Hybrid scalar/vector implementations of Keccak and SPHINCS$^+$ on AArch64

Hanno Becker[1] and Matthias J. Kannwischer[2]

[1] Arm Limited
hanno.becker@arm.com
[2] Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu

Abstract. This paper presents two new techniques for the fast implementation of the Keccak permutation on the A-profile of the Arm architecture: First, the elimination of explicit rotations in the Keccak permutation through Barrel shifting, applicable to scalar AArch64 implementations of `Keccak-f1600`. Second, the construction of hybrid implementations concurrently leveraging both the scalar and the Neon instruction sets of AArch64. The resulting performance improvements are demonstrated in the example of the hash-based signature scheme SPHINCS$^+$, one of the recently announced winners of the NIST post-quantum cryptography project: We achieve up to 1.89× performance improvements compared to the state of the art. Our implementations target the Arm Cortex-{A55,A510,A78,A710,X1,X2} processors common in client devices such as mobile phones.

Keywords: Arm· AArch64· Armv8-A· Keccak· SIMD· Neon · Post-Quantum Cryptography · SPHINCS+

1 Introduction

Hash functions and extendable-output functions based on the Keccak-p permutations have gained popularity since their standardization as SHA-3 and SHAKE in FIPS202 [Dwo15] through the US National Institute for Standards and Technology (NIST) in 2012. Post-quantum cryptography (PQC) in particular makes extensive use of SHA-3 and SHAKE as building blocks: In July 2022, NIST announced [ACD+22] the four schemes it intends to include in its first PQC standard – updating the standards for key-establishment [Nat18,Nat19] and digital signatures [Nat13] – and all four selected schemes make use of SHA-3. Among them is the hash-based signature scheme SPHINCS$^+$ [HBD+22], and three lattice-based schemes: the key-encapsulation scheme Kyber [SAB+22], and the digital signature schemes Dilithium [LDK+22] and Falcon [PFH+22].

While the selected lattice-based schemes provide very good performance and often outperform classical public-key cryptography, hash-based signatures come at a much higher cost. For example, pqm4 [KPR+] – a benchmarking project for post-quantum cryptography on the Arm Cortex-M4 – reports 4 million clock

cycles for signing of `dilithium2`, 18 million clock cycles for `falcon512-tree`, and 400 million clock cycles for `sphincs-sha256-128f-simple` – the *fastest* SPHINCS$^+$ parameter set. While signing performance appears to favour lattice-based signatures, hash-based signatures come with two important advantages: Firstly, they only rely on the collision-resistance and pre-image resistance of the underlying hash function, while lattice-based signatures rely on computation problems (M-LWE, M-SIS, and NTRU). Secondly, hash-based signatures have much smaller public keys of just 32 to 64 bytes, while Dilithium requires at least 1 312 bytes and Falcon requires at least 897 bytes.

Due to the expected upcoming deployment of SPHINCS$^+$, it is essential to understand the performance of SPHINCS$^+$ on a variety of platforms. Unsurprisingly, the computational bottleneck of hash-based signatures are the invocations of the used hash function and, consequently, having a fast hash implementation results in a fast SPHINCS$^+$ implementation. Furthermore, SPHINCS$^+$ can make use of parallel hash implementations which is particularly useful on platforms providing SIMD instructions allowing to compute multiple hashes at once.

In this work, we study scalar and SIMD implementations of `Keccak-f1600` on the AArch64 instruction set of the Arm architecture, and showcase their performance by integrating them into implementations of SPHINCS$^+$. We target the Arm Cortex-{A55,A510,A78,A710,X1,X2} processors common in client devices such as mobile phones, and which are representative of the breadth of implementations of the A-profile of the Arm architecture across the performance/power/area spectrum.

Contributions. We make the following contributions:

1. We shorten scalar AArch64 implementations of `Keccak-f1600` by trading standalone rotations for extensive use of the Barrel shifter. On our target CPUs, this technique leads to a significant performance improvement.
2. We show that 2-way parallel implementations of `Keccak-f1600` using the Armv8.4-A SHA-3 Neon instructions can sometimes be sped up by also mixing in regular Neon instructions, leading to better hardware utilization.
3. We present Scalar/Neon hybrid implementations for 3-, 4- and 5-way parallel `Keccak-f1600`. They compute a 2-way parallel Keccak on the Neon units in parallel with further permutation(s) on the scalar execution units. We investigate such Scalar/Neon hybrids with and without the SHA-3 instructions.
4. We showcase our `Keccak-f1600` implementations by plugging them into SPHINCS$^+$ and achieve signing speed-ups of up to 1.89× over the state of the art.
5. We present detailed analyses of the relation between our target microarchitectures and optimization potentials for our `Keccak-f1600` implementations.

Source code. Our implementations are available at https://gitlab.com/arm-research/security/pqax.

Stateful hash-based signatures. Stateful hash-based signature schemes like XMSS [HBG+18] or LMS [MCF19] can also be implemented in a parallel fashion. Hence, our implementations can be integrated into XMSS or LMS.

Other post-quantum candidates. Other post-quantum schemes also benefit from faster hashing. Notably, Kyber and Dilithium are designed to leverage fast parallel hashing. We therefore believe that our implementations will enable speed-ups for those schemes, but leave a detailed evaluation to future work.

KangarooTwelve. Closely related to SHA-3 is KangarooTwelve [BDP+18] which also builds on the Keccak-p permutation but uses 12 rounds instead of 24. The techniques presented here apply to KangorooTwelve as well.

Related work. Kölbl [Köl18] studies the implementations of SPHINCS (the predecessor of SPHINCS$^+$) for AArch64, targeting the Cortex-A57 and Cortex-A72 CPUs. For Keccak, Kölbl makes use of a two-way parallel Neon implementation from the eXtended Keccak code package (XKCP) [DHP+]. Westerbaan [Wes] presents a two-way parallel Neon implementation of Keccak using the Armv8.4-A SHA-3 instructions. This implementation is also used in the SPHINCS$^+$ NIST PQC submission [HBD+22]. Lattice-based cryptography on AArch64 has been studied by Nguyen and Gaj [NG21] as well as Becker, Hwang, Kannwischer, Yang, and Yang [BHK+21]. Hybrid implementations have previously been applied in other contexts: Bernstein and Schwabe [BS12] present a scalar/vector hybrid implementation of the Salsa20 cipher for Armv7-A, and Lenngren [Len19] presents a scalar/vector hybrid implementation of the key-exchange mechanism X25519 for Armv8-A.

Structure. Section 2 provides background on Keccak, SPHINCS$^+$ and the Arm architecture. Section 3 and Section 4 present our novel implementation techniques for `Keccak-f1600`, including improvements to scalar implementations, parallel Neon implementations, and as the main novelty, hybrid implementations. Finally, in Section 5 we present the performance results for our `Keccak-f1600` and SPHINCS$^+$ implementations on the Cortex-{A55,A510,A78,A710,X1,X2}.

2 Preliminaries

2.1 Keccak

Keccak [BDH+] is a family of permutations, instances of which form the basis of the SHA-3 standard [Dwo15] including the SHA-3 hash functions and the SHAKE extendable output functions (XOF); the reader unfamiliar with the notion of a XOF may think of it as a cryptographic hash function with flexible output size, generalizing the classical use case of hashing arbitrary-size inputs into a fixed-size output. The reverse use case – expanding a fixed-sized input

```
// r[x,y], RC[i] are constants fixed in the specification
keccak-f1600(A)
  for i in 0..23:
    // Θ step
    C[x]   = A[x,0] xor ... xor A[x,4],       for   x=0..4
    D[x]   = C[x-1] xor rot(C[x+1],1),        for   x=0..4
    A[x,y] = A[x,y] xor D[x],                 for x,y=0..4
    // ρ + π steps
    B[y,2*x+3*y] = rot(A[x,y], r[x,y]),       for x,y=0..4
    // χ step
    A[x,y] = B[x,y] xor
             ((not B[x+1,y]) and B[x+2,y]),   for x,y=0..4
    // ι step
    A[0,0] = A[0,0] xor RC[i]
```

Fig. 1: Pseudocode for `Keccak-f1600`

into a variable-size output – is useful, for example, for randomness expansion, and is being used for that purpose in the various NIST PQC schemes.

The core of Keccak within SHA-3 is the Keccak permutation `Keccak-f1600`, operating on a 1600-bit state viewed as a 5×5 matrix $\mathtt{A}[x, y]$ of 64-bit values. It consists of 24 rounds of 5 operations $(\theta, \rho, \pi, \chi, \iota)$ each. χ is the only non-linear operation, while ρ and π are mere bit-permutations, and θ and ι are linear operations. The pseudocode specification of `Keccak-f1600` is given in Figure 1.

2.2 SPHINCS$^+$

Based on SPHINCS [BHH+15], SPHINCS$^+$ [HBD+22] is a stateless hash-based signature scheme that was selected as one of the winners of the NIST PQC project [NIS16]. At the core, SPHINCS$^+$ relies on three building blocks: An improved version of Winternitz One-Time Signatures (WOTS$^+$), the multi-tree version of the eXtended Merkle Signature Scheme (XMSS^{MT}), and the Forest Of Random Subsets (FORS) few time signature scheme. We briefly recall the main concepts and refer to the SPHINCS$^+$ specification [HBD+22] for details.

WOTS$^+$. WOTS$^+$ [Hül13] is a hash-based one-time signature scheme, working roughly as follows: The secret key is a tuple of random values $s_0, s_1, \ldots, s_{\ell-1}$ in the domain of an underlying hash function h, and the public key consists of the repeated hash $h^{2^k-1}(s_i)$, where k is a fixed parameter. Signing works by splitting a message in k-bit blocks $m_i < 2^k$ and revealing the partial preimages $h^{m_i}(s_i)$ of the public keys. Verification checks that they are, in fact, preimages. As stated, this is flawed since knowing the signature for a block m_i allows forging a signature for any $m_i' > m_i$, but this can be fixed through a checksum. We refer the interested reader to [Hül13] for further details.

XMSS^{MT}. The idea of XMSS [BDH11] is to combine multiple one-time public keys into a single many-time public key by means of a hash tree. The leaves of the hash tree correspond to hashes of the one-time public keys, and the root of

the hash tree is the XMSS public key. Signing means signing with one of the one-time keys at the leaves, and providing an authentication path to the root of the hash tree. The signer must carefully track which leaf keys have already been used, and never use the same leaf key twice. XMSS^{MT} builds on XMSS, replacing the single hash tree by a hyper-tree, i.e., multiple layers of XMSS where the WOTS$^+$ keys on upper layers are used to sign the XMSS roots of the lower layers. By doing so, key generation is limited to the upmost tree and signing only needs to compute relatively small trees. However, this comes at the cost of inflated signature sizes as one XMSS^{MT} consists of multiple XMSS signatures.

Eliminating the state. SPHINCS$^+$ eliminates the state from XMSS^{MT} by using a very large hyper-tree and pseudo-randomly selecting leaves for signing. As collisions may still occur, it uses FORS on the lowest layer.

SPHINCS$^+$ parameters. SPHINCS$^+$ specifies 36 parameter sets consisting of all possible combinations of (a) a hash function (SHAKE, SHA-2, or Haraka), (b) a security level (128, 192 or 256 bits), (c) an optimization target (**s** for small signature, or **f** for fast signing), and (d) a tweakable hash function ("simple", comparable to LMS [MCF19], or "robust", comparable to XMSS [BDH11,HBG+18]). Parameter sets are named accordingly, e.g., `sphincs-shake-128f-simple`. In this work, we focus on the SHAKE parameter sets.

(Parallel) hashing in SPHINCS$^+$. Key generation, signing, and verification in SPHINCS$^+$ are dominated by hashing and benefit from parallelization.

We begin with WOTS$^+$-based XMSS, which offers three independent potentials for parallelization: First, it is straightforward to extend WOTS$^+$ key generation to compute multiple hash chains in parallel. This works for any parallelization degree and benefits XMSS key generation, signing and verification. Second, XMSS key generation and signing require the computation of a large hash tree where the leaves are the hashes of freshly generated WOTS$^+$ public keys. This is dominated by the leaf computations and can be sped up by parallelizing multiple WOTS$^+$ key generations. Again, the approach works for any parallelization degree. Third, for 2/4-fold parallelization, a single hash-tree computation may be further parallelized as demonstrated in [HBD+22], though for a WOTS$^+$-based hash tree, this offers only a negligible performance improvement.

We parallelize XMSS verification via the first approach for parallelization, and XMSS key generation and signing via the second. For 2/4-fold parallelization, we also apply the third approach, but mainly for uniformity with FORS: FORS also relies on tree hashing and benefits from the second and third parallelization approaches – moreover, parallelized tree hashing is much more impactful for FORS due to the cheaper leaf computations. For FORS only, which involves *multiple* hash trees, there is also the potential of performing an N-way parallel hash tree computation, but we leave exploring this for future work.

It should be noted that for degrees of parallelization which are not aligned to the total number of invocations, an overhead occurs. For example, the hypertrees

in `sphincs-shake-128f-simple` have only 8 leaves, which does not suit 5-way parallelization. We believe that further study in the best use of the parallelization potentials would be beneficial, and encourage research.

2.3 Arm architecture

Computing based on the Arm architecture is ubiquitous. At the coarsest level, one can distinguish three *profiles*: The application (A) profile targeting high performance markets, such as PC, mobile, and server; the real-time (R) profile for timing-sensitive and safety-critical systems; and the embedded (M) profile for secure, low-power, and low-cost microprocessors.

In this article, we focus on the A-profile of the Arm architecture. Numerous iterations of the A-profile exists, such as Armv7-A, Armv8-A and, as of late, Armv9-A, each including a respective set of extensions. We specifically focus on the AArch64 64-bit instruction set architecture introduced with Armv8-A, as well as the SHA-3 extension which is part of Armv8.4-A.

Implementations of the A-profile of the Arm architecture form a spectrum in itself: To name some, it includes power-efficient CPUs like the Cortex-A7 for Linux-based embedded IoT devices, cores like Cortex-A710 and Cortex-X2 for client devices such as desktops or mobile phones, as well as the Arm® Neoverse™ IP for infrastructure applications. In this article, we focus on two recent generations of cores for the client market, Cortex-{A55,A510,A78,A710,X1,X2}. However, we expect that our optimizations do also apply to the Neoverse N1 and Neoverse V1 infrastructure cores.

It is common for Arm-based SoCs, particularly in the client market, to host multiple CPUs targeting different power/performance profiles, and to dynamically switch between them depending on demand. Originally, this was known as Arm® big.LITTLE™, distinguishing between a high-efficiency "LITTLE" CPU and a high-performance "big" CPU. Nowadays, Arm® DynamIQ™ allows for more flexibility in the configuration of CPUs on a SoC, and DynamIQ-based SoCs often host *three* different Arm CPUs targeting different performance/power profiles. Two such triples are Cortex-{A55,A78,X1} and Cortex-{A510,A710,X2}.

On a microarchitectural level, "LITTLE" cores are typically based on an *in-order* pipeline with some support for superscalar execution. For example, the Cortex-A53 and Cortex-A55 CPUs support dual-issuing of scalar instructions, while the Cortex-A510 CPU is even capable of *triple*-issuing scalar instructions. In terms of SIMD capabilities, Cortex-A53 and Cortex-A55 can single-issue 128-bit Neon instructions, while Cortex-A510 offers an interesting novelty: Pairs of Cortex-A510 CPUs are joined to a Cortex-A510 complex, sharing up to two 128-bit SIMD units. That is, if only one of the cores uses the SIMD units, dual-issuing of 128-bit Neon instructions is possible.

The "medium" Cortex-A7x and "big" Cortex-X cores are based on *out-of-order* pipelines with multiple scalar and SIMD execution units. For example, all of Cortex-{A78,A710,X1,X2} have four scalar execution units. In terms of their SIMD capabilities, Cortex-A7x cores typically have two Neon execution units, while Cortex-X CPUs have four. Such information, as well as detailed listings of

```
// θ step
eor C0, A00, A10
eor C0, C0,  A20
eor C0, C0,  A30
eor C0, C0,  A40
eor C1, A01, A11
eor C1, C1,  A21
eor C1, C1,  A31
eor C1, C1,  A41
eor C2, A02, A12
eor C2, C2,  A22
eor C2, C2,  A32
eor C2, C2,  A42
eor C3, A03, A13
eor C3, C3,  A23
eor C3, C3,  A33
eor C3, C3,  A43
eor C4, A04, A14
eor C4, C4,  A24
eor C4, C4,  A34
eor C4, C4,  A44
eor D1, C0, C2, ROR #63
eor D3, C2, C4, ROR #63
eor D0, C4, C1, ROR #63
eor D2, C1, C3, ROR #63
eor D4, C3, C0, ROR #63
// ρ, π, rest of θ steps
eor B00, A00, D0
eor B40, A02, D2
ror B40, B40, #2
eor B02, A22, D2
ror B02, B02, #21
eor B22, A23, D3
ror B22, B22, #39
eor B23, A34, D4
ror B23, B23, #56
eor B34, A43, D3
ror B34, B34, #8
...
// χ step
bic tmp, B12, B11
eor A10, tmp,  B10
bic tmp, B13, B12
eor A11, tmp,  B11
bic tmp, B14, B13
eor A12, tmp,  B12
bic tmp, B10, B14
eor A13, tmp,  B13
...

// ι step
eor A00, A00, RC
```

Fig. 2: 'Canonical' scalar AArch64 implementation of one `Keccak-f1600` round.

latencies and throughput per instructions, are provided in the publicly available software optimization guides [Armb,Arma,Armd,Armc,Arme,Armf].

3 Keccak on AArch64 — Architecture

This is the first of two section presenting our implementations of `Keccak-f1600` on the AArch64 instruction set architecture, the second being Section 4. Here, we focus on *architectural* considerations: We exhibit ways to express `Keccak-f1600` through the AArch64 instruction set and its extensions. We discuss three approaches: A scalar implementation, an Armv8-A Neon SIMD implementations, and an Armv8.4-A Neon SIMD implementation leveraging the SHA-3 extension.

It is difficult to define meaningful metrics for performance at the architectural level: The number of instructions, the depth and the width (i.e., the amount of instruction level parallelism) of a computation are first approximations, but the actual performance will typically also heavily depend on the target microarchitecture – which is to be expected considering wide range of implementations of the Arm architecture across the performance/power spectrum.

In light of the above, the goal of this section is merely to provide us with a 'pool' of implementation approaches for `Keccak-f1600`. The study of their suitability for our target microarchitectures, as well as further microarchitecture specific optimizations, are the subject of Section 4.

3.1 Scalar implementation

The description of `Keccak-f1600` from Section 2.1 admits a straightforward mapping to the AArch64 instruction set architecture: The 1600-bit state can be maintained in 25 general purpose registers of 64 bits each, and the bitwise operations performed in the θ, ρ, χ and τ steps can be implemented using the `XOR`, `ROR`, `BIC` instructions. This 'canonical' implementation is presented in Figure 2.

Eliminating rotations. The canonical implementation can be shortened by eliminating explicit rotations as follows. For any bitwise operation `OP` such as `XOR` or `BIC`, it holds that $(x \texttt{ OP } y) \lll \texttt{imm} = (x \lll \texttt{imm}) \texttt{ OP } (y \lll \texttt{imm})$, so

$$(x \lll \texttt{imm0}) \texttt{ OP } (y \lll \texttt{imm1}) = (x \texttt{ OP } (y \lll \texttt{imm1-imm0})) \lll \texttt{imm0} \quad (1)$$

```
// θ step
eor C2, A42, A02, ROR #52
eor C0, A00, A10, ROR #61
eor C4, A24, A14, ROR #50
eor C1, A21, A31, ROR #57
eor C3, A03, A23, ROR #63
...
eor C2, C2, A12, ROR #5
eor C0, C0, A40, ROR #25
eor C4, C4, A44, ROR #15
eor C1, C1, A11, ROR #27
eor C3, C3, A43, ROR #2
eor D1, C0, C2, ROR #61
ror C2, C2, 62
eor D3, C2, C4, ROR #57
ror C4, C4, 58
eor D0, C4, C1, ROR #55
ror C1, C1, 56
eor D2, C1, C3, ROR #63
eor D4, C3, C0, ROR #63
```

```
// ρ, π, rest of θ steps
eor B00, D0, A00
eor B40, D2, A02, ROR #50
eor B02, D2, A22, ROR #46
eor B22, D3, A23, ROR #63
eor B23, D4, A34, ROR #28
eor B34, D3, A43, ROR #2
eor B43, D0, A30, ROR #54
eor B20, D1, A01, ROR #43
eor B41, D3, A13, ROR #36
eor B13, D1, A31, ROR #49
eor B21, D2, A12, ROR #3
eor B12, D0, A20, ROR #39
eor B10, D3, A03
eor B03, D3, A33, ROR #37
eor B33, D2, A32, ROR #8
eor B32, D1, A21, ROR #56
eor B11, D4, A14, ROR #44
eor B14, D2, A42, ROR #62
...
```

```
// χ step
bic tmp0, B12,  B11, ROR #47
bic tmp1, B13,  B12, ROR #42
eor A10,  tmp0, B10, ROR #39
bic tmp0, B14,  B13, ROR #16
eor A11,  tmp1, B11, ROR #25
bic tmp1, B10,  B14, ROR #31
eor A12,  tmp0, B12, ROR #58
bic tmp0, B11,  B10, ROR #56
eor A13,  tmp1, B13, ROR #47
bic tmp1, B22,  B21, ROR #19
eor A14,  tmp0, B14, ROR #23
bic tmp0, B23,  B22, ROR #47
eor A20,  tmp1, B20, ROR #24
bic tmp1, B24,  B23, ROR #10
eor A21,  tmp0, B21, ROR #2
bic tmp0, B20,  B24, ROR #47
...
// ι step
eor A00, A00, RC
```

Fig. 3: `Keccak-f1600` round without explicit rotations.

This trivial identity replaces the explicit rotations $x \lll \texttt{imm0}$ and $y \lll \texttt{imm1}$ with the combination of a shifted application of `OP` and an explicit rotation of its result. Since AArch64 offers shifted variants of logical operations as discussed in Section 2.3, this eliminates one explicit rotation. Moreover, if $\texttt{imm0} = 0$ or $\texttt{imm1} = 0$, no explicit rotation remains. Finally, if the result is used in another bitwise operation, the process can be repeated, deferring all explicit rotations to the very end of the computation, where only one rotation per output has to be performed. We call this process "lazy rotation" in the following.

The idea of lazy rotations can be applied to `Keccak-f1600` in order to defer the explicit rotations in the π-step. At first, however, it would seem that the entire `Keccak-f1600` loop would need to be unrolled to benefit from the idea, as performing rotations at a later stage in the loop is still as expensive as performing them at the π-step. Luckily, this is not the case, as we explain now.

Assume we have deferred explicit rotations in the π-step to the end of the first `Keccak-f1600` iteration, so that the true state $\texttt{A}[x, y]$ can be obtained form the software state $\texttt{A}'[x, y]$ via $\texttt{A}[x, y] = \texttt{A}'[x, y] \lll \texttt{s}[x, y]$ for some constants $\texttt{s}[x, y]$. In the θ-step for the next iteration, we can then compute $\texttt{D}[x]$ via lazy rotations, obtaining a value $\texttt{D}'[x]$ so that the true $\texttt{D}[x]$ is again given by $\texttt{D}[x] = \texttt{D}'[x] \oplus \texttt{t}[x]$ for suitable constants $\texttt{t}[x]$. If we then *explicitly* rotate $\texttt{D}'[x]$ to obtain the true $\texttt{D}[x]$, the final part $\texttt{A}[x, y] \leftarrow \texttt{A}[x, y] \oplus \texttt{D}[x] = (\texttt{A}'[x, y] \lll \texttt{s}[x, y]) \oplus \texttt{D}[x]$ can be computed using a shifted `XOR` without any deferred rotation. By breaking the chain of deferred rotations at $\texttt{D}[x]$, we prevent an accumulation of deferred rotations which would otherwise force us to unroll the loop.

The above explains how to trade 25 explicit rotations in the π-step for 5 explicit rotations in the θ-step. In fact, it turns out that 2 of the 5 deferred rotations for $\texttt{D}[x]$ are 0, so that only 3 explicit rotations are necessary. The final result is presented in Figure 3.

Register allocation. We aim to keep most operations in-place to reduce the number of `MOV` operations. In the notation of Figure 1, we'd like $\texttt{loc B}[x, y] = \texttt{loc A}[x, y]$ for most (x, y), where `loc X` is the register location used by `X`. Without

backup MOVs, however, we cannot have loc B$[x, y]$ = loc A$[x, y]$ for all x, y: Otherwise, there'd be cyclic dependencies in the computation of both

$$\mathtt{B}[x', y'] = \mathtt{A}[x, y] \oplus \mathtt{D}[x] \quad \text{and} \tag{$\theta\pi$}$$

$$\mathtt{A}[x, y] = \mathtt{B}[x, y] \oplus (\neg \mathtt{B}[x+1, y] \,\&\, \mathtt{B}[x+2, y]) \tag{χ}$$

preventing in-place computation – we use the shorthand $(x', y') := (y, 2x + 3y)$ here and below. The goal is to slightly offset {loc B[]} from {loc A[]} for the computation of ($\theta\pi$), and to move entries back to their original place in (χ). Concretely, we set loc B$[x, y]$ = loc A$[x, y]$ for $x \notin \{0, 1\}$ and loc B$[x, y]$ = loc A$[x, (y + 1)\%5]$ for $x \in \{0, 1\}$ and $y \in \{1, 2, 3, 4\}$, while using fresh registers for B$[0, 0]$ and B$[1, 0]$ – this choice will become clear soon.

The computation of ($\theta\pi$) then proceeds in a chained fashion: After computing B$[x'_0, y'_0]$ from A$[x_0, y_0]$, we continue with B$[x'_1, y'_1]$ s.t. loc B$[x'_1, y'_1]$ = loc A$[x_0, y_0]$ – that is, once we have used some A[] to compute the corresponding B[], we overwrite its location next. Starting with B$[0, 0]$ or B$[1, 0]$ (which use fresh registers), it terminates once we reach the computation of B$[0', 0']$ or B$[1', 0']$ from A$[0, 1]$ or A$[1, 1]$, because loc A$[0, 1]$ and loc A$[1, 1]$ aren't used by B[].

In principle, the chained computation of ($\theta\pi$) just described does not depend on the particular choice of loc B[], but the lengths of the resulting chains do: Our specific choice leads to a length-24 chain from B$[0, 0]$ to A$[0, 1]$, and a length-1 chain from B$[1, 0]$ to A$[1, 1]$. This matters for register allocation: At the time of ($\theta\pi$), we are already using 30 registers – 25 for the state A[] and 5 for D[] – so only *one* remains, yet we need *two* fresh locations for B$[0, 0]$ and B$[1, 0]$. We solve this by using the single free location for B$[0, 0]$, while *after* computing its length-24 chain, all but one D[] are no longer needed, so B$[1, 0]$ can use any of those.

Finally, we compute (χ), where the special role of $x = 0, 1$ in the definition of loc B[] becomes important: Namely, when we compute A$[0, y]$, A$[1, y]$ from B$[*, y]$, we cannot yet overwrite any loc B$[*, y]$ as they're still needed for subsequent (χ) steps. We thus require loc A$[0, y]$, loc A$[1, y] \notin$ {loc B$[*, y]$}. On the other hand, after computing A$[0, y]$, A$[1, y]$ out-of-place, we may compute A$[2, y]$, A$[3, y]$, A$[4, y]$ in-place since they're no longer used as input for (χ). This motivates our choice of loc B$[x, y]$ = loc A$[x, y]$ for $x \neq 0, 1$, while offsetting loc B$[0, y]$, loc B$[1, y]$.

Overall, the above yields an in-place implementation of a single Keccak-f1600 round using 31 registers, and without using any MOV instructions or stack spilling.

Statistics. Each round in our scalar Keccak-f1600 implementation uses 76× EOR, 25× BIC and 3× ROR instructions, totalling 104 arithmetic instructions. In fact, the first round does not require RORs, but we need 23 RORs after the last round. Overall, we get to $24 \times 104 - 3 + 23 = 2516$ arithmetic instructions for the core of Keccak-f1600. Taking into account function preamble and postamble, we get to 2747 instructions executed per Keccak-f1600 invocation.

3.2 Armv8.4-A Neon implementation

The Armv8.4-A architecture introduces the SHA-3 extension adding the following instructions: EOR3 for the computation of three-fold XORs; RAX1 for the

combination of a rotation-by-1 and a `XOR`; `XAR` for the combination of a `XOR` and a rotation; and finally `BCAX` for the combination of a `BIC` and `XOR`. Those instructions enable a straightforward implementation of `Keccak-f1600` on Armv8.4-A, with `EOR3` and `RAX1` handling the first part of the θ-step, `XAR` handling the second part of the θ-step merged with the ρ-step, and `BCAX` handling the τ-step. The first public implementation along those lines was [Wes]. We slightly refine it by removing explicit `MOV` instructions as detailed in Section 3.1.

Statistics Each round requires 10× `EOR3` instructions, 5× `RAX1` instructions, 24× `XAR`, 2× `EOR` and 25× `BCAX`. Overall, it thus uses $24 \times 66 = 1584$ vector instructions, $24 \times 64 = 1536$ of which from the Armv8.4-A SHA-3 extension.

3.3 Armv8-A Neon implementation

To implement `Keccak-f1600` on Armv8-A Neon instructions, the structure of the Armv8.4-A code can be retained, while implementing `EOR3`, `RAX1`, `XAR`, and `BCAX` as macros based on Armv8-A Neon instructions. Rotations are constructed from a left shift (`SHL`) and a right shift with insert (`SRI`). An implementation along those lines was first developed in [Ngu] based on intrinsics; here, we use a version in handwritten assembly, minimizing vector moves and stack spills.

Statistics When implementing `EOR3` via 2× `EOR`, `RAX1` and `XAR` via `EOR+SHL+SRI`, and `BCAX` via `BIC+EOR`, each `Keccak-f1600` round consists of 76× `EOR`, 29× `SRI`, 30× `SHL` and 25× `BIC` instructions totalling 160 Neon instructions per round, and $24 \times 160 = 3840$ Neon instructions for all of `Keccak-f1600`.

4 `Keccak-f1600` on AArch64 — Microarchitecture

Here, we study the implementations presented in Section 3 from a microarchitectural perspective. We first comment on each approach separately, and then present Scalar/Neon hybrid implementations, the main novelty of this paper.

4.1 Scalar implementation

Recall from Section 3.1 the main ideas of our scalar `Keccak-f1600` implementation: Eliminating explicit rotations through extensive use of shifted instructions, and eliminating explicit `MOV`s through careful register management.

The cost of shifted instructions. Our rotation-elimination implementation is only useful if shifted instructions have the same throughput as unshifted instructions, which is the case for all our targets Cortex-{A55,A510,A78,A710,X1,X2}. However, there *are* exceptions, such as the Cortex-A72, and for such CPUs, rotation-elimination may lead to worse performance despite having a lower instruction count. However, as we see below for Cortex-A55 and Cortex-A510, an increased *latency* for shifted instructions need not be problematic.

```
// Naive
eor C0, A20, A40, ROR #50
eor C0, A30, C0, ROR #49
eor C0, A10, C0, ROR #57
eor C0, A00, C0, ROR #61
eor C1, A41, A11, ROR #60
eor C1, A01, C1, ROR #44
eor C1, A31, C1, ROR #58
eor C1, A21, C1, ROR #57
...
eor C4, A04, A44, ROR #53
eor C4, A34, C4, ROR #56
eor C4, A14, C4, ROR #48
eor C4, A24, C4, ROR #50

ror C1, C1, 56
ror C4, C4, 58
ror C2, C2, 62

eor D1, C0, C2, ROR #63
eor D3, C2, C4, ROR #63
eor D0, C4, C1, ROR #63
eor D2, C1, C3, ROR #63
eor D4, C3, C0, ROR #63
```

```
// Better (fine for A55)
eor C0, A20, A40, ROR #50
eor C1, A41, A11, ROR #60
eor C2, A32, A12, ROR #59
eor C3, A13, A43, ROR #30
eor C4, A04, A44, ROR #53
eor C0, A30, C0,  ROR #49
eor C1, A01, C1,  ROR #44
eor C2, A22, C2,  ROR #26
eor C3, A33, C3,  ROR #63
eor C4, A34, C4,  ROR #56
...
eor C0, A00, C0,  ROR #61
eor C1, A21, C1,  ROR #57
eor C2, A42, C2,  ROR #52
eor C3, A03, C3,  ROR #63
eor C4, A24, C4,  ROR #50

ror C1, C1, 56
ror C4, C4, 58
ror C2, C2, 62

eor D1, C0, C2, ROR #63
eor D3, C2, C4, ROR #63
eor D0, C4, C1, ROR #63
eor D2, C1, C3, ROR #63
eor D4, C3, C0, ROR #63
```

```
// Even better (for A510)
eor C2, A42, A02, ROR #52
eor C0, A00, A10, ROR #61
eor C4, A24, A14, ROR #50
eor C1, A21, A31, ROR #57
eor C3, A03, A23, ROR #63
eor C2, C2,  A22, ROR #48
eor C0, C0,  A30, ROR #54
eor C4, C4,  A34, ROR #34
eor C1, C1,  A01, ROR #51
eor C3, C3,  A33, ROR #37
...
eor C2, C2,  A12, ROR #5
eor C0, C0,  A40, ROR #25
eor C4, C4,  A44, ROR #15
eor C1, C1,  A11, ROR #27
eor C3, C3,  A43, ROR #2

eor D1, C0, C2, ROR #61
ror C2, C2, 62
eor D3, C2, C4, ROR #57
ror C4, C4, 58
eor D0, C4, C1, ROR #55
ror C1, C1, 56
eor D2, C1, C3, ROR #63
eor D4, C3, C0, ROR #63
```

Fig. 4: θ-step optimized for dual-issuing capability of the A55 (middle) and triple-issuing capability of the A510 (right) compared to the naïve approach (left)

The cost of `MOV`s. Eliminating `MOV`s for general purpose registers is a micro-optimization primarily useful for LITTLE CPUs. High-end out-of-order CPUs, in turn, can sometimes implement such `MOV`s with zero latency (e.g. [Arme, Section 4.14]) and therefore show little benefit from reduced register movement.

Optimization for in-order execution. Optimizing code for in-order execution requires careful scheduling of instructions for latency, throughput, and the width of the superscalar pipeline. We now make this concrete for `Keccak-f1600` and our in-order target microarchitectures Cortex-A55 and Cortex-A510.

We begin by discussing Cortex-A55. As detailed in the Software Optimization Guide [Armb], Cortex-A55 is capable of issuing logical instructions with shift at a rate of 2 IPC and a latency of 2 cycles. This is sufficient for a stall-free execution of the column-wise 5-fold `XOR`s in the θ-step, *provided* one alternates between the columns; Figure 4 shows both the naïve and slow approach (left column), as well as an interleaved implementation suitable for Cortex-A55 (middle column).

We next consider the χ-step: Looking at the naïve implementation in Figure 5 (left column), it would seem that with a dual-issuing core and a latency of 2-cycles per shifted instruction, it should stall. However, as explained in [Armb, Figure 1 and Section 3.1.1], the execution of shifted instructions is pipelined, and appropriate forwarding paths provide an effective 1-cycle latency between shifted instructions in case the output of a shifted instruction is used as an *unshifted* input in the consuming instruction; luckily, we are in such a special case.

We now turn to Cortex-A510. As can be seen in the software optimization guide [Arma], Cortex-A510 can issue shifted instructions at a rate of *three* instructions per cycle (the first "LITTLE" core with such capabilities) and 2-cycle

```
// Naive
bic tmp, A12_, A11_, ROR #47
eor A10, tmp,  A10_, ROR #39
bic tmp, A13_, A12_, ROR #42
eor A11, tmp,  A11_, ROR #25
bic tmp, A14_, A13_, ROR #16
eor A12, tmp,  A12_, ROR #58
bic tmp, A10_, A14_, ROR #31
eor A13, tmp,  A13_, ROR #47
bic tmp, A11_, A10_, ROR #56
eor A14, tmp,  A14_, ROR #23
bic tmp, A22_, A21_, ROR #19
eor A20, tmp,  A20_, ROR #24
bic tmp, A23_, A22_, ROR #47
eor A21, tmp,  A21_, ROR #2
...
```

```
// Improved for triple-issuing
bic tmp0, A12_, A11_, ROR #47
bic tmp1, A13_, A12_, ROR #42
eor A10, tmp0,  A10_, ROR #39
bic tmp0, A14_, A13_, ROR #16
eor A11, tmp1,  A11_, ROR #25
bic tmp1, A10_, A14_, ROR #31
eor A12, tmp0,  A12_, ROR #58
bic tmp0, A11_, A10_, ROR #56
eor A13, tmp1,  A13_, ROR #47
bic tmp1, A22_, A21_, ROR #19
eor A14, tmp0,  A14_, ROR #23
bic tmp0, A23_, A22_, ROR #47
eor A20, tmp1,  A20_, ROR #24
bic tmp1, A24_, A23_, ROR #10
eor A21, tmp0,  A21_, ROR #2
...
```

Fig. 5: χ-step optimized for triple-issuing on the A510 (right) compared to the naïve implementation (left)

latency. Moreover, our experiments suggest that we again have a 1-cycle effective latency for outputs of shifted instructions being used as non-shifted inputs.

To leverage the triple-issuing capability of Cortex-A510, the following adjustments have to be made: Firstly, for the columnwise `XOR`s in the θ-step, use the accumulator as a *non-shifted* input only. The right column in Figure 4 shows an adjusted version suitable for triple-issuing on Cortex-A510. Secondly, the χ-step cannot be triple-issued when written as in Figure 5 (left column); instead, one has to manually interleave the computation as in Figure 5 (right column). With those adjustments in place, the `Keccak-f1600` code is mostly triple-issuable, as the performance numbers in Section 5 will confirm.

4.2 Armv8-A Neon implementation

Recall that our Armv8-A implementation replaces the SHA-3 instructions `RAX1`, `XAR`, `BCAX`, and `EOR3` by macros based on Armv8-A Neon instructions.

Suitability for in-order microarchitectures. Generally, implementations based on defining high-level operations such as the SHA-3 operations as assembly macros tend to be unsuitable for in-order execution, as they cannot exploit parallelism at the macro-level and are thus unlikely to obey instruction latencies. For example, on Cortex-A510, `EOR`, `SRI`, and `SHL` have a latency of 3 cycles, so an implementation of `XAR` as `EOR+SHL+SRI` will have a total latency of 9 cycles.

On Cortex-A55, however, we are lucky that logical SIMD instructions have a 1-cycle latency. Moreover, we observe experimentally that `SHL+SRI` pairs synthesizing a rotation do also run without stalls – the implementations of the SHA-3 macros `EOR3`, `BCAX`, `RAX1`, `XAR` can therefore run stall-free. We should expect a performance not far off the optimum of 1 Neon instruction per cycle, and we do not see significant further optimization potential in this approach.

Improving performance of our Armv8-A Neon code on Cortex-A510 requires instruction level parallelism through the interleaving of the SHA-3 macros. Due to the very high register pressure, however, this requires a lot of stack spilling, which is tedious to do by hand. Instead, it makes more sense in this case to work with intrinsics as done in [Ngu], and leave register allocation and stack spilling to the compiler. However, as we shall see in Section 5, both scalar and Armv8.4-A-based `Keccak-f1600` implementations perform better than the Armv8-A based implementation anyway, so the point is moot and we do not explore it further.

Suitability for out-of-order microarchitectures. Generally speaking, the fact that the SHA-3 macros are not scheduled for latency is less problematic for out-of-order cores than for in-order cores, as the microarchitecture will leverage out-of-order execution and register renaming to create the required instruction level parallelism. Still, there is room for further optimization, as we now explain.

The first optimization concerns the availability of functionality on the different SIMD units. For our out-of-order target microarchitectures, the `EOR` and `BIC` instructions can run on all SIMD units. However, the `SHL` and `SRI` instructions, which we use heavily to synthesize 64-bit rotations, are only supported by 50% of the SIMD units – one in the case of Cortex-A78 and Cortex-A710, and two in the case of Cortex-X1 and Cortex-X2. This limits the maximum throughput of the `XAR` and `RAX` blocks, at least when looked at in isolation. In the context of an entire `Keccak-f1600` round, however, `SHL+SRI` make up for *less* than 50% of SIMD instructions, so that manual interleaving of the `XAR` and `RAX` blocks with surrounding code mitigates the throughput loss. Additionally, we replace instances of `SHL X, A, #1` by `ADD X, A, A` (this applies to all `RAX1` blocks and one `XAR` invocation), reducing the pressure on the `SHL/SRI`-capable SIMD units, since (like `EOR` and `BIC`) `ADD` can run on all SIMD units.

The second optimization concerns the θ step: We found that by moving the 5-fold `EOR`s into the previous iteration, we can alleviate a performance bottleneck at the θ step resulting from the lack of instruction level parallelism. For example, with `EOR` having a latency of 2 cycles, one would need at least 8 independent data streams to keep all 4 SIMD units on the Cortex-X1 and Cortex-X2 busy.

4.3 Armv8.4-A Neon implementation

Suitability for in-order cores. As for the scalar implementation, we schedule code for latency to ensure fast execution on Cortex-A510, the basis being the latencies of the SHA-3 instructions as documented in the SWOG [Arma], and the fact each core in a Cortex-A510 complex has up to two SIMD units, depending on whether the other core in the complex is also performing SIMD operations. It is noteworthy that in such a configuration, Cortex-A510 has more throughput for SHA-3 operations than Cortex-A710 and Cortex-X2.

We found that scheduling the code for latency was mostly straightforward, one exception being the `RAX1` instruction, which on Cortex-A510 has a latency of 8 cycles: Here, it seems preferable to express the operation through other Neon instructions of lower latencies.

Suitability for out-of-order cores. For our out-of-order Armv8.4-A targets Cortex-A710 and Cortex-X2, we believe that a "standard" Armv8.4-A implementation along the lines of [Wes] does not have significant microarchitecture-specific optimization potential: As explained in [Armc,Armf], both cores have a single SIMD unit supporting the SHA-3 instructions, limiting a pure Armv8.4-A implementation to 1536 cycles at best, which our implementations already come very close to both for Cortex-A710 and Cortex-X2 – see Section 5.

4.4 Hybrid implementations

The idea for hybrid implementations is simple and general: Given code paths A and B exercising different execution resources, interleave them to facilitate parallel execution by the underlying microarchitecture. Ideally, if the runtimes of A and B are t_A and t_B, respectively, one hopes to achieve the *joint* functionality of A, B in runtime $\max\{t_A, t_B\}$, instead of the sequential $t_A + t_B$.

When constructing a hybrid, one has to consider the individual performance of the code paths to be interleaved, and balance them accordingly to maximize the gain $(t_A + t_B) - \max\{t_A, t_B\} = \min\{t_A, t_B\}$: For example, if path A is $2\times$ as fast as path B, one should interleave 2 copies of A with a single copy of B.

Hybrid implementations have previously been applied in other contexts: Bernstein and Schwabe [BS12] present a scalar/Neon hybrid implementation of the Salsa20 cipher for Armv7-A, and Lenngren [Len19] presents a scalar/Neon hybrid implementation of the key-exchange mechanism X25519 for Armv8-A.

Suitability for different microarchitectures. A hybrid can reach ideal performance $\max\{t_A, t_B\}$ only if the target has the bandwidth to process A and B in parallel. Otherwise, there will be arbitration, with full arbitration leading to sequential performance $t_A + t_B$. It is therefore important to understand the target's maximum $w_{\max}$ of instructions per cycle (IPC), as well as the IPCs w_A ad w_B targetted by A and B. Only if $w_A + w_B \leq w_{\max}$ there is a chance to unlock performance $\max\{t_A, t_B\}$ through a hybrid.

For example, Lenngren [Len19] constructs a Scalar/Neon hybrid for X25519 on Cortex-A53, leveraging that (a) generally, Cortex-A53 can achieve up to 2 IPC, but (b) scalar multiplication and SIMD instructions are limited to 1 IPC. Manual interleaving of scalar and SIMD implementations unlocks an IPC of ≈ 2.

For out-of-order CPUs, the necessity for manual interleaving depends on the target microarchitecture: If the paths to be interleaved are loops of the same size and the out-of-order execution window exceeds the loop body, an alternation of iterations from the two paths may eventually execute in parallel even without manual interleaving. For `Keccak-f1600`, however, each round is large, so we manually interleave scalar and Neon iterations to facilitate parallel execution.

Scalar/Neon Hybrid. We apply the idea of hybrid implementations to our scalar and Neon implementations of `Keccak-f1600`: Concretely, we construct implementations of N-way parallel `Keccak-f1600` by interleaving $N-2$ scalar computations of `Keccak-f1600` with a Neon-based computation of `Keccak-f1600-x2`.

```
1  eor sC0, sA30, sA40   ;   eor3 vC0.16b, vA00.16b, vA10.16b, vA20.16b
2  eor sC1, sA31, sA41   ;
3  eor sC2, sA32, sA42   ;
4  eor sC3, sA33, sA43   ;   eor3 vC0.16b, vC0.16b, vA30.16b, vA40.16b
5  eor sC4, sA34, sA44   ;
6  eor sC0, sA20, sC0    ;
7  eor sC1, sA21, sC1    ;   eor3 vC1.16b, vA01.16b, vA11.16b, vA21.16b
8  eor sC2, sA22, sC2    ;
9  eor sC3, sA23, sC3    ;
10 eor sC4, sA24, sC4    ;   eor3 vC1.16b, vC1.16b, vA31.16b, vA41.16b
11 eor sC0, sA10, sC0    ;
12 eor sC1, sA11, sC1    ;
13 eor sC2, sA12, sC2    ;   eor3 vC2.16b, vA02.16b, vA12.16b, vA22.16b
14 ...
```

Fig. 6: Interleaving of scalar and Armv8.4-A `Keccak-f1600` code

Interleaving the scalar and Neon `Keccak-f1600` implementations was straightforward since the only shared architectural resource is the loop counter. Practically, we wrote code side by side to facilitate readability, as shown in Figure 6.

The choice of N depends on the relative speed of the scalar and Neon code. For example, on Cortex-X1 and Cortex-X2, we chose $N = 3$ and $N = 4$, implementing `Keccak-f1600-xN` from one or two scalar `Keccak-f1600` and one Neon `Keccak-f1600-x2`. On Cortex-A78, we found that $N = 5$ was more suitable.

We comment on the feasibility of hybrids on our targets: For Cortex-A55 and Cortex-A510, our scalar code come close to the issue limit of 2 and 3 IPC, while the SIMD code reaches less than 1 IPC on Cortex-A55 and close to 2 IPC on Cortex-A510. We don't see meaningful speedup through hybrids.

For Cortex-A78 and Cortex-A710, the scalar and Neon implementations target an IPC of 4 and 2, respectively. Since Cortex-A710 has a maximum IPC of 5, they cannot be interleaved without penalty. Cortex-A78, in turn, has a maximum IPC of 6, so a scalar/Neon appears feasible. However, our initial attempt of constructing `Keccak-f1600-x5` on Cortex-A78 fell $> 20\%$ short of our expectations, and only after a significant code-size reduction, we achieved the desired performance. We explain this as follows: While Cortex-A78 has a maximum IPC of 6, the instruction decoder has a maximum IPC of 4. An IPC > 4 can only be unlocked through the use of the "MOP-cache", hosting decoded instructions, but our unrolled code failed to achieve a good hitrate. Once the code was shortened to fit in the MOP-cache, performance reached the expected level.

Neon/Neon Hybrid. An implementation based purely on the Armv8.4-A SHA-3 instructions will only exercise those Neon units implementing the SHA-3 extension. In the case of our targets Cortex-A710 and Cortex-X2, these are 50% and 25% of all Neon units, respectively – the remaining units stay idle.

We have therefore developed hybrid Armv8-A/Armv8.4-A implementations of `Keccak-f1600-x2`, mixing SHA-3 instructions with regular Neon instructions, to achieve better utilization of the SIMD units. This is a different kind of hybrid than the Scalar/Neon one, as we're alternating between different implementation

strategies rather than interleaving them. The balance between SHA-3 and regular Neon instructions depends on the share of SIMD execution units implementing the SHA-3 instructions. For example, on Cortex-X2, we strive for 3 regular Neon instructions for 1× SHA-3 instruction, keeping all four SIMD units busy, while on Cortex-A710, the balance should be 1/1.

Scalar/Neon/Neon Hybrid. Finally, we have also experimented with "triple" hybrid implementations interleaving a scalar implementation with the Neon/-Neon hybrid described in the previous section. In addition to `Keccak-f1600-x4`, we also considered an implementation `Keccak-f1600-x3` interleaving one scalar computation with one hybrid Neon/Neon implementation of `Keccak-f1600-x2`.

5 Results

5.1 Benchmarking environments

Cortex-{X1,A78,A55}. Our first benchmarking platform is a Lantronix Snapdragon 888 hardware development kit with a Qualcomm Snapdragon SM8350P SoC. It is an Arm DynamIQ SoC featuring one high-performance Arm Cortex-X1 core, three Arm Cortex-A78 cores, and four energy-efficient in-order Cortex-A55 cores. The SoC implements the Armv8.2-A instruction set. It also implements the Armv8.4-A dot product instructions, but not the Armv8.4-A SHA-3 instructions. The hardware development kit comes with a rooted Android 11 which allows us to run cross-compiled static executables.

Cortex-{X2,A710,A510}. Our second benchmarking platform is a Samsung S22 smartphone with a Samsung Exynos 2200 (S5E9925) SoC. It is an Arm DynamIQ SoC consisting of one high-performance Cortex-X2 core, three Cortex-A710 cores, and 4 energy-efficient in-order Cortex-A510 cores – the first generation of cores implementing the Armv9-A architecture. The Armv8.4-A SHA-3 extension is also implemented. The SoC is running a rooted Android 12. Our benchmarks suggest that the four Cortex-A510 cores are paired in two Cortex-A510 complexes with shared SIMD units; our benchmarks only use one Cortex-A510 a time, therefore allowing it to leverage 2 SIMD units.

Compiler and benchmarking. We cross-compile our software using the Arm GNU toolchain[3] version 11.3.Rel1. We then copy the executable to the device and run it on a specific core via `taskset`. If we find the desired core disabled for power-saving, we first create artifical load on the system to re-enable it. We use the `perf_events` syscalls for cycle counting. For benchmarking our individual `Keccak-f1600` functions, we warm the cache by running the function 1 000 times, and then report the median of 100 samples of the average performance of 100 function invocations (the averaging amortizes the cost of the `perf` syscalls).

[3] https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads

5.2 `Keccak-f1600` performance

The results of our performance measurements for `Keccak-f1600` are shown in Table 1. As reference points, we use the `crypto_hash/keccakc512/simple` scalar C implementation from SUPERCOP [Kee], the Armv8-A implementation from [Ngu], and the Armv8.4-A implementation from [Wes]. We will now comment and interpret results for each CPU separately.

Approach			Cortex-X1	Cortex-A78	Cortex-A55
C	[Kee]	1x	811 (811)	819 (819)	1 935 (1935)
Scalar	Ours	1x	690 (690)	709 (709)	1 418 (1418)
Neon	[Ngu]	2x	1 370 (685)	2 409 (1204)	5 222 (2611)
Neon	Ours	2x	1 317 (658)	2 197 (1098)	4 560 (2280)
Scalar/Neon	Ours	4x	1 524 (381)	2 201 (550)	7 288 (1822)
Scalar/Neon	Ours	5x	2 161 (432)	2 191 (438)	8 960 (1792)
Approach			Cortex-X2	Cortex-A710	Cortex-A510
C	[Kee]	1x	817 (817)	820 (820)	1 375 (1375)
Scalar	Ours	1x	687 (687)	701 (701)	968 (968)
Neon	[Ngu]	2x	1 325 (662)	2 391 (1195)	3 397 (1698)
Neon	Ours	2x	1 274 (637)	2 044 (1022)	6 970 (3485)
Neon+SHA-3	[Wes]	2x	1 547 (773)	1 550 (775)	2 268 (1134)
Neon+SHA-3	Ours	2x	1 547 (773)	1 549 (774)	1 144 (572)
Neon/Neon+SHA-3	Ours	2x	944 (472)	1 502 (751)	4 449 (2224)
Scalar/Neon/Neon+SHA-3	Ours	3x	985 (328)	1 532 (510)	4 534 (1511)
Scalar/Neon	Ours	4x	1 469 (367)	2 229 (557)	7 384 (1846)
Scalar/Neon+SHA-3	Ours	4x	1 551 (387)	1 608 (402)	3 545 (886)
Scalar/Neon	Ours	5x	2 152 (430)	2 535 (507)	7 169 (1433)
Scalar/Neon/Neon+SHA-3	Ours	4x	1 439 (359)	1 755 (438)	4 487 (1121)

Table 1: Cycle counts for various implementations of `Keccak-f1600`. "Neon+SHA-3" indicates implementations using the SHA-3 instructions. Numbers in brackets are normalized with respect to the amount of parallelization.

Cortex-A55 and Cortex-A510. We observe a significant speedup from the C scalar implementation to our hand-optimized assembly implementation: 1.36× for Cortex-A55 and 1.42× for Cortex-A510. We further note that the scalar performance is close to the theoretical optimum: With ≈ 2750 instructions in total (see Section 3.1) and a maximum issue rate of 2 instructions per cycle on Cortex-A55, the theoretical performance limits on Cortex-A55 are ≈ 1375 cycles. Similar, the maximum issue rate of 3 instructions per cycle on Cortex-A510 leads to a theoretical performance limit of ≈ 917 cycles.

As expected (see Section 4.2), the pure Neon implementation is not competitive for neither Cortex-A55 nor Cortex-A510. In particular, we confirm that the

macro-based implementation performs very poorly on Cortex-A510 since latencies are not obeyed, while the intrinics-based implementation from [Ngu] does better at scheduling the code for latency.

For Cortex-A510, we observe a significant speedup from the Armv8.4-A implementation, explained by the presence of 2 SIMD units capable of executing the SHA-3 Neon instructions. The very large performance gap between our implementation and that of [Wes] is largely due to the high latency of `RAX1`, which we have circumvented as described in Section 4.3.

Finally, we observe that hybrid implementations are not beneficial on in-order cores, as we expected in Section 4.4.

We take away that Cortex-A55 and Cortex-A510 have very efficient scalar implementations which fully leverage the potential for superscalar execution. On Cortex-A55, the scalar implementation should even be used for batched applications of `Keccak-f1600`. On Cortex-A510, batched applications of `Keccak-f1600` should use the Armv8.4-A based implementation.

Cortex-A78 and Cortex-A710. We observe a speedup of 1.15× for our scalar implementation compared to the baseline C implementation. We don't gain as much as for Cortex-A55 and Cortex-A510, which is expected since scheduling for latency is less important for out-of-order cores. Moreover, our scalar implementation is close to the theoretical optimum: With 2516 arithmetic instructions in the core of `Keccak-f1600`, and 4 scalar units, performance is bounded by ≈ 629 cycles, *ignoring* preamble and postamble.

Next, we comment at the Armv8-A Neon performance. Recalling that the core of the implementation performs 3840 Neon arithmetic instructions, and Cortex-A78 and Cortex-A710 have maximum Neon IPC of 2, our implementations are reasonably close to the theoretical optimum, yet around 1.5× slower than the scalar implementation. For Cortex-A78, the `Keccak-f1600-x5` hybrid achieves near optimal performance, leveraging up to 6 IPC on Cortex-A78. For Cortex-A710 in turn, we confirm that the 5-way hybrid cannot work due to the maximum of 5 IPC on Cortex-A710.

Finally, we look at the Armv8.4-A Neon performance. With a single Neon unit implementing the SHA-3 instructions, we cannot do better than 1536 cycles, and our implementation comes very close to that, providing a meaningful speedup of 1.32× over the Armv8-A Neon implementation. Yet, it is still slightly slower than the scalar implementation, but a `Keccak-f1600-x4` scalar/Armv8.4-A Neon hybrid gets the best of the fast scalar implementation and the SHA-3 instructions. This implementation leverages the maximum throughput of 5 IPC for Cortex-A710: 4 IPC for the scalar implementation, and 1 IPC for the Neon implementation. This also explains why the Scalar/Neon/Neon hybrid is worse than the Scalar/Neon hybrid: There is no bandwidth to leverage all four scalar units and both Neon units in every cycle.

Cortex-X1 and Cortex-X2. For scalar `Keccak-f1600`, we get essentially the same performance as for Cortex-A78 and Cortex-A710, and the same comments

apply – this is unsurprising given that Cortex-{A78,A710,X1,X2} all have the same throughput and latency for the relevant scalar instructions.

Next, we look at the performance of the Armv8-A Neon implementations. We observe that it is 5%-10% faster than 2× the scalar implementation – i.e., for batched computations of `Keccak-f1600`, scalar and Armv8-A Neon implementation are roughly on par. We also note that the performance is lower than what the theoretical maximum of 4 Neon IPC for Cortex-X1 and Cortex-X2 would suggest: With 3840 Neon arithmetic instructions, one could hope for ≈ 1000 cycles. We believe that the difficulty in going significantly beyond 3 IPC lies in the `Keccak-f1600` computation "narrowing" at the θ step, and in the `SHL+SRI`-based rotations having a maximum IPC of 2 (see Section 4.2). Nonetheless, we cannot exclude further optimization potential, and encourage research.

The roughly equal performance of scalar and Armv8-A Neon implementation motivates why we pair 2× and 1× `Keccak-f1600-x2` when constructing the Scalar/Neon-Armv8-A hybrid for `Keccak-f1600-x4`. We observe that the hybrid is only slightly above the theoretical optimum, confirming that the frontends of Cortex-X1 and Cortex-X2 are wide enough to process both implementations.

Next, we comment on the performance of the Armv8.4-A Neon implementation on Cortex-X2. First, one observes that the pure Armv8.4-A implementation is slower than the Armv8-A implementation. While this may come as a surprise, the reason is clear: The SHA-3 instructions are implemented on 1 out of 4 Neon units, while the logical operations underlying the Armv8-A implementation are available on all units. Accordingly, we observe a significant speedup for the Neon/Neon hybrid, since it puts all Neon units to use. In fact, this hybrid is sufficiently fast to make a 3-way batched Scalar/Neon/Neon hybrid useful, and this implementation yields the best batched performance. A 4-way batched Scalar/-Neon/Neon implementation brings little benefit compared to a Scalar/Armv8-A Neon hybrid: that's because the bottleneck is the scalar code anyway.

5.3 SPHINCS$^+$ performance

Table 2 shows the performance of SPHINCS$^+$ based on our `Keccak-f1600` implementations, in comparison to previous implementations. We only display results for the "robust" 128-bit parameter sets, but note that our implementations work for all other parameter sets, too, and show similar speedups. Full results are available alongside the code. We see significant performance improvements of up to 1.89× compared to the state of the art.

Acknowledgments

Matthias J. Kannwischer was supported by the Taiwan Ministry of Science and Technology through Academia Sinica Investigator Award AS-IA-109-M01 and the Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

Parameter set	Impl.	Key Generation	Signing	Verification
		Cortex-X1		
128f-robust	C [Kee]	7 358k	170 826k	11 503k
	[Ngu]	6 112k (1.00 ×)	141 857k (1.00 ×)	9 835k (1.00 ×)
	Ours	3 491k (1.75 ×)	81 198k (1.75 ×)	5 881k (1.67 ×)
128s-robust	C [Kee]	470 976k	3 546 272k	4 168k
	[Ngu]	391 075k (1.00 ×)	2 937 624k (1.00 ×)	3 634k (1.00 ×)
	Ours	223 778k (1.75 ×)	1 681 496k (1.75 ×)	2 139k (1.70 ×)
		Cortex-A78		
128f-robust	C [Kee]	7 507k (1.00 ×)	174 285k (1.00 ×)	11 912k (1.00 ×)
	[Ngu]	10 731k	249 061k	16 939k
	Ours	5 043k (1.49 ×)	117 280k (1.49 ×)	7 949k (1.50 ×)
128s-robust	C [Kee]	479 608k (1.00 ×)	3 603 102k (1.00 ×)	4 277k (1.00 ×)
	[Ngu]	686 059k	5 153 452k	6 359k
	Ours	262 264k (1.83 ×)	2 029 133k (1.78 ×)	2 534k (1.69 ×)
		Cortex-A55		
128f-robust	C [Kee]	18 035k (1.00 ×)	418 555k (1.00 ×)	27 322k (1.00 ×)
	[Ngu]	23 444k	544 203k	37 017k
	Ours	13 078k (1.38 ×)	304 188k (1.38 ×)	21 855k (1.25 ×)
128s-robust	C [Kee]	1 153 927k (1.00 ×)	8 667 372k (1.00 ×)	10 415k (1.00 ×)
	[Ngu]	1 500 186k	11 269 260k	13 301k
	Ours	835 847k (1.38 ×)	6 278 826k (1.38 ×)	6 916k (1.51 ×)
		Cortex-X2		
128f-robust	C [Kee]	7 481k	173 680k	11 409k
	[Ngu]	5 946k (1.00 ×)	138 094k (1.00 ×)	9 400k (1.00 ×)
	[Wes]	6 930k	160 942k	11 298k
	Ours	3 315k (1.79 ×)	77 038k (1.79 ×)	5 544k (1.70 ×)
128s-robust	C [Kee]	479 373k	3 601 405k	4 374k
	[Ngu]	381 170k (1.00 ×)	2 863 365k (1.00 ×)	3 312k (1.00 ×)
	[Wes]	443 343k	3 330 902k	3 937k
	Ours	194 295k (1.96 ×)	1 517 988k (1.89 ×)	1 849k (1.79 ×)
		Cortex-A710		
128f-robust	C [Kee]	7 571k	175 706k	11 796k
	[Ngu]	10 641k	247 082k	17 210k
	[Wes]	6 980k (1.00 ×)	162 090k (1.00 ×)	11 338k (1.00 ×)
	Ours	3 743k (1.86 ×)	87 052k (1.86 ×)	6 071k (1.87 ×)
128s-robust	C [Kee]	483 664k	3 633 790k	4 194k
	[Ngu]	681 006k	5 118 302k	6 188k
	[Wes]	446 644k (1.00 ×)	3 356 044k (1.00 ×)	3 850k (1.00 ×)
	Ours	239 634k (1.86 ×)	1 800 720k (1.86 ×)	2 147k (1.79 ×)
		Cortex-A510		
128f-robust	C [Kee]	13 787k	315 780k	21 640k
	[Ngu]	15 270k	354 191k	24 771k
	[Wes]	10 600k (1.00 ×)	245 623k (1.00 ×)	16 866k (1.00 ×)
	Ours	5 428k (1.95 ×)	125 818k (1.95 ×)	8 920k (1.89 ×)
128s-robust	C [Kee]	871 396k	6 548 093k	7 969k
	[Ngu]	974 307k	7 322 458k	8 397k
	[Wes]	661 699k (1.00 ×)	4 991 715k (1.00 ×)	5 791k (1.00 ×)
	Ours	347 614k (1.90 ×)	2 610 123k (1.91 ×)	3 322k (1.74 ×)

Table 2: Performance results for SPHINCS$^+$. For each platform, we pick the `Keccak-f1600` implementation that achieves the best performance.

References

ACD+22. Gorjan Alagic, David A. Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Daniel Apon. Status report on the third round of the NIST post-quantum cryptography standardization process, 2022-07-05 04:07:00 2022. 1

Arma. Arm Limited. Cortex-A510 Software Optimization Guide. 7, 11, 13

Armb. Arm Limited. Cortex-A55 Software Optimization Guide. 7, 11

Armc. Arm Limited. Cortex-A710 Software Optimization Guide. 7, 14

Armd. Arm Limited. Cortex-A78 Software Optimization Guide. 7

Arme. Arm Limited. Cortex-X1 Software Optimization Guide. 7, 11

Armf. Arm Limited. Cortex-X2 Software Optimization Guide. 7, 14

BDH+. Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, , and Ronny Van Keer. Keccak. https://keccak.team/keccak.html. 3

BDH11. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In *Post-Quantum Cryptography – PQCrypto 2011*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011. https://doi.org/10.1007/978-3-642-25405-5_8. 4, 5

BDP+18. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. KangarooTwelve: Fast hashing based on Keccak-p. In *Applied Cryptography and Network Security – ACNS 2018*, volume 10892 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2018. 3

BHH+15. Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *Advances in Cryptology - EUROCRYPT 2015 - Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015. https://doi.org/10.1007/978-3-662-46800-5_15. 4

BHK+21. 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, 2021. 3

BS12. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012. 3, 14

DHP+. Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP. 3

Dwo15. Morris Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015. 1, 3

HBD+22. Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, 2022. https://sphincs.org/. 1, 3, 4, 5

HBG+18. Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018. https://rfc-editor.org/rfc/rfc8391.txt. 3, 5

Hül13. Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In *Progress in Cryptology - AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013. https://doi.org/10.1007/978-3-642-38553-7_10. 4

Kee. Ronny Van Keer. "simple" Keccak C implementation in SUPERCOP (`crypto_hash/keccakc512/simple`). https://bench.cr.yp.to/supercop.html. 17, 20

Köl18. Stefan Kölbl. Putting wings on SPHINCS. In *Post-Quantum Cryptography – PQCrypto 2018*, volume 10786 of *Lecture Notes in Computer Science*, pages 205–226. Springer, 2018. 3

KPR+. Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4. 1

LDK+22. Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, 2022. https://pq-crystals.org/dilithium. 1

Len19. Emil Lenngren. AArch64 optimized implementaton for X25519, 2019. https://github.com/Emill/X25519-AArch64. 3, 14

MCF19. David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019. https://rfc-editor.org/rfc/rfc8554.txt. 3, 5

Nat13. National Institute of Standards and Technology. FIPS186-4: Digital Signature Standard (DSS), 2013. https://doi.org/10.6028/NIST.FIPS.186-4. 1

Nat18. National Institute of Standards and Technology. NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3. 1

Nat19. National Institute of Standards and Technology. NIST SP 800-56B Rev. 2: Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography, 2019. https://doi.org/10.6028/NIST.SP.800-56Br2. 1

NG21. Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In *Post-Quantum Cryptography – PQCrypto 2021*, volume 12841 of *Lecture Notes in Computer Science*, pages 234–254. Springer, 2021. 3

Ngu. Duc Tri Nguyen. Armv8-A Neon implementation for Keccak-f1600. https://github.com/cothan/NEON-SHA3_2x. 10, 13, 17, 18, 20

NIS16. NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 4

PFH+22. Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, 2022. https://falcon-sign.info/. 1

SAB+22. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, 2022. https://pq-crystals.org/kyber. 1

Wes. Bas Westerbaan. Armv8.4-A implementation for Keccak-f1600. https://github.com/bwesterb/armed-keccak. 3, 10, 14, 17, 18, 20