

K-LMS: Design, Implementation, and Evaluation of Leighton–Micali Signatures with Korean Cryptographic Primitives

Huiju Kang¹[0009–0008–3198–4598], Yulim Hyoung¹[0009–0009–7530–6497],
Hagyeong Kim¹[0009–0008–5840–5517], Sumin Jeong¹[0009–0007–9643–4872],
Hangsin Cho²[0009–0000–0456–6536], and Hwajeong Seo¹[0000–0003–0069–9061]

¹ Hansung University, Seoul, South Korea
huiju9190@gmail.com yulim4hyoung@gmail.com
kimhaha4420@gmail.com jeong9sumin@gmail.com
hwajeong84@gmail.com

² Independent Researcher, South Korea
wgt0302@naver.com

Abstract. Hash-based signatures are the most conservative post-quantum signature family, and the stateful schemes XMSS and LMS are both standardized by NIST. Prior work has instantiated XMSS and SPHINCS⁺ with Korean cryptographic primitives, but no comparable Korean-primitive study exists for LMS. We fill that gap with K-LMS, an experimental variant that preserves the full LMS structure and domain separation and replaces only the hash. The substituted hashes are the Korean hash function LSH-256, in both its reference and its vectorized implementation, and a Tandem-DM double-block-length construction over the Korean block ciphers CHAM, LEA, and ARIA. A non-invasive bridge over the reference LMS implementation realizes eight interchangeable backends without editing a line of it. We evaluate key generation, signing, and verification latency, key and signature sizes, peak memory, a sweep over tree height and the Winternitz parameter, and implementation complexity. Hash substitution leaves key and signature sizes, and the memory requirement of the LMS data structures, unchanged. We also isolate two measurement pitfalls that can distort such comparisons: a vector dispatch wrapper that re-runs its capability check on every digest, which costs an order of magnitude on our virtualized host, and the choice of software or hardware-accelerated SHA-256 as the baseline, which shifts any LSH vs. SHA-2 conclusion by a factor of about six. Profiling shows that LMS is dominated by 55-byte one-shot hashes (about 94% of all calls), so scheme-level ranking cannot be extrapolated from long-message hash benchmarks. Against a fair software baseline, LSH-256 K-LMS signing is on par with SHA-256 LMS (0.84×), while hardware acceleration retains a ~3.6× advantage where it is available. On the same host and with identical LSH-256 code, K-LMS runs 3.3–3.9× faster than the published K-XMSS implementation, because LMS needs one hash call per Winternitz chain step where XMSS needs three.

Keywords: Hash-based signatures · LMS · LSH · Tandem-DM · CHAM
 · LEA · ARIA · Post-quantum cryptography.

1 Introduction

Hash-based signature (HBS) schemes derive their security solely from the properties of an underlying hash function, making them the most conservative choice among post-quantum signatures. Two stateful HBS schemes are standardized: XMSS (RFC 8391) [1] and the Leighton–Micali Signature scheme LMS (RFC 8554) [2], both approved by NIST SP 800-208 [3], alongside the stateless SPHINCS⁺ [4] standardized as SLH-DSA (FIPS 205) [5].

A natural question for national cryptographic ecosystems is whether domestic primitives can serve as the underlying hash. For Korean cryptography, Sim et al. [6] answered this for XMSS and SPHINCS⁺: their K-XMSS and K-SPHINCS⁺ instantiate the schemes with LSH [7] and with hashes built from the block ciphers CHAM [8] and LEA [9]. However, no corresponding Korean-primitive study exists for LMS, the HBS scheme with the simplest structure, the smallest verifier cost, and wide deployment in firmware signing, the use case for which the NSA’s CNSA 2.0 suite designates LMS and XMSS [10] and in which secure-boot studies have evaluated LMS and SPHINCS⁺ directly [11, 12].

This paper presents K-LMS, closing that gap. We evaluate not only key generation, signing, and verification performance but also public-key/signature sizes, parameter trade-offs, and implementation complexity, against both the standard LMS baseline and the related Korean HBS work. Our contributions:

1. **Design and implementation.** K-LMS preserves RFC 8554 message formats, parameter sets, and domain-separation prefixes, swapping only the hash primitive. A compile-time bridge over the Cisco LMS reference implementation [13] yields eight backends without modifying upstream sources: OpenSSL SHA-256 (SHA-NI), portable C SHA-256, LSH-256 (reference and AVX2), an AVX2 variant that repeats its CPUID check on every digest, and Tandem-DM-256 over CHAM/LEA/ARIA. The bridge adds 557 new lines against an $\sim 11\text{k}$ -line core.
2. **Reproducible measurement methodology.** Single-core pinned benchmarks reporting medians, and a fully scripted pipeline that regenerates every table and figure from a clean build.
3. **Two measurement pitfalls, quantified.** (a) An AVX2 dispatch wrapper that executes CPUID on every digest slows K-LMS by 12–13 $\times$ on our WSL2 host, where CPUID traps to the hypervisor, silently erasing SIMD gains; (b) the SHA-256 baseline choice (SHA-NI vs. portable C) shifts all “LSH vs. SHA-2” conclusions by 6.0–6.2 $\times$.
4. **Workload characterization.** LMS hashing is dominated by 55-byte one-shot calls ($\sim 94\%$), implying that short-message performance, not long-message throughput, predicts HBS behavior, a caveat for interpreting hash microbenchmarks in prior K-XMSS/K-SPHINCS⁺ comparisons; this

quantifies in software the short chain inputs that Wagner et al. noted for hardware accelerators [14].

Because RFC 8554 and NIST SP 800-208 define LMS only for approved hash instantiations (SHA-256 in RFC 8554, to which SP 800-208 and later RFC 9858 [15] add SHAKE256 and 192-bit truncations, all within NIST-approved hash families), any substituted-hash variant, K-LMS included, lies outside those standards and outside KCMVP certification by definition. We therefore scope our claims to feasibility and performance (Sect. 5.4), and view the results as engineering groundwork for any future standardization of Korean-primitive hash-based signatures.

The rest of the paper is organized as follows. Sect. 2 reviews LMS, the Korean primitives, and the prior K-XMSS/K-SPHINCS⁺ work. Sect. 3 describes the K-LMS design, bridge, and correctness strategy. Sect. 4 evaluates performance, sizes, and memory, quantifies the two measurement pitfalls, characterizes the hash workload and parameter trade-offs, and compares K-LMS with the related schemes on the same host. Sect. 5 discusses the fair-baseline methodology, which results transfer across hosts, how to choose a backend, and the limitations of this study, and concludes.

2 Background

2.1 LMS, LM-OTS, and HSS

LMS [2] builds a Merkle tree [16] of height h over 2^h one-time signature (LM-OTS) public keys. LM-OTS signs a message digest using p Winternitz hash chains controlled by parameter $w \in \{1, 2, 4, 8\}$: larger w shortens signatures (fewer chains) but lengthens chains ($2^w - 1$ steps), which beyond $w=2$ increases computation (Sect. 4.4). The Hierarchical Signature System (HSS) wraps L LMS levels; $L = 1$ adds a 4-byte level count to keys and signatures. Every hash call in LMS is domain-separated by a prefix built from the key id I and the leaf index q (the node number r for tree hashes), followed by the Winternitz chain/step indices (i, j) for chain hashes or a typed constant D (D_PBLC, D_MESG, ...) for the others; the chain-step input $I\|q\|i\|j$ with the 32-byte chaining value gives the fixed 55-byte length that dominates our profiles (Sect. 4.4). By contrast, the WOTS⁺ chains [17] of XMSS [18] derive a fresh key and bitmask by PRF before every step, so one chain step costs three hash calls instead of one. With SHA-256, Kampanakis and Fluhrer [19] measured XMSS to be 3.7–5.2× slower than LMS in software, and Campos et al. [20] measured the RFC 8391 variant to be 4.9–6.3× slower on a Cortex-M4. LMS has also been accelerated in hardware [21] and on GPUs [22].

2.2 Korean Cryptographic Primitives

LSH [7] is a hash function family standardized in Korea. It iterates in the Merkle–Damgård style but carries an internal state twice as wide as its output, so internal

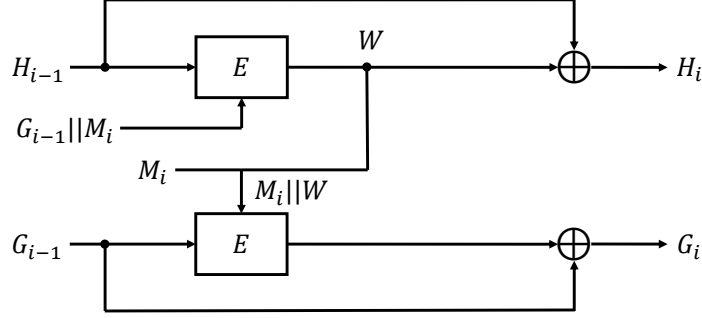


Fig. 1. One compression step of Tandem-DM.

collisions are as hard as colliding a state of twice the output size; its compression function is built from word-wise addition, rotation, and XOR, which map directly onto SIMD lanes [23]; later work implemented LSH with AVX-512 and on GPUs [24, 25]. We use the 256-bit variant in both the reference and the AVX2 implementation distributed by KISA.

CHAM [8, 26], LEA [9], and ARIA [27] are Korean block ciphers. CHAM and LEA are ARX designs, CHAM targeting resource-constrained devices and LEA fast software encryption on general-purpose processors; ARIA is a national standard block cipher and follows a substitution-permutation structure. In the variants we use, all three offer a 128-bit block under a 256-bit key, which is exactly the shape a double-block-length construction needs to turn a block cipher into the 256-bit hash LMS requires. For CHAM we use the revised specification, whose 128/256 variant runs 120 rounds [26].

That construction is Tandem-DM [28] (Fig. 1). A 128-bit block cipher compresses only 128 bits at a time, half of what LMS requires, so Tandem-DM maintains a chaining value of two blocks and absorbs each message block with two chained cipher calls, the second keyed in part by the output of the first; each output is folded back into its half of the chaining value by a feed-forward. Later work proved the construction collision-resistant in the ideal-cipher model up to close to 2^n queries for an n -bit block [29], which is why it remains, alongside constructions such as Hirose’s [30], a standard choice when a $2n$ -bit hash must be built from an n -bit cipher. Two of its properties matter for our measurements: every message block costs two cipher invocations rather than one compression call, and each invocation runs under a freshly derived key, so key schedules cannot be amortized across blocks. Sect. 4.5 shows both costs directly.

2.3 K-XMSS and K-SPHINCS⁺

Sim et al. [6] replaced the hash layers of XMSS and SPHINCS⁺ with LSH and CHAM/LEA-based constructions, reporting reference-C and AVX2 results on

Table 1. HBS schemes instantiated with Korean cryptographic primitives.

Study	Base scheme (state)	Korean hash primitives	Eval. host
K-XMSS [6]	XMSS (stateful)	LSH-256/512; CHAM/LEA (Tandem-DM)	i7-9750H
K-SPHINCS ⁺ [6]	SPHINCS ⁺ (stateless)	LSH-256; CHAM/LEA (Tandem-DM)	i7-9750H
K-LMS (this work)	LMS (stateful)	LSH-256; CHAM/LEA/ARIA (Tandem-DM)	9800X3D

an Intel Core i7-9750H, a Coffee Lake part without SHA extensions. Their study established feasibility and relative performance for those schemes; K-LMS extends the same line of work to LMS, and Sect. 4.6 compares the three HBS families structurally. Absolute cycle counts across the two studies are not comparable (different CPU, OS, compiler), so cross-paper statements are limited to structure, sizes, and relative trends that input lengths explain (Sect. 4.6). Similar substitution work exists for other national standards: Kiktenko et al. instantiated SPHINCS⁺ with the Russian hash function Streebog [31]. Sun et al. likewise instantiated LMS/HSS [32] and XMSS [33] with the Chinese standard hash function SM3 and reported preliminary performance results.

Table 1 places these studies side by side. With K-LMS, all three standardized HBS families now have a Korean-primitive instantiation; relative to the earlier work our study also adds an ARIA backend and an explicit hardware/software SHA-256 baseline separation on a single-core-pinned host.

3 K-LMS Design and Implementation

3.1 Hash-Substitution Model

K-LMS is LMS with the compression primitive replaced. All RFC 8554 serialization, parameter encodings, tree arithmetic, and domain-separation prefixes are untouched; only the map $H : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$ changes. This design isolates a single variable, the hash, so performance differences are attributable to the primitive (the ARIA backend adds library-call overhead, Sect. 4.5), and any future security argument can reuse the LMS structure analysis with assumptions on the substituted hash. We instantiate H two ways: LSH-256 directly, and Tandem-DM-256 over a 128-bit block cipher with a 256-bit key (CHAM-128/256, LEA-256, ARIA-256) with a length-strengthening padding of our own.³ The padding is Merkle–Damgård strengthening over 16-byte blocks: a 0x80 byte, zero bytes up to 8 bytes short of a block boundary, and the 64-bit message length in bits. A 55-byte LMS input thus becomes four blocks, and a 96-byte XMSS-style input would take seven. The padded Tandem-DM digests are deliberately

³ KLMS_TDM_PAD_V1 in the accompanying artifact, <https://github.com/yulim4hyoung/K-LMS>.

Table 2. Evaluated schemes, all at LMS $h=10$, LM-OTS $w=4$, $n=32$, HSS $L=1$.

Scheme ID	Hash / cipher	Role
LMS_SHA256_OPENSSL	SHA-256, OpenSSL (SHA-NI)	HW baseline
LMS_SHA256_PORTABLE	SHA-256, portable C	fair SW baseline
KLMS_LSH256_REF	LSH-256, KISA reference C	main subject
KLMS_LSH256_AVX2	LSH-256, AVX2 (cached CPUID)	main subject (SIMD)
KLMS_LSH256_AVX2_CPUID	LSH-256 AVX2, CPUID per call	pitfall illustration
KLMS_CHAM_TDM256	Tandem-DM / CHAM [6]	extension
KLMS_LEA_TDM256	Tandem-DM / LEA (KISA generic)	extension
KLMS_ARIA_TDM256	Tandem-DM / ARIA (EVP)	extension

not bit-compatible with the unpadded streaming constructions in [6]; a correctness cross-check against the construction of [6] on raw blocks is part of the test suite.

3.2 Bridge Architecture

The Cisco implementation passes every hash call in LMS and LM-OTS through a single dispatch layer, the `hss_hash_*` family, and that layer is where we interpose: each of the eight build targets compiles the unmodified upstream sources against a substitute header that routes hash calls to the LSH backend (three K-LMS targets), the Tandem-DM backend (three targets), or, for the two baselines, OpenSSL’s or the upstream portable SHA-256 behind the same instrumented interface. No upstream file is edited, so all eight binaries in Table 2 share identical tree arithmetic, serialization, and domain separation. The CHAM code is taken from the K-XMSS/K-SPHINCS⁺ repository of [6] and implements the revised 120-round CHAM-128/256 [26]; LSH and LEA use official KISA distributions, and ARIA uses OpenSSL.

3.3 Correctness Strategy

The test suite validates: RFC SHA-256 LMS vectors on the baselines; KISA LSH known-answer tests for reference and AVX2 (including a YMM-register clobber check); cross-backend digest equality between LSH reference and AVX2; Tandem-DM self-tests against the construction of [6] on raw blocks plus streaming consistency; a cross-implementation check that the official KISA LEA-256 core is bit-identical to the LEA-256 core in the repository of [6] over 10^4 random key/block pairs; and sign/verify round trips with negative tests (bit-flipped signatures and messages) for every backend. All tests pass on the evaluation host.

3.4 Implementation Complexity

One comparison axis is how much engineering a Korean-primitive HBS requires. Table 3 reports measured line counts. The hash substitution itself costs 557 lines

Table 3. Implementation size (non-blank lines of C source and headers, measured).

Component	Lines	Role
Cisco <code>hash-sigs</code> core (bundled)	11,097	LMS/HSS logic, unmodified
<code>src/overrides</code> (hash bridges)	318	LSH / TDM redirection
<code>src/tdm</code> (Tandem-DM-256 + adapters)	239	CHAM/LEA/ARIA compression
<code>src/hash</code> (microbenchmark backends)	394	standalone hash timing
<code>src/common</code> (cycle counter, stats)	258	measurement support
Benchmarks / tests	306 / 801	harness

(overrides + Tandem-DM core), about 5 % of the $\sim 11\text{k}$ -line LMS core it retargets, and no upstream modification. The published Korean-primitive code of [6] follows the same pattern: in K-XMSS, every hash call passes through one hash layer (`hash.c/hash.h`, 183 non-blank lines), and in K-SPHINCS⁺ the CHAM hash and tweakable-hash layer spans 140 lines (`hash_CHAM.c`, `thash_CHAM.c`). In all three schemes the substitution is thus confined to a single layer of a few hundred lines, and the engineering effort lies mainly in the primitive implementations rather than in the scheme. Our 557 lines are larger because they serve all eight build targets, including the instrumented SHA-256 baselines, count every hash call, and implement the padded Tandem-DM construction itself.

4 Evaluation

4.1 Experimental Setup

All measurements ran on an AMD Ryzen 7 9800X3D under WSL2 (Linux 6.6), compiled at `-O3` without architecture-specific tuning or link-time optimization,⁴ pinned to a single core. This CPU exposes both AVX2 and the SHA extensions, which is what makes the baseline study of Sect. 4.3 possible. Sample counts are 30 for key generation, 256 for signing, and 10,000 for verification; we report medians. External sources are pinned by commit or archive hash, and the full pipeline regenerates every table and figure from a clean build; the code, raw measurements, and scripts are available in the accompanying artifact at <https://github.com/yulim4hyoung/K-LMS>. Absolute numbers are specific to this host, so we emphasize relative comparisons throughout. Table 4 summarizes the instrument behind each quantity reported in this section.

4.2 Results at the Standard Parameters

Core performance. Table 5 and Fig. 2 give the main results at $h=10$, $w=4$.

We highlight three relative results. First, in the *fair software* comparison, LSH-256 (reference C) is slightly *faster* than portable C SHA-256 for signing

⁴ gcc 15.2.0; `-O3 -DNDEBUG -std=c11`; no `-march=native`, with `-mavx2` only for the AVX2 sources; LTO off; pinned with `taskset -c 0`.

Table 4. Measurement instruments behind the results of this section.

Quantity	Instrument	Samples (keygen / sign / verify) or condition	Reported
Latency (Sect. 4.2, 4.5)	bench_lms : per-call rdtsc/rdtscp with lfence , wall clock CLOCK_MONOTONIC_RAW	30 / 256 / 10,000	median ms and cycles
Dispatch pitfall (Sect. 4.3)	bench_lms , with the AVX2 bridge built to execute CPUID on every digest	30 / 256 / 10,000	median cycles
Peak memory (Sect. 4.2)	GNU time -v , maximum resident set size	one full keygen/sign/verify cycle	MiB
Hash-call profile (Sect. 4.4)	counters in the hash bridge, bucketed by input length	one keygen, $h=10$, $w=4$	calls, bytes
Parameter sweep (Sect. 4.4)	bench_lms	30 / 256 / 10,000 (32 signatures at $h=5$ and $h=15$); $h \in \{5, 10, 15\}$, $w \in \{1, 2, 4, 8\}$	median cycles
Hash-level latency (Sect. 4.6)	bench_hash : one-shot digest over fixed input lengths, timed as above	10,000 per length	median cycles
Same-host K-XMSS / K-SPHINCS ⁺ (Sect. 4.6)	published code [6] built with our flags, timed as above	30 / 256 / 10,000	median ms

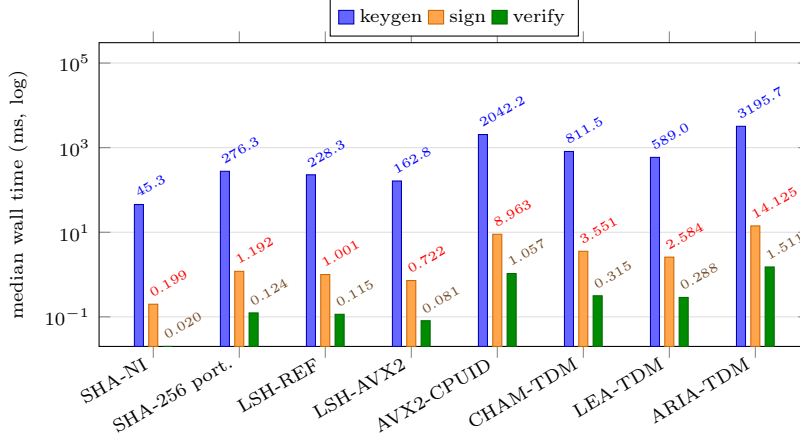
($0.84\times$) and keygen ($0.83\times$): with no hardware acceleration on either side, the Korean hash is fully competitive inside LMS. Second, LSH-256 AVX2 gains $1.40\times$ over the reference for keygen ($1.39\times$ sign, $1.42\times$ verify). Third, against the *hardware* baseline, SHA-NI-accelerated LMS remains $\sim 3.6\times$ faster than the best LSH build for keygen and sign ($4.0\times$ for verify). This gap reflects dedicated hardware support rather than the intrinsic efficiency of either hash design.

Sizes: keys and signatures. Because K-LMS preserves the RFC 8554 formats with $n=32$, sizes equal standard LMS and are identical across all eight backends (Table 6): a hash substitution changes no wire format. The private key is implementation-defined by RFC 8554; the Cisco implementation stores a fixed 64-byte seed-derived private key, plus optional auxiliary Merkle data that trades memory for keygen/sign speed.

Memory usage. We measured peak resident set size (RSS) of each benchmark binary over a complete keygen/sign/verify cycle ($h=10$, $w=4$), pinned to one

Table 5. Median latency (ms) and throughput at $h=10$, $w=4$.

Scheme	keygen (ms)	sign (ms)	verify (ms)	verify ops/s
LMS_SHA256_OPENSSL (SHA-NI)	45.3	0.199	0.020	49,905
LMS_SHA256_PORTABLE	276.3	1.192	0.124	8,088
KLMS_LSH256_REF	228.3	1.001	0.115	8,678
KLMS_LSH256_AVX2	162.8	0.722	0.081	12,356
KLMS_LSH256_AVX2_CPUID	2042.2	8.963	1.057	946

**Fig. 2.** Median keygen/sign/verify latency per scheme ($h=10$, $w=4$); the AVX2-CPUID and Tandem-DM backends are discussed in Sects. 4.3 and 4.5.

core, using GNU `time -v` (Table 7); peak RSS does not depend on the repetition count. The result mirrors the size analysis: under static linking, hash substitution adds no memory. All statically linked backends (LSH-256 reference, AVX2, and the CPUID-per-call variant; CHAM and LEA Tandem-DM; and portable SHA-256) peak at 1.9–2.0 MiB, dominated by the process baseline (code, libc, stack) rather than by any hash state; the $h=10$ tree itself is about 64 KB. The only variation comes from linking `libcrypto`: OpenSSL SHA-256 peaks at 3.6 MiB and ARIA (whose Tandem-DM adapter allocates EVP cipher contexts) at 7.1 MiB. On the memory axis, K-LMS with LSH-256 therefore needs no more memory than the leanest SHA-256 build (1.9 vs. 2.0 MiB).

4.3 Measurement Pitfalls

Pitfall A: CPUID-per-call AVX2 dispatch. CPU-capability dispatch is commonly wrapped around SIMD code. If the wrapper executes the serializing CPUID instruction on every digest, which happens easily when a per-call feature check is not cached, LMS performance collapses (Table 8, Fig. 3): $12.5\times$ slower keygen, $12.4\times$ slower sign, i.e., far below even the scalar reference build.

Table 6. K-LMS sizes at $h=10$, $w=4$, $n=32$ (bytes), and signature size across w and h from the parameter sweep.

Format	Public key	Signature
HSS ($L=1$), as reported by the API	60	2512
Raw LMS (without the 4-byte level count)	56	2508
$w=1$ (HSS)	60	8848
$w=2$ (HSS)	60	4624
$w=8$ (HSS)	60	1456
$h=15$, $w=4$ (HSS)	60	2672

Table 7. Peak resident set size per backend (complete keygen/sign/verify cycle, $h=10$, $w=4$); values rounded to 0.1 MiB.

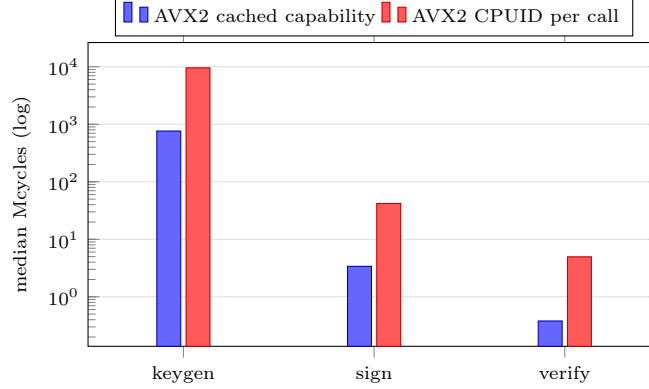
Backend	RSS (MiB)	Note
All LSH-256 builds, LEA-TDM	1.9	static
SHA-256 portable C, CHAM-TDM	2.0	static
SHA-256 OpenSSL (SHA-NI)	3.6	links <code>libcrypto</code>
ARIA-TDM (OpenSSL EVP)	7.1	<code>libcrypto</code> + EVP

Because LMS issues $\sim 10^6$ short hash calls per keygen (Sect. 4.4), per-call fixed costs are amplified more than in long-message settings; benchmark reports of SIMD hashes inside HBS schemes should state their dispatch strategy. Two qualifications apply. The CPUID-per-call variant is our own construction (KLMS_CPUID_EACH_CALL in the bridge), built to expose the pitfall; the dispatcher shipped with the KISA distribution runs its check once and then rebinds its function pointers, so it is not affected. Its magnitude is host-specific. Each uncached check issues four CPUID instructions (three leaf-0 queries from the compiler’s `cpuid.h` helpers and one leaf-7 query), and the check runs once per one-shot digest or context initialization, 1,100,799 times per keygen at $h=10$, $w=4$ (Sect. 4.4). Table 8 therefore implies about 8.0 k extra cycles per check, or about 2.0 k cycles per CPUID (timing the check directly on our host gives 7.8 k ticks, 1.9 k per instruction), a cost explained by WSL2, under which every CPUID traps to the Hyper-V hypervisor. On bare metal, assuming 100–200 cycles per CPUID, the same wrapper would add $0.44\text{--}0.88 \times 10^9$ cycles to keygen, a slowdown of roughly $1.6\text{--}2.2\times$ rather than $12\times$, still a substantial penalty for a scheme that issues $\sim 10^6$ short calls per keygen.

Pitfall B: which SHA-256 baseline? On CPUs with the SHA extensions (SHA-NI) [34], OpenSSL’s SHA-256 is hardware-accelerated; portable C SHA-256 is $6.0\text{--}6.2\times$ slower for LMS operations (Fig. 4). Consequently, “LSH vs. SHA-2” claims are meaningless without stating the baseline: LSH-256 reference *beats* portable SHA-256 ($0.84\times$ sign latency ratio); the AVX2 build still trails SHA-NI by $3.6\times$, the reference build by $5.0\times$. Studies on platforms without SHA

Table 8. Cached-capability vs. CPUID-per-call AVX2 (median cycles).

Operation	AVX2 cached	AVX2 CPUID per call	Slowdown
keygen	763.8×10^6	$9,580.1 \times 10^6$	$12.5\times$
sign	3.39×10^6	42.0×10^6	$12.4\times$
verify	3.80×10^5	4.96×10^6	$13.1\times$

**Fig. 3.** AVX2 dispatch pitfall: cached capability check vs. CPUID per digest.

extensions (as in [6], whose Coffee Lake host lacks them) and studies on SHA-NI hosts are answering different questions; we report both sides explicitly.

4.4 Workload and Parameter Effects

55-byte hashes dominate. Instrumenting the hash layer shows a keygen at $h=10$, $w=4$ performs 1,169,407 hash calls over 62.7 MB, of which 1,097,728 ($\sim 94\%$) are 55-byte one-shot calls, namely Winternitz chain steps and the derivations of the one-time secret-key elements ($1024 \times 67 = 68,608$ of them, with $j=0xFF$), both carrying the $I||q||i||j$ prefix. The distribution is identical across all backends (it is fixed by the scheme). HBS performance is therefore governed by short-input latency, including per-call setup, the regime that motivated dedicated short-input hashes such as Haraka [35]: this explains both the severity of Pitfall A and why long-message hash throughput (where LSH’s 128-byte message block and SIMD-friendly ARX design pay off) does not directly predict scheme-level results. Wagner et al. [14] observe the same 55-byte (LMS) and 96-byte (XMSS) chain inputs from the hardware side, where they make generic hash accelerators communication-bound, and find the hash chains to account for over 80 % of verification latency; our profile quantifies the same effect per call in software.

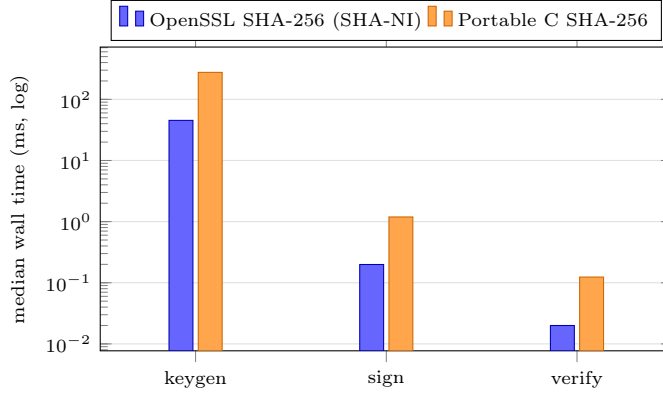


Fig. 4. SHA-NI (OpenSSL) vs. portable C SHA-256 inside LMS.

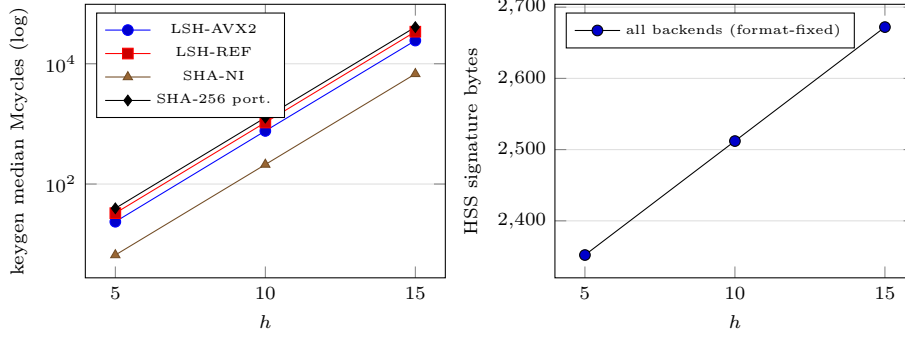


Fig. 5. Tree height h : signature size vs. keygen cost.

Parameter sweep over h and w . Figures 5 and 6 plot the sweep, run with the sample counts of Table 4 (32 signatures at $h=5$, whose tree holds only 32 leaves, and at $h=15$); its $h=10$, $w=4$ point coincides with Table 8. Height h scales keygen exponentially (2^h leaves: 23.6×10^6 cycles at $h=5$ vs. 24.3×10^9 at $h=15$ for LSH-AVX2) while adding only 32 bytes of signature per unit of tree height. The Winternitz parameter trades signature size against computation non-monotonically: $w=8$ cuts the signature by 42 % (1456 B vs. 2512 B at $w=4$) but costs $7.6\times$ the keygen cycles; $w=2$ is the key-generation optimum for LSH-AVX2 in our sweep (457×10^6 cycles), at 4624 B signatures. The relative ordering of backends is stable across the evaluated range.

4.5 Tandem-DM Extensions (CHAM / LEA / ARIA)

Table 9 reports medians for the block-cipher-based backends. All remain well behind the dedicated hashes, at $3.6\text{--}20\times$ the LSH-256 AVX2 latencies and up to $75\times$ versus SHA-NI. The gap follows directly from the construction (Sect. 2.2):

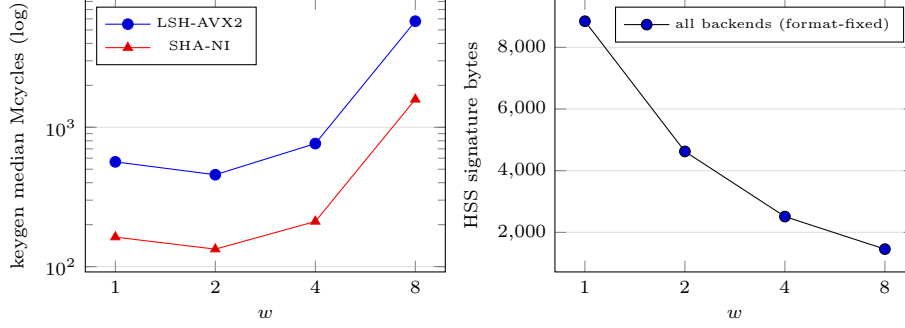


Fig. 6. Winternitz parameter w : signature size vs. keygen cost.

Table 9. Tandem-DM K-LMS, median latency (ms), $h=10$, $w=4$.

Scheme	keygen	sign	verify
KLMS_CHAM_TDM256	811.5	3.551	0.315
KLMS_LEA_TDM256	589.0	2.584	0.288
KLMS_ARIA_TDM256	3195.7	14.125	1.511

two keyed cipher calls per 16-byte block, each with a fresh key schedule, against a single compression call for a dedicated hash, and the 55-byte workload gives those per-call costs nowhere to amortize. ARIA is slowest, most likely because its adapter reaches the cipher through the OpenSSL EVP interface: since Tandem-DM changes the key on every call, each call must re-run `EVP_EncryptInit_ex`, which rebuilds the cipher context and key schedule, so these figures reflect the library interface as much as the ARIA cipher itself. These backends are validated for correctness but not optimized; we include them to establish feasibility. The LEA backend uses the official KISA C source (generic build, bit-identical to the LEA core of [6] per the cross-check of Sect. 3.3); CHAM has no official KISA C release (KISA’s source-code library includes SEED, HIGHT, ARIA, LEA, and LSH but not CHAM), so its code is taken from the repository of [6] and implements the revised 120-round CHAM-128/256 [26].

4.6 Comparison with Related Hash-Based Signatures

Table 10 situates K-LMS structurally against the related schemes at the same hash output length, $n=32$ (256-bit digests). It lists LMS at $h=10$, $w=4$, XMSS at $h=10$, $w=16$, and SPHINCS⁺-256s/f (NIST category 5); “Calls/step” and “Input” give the hash invocations and the input length of one Winternitz chain step (Sect. 4.4), for SPHINCS⁺ in its simple instantiation.⁵ Sizes for

⁵ The robust SPHINCS⁺ variant adds one 58-byte PRF call per step for the bit-mask [4].

XMSS/SPHINCS⁺ follow their specifications; K-XMSS and K-SPHINCS⁺ inherit the sizes of their base schemes, exactly as K-LMS inherits LMS’s (Sect. 4.2). LMS/K-LMS offers the smallest verifier cost and simple structure at the price of state management; SPHINCS⁺-style schemes eliminate state at a $12\times\text{--}20\times$ signature-size cost (SPHINCS⁺-256s to -256f). Within this line of work, K-LMS completes the set: stateful (K-XMSS, K-LMS) and stateless (K-SPHINCS⁺) schemes are now all demonstrated with LSH and block-cipher-based hashes.

To compare performance without crossing hosts, we re-benchmarked the published K-XMSS code [6] on the same machine with the same method (Table 11), building it with the parameter file of the XMSS reference implementation [36]⁶ and our compiler flags. We pair XMSS $w=16$ with LMS $w=4$ (both $\log_2 w = 4$: 67 chains of 15 steps). With the same LSH-256 reference code, K-LMS is $3.3\text{--}3.9\times$ faster; with the Tandem-DM hashes the gap widens to $5.4\text{--}9.1\times$. The structural reason, which the LSH rows isolate because both sides run identical hash code, is that XMSS derives a key and a bitmask by PRF before every chain step, so each step costs three hash calls on 96-byte inputs against one 55-byte call in LMS (Sect. 2.1 and Table 10; the same accounting underlies the SHA-256 gaps reported in [19, 20]), and the per-block rekeying of Tandem-DM makes those longer inputs proportionally more expensive, consistent with software benchmarks of block-cipher-based hashing, where rekeying is far more frequent than in encryption [37]. For the LSH rows this is a property of the schemes, not of the primitives, and it agrees with Sect. 4.4: the cost of short inputs per chain step decides HBS performance. The Tandem-DM rows compare different implementations as well: K-LMS uses its length-padded construction (Sect. 3.1) and, for LEA, the KISA core, whereas K-XMSS uses the unpadded construction and cipher code of [6]. Part of their larger gap, including the wider spread for LEA ($7.7\text{--}9.1\times$) than for CHAM ($5.4\text{--}7.4\times$), therefore reflects implementation rather than scheme structure. A direct hash-level measurement on the evaluation host confirms this block-boundary effect (the one-shot digest microbenchmark of Table 4). Both portable SHA-256 and LSH-256 process a 55-byte LMS chain-step input in a single compression block (1128 vs. 940 cycles, a $0.83\times$ ratio consistent in direction with the scheme-level LMS sign ratio of $0.84\times$), but a 96-byte XMSS chain-step input crosses SHA-256’s 64-byte block boundary into two blocks (2068 cycles) while LSH-256’s 128-byte block still absorbs it in one (940 cycles, the same as for 55 bytes), a $2.2\times$ gap. Much of the LSH advantage reported for K-XMSS is therefore an interaction between input length and block boundary rather than a property of the hash itself, the same lesson as Sect. 4.4.

The published K-SPHINCS⁺ code provides only the CHAM-based hash layer, and its hardcoded sizes (pk 64 B, sig 49,856 B) are those of SPHINCS⁺-256f (Table 10), the parameter set evaluated in [6]; we build it with that parameter set, so it runs at the same hash output length, $n=32$, as K-LMS. With the CHAM backend, K-SPHINCS⁺-256f takes 11.3 ms for key generation, 239.1 ms

⁶ The published repository does not include this file; we took it from [36] and added the three Korean-hash identifiers that its hash layer expects. All other code is used as published.

Table 10. Structural comparison of HBS schemes (SHA-256, $n=32$; sizes in bytes).

Scheme	Stateful	pk	sig	Calls/step	Input (B)	Korean variant
LMS (RFC 8554)	yes	56	2508	1	55	K-LMS (this work)
XMSS (RFC 8391)	yes	68	2500	3	96	K-XMSS [6]
SPHINCS ⁺ -256s	no	64	29,792	1	118	K-SPHINCS ⁺ [6]
SPHINCS ⁺ -256f	no	64	49,856	1	118	K-SPHINCS ⁺ [6]

Table 11. Same-host median latency (ms): published K-XMSS [6] ($h=10$, $w=16$) vs. K-LMS ($h=10$, $w=4$).

Hash	Op	K-XMSS	K-LMS	XMSS/LMS
LSH-256 (reference C)	keygen	770.4	228.3	3.4×
	sign	3.920	1.001	3.9×
	verify	0.377	0.115	3.3×
CHAM Tandem-DM	keygen	4379.8	811.5	5.4×
	sign	22.823	3.551	6.4×
	verify	2.320	0.315	7.4×
LEA Tandem-DM	keygen	4531.5	589.0	7.7×
	sign	23.636	2.584	9.1×
	verify	2.341	0.288	8.1×

for signing, and 6.13 ms for verification. The stateless trade-off is visible on one machine: key generation is about $72\times$ cheaper than K-LMS with the same hash, because only the top subtree is built, while signing costs $67\times$ more, verification $19.5\times$ more, and the signature is $19.9\times$ larger.

5 Discussion and Conclusion

5.1 Fair-Baseline Methodology

Our central methodological claim is that HBS hash-substitution studies must control three variables that are easy to conflate: hardware hash acceleration (Pitfall B), SIMD dispatch cost (Pitfall A), and input-length profile (Sect. 4.4). All three materially change conclusions, yet the closest prior study [6] reports none of them: its LSH-versus-SHA-2 comparison runs on a host without SHA extensions and states neither the SHA-256 implementation used, the AVX2 dispatch strategy, nor the input-length profile.

5.2 What Transfers Across Hosts

Not all of our results depend equally on the evaluation host. Key and signature sizes and the hash-call profile are fixed by the scheme and hold on any platform (Sects. 4.2 and 4.4). The block-boundary effect of Sect. 4.6 follows from input lengths and block sizes alone. We expect the relative ranking of the software

backends, and the gap between the Tandem-DM backends and the dedicated hashes, to carry over as well, since both follow from the work per hash call. By contrast, the magnitude of Pitfall A depends on the cost of `CPUID`, about 2.0k cycles under WSL2 and estimated at a $1.6\text{--}2.2\times$ slowdown on bare metal, and the SHA-NI advantage of Pitfall B exists only where the extension does.

5.3 Choosing a Backend

For firmware signing, where verification dominates, LSH-256 K-LMS matches or slightly beats portable SHA-256 on hosts without SHA extensions ($0.93\times$ the verification latency for the reference build, Table 5); where SHA-NI is available, SHA-256 LMS remains $4.0\times$ faster to verify than the best LSH build. Among the block-cipher hashes, LEA-TDM is the fastest but still $3.6\times$ behind LSH-256 AVX2, so it may be attractive where a single cipher core must serve both encryption and hashing, e.g., on constrained devices that already carry LEA or CHAM.

5.4 Scope and Limitations

Replacing the hash removes RFC 8554/SP 800-208 compliance: K-LMS is an uncertified, non-standardized research artifact for feasibility and performance study, and no KCMVP claim is made or implied. Beyond this scope, three limitations apply. (i) Results come from one x86 host (AMD Zen 5) under WSL2; only relative statements should be reused, and replication on bare-metal Intel and ARM hosts is left to future work (Sect. 5.5). (ii) The CHAM implementation is taken from the repository of [6] (no official KISA C release exists), and the Tandem-DM backends are feasibility-oriented rather than optimized implementations. (iii) Security of LMS with LSH or Tandem-DM hashes is assumed, not proven: existing LMS proofs model the hash [38], or its Merkle–Damgård compression function [39], as a random oracle, and they do not automatically transfer to LSH-256 or to a Tandem-DM compression function built from a block cipher.

5.5 Conclusion and Future Work

We designed, implemented, and evaluated K-LMS, the first LMS instantiation with Korean cryptographic primitives, completing the K-XMSS/K-SPHINCS⁺ research line for the remaining standardized HBS scheme. With a fair software baseline, LSH-256 K-LMS matches SHA-256 LMS performance while retaining identical key and signature sizes and requiring only 557 lines of integration code; hardware SHA-NI remains decisive where present. On the same host, K-LMS runs $3.3\text{--}3.9\times$ faster than the published K-XMSS code with identical LSH-256 code, a consequence of LMS’s single hash call per chain step ($5.4\text{--}9.1\times$ with the Tandem-DM hashes, whose implementations also differ), and the published K-SPHINCS⁺ code (CHAM, 256f), at the same hash output length, exhibits the stateless performance and size trade-off directly (Sect. 4.6). Beyond the schemes

themselves, we quantified two measurement pitfalls and a workload property that should inform any future HBS primitive-substitution study. Future work: replication on bare-metal Intel and ARM hosts; ARM NEON and AVX-512 ports of LSH within K-LMS (a standalone AVX-512 implementation of LSH-256 already reports about $1.16\times$ over the KISA AVX2 code [24]); and a formal security analysis of LMS under the substituted hash assumptions.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: extended Merkle signature scheme. RFC 8391 (2018). <https://doi.org/10.17487/RFC8391>
2. McGrew, D., Curcio, M., Fluhrer, S.: Leighton–Micali hash-based signatures. RFC 8554 (2019). <https://doi.org/10.17487/RFC8554>
3. Cooper, D., Apon, D., Dang, Q., Davidson, M., Dworkin, M., Miller, C.: Recommendation for stateful hash-based signature schemes. NIST Special Publication 800-208 (2020). <https://doi.org/10.6028/NIST.SP.800-208>
4. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS⁺ signature framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019). pp. 2129–2146. ACM (2019). <https://doi.org/10.1145/3319535.3363229>
5. National Institute of Standards and Technology: Stateless hash-based digital signature standard. FIPS PUB 205 (2024). <https://doi.org/10.6028/NIST.FIPS.205>
6. Sim, M., Eum, S., Song, G., Yang, Y., Kim, W., Seo, H.: K-XMSS and K-SPHINCS⁺: Enhancing security in next-generation mobile communication and internet systems with hash based signatures using Korean cryptography algorithms. *Sensors* **23**(17), 7558 (2023). <https://doi.org/10.3390/s23177558>
7. Kim, D.C., Hong, D., Lee, J.K., Kim, W.H., Kwon, D.: LSH: A new fast secure hash function family. In: Lee, J., Kim, J. (eds.) *Information Security and Cryptology – ICISC 2014*. LNCS, vol. 8949, pp. 286–313. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15943-0_18
8. Koo, B., Roh, D., Kim, H., Jung, Y., Lee, D.G., Kwon, D.: CHAM: A family of lightweight block ciphers for resource-constrained devices. In: Kim, H., Kim, D.C. (eds.) *Information Security and Cryptology – ICISC 2017*. LNCS, vol. 10779, pp. 3–25. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78556-1_1
9. Hong, D., Lee, J.K., Kim, D.C., Kwon, D., Ryu, K.H., Lee, D.G.: LEA: A 128-bit block cipher for fast encryption on common processors. In: Kim, Y., Lee, H., Perrig, A. (eds.) *Information Security Applications – WISA 2013*. LNCS, vol. 8267, pp. 3–27. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05149-9_1
10. National Security Agency: Commercial National Security Algorithm Suite 2.0. Cybersecurity Advisory (2022), updated 2024
11. Kampanakis, P., Panburana, P., Curcio, M., Shroff, C., Alam, M.M.: Post-quantum LMS and SPHINCS⁺ hash-based signatures for UEFI secure boot. *Cryptology ePrint Archive*, Report 2021/041 (2021), <https://eprint.iacr.org/2021/041>
12. Wagner, A., Oberhansl, F., Schink, M.: To be, or not to be stateful: Post-quantum secure boot using hash-based signatures. In: Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security (ASHES 2022). pp. 85–94. ACM (2022). <https://doi.org/10.1145/3560834.3563831>

13. Cisco Systems: hash-sigs: An implementation of LMS/HSS (RFC 8554). <https://github.com/cisco/hash-sigs>, last accessed 2026/08/30
14. Wagner, A., Oberhansl, F., Schink, M.: Extended version—to be, or not to be stateful: Post-quantum secure boot using hash-based signatures. *Journal of Cryptographic Engineering* **14**, 631–648 (2024). <https://doi.org/10.1007/s13389-024-00362-4>
15. Fluhrer, S., Dang, Q.: Additional parameter sets for HSS/LMS hash-based signatures. RFC 9858 (2025). <https://doi.org/10.17487/RFC9858>
16. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) *Advances in Cryptology – CRYPTO ’89*. LNCS, vol. 435, pp. 218–238. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_21
17. Hülsing, A.: W-OTS⁺ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) *Progress in Cryptology – AFRICACRYPT 2013*. LNCS, vol. 7918, pp. 173–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38553-7_10
18. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS – a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.Y. (ed.) *Post-Quantum Cryptography – PQCrypto 2011*. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_8
19. Kampanakis, P., Fluhrer, S.: LMS vs XMSS: Comparison of two hash-based signature standards. *Cryptology ePrint Archive, Report 2017/349* (2017), <https://eprint.iacr.org/2017/349>
20. Campos, F., Kohlstadt, T., Reith, S., Stöttinger, M.: LMS vs XMSS: Comparison of stateful hash-based signature schemes on ARM Cortex-M4. In: Nitaj, A., Youssef, A. (eds.) *Progress in Cryptology – AFRICACRYPT 2020*. LNCS, vol. 12174, pp. 258–277. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51938-4_13
21. Thoma, J.P., Hartlief, D., Güneysu, T.: Agile acceleration of stateful hash-based signatures in hardware. *ACM Transactions on Embedded Computing Systems* **23**(2), 29:1–29:29 (2024). <https://doi.org/10.1145/3567426>
22. Wang, Z., Dong, X., Kang, Y., Chen, H., Wang, Q.: An example of parallel Merkle tree traversal: Post-quantum Leighton–Micali signature on the GPU. *ACM Transactions on Architecture and Code Optimization* **21**(3), 44:1–44:25 (2024). <https://doi.org/10.1145/3659209>
23. Kim, D., Jung, Y.B., Ju, Y., Song, J.: Fast implementation of LSH with SIMD. *IEEE Access* **7**, 107016–107024 (2019). <https://doi.org/10.1109/ACCESS.2019.2932129>
24. Lee, S.W., Song, M.H., Kim, H.G., Kim, U.J., Eum, S.W., Seo, H.J.: Advanced vector extensions 512 acceleration of LSH and LEA-GCM. *Applied Sciences* **16**(14), 6846 (2026). <https://doi.org/10.3390/app16146846>
25. Choi, H., Choi, S., Seo, S.: Parallel implementation of lightweight secure hash algorithm on CPU and GPU environments. *Electronics* **13**(5), 896 (2024). <https://doi.org/10.3390/electronics13050896>
26. Roh, D., Koo, B., Jung, Y., Jeong, I.W., Lee, D.G., Kwon, D., Kim, W.H.: Revised version of block cipher CHAM. In: Seo, J.H. (ed.) *Information Security and Cryptology – ICISC 2019*. LNCS, vol. 11975, pp. 1–19. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40921-0_1
27. Kwon, D., Kim, J., Park, S., Sung, S.H., Sohn, Y., Song, J.H., Yeom, Y., Yoon, E.J., Lee, S., Lee, J., Chee, S., Han, D., Hong, J.: New block cipher: ARIA. In: Lim, J.I., Lee, D.H. (eds.) *Information Security and Cryptology – ICISC 2003*. LNCS, vol. 2971, pp. 432–445. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24691-6_32

28. Lai, X., Massey, J.L.: Hash functions based on block ciphers. In: Rueppel, R.A. (ed.) *Advances in Cryptology – EUROCRYPT 1992*. LNCS, vol. 658, pp. 55–70. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-47555-9_5
29. Lee, J., Stam, M., Steinberger, J.: The collision security of Tandem-DM in the ideal cipher model. In: Rogaway, P. (ed.) *Advances in Cryptology – CRYPTO 2011*. LNCS, vol. 6841, pp. 561–577. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_32
30. Hirose, S.: Some plausible constructions of double-block-length hash functions. In: Robshaw, M. (ed.) *Fast Software Encryption – FSE 2006*. LNCS, vol. 4047, pp. 210–225. Springer, Heidelberg (2006). https://doi.org/10.1007/11799313_14
31. Kiktenko, E.O., Bulychev, A.A., Karagodin, P.A., Pozhar, N.O., Anufriev, M.N., Fedorov, A.K.: SPHINCS⁺ post-quantum digital signature scheme with Streebog hash function. In: *AIP Conference Proceedings*. vol. 2241, p. 020014. AIP Publishing (2020). <https://doi.org/10.1063/5.0011441>, preprint: arXiv:1904.06525
32. Sun, S., Liu, T., Guan, Z., He, Y., Jing, J., Hu, L., Zhang, Z., Yan, H.: LMS-SM3 and HSS-SM3: Instantiating hash-based post-quantum signature schemes with SM3. *Cryptology ePrint Archive*, Paper 2022/1491 (2022), <https://eprint.iacr.org/2022/1491>
33. Sun, S., Liu, T., Guan, Z., He, Y., Jing, J., Hu, L., Zhang, Z., Yan, H.: XMSS-SM3 and MT-XMSS-SM3: Instantiating extended Merkle signature schemes with SM3. *Cryptology ePrint Archive*, Paper 2022/1552 (2022), <https://eprint.iacr.org/2022/1552>
34. Gulley, S., Gopal, V., Yap, K., Feghali, W., Guilford, J., Wolrich, G.: Intel SHA extensions: New instructions supporting the secure hash algorithm on Intel architecture processors. Intel Corporation, White Paper (2013)
35. Kölbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 – efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology* **2016**(2), 1–29 (2016). <https://doi.org/10.13154/tosc.v2016.i2.1-29>
36. Hülsing, A., Rijneveld, J.: xmss-reference: XMSS reference implementation (RFC 8391). <https://github.com/XMSS/xmss-reference>, last accessed 2026/08/30
37. Bos, J.W., Özen, O., Stam, M.: Efficient hashing using the AES instruction set. In: Preneel, B., Takagi, T. (eds.) *Cryptographic Hardware and Embedded Systems – CHES 2011*. LNCS, vol. 6917, pp. 507–522. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_33
38. Katz, J.: Analysis of a proposed hash-based signature standard. In: Chen, L., McGrew, D., Mitchell, C. (eds.) *Security Standardisation Research – SSR 2016*. LNCS, vol. 10074, pp. 261–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49100-4_12
39. Fluhrer, S.: Further analysis of a proposed hash-based signature standard. *Cryptology ePrint Archive*, Report 2017/553 (2017), <https://eprint.iacr.org/2017/553>