we can process 4 blocks in parallel. On Intel platforms we use the original AVX2 implementation of ChaCha provided with SPHINCS in [6]. For ARM we use the implementation by Romain Dolbeau available in Supercop [6], as it is the fastest available using on the ARM Cortex A57, to construct ChaCha-**F** and ChaCha-**H**.

4.5 SHA256-F and -H

SHA256 is also based around operations on 32-bit words and therefore benefits in the same way as ChaCha from the use of SIMD instructions. For Intel Haswell and Skylake we implemented SHA256 using AVX2 processing 8 blocks in parallel.

We use eight registers, where each one contains one 32-bit word of the state S_i for all eight blocks (see Figure 5). We assume that the incoming message blocks lie consecutively in memory and load them into 16 256-bit vectors. In order to have an efficient implementation of the message expansion we have to transpose the content of these vectors. This adds an overhead of 32 pack/unpack and 16 permute instructions. Note that this is not required for the state words, as we can simply the transposed initial value.

By using this data representation the round function and message expansion can be implemented very efficiently and we only require. In order to get the correct output representation we have to again transpose the state which adds another 16 pack/unpack and 8 permute instructions.

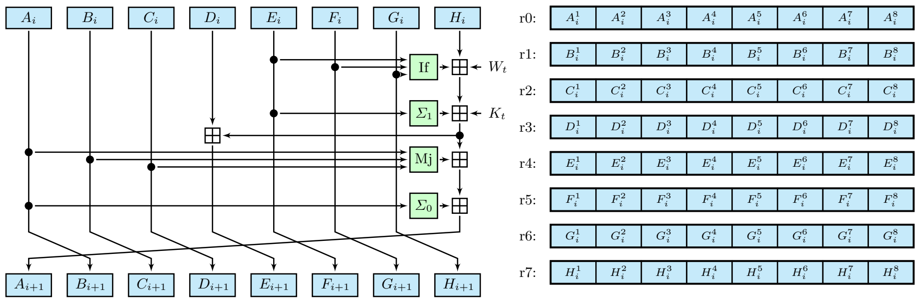

Fig. 5. Mapping of the SHA256 state for the eight blocks to the registers. A_i^j corresponds to the word A as input to round i for block j.

For AMD Ryzen and NEON we use the SHA256 crypto extensions as they result in better performance. As the latency of these instructions is fairly high on both platforms we always interleave four calls in parallel.

4.6 Keccak-F and -H

KangarooTwelve already utilizes SIMD instructions and we base our construction of Keccak-F and Keccak-H on the available implementation [7] of Keccak$[b = 1600, r = 12]$ processing 4 blocks in parallel. The same strategy can be used to implement Keccak$[b = 800, r = 12]$ processing 8 blocks in parallel. Compared to SHA-3 as defined in FIPS PUB 202 we can gain a factor of 4 in speed as we can process double the number of blocks with half the number of rounds when using Keccak$[b = 800, r = 12]$.

For ARM we can use a similar approach, however only 2 (for Keccak$[b = 1600, r = 12]$) resp. 4 blocks can be processed in parallel. For hashing a single input we use the ARMv8 implementation provided in the Keccak Code package [7] and for multiple inputs we implemented a version of Keccak$[b = 800, r = 12]$ processing four blocks in parallel using a strategy similar to the x86 implementation.

4.7 Haraka

For x86 we use the latest version of Haraka available online[4] and the only difference between the platforms is to find the optimal number of parallel calls. Depending on the platform it is better to interleave four or eight calls to Haraka-F resp. Haraka-H, which is related to the latency of the `aesenc` instruction (see Table 3). We therefore use eight calls in parallel on Haswell and four on Skylake/Ryzen.

[4] See https://github.com/kste/haraka

One of the main difference between the AES instructions on Intel and ARM is that on ARM one round of AES is split up in two instructions `aese` and `aesmc`. It is very important that these two instructions are adjacent, as this allows to significantly reduce the latency[5]. Another difference is that on Intel the key is added at the end of the round, as in the Haraka specification. The `aese` instruction on ARM adds the key at the beginning and one AES round is therefore defined as

$$\begin{aligned} \texttt{aesenc} &= \text{AddKey} \circ \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \\ \texttt{aesmc} \circ \texttt{aese} &= \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \circ \text{AddKey}. \end{aligned} \tag{10}$$

For an efficient implementation we can use a different set of round constants to take this into account. Haraka-256 uses the round constants RC_{2i} and RC_{2i+1} in the ith AES layer. In the ARM implementation we use an all zero constant for the first call and RC_0, RC_1 for the second layer. For the third AES layer we compute $RC_2'||RC_3' = \mathbf{mix}_{256}^{-1}(RC_2, RC_3)$ (see Figure 6). Apart from that the implementation can be done in the same way as on Intel. For the mixing operation used in Haraka we can use instead of pack/unpack the equivalent instruction on ARM `zip1` and `zip2`.

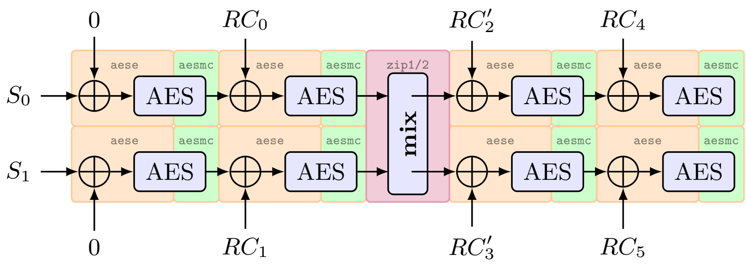

Fig. 6. Implementation of Haraka-256 on ARM using the AES specific instructions. The order of **mix** and the addition of round constants are exchanged to facilitate the free XOR from the key addition of `aese`.

4.8 Simpira

Simpira is another design which utilizes the AES round function in a Feistel network and therefore can be implemented with the AES instructions available on both Intel and ARM. The key addition is used to add a constant and to realize the XOR in the Feistel. On Intel we use the implementation provided by the Simpira designers[6] while for ARM we provide a new implementation.

[5] see ARM Cortex A57 Software Optimization Guide, Page 35

[6] See http://mouha.be/simpira/

Similar to the case of HARAKA it is important to have `aese` and `aesmc` aligned. Also the different order of the key addition needs to be taken into account, which requires an additional XOR per round for $b = 2$ respectively two XORs for $b = 4$ to realize the Feistel networks used in SIMPIRA. In the x86 implementation these XORs are for free as the key addition happens at the end of `aesenc` which can used to XOR with the other branches. Overall this adds a slight overhead compared on the ARM platforms, but still allows a very efficient implementation.

5 Performance Results

We base our implementation of SPHINCS on the source code provided by the SPHINCS authors, which is also available in [6], and instantiate **F** and **H** with the previously discussed primitives to measure the number of cycles required to perform key generation, signing and verification.

The platforms we use for benchmarking include an Intel Haswell (i7-4770S with 3.1 GHz), an Intel Skylake (i7-6700 with 3.4 GHz), an AMD Ryzen (1700 with 3.7 GHz), ARM Cortex A57 (Samsung Galaxy S6 with 2.1 GHz) and an ARM Cortex A72 (Samsung Chromebook Plus with 2.0 GHz). All benchmarks are done on a single core and any frequency scaling technologies like Turbo Boost are deactivated. For measuring the cycle count we use the available performance counter on Intel/AMD and the wall-clock time on ARM. For compiling we use gcc version 6.3.0 with the flags `-O3 -mavx2 -march=native -mtune=native -fomit-frame-pointer` on Intel/AMD and for ARM we crosscompile with `-O3 -mcpu=A57+crypto -fomit-frame-pointer`.

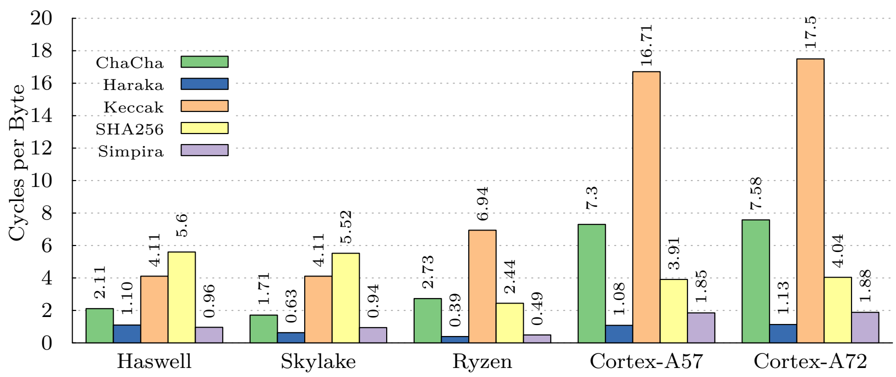

Fig. 7. Performance of **F** on different platforms for processing multiple inputs in parallel. All numbers given are in cycles per byte.

As a first step we measured the performance of **F** and **H** for all our primitives on all platforms (see Figure 7 and Figure 8). We only highlight here the perfor-

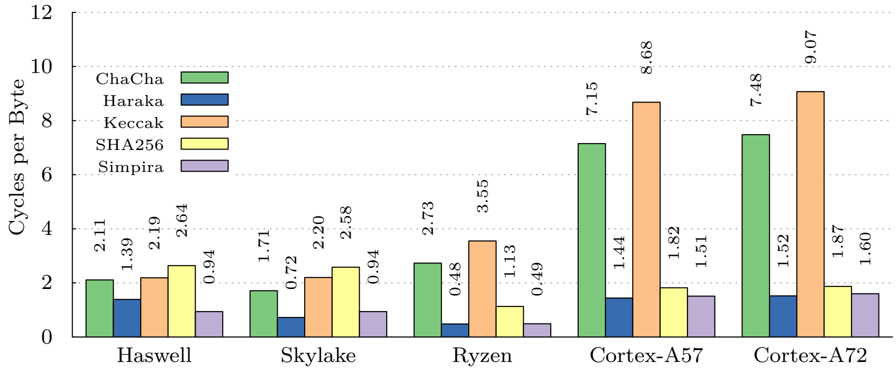

Fig. 8. Performance of **H** on different platforms for processing multiple inputs in parallel. All numbers given are in cycles per byte.

mance for processing multiple inputs in parallel, as in SPHINCS only a minority of the operations can not be parallelized. For single inputs the performance drops especially for the otherwise vectorized implementations of ChaCha, Keccak and SHA256 (on Intel). In general the gap between the implementations utilizing crypto specific instructions and the vectorized implementations is much smaller on Intel than on ARM. Especially, Keccak suffers from the smaller vector size and the higher latency and worse throughput of the vector instructions on ARM (see Table 3).

The performance numbers of these functions reflect directly in the costs for carrying out key generation, signing and verification in SPHINCS. In Table 2, we give an overview of the exact number of cycles required for each operation for the different instantiations of SPHINCS. Unsurprisingly, signing is the most costly operation and allows the biggest gains for highly optimized designs like Haraka and Simpira. As we can see in Table 1, signing requires to call **F** five times more often than **H** and therefore the performance for **F** is of greater importance.

On ARMv8-A the gap between the performance of the primitives without hardware support (ChaCha and Keccak) and those with is much wider. SPHINCS-Haraka is around eight times faster for signing than SPHINCS-Keccak on the ARM Cortex A57, while the biggest gap on Skylake is only a factor of five. This again comes with no surprise, as the underlying functions exhibit a similar difference in performance on this platform. The performance of SPHINCS on mobile devices with the ARM Cortex A57 is very practical and on the Samsung Galaxy S6 used here which has four cores we can compute over hundred signatures per second for the SPHINCS instantiations which utilize hardware support.

TODO: Add some comments on optimality? Level of parallelism is still same as in original SPHINCS, but this will only slightly improve. Assembly, better instruction scheduling could improve some of the implementations

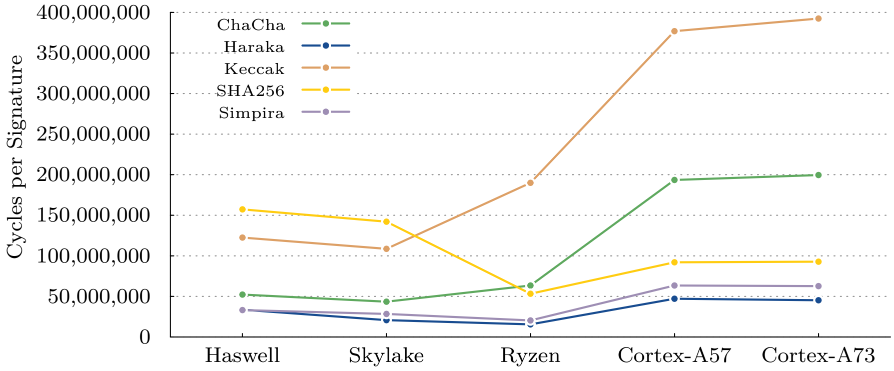

Fig. 9. Number of cycles for signing one message.

5.1 Comparison with other Signature Schemes

To put the performance of SPHINCS into context with other recently proposed post-quantum digital signature schemes we provide a short overview with some of the candidates submitted to the NIST post-quantum competition. For most schemes there is only a limited amount of benchmarks available and those implementations are usually only optimized for x86. We therefore restrict this comparison to the platforms where optimized implementations exist.

Dilithium is a lattice-based signature scheme based on module lattices [15]. The set of parameters for which the authors claim 128-bit post-quantum security leads to a signature size of 2.7kB. The authors also provide an optimized implementation utilizing AVX2 which on Haswell takes 251.590 cycles for key generation, signing 112.716.000 and verification 58.680.000.

Another candidate for lattice-based signature schemes is Falcon [17], which is based on the *short integer solution* problem over NTRU lattices. Choosing parameters which provide a similar security level as SPHINCS leads to a signature size of 1.2kB. The authors also provide benchmarks on Skylake: Key generation takes 64.812.000, signing 1.074.219 and verification 186.472 cycles.

MQDSS [12] is a signature scheme based on the problem of solving multivariate quadratic equations. For the 128-bit post-quantum security level the signature size is comparable to SPHINCS at 41kB. An optimized AVX2 implementation exists and on Haswell the scheme achieves a performance of 1.826.612 cycles for key generation, 8.510.616 for signing and 5.752.612 for verification.

Recently a new digital signature scheme, based on non-interactive zero-knowledge proofs, named *Picnic* has been proposed [11]. The security also is based on the security of symmetric-key primitives similar to SPHINCS[7]. For the proposed parameters and instantiation Picnic has a signature size of 195kB,

[7] The main difference is that SPHINCS has a security proof in the standard model and Picnic in QROM.

Table 2. Benchmarks of SPHINCS on different platforms. All results are the median value of 100 measurements.

Architecture	Primitive	KeyGen	Sign	Verify
Intel Haswell	ChaCha	3.295.808	52.249.518	1.495.416
	Haraka	2.027.136	33.640.796	592.036
	Keccak	7.564.068	122.517.136	2.366.644
	SHA256	9.676.984	157.270.152	3.804.288
	Simpira	2.108.364	33.210.104	595.524
Intel Skylake	ChaCha	2.839.018	43.495.454	1.291.980
	Haraka	1.340.338	20.782.894	415.586
	Keccak	6.589.798	108.629.952	2.152.066
	SHA256	8.724.516	142.063.840	2.812.466
	Simpira	1.808.830	28.408.658	520.832
AMD Ryzen	ChaCha	3.648.660	63.427.980	1.587.120
	Haraka	965.430	15.545.370	258.660
	Keccak	11.354.460	189.986.970	3.739.140
	SHA256	3.267.180	53.332.380	1.090.650
	Simpira	1.261.590	20.439.600	335.790
ARM Cortex A57	ChaCha	10.361.344	193.512.960	3.488.256
	Haraka	2.246.656	47.100.928	717.824
	Keccak	22.006.272	376.908.288	7.358.464
	SHA256	5.292.032	92.088.832	1.679.872
	Simpira	3.362.304	63.489.536	1.108.992
ARM Cortex A72	ChaCha	10.940.928	199.582.208	3.666.944
	Haraka	2.320.384	45.261.312	737.280
	Keccak	22.963.712	392.445.952	7.640.064
	SHA256	5.359.616	92.767.744	1.717.760
	Simpira	3.412.480	62.707.712	1.131.520

however contrary to SPHINCS the size of the signature is also influenced by the choice of the symmetric-key primitive. The authors provide benchmarks on Haswell using LowMC: Key generation takes 36.000, signing 112.716.000 and verification 58.680.000 cycles.

6 Conclusion

We presented a detailed discussion of how to instantiate SPHINCS, what the requirements are and how the performance relates to the underlying cryptographic hash function. Further, we provide an overview of promising candidates for instantiating SPHINCS and discuss their security and performance characteristics.

We provided benchmarks on Intel Haswell, Intel Skylake and ARM Cortex A57 for these primitives based on implementations optimized towards the requirements for hash-based signature schemes. Further, we provided a comparison of SPHINCS instantiated with those primitives.

Overall we can see that on current platforms the performance for primitives utilizing the crypto extensions is favorable compared to others and also the difference between Intel and ARMv8-A is smaller. However, all primitives relying on vectorized implementations get a significant slow down on ARMv8-A. Future platforms, with support for larger vectors, are in the pipeline and will very likely give a significant performance boost to hash-based signature schemes and will make those primitives more competitive.

Acknowledgments We would like to thank Christoffer Brøndum for providing a first version of the ARM implementation of Haraka and Jacob Appelbaum for running the benchmarks on the Cortex A72.

This work was supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO).

References

1. Amy, M., Matteo, O.D., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.: Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. Cryptology ePrint Archive, Report 2016/992 (2016), http://eprint.iacr.org/2016/992 9, 10
2. Aumasson, J., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New features of latin dances: Analysis of salsa, chacha, and rumba. In: Nyberg, K. (ed.) Fast Software Encryption, 15th International Workshop, FSE 2008. Lecture Notes in Computer Science, vol. 5086, pp. 470–488. Springer (2008) 9
3. Aumasson, J., Meier, W., Phan, R.C., Henzen, L.: The Hash Function BLAKE. Information Security and Cryptography, Springer (2014) 9
4. Bernstein, D.J.: Chacha, a variant of salsa20. http://cr.yp.to/papers.html#chacha (2008) 8
5. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015. Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer (2015) 2, 3
6. Bernstein, D.J., Lange, T.: ebacs: Ecrypt benchmarking of cryptographic systems. https://bench.cr.yp.to, accessed 11.05.2017 12, 15
7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak code package. https://github.com/gvanas/KeccakCodePackage, accessed 02.05.2017 13
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Kangarootwelve: fast hashing based on keccak-p. Cryptology ePrint Archive, Report 2016/770 (2016), http://eprint.iacr.org/2016/770 9
9. Buchmann, J.A., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Yang, B. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011. Lecture Notes in Computer Science, vol. 7071, pp. 117–129. Springer (2011) 2
10. Chang, D., Kumar, A., Morawiecki, P., Sanadhya, S.K.: 1st and 2nd preimage attacks on 7, 8 and 9 rounds of keccak-224,256,384,512. SHA-3 workshop (August 2014 10

11. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1825–1842. ACM (2017), http://doi.acm.org/10.1145/3133956.3133997 17
12. Chen, M., Hülsing, A., Rijneveld, J., Samardjiska, S., Schwabe, P.: From 5-pass $\mathcal{MQ}$ -based identification to $\mathcal{MQ}$ -based signatures. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10032, pp. 135–165 (2016), https://doi.org/10.1007/978-3-662-53890-6_5 17
13. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Buchmann, J.A., Ding, J. (eds.) Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008. Lecture Notes in Computer Science, vol. 5299, pp. 109–123. Springer (2008) 3, 4
14. David McGrew and, Michael Curcio and, S.F.: Hash-based signatures. https://datatracker.ietf.org/doc/draft-mcgrew-hash-sigs/, accessed 22.05.2017 2
15. Ducas, L., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - dilithium: Digital signatures from module lattices. IACR Cryptology ePrint Archive 2017, 633 (2017), http://eprint.iacr.org/2017/633 17
16. Espitau, T., Fouque, P., Karpman, P.: Higher-order differential meet-in-the-middle preimage attacks on SHA-1 and BLAKE. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology - CRYPTO 2015. Lecture Notes in Computer Science, vol. 9215, pp. 683–701. Springer (2015) 9
17. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier, lattice-based, compact signatures over ntru. Submission to NIST Post-Quantum Competition (2017) 17
18. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004) 2
19. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing. pp. 212–219 (1996) 9
20. Gueron, S., Mouha, N.: Simpira v2: A family of efficient permutations using the AES round function. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016. Lecture Notes in Computer Science, vol. 10031, pp. 95–125 (2016) 10, 11
21. Gueron, S., Mouha, N.: Sphincs-simpira: Fast stateless hash-based signatures with post-quantum security. Cryptology ePrint Archive, Report 2017/645 (2017), http://eprint.iacr.org/2017/645 3
22. Guo, J., Ling, S., Rechberger, C., Wang, H.: Advanced meet-in-the-middle preimage attacks: First results on full tiger, and improved results on MD4 and SHA-2. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010. Lecture Notes in Computer Science, vol. 6477, pp. 56–75. Springer (2010) 8
23. Guo, J., Liu, M., Song, L.: Linear structures: Applications to cryptanalysis of round-reduced keccak. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016. Lecture Notes in Computer Science, vol. 10031, pp. 249–274 (2016) 10

24. Hülsing, A.: W-OTS+ - shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology - AFRICACRYPT 2013. Lecture Notes in Computer Science, vol. 7918, pp. 173–188. Springer (2013) 4

25. Hülsing, A., Rijneveld, J., Schwabe, P.: Armed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In: Cheng, C., Chung, K., Persiano, G., Yang, B. (eds.) Public-Key Cryptography - PKC 2016. Lecture Notes in Computer Science, vol. 9614, pp. 446–470. Springer (2016) 3

26. Jean, J.: Cryptanalysis of haraka. IACR Trans. Symmetric Cryptol. 2016(1), 1–12 (2016) 10

27. Jean, J., Nikolic, I.: Efficient design strategies based on the AES round function. In: Peyrin, T. (ed.) Fast Software Encryption - 23rd International Conference, FSE 2016. Lecture Notes in Computer Science, vol. 9783, pp. 334–353. Springer (2016) 11

28. Khovratovich, D., Rechberger, C., Savelieva, A.: Bicliques for preimages: Attacks on skein-512 and the SHA-2 family. In: Canteaut, A. (ed.) Fast Software Encryption - 19th International Workshop, FSE 2012. Lecture Notes in Computer Science, vol. 7549, pp. 244–263. Springer (2012) 9

29. Kölbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 - efficient short-input hashing for post-quantum applications. IACR Trans. Symmetric Cryptol. 2016(2), 1–29 (2016) 3, 10, 11

30. Leurent, G.: MD4 is not one-way. In: Nyberg, K. (ed.) Fast Software Encryption, FSE 2008. vol. 5086, pp. 412–428. Springer (2008) 8

31. Reyzin, L., Reyzin, N.: Better than biba: Short one-time signatures with fast signing and verifying. In: Batten, L.M., Seberry, J. (eds.) Information Security and Privacy, 7th Australian Conference, ACISP. Lecture Notes in Computer Science, vol. 2384, pp. 144–153. Springer (2002) 4

32. Rompel, J.: One-way functions are necessary and sufficient for secure signatures. In: Ortiz, H. (ed.) Proceedings of the 22nd Annual ACM Symposium on Theory of Computing. pp. 387–394. ACM (1990) 1

33. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. 26(5), 1484–1509 (1997) 1

34. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al.: The arm scalable vector extension. IEEE Micro 37(2), 26–39 (2017) 12

35. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the hash functions MD4 and RIPEMD. In: Advances in Cryptology - EUROCRYPT 2005. pp. 1–18 (2005) 8

36. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Advances in Cryptology - CRYPTO 2005. pp. 17–36 (2005) 8

37. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Advances in Cryptology - EUROCRYPT 2005. pp. 19–35 (2005) 8

A Instructions

In Table 3 we give an overview of the performance characteristics[8][9] of the instructions on the different platforms. Note that on the ARM Cortex A57/A73 a pair of `aese` and `aesmc` will have a latency of 3 and inverse throughput of 1.

Table 3. Comparison of the latency L and inverse throughput T of several instructions used in the implementations.

Instruction	Platform	L	T	Description
`vpxor`, `vpand`, `vpor`	Haswell	1	0.33	XOR/AND/OR of 256-bit vectors.
	Skylake	1	0.33	
	Ryzen	1	0.5	
`veor`, `vand`, `vorr`	Cortex A57	3	2	XOR/AND/OR of 128-bit vectors.
	Cortex A72	3	2	
`vpslld`	Haswell	1	1	Shift of words in 256-bit vectors.
	Skylake	1	1	
	Ryzen	1	2	
`vshl`	Cortex A57	3	1	Shift of words in 128-bit vector.
	Cortex A72	3	1	
`punpckhdq`, `punpckldq`	Haswell	1	1	Interleave upper/lower halves of two 128-bit vectors.
	Skylake	1	1	
	Ryzen	1	0.5	
`zip1`, `zip2`	Cortex A57	3	2	
	Cortex A72	3	2	
`aesenc`	Haswell	7	1	SubBytes, ShiftRows, MixColumns, AddKey.
	Skylake	4	1	
	Ryzen	4	0.5	
`aese`, `aesmc`	Cortex A57	3	1	AddKey, SubBytes, ShiftRows / MixColumns.
	Cortex A72	3	1	
`SHA256RNDS2`	Ryzen	4	2	Two rounds of SHA256.
`SHA256MSG1`	Ryzen	2	0.5	Helper for message expansion.
`SHA256MSG2`	Ryzen	3	2	
`sha256h`	Cortex A57/A72	6	1	SHA256 state update.
`sha256h2`	Cortex A57/A72	6	1	
`sha256su0`	Cortex A57/A72	3	1	SHA256 message expansion.
`sha256su1`	Cortex A57/A72	6	1	

[8] For Intel/AMD see: https://software.intel.com/sites/landingpage/IntrinsicsGuide and http://agner.org/optimize/instruction_tables.pdf.

[9] For ARM see: http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf.