# Multi-Armed SPHINCS$^+$

Gustavo Banegas and Florian Caullery

Qualcomm France SARL, Valbonne, France
{gsouzaba,fcauller}@qti.qualcomm.com

**Abstract.** Hash-based signatures are a type of Digital Signature Algorithms that are positioned as one of the most solid quantum-resistant constructions. As an example SPHINCS$^+$, has been selected as a standard during the NIST Post-Quantum Cryptography competition. However, hash-based signatures suffer from two main drawbacks: signature size and slow signing process. In this work, we give a solution to the latter when it is used in a mobile device. We take advantage of the fact that hash-based signatures are highly parallelizable. More precisely, we provide an implementation of SPHINCS$^+$ on the Snapdragon™ 865 Mobile Platform taking advantage of its eight CPUs and their vector extensions. Our implementation shows that it is possible to have a speed-up of 15 times when compared to a purely sequential and non-vectorized implementation. Furthermore, we evaluate the performance impact of side-channel protection using vector extensions in the SPHINCS$^+$ version based on SHAKE.

**Keywords:** SPHINCS$^+$ · Post-Quantum Cryptography · Digital Signature Algorithms · Hash-based Signatures.

## 1 Introduction

With the last development in the standardization of Post-Quantum Cryptography (PQC) schemes, the migration to these new algorithms has become more and more urgent. However, the constraints for this transition depend highly on the deployment environments and will dictate the choice of algorithms among the standardized options. It is thus important to know about the performance of every scheme on specific targets. Among the standardized algorithms, signatures schemes based on hash functions are standing out by their strong and well-studied security assumptions.

From an historical view, hash-based signatures started with the seminal work of Lamport on One Time Signature (OTS) [17]. Building on Lamport's OTS scheme, Winternitz [14] and Merkle [20] proposed schemes that could sign more than one bit efficiently. Being built on symmetric primitives (hash functions) these schemes are inherently resistant to quantum attacks as long as the underlying primitive is also secure. Some of the most recent hash-based signatures

* Author list in alphabetical order; see https://www.ams.org/profession/leaders/CultureStatement04.pdf. Date of this document: 2023-05-04.

are eXtended Merkle Signature Scheme (XMSS) [9] and its Multi-Trees variant $XMSS^{MT}$ [15], as well as Leighton-Micali Signature (LMS) [19]. These three schemes are now part of the RFC 8708 [11], and the NIST SP 800-208 [22]. At a high level, LMS and XMSS are built over One Time Signatures and therefore suffer from a security loss if they sign two different messages with the same key material. Hence, one needs to keep track of the signatures already issued making these schemes *stateful*.

Unfortunately, this approach might not be realistic for several use-cases. On top of that, the NIST Post-Quantum Cryptography standardization process [21] did not allow for stateful candidates. To overcome these problems, Bernstein et al. proposed a tweak on XMSS in [5], and developed a scheme called SPHINCS relying on Few Times Signature (FTS) instead of OTS schemes and some amount of randomization to make it *stateless*. Its variant $SPHINCS^+$ [5, 7] has been chosen to be a standard in the NIST process. We remark that despite some small degradation in its security claim [23], $SPHINCS^+$ continues to be secure.

When one makes a comparison of the finalists in the NIST process, $SPHINCS^+$ presents the longest signatures of the three standardized schemes, the two others being Falcon [12] and Dilithium [3]. Also, the latency for a signature generation is considerably higher. To illustrate the difference, we can use the data from SUPERCOP[1]. On an Intel® i5, $SPHINCS^+$ signature is 11 times slower than Falcon and 60 times slower than Dilithium. It thus becomes important to tackle the performance issue and implement $SPHINCS^+$ taking advantage of modern environments.

*Contributions.* We concern ourselves with the scenario of implementing $SPHINCS^+$ on mobile phones with high-performance System on Chips (SoC). These devices can be tasked with signing a message with stringent time constraints. On the other hand, they often have multiple cores available for parallelism. More specifically, we are exploring strategies to accelerate $SPHINCS^+$ on a Snapdragon® 865 that supports four ARM® Cortex-A77 and four Cortex-A55. Our implementation techniques allow us to speed-up the optimized implementation based on SHA-256 proposed in the $SPHINCS^+$ submission package by a factor of 15. We chose to illustrate our approach $SPHINCS^+$ since it uses $XMSS^{MT}$as a basic structure and our results could easily be extended to the original $XMSS^{MT}$. Moreover, we present details when one uses a side-channel protected version of SHAKE in $SPHINCS^+$. Note that we only discuss the signature procedure in our paper as the verification procedures is a simple hash chain and is on par with other algorithms performance-wise.

*Organization of paper.* In Section 2 we succinctly introduce $SPHINCS^+$. Section 3 presents our test platform. We then describe the optimization that we apply using Single Instruction Multiple Data (SIMD) in Section 4 and those using multithreading strategies in Section 5.

[1] https://bench.cr.yp.to/results-sign.html

## 2 SPHINCS$^+$

SPHINCS$^+$ was announced to be standardized in the ongoing process from NIST on Post-Quantum Cryptography. As mentioned earlier, its security relies on the symmetric primitive that it uses, in this case, a hash function, meaning that, as long as this hash function is quantum-safe, SPHINCS$^+$ is safe too. Its construction depends on three main blocks:

– A FTS scheme called Forest Of Random Subset (FORS),
– A OTS scheme named Winternitz One time Signature + (WOTS+),
– A variant of the Merkle tree Signature scheme eXtended Merkle Signature Scheme (XMSS) called XMSS$^{MT}$(MT stands for Multi-Tree).

Note that those subcomponents only support one or a limited number of signatures. This is circumvented by the instantiation parameters that allow virtually an infinite number of signatures. On top of that, the signature material that will be used is selected by deriving a pseudo-random index from a public seed, and the message. This makes unlikely the reuse an OTS. We now give a short description of the subcomponents.

### 2.1 FORS

The FORS is a FTS scheme, and in the words of [5]: "an improvement of HORST [6] which in turn is a variant of HORS [24]". FORS starts by selecting a hash-function that outputs a $n$-bytes hash, a parameter value $t > 1$ that is a power of 2, and a parameter $k$. The secret key is composed of $kt$ random strings of $n$ bytes each. A FORS tree is a Merkle tree whose $t$ leaves are secret key values. The roots of all the $k$ trees are then hashed all together to compose the public key of FORS. To sign the message $M$, the signature process is the following:

1. Hash $m$, and the hash is split into $k$ parts;
2. each part is interpreted as an integer in $0, \ldots, t-1$;
3. use each of those integers gives to get the index of the leaf to use in the corresponding FORS tree;
4. the leaf and the nodes of the tree are necessary to rebuild the tree root (i.e. the *authentication path*), and it is used as the signature of the corresponding part;
5. the full signature is the concatenation of all the signatures of the different parts.

Figure 1 illustrates a FORS structure. The verification process is simply the hash sequence necessary to obtain the top root, and the comparison with the public key.

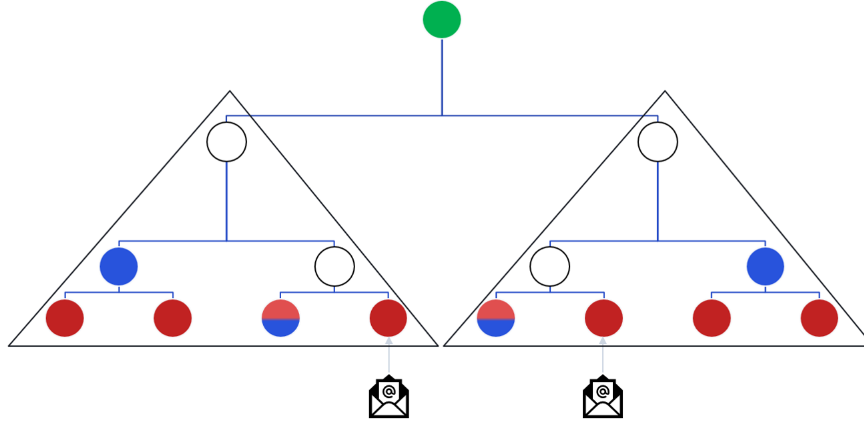

**Fig. 1.** FORS signature explained. The red nodes are the secret key parts, the green node is the public key, the blue nodes are the part of the signature. The blue nodes written *ots* are part of the One Time Signature while others are part of the Authentication path. Blue and red nodes are part of both the signature and the private key, the empty nodes are being reconstructed during the verification.

### 2.2 Winternitz One Time Signature+

The other subcomponent used by SPHINCS$^+$ is the OTS introduced by Hülsing in [14] called Winternitz One Time Signature (WOTS+). WOTS+ starts by selecting a hash-function that outputs a $n$-bytes hash, a parameter value $w > 1$ that is a power of 2 and an extra parameter $l_2$ that represents the checksum length. The private key is composed of $l_1(:= 8 * n/\log_2(w)) + l_2$ blocks of $n$-bytes. The public key is composed of all the private key blocks hashed $w$ times. After hashing the message $M$, a WOTS+ signature is obtained as follow:

1. It splits the hash into $l_1$ parts;
2. each part is interpreted as an integer $b_i$ in $0, \ldots, w-1$;
3. hash the $i$-th private key block $b_i$ times;
4. compute a checksum with as the sum of all the $w - b_i$ and express that number in $l_2$ integers in $0, \ldots, w-1$;
5. hash the $l_2$ remaining private key a number of times given by the checksum integers;
6. the signature consists of all the obtained hashes.

The verification procedure is simply the completion of the hash chains and comparisons with the public key blocks.

### 2.3 XMSS$^{MT}$

The last building block is a hyper tree scheme called XMSS$^{MT}$. To understand the scheme, one needs to notice that it is possible to use an unbalanced Merkle tree to hash the public key of WOTS+ into a single block. Then, by placing $2^a$ WOTS+ public keys as the leaves of the Merkle tree, one can compress those $2^a$ keys into one, that describes XMSS. XMSS$^{MT}$ is built over XMSS by placing an XMSS tree over $2^a$ XMSS trees, so that each leaf of the top tree will sign the public key of the tree below. One can repeat this process iteratively to obtain $d$ layers of trees. This allows for a faster public key generation when compared to a XMSS tree of height $da$, as only the public key of the top tree has to be computed. The signature of XMSS$^{MT}$ is the concatenation of the WOTS+ signatures of the XMSS trees root, and the nodes necessary to reconstruct the public key (i.e. the authentication path).

Figure 2 displays the structure of XMSS$^{MT}$. The verification is simply the hash sequence necessary to obtain the top root, and a comparison with the public key.

### 2.4 SPHINCS$^+$

Finally, after all the previous subcomponents, we can describe SPHINCS$^+$. SPHINCS$^+$ is simply an instance of XMSS$^{MT}$ where the bottom leaves are FORS signatures instead of WOTS+ signatures. The index of the bottom leaf is pseudo-randomly generated from the message and the public key. SPHINCS$^+$ has different parameters, we simplify them as:

- $n$ : the security parameter in bytes.
- $w$ : the Winternitz parameter.
- $a$ : the height of the XMSS trees.
- $d$ : the number of layers in the hypertree.
- $h$ : the height of the hypertree being equal to $da$.
- $k$ : the number of trees in FORS.
- $t$ : the number of leaves of a FORS tree.

Table 1 gives the concrete parameters, we acquire them from the reference documentation [5].

Note that the security claims of the authors of SPHINCS$^+$ and do not integrate the latest cryptanalysis results of [23]. The parameters will certainly evolve according to these attacks, and we will update our implementations when an official change will be made.

## 3 Our test platform: Snapdragon 865

In our tests, we use a Snapdragon® 865. The 865 is an octa core processor, and in its composition it has four ARM® processors Cortex-A77, and four Cortex-A55. Qualcomm® uses different speeds in the processors. Also, they define one

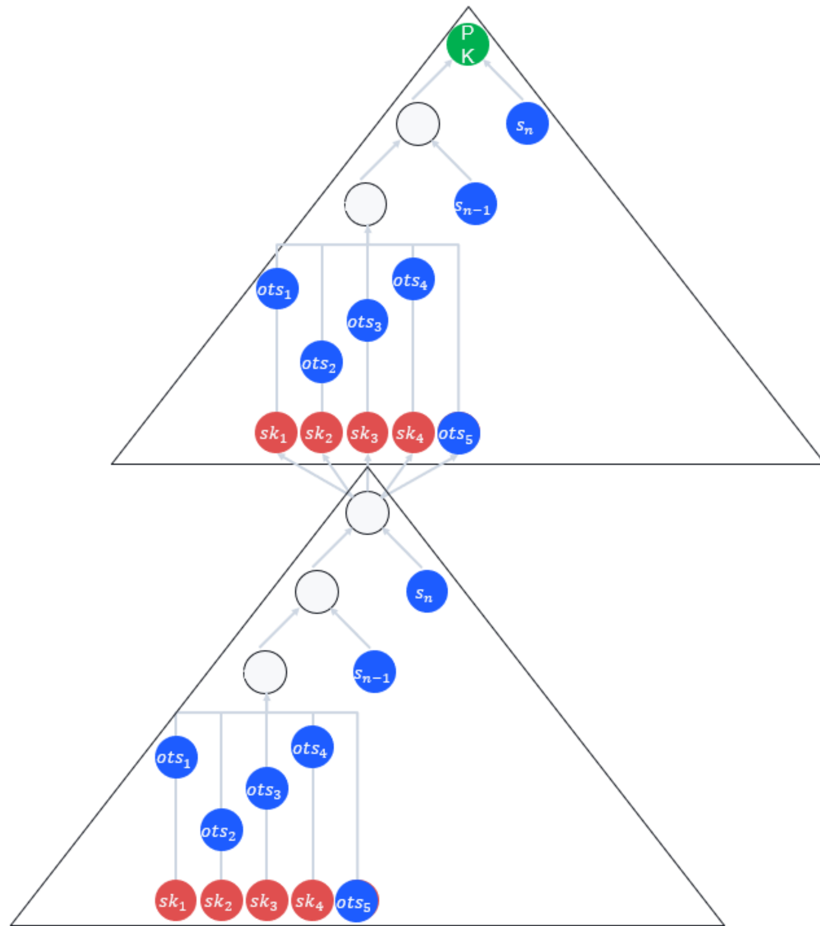


**Fig. 2.** XMSS$^{\mathrm{MT}}$signature explained. The red nodes are the secret key parts, the green node is the public key, the blue nodes are the part of the signature, the empty nodes are being reconstructed during the verification.

**Table 1.** Parameters of SPHINCS$^+$.

| Parameters | $n$ | $h$ | $d$ | $\log(t)$ | $k$ | $w$ | bit security | NIST sec. lvl. | signature (B) |
|---|---|---|---|---|---|---|---|---|---|
| SPHINCS$^+$-128s | 16 | 63 | 7 | 12 | 14 | 16 | 133 | 1 | 7,856 |
| SPHINCS$^+$-128f | 16 | 66 | 22 | 6 | 33 | 16 | 128 | 1 | 17,088 |
| SPHINCS$^+$-192s | 24 | 63 | 7 | 14 | 17 | 16 | 193 | 3 | 16,224 |
| SPHINCS$^+$-192f | 24 | 66 | 22 | 8 | 33 | 16 | 194 | 3 | 35,664 |
| SPHINCS$^+$-256s | 32 | 64 | 8 | 14 | 22 | 16 | 255 | 5 | 29,792 |
| SPHINCS$^+$-256f | 32 | 68 | 17 | 9 | 35 | 16 | 255 | 5 | 49,856 |

Cortex-A77 as Gold tier running at 2.8GHz, three at Silver tier running at 2.4GHz, and the four Cortex-A55 running at 1.8GHz with the DynamIQ ARM® technology.

In our setup, the Snapdragon® 865 is embedded in a smart phone Samsung S20 FE 5G running Android 11. The architecture of the processors is ARM-v8a and all of them have NEON vector coprocessors that operates on 128 bits vectors. The NEON coprocessors have dedicated cryptographic instructions which can accelerate symmetric primitives like AES and SHA-256 but not SHA3.

To compile our code, we are using Clang 14.0.6 with "-Ofast" flag that auto-vectorizes the code and our multithreading is done via posic thread (`pthread`). We are running our compiled code directly on the target via the Android Debug Bridge (ADB) tool, and without an intermediate application. All our results are averaged over 1,000 runs.

We make sure that our code is running on a certain core via the C function `sched_setaffinity`. Note that the Android OS allows to use the three Cortex-A77 at Silver tier, and the four Cortex-A55 but not the Cortex-A77 at Gold tier. This means that we can effectively use 7 out of 8 cores in our experiments.

## 4 SIMD improvements on our test platforms

Prior to any optimization work, the first thing to do is to profile the reference implementation. Our analysis shows that the hash computation represents 99.6% of the time spent in SPHINCS$^+$. The natural step from this analysis is to find strategies to accelerate the underlying hash functions.

### 4.1 Using SIMD to speed-up SPHINCS$^+$

The reference implementation of SPHINCS$^+$ uses the tree hash algorithm given by [10]. The algorithm in [10] is the most optimal currently available in the literature. For the optimized implementations, two strategies are given; The first one applies to the SPHINCS$^+$ variant using Haraka, and it uses the AES-New Instructions (AES-NI) available on both modern Intel processors and ARM-Neon to speed-up single node computations. The speed-up obtained is directly

depending on the ratio between the AES-NI and the pure software AES. The second strategy uses the fact that vector instructions can be used to compute independent hashes in parallel. For example, on ARM-Neon, as Keccak operates on lanes of 64 bits and each vector of ARM-Neon is 128 bits, one can store two independent Keccak states on 25 vectors, and then compute the result of the permutation over the two states at once. When one uses this strategy, it results in a speed-up of *roughly* a factor of 2. The only drawback is that one hash computation is wasted during the computation of the tree root but that is marginal. The AVX2 optimized implementation of SPHINCS$^+$ with SHA-256 uses the second strategy. However, it computes 8 independent hashes in parallel since AVX2 vectors can go up to 256 bits, and SHA-256 operates on 32 bits words. These strategies were first mentioned in [16].

Contrary to Intel AVX2, ARM-Neon offers native instructions to speed-up the computation of SHA-256. One natural question is which strategy is better? Should we use native instructions or take advantage of vectorization to compute several (4 in that case) instances of SHA-256 in parallel. To answer this question, we implement both options and compare them. Our benchmark shows that the throughput of the SHA-256 instructions is around 5 times higher on Cortex-A77 and Cortex-A55 than computing 4 SHA-256 in parallel with Neon instructions. The results are given in Table 2. Hence, we use the native SHA-256 instructions instead of the 4 parallel computations. This fact was already pointed out in [16] for Cortex-A72 and Cortex-A5 and we confirm that those results extrapolate to more recent ARM processors. We also added the comparison with Haraka which is an AES-based hash function specific to SPHINCS$^+$. We implemented it in the way advised in the ARM Software Optimization Guides for Cortex A77 and A55 [1, 2] to take advantage of the pipelining and the instruction fusion. However, Haraka remains slower than SHA-256 on Neon.

Table 3 shows the comparison between our implementation using Neon instructions and the other strategies. For simplicity, we use SPHINCS$^+$ with different primitives but all are the implementations uses security parameter 128f. To differentiate our implementations from the reference package, we present them in italic. It is clear that the SHA-256 variant based on our implementation of SHA-256 with Neon intrinsics is the fastest on our test platform. In the rest of the work, we will use this implementation as a benchmark as any speed-up on it can only translate to equivalent or more important speed-up on slower parameters. We show also for the sake of comparison the performance of Dilithium-90s version (e.g. replacing calls to SHA3 by AES) for a security level 2. We use the implementation of [4] using the AES Neon instructions.

### 4.2 Using SIMD to mitigate side-channel attacks

Some use-cases might require protection against side-channel attacks. Recent works show that vector cryptographic instruction might not offer a satisfying level of protection, see for example [13,18]. Hence, we propose to target SHAKE and implement the software countermeasures described in [8] to raise the level

**Table 2.** Performance comparison on a single core in op/s (higher operations per second is better).

| Primitive | Cortex-A55 | Cortex-A77 |
|---|---|---|
| *SHA-256-Neon (ours)* | 3,955,226 | 9,486,676 |
| SHA-256x4 | 688,912 | 2,145,526 |
| Haraka Neon | 2,589,348 | 9,047,010 |

**Table 3.** Performance comparison on a single core in op/s (higher operations per second is better). The parameters set chosen is SPHINCS$^+$-128f.

| Primitive | Cortex-A55 | Cortex-A77 |
|---|---|---|
| *SPHINCS$^+$ SHA-256 Neon sign (ours)* | 19 | 42 |
| SPHINCS$^+$ Haraka Neon sign | 12 | 35 |
| SPHINCS$^+$ SHAKE256 Neon sign | 3 | 8.6 |
| Dilithium2-90s sign | 2,548 | 4,624 |

of protection against side-channel attacks.

We recall that Keccak, the underlying permutation of SHAKE, is defined by two operations repeated for 24 rounds over a state of 1600 bits: $\lambda$ and $\chi$, $\lambda$ being a purely linear layer and $\chi$ being the non-linear part. The countermeasure proposed in [8] is using two or three shares, that is, every value $x$ of the state is decomposed into two or three values $a$ and $b$, possibly $c$, such that $x = a \oplus b$ or $x = a \oplus b \oplus c$. The linear layer $\lambda$ is kept and applied on the two shares of each value as it is linear (i.e. we have $\lambda(x) = \lambda(a) \oplus \lambda(b)(\oplus \lambda(c))$). The non-linear layer $\chi$ requires more work. The operation $\chi$ is defined by $x_i \leftarrow x_i \oplus ((x_{i+1} + 1).x_{i+2})$ where the indexes are computed in a specific way. Keccak is usually implemented in a bitsliced manner, meaning that $\chi$ will be actually performed with operands of 64 bits (we refer the reader to [8] for a full description of Keccak and its implementation). The two-shared computation of $\chi$ is done using the following formula:

$$a_i \leftarrow a_i \oplus ((a_{i+1} + 1).a_{i+2}) + a_{i+1} b_{i+2}$$
$$b_i \leftarrow b_i \oplus ((b_{i+1} + 1).b_{i+2}) + b_{i+1} a_{i+2}.$$

The authors of this countermeasure then remark that the formulas can be simplified by

$$a_i \leftarrow a_i \oplus ((a_{i+1} + 1).a_{i+2}) + a_{i+1} b_{i+2} \oplus (\bar{b}_{i+1}.b_{i+2}) + b_{i+1} a_{i+2}$$
$$b_i \leftarrow b_i,$$

and that we can pre-compute a fixed mask $b$ such that $b \oplus \lambda(b)$ has a minimal Hamming weight. We argue that such optimization defeats the purpose of masking as a fixed mask satisfying this condition has a vast majority of its bits fixed to 0 and we would be back to perform the operations on an unmasked value. For this reason, we implement the countermeasure without the simplification.

To use make the most out of vector instructions, we put a masked 64bits value into the first part of a Neon vector and its mask in the second part. That is done as presented in Listing 1.1 for a value $x$.

**Listing 1.1.** masking in Neon intrinsic

```
uint64_t x;
uint64_t b = rand_64();
uint64x2_t masked_vector = {x ^ b, b};
```

Performing the $\chi$ operation is then a simple matter of programming with Neon intrinsics and can be done, for example, as presented in Listing 1.2.

**Listing 1.2.** $\chi$ in Neon intrinsic

```
uint64x2_t v0 = {a0, b0};
uint64x2_t v1 = {a1, b1};
uint64x2_t v2 = {a2, b2};

uint64x2_t result = vnegq_s32(v1);
//Negates v1

result = vandq_u64(result, v2);
//result = result AND v2
// result = {(a1 + 1).a2, (b1 + 1).b2}

v2 = vextq_u64(v2, v2, 1);
// swap v2 -> v2 = {b2, a2}

v2 = vandq_u64(v2, v1);
// v2 = {a1.b2, b1.a2}

result = veor3q_u64(result, v2, v0);
//XOR 3 ways

return result;
```

We implemented the whole Keccak permutation using this technique and measured the impact on the performance. The results are presented in Table 4. Note that we are comparing to the reference implementation in the SPHINCS$^+$ package that performs 2 Keccak permutation in parallel. Our result shows that the permutation itself suffers a slow-down of around 60% without including the random generation of the mask. If we include the generation of the mask, we are observing that the permutation is now more than 10 times slower. This

would translate into a signature time 10 times slower as well. Considering that SPHINCS$^+$ with SHAKE is already performing less than 10 signatures per second, we deem this countermeasure as not realistic in an real-world context, unless paired with the multithreading strategy depicted in Section 5.

**Table 4.** Performance comparison on a single core in op/s (higher operations per second is better). The parameter for SPHINCS$^+$ is 128f.

| Primitive | Cortex-A55 | Cortex-A77 |
|---|---|---|
| SHAKEx2 (ref. impl.) | 377,786 | 1,060,445 |
| Masked SHAKE w/o random mask generation | 117,233 | 391,236 |
| Masked SHAKE with random mask generation | 22,099 | 82,911 |

## 5 Parallelization strategies

On a mobile platform, some applications can consider more important to sign a single message than have a higher throughput. In this context, we present a trade-off between fast signing a message and throughput. To speed-up signing in mobile devices, we propose to use multithreading on SPHINCS$^+$. It is important to remark that SPHINCS$^+$ is highly parallelizable. This is possible because the secret key used in the subtrees are solely depending on the master secret key seed and the indexes that are derived from the message. This remark is also valid for FORS and XMSS$^{MT}$.

### 5.1 When Multithreading goes wrong

One of the ways to parallelize computations in SPHINCS$^+$ is to parallelize a single tree hash. For this, one assigns the computation of the nodes to different threads and wait to join the results in a top thread that would compute the root. This strategy might be valid in other contexts, but the subtrees height we are dealing with is at maximum 14 for FORS, and 8 for XMSS$^{MT}$. Creating a thread to perform at best $2^{13}$ hash operations is not a positive trade-off as we show in Table 5. The timing for creating an empty thread is already longer than performing a single tree hash on both Cortex-A55 and Cortex-A77.

### 5.2 Multithreading on FORS

FORS is based on the computation of multiple independent subtrees root, and a final hash of all these roots concatenated. We assign a certain number of trees to each of the threads. As expected, we found out that the speed-up is directly

**Table 5.** Thread creation versus SHA-256 in op/s.

| | Cortex-A55 | Cortex-A77 |
|---|---|---|
| SHA-256-Neon | 3,955,226 | 9,486,676 |
| Thread creation | 3,803 | 6,730 |

linked to the number of cores that can be mobilized. The first conclusion here is that the time used to create the threads is low enough when compared to the tree hashing operations in each thread. To measure this difference, we pushed the number of threads to the number of FORS trees (we assigned one thread per tree hash). We illustrate our results for the fastest variant 128f-simple with SHA-256 intrinsics in Figure 3 [2]. The second conclusion is that FORS on four Cortex-A55 is as fast as on a single Cortex-A77, this makes one Cortex-A77 a more attractive choice than a distribution of computations on the smaller cores.

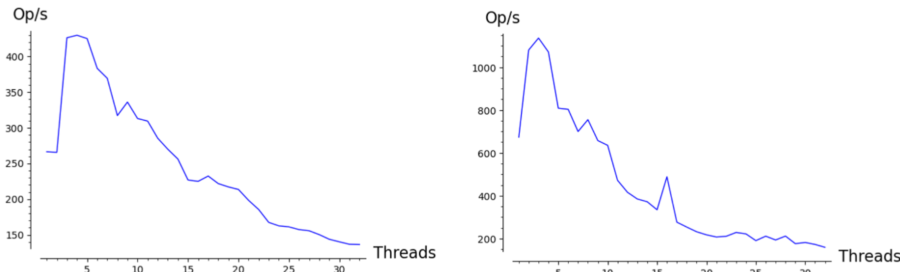


**Fig. 3.** Performance of multithreaded FORS 128f-simple with SHA-256 intrinsics. The abscissa is the number of threads an the ordinate is the number of FORS signatures generated per second. The right graphic is for Cortex-A77 and the left one for Cortex-A55.

### 5.3 Multithreading on $\text{XMSS}^{\text{MT}}$

Multithreading for $\text{XMSS}^{\text{MT}}$ requires a bit more care than FORS as each roots of a subtree needs to be signed by the tree above. That means that, naively, a thread should wait for the computation of the top root of another thread

[2] We are only showing our improvements on the fastest parameter sets as the ratio thread creation time over thread execution time is the least favorable for our experiment. The improvements for slower parameter set will only be better

to be able to start, rendering multithreading effectively useless. However, to compute the authentication path in an XMSS tree, we do not necessarily need to know the message to sign but only its leaf index, we hence use that fact to compute the different parts of the signature in parallel while delegating the signature of the top root of each thread to that same thread (except for the one computing the top tree root). This results into slightly unbalanced thread as earlier thread will often compute one WOTS+ signature more than the last one as shown in Figure 4. What is lost with that strategy is that the WOTS+ signature is normally computed during the computation of the leaf value. This is impossible while we do not know the value of the root to sign, effectively leading to computing this particular WOTS+ twice. Our results show that it only has a marginal effect, and the speed-up is up equivalent to the number of cores available as can be seen for the variant 128f-simple with SHA-256 intrinsics in Figure 5.

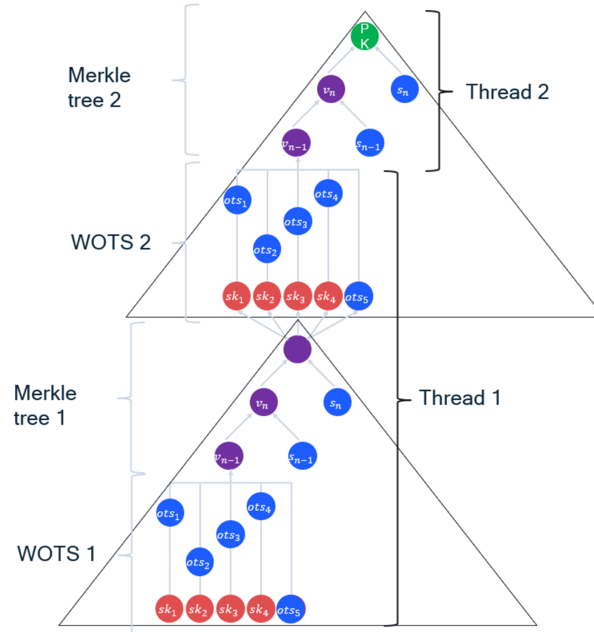


**Fig. 4.** Graphical description of our multithreading strategy

### 5.4 Merging in best case scenarios

As one can notice, the FORS computation on 4 Cortex-A55 takes roughly the same time than the computation of the XMSS$^{\text{MT}}$part of the signature on the

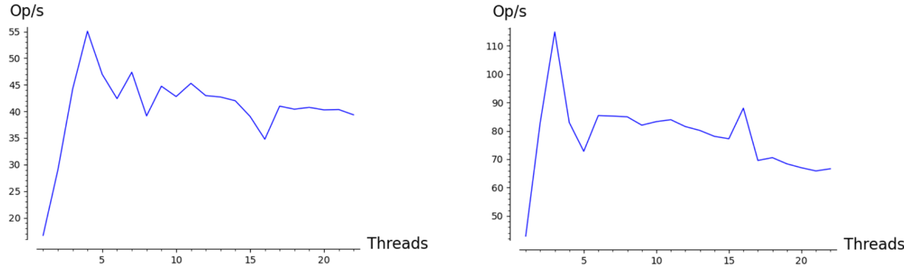


**Fig. 5.** Performance of multithreaded XMSS$^{MT}$ 128f-simple with SHA-256 intrinsics. The abscissa is the number of threads an the ordinate is the number of XMSS$^{MT}$ signature generated per second. The right graphic is for Cortex-A77 and the left one for Cortex-A55.

three available Cortex-A77. Hence, by simply waiting for the computation of FORS to be done on the Cortex-A55 and computing its WOTS+ signature on one of the Cortex-A77, we can parallelize efficiently the full SPHINCS$^{+}$ signature. We measured the performance in this optimal scenario and found out that we can reach 114 signature per second compared to the 40 from the single thread implementation and 8 of the reference implementation. That means an improvement by a factor of roughly 3 when compared to the best implementation and 15 to the reference.

## 6 Conclusion

We have shown that SPHINCS$^{+}$ can greatly beneficiate from a rather simple parallelization strategy on a mobile platform and can come closer to the performance of lattice-based cryptography but are still an order of magnitude slower even while using a lot more resources. We also showed that it is more efficient for hash-based signatures to use SIMD instructions to speed-up a single symmetric primitive computation than using SIMD to compute several primitives in parallel.

*Open Problems.* One future work would be to see if that remark still stands for ARM architecture equipped with Scalable Vector Extensions and SHA3 optimizations. Also, we have seen that software-based side-channel protections are too expansive for realistic deployment. One solution for this problem could be to get the randomness from an efficient hardware Random Number Generator. However, not every microprocessor has such capability and speeding up the random generation in pure software is still an open problem.

## References


1. ARM. Arm cortex-a55 core software optimization guide. https://developer.arm.com/documentation/EPM128372/0300/?lang=en.
2. ARM. Arm cortex-a77 core software optimization guide. https://developer.arm.com/documentation/swog011050/c/.
3. Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation, February 2021. Specification v3.
4. Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, Nov. 2021.
5. Daniel J Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Koölbl, Tanya Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS$^+$, 2017. NIST Submission, available in https://sphincs.org/resources.html.
6. Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.
7. Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2129–2146. ACM, 2019.
8. G. Bertoni, J. Daemen, M. Peeters G. Van Assche, and R. Van Keer. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf.
9. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129, 2011.
10. Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 63–78, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
11. Internet Engineering Task Force. Use of the HSS/LMS Hash-Based Signature Algorithm in the Cryptographic Message Syntax. Internet-Draft RFC 8708, Internet Engineering Task Force, 2020.
12. Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU, 2020. Specification v1.2.
13. Gregor Haas and Aydin Aysu. Apple vs. ema: Electromagnetic side channel attacks on apple corecrypto. In *Proceedings of the 59th ACM/IEEE Design Automation*

*Conference*, DAC '22, page 247–252, New York, NY, USA, 2022. Association for Computing Machinery.

14. Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013.
15. Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS MT. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar R. Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings*, volume 8128 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2013.
16. Stefan Kölbl. Putting wings on SPHINCS. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 205–226. Springer, 2018.
17. Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.
18. Pierre-Yvan Liardet. A potholing tour in a soc. https://eshard.com/posts/sca-attacks-on-armv8.
19. Dr. David A. McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. Internet-Draft draft-mcgrew-hash-sigs-11, Internet Engineering Task Force, 2019.
20. Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238, 1989.
21. NIST. Post-Quantum Cryptography Call for Proposals. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals, 2018. Accessed: 2020-01-01.
22. National Institute of Standards and Technology. Recommendation for stateful hash-based signature schemes. Technical report, U.S. Department of Commerce, Washington, D.C., 2020.
23. Ray A. Perlner, John Kelsey, and David A. Cooper. Breaking category five SPHINCS$^{+}$ with SHA-256. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 501–522. Springer, 2022.
24. Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In Lynn Margaret Batten and Jennifer Seberry, editors, *Information Security and Privacy, 7th Australian Conference, ACISP 2002, Melbourne, Australia, July 3-5, 2002, Proceedings*, volume 2384 of *Lecture Notes in Computer Science*, pages 144–153. Springer, 2002.