Accelerating FAEST Signatures on ARM: NEON SIMD AES and Parallel VOLE Optimization

Seung-Won Lee[1 [0009−0009−9583−1387]], Ha-Gyeong Kim[1 [0009−0008−5840−5517]], Min-Ho Song[1 [0009−0007−6277−0069]], Si-Woo Eum[1 [0000−0002−9583−5427]], and Hwa-Jeong Seo[1 [0000−0003−0069−9061]]

Department of Information Computer Engineering,
Hansung University, Seoul, 02876 Korea
dkajdfhd1@gmail.com, kimhaha4420@gmail.com, smino0906@gmail.com, shuraatum@gmail.com, hwajeong84@gmail.com

Abstract. FAEST is a post-quantum digital signature candidate whose performance is dominated by repeated AES-CTR-based PRG calls in the VOLE-in-the-Head phase, yet its reference implementation provides no FAEST-specialized ARM NEON acceleration path. We present an ARM-oriented optimization that accelerates this bottleneck using general-purpose NEON SIMD instructions without relying on ARMv8 Crypto Extensions. The proposed implementation combines a register-resident 256-byte S-box with TBL/TBX-based four-stage SubBytes, 4-way and 8-way parallel AES block processing, a fixed-size PRG path specialized for the FAEST tree structure, and pthread-based batch-level parallelization of independent VOLE tasks. Evaluated on all 12 parameter sets of FAEST v2 on Raspberry Pi 4 and Apple M2, the combined optimization achieves speedups of up to 136.9× and 330.1×, respectively, over the pure-C reference. On RPi4, the single-thread NEON implementation outperforms OpenSSL's software AES, and on M2, the full NEON-plus-pthread configuration outperforms the best available reference configuration, including hardware-accelerated OpenSSL, across all tested parameters.

Keywords: Post-Quantum Cryptography · VOLE-in-the-Head · Register-resident AES · ARM NEON SIMD · Performance optimization

1 Introduction

The advancement of quantum computing poses a serious threat to conventional public-key cryptosystems. RSA and Elliptic Curve Cryptography (ECC) base their security on the computational hardness of integer factorization and the discrete logarithm problem, respectively. However, it has been proven that Shor's algorithm can solve these problems in polynomial time in a quantum computing environment [20]. In response, NIST has pursued the Post-Quantum Cryptography (PQC) standardization project and, in July 2022, selected three digital signature schemes – CRYSTALS-Dilithium, Falcon, and SPHINCS+ – for standardization, which were subsequently published as FIPS standards (ML-DSA,

FN-DSA, SLH-DSA) beginning in August 2024 [2]. However, concerns have been raised that the majority of the selected algorithms are lattice-based and that excessive dependence on a specific hard problem may undermine cryptographic diversity. To mitigate this potential risk, NIST issued a call for additional non-lattice-based digital signature algorithms in September 2022 [17] and is currently evaluating 14 Round 2 candidates selected in October 2024.

The FAEST signature algorithm is one of the 14 Round 2 candidates in this additional standardization process, featuring a structure that uses AES [18] as a One-Way Function (OWF) and employs zero-knowledge proofs based on VOLE-in-the-Head [6]. FAEST is designed using only symmetric-key cryptographic primitives with proven security, such as AES and SHA-3. The public key consists of a plaintext–ciphertext pair (x, y), and the secret key k is an AES key that satisfies $E_k(x) = y$. According to the FAEST v2 specification, FAEST-128s has a signature size of approximately 4,506 B[4], which is approximately a 42% reduction compared to 7,856 B for SPHINCS+-128s [8]. However, FAEST has a trade-off in that achieving a small signature size requires a substantial amount of computation. In particular, the GGM-tree-based seed expansion and repeated PRG calls in the VOLE-in-the-Head phase act as major performance bottlenecks. In this process, thousands of PRG calls are made, and internally tens of thousands of AES-CTR block encryptions are performed. The AES-CTR mode [10] guarantees the independence between blocks, making it suitable for SIMD parallelization. However, the FAEST reference implementation [16] provides architecture-specific SIMD acceleration (`AES-NI`, `AVX2`) only for x86/x64 environments and otherwise delegates AES processing to an external library such as OpenSSL [19]. Because OpenSSL's general-purpose AES routines are not tailored to the frequent short-output PRG calls characteristic of FAEST's tree structure, significant performance overhead remains on ARM platforms, particularly those without ARMv8 Crypto Extensions. The scope of this paper is to fill this gap by providing a lightweight, FAEST-specialized ARM NEON optimization.

The ARMv8-A architecture supports efficient data-level parallel operations through the NEON SIMD extension. In particular, NEON's Table Lookup (TBL) instruction and Table Lookup with Extension (TBX) instruction can be used effectively to parallelize AES S-box substitution and byte-permutation operations.

The main contributions of this work are as follows.

1. We analyze the PRG-computation bottleneck that occurs in the VOLE-in-the-Head phase of FAEST, and we propose a method to parallelize the computation using pure NEON SIMD in ARM environments where Crypto Extensions are absent.
2. We vectorize AES-based PRG and OWF calls in multi-block granularity (4-way, 8-way), and we design an implementation strategy that minimizes memory accesses via register-resident S-box lookups while guaranteeing constant-time operation.

3. We design pthread-based task-level parallelization that leverages the independence of each instance in the VOLE generation process, and we combine it with SIMD to maximize performance.
4. We implement the proposed method on RPi4 (Cortex-A72) without ARMv8 Crypto Extensions and on an Apple M2 environment that supports the extension, and we empirically evaluate performance improvements and generality depending on the presence or absence of hardware acceleration through comparisons with the existing reference (pure-C implementation) and OpenSSL.

2 Background

2.1 FAEST Algorithm Structure

FAEST is a post-quantum digital signature scheme based on the *MPC-in-the-Head* paradigm, where the correctness of an AES computation is proven in zero knowledge. It combines VOLE-in-the-Head techniques with AES circuit verification to construct an efficient signature scheme. The algorithm consists of three main procedures: key generation, signing, and verification.

KeyGen: A random AES key k is selected, and the ciphertext $y = E_k(x)$ for a plaintext x is computed to form the public key in the form (x, y). The secret key is k [4].

Sign: First, a VOLE-in-the-Head-based commitment is generated [6]. Multiple seeds are expanded through a GGM tree [14], and a random vector is generated by applying a PRG to each expanded seed. The generated randomness forms VOLE correlations through linear operations, and then the integrity of AES-circuit computations is proven through a constraint-verification algorithm within the VOLE-in-the-Head framework. Finally, the signer applies the Fiat–Shamir transform [12] to the hash chain and multi-challenge generation structure to construct a deterministic non-interactive signature [4].

Verify: The validity of the zero-knowledge proof is checked using the VOLE opening values and constraint-verification values provided by the signer. In this process, PRG-based seed reconstruction and VOLE-vector reconstruction are also performed, and the consistency of the commitment is finally checked through constraint verification [4].

In addition to the standard AES construction, a representative variant is FAEST-EM, which applies the Even–Mansour construction [11]. FAEST-EM replaces the AES key schedule with a fixed permutation, thereby reducing the number of multiplicative gates in the zero-knowledge proof circuit from 200 to 160 for AES-128 [5]. Because in the VOLE-in-the-Head structure, the proof size and computation cost are proportional to the number of multiplicative gates, this brought the effect of reducing the signature size to about 3.9 KB for FAEST-EM-128s [4].

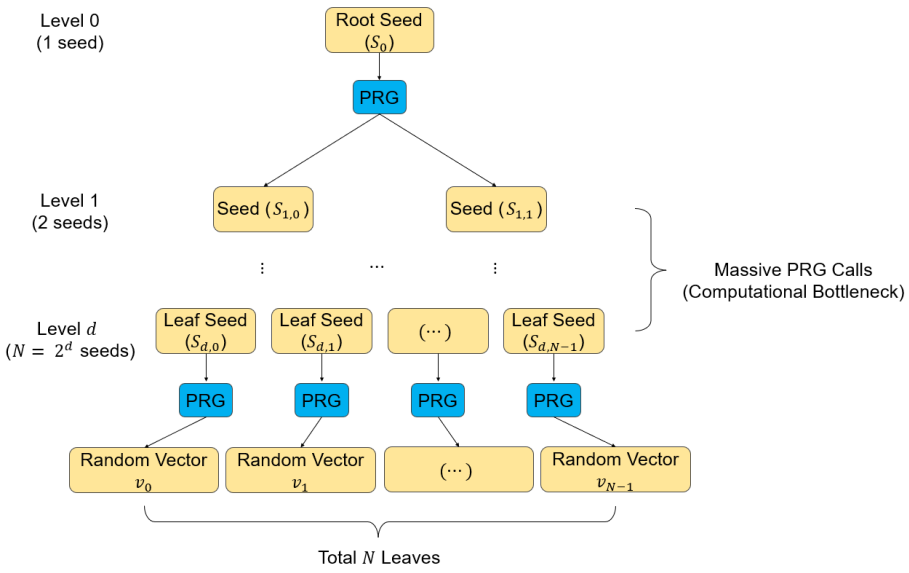

Fig. 1. Structure of GGM Tree Expansion and PRG Calls in FAEST

2.2 Role of VOLE and PRG

The VOLE generation process plays a key role in FAEST signing and verification. The leaf seeds generated through the GGM tree are input to a PRG and expanded into random vectors of length ℓ.

These vectors are converted into vectors u, v through XOR-based binary-tree reduction, and they are combined with the global challenge scalar $\Delta \in F$ held by the verifier to form the VOLE relation as in (1) [4].

$$q = u \cdot \Delta + v \tag{1}$$

Although the computational cost of the binary-tree reduction itself is negligible, the PRG that generates randomness functions as a base computation that constructs VOLE correlations, beyond being a simple random generator. In FAEST, the PRG is implemented by running AES-128, AES-192, or AES-256 in CTR mode depending on the security level, and as a result the PRG acts as a major bottleneck in the overall performance of FAEST [4].

2.3 ARM NEON SIMD

ARM NEON is a SIMD extension technology included in the ARMv8-A architecture [3], providing 32 vector registers `v0-v31` with a width of 128 bits. This corresponds to a total of 512 bytes of register space, and each register can fully store and process one 16-byte AES state matrix. Among NEON instructions, `TBL`/`TBX` are byte-wise substitution instructions that use up to four consecutive

registers (64 bytes) as a lookup table. The `TBL` instruction returns 0 for indices that exceed the reference range, whereas the `TBX` instruction has the property of keeping the existing value of the destination register when the index is out of range.

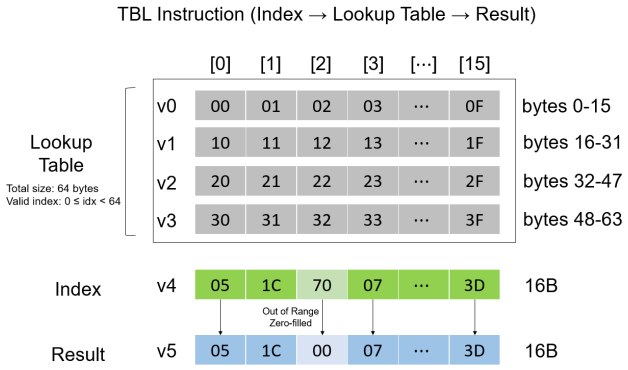

Fig. 2. `TBL` instruction (zero-fill)

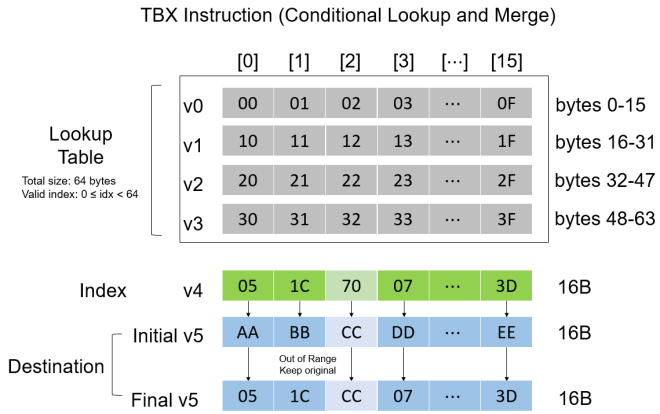

Fig. 3. `TBX` instruction (conditional merge behavior)

By combining the out-of-range → 0 property of `TBL` and the keep property of `TBX` step by step, multi-stage substitution (one `TBL` call followed by three chained `TBX` calls) is possible, covering the entire 256-byte S-box even though a single instruction has a 64-byte limit.

2.4 pthread-Based Thread Parallelization

A multi-core processor contains multiple compute cores on a single chip, so the overall computation time can be reduced by assigning independent tasks to each core and executing them simultaneously. This approach, which decomposes work into independent execution units, distributes them across multiple threads, and processes them in parallel, is called task-level parallelism. POSIX threads (pthreads) [9] are an IEEE-standard thread library for implementing such a multi-thread environment, providing thread creation, execution, and synchronization functions so that multiple tasks can be processed simultaneously within a single program.

2.5 Related Work

Many studies have been conducted on NEON-based optimization to improve AES throughput in ARM environments. Hamburg first presented an AES implementation technique using vector substitution instructions [15], laying the foundation for later use of NEON's TBL/TBX instructions. Fujii et al. proposed a TBL-based AES implementation for ARMv8 processors without Crypto Extensions, processing four blocks in parallel with an optimized MixColumns formula and achieving a 5% speedup over the Linux kernel's NEON AES routine [13]. Adomnicai et al. proposed a bitslicing-based Fixslicing technique [1], and Wang et al. proposed a vector implementation that applies masking to defend against side-channel attacks [21]. However, these studies target general-purpose bulk encryption of large continuous data; none addresses the workload pattern specific to FAEST, where thousands of short-output PRG calls with frequent seed changes dominate the computation.

At the algorithm level, Baum et al. proposed an optimization that removes unnecessary PRG calls in the GGM-tree seed-expansion process, reducing the computation in the VOLE generation phase, and also improved the efficiency of OWF proofs within the VOLE-in-the-Head paradigm [5]. On the proof-system side, FAEST builds on the VOLE-in-the-Head paradigm [6], which lets the signer alone simulate VOLE correlations to produce a publicly verifiable signature, and Yang et al. proposed QuickSilver [22], an efficient VOLE-based zero-knowledge proof system for circuits over any field. These contributions advance protocol design and theoretical efficiency, but do not address platform-specific software optimization; in particular, ARM-oriented acceleration of the short-output AES-CTR PRG workloads that dominate FAEST's runtime has not been explored.

3 Optimized Implementation

The existing implementation of the FAEST reference [16] processes AES computations sequentially on a single-block basis in the VOLE-in-the-Head process, and therefore it does not utilize the advantages of SIMD parallel processing at all. To resolve this bottleneck, this work proposes and implements the following three optimization techniques.

3.1 SIMD AES Acceleration Based on ARM NEON Instructions

The FAEST reference [16] supports only pure software-based AES computation and AES acceleration using OpenSSL in ARM environments. In this work, instead of relying on ARMv8 Crypto Extensions, we propose an AES optimization technique that processes multiple AES blocks in parallel using general-purpose ARM NEON SIMD instructions.

The core of the proposed implementation is the "register-resident S-box" technique, which keeps the entire 256-byte S-box resident in 16 128-bit ARM64 vector registers (v16–v31). This fundamentally eliminates memory accesses that occur during S-box substitution and performs immediate computations within CPU registers.

To use the loaded S-box efficiently, we apply the `TBL`/`TBX` instructions described in Section 2.3. As shown in Figure 2 and Figure 3, the S-box data are divided into four segments of 64 bytes and stored in registers, and the substitution values corresponding to each segment are extracted step by step. The SubBytes macro that implements this consists of four stages in total.

In the first stage, the `TBL` instruction references values in the 0x00–0x3F index range, and initializes the result to 0 for indices outside the range. Then, using the `TBX` instruction, the 0x40–0x7F, 0x80–0xBF, and 0xC0–0xFF ranges are searched sequentially. From stages 2 to 4, the state is overwritten with the S-box value only when the input index is within the range of that stage, and bytes outside the range keep the result from the previous stage. In this way, the entire 256-byte S-box table is searched through four stages. With this technique, the substitution of an entire 16-byte block is parallelized in only a few cycles, greatly improving the computational efficiency of the SubBytes step.

Algorithm 1 NEON SubBytes – 4-Stage Table Lookup

Require: state: 16-byte AES state register
Require: S_0–S_3: S-box register groups, each covering 64 bytes
Ensure: state: State with S-box substitution applied

1: $idx_1 \leftarrow \text{state} - \texttt{0x40}$
2: $\text{state} \leftarrow \mathbf{TBL}(S_0,\ \text{state})$ ▷ 0x00–0x3F: match → value, else → 0
3: $idx_2 \leftarrow idx_1 - \texttt{0x40}$
4: $\text{state} \leftarrow \mathbf{TBX}(S_1,\ idx_1)$ ▷ 0x40–0x7F: match → overwrite, else → keep
5: $idx_3 \leftarrow idx_2 - \texttt{0x40}$
6: $\text{state} \leftarrow \mathbf{TBX}(S_2,\ idx_2)$ ▷ 0x80–0xBF: match → overwrite, else → keep
7: $\text{state} \leftarrow \mathbf{TBX}(S_3,\ idx_3)$ ▷ 0xC0–0xFF: match → overwrite, else → keep

The ShiftRows step is an operation that cyclically shifts each row of the 4×4 state array to the left. In this implementation, after storing the byte-permutation indices in a register in advance, we rearrange byte positions using a single `TBL` instruction.

Because this approach completes the row shift of the entire 16 bytes with only one instruction execution, it efficiently replaces the many byte-move operations required in the existing FAEST reference [16].

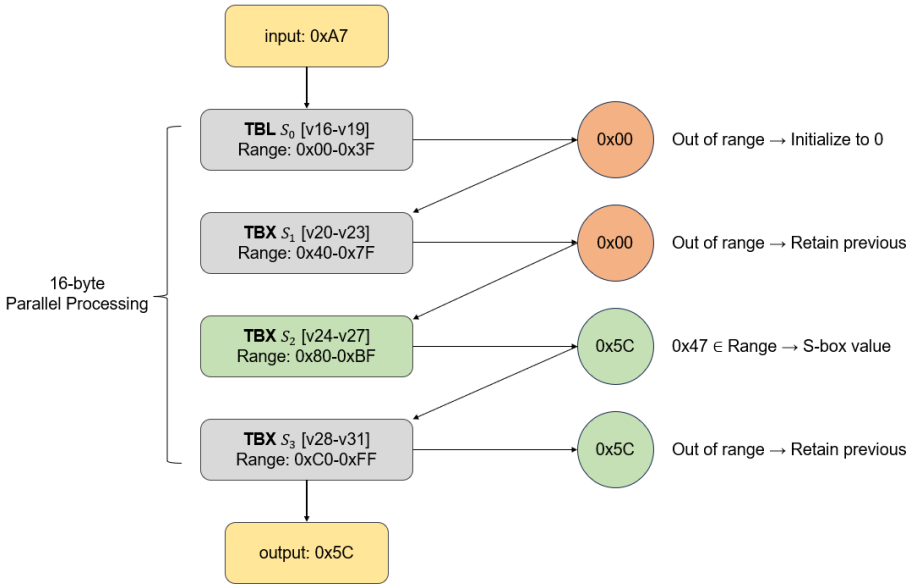

Fig. 4. Example execution of Algorithm 1 with input 0xA7

```
// ShiftRows index: {0,5,10,15, 4,9,14,3, 8,13,2,7, 12,1,6,11}
.macro ShiftRows4 reg
tbl    \reg.16b, {\reg.16b}, v8.16b
.endm
```

Listing 1.1. Efficient ShiftRows via Single TBL Instruction

MixColumns is the column-mixing operation of AES, performing polynomial multiplication over the Galois field GF(2^8). The `xtime` ($\times 2$) function, which is the core of this operation, shifts the input left by 1 bit and then checks the most significant bit (MSB); if the MSB is 1, it performs an XOR operation with the constant `0x1b`, corresponding to the irreducible polynomial ($x^8+x^4+x^3+x+1$), to perform modular reduction. In the NEON implementation, the result of cyclically rotating bytes within a column by one position is first stored in a temporary vector. Then, for the `xtime` computation, the value is shifted left by 1 bit, and the MSB is expanded to `0x00` or `0xFF` using a signed arithmetic shift instruction to generate a conditional mask. This mask is ANDed with `0x1b` to generate the final mask, and then XORed with the shifted value to complete conditional reduction. This bit-masking method enables perfect 16-byte parallel processing without branches, maximizing computation speed and pipeline efficiency.

Furthermore, to maximize utilization of the CPU pipeline, we introduced 4-way and 8-way parallel structures. This is a technique that encrypts 4 or 8 AES blocks simultaneously with a single function call. In the 4-way implementation, four state blocks are loaded into vector registers, and the S-box data are kept

```
sshr.16b v15, v14, #7    // Extract MSB: 0x00 or 0xFF
shl.16b v14, v14, #1     // 1-bit left shift
and.16b v15, v15, v10    // Mask: 0x00 or 0x1b
eor.16b v14, v14, v15    // Conditional reduction
```

Listing 1.2. NEON xtime – Branchless Conditional Reduction

Table 1. Comparison of NEON register allocation for 4-way and 8-way parallelism

4-way Parallelization		**8-way Parallelization**	
Registers	**Usage**	**Registers**	**Usage**
v0–v3	Store state value	v0–v7	Store state values
v4–v7, v12–v15	Temporary registers	v8–v11	Store index/constant
v8–v11	Store index/constant	v12–v15	Temporary registers
v16–v31	Store the S-box table	v16–v31	Store the S-box table

resident in 16 registers and shared by all state blocks. During AddRoundKey, the round key is loaded only once into a register, and XOR operations are performed sequentially with the four state registers, minimizing the number of memory loads. In the 8-way implementation, eight state blocks to be encrypted are placed first, and the remaining registers are filled with constants and control vectors, temporary space to store intermediate values of computations, and the 256-byte S-box data. The detailed register allocation is summarized in Table 1.

Because such a multi-block implementation has no data dependencies between blocks, the out-of-order superscalar pipeline can fully exploit the instruction-level parallelism (ILP) exposed by the independent block operations, greatly improving computational throughput.

In addition, this implementation strengthens constant-time properties in the encryption path by keeping the entire S-box data resident in registers, thereby eliminating secret-dependent memory-access patterns during block encryption that could be exploited through cache-timing side channels [7]. Although the existing FAEST reference [16] already employs mask-based field arithmetic without input-dependent branches, its byte-by-byte S-box computation still involves repeated memory loads whose access pattern may vary with cache state. In contrast, the proposed register-resident approach confines all substitution operations within the CPU register file, removing data-dependent cache behavior. The `xtime` computation in MixColumns also uses bit-masking logic instead of conditional reduction, further strengthening uniform execution timing.

3.2 Applying PRG and OWF Vectorization

We designed a structure that integrates the SIMD AES engine optimized above into the OWF and PRG layers of the FAEST algorithm. The FAEST algorithm provides three security-level parameters of 128-bit, 192-bit, and 256-bit, and uses an OWF that has a different output size depending on the level. Because the

size of a single AES block is 128 bits, for the 192-bit and 256-bit security levels at least two blocks must be encrypted to produce the output. The FAEST reference [16] processes these two blocks sequentially internally, causing a computation bottleneck, but in this work we introduced a 4-way parallel structure to improve it. After generating, in parallel, the original input block and a modified block obtained by XORing `0x01` into the first byte of that input, we place these two blocks into individual slots of a 4-way vector register and call a single parallel AES function to encrypt them simultaneously.

For the 128-bit security level, only one block of output is required, so only one of the four 4-way slots is filled with valid data and the rest are zero-padded. For the Even–Mansour (EM) construction, we used a separate processing path that swaps the roles of input and key, using the original input as the AES key and the key as the plaintext, and additionally XORing the key into the encryption result. This parallel processing strategy effectively offsets encryption-cost computations that increase proportionally as the security level increases.

To resolve the PRG-computation bottleneck analyzed in Section 2.2, we applied the following optimization techniques.

(1) Bulk generation of 8-way IVs We pre-store eight consecutive initialization vectors, consisting of counter values (IVs) that increase by 1 per block, in an aligned memory buffer. These eight prepared initialization vectors are immediately passed to the 8-way parallel AES function, completing AES-CTR encryption of eight blocks with a single call. Compared to the approach that increments the counter individually and encrypts sequentially per block, this minimizes the number of memory loads and maximizes the pipeline data throughput.

(2) Applying a fixed-length PRG function This is the part that produced the largest performance improvement in our implementation, where we designed a fixed-length PRG function specialized for the FAEST tree structure. A general PRG function typically uses logic that dynamically calculates the number of encryption iterations according to the requested length, but in the tree-expansion step of FAEST the required PRG output size is always constant as 2λ or 4λ. Using this property, we implemented a dedicated function that removes loops and applies the loop-unrolling technique, listing the computations directly in order. Because the output-size computation and repetition-control logic are omitted, pipeline overhead due to conditional branches is fundamentally eliminated. In addition, because the number of required blocks is also fixed depending on the security-level parameter λ, appropriate slot placement becomes possible.

As shown in Table 2, when outputting 4λ (128 bytes) in an AES-256 ($\lambda = 256$) setting, eight blocks are required, so the 8-way slots are fully utilized without waste. In contrast, for the 4λ (96 bytes) output of AES-192, six blocks are needed, so we use six out of the eight 8-way slots and zero-pad the remaining two slots to maintain reusability of the parallel function.

Algorithm 2 fixed-length PRG – 4λ Output ($\lambda = 256$)

Require: key, iv, tweak
Ensure: out: $4\lambda = 128$-byte output
1: $rk \leftarrow \text{AesKeygen}_{256}(\text{key})$
2: $iv' \leftarrow iv + \text{tweak}$
3: ivs[0..7] $\leftarrow$ Prepare8IVs(iv') ▷ Generate 8 IVs at once without loop
4: out $\leftarrow$ AES_8way(ivs, rk) ▷ Single call produces 128-byte output

Table 2. fixed-length PRG Slot Usage by Security Level

Security Level	Output Size	Blocks	Parallel Config	Slot Usage
128-bit	$4\lambda = 64$B	4 blocks	4-way	4/4 (100%)
192-bit	$4\lambda = 96$B	6 blocks	8-way	6/8 (75%)
256-bit	$4\lambda = 128$B	8 blocks	8-way	8/8 (100%)

(3) Hierarchical throughput optimization For ranges where variable and large amounts of randomness must be generated, we implemented a hierarchical fallback structure that automatically selects the optimal parallel width according to the requested data size. If at least 128 bytes of randomness remain to be generated, 8-way parallel processing is performed with the highest priority; if at least 64 bytes remain, 4-way processing is performed; and for remaining blocks smaller than that, only some slots of 4-way are used with padding. This structure maximizes the overall data throughput by selecting the maximum parallel width available in each cycle of the loop.

3.3 pthread-Based Parallelization of VOLE Computations

The VOLE commitment process of the FAEST signature algorithm internally generates τ independent VOLE instances. Each instance takes different unique seed data as input, and records results in an independent buffer whose region does not overlap in memory. Shared resources such as initialization vectors or parameter structures are referenced only as read-only, so there are no data dependencies between instances. The VOLE reconstruction process in the verification phase is also strictly independent of the rounds in terms of seed reconstruction and VOLE transformation. In this implementation, we actively leverage this algorithmic independence and design and apply multi-thread, task-level parallelization based on POSIX threads (pthread).

(1) Precomputation of input/output offsets In the sequential processing structure of the existing FAEST reference [16], the location of the memory start of the next round is determined by cumulatively adding the output size of the previous round. For example, if the output of the first round occupies five columns, the second round starts writing from the sixth column. This design induces strong ordering dependencies between rounds, because the next round can

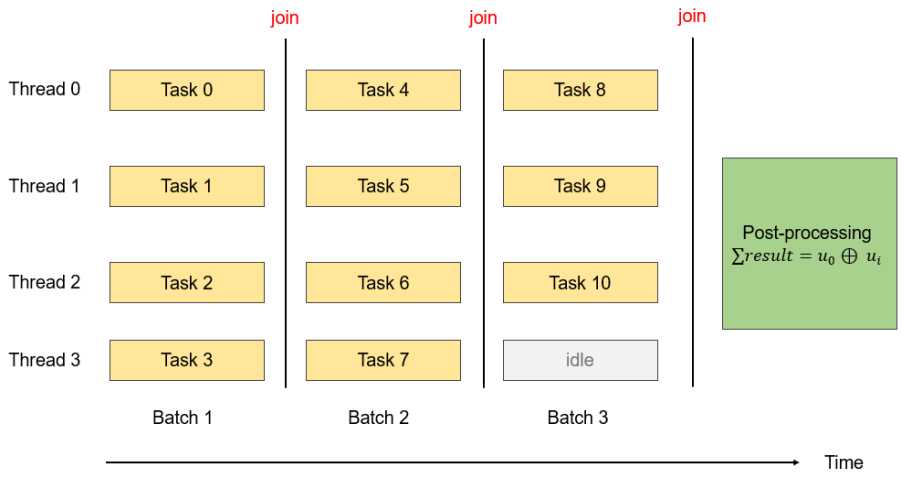

Fig. 5. Batch-Parallel VOLE Execution Timeline ($\tau = 11$, $N = 4$)

determine its write location only after the computation of the previous round has fully finished.

To remove this dependency in a multi-thread environment, we introduced logic that precomputes the seed and output offsets of each round based only on parameter information before performing the actual computation. Because the tree depth and the number of nodes in a given round are values determined by parameters, all offsets can be fixed mathematically without directly executing the VOLE transformation. As a result, each task thread can immediately access its assigned memory address and perform computation in parallel without waiting for the completion of other threads.

(2) Batch-level parallel scheduling To control resource waste caused when the CPU temporarily stops the currently running thread and switches to another thread, we do not create τ tasks simultaneously; instead, we split and execute them in batches according to the number of available cores. For example, if there are 11 instances ($\tau = 11$) and the number of logical threads is set to 4, tasks 0–3 are executed in the first batch and we wait until their parallel processing is completed; then tasks 4–7 are scheduled in the second batch. If the number of remaining tasks in the last batch is smaller than the configured thread count, the number of created threads is reduced dynamically.

Because the `pthread_join` operation performed at the end of each batch period implicitly acts as a synchronization barrier, safe execution ordering is guaranteed without complex lock structures such as mutexes. In addition, we minimize the overall memory footprint by allocating and reusing thread-handle resources only up to the batch size.

(3) Fundamental prevention of data races When multiple threads run simultaneously, a data race can occur if they access the same memory address at the same time, causing the result value to differ depending on the execution order. In this implementation, without an additional software lock structure, we prevent this by establishing three principles at the design level.

1. All global data shared by all threads, such as initialization vectors and parameter structures, are accessed strictly as read-only.
2. Because the output region of each thread is separated into an independent memory space that does not overlap with others by offsets computed in advance, no conflicts occur even if multiple threads perform write operations simultaneously.
3. In the parallel implementation of the verification process, the temporary buffer space used for seed reconstruction is allocated separately for the number of threads, physically blocking memory sharing between threads.

(4) Sequential post-processing and resource reclamation After execution of all batches is completed, the main thread computes the final offset by iterating over and summing the result lengths returned from the τ task structures. Because this step is simple integer addition that reads independent memory regions, the overhead is negligible compared to the overall parallel-processing gains. After computation and post-processing are finished, dynamically allocated resources such as the task array, handle objects, and offset-array resources are released at once to prevent memory leaks.

4 Performance

4.1 Experimental Setup

To objectively evaluate the performance of the proposed implementation, performance measurements were conducted on two types of ARM platforms that differ in whether ARMv8 Crypto Extensions hardware acceleration is supported. Table 3 summarizes the key hardware and software specifications of the platforms used in this experiment.

To individually analyze the contribution of each optimization technique to performance improvement, four build configurations were set up as shown in Table 4. `ref` is a pure-C-based AES implementation from the FAEST reference [16] with no hardware acceleration applied, serving as the baseline. `opt_ref` is a build that integrates OpenSSL's [19] AES-CTR implementation with the FAEST reference [16]. In the M2 environment, it internally utilizes AES-CE hardware instructions, whereas in the RPi4 environment, which does not support these, it relies on OpenSSL's software AES operations. `neon` is a build that applies the NEON TBL/TBX-based SIMD AES acceleration and PRG/OWF vectorization proposed in this work, and `neon-pthread` is the final integrated configuration that additionally activates pthread-based VOLE task parallelization.

Table 3. Experimental platforms

	RPi4	**Apple M2**
Model	RPi4 Model B	MacBook Air (M2, 2022)
Processor	ARM Cortex-A72	Apple M2
Architecture	ARMv8-A	ARMv8.5-A
Number of Cores	4	8 (4P + 4E)
Clock	1.5 GHz	3.49 GHz (P-core)
Crypto Extensions	**Not Supported**	Supported
L1 Cache (P-core)	48 kB I + 32 kB D	192 kB I + 128 kB D
L2 Cache	1 MB (Shared)	16 MB (4P Shared)
RAM	4 GB LPDDR4	8 GB LPDDR5
OS	Debian GNU/Linux 13	macOS 26.2
Compiler	GCC 14.2.0	Apple Clang 17.0.0

Table 4. Build configurations

Build	AES Implementation	NEON SIMD	pthread	Thread Count
`ref`	Pure C (Software)	–	–	1
`opt_ref`	OpenSSL AES-CTR	–	–	1
`neon*`	NEON TBL/TBX	4/8-way	–	1
`neon-pthread*` (`neon-pt`)	NEON TBL/TBX	4/8-way	O	4

All benchmarks were performed using the Catch2 v3.4.0 framework with 100 repeated measurements for each of the KeyGen, Sign, and Verify operations, and performance was compared based on the computed average values. The evaluation was conducted across 12 parameter sets defined in the FAEST v2 [4] specification (128s/f, 192s/f, 256s/f, and EM variants), totaling 96 benchmark configurations (12 parameters × 4 builds × 2 platforms).

4.2 Benchmark Results

KeyGen Performance The KeyGen operation is limited to a single AES encryption, so it completes quickly in microseconds (μs) across all build configurations. For the standard FAEST configuration, the key-generation time of the `ref` build was 59–147 μs on RPi4, but was reduced to 2.7–3.8 μs when the `neon` build was applied, showing approximately a 20–38× improvement in computational efficiency. In contrast, for the FAEST-EM 192/256-bit configuration, the key-generation process operates as a fixed permutation algorithm and is unaffected by AES acceleration; as a result, consistent performance of approximately 99 μs (192-bit) and 165 μs (256-bit) was measured on RPi4 across all builds.

Signing Performance Table 5 and Table 6 present the signing operation performance for both the standard FAEST and FAEST-EM configurations. The signing operation involves tens of thousands of PRG calls during the VOLE generation process, accounting for the highest latency among all cryptographic

Table 5. Sign performance on RPi4 (ms) with `ref` → `neon-pt` speedup (×)

Param	ref	opt_ref	neon	neon-pt	×
FAEST					
128s	21,787	489	400	259	84.1
128f	3,001	96	85	65	46.2
192s	79,388	1,779	1,567	1,111	71.5
192f	10,091	391	361	306	33.0
256s	175,694	3,268	3,005	2,048	85.8
256f	20,776	657	634	513	40.5
FAEST-EM					
EM-128s	16,507	381	304	190	86.9
EM-128f	2,278	79	69	53	43.0
EM-192s	70,067	1,251	1,064	599	117.0
EM-192f	7,429	259	238	191	38.9
EM-256s	146,980	2,228	1,959	1,074	136.9
EM-256f	17,411	523	497	394	44.2

Table 6. Sign performance on Apple M2 (ms) with `ref` → `neon-pt` speedup (×)

Param	ref	opt_ref	neon	neon-pt	×
FAEST					
128s	5,430	43.9	43.9	34.6	156.9
128f	748	16.0	16.0	14.9	50.2
192s	19,779	120.8	133.1	107.2	184.5
192f	2,485	49.9	51.6	48.6	51.1
256s	43,280	186.6	235.8	172.8	250.5
256f	5,103	70.3	76.2	69.2	73.7
FAEST-EM					
EM-128s	4,104	32.7	30.2	23.3	176.1
EM-128f	568	13.4	13.0	12.3	46.2
EM-192s	17,412	83.4	92.3	64.4	270.4
EM-192f	1,860	34.9	36.0	33.6	55.4
EM-256s	36,479	128.0	163.0	110.5	330.1
EM-256f	4,297	61.2	65.2	59.7	72.0

operations; therefore, it is the phase where the effects of the optimization techniques proposed in this work are most prominently observed. Figure 6 and Figure 7 visualize each result on a log scale to intuitively show the performance differences between builds.

On the RPi4 platform, the signing time of the baseline `ref` build reaches approximately 175.7 s for FAEST-256s, and approximately 147.0 s for FAEST-EM-256s. This is the result of the enormous accumulated computation cost incurred by software-based AES operations performed through block-level serial processing. In contrast, when `neon-pthread` is applied, the same parameters complete

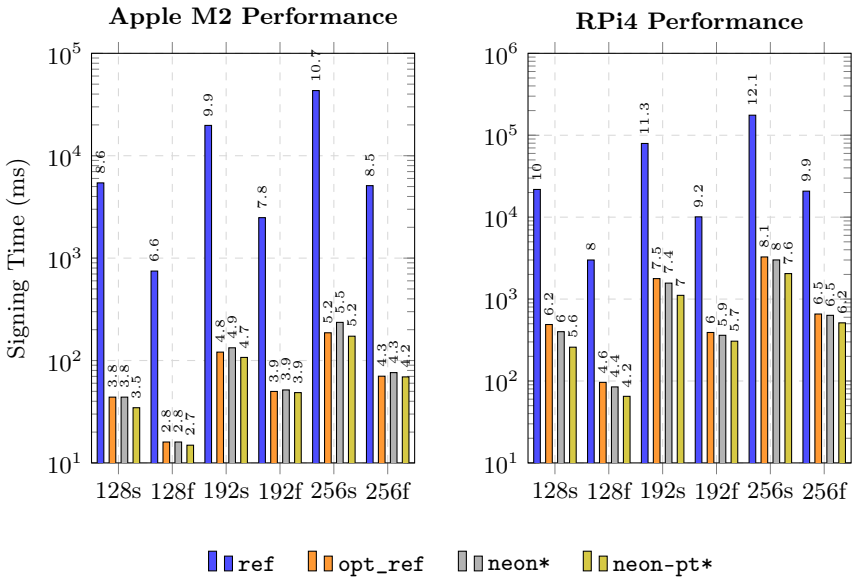

Fig. 6. Experimental results of FAEST signing performance across security levels.

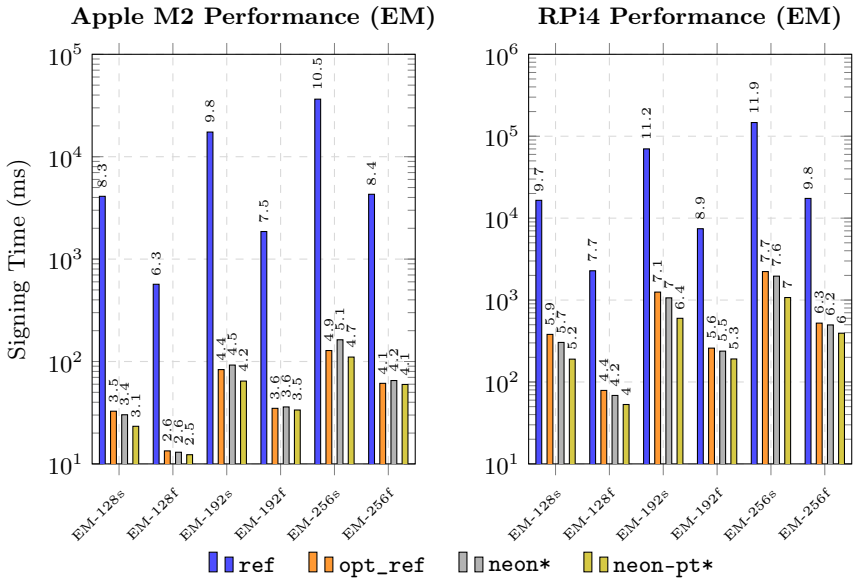

Fig. 7. Experimental results of FAEST-EM signing performance.

in 2,048 ms and 1,074 ms, respectively, achieving speedups of 85.8× and 136.9× compared to `ref`.

In the Apple M2 environment, due to the advantages of the processor's high operating clock and wide out-of-order execution pipeline, the absolute execution time of `ref` was measured to be approximately 4.0× shorter compared to RPi4.

Table 7. Verify performance on RPi4 (ms) with `ref` → `neon-pt` speedup (×)

Param	ref	opt_ref	neon	neon-pt	×
FAEST					
128s	21,746	471	381	239	91.0
128f	2,963	85	73	54	54.9
192s	79,227	1,662	1,455	1,009	78.5
192f	9,952	324	295	243	41.0
256s	175,427	3,167	2,910	1,911	91.8
256f	20,523	560	533	415	49.5
FAEST-EM					
EM-128s	16,473	364	287	174	94.7
EM-128f	2,245	68	57	43	52.2
EM-192s	69,977	1,194	1,008	545	128.4
EM-192f	7,319	211	190	142	51.5
EM-256s	146,781	2,127	1,851	966	151.9
EM-256f	17,183	422	390	290	59.3

Table 8. Verify performance on Apple M2 (ms) with `ref` → `neon-pt` speedup (×)

Param	ref	opt_ref	neon	neon-pt	×
FAEST					
128s	5,399	39.2	38.7	29.5	183.0
128f	737	12.4	12.4	11.1	66.4
192s	20,056	101.2	114.5	87.5	229.2
192f	2,453	37.0	38.4	35.3	69.5
256s	43,254	169.8	218.7	156.0	277.3
256f	5,066	53.6	58.8	52.2	97.0
FAEST-EM					
EM-128s	4,095	28.4	26.1	18.9	216.7
EM-128f	559	9.9	9.5	8.8	63.5
EM-192s	17,398	72.5	81.7	53.0	328.3
EM-192f	1,819	25.1	26.3	23.7	76.8
EM-256s	36,409	109.7	144.2	90.4	402.8
EM-256f	4,249	43.4	47.5	42.7	99.5

Verification Performance The overall performance trend of the Verify operation was derived similarly to that of the signing operation. Table 7 and Table 8 present the verification performance across all build configurations. In general, the verification operation shows an execution time approximately 10–25% shorter

than the signing operation, because only the seeds along the path opened (Open) by the signer are reconstructed in a limited manner, resulting in structurally fewer PRG calls than in the signing process, which requires expanding the entire GGM tree. The speedup of `neon-pthread` over `ref` was also observed in a range similar to that of signing.

Figure 8 summarizes the results of a comprehensive comparison of signing-operation speedups measured on both platforms, organized by build configuration. Each subsequent subsection analyzes step by step how the performance improvement stems from NEON SIMD acceleration, pthread task parallelization, and implementation characteristics compared to OpenSSL.

4.3 Effect of NEON SIMD Acceleration

The performance improvement metric from the `ref` configuration to the `neon` configuration demonstrates the effectiveness of the NEON TBL/TBX-based AES acceleration and PRG/OWF vectorization, which are the core contributions of

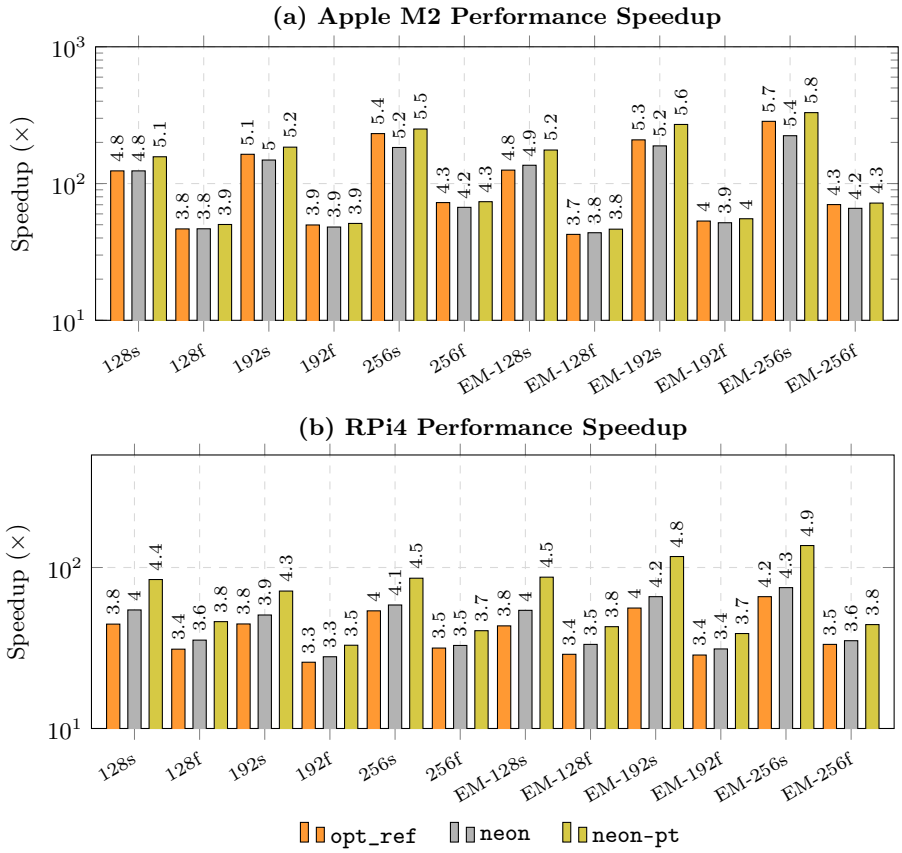

Fig. 8. Comprehensive speedup analysis of FAEST signing.

this work. As shown in Table 5 and Table 6, when only NEON SIMD acceleration was applied, speedups of 28.0–75.0× on RPi4 and 43.7–223.8× on M2 over `ref` were observed. This performance improvement is the result of the combined effect of the following three technical factors.

1. SubBytes acceleration through register-resident S-box and TBL/TBX-based 4-stage lookup.
2. Effective instruction-level parallelism was achieved by simultaneously encrypting multiple blocks through the 4-way and 8-way parallel AES structures.
3. The efficiency of the pipeline was maximized by designing a fixed-length PRG function optimized for the FAEST tree structure, eliminating loops and conditional branches.

In particular, a clear trend is observed in which the magnitude of speedup increases as the security level increases. This is because AES-256 (14 rounds) requires more rounds than AES-128 (10 rounds), so the cumulative effect of SIMD acceleration applied on a per-round basis manifests itself significantly. Furthermore, the s variant achieves higher speedups than the f variant because the s variant generates more VOLE instances, and as the PRG call frequency increases, the contribution of the optimized AES engine is relatively amplified.

In the M2 environment, speedups of up to 43.7–223.8× were achieved with the same NEON SIMD code. Despite M2's baseline `ref` speed being 4.0× faster than RPi4, the speedup ratios from optimization are even larger, which is analyzed to be because M2's wide out-of-order execution pipeline schedules the ILP characteristics of 4/8-way parallel AES more effectively than RPi4.

4.4 Effect of pthread Parallelization

Performance scaling from `neon` build to `neon-pthread` build demonstrates the effectiveness of batch-level VOLE task parallelization designed in Section 3.3. Table 9 summarizes the additional speedup ratios obtained when pthread parallelization is applied.

On RPi4, task parallelization yielded additional computational speedups of 1.18–1.82×, while on M2 it showed improvements of 1.06–1.48×. The reason why parallelization efficiency is relatively higher on RPi4 can be analyzed from two perspectives.

1. Because RPi4 has relatively low single-core performance and individual VOLE instances have longer execution latencies, the system overhead incurred during thread creation and joining constitutes a smaller proportion of the overall computation.
2. In the M2 environment, individual VOLE instances complete within only a few tens of milliseconds, so the relative proportion of thread-management overhead inversely increases.

Table 9. pthread parallelization speedup (×): `neon` → `neon-pthread`

Param	RPi4	Apple M2
FAEST		
128s	1.55	1.27
128f	1.30	1.08
192s	1.41	1.24
192f	1.18	1.06
256s	1.47	1.36
256f	1.24	1.10
FAEST-EM		
EM-128s	1.61	1.29
EM-128f	1.29	1.06
EM-192s	1.78	1.43
EM-192f	1.25	1.07
EM-256s	1.82	1.48
EM-256f	1.26	1.09

From a parameter perspective, the s variant consistently showed higher parallelization efficiency than the f variant. This is because the number of VOLE instances (τ) generated by the s variant is larger, which allows idle time of cores to be reduced and processor utilization to be increased in a batch-level scheduling environment.

4.5 Comparative Analysis with OpenSSL

The performance comparison between the `opt_ref` build and the `neon` build shows the difference in AES acceleration efficiency between the widely used OpenSSL library and the NEON TBL/TBX structure independently implemented in this work. The experimental results showed contrasting patterns on the two platforms depending on whether ARMv8 Crypto Extensions is supported.

RPi4 (without Crypto Extensions): The Cortex-A72 architecture on RPi4 does not support hardware cryptographic instructions, so OpenSSL also runs as a software-based general-purpose AES routine. Under these conditions, the `neon` build showed 1.04–1.25× better performance than `opt_ref` across all parameter ranges. This suggests that the proposed TBL/TBX-based register-resident S-box algorithm utilizes ARM NEON register resources more efficiently than OpenSSL's general-purpose software implementation.

M2 (with Crypto Extensions): OpenSSL in the M2 environment utilizes the processor's built-in AESE/AESMC hardware instructions. At the 128-bit security level, `neon` and `opt_ref` showed virtually equivalent performance. However, as the security level increases, the efficiency of hardware-dedicated instructions

becomes increasingly prominent, and for FAEST-256s, `opt_ref` performed approximately 21% faster computation than `neon`. This is because the hardware method that processes AES rounds with a single instruction is inherently more efficient than the TBL/TBX-based software substitution method, and the gap accumulates in the AES-256 environment performing 14 rounds.

Nevertheless, the `neon-pthread` build, which integrates multi-threading, showed faster results than `opt_ref`'s hardware-accelerated performance across all parameters even in the M2 environment. For FAEST-256s, `neon-pthread` showed a 7.4% advantage over `opt_ref`, and for FAEST-EM-256s, it recorded a 13.7% reduction in processing time. This demonstrates that the combination of NEON SIMD and task-level parallelization can exceed the throughput of the best available single-thread configuration, including hardware-accelerated OpenSSL, without dedicated hardware cryptographic instructions.

4.6 Cross-Platform Analysis

The performance gap between M2 and RPi4 derived under the same build conditions reflects the impact of architectural characteristics on performance. Based on the pure-software `ref` build, M2 showed a consistent performance advantage of approximately 4.0× over RPi4 across all parameter ranges. This reflects the difference in single-threaded baseline computation performance between the two processors.

In contrast, in the optimized `neon-pthread` build, the gap between M2 and RPi4 widened to 4.3–11.9×, with the performance difference becoming more pronounced for parameters with higher security levels. Since these figures exceed the gap attributable to simple single-core performance differences, this is analyzed to be because M2's wider out-of-order execution pipeline, larger per-core cache, and higher memory bandwidth improve the throughput of each individual thread, amplifying the gains from the same 4-thread parallel configuration.

4.7 Structural Comparison of FAEST and FAEST-EM

The FAEST-EM algorithm configuration consistently shows shorter signing times than the standard FAEST configuration across all parameters. For example, based on the RPi4 `neon-pthread` build, FAEST-EM-256s showed 1.91× computational efficiency compared to standard FAEST-256s. This is because as the EM variant replaces the key schedule with fixed permutation logic, the number of multiplicative gates constituting the zero-knowledge proof circuit decreases, and the absolute amount of base data to be processed per VOLE instance is reduced [4]. Since this algorithm-level reduction operates independently of the SIMD and multithreading proposed in this work, the highest absolute performance improvement can be achieved when the proposed techniques are integrated and applied to FAEST-EM.

5 Conclusion

In this paper, we proposed an optimization technique for implementation that accelerates the AES-CTR-based PRG operation, which is the core computational bottleneck of the FAEST digital signature algorithm, using ARM NEON SIMD instructions. The proposed technique consists of three layers of optimization strategies.

1. We implemented a SIMD AES engine that keeps the entire 256 B S-box resident in vector registers using `TBL`/`TBX` table lookup instructions and simultaneously processes multiple AES blocks through `4-way` and `8-way` parallel structures.
2. By exploiting the fixed output-size property inherent to the FAEST tree structure, we designed a dedicated PRG function with loop unrolling and conditional branches eliminated, along with a hierarchical throughput optimization structure.
3. Based on the data independence between VOLE instances, we applied pthread-based batch-level task parallelization.

A comprehensive performance evaluation was conducted on all 12 parameter sets of the FAEST v2 specification in RPi4 and Apple M2 environments, which differ in whether ARMv8 Crypto Extensions hardware acceleration is supported. The results show that, compared to the pure-C reference implementation, speedups of up to 136.9× on RPi4 and up to 330.1× on M2 were achieved for the signing operation. In particular, in the RPi4 environment without Crypto Extensions, the proposed NEON implementation showed superior performance over the OpenSSL-based software implementation across all parameters, and even in the M2 environment with hardware acceleration support, the combined NEON SIMD and pthread optimization outperformed the best available reference configuration, including hardware-accelerated OpenSSL. This empirically demonstrates that the combination of general-purpose NEON SIMD and task-level parallelization can exceed the throughput of single-thread hardware-accelerated implementations, without dedicated hardware cryptographic instructions.

Furthermore, by adopting a register-resident S-box architecture and bitmasking-based MixColumns operations, the implementation confines all substitution data within the CPU register file, reducing secret-dependent memory-access patterns that could be exploited through cache-timing side channels. This strengthens constant-time properties compared with the byte-level memory operations in the existing reference implementation.

References

1. Adomnicai, A., Peyrin, T.: Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM cortex-m and RISC-V. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(1), 402–425 (2021). `https://doi.org/10.46586/tches.v2021.i1.402-425`

2. Alagic, G., Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., et al.: Status report on the third round of the nist post-quantum cryptography standardization process (2022)
3. Arm Ltd.: Arm Architecture Reference Manual for A-profile Architecture (2024), https://developer.arm.com/documentation/ddi0487/latest/, document DDI 0487. Accessed February 2026
4. Baum, C., Beullens, W., Braun, L., Delpech de Saint Guilhem, C., Klooß, M., Majenz, C., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: FAEST: Algorithm specifications and supporting documentation, version 2.0 (2024), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/faest-spec-round2-web.pdf, nIST PQC Additional Signatures, Round 2 submission. Accessed February 2026
5. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: optimizing ggm trees and owfs for post-quantum signatures. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 463–493. Springer (2024)
6. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Orsini, E., Roy, L., Scholl, P.: Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In: Annual international cryptology conference. pp. 581–615. Springer (2023)
7. Bernstein, D.J.: Cache-timing attacks on AES (2005)
8. 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. pp. 2129–2146 (2019)
9. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Professional (1993)
10. Dworkin, M.: Recommendation for block cipher modes of operation. NIST special publication **800**, 38B (2001)
11. Even, S., Mansour, Y.: A construction of a cipher from a single pseudorandom permutation. Journal of cryptology **10**(3), 151–161 (1997)
12. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
13. Fujii, H., Rodrigues, F.C., López, J.: Fast AES implementation using ARMv8 ASIMD without cryptography extension. In: International Conference on Information Security and Cryptology. pp. 84–101. Springer (2019)
14. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM (JACM) **33**(4), 792–807 (1986)
15. Hamburg, M.: Accelerating aes with vector permute instructions. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 18–32. Springer (2009)
16. Mukherjee, S., Ramacher, S.: FAEST reference implementation (2023), https://github.com/faest-sign/faest-ref, gitHub repository. Accessed February 2026
17. National Institute of Standards and Technology: Call for additional digital signature schemes for the post-quantum cryptography standardization process (2022), https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals, accessed February 2026
18. National Institute of Standards and Technology: Advanced encryption standard (AES). FIPS Publication 197, National Institute of Standards and Technology (2023). https://doi.org/10.6028/NIST.FIPS.197-upd1

19. OpenSSL Software Foundation: OpenSSL: Cryptography and SSL/TLS toolkit (2024), `https://www.openssl.org/`, version 3.x. Accessed February 2026
20. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
21. Wang, J., Vadnala, P.K., Großschädl, J., Xu, Q.: Higher-order masking in practice: A vector implementation of masked AES for ARM NEON. In: Cryptographers' Track at the RSA Conference. pp. 181–198. Springer (2015)
22. Yang, K., Sarkar, P., Weng, C., Wang, X.: Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. pp. 2986–3001 (2021)