

Batched LSH on ARMv8 NEON and Its Application to K-SPHINCS⁺

Ji-Won Bang¹[0009-0006-5453-1722], Su-Been Cho¹[0009-0004-4352-8021],
Yu-Lim Hyoun¹[0009-0009-7530-6497], Ha-Gyeong Kim¹[0009-0008-5840-5517],
and Hwa-Jeong Seo¹[0000-0003-0069-9061]

¹Department of Convergence Security,
Hansung University, Seoul (02876), South Korea,
{bgjiiw0n754, chosubin1208, yulim4hyoung, kimhaha4420, hwaajeong84}@gmail.com

Abstract. With FIPS 205 standardizing post-quantum signatures and ARMv8 spanning mobile, embedded, and server platforms, fast hash-based signing on ARM has become a pressing need. Since these schemes are dominated by hash calls, hash optimization largely determines performance. K-SPHINCS⁺ replaces the internal hash of stateless SPHINCS⁺ with LSH, a Korean standard hash family. However, prior K-SPHINCS⁺ work provides only reference C code, and existing ARMv8 LSH implementations mainly optimize single-call latency. We present two results on ARMv8-A. First, we reimplement three published LSH optimizations in hand-written AArch64 NEON assembly: the 2019 word-permutation technique; the 2023 ARMv8 P-domain placement and LSH-256 2-way approach; and the 2024 AVX-512 word-parallel and GPU lane-batching designs. We also test whether NEON scheduling guidance from a 2022 AArch64 Keccak/SPHINCS⁺ paper transfers to LSH. Every variant must pass 2,063 public known-answer test vectors and a differential test suite before benchmarking. No single-message NEON implementation beats the KISA portable C code compiled with `clang -O3` (0.75–0.90×). Only message-parallel batching wins: for LSH-256, hashing two or four independent messages across NEON lanes reaches 1.53–1.66× and about 2.0× the throughput of sequential C. Second, we build, to our knowledge, the first optimized K-SPHINCS⁺ by integrating a four-message LSH batch core into the WOTS⁺ chain, FORS leaf, and Merkle tree computations of the SPHINCS⁺ reference framework. All six parameter sets pass the framework’s tests, and all three hash backends (reference C, single-message NEON, four-message NEON) produce byte-identical signatures. On the same machine, batched K-SPHINCS⁺ signs 2.5–2.6× faster than K-SPHINCS⁺ with reference C LSH and, for the 128-bit parameter sets, 2.9–3.2× faster than the SHA-256/SHAKE256 SPHINCS⁺ reference implementations. All code and raw measurements are public.

Keywords: LSH · SPHINCS⁺ · Hash-based signatures · ARMv8 NEON

1 Introduction

Hash-based signatures are widely regarded as the most conservative choice among post-quantum signatures, because their security relies only on standard properties of the underlying hash function. NIST standardized SPHINCS⁺ as SLH-DSA

in FIPS 205 [1]. In these schemes, however, the cost of hashing dominates the overall cost. Producing a single SPHINCS⁺ signature requires tens of thousands of tweakable hash evaluations, so the practical speed of a hash-based signature is determined largely by the speed of its hash function.

LSH [2], the Korean standard hash function, is an ARX design with a 32-bit core (LSH-256) and a 64-bit core (LSH-512). Sim et al. proposed K-XMSS and K-SPHINCS⁺ [3,4], which replace the SHA-2/SHAKE instances of XMSS [5] and SPHINCS⁺ [6] with LSH and block-cipher-based hashes. Their original implementation is a proof of concept written in reference C, and they reported that the LSH variants were about 2× slower than the SHA-2 variants on an x86 laptop [3]. A later version added AVX2 implementations [4], but no ARMv8 NEON implementation has been reported. Three works have optimized LSH itself with SIMD: the 2019 word-permutation technique for SSE/AVX [7], a 2023 thesis on ARMv8 NEON [8], and the 2024 AVX-512 and GPU implementations [9]. None of these works measures the hash inside a signature scheme. Moreover, the ARMv8 figures were reported for a single machine and a single compiler, without released code.

This paper aims to close this gap by connecting two lines of research: LSH optimization and LSH-based signatures. To this end, we address two questions. First, which of the published LSH optimization techniques actually make LSH faster on current ARMv8 cores than the vendor C implementation compiled with -O3? Second, what performance gains does the answer bring to hash-based signature schemes? Our contributions are as follows.

1. **Reproducible reimplementations (Section 3).** We reimplement three LSH optimization techniques and one Keccak technique in AArch64 NEON assembly. Each core is a self-contained project that includes a copy of the KISA reference implementation, 2,063 public known-answer test vectors collected from three public sources, and a differential test suite. The benchmark measures only binaries that pass all of these checks. Before writing any assembly, we validated the permutation algebra of the 2019 technique and the P-domain placement of the 2023 technique against byte-level Python models. In the process, we documented ten errata in the published descriptions (Section 3.5).
2. **A negative result for single-message NEON LSH (Section 4).** On an Apple M1 with Apple clang 21, every single-message NEON variant is slower than the portable C reference implementation (0.75–0.90×). Controlled ablations of the 2019 permutation, the 2023 folded rotation, and the Keccak scheduling techniques show no repeatable gain beyond measurement noise. We identify three causes: the Mix step is a serial add–rotate–XOR chain, AArch64 Advanced SIMD has no vector rotate instruction, and the LSH-512 message schedule does not fit in the 32 vector registers. On the same hardware under GNU/Linux with GCC 11.4, the ranking reverses (1.08×). In other words, single-message performance depends more on the compiler than on the assembly.

3. **A positive result for message-parallel LSH (Section 4).** Hashing independent messages across NEON lanes is the only technique that outperforms C under both compilers. The 2023 LSH-256 2-way core reaches $1.53\text{--}1.66\times$ the throughput of sequential C, and a 4-message SoA core following the 2024 design reaches about $2.0\times$ ($2.6\times$ with GCC).
4. **To our knowledge, the first optimized K-SPHINCS⁺ (Section 5).** Following the K-SPHINCS⁺ definition, we instantiate the SPHINCS⁺ hash layer with LSH. Using the 8-way leaf/tree framework of the SPHINCS⁺ reference code, we process WOTS⁺ chains, FORS leaves, and Merkle nodes with the 4-message batch core. All six parameter sets pass the framework’s tests, and the reference C, NEON, and batched backends produce byte-identical signatures from the same seed. Batched K-SPHINCS⁺ signs $2.5\text{--}2.6\times$ faster than reference C K-SPHINCS⁺ (Table 8) and, for the 128-bit parameter sets, $2.9\text{--}3.2\times$ faster than the SHA-256 and SHAKE256 SPHINCS⁺ reference implementations measured on the same machine (Table 9).

We state the limitations explicitly in Section 6. Source code, raw CSV data, environment records, and test logs are included with the submission [10].

2 Background

2.1 LSH

LSH [2] is a Merkle–Damgård hash function with a wide-pipe compression function. The word size w is 32 bits for LSH-256 and 64 bits for LSH-512. In both functions, the chaining variable consists of 16 words and a message block consists of 32 words (128 bytes and 256 bytes, respectively). The compression function performs $N_s = 26$ and $N_s = 28$ steps, respectively.

Message expansion. Each message block is split into two 16-word sub-messages M_0 and M_1 . For $j \geq 2$, the remaining sub-messages are generated as

$$M_j[l] = M_{j-1}[l] \boxplus M_{j-2}[\tau(l)], \quad 0 \leq l < 16, \quad (1)$$

where τ is a fixed word permutation.

Step function. Step j consists of MsgAdd, Mix, and WordPerm. MsgAdd XORs the sub-message into the state ($T \leftarrow T \oplus M_j$). Mix processes each of the eight word pairs (T_l, T_{l+8}) , $0 \leq l < 8$, as follows:

$$\begin{aligned} T_l &\leftarrow ((T_l \boxplus T_{l+8}) \lll \alpha_j) \oplus SC_j[l], \\ T_{l+8} &\leftarrow (T_l \boxplus T_{l+8}) \lll \beta_j, \\ T_l &\leftarrow T_l \boxplus T_{l+8}, \\ T_{l+8} &\leftarrow T_{l+8} \lll \gamma_l. \end{aligned} \quad (2)$$

Here, $\boxplus$ denotes addition modulo 2^w , $\lll$ denotes a w -bit left rotation, and SC_j is the step constant. WordPerm rearranges the 16 words by a fixed permutation σ . After the N_s steps, a final MsgAdd with the last sub-message M_{N_s} produces the new chaining variable.

Rotation amounts. The amounts (α_j, β_j) alternate between even and odd steps: (29, 1) and (5, 17) for LSH-256, and (23, 59) and (7, 3) for LSH-512. The per-word amounts γ_l are (0, 8, 16, 24, 24, 16, 8, 0) for LSH-256 and (0, 16, 32, 48, 8, 24, 40, 56) for LSH-512. All of them are multiples of 8, so these are byte-granular rotations.

Output. The initial chaining variable depends on the output length. From the final chaining variable, the function computes $h_l = T_l \oplus T_{l+8}$ for $0 \leq l < 8$ and truncates the result to the output length to obtain the digest. Six output variants are defined: LSH-256-224/256 and LSH-512-224/256/384/512.

2.2 ARMv8 Advanced SIMD (NEON)

AArch64 provides 32 128-bit vector registers, v0–v31. Each register holds four LSH-256 words (.4s) or two LSH-512 words (.2d). The instructions relevant to LSH are the following. ADD and EOR provide the arithmetic and logic. Advanced SIMD has no vector rotate instruction, so an arbitrary rotation (α_j, β_j) takes two instructions, SHL and SRI. A byte-granular rotation (γ_l) , however, takes a single REV or TBL. TBL performs an arbitrary byte permutation of one or two source registers in one instruction. EXT, ZIP, UZP, TRN, and INS move data between lanes.

We do not use the optional Armv8.2 SHA-3 instructions (EOR3, RAX1, XAR, BCAX). They are not part of the base ISA, and they support only 64-bit lanes (.2d), so they cannot be applied to LSH-256 (Section 3.5). For LSH-512, XAR with a zero operand would give a single-instruction 64-bit rotation; we leave its evaluation to future work.

Two kinds of parallelism. We distinguish two kinds of parallelism. *Word parallelism* places several words of one message in the lanes of a register. This is what single-message implementations do, and it is limited by the data dependencies within one hash computation. *Message parallelism* places the same word of several independent messages in the lanes. It has no dependencies between lanes, but the caller must supply several messages of equal length at once. Four 32-bit lanes do not automatically mean four messages. Wherever we claim message parallelism, we state the register layout that achieves it and check the output of every lane individually against the reference implementation.

2.3 SPHINCS⁺ and K-SPHINCS⁺

SPHINCS⁺ [6] is a stateless hash-based signature scheme with three components: a hypertree of d layers of XMSS trees, each of height h/d ; WOTS⁺ one-time signatures at the leaves of each XMSS tree; and the few-time signature scheme FORS, which signs the message at the bottom of the hypertree. The scheme is parameterized by the security parameter $n \in \{16, 24, 32\}$ bytes, the hypertree height h , the number of layers d , and the FORS parameters (a, k) ; the Winternitz parameter is 16 in all parameter sets. The “f” (fast) and “s” (small) parameter sets trade signing time against signature size.

SPHINCS⁺ uses six functions instantiated from a single hash function: the tweakable hash functions F , H , and T_ℓ , the pseudorandom functions PRF and PRF_{msg}, and the message digest H_{msg} . Almost all of the work is in F and H . A WOTS⁺ public key consists of len hash chains of 15 steps each; for $n = 16$, $len = 35$, so generating one public key takes $35 \cdot 15 = 525$ evaluations of F . A FORS signature evaluates F on $k \cdot 2^a$ leaves and H on the tree nodes. Each XMSS tree compresses $2^{h/d}$ WOTS⁺ public keys with T_ℓ and hashes them up to the root with H . The reference framework [11] exposes this work through an x -way leaf generator (`wots_gen_leafx8`) and tree hash (`treehashx8`). An x -way hash implementation can therefore process x independent chains, leaves, or nodes in a single call.

K-SPHINCS⁺ [3,4] keeps the SPHINCS⁺ structure and instantiates all six functions with LSH-256. Sim et al. [3] define them as

$$\begin{aligned} F = H = T_\ell &= \text{LSH256}(\text{PK.seed} \parallel \text{ADRS} \parallel M), \\ \text{PRF} &= \text{LSH256}(\text{SEED} \parallel \text{ADRS}), \\ \text{PRF}_{\text{msg}} &= \text{LSH256}(\text{SK.prf} \parallel \text{OptRand} \parallel M), \\ H_{\text{msg}} &= \text{LSH256}(R \parallel \text{PK.seed} \parallel \text{PK.root} \parallel M), \end{aligned} \tag{3}$$

where each output is truncated to n bytes, except that the output of H_{msg} is m bytes. The tweakable hash functions use the “simple” (unmasked) construction, which is also the only construction adopted in FIPS 205. Section 5.1 describes our instantiation and how it differs from Eq. (3).

3 NEON Reimplementation of Published LSH Techniques

3.1 Method and Measurement Protocol

We built six independent projects, one per experimental condition (Table 1). Each project contains a byte-identical copy of the KISA reference C code v1.0.2 (“clean C”), the same test harness, the same 2,063 known-answer test vectors (KISA’s own 32 vectors, the KCMVP V3.0 validation vectors, and the Crypto++ set), and its own hand-written .S files. Nothing is shared at build time. Any project copied elsewhere builds, tests, and measures on its own, and a script verifies this by doing exactly that. The duplication is intentional: a folder that shares headers with a sibling cannot be verified independently, and assembly inherited from a sibling must not be counted twice as a result. The report lists the hash of every .S file in every folder. Exactly two assembly files (the baseline LSH-256 and LSH-512 cores) appear in two folders, and the report attributes each to one of them.

Correctness before measurement. A variant is measured only after it passes the following checks, built from exactly the same sources and flags as the measured binary:

Table 1: The six independent projects. `clean` is the KISA C code built in the same directory with the same compiler and flags. `clean_sequential_xK` calls that C code K times in a row, giving each batched variant a control with the same number of messages.

Project	Technique	Measured variants
ref	Original KISA C, pinned to the public <code>clean</code> , test vectors	<code>clean_sequential_x2/x4</code>
neon_common	Direct NEON translation of the speci- fication, with no published technique	baseline
neon_2019	Word permutation P of [7], re-selected for NEON cost	paper , neon_selected
neon_2023	P -domain placement of [8]; LSH-256 2- way	single , x2
neon_2024	Word-parallel LSH-512 and SoA lane batching of [9]	cpu_word , gpu_x2 , gpu_x4
aarch64_asm	NEON scheduling guidance of [12] ap- plied to the 2023 core	base , keccak_neon

- *Known answers*: all 2,063 vectors, for all six output lengths, in both one-shot and streaming mode.
- *Differential tests*: at least 1,000 seeded random messages per output length, compared against the local clean C.
- *Edge cases*: empty input, block boundaries, multi-block input, and bit lengths that are not a multiple of 8.
- *Memory*: unaligned buffers and guard pages around the input and output.
- *ABI*: preservation of `x19–x28`, `d8–d15`, `x18`, and the stack pointer.
- *Batch lanes*: per-lane checks with distinct inputs and with the lane order reversed.

The benchmark binary itself enforces this. It binds the hashes of all sources, headers, objects, and executables to the recorded test results, and it reruns the known-answer tests immediately before measuring.

Measurement. Each condition runs a separate warm-up (at least 100 calls and at least 10 ms, excluded from the totals), followed by exactly 10,000 complete hash calls. We report the arithmetic mean in nanoseconds from a monotonic timer. The timed region is the whole call: initialization, padding, and finalization, and for batched variants also the input transposition and output restoration. Every variant and output length is measured at eight message lengths: 0, 64, 128, 256, 1024, 2048, 16384, and 2^{20} bytes. We also recorded three repetitions and one pass on an independent copy, but we report these only as variability; they do not replace the primary mean. CPU cycle counters were unavailable, because the macOS `kpc` interface returned `NO_PERMISSION`. We do not convert nanoseconds to cycles using a nominal frequency.

The primary machine is an Apple M1 (four performance and four efficiency cores) running macOS 26.6 with Apple clang 21 and the flags `-std=c11 -O3 -fno-lto`. Folders are run one at a time. For the compiler comparison in Section 4 and for the K-SPHINCS⁺ measurements in Section 5, we also use a GNU/Linux AArch64 virtual machine on the same M1 with GCC 11.4 (details in Section 5.4).

3.2 Baseline: A Direct NEON Translation of the Specification

The baseline adopts none of the published techniques. Throughout this paper, a permutation π is written as a pull table, $new[i] = old[\pi(i)]$. In this form, the word permutation of the specification is

$$\sigma = (6, 4, 5, 7, 12, 15, 14, 13, 2, 0, 1, 3, 8, 11, 10, 9). \quad (4)$$

For LSH-256, every destination register then takes all four of its lanes from a single source register. The permutation therefore reduces to two parts. The first is a renaming of which register plays which role; this is free, because the step macro takes its destination registers as parameters and two register sets alternate. The second is one in-register lane shuffle, a single TBL per register.

Since every γ_l is a multiple of 8, γ is also a byte permutation. Two consecutive byte permutations compose into one table:

$$\text{TBL}(\text{TBL}(x, G), P) = \text{TBL}(x, G[P]). \quad (5)$$

Folding γ into σ removes two TBL instructions per step. The rotations by α_j and β_j use SHL+SRI. The result is 25 instructions per step and 898 instructions per 128-byte block for LSH-256.

LSH-512 needs 59 instructions per step and 2,319 per 256-byte block, for three reasons. First, with eight state registers, the permutation actually moves words between registers: four results take one TRN each, and the other four take a copy plus an INS. Second, γ cannot be folded. Third, the 16 message registers do not fit alongside the state, so the expanded sub-messages live in a 256-byte stack scratch area and eight registers are reloaded at every step. For both layouts, we first wrote a Python model with exactly the same register split as the assembly and checked it against the reference digests. The assembly then matched on its first run.

3.3 Permuted-State Techniques (2019 and 2023)

2019: word permutation. Kim et al. [7] observe that LSH can be computed on a permuted state. Let $S[i] = T[P(i)]$. The computation then runs

$$\sigma' = P^{-1} \circ \sigma \circ P, \quad \tau' = P^{-1} \circ \tau \circ P, \quad (6)$$

permutes the step constants, rotation amounts, IV, and message loading once, and applies the inverse permutation to the digest. MsgAdd and Mix commute with any word-wise P that keeps each pair $(l, l+8)$ eight positions apart. The

paper searches $(4!)^2 = 576$ such candidates and ranks them with an x86 shuffle-cost model (TYPE1–TYPE5).

This ranking does not transfer to NEON. A two-register TBL expresses in one instruction a pattern that needs two x86 shuffles. Moreover, once γ is folded, every output of σ' that comes from T_{l+8} costs exactly one TBL, regardless of P . We therefore repeated the search over the same 576 candidates with an explicit NEON instruction-count model:

$$\text{cost}(P) = N_s \cdot \text{cost}(\sigma') + (N_s - 1) \cdot \text{cost}(\tau') + \text{cost}(\text{load}). \quad (7)$$

This model counts permutation instructions only, not the whole compression function. The paper’s choice, $P = (0123, 2013, 0123, 2013)$ in the group notation of [7], costs 182 permutation instructions per block for LSH-256 (204 for the identity) and 284 for LSH-512 (386 for the identity). For LSH-512, the NEON search selects a different P that costs 268.

Before writing any assembly, a Python model built the entire transformed algorithm from P alone. It confirmed that all 576 candidates reproduce the specification’s digests for four messages and six output lengths. We implemented both the paper’s P (`paper`) and the NEON-selected P (`neon_selected`). An identity-permutation control with the same code generation separates the effect of P from the effect of the fused tables.

2023: P-domain placement. Sim’s thesis [8] applies the same framework to ARMv8. It uses a core-specific P ; for LSH-256, the pull table is

$$P_{256} = (3, 0, 2, 1, 7, 4, 5, 6, 11, 8, 10, 9, 15, 12, 13, 14), \quad (8)$$

and for LSH-512, P is an involution ($P = P^{-1}$). The thesis also pre-permutes the step constants and rotation amounts, and uses a message expansion τ' that becomes a uniform rotation by one word within each four-word group. We reproduced this design and measured on the built objects (with `objdump`) what actually changes.

The expansion τ' is a real saving only for LSH-512. It replaces three `EXT` instructions and one copy per eight words with one `EXT` per register. This frees enough registers to keep the expanded sub-messages in registers instead of on the stack, reducing the count from 2,319 to 2,158 instructions per block (−6.9%). For LSH-256, it is neutral (900 vs. 898). On NEON, σ' is never cheaper than σ : LSH-256 needs four TBL instructions either way, and LSH-512 needs 14 instead of 12. We replaced the thesis’s P listing (13 instructions) and γ' listing (19 instructions) with one TBL+EXT per group and by folding into the σ' table. The listing forms are kept as buildable ablations.

3.4 Word-Parallel and Message-Parallel Cores (2023 and 2024)

2023: LSH-256 2-way. The thesis’s 2-way design interleaves two messages lane by lane and performs every permutation at doubleword granularity. Its claim that “the number of additions is halved” refers to the number of `ADD` instructions.

Table 2: Instruction counts measured on the built objects. “Perm.” counts only permutation instructions under the model of Section 3.3; all other rows count the whole compression.

Hash	Quantity	Baseline	Technique	Count
LSH-256	per block	898	2023 <i>P</i> -domain	900
LSH-256	per message	898	2023 2-way	993
LSH-256	Perm. per block	204 (identity)	2019 <i>P</i>	182
LSH-512	per block	2,319	2023 τ'	2,158
LSH-512	Perm. per block	386 (identity)	2019 <i>P</i> / NEON-selected	284 / 268
LSH-512	σ vs. σ'	12	2023 σ'	14
LSH-512	per step	59	2024 <code>cpu_word</code>	49
LSH-512	per expansion	22	2024 <code>cpu_word</code>	14

For LSH-256, this cannot hold even in principle, because the single-message path already fills all four lanes. The 2-way core executes 993 instructions per message, more than the 900 of the single-message path. Whether it is nevertheless faster is decided by the timer (Section 4).

2024: word-parallel LSH-512 and SoA batching. Choi et al. [9] present two designs. The AVX-512 design keeps six logical 512-bit registers ($c_l, c_r, e_l, e_r, o_l, o_r$) live for the whole compression and replaces the eight per-word γ rotations with a single `VPSHUF`. The GPU design assigns one message per thread in a structure-of-arrays (SoA) batch. The paper does not release code, so we re-expressed both designs in NEON rather than transliterating AVX-512 or PTX code.

The word-parallel LSH-512 core (`cpu_word`) keeps the entire message schedule in `v8–v23` for all 28 steps. It fuses γ with the following permutation into four two-register `TBL` instructions. This gives 49 instructions per step and 14 per expansion, fewer than the baseline’s 59 and 22. The rule of thumb “one AVX-512 instruction equals four NEON instructions” fails in both directions. A uniform rotation takes one AVX-512 instruction but eight NEON instructions, $2\times$ worse than the rule. The fused γ and σ take three AVX-512 instructions but eight NEON instructions instead of the predicted 12, $1.5\times$ better than the rule. We deliberately did not build an LSH-256 `cpu_word`. With four words per register, the schedule is resident regardless of intent, so the result would simply report the baseline twice.

The SoA cores (`gpu_x4` for LSH-256 and `gpu_x2` for LSH-512) transpose K blocks into a stack buffer. They then run the compression with every register holding the same word of K messages, and restore the output. The transposition (104 instructions per block for `gpu_x4`) is inside the timed call. Table 2 summarizes the instruction counts of Sections 3.2–3.4.

3.5 Keccak Scheduling Guidance and Errata

2022: Keccak scheduling guidance as an ablation. Becker and Kannwischer [12] optimize Keccak-f[1600] and SPHINCS⁺ on AArch64. Their scalar barrel-shifter technique and scalar/NEON hybrid are out of scope because our cores are NEON-only, and the SHA-3 instructions are excluded as explained in Section 2.2. What remains is general NEON advice: interleaving independent dependency chains, choosing the rotation idiom, and unrolling. We applied this advice one flag at a time to a local copy of the 2023 single-message core (**base**). The shipped delta, **KN_BRAID=1**, groups independent shifts before the dependent inserts. We measured 22 configurations in alternating pairs.

Two of the paper’s central ideas provably do not apply to LSH. The rotation-elimination identity relies on rotation distributing over XOR, which fails for modular addition:

$$\begin{aligned}(a \lll r) \oplus (b \lll r) &= (a \oplus b) \lll r, \\ (a \lll r) \boxplus (b \lll r) &\neq (a \boxplus b) \lll r \quad \text{in general.}\end{aligned}\tag{9}$$

Similarly, **XAR** rotates the XOR of its operands, whereas Mix rotates sums (Eq. (2)). No Mix operation can therefore be fused into **XAR**; at most, it can serve as a bare 64-bit rotation (Section 2.2).

Errata in the published descriptions. While reproducing the three LSH papers and the Keccak paper, we found errors that anyone reimplementing them would encounter. Because they are part of the result, we record them in Table 3; with the two errors in Algorithm 3-3 counted separately, there are ten in total. Each entry, with page numbers, is documented in the `docs/errata.md` file of the corresponding project.

4 LSH Results

4.1 Single-Message Latency

Table 4 reports representative rows of the primary run: every single-message variant for LSH-256-256 and LSH-512-512 at 1,024 and 16,384 bytes, next to the clean C built in the same folder. The full table (six output lengths, eight message lengths, fourteen columns) is in the attached `results/summary.md`.

The picture is the same across all six output lengths and all eight message lengths. Relative to C compiled in the same folder, the single-message NEON cores run at 0.85–0.90 $\times$ for LSH-256 and 0.75–0.89 $\times$ for LSH-512. For LSH-512, the techniques do rank: the 2019 permutation is best (0.85 $\times$), followed by the 2024 word-parallel core (0.80–0.81 $\times$) and then the Keccak-scheduled and baseline cores (0.79–0.80 $\times$). However, all of them lie within 0.1 $\times$ of each other, and none exceeds 1.0.

Table 3: Errata found while reimplementing the published techniques.

Source Issue	
[7]	The prose mixes the push and pull directions of P once.
[7]	“576 candidates” is not the number of word permutations of LSH, but the number left after three constraints: P is word-wise, keeps each pair eight positions apart, and fixes the group order.
[8]	The need to permute the IV is not mentioned; without it, the first block is wrong.
[8]	Algorithm 3-3 (γ') contains two errors.
[8]	In Algorithm 3-4, 9 of the 21 instructions are dead or cancel out.
[8]	All four listings use syntax that the assembler rejects.
[9]	“One AVX-512 instruction $\rightarrow$ four NEON instructions” is off by $2\times$ and $1.5\times$ in opposite directions (Section 3.4).
[12]	The rotation-elimination identity does not hold for modular addition.
[12]	XAR (rotate after XOR) does not match Mix (XOR after rotating a sum).

Controlled ablations. Table 5 summarizes the ablations. Only the folded γ shows a clear effect, and it is also the only ablation with a large difference in instruction count (25 vs. 44 per step). The remaining differences stay within the platform’s noise, given a per-folder repetition interquartile range of 14–19%. Two Keccak configuration means that look large ($1.53\times$ and $1.47\times$) each contain a single outlier pair (0.24, 0.22) among five to seven pairs near 1.0; we keep them in the data but draw no conclusion from them.

Why instruction counts do not become time. Three facts explain the result, and the counts in Section 3 already point to them. First, for each of the eight word pairs, Mix is a serial chain

EOR $\rightarrow$ ADD $\rightarrow$ SHL $\rightarrow$ SRI $\rightarrow$ EOR $\rightarrow$ ADD $\rightarrow$ SHL $\rightarrow$ SRI $\rightarrow$ ADD $\rightarrow$ permutation,

and the pairs share registers. Filling four lanes with one message does not shorten this chain, so each step is latency-bound, and a wide out-of-order core with several vector pipes has little to overlap. The 2023 LSH-512 core removes 6.9% of the instructions with no measurable gain, which is exactly how a latency-bound loop behaves. Second, every rotation by α_j or β_j costs two dependent instructions in Advanced SIMD. Scalar C pays one ROR, or nothing when the compiler folds the rotation into the shifted operand of an EOR or ADD. Third, LSH-512 needs more than 32 registers (8 state, 16 message, a second state bank, temporaries, and tables). Whether it gives up the stack scratch or the second bank, it pays loads or moves at every step, so even the 2024 core, which keeps the schedule resident, loses to C. Meanwhile, the compiler is not idle: Apple clang -O3 produces scalar LSH-256 that hashes 16 KiB in 19.1 μ s, and this core has as many scalar pipes as vector pipes.

Table 4: Single-message latency on Apple M1, macOS, Apple clang 21: arithmetic mean over 10,000 calls in nanoseconds per call, with speed relative to the local clean C in parentheses (> 1 is faster). Every NEON variant is slower than C.

Project	Variant	LSH-256-256		LSH-512-512	
		1,024 B	16,384 B	1,024 B	16,384 B
ref	clean (KISA C)	1,362	19,132	816	10,333
common	baseline	1,546 (0.88 $\times$)	21,953 (0.87 $\times$)	1,024 (0.80 $\times$)	13,136 (0.79 $\times$)
2019	paper	1,536 (0.89 $\times$)	21,830 (0.88 $\times$)	959 (0.85 $\times$)	12,220 (0.85 $\times$)
2019	neon_selected	1,536 (0.89 $\times$)	21,857 (0.88 $\times$)	960 (0.85 $\times$)	12,212 (0.85 $\times$)
2023	single	1,547 (0.88 $\times$)	21,948 (0.87 $\times$)	1,033 (0.79 $\times$)	13,251 (0.78 $\times$)
2024	cpu_word	–	–	1,010 (0.81 $\times$)	12,888 (0.80 $\times$)
keccak	keccak_neon	1,544 (0.88 $\times$)	21,897 (0.87 $\times$)	1,022 (0.80 $\times$)	13,131 (0.79 $\times$)

The compiler decides the single-message case. On a Linux VM on the same hardware with GCC 11.4 (-O3), KISA C hashes a 96-byte message in 202 ns, the NEON baseline in 188 ns (1.08 $\times$), and the 4-message core in 78 ns per message (2.6 $\times$; Section 5.4). The assembly is unchanged; only the C code became slower. Single-message NEON LSH is therefore neither a general gain nor a general loss, but a hedge against the compiler, worth at most about $\pm 10\%$.

4.2 Message-Parallel Throughput

Table 6 compares the native batched cores with clean C called K times on the same K messages. This is the comparison that matters for applications that compute many independent hashes, and it is the only place where the assembly wins.

The 2-way core delivers 1.53–1.66 $\times$ at every LSH-256 length, and the 4-way core delivers 1.95–2.02 $\times$ for LSH-256-256. Two rows need explanation. First, the same `gpu_x4` binary measures 1.0–1.8 $\times$ for LSH-256-224, with no monotonic trend in length. The two output variants differ only in the IV and the truncation, so this is measurement variance on the platform, not an algorithmic difference; we report it as measured rather than choosing the better run. Second, the 2-way SoA core gains nothing for LSH-512. Two 64-bit lanes hold two messages, but the single-message core already uses both lanes for the word pairs of one message, so the transposition is pure overhead. Lane batching for LSH-512 therefore needs $K \geq 4$, which we leave to future work.

Batching wins where word parallelism loses for the reverse of the same dependency argument. With four independent messages in the lanes, the chains in each register are independent by construction, so the core can overlap them and pays the Mix latency once for four hashes. In addition, the word permutation becomes a pure register renaming. The measured gain is 2.0 $\times$ rather than 4 $\times$ because of the transposition, the per-message padding and finalization performed in C, and the four-times-wider loads.

Table 5: Controlled ablations, measured as alternating pairs of 10,000-call runs. Ratios are speeds of the first variant relative to the second (> 1 is faster).

Ablation	Hash	Result	Verdict
2019 P vs. identity P	LSH-512	$1.012\times$ (1 KiB), $0.934\times$ (16 KiB); pairs at 16 KiB: 0.97, 1.27, 0.97	no consistent effect
2023 listing γ' vs. folded γ	LSH-256	$0.54\text{--}0.56\times$ (2,791 vs. 1,555 ns at 1 KiB; all pairs within 1%)	folding is worth $1.8\times$
2023 listing γ' vs. folded γ	LSH-512	$1.00\text{--}1.01\times$ (γ is a separate TBL in both)	no difference
Braid only (<code>keccak_neon</code>) vs. base	both	$0.5\text{--}1.9\%$ faster (1 KiB), $0.1\text{--}0.9\%$ (16 KiB)	within noise
SHL/USHR/ORR rotation; deeper pipelining	both	$1.1\text{--}1.7\%$ slower	within noise

Table 6: Batched throughput on Apple M1 (clang): mean nanoseconds per message, with speed relative to sequential C on the same K messages. The transposition into and out of the SoA layout is inside the timed call.

Output	Core	1,024 B		16,384 B	
		Seq. C	Batched	Seq. C	Batched
LSH-256-256	2023/x2 ($K = 2$)	1,358	856 ($1.59\times$)	19,049	12,223 ($1.56\times$)
LSH-256-256	2024/gpu_x4 ($K = 4$)	1,338	674 ($1.99\times$)	19,032	9,511 ($2.00\times$)
LSH-256-224	2024/gpu_x4 ($K = 4$)	1,341	1,146 ($1.17\times$)	19,046	14,828 ($1.28\times$)
LSH-512-512	2024/gpu_x2 ($K = 2$)	815	820 ($0.99\times$)	10,326	10,484 ($0.98\times$)

5 K-SPHINCS⁺ with Batched LSH

The batching result points to an application. A hash-based signature is a large collection of independent, equal-length hash calls. The *len* WOTS⁺ chains of a leaf advance in lock-step, the $k \cdot 2^a$ FORS leaves are independent, and every level of a Merkle tree is a set of independent node hashes. This is exactly the workload the 4-way LSH-256 core was built for. It is also how Becker and Kannwischer accelerated SPHINCS⁺-SHAKE on AArch64 [12] and how Kölbl accelerated SPHINCS with vectorized hashing [13].

5.1 Instantiating the Hash Layer

We start from the SPHINCS⁺ reference framework [11], specifically its `sha2-avx2` tree. Its generic files (`wots.c`, `fors.c`, `merkle.c`, `utilsx8.c`) already drive an 8-way leaf generator and tree hash through two hash-layer interfaces: `thashx8`

(eight tweakable hashes with eight addresses) and `prf_addrx8`. We replaced the SHA-2 hash layer with an LSH layer and left the generic files untouched. We follow the K-SPHINCS⁺ definition for $n = 32$, extend it to $n \in \{16, 24\}$ by truncation, and use the 32-byte ADRS layout of the SHAKE instance:

$$\begin{aligned} F, H, T_\ell(\text{PK.seed}, \text{ADRS}, M) &= \text{LSH256}(\text{PK.seed} \parallel \text{ADRS} \parallel M)[0..n], \\ \text{PRF}(\text{PK.seed}, \text{SK.seed}, \text{ADRS}) &= \text{LSH256}(\text{PK.seed} \parallel \text{ADRS} \parallel \text{SK.seed})[0..n], \end{aligned} \quad (10)$$

$$\begin{aligned} \text{PRF}_{\text{msg}}(\text{SK.prf}, \text{OptRand}, M) &= \text{LSH256}(\text{SK.prf} \parallel \text{OptRand} \parallel M)[0..n], \\ H_{\text{msg}}(R, \text{PK.seed}, \text{PK.root}, M) &= \text{LSH512}(R \parallel \text{PK.seed} \parallel \text{PK.root} \parallel M)[0..m], \end{aligned}$$

where LSH256 and LSH512 denote LSH-256-256 and LSH-512-512, and $X[0..k]$ denotes the first k bytes of X .

This instantiation differs from Eq. (3) in two places. First, the message digest has length

$$m = \left\lceil \frac{ka}{8} \right\rceil + \left\lceil \frac{h(d-1)}{8d} \right\rceil + \left\lceil \frac{h}{8d} \right\rceil \leq 49 \text{ bytes} \quad (11)$$

for all six parameter sets. For several sets, m exceeds the 32-byte output of LSH-256, so [3] writes H_{msg} as an LSH call with an extended output. We make this extension explicit by using the 512-bit member of the family: a single 64-byte LSH-512 output covers the digest without an MGF. Second, PRF takes PK.seed, following the argument order of the SHAKE instance in round-3 SPHINCS⁺ and FIPS 205. All tweakable hashes use the “simple” (unmasked) construction.

The input of F is $n + 32 + n$ bytes, that is, 64 bytes for $n = 16$ and 96 bytes for $n = 32$. It therefore always fits in one 128-byte LSH-256 block, and each WOTS⁺ chain step is exactly one compression plus padding. H on two nodes takes $n + 32 + 2n$ bytes: one block for $n \leq 24$ and two blocks for $n = 32$.

5.2 Three Backends Behind One Interface

The LSH layer calls three backend functions, `hash256`, `hash256_x4`, and `hash512`, and the backend is selected at build time (Table 7).

The assembly is unchanged. The `.S` files are those measured in Section 4, assembled as ELF instead of Mach-O through the existing macro layer. Because the framework already batches leaves eight at a time, key generation, FORS signing, and hypertree signing all take the batched path. The calls that remain unbatched are PRF_{msg} , H_{msg} , the T_ℓ that compresses each WOTS⁺ public key, and the authentication-path hashes in verification.

5.3 Correctness

For each of the six parameter sets (128f/s, 192f/s, 256f/s) and each backend, we ran the framework’s own tests: `test/spx` (signing, verification, in-place verification, and rejection of a message with a flipped bit) and `test/fors`. All 36 runs (six sets, three backends, two tests) pass. In addition, a deterministic

Table 7: The three hash backends of K-SPHINCS⁺.

Backend	Single-message calls	4-message calls	Purpose
C	KISA reference C	Four sequential C calls	Control: K-SPHINCS ⁺ as the reference code of [3] would compute it
NEON	Single-message cores of <code>neon_common</code>	Four single-message calls	Isolates the effect of single-message assembly inside the scheme
NEON×4	Same as NEON	<code>gpu_x4</code> SoA core of <code>neon_2024</code>	Batched; each <code>thashx8</code> or <code>prf_addrx8</code> call becomes two 4-message batches

driver fixes the key-generation seed and the signing randomness and prints the public key and signature. For every parameter set, the three backends produce byte-identical output. This lifts the per-lane evidence of the LSH projects to the scheme level: the batched core computes exactly the hashes that the reference C computes. No public K-SPHINCS⁺ test vectors exist, so the reference C backend serves as the reference; it is the original KISA code that the LSH projects pin to the public LSH vectors.

5.4 Results

We measured the scheme on the same Apple M1, this time in a GNU/Linux AArch64 guest (Ubuntu 22.04, kernel 6.8, GCC 11.4, `-O3 -std=c99`, four virtual cores), because the SPHINCS⁺ reference framework builds there without modification. This is also the first Linux/ELF run of the LSH assembly, which the macOS measurements could not claim. The guest exposes no hardware cycle counters, so we report wall-clock nanoseconds from `CLOCK_MONOTONIC`. Each operation is measured 20 times per pass. We run three passes with the backend order rotated and report the median of the three pass means. The pass-to-pass spread is below 3% for every batched (NEON×4) row and for most other rows. The widest spread is in the reference C rows (up to 20% for 128f signing), which is why we report the median rather than the best pass. The raw CSV files of all passes are attached.

Table 8 shows key generation, signing, and verification for the six parameter sets and the three backends. Table 9 places the results for the 128-bit parameter sets next to the SHA-256 and SHAKE256 SPHINCS⁺ reference implementations, compiled and measured in the same session on the same machine with the same driver.

We make three observations.

Single-message NEON. Under GCC, the single-message NEON backend alone gains 8–15% for key generation and signing, and 6–13% for verification. This

Table 8: K-SPHINCS⁺ on Apple M1 (Linux guest, GCC 11.4): median of three passes of 20-run means, in milliseconds. “×4” is the NEON×4 backend; “Gain” is its speed relative to the reference C backend.

Set	Key generation				Signing				Verification			
	C	NEON	×4	Gain	C	NEON	×4	Gain	C	NEON	×4	Gain
128f	0.948	0.869	0.371	2.55	22.27	20.41	8.76	2.54	1.561	1.465	0.653	2.39
128s	63.54	55.38	23.64	2.69	459.2	422.7	179.8	2.55	0.543	0.498	0.249	2.18
192f	1.390	1.273	0.558	2.49	36.24	32.98	14.46	2.51	2.197	1.998	0.930	2.36
192s	88.96	82.42	34.92	2.55	801.9	728.6	314.7	2.55	0.750	0.664	0.345	2.17
256f	3.666	3.343	1.441	2.54	77.50	70.24	30.41	2.55	2.177	2.007	1.000	2.18
256s	58.68	53.44	23.45	2.50	772.5	693.6	301.7	2.56	1.104	1.046	0.541	2.04

Table 9: Same machine, session, and driver: SHA-256 and SHAKE256 SPHINCS⁺ reference implementations (**ref**, simple **thash**) versus K-SPHINCS⁺ (LSH) with the reference C and NEON×4 backends, in milliseconds. The last column is the speed of batched K-SPHINCS⁺ relative to each reference.

Set	Operation	SHA-256	SHAKE256	LSH C	LSH NEON×4	vs. SHA-256 / SHAKE
128f	Key generation	1.077	1.172	0.948	0.371	2.90 / 3.16
128f	Signing	25.02	27.63	22.27	8.76	2.86 / 3.15
128f	Verification	1.520	1.618	1.561	0.653	2.33 / 2.48
128s	Key generation	68.45	74.57	63.54	23.64	2.90 / 3.15
128s	Signing	521.0	565.5	459.2	179.8	2.90 / 3.15
128s	Verification	0.503	0.542	0.543	0.249	2.02 / 2.17

agrees with the $1.08\times$ hash-level measurement for this compiler; as Section 4.1 showed, the same gain cannot be expected under clang.

Batching. Batching adds a further $2.3\times$ on top of this, for a total of $2.5\text{--}2.7\times$ over the reference C in key generation and $2.5\text{--}2.6\times$ in signing. The gain is uniform across parameter sets because leaf generation dominates both operations. Verification gains less ($2.0\text{--}2.4\times$). The verifier recomputes one WOTS⁺ public key per layer (batched) and then walks d authentication paths, each a sequence of h/d calls to H (unbatched). The “s” sets have taller trees and therefore gain the least.

Comparison with the standard instances. For the 128-bit parameter sets on this core, K-SPHINCS⁺ with the reference C LSH is already 8–14% faster than SPHINCS⁺-SHA-256 and 17–24% faster than SPHINCS⁺-SHAKE in key generation and signing, and within 8% of both in verification. This is the opposite of the roughly $2\times$ deficit that [3] reported on x86 with a different LSH build. The batched version is $2.9\text{--}3.2\times$ faster than both in key generation and signing, and $2.0\text{--}2.5\times$ faster in verification. We stress that these comparisons are against the portable SPHINCS⁺ reference code. They are not against the NEON-optimized SHAKE implementation of [12], which needs the Armv8.2 SHA-3 extension and

Apple assembler syntax and did not build with our Linux toolchain. A fair comparison with it is left to future work.

Where the remaining time goes. In the batched build, one `thashx8` call takes 0.66 μs , or 82 ns per tweakable hash, compared with 208 ns for the C build and 191 ns for the single-message NEON build. One 8-leaf WOTS⁺ generation for $n = 16$ takes 368 μs , or 46 μs per leaf. Each leaf needs

$$\underbrace{35}_{\text{PRF}} + \underbrace{35 \cdot 15}_{\text{chain steps}} + \underbrace{1}_{T_\ell} = 561 \quad (12)$$

hash calls, so the leaf also costs about 82 ns per hash, including the address handling and buffer assembly done in C. The batched core itself is therefore no longer the bottleneck. The remaining cost lies in three places: building the $n + 32 + n$ -byte input buffer for every call, padding and finalizing the four lanes in C, and splitting each 8-way interface call into two 4-message batches.

6 Conclusion and Limitations

On a modern ARMv8 core, the published single-message SIMD techniques do not make LSH faster than the vendor C code compiled by a good compiler. They only reorder instructions inside a loop that is bound by the latency of a serial add–rotate–XOR chain. What makes LSH fast on NEON is message parallelism, and hash-based signatures are an application with messages to spare. Routing the WOTS⁺, FORS, and Merkle computations of SPHINCS⁺ through the 4-message LSH core yields, to our knowledge, the first optimized K-SPHINCS⁺. It is byte-identical to the reference implementation, signs 2.5–2.6 $\times$ faster, and, for the 128-bit parameter sets, is 2.9–3.2 $\times$ faster than the reference SHA-256 and SHAKE SPHINCS⁺ on the same machine.

Limitations. All measurements were taken single-threaded on one CPU, an Apple M1, with two operating systems and compilers: macOS with clang for the hash cores and a Linux guest with GCC for the scheme. The ranking of the single-message variants may differ on other ARM cores, in particular the in-order Cortex-A cores among the targets of the Keccak guidance [12]. We report no hardware cycle counts, because neither the macOS `kpc` interface nor the guest kernel exposed them. The scheme-level timings use 20 runs and three passes instead of the 10,000-call protocol of the hash measurements (Section 5.4). We deliberately excluded the Armv8.2 SHA-3 instructions: `EOR3` would help `MsgAdd` and the digest but not `Mix`, and `XAR` could serve only as a 64-bit rotation in LSH-512. Finally, our K-SPHINCS⁺ follows the public definition for $n = 32$, but the extension to $n \in \{16, 24\}$ and the use of LSH-512 for H_{msg} are our own choices. No test vectors exist to check them against, and a future K-SPHINCS⁺ standard may choose differently.

Future work. The next steps follow directly from the profile in Section 5.4. An 8-message LSH-256 core would fill the framework’s 8-way interface in one call, including the two-block H of the $n = 32$ sets. A compression function with fused padding for the fixed 64- or 96-byte inputs would remove the C front end that now dominates the remaining time; both steps are independent of any LSH algebra. An LSH-512 batch core with $K \geq 4$ would make lane batching pay off for LSH-512 as well. Beyond this, K-XMSS shares WOTS⁺ and the tree hash and would inherit the same cores. Multithreading across hypertree layers is orthogonal to lane batching and composes with it. The hybrid scalar/NEON scheduling of [12], excluded here, is a natural way to use the scalar pipes that sit idle during batched compression. We release all code, byte-level models, errata, and raw measurements, so that each of these steps can be measured against this baseline rather than against numbers copied from papers.

References

1. National Institute of Standards and Technology, “Stateless hash-based digital signature standard,” Federal Information Processing Standards Publication FIPS 205, NIST, 2024.
2. D.-C. Kim, D. Hong, J.-K. Lee, W.-H. Kim, and D. Kwon, “LSH: A new fast secure hash function family,” in *Information Security and Cryptology – ICISC 2014*, vol. 8949 of *Lecture Notes in Computer Science*, pp. 286–313, Springer, 2015.
3. M. Sim, S. Eum, G. Song, H. Kwon, K. Jang, H. Kim, H. Kim, Y. Yang, W. Kim, W.-K. Lee, and H. Seo, “K-XMSS and K-SPHINCS⁺: Hash based signatures with Korean cryptography algorithms.” Cryptology ePrint Archive, Paper 2022/152, 2022.
4. M. Sim, S. Eum, G. Song, Y. Yang, W. Kim, and H. Seo, “K-XMSS and K-SPHINCS⁺: Enhancing security in next-generation mobile communication and internet systems with hash based signatures using Korean cryptography algorithms,” *Sensors*, vol. 23, no. 17, p. 7558, 2023.
5. A. Hülsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, “XMSS: eXtended Merkle signature scheme,” RFC 8391, Internet Research Task Force (IRTF), 2018.
6. D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, “The SPHINCS⁺ signature framework,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS ’19)*, pp. 2129–2146, ACM, 2019.
7. D. Kim, Y. Jung, Y. Ju, and J. Song, “Fast implementation of LSH with SIMD,” *IEEE Access*, vol. 7, pp. 107016–107024, 2019.
8. M. Sim, “Optimized implementation of the hash function LSH on ARMv8 processors (in Korean),” Master’s thesis, Hansung University, 2023.
9. H. Choi, S. Choi, and S. Seo, “Parallel implementation of lightweight secure hash algorithm on CPU and GPU environments,” *Electronics*, vol. 13, no. 5, p. 896, 2024.
10. Anonymous, “LSH-on-ARMv8: Hand-written AArch64 NEON implementations of LSH with a reproducible measurement harness.” Source code accompanying this submission, 2026.

11. The SPHINCS⁺ team, “SPHINCS⁺ reference and optimized implementations.” <https://github.com/sphincs/sphincsplus>, 2022.
12. H. Becker and M. J. Kannwischer, “Hybrid scalar/vector implementations of Keccak and SPHINCS⁺ on AArch64,” in *Progress in Cryptology – INDOCRYPT 2022*, vol. 13774 of *Lecture Notes in Computer Science*, pp. 272–293, Springer, 2022.
13. S. Kölbl, “Putting wings on SPHINCS,” in *Post-Quantum Cryptography – PQCrypto 2018*, vol. 10786 of *Lecture Notes in Computer Science*, pp. 205–226, Springer, 2018.