Adapting AES-Oriented Optimizations to Rijndael-256: Cortex-M4, ARMv8-A, and CUDA

Siwoo Eum[1], Minho Song[1], Minjoo Sim[1], Anupam Chattopadhyay[2] and Hwajeong Seo[1]

[1] Hansung University, Seoul, Republic of Korea,
shuraatum@gmail.com,smino0906@gmail.com,minjoos9797@gmail.com,hwajeong84@gmail.com
[2] Nanyang Technological University, Singapore, anupam@ntu.edu.sg

Abstract. Rijndael-256 (R256), the 256-bit block variant of the Rijndael family, is practically relevant in ongoing NIST draft discussions on wider-block standardization and in several NIST post-quantum signature candidates. Relative to AES, R256 combines a wider 4×8 state with non-standard ShiftRows offsets $(0, 1, 3, 4)$, invalidating key assumptions behind many AES-oriented optimizations. We study how these mismatches manifest on three targets and develop three corresponding adaptation strategies: *wider classical bitslicing* on Cortex-M4, *Reuse with Reshuffling* on AArch64, and *Amortize the Mismatch* on CUDA. This yields a bitsliced Cortex-M4 implementation with secret-independent control flow and memory access patterns (4,962 cycles/block), an AArch64 pre-shuffle design that reuses the `AESE` instruction with AddRoundKey folded in (up to 6,520 MB/s), and a CUDA shared-memory T-table implementation reaching 81.16 GiB/s; we report throughput on AArch64 and CUDA because per-block cycle measurement is not directly available on those platforms. Replacing the R256 cryptographic core in four NIST PQC reference codebases on Apple M2, with minimal integration changes where needed, yields signing speedups of 1.18×–114.3× and verification speedups up to 155.6×, with the end-to-end gain governed by each scheme's R256 fraction.

Keywords: Rijndael-256 · AES · ARM Cortex-M4 · ARMv8-A · NEON · CUDA · ShiftRows · Bitslicing · T-table

1 Introduction

The Rijndael family of block ciphers was designed to allow independent selection of block size (128, 192, or 256 bits) and key size [DR02]. In 2001, however, NIST standardized only the 128-bit block variant as AES [DBN+01], and the implementation techniques refined over the subsequent 25 years—T-tables [DR02], bitslicing [Bih97], fixslicing [AP21], hardware instructions [Arm23, Gue09], and GPU T-tables [Man07, Tez21]—have all been built around the 4×4 state and ShiftRows$(0, 1, 2, 3)$ (the per-row cyclic left-shift offsets, for rows from top to bottom) of the 128-bit block.

Rijndael-256 (hereafter R256), with its 256-bit block size, extends the state to a 4×8 matrix and changes ShiftRows to $(0, 1, 3, 4)$ [DR02]. The central question addressed by this paper is: *What prevents existing AES optimizations from carrying over to R256, and how can these obstacles be circumvented on each target platform?*

An efficient implementation of R256 matters on two fronts. NIST has issued pre-draft calls for comments for SP 800-197 [Nat24] and SP 800-197A [Nat25], placing Rijndael-256 and accordion modes for 256-bit block ciphers in ongoing standardization discussions. SP 800-197 discusses approval of R256 with a 256-bit key, while SP 800-197A

initiates the development of cryptographic accordions and identifies Acc256 as a mode for a 256-bit block cipher. At the same time, several candidates in NIST's additional-signatures call [Nat22]—including SDitH [MFG+23], Mirath [AAB+25], RYDE [ABB+25], and MQOM [BFR24]—employ R256 as an internal symmetric primitive, so the implementation performance of R256 directly affects the practical viability of these schemes.

Our analysis shows that the main obstacles to adapting AES optimizations arise from the combination of R256's wider 4×8 state and its modified ShiftRows offsets. The relative importance of these two changes depends on the target architecture and on how tightly a given AES optimization is coupled to the AES $(0, 1, 2, 3)$ row-shift pattern. We propose three adaptation strategies to resolve these mismatches:

First, on ARM Cortex-M4, the highest-performing AES technique for this target is fixslicing [AP21]. Fixslicing assumes uniform inter-row ShiftRows differences, and R256's offsets $(0, 1, 3, 4)$ violate this assumption. We adopt a **wider classical bitslicing** strategy (§3): abandon fixslicing and fall back to classical bitslicing redesigned for R256. This negative result constitutes concrete evidence that ShiftRows regularity is a critical precondition for adapting fixslicing.

Second, on AArch64, the `AESE` instruction hard-wires ShiftRows$(0, 1, 2, 3)$, yet the position-independence of SubBytes allows the input to be pre-shuffled so that hardware acceleration is preserved. We apply a **Reuse with Reshuffling** strategy (§4).

Third, on CUDA GPUs, we employ an **Amortize the Mismatch** strategy (§5), absorbing the overhead from the wider state through massive thread-level parallelism.

Contributions. The contributions of this paper are as follows.

1. We identify the specific mechanisms by which the modified ShiftRows pattern and widened state obstruct the adaptation of AES optimizations, on a per-platform basis, and propose three adaptation strategies to address them.

2. We present R256-specific optimized implementations for three platforms (Cortex-M4, AArch64, and CUDA) and conduct a cross-platform performance evaluation covering cycle counts, throughput, and side-channel-relevant implementation properties, including secret-independent control flow and memory access patterns where applicable.

3. We integrate our implementations into four NIST PQC signature candidates on AArch64 and quantitatively analyze the end-to-end performance impact within the reference codebases, with minimal benchmark or interface adaptations where required.

Paper organization. §2 describes the structure of the Rijndael family and its differences from AES. §3–§5 present the per-platform adaptation strategies and implementations. §6 provides the PQC impact analysis, and §7 concludes.

2 Background

This section surveys the structure of the Rijndael family of ciphers, highlights the key differences between AES (Rijndael-128) and R256, and, alongside prior work, analyzes why existing AES optimization techniques do not carry over directly to R256.

2.1 Overview of the Rijndael Family

Rijndael, designed by Joan Daemen and Vincent Rijmen, is a family of block ciphers that allows independent selection of block size and key size from 128, 192, and 256 bits [DR02].

Table 1: Parameter comparison of AES-128, AES-256, and Rijndael-256. The ShiftRows offsets denote the cyclic left-shift amounts for rows $0, 1, 2, 3$.

Parameter	AES-128	AES-256	Rijndael-256
Block size (bits)	128	128	256
State columns (N_b)	4	4	8
Key size (bits)	128	256	128/192/256
Key columns (N_k)	4	8	4/6/8
ShiftRows	$(0, 1, 2, 3)$	$(0, 1, 2, 3)$	$(0, 1, 3, 4)$
Rounds (N_r)	10	14	14

Although this flexibility was a defining feature of the original specification, the variant adopted by NIST as the Advanced Encryption Standard (AES) in 2001 is restricted to a fixed 128-bit block size [DBN+01]. AES-128, AES-192, and AES-256 all share the 128-bit block ($N_b = 4$) and differ only in key size (128/192/256 bits) and number of rounds (10/12/14).

R256 denotes the variant with a 256-bit block ($N_b = 8$). It operates on a 32-byte state and, as prescribed by the original specification, executes 14 rounds regardless of key size [DR02]. The practical motivation for efficient R256 implementations—NIST's pre-draft calls for comments on SP 800-197/197A [Nat24, Nat25], including Rijndael-256 and the Acc256 accordion mode, and R256's adoption as an internal primitive in NIST additional-signature PQC candidates [Nat22]—is detailed in §1.

2.2 Structural Differences from AES

The round function of R256 consists of the same four steps as AES—SubBytes, ShiftRows, MixColumns, and AddRoundKey. SubBytes applies the same 8-bit S-box independently to each byte, and MixColumns applies the same 4×4 MDS matrix over $\mathrm{GF}(2^8)$ to each column. Because these two transformations are independent of the number of columns N_b, an existing AES S-box implementation or MixColumns logic can be reused for R256 without modification.

The key differences lie in the state array dimensions and the ShiftRows shift offsets. The AES state is a 4×4 byte matrix ($N_b = 4$), whereas R256 uses a 4×8 byte matrix ($N_b = 8$). Doubling the state size also changes the ShiftRows offsets: the cyclic left-shift amounts for row i are $(0, 1, 2, 3)$ in AES but $(0, 1, 3, 4)$ in R256 [DR02].

Table 1 summarizes the key parameters of AES-128, AES-256, and R256. While AES-256 and R256 share the same key size and round count, they differ in block size, number of state columns, and ShiftRows offsets.

The shift-offset discrepancy affects implementation in two principal ways. First, the shift amount for row 2 changes from 2 to 3, breaking the arithmetic-progression structure of $(0, 1, 2, 3)$ that AES exhibits. This regularity underpinned a variety of indexing optimizations, none of which carry over to the $(0, 1, 3, 4)$ pattern. Second, the doubled state size widens each bit-plane in a bitslicing representation from 16 bits (AES) to 32 bits (R256). On a 32-bit register file, each AES bit-plane is only 16 bits wide, allowing two blocks to be packed into a single register; a single R256 bit-plane already fills an entire register, precluding this form of parallelism.

2.3 Prior Work and Adaptation Assumptions

Existing AES optimization techniques implicitly or explicitly assume $N_b = 4$ and ShiftRows $(0, 1, 2, 3)$. This subsection reviews the main techniques alongside prior work, identifying which assumptions break when targeting R256.

Bitslicing. Bitslicing [Bih97] maps each bit of a block cipher's state to a separate register bit, achieving gate-level parallelism. Schwabe and Stoffelen [SS16] presented a bitslicing implementation that processes two AES blocks in parallel using 14 registers on an ARM Cortex-M, demonstrating that side-channel-resistant AES is practical even on resource-constrained microcontrollers. In bitslicing, ShiftRows is realized as a bit permutation. For $N_b = 4$, the $(0, 1, 2, 3)$ shifts form an arithmetic progression and yield a relatively regular permutation. For $N_b = 8$, the $(0, 1, 3, 4)$ shifts lack this regularity, requiring more complex explicit bit swaps.

Fixslicing. Fixslicing [AP21] refines bitslicing and holds the AES speed records on ARM Cortex-M and RISC-V. In classical bitslicing, the per-round cost of rearranging bits for ShiftRows is substantial. Fixslicing eliminates this cost by **keeping the bit layout of the state fixed across rounds and instead transforming the round function itself**. Because the AES $(0, 1, 2, 3)$ shifts align with 32-bit register rotations, the accumulated shifts cycle through $0, 8, 16, 24, 0, \ldots$ bits with a period of four rounds, and the entire cipher can be covered by only 2–4 round variants. With the R256 $(0, 1, 3, 4)$ shifts, rows 1 (8 bits) and 2 (24 bits) do not align with register boundaries, causing the accumulation period to grow to eight rounds or more and drastically increasing both the number of variants and the code size. A detailed analysis is presented in §3.

Hardware AES instructions. On ARMv8-A, `AESE`/`AESD` [Arm23] perform AddRoundKey, SubBytes, and ShiftRows on a 128-bit input in a single instruction (MixColumns is handled separately by `AESMC`). On x86, `AESENC`/`AESDEC` [Gue09] fuse SubBytes, ShiftRows, MixColumns, and AddRoundKey into one instruction. On both architectures, the ShiftRows pattern is hardwired to $(0, 1, 2, 3)$, which is incompatible with the $(0, 1, 3, 4)$ offsets of R256. Moreover, because these instructions operate on 128-bit registers, handling the 256-bit state requires splitting it into two 128-bit halves and resolving inter-half cross-column shifts separately. On x86, a technique that corrects the fixed ShiftRows of `AESENC` via pre- and post-byte shuffles to adapt it for R256 has been reported [DG22]. The AArch64 implementation presented in this paper (§4) proposes a pre-shuffle approach based on a similar principle, using the ARM `TBL` instruction.

T-table. The T-table technique proposed by Daemen and Rijmen [DR02] merges SubBytes, ShiftRows, and MixColumns into four precomputed lookup tables. Because T-table indexing depends on the post-ShiftRows byte positions, changing the shift offsets requires adjusting the index computation. While this adjustment is straightforward, scaling to $N_b = 8$ doubles the number of table lookups per round from 16 to 32, altering cache and shared-memory access patterns. T-table implementations are inherently vulnerable to cache-timing attacks because the table indices depend on secret data [OST06]; accordingly, this paper uses T-table implementations only as a performance baseline.

GPU shared memory. GPU-based AES implementations [Man07, HW07] load T-tables into shared memory and optimize the bank conflict pattern for the 16-byte state. Tezcan [Tez21] proposed a T-table layout that completely eliminates bank conflicts, achieving high throughput. With a 32-byte state, the number of lookups per round doubles to 32, and the increased per-thread register and shared-memory usage also affects occupancy. This trade-off is analyzed in detail in §5.

Novelty boundary. Most prior work has focused on optimizing AES-128 ($N_b = 4$). To the best of our knowledge, no peer-reviewed work has systematically analyzed the portability of AES optimization techniques to R256 across multiple platforms. The closest single-platform studies are Drucker and Gueron's AES-NI adaptation for x86-64 [DG22]

and recent work by Lestari et al. [LSR+26], who present a constant-time bitsliced R256 implementation for Cortex-M4, likewise use explicit `UBFX`/`BFI` ShiftRows routing, and find fixslicing suboptimal. Their Cortex-M4 findings corroborate ours. Our study instead provides a unified portability analysis across multiple CPU and GPU platforms and evaluates the AArch64 implementations in PQC signature schemes.

2.4 R256 in Post-Quantum Cryptography

Among the second-round candidates of NIST's additional-signatures call [Nat22], several MPC-in-the-Head-based signature schemes use AES-128 as their default symmetric primitive but switch to R256 for parameter sets at Level 3 (192-bit security) and above to obtain a wider security margin. The larger 256-bit block size raises the generic birthday bound from about 2^{64} queries for 128-bit-block AES instantiations to about 2^{128} for R256. This provides substantially more headroom when the cipher is used repeatedly as an internal symmetric primitive for commitments, seed expansion, and related PRF-like calls in higher-security parameter sets. SDitH [MFG+23], Mirath [AAB+25], RYDE [ABB+25], and MQOM [BFR24] fall into this category. Because a significant fraction of these algorithms' execution time is spent in R256-based primitive calls—ranging from roughly 16% to over 99%, depending on the algorithm and operation—an efficient R256 implementation is directly tied to the practical viability of these PQC schemes. Precise breakdowns and measurement results are presented in §6.

2.5 Constant-Time Properties

This paper reports the constant-time (CT) status of each implementation as a security-relevant property alongside performance. We use CT in a software-level sense: an implementation is CT if its control flow and memory access patterns are independent of secret data. This definition does not by itself imply masking or resistance against physical side-channel attacks such as power or electromagnetic analysis; such countermeasures are outside the scope of this work. §7 only discusses why masked R256 implementations remain future work.

Among the techniques considered in this paper, bitslicing [Bih97] is inherently compatible with this definition because it expresses the cipher using only bitwise logic instructions and fixed shifts, without data-dependent branches or table accesses. Schwabe and Stoffelen [SS16] showed that this property enables constant-time AES software on ARM Cortex-M, and our Cortex-M4 bitsliced implementation in §3 follows the same principle.

Under the same definition, the AArch64 ARM Crypto path in §4 is also CT at the software level. The `AESE`/`AESMC` round operations [Arm23] and the `TBL`-based pre-shuffle are executed through a fixed instruction sequence on register-resident data and therefore do not introduce secret-dependent branches or memory accesses. The NEON software fallback in §4 likewise keeps the entire S-box in vector registers and performs substitution via `TBL`/`TBX`, eliminating memory-based table lookups while preserving secret-independent control flow and memory access patterns.

Software-level CT does not by itself guarantee data-independent instruction *timing*. AArch64 provides FEAT_DIT [Arm23] for this, forcing documented instructions such as `AESE` and `AESMC` to run in operand-independent time once `PSTATE.DIT` is set. With DIT enabled around our ARM Crypto encrypt loop on the Apple M2, we observe no measurable steady-state slowdown (the point estimates differ by at most 0.06%). FEAT_DIT (ARMv8.4-A) is unavailable on the older Cortex-A76 and A72 cores used here, where data-independence instead rests on the microarchitecture.

By contrast, T-table implementations are not CT under this definition: the lookup indices depend on secret data, producing data-dependent memory access patterns [OST06].

This applies to the Cortex-M4 and AArch64 T-table baselines as well as to the CUDA implementation in §5. These variants are therefore used only as performance baselines.

3 Cortex-M4: When Fixslicing Does Not Carry Over

The ARM Cortex-M4 exposes 16 general-purpose 32-bit registers, of which 14 are usable for data (`r13` is reserved as `SP` and `r15` as `PC`; `r14`/`LR` serves as a temporary). It has no SIMD extensions and no AES hardware instructions, so AES implementations on this platform rely on software techniques; the current state of the art is fixslicing [AP21]. We start from the observation that fixslicing's core assumption is incompatible with R256's $\mathsf{ShiftRows}(0, 1, 3, 4)$. We first analyze an attempt to apply fixslicing to R256 and its limitations, then propose a **wider classical bitslicing** strategy that falls back to classical bitslicing [Bih97] while redesigning the permutation circuit to suit R256.

3.1 T-table Baseline

The T-table technique [DR02] merges SubBytes, ShiftRows, and MixColumns into four precomputed lookup tables (T_0, T_1, T_2, T_3). Each table occupies $256 \times 4 = 1{,}024$ bytes, requiring 4 KB of ROM/Flash in total. In AES ($N_b = 4$), a single round requires 16 table lookups and XOR operations, whereas R256 ($N_b = 8$) doubles the column count to eight, increasing the per-round lookup count to 32. Because $\mathsf{ShiftRows}(0, 1, 3, 4)$ must be reflected in the lookup index computation, the index pattern from AES code cannot be reused directly; byte indexing must be rewritten to match $(0, 1, 3, 4)$ mod 8. However, this modification amounts to changing index offset constants and thus incurs negligible implementation effort.

Starting from the reference implementation in Appendix C of the original Rijndael specification [DR02], our C implementation using T-tables records 3,785 cycles for 256-bit keys (14 rounds). Rewriting in assembly with manual register allocation optimization reduces this to 3,174 cycles, roughly 16% improvement over the C version. Because T-table implementations use data-dependent table lookups, they are vulnerable to timing side-channels [OST06]. We use them here solely as a performance baseline.

3.2 Bitsliced Design with Explicit ShiftRows

Bitslicing [Bih97] expresses all operations using only bitwise logic instructions and fixed shifts, providing secret-independent control flow and memory access patterns.

Register pressure relative to AES. In bitslicing, an N-byte state is decomposed into eight bit-planes, each containing N bits. On the Cortex-M4's 32-bit registers, a single register suffices for any bit-plane of 32 bits or fewer.

In AES ($N_b = 4$) bitslicing [SS16], each bit-plane of the 128-bit state is only 16 bits wide, allowing multiple blocks (up to two) to be interleaved within a single register for parallel processing. Eight bit-planes occupy eight registers, leaving six spare registers available for intermediate computations.

For R256, the 256-bit state yields 32-bit bit-planes, each filling a register exactly. Our implementation maps the eight bit-planes to `r4`–`r11`, consuming eight of the 14 usable registers for the state alone. Only six registers remain for operations such as the S-box computation. The multi-block interleaving possible in AES bitslicing is infeasible, as sketched in Figure 1, and frequent spill/reload of intermediate values to the stack during the GF(2^8) algebraic decomposition of the S-box becomes unavoidable.

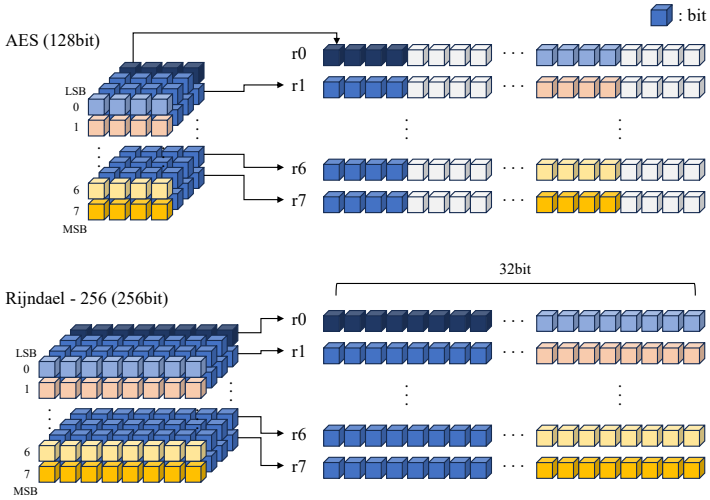


Figure 1: Bitslicing byte-to-bit-plane transpose on 32-bit registers: AES (top) vs. Rijndael-256 (bottom). Register r_i holds bit i of every state byte.

Packing/unpacking. Conversion between byte order (normal domain) and bitsliced order (bitsliced domain) is implemented via SWAPMOVE-based bit transposition, identical to AES bitslicing [SS16]. The 32-byte plaintext is loaded in big-endian order into eight 32-bit words (`r4`–`r11`), after which a three-stage SWAPMOVE chain at distances 1, 2, and 4 (masks `0x55555555`, `0x33333333`, `0x0f0f0f0f`) transforms each register into a single 32-bit bit-plane. In AES, where bit-planes are 16 bits wide, the same packing operates on four words; in R256 it must be applied to all eight words. Unpacking reverses the three stages in order (distances 4, 2, 1).

SubBytes: GF(2^8) algebraic decomposition. In the bitsliced domain, SubBytes is implemented as a logic-gate circuit for GF(2^8) inversion followed by the affine transformation. The standard technique of decomposing GF(2^8) as a tower extension of GF(2^4) reduces the inversion to multiplications, squarings, and inversions over GF(2^4). Since AES and R256 share the same S-box, the existing Boyar–Peralta optimized S-box circuit [BP12] (109 gates) can be reused without modification. All intermediate operations use only bitwise logic instructions (`AND`, `EOR`, `ORR`), preserving the constant-time property.

ShiftRows: explicit byte permutation. Even in classical AES ($N_b = 4$) bitslicing, ShiftRows requires an explicit bit permutation, but the arithmetic progression $(0, 1, 2, 3)$ keeps the instruction count relatively low. In R256 ($N_b = 8$), the shift offsets $(0, 1, 3, 4)$ do not align with 32-bit boundaries, making ShiftRows the dominant cost factor in our implementation. In the bitsliced representation, each 32-bit register holds one bit-plane in which the bytes of rows 0–3 are concatenated at 8-bit intervals. The difficulty is that each row requires a different shift amount (row 1 a 1-byte rotation, row 2 a 3-byte rotation, and row 3 a 4-byte rotation), yet all rows coexist within a single register, so a full-register `ROR` cannot address them. Each row's byte must be individually extracted, rotated, and reinserted.

Our implementation resolves this within the bitsliced domain by combining ARM's `UBFX` (bit-field extract) and `BFI` (bit-field insert) into a byte-rotation macro. Rotating a single byte requires four instructions: `UBFX`–`ORR`–`UBFX`–`BFI`. Applying this to rows 1, 2, and 3 costs 12 instructions per register, totaling **96 instructions** across all eight bit-planes. By comparison, AES bitslicing also requires an explicit ShiftRows permutation, but the

regularity of $(0, 1, 2, 3)$ keeps the instruction count much lower.

MixColumns, AddRoundKey, and key schedule. MixColumns follows the same algebraic structure as in AES. It consists of `xtime` (multiplication by 2) over GF(2^8) combined with XOR, which in the bitsliced domain translates to shifts and XORs between bit-planes. Although the number of columns doubles from 4 to 8, the per-column operation remains identical, so the AES bitsliced MixColumns logic is simply applied column by column.

AddRoundKey is a straightforward XOR in the bitsliced domain. The round key is converted to bitsliced form, and each bit-plane register is XORed with the corresponding round key bit-plane.

The key schedule is performed in the normal domain, following the R256 key expansion algorithm from the specification [DR02]. The expanded round key array is converted to bitsliced form via SWAPMOVE-based packing before encryption begins.

3.3 Fixslicing: A Negative Result

Fixslicing [AP21], proposed by Alexandre Adomnicai and Thomas Peyrin, is an improved bitslicing technique that holds the AES speed record on ARM Cortex-M and RISC-V platforms. This section presents the result of applying fixslicing to R256. The result is **negative**: fixslicing turns out to be slower than classical bitslicing.

Core principle of fixslicing. In classical bitslicing, the bit rearrangement corresponding to ShiftRows must be performed explicitly each round. Fixslicing [AP21] eliminates this cost by keeping data in a fixed layout and instead transforming the round function itself.

This strategy is efficient for AES because of how MixColumns is implemented. The key operation in bitsliced MixColumns is the adjacent-row XOR $d_i = s_i \oplus s_{(i+1) \bmod 4}$, which, when the data is column-aligned, can be computed as $s \oplus (s \texttt{ ror } 24)$, i.e., a single full-register `ROR`. When ShiftRows is omitted and the positional offsets accumulate across rounds, a displacement arises between adjacent rows. In AES $(0, 1, 2, 3)$, the inter-row shift differences are uniformly $(1, 1, 1, 1)$, so this positional correction is likewise absorbed by a single full-register `ROR`. As a result, each MixColumns variant incurs almost no additional cost, and only 2–4 variants suffice to cover all rounds.

The misalignment problem in R256. For the `ROR` absorption to work, the accumulated inter-row positional differences after omitting ShiftRows must be identical across all row pairs. Given shift offsets $(\sigma_0, \sigma_1, \sigma_2, \sigma_3)$, define the inter-row differences as $\Delta_i = (\sigma_{i+1} - \sigma_i) \bmod N_b$. In fixslicing, when computing the adjacent-row XOR $d_i = s_i \oplus s_{(i+1) \bmod 4}$, rows i and $i{+}1$ are misaligned by $8\Delta_i$ bits. Including the wrap-around pair (d_3: row 3→row 0), the rotation amounts required to correct each row pair are (Equation 1):

$$
\begin{aligned}
d_0 = s_0 \oplus (s \texttt{ ror } 8\Delta_0): &\quad \text{AES} \to \texttt{ror } 8, &\quad \text{R256} \to \texttt{ror } 8,\\
d_1 = s_1 \oplus (s \texttt{ ror } 8\Delta_1): &\quad \text{AES} \to \texttt{ror } 8, &\quad \text{R256} \to \texttt{ror } 16,\\
d_2 = s_2 \oplus (s \texttt{ ror } 8\Delta_2): &\quad \text{AES} \to \texttt{ror } 8, &\quad \text{R256} \to \texttt{ror } 8,\\
d_3 = s_3 \oplus (s \texttt{ ror } 8\Delta_3): &\quad \text{AES} \to \texttt{ror } 8, &\quad \text{R256} \to \texttt{ror } 32.
\end{aligned} \tag{1}
$$

In AES, all four row pairs use `ROR#8`, enabling simultaneous correction with a single word rotation. In R256, $\Delta = (1, 2, 1, 4)$ yields three distinct rotation amounts, making simultaneous correction with a single full-register rotation impossible. Consequently, each MixColumns variant must apply a pre-rotation to correct column alignment using per-byte rotations (`UBFX`/`BFI`, four instructions per byte), execute the MixColumns body, and then apply a post-rotation to restore the layout. The most expensive variant consumes

Table 2: Comparison of Rijndael-256 implementations on the Cortex-M4 (256-bit key, $N_r = 14$). ROM = encrypt-path code only (encrypt + key schedule + lookup tables). Decrypt code and inverse tables are excluded. Stack = call-chain peak across encrypt and key schedule.

Implementation	Enc (cyc)	KS (cyc)	ROM (B)	Stack (B)	CT
Reference C	12,374	3,525	3,216	240	No
T-table C	3,785	3,525	5,972	136	No
T-table ASM	3,174	3,525	2,880	76	No
Bitslice ASM	4,962	5,931	3,352	576	Yes
Fixslice ASM	5,481	10,299	8,924	696	Yes

pre-rotation 96 + MixColumns 35 + post-rotation 96 = 227 instructions (176 on average across the eight variants), over five times the roughly 40 instructions per variant in AES fixslicing. Moreover, the accumulated shift period extends to eight rounds, expanding the number of variants from the 2–4 needed for AES to **eight**.

Conclusion: Wider Classical Bitslicing. The performance advantage of fixslicing rests on the assumption that inter-row ShiftRows differences are uniform, allowing MixColumns positional correction to be absorbed into a single `ROR`. AES's $(0, 1, 2, 3)$ satisfies this assumption with inter-row differences $(1, 1, 1, 1)$, but R256's $(0, 1, 3, 4)$ violates it with $(1, 2, 1, 4)$. The result is hundreds of additional byte-rotation instructions per variant, combined with the increase to eight variants, which eliminates the core advantage of fixslicing.

Therefore, the correct approach for constant-time R256 on the Cortex-M4 is to abandon fixslicing and fall back to classical bitslicing, combined with the explicit ShiftRows permutation circuit from §3.2. This is the wider classical bitslicing strategy.

3.4 Performance Comparison and Discussion

Comparison across implementations. Table 2 summarizes five R256 implementations on the Cortex-M4, comparing encryption cycles, key schedule cycles, and constant-time (CT) status. All measurements were taken on an STM32F407 (168 MHz) using the DWT cycle counter with 256-bit keys (14 rounds). In R256, $N_r = \max(N_k, N_b) + 6$ with $N_b = 8$, so 128/192/256-bit keys all yield $N_r = 14$; consequently, **encryption cycles are identical regardless of key size**. Only the key schedule cost varies with key size.

ROM was measured for the encryption path per-function using `-ffunction-sections` and `-fdata-sections`; stack was obtained via `-fstack-usage` for C functions and prologue inspection for assembly routines. All implementations use a 480-byte round key context in RAM. The T-table ASM variant stores only `Te0` (1 KB) in ROM and derives `Te1`–`Te3` via byte rotations at runtime, accounting for its lower ROM relative to T-table C (four full tables, 4 KB).

The reference C implementation and both T-table baselines share the same byte-level R256 key-expansion routine, hence the identical key schedule cycle counts. The bitslice and fixslice variants incur higher costs because they perform additional SWAPMOVE-based bit-plane packing after key expansion; fixslice further requires round key rotation for each of its eight positional variants, reaching 1.7× the cost of bitslice.

Among the non-constant-time implementations, T-table ASM is the fastest at 3,174 cycles. Among the constant-time implementations, bitslice ASM is optimal at 4,962 cycles, outperforming fixslice ASM (5,481 cycles) by 10%. Reference C (12,374 cycles) is a straightforward byte-level reference implementation without table lookups or bitslicing, and thus has the lowest performance.

Constant-time overhead and loss of parallelism. Bitslice ASM is approximately 56% slower than T-table ASM (4,962 vs. 3,174 cycles). Two factors contribute to this gap. The first is the inherent cost of constant-time execution via bitslicing. The second is the loss of multi-block parallelism. In AES bitslicing [SS16], 16-bit bit-planes allow two blocks to be packed simultaneously into a single 32-bit register, halving the per-block cost. In R256, 32-bit bit-planes fill each register entirely, precluding such parallelism; processing always proceeds one block at a time, reducing the opportunity to amortize packing overhead.

Bitslice vs. fixslice. While fixslicing outperforms bitslicing for AES, in R256 bitslice surpasses fixslice on every metric—10% faster encryption, 1.7× faster key schedule. The direct cause of this reversal is the increased per-round instruction count. Each fixslice MixColumns variant, including the pre/post byte rotations for column alignment, consumes roughly 176 instructions on average, whereas classical bitslice handles the same work with ShiftRows (96) + MixColumns (35; `xtime`-based XOR chain across eight bit-planes) ≈ 131 instructions. Over 14 rounds, this difference accumulates to produce the observed reversal in encryption cycles. The key schedule cost also increases by 1.7× because round keys must be precomputed separately for each of the eight variants. The code footprint tells the same story: fixslice requires 8,924 bytes of ROM versus 3,352 bytes for bitslice (2.7×), because the eight MixColumns variants are each inlined as distinct code sequences. Bitslice is also more compact than T-table C (5,972 B), whose ROM is dominated by 4 KB of lookup tables. The higher peak stack usage of the bitsliced variants (576–696 B vs. 76–240 B) stems from SWAPMOVE-based round-key packing during key schedule. For the selected constant-time Cortex-M4 path, the 576 B peak stack is therefore a deliberate memory-for-CT trade-off relative to the non-CT T-table ASM baseline (76 B), rather than an accidental overhead. It is below 0.5% of the 128 KB main SRAM region our linker maps on the STM32F407. For deployments of the PQC schemes considered in §6 on Cortex-M4-class devices, this few-hundred-byte cipher stack should be evaluated together with the scheme-level working memory—seed trees, party states, and signatures—which is typically multi-kilobyte, so the R256 core is unlikely to be the dominant memory constraint.

This confirms that fixslicing is tightly coupled to AES's uniform inter-row ShiftRows differences. For R256 on Cortex-M4 the robust choice is therefore wider classical bitslicing with an explicit ShiftRows permutation. We return to this cross-platform portability pattern in §7.

4 ARMv8-A: Pre-shuffling for AES-Instruction Reuse

The ARM Cryptographic Extension (CE) provides dedicated instructions for accelerating AES operations: `AESE` (AddRoundKey+SubBytes+ShiftRows), `AESD` (InvSubBytes+InvShiftRows), `AESMC` (MixColumns), and `AESIMC` (InvMixColumns). Although `AESE` performs AddRoundKey, SubBytes, and ShiftRows in a single instruction, ShiftRows$(0, 1, 2, 3)$ is hardwired. Because R256's offsets $(0, 1, 3, 4)$ differ, applying the instruction directly produces incorrect shifts in rows 2 and 3.

We present a pre-shuffle technique that correctly implements R256's ShiftRows without modifying the hardware ShiftRows behavior.

4.1 Pre-shuffle Design

Position independence of SubBytes. The key insight behind the pre-shuffle is that SubBytes substitutes each byte independently of its position in the state matrix. Internally, `AESE` first applies AddRoundKey, then SubBytes, and finally ShiftRows$(0, 1, 2, 3)$. By rearranging the bytes *before* passing the state to `AESE`, the result of the hardware ShiftRows$(0, 1, 2, 3)$ can be made to coincide with R256's ShiftRows$(0, 1, 3, 4)$. Because

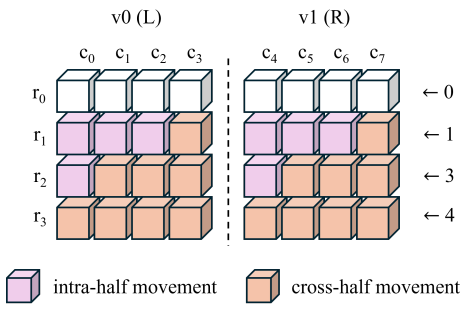


Figure 2: L/R partition of the Rijndael-256 state and the byte-movement pattern under ShiftRows$(0, 1, 3, 4)$. Orange cells indicate bytes that cross the L↔R boundary.

Listing 1: Pre-shuffle masks (encryption)

```
// L_output: byte indices composing the new L half
mask_for_L:
    .byte 0, 17, 22, 23,  4, 5, 26, 27,
          8, 9, 14, 31,  12, 13, 18, 19
// R_output: byte indices composing the new R half
mask_for_R:
    .byte 16, 1, 6, 7,  20, 21, 10, 11,
          24, 25, 30, 15,  28, 29, 2, 3
```

SubBytes is insensitive to byte positions, applying this pre-shuffle and then invoking `AESE` suffices to implement a correct R256 round.

Splitting the 256-bit state. The 4×8 state matrix of R256 is distributed across two 128-bit NEON registers: `v0` (L: columns 0–3) and `v1` (R: columns 4–7). The split is column-wise because `AESMC` applies MixColumns independently to each 4-byte column within a 128-bit register, so every column must reside entirely in one register. A row-wise split would make `AESMC` inapplicable.

A consequence of this column-wise split is that when ShiftRows$(0, 1, 3, 4)$ is applied, some bytes in rows 1–3 must move across the L/R boundary (Figure 2). Shuffles confined to a single 128-bit register cannot handle these cross-boundary movements.

Implementation via ARM `TBL`. The ARMv8-A `TBL` instruction can treat up to four consecutive registers (64 bytes) as a single lookup table. When a two-register table `{v0, v1}` (32 bytes) is specified, indices 0–15 select bytes from `v0` and indices 16–31 from `v1`. Cross-boundary byte movements between L and R are therefore handled in a single instruction. This contrasts with x86's `PSHUFB`, which operates only within a 128-bit lane and thus requires two shuffles and a `POR`.

The permutation masks are precomputed and loaded as constants; Listing 1 shows the encryption masks.

Index values of 16 or above select bytes from `v1` (R), while those below 16 select from `v0` (L). After applying these masks, the subsequent `AESE` performs ShiftRows$(0, 1, 2, 3)$, and the final result is identical to having applied ShiftRows$(0, 1, 3, 4)$.

MixColumns compatibility. Because the column-wise split is preserved through the pre-shuffle and `AESE`, `AESMC` applies exactly as in standard AES, with no extra processing for MixColumns.

Single-round structure. Listing 2 shows a complete R256 round based on the pre-shuffle technique. Registers `v16` and `v17` hold the precomputed masks, and `v20` and `v21` hold the pre-shuffled round-key halves.

Listing 2: One non-final folded R256 round (ARM Cryptographic Extension)

```
// --- Pre-shuffle: Rijndael-256 ShiftRows correction ---
TBL  v2.16b, {v0.16b, v1.16b}, v16.16b  // rearrange L half
TBL  v3.16b, {v0.16b, v1.16b}, v17.16b  // rearrange R half

// --- AESE: AddRoundKey(Kpre_i) + SubBytes + ShiftRows(0,1,2,3) ---
AESE v2.16b, v20.16b                    // v20 = Kpre_i (L)
AESE v3.16b, v21.16b                    // v21 = Kpre_i (R)

// --- AESMC: MixColumns (column-independent, no modification) ---
AESMC v0.16b, v2.16b
AESMC v1.16b, v3.16b
// final round omits AESMC; adds K_14 via a natural-order EOR (see
    text)
```

Round-key injection. `AESE` bundles AddRoundKey→SubBytes→ShiftRows, placing its internal $\oplus$ *before* SubBytes. In our pipeline that position is reached immediately after the `TBL` pre-shuffle, where the state still sits in the "pre-rotated" column order that `AESE`'s internal ShiftRows has yet to unwind. The natural-order round key is therefore column-misaligned with the state at that point, and XOR-ing it there would apply key bytes to state bytes of the wrong column; Figure 3(b) marks this rejected injection with a red cross and shows the EOR correction. The *EOR form* disables `AESE`'s internal AddRoundKey ($\oplus\,\mathbf{0}$) and applies K_i via a separate `EOR` after `AESMC`, where the state is back in natural column order. The byte-level key schedule is unchanged. The *folded form* instead pre-shuffles each round key once ($K_i^{\mathrm{pre}} = \texttt{TBL}(K_i)$, an offline per-key cost) and feeds it to `AESE`'s internal AddRoundKey. Since that step precedes `AESE`'s ShiftRows and `TBL` distributes over $\oplus$, the result is bit-exact (KAT-verified), leaving only the final key K_{14} as a natural-order `EOR`, as in standard AES on ARM. Folding removes the per-round `EOR` from the latency-bound `TBL`→`AESE`→`AESMC` chain and is our primary bulk, key-reuse path: on Apple M2 it lifts single-block throughput from 2193.9 MB/s to 4196.7 MB/s ($\approx 1.9\times$), and to 6520.0 MB/s with four-block interleaving. When the key changes every block—the per-node rekeying of the §6 PQC integration—the one-time K^{pre} is paid per key and interleaving does not apply, so we keep the EOR form there.

4.2 NEON Software Fallback

Not all ARMv8-A processors include the CE. The Raspberry Pi 4 (Cortex-A72) is a notable example: it supports the ARMv8-A instruction set but lacks the AES/SHA hardware instructions. For such environments, we designed a software-only path that operates exclusively with NEON SIMD instructions.

`TBL`/`TBX`-based S-box lookup. The R256 S-box, which is identical to the AES S-box, is loaded into vector registers (256 bytes), and `TBL`/`TBX` instructions perform parallel substitution in 16-byte units. Because the S-box resides in registers, no data-dependent memory accesses occur, so the software path avoids secret-dependent branches and table lookups while also reducing latency compared to memory lookups. Since ARMv8-A's `TBL` can use at most four registers (64 bytes) as a table, the 256-byte S-box is processed in four stages (Listing 3):

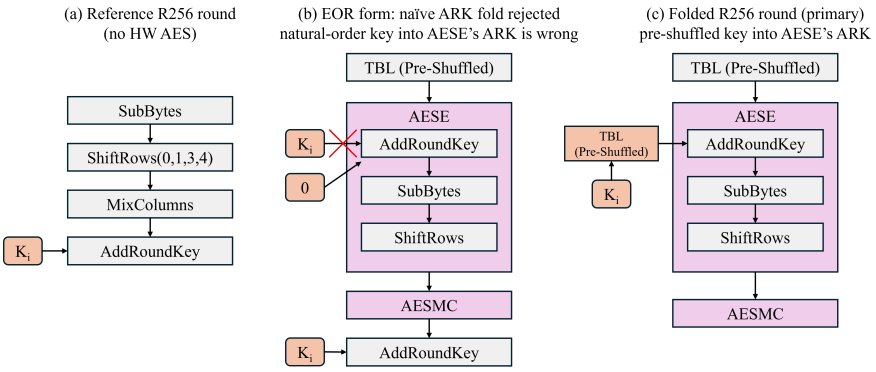


Figure 3: Round-key injection on AArch64 for a non-final R256 round: (a) the reference round without hardware AES, (b) the EOR correction, with the rejected natural-order key injection into `AESE`'s AddRoundKey marked by a red cross, and (c) the folded bulk/key-reuse design using per-key pre-shuffled round keys.

Listing 3: TBL/TBX-based S-box lookup (16-byte unit)

```
.macro sbox_byte16 reg
    SUB  v30.16b, \reg, v12         // idx - 64
    TBL  \reg.16b, {v8-v11}, \reg   // range [0, 63]
    SUB  v31.16b, v30, v12          // idx - 128
    TBX  \reg.16b, {v16-v19}, v30   // range [64, 127]
    SUB  v30.16b, v31, v12          // idx - 192
    TBX  \reg.16b, {v20-v23}, v31   // range [128, 191]
    TBX  \reg.16b, {v24-v27}, v30   // range [192, 255]
.endm
```

`TBL` returns zero for out-of-range indices, whereas `TBX` preserves the existing value. By exploiting this difference, a four-stage `TBL`+`TBX` chain covers the full 256-byte range. Register `v12` holds the constant `0x40` (64) used for index adjustment.

Four-block parallel processing. When processing a single block, the cost of loading the 16 vector registers for the S-box lookup dominates execution time. To amortize this cost, we encrypt four plaintext blocks simultaneously under the same key. The four-block count is set by NEON's 32-register budget: eight registers hold the four block states and the remaining 24 cache the 256-byte S-box, ShiftRows masks, MixColumns temporaries, and round keys. Five blocks would leave too little scratch, while three or fewer under-utilize the pipeline.

A common alternative interleaves the four blocks' matching columns into a single register (NEON IL 4PT, for *interleaved lanes*), so SubBytes and MixColumns cover all four blocks per instruction. For R256 this backfires: the per-row ShiftRows$(0, 1, 3, 4)$ rotations become a serial `BIT` chain, AddRoundKey needs `DUP` broadcasts, and input/output layout conversions add further per-round overhead.

Our implementation (NEON 4PT) instead places the four blocks in independent register pairs (`v0`–`v1`, `v2`–`v3`, ...). ShiftRows for each block is handled by `TBL` (4 instructions per block), and AddRoundKey XORs the same round key directly into all four blocks, eliminating the need for broadcasts. Because there are no data dependencies among the four blocks, wide out-of-order pipelines such as Apple M2's naturally issue instructions from different blocks in parallel across superscalar execution units. To facilitate this, the code interleaves corresponding operation phases of all four blocks within each round, maximizing instruction-level parallelism (ILP).

Table 3: Rijndael-256 versus hardware AES baselines on AArch64 and x86. AArch64 entries report MB/s, whereas x86 reports cycles per byte (cpb). Only the unitless slowdown rows are intended for cross-platform comparison.

	Apple M2 (MB/s)	Cortex-A76 (MB/s)	x86-64 (cpb)
Best interleave (M2 N=4, A76 N=2, x86 16-block VAES-512)			
AES-128	15,252.6	3,441.0	0.188
AES-256	11,721.6	2,435.6	0.234
R256	6,520.0	1,014.1	0.255
slowdown vs. AES-256	1.80×	2.40×	1.09×
slowdown vs. AES-128	2.34×	3.39×	1.36×
Single-block, MB/s (AArch64 only)			
R256 folded	4,196.7	608.3	—
R256 `EOR`	2,193.9	509.1	—

As a result, NEON 4PT runs about 12% faster than NEON IL 4PT on Apple M2. The per-round S-box loading cost is identical in both. Extensions to multi-key settings and results for PQC algorithm integration are discussed in §6.

4.3 Performance Analysis

Table 3 compares R256 against AES on the two ARM Cryptographic Extension (CE) cores we test, Apple M2 and Cortex-A76, and includes an x86-64 VAES-512 reproduction of [DG22] only for unitless AES→R256 slowdown comparison. AES-128 uses 10 rounds, whereas AES-256 and R256 use 14 rounds. We therefore use AES-256 as the round-matched baseline and report AES-128 as an additional reference point. Cores without the CE (e.g. Cortex-A72) are discussed separately below. On AArch64, macOS restricts PMU access, making direct per-block cycle measurements difficult. We therefore report best-of-10 throughput (MB/s) obtained by encrypting 32 MB (1M blocks). Since all key sizes yield $N_r = 14$, the 256-bit key result serves as the representative figure. The M2 uses Apple clang 17.0 (`-O3 -march=armv8-a+crypto`). The Cortex-A76 (Raspberry Pi 5) and Cortex-A72 (Raspberry Pi 4) use gcc with the matching `-march`. Throughputs exclude the one-time key setup, including the folded path's per-key pre-shuffle.

Apple M2 (with CE). Folding makes the pre-shuffle+`AESE` path our fastest constant-time option (Table 3): against round-matched AES-256 on the same `AESE`/`AESMC` hardware its per-byte slowdown is only 1.80× (2.34× vs. AES-128), so the two `TBL` pre-shuffles are essentially the entire structural cost of the wider state. Where the CE is absent, a `TBL`/`TBX`-based S-box gives a constant-time software fallback: NEON 4PT reaches 480 MB/s on M2, and four-block batching alone raises it **4.0**× over single-block NEON (120 MB/s) with no algorithmic change, since the blocks are independent.

Beyond Apple: Cortex-A76 (Raspberry Pi 5). The same ARM Crypto path runs unmodified on the Cortex-A76, the first non-Apple CE core we test. Folded R256 reaches 1014.1 MB/s at its best depth (N=2), a per-byte slowdown of 2.40× over AES-256 (3.39× vs. AES-128) — larger than M2's 1.80× because the A76's narrower NEON issue contends more with the two `TBL` pre-shuffles. That an unrelated CE core exhibits the same qualitative cost indicates it is structural rather than Apple-specific.

Without the CE (Cortex-A72). On cores lacking AES hardware the `AESE` path is unavailable; there T-table is fastest (54 MB/s, 2.1× NEON 4PT's 26 MB/s), because the A72's narrow pipeline and small L1 cache bottleneck on the seven-instruction `TBL`/`TBX` S-box chain whereas T-table folds SubBytes+ShiftRows+MixColumns into one cached lookup. NEON 4PT remains the constant-time choice when T-table's data-dependent lookups are unacceptable. The ARM Crypto and NEON paths are constant-time (only `AESE`/`AESMC`/`TBL`/`TBX` on register-resident data). The T-table and Reference baselines use memory-resident lookups and are not.

All implementations presented in this section target encryption only, which is the operation required by the PQC use cases in §6. The decryption path follows analogously using `AESD`/`AESIMC` with the corresponding inverse pre-shuffle mask.

5 CUDA: Amortizing the Mismatch via Shared-Memory T-tables

This section presents a T-table-based CUDA implementation of R256 and analyzes the performance trade-offs arising from different shared-memory layouts.

We include CUDA to complement the CPU targets with a massively parallel platform. Because the CUDA programming model on our target exposes no AES-round primitive analogous to `AESE` or AES-NI, shared-memory T-tables provide an established SIMT implementation strategy [Man07, Tez21], with each thread encrypting an independent block. These kernels are not CT under §2.5 and serve only as a non-CT throughput reference point, not as side-channel-secure deployment implementations. Because the R256 state expands to 32 bytes ($N_b = 8$), the number of T-table lookups per round doubles from 16 (AES) to 32. The T-tables themselves are identical to those of AES; only the column offsets need to be adjusted to reflect R256's ShiftRows$(0, 1, 3, 4)$. The increased number of lookups amplifies the impact of bank conflicts in shared memory—the serialization that occurs when multiple threads access the same memory bank simultaneously. How this issue is addressed gives rise to two distinct layouts: a *Compact* layout that minimizes shared-memory usage at the cost of tolerating bank conflicts, thereby achieving high occupancy, and a *Conflict-free* (CF) layout that replicates the table to eliminate bank conflicts entirely at the cost of reduced occupancy.

All experiments were conducted on an NVIDIA GeForce RTX 4080 (Ada Lovelace, 76 SMs), compiled with nvcc 12.0 and `-O3 -arch=sm_89`. Since $N_r = 14$ is the same for all key sizes, we report the 256-bit key results as representative.

5.1 Shared-Memory Layouts

Compact layout. The Compact layout places the four T-tables (`Te0`–`Te3`) and the four last-round tables directly in shared memory, consuming $8 \times 256 \times 4\,\mathrm{B} = 8\,\mathrm{KiB}$ per block. The 32 lookups per round in R256 and its non-uniform ShiftRows offsets raise the probability of bank conflicts relative to AES, but the key advantage of the Compact layout is its high occupancy. Occupancy denotes the fraction of threads that can be simultaneously active on a Streaming Multiprocessor (SM); higher occupancy enables the hardware to hide memory-access latency by overlapping it with computation from other threads. Each SM of the RTX 4080 provides up to 100 KiB of shared memory, so the 8 KiB footprint of the Compact layout allows many blocks to reside on a single SM concurrently. Peak throughput was achieved with 128 threads/block, yielding 11 blocks per SM and a measured occupancy of 86.9% (Nsight Compute).

Conflict-free (CF) layout. The CF layout replicates the `Te0` table into a 256×32 two-dimensional array so that each thread within a warp accesses its own dedicated column,

Listing 4: Eliminating CF initialization bank conflicts: naive vs. staggered store

```
// naive: all 32 warp threads store to column b simultaneously
//        -> 32-way bank conflict every iteration
for (b = 0; b < 32; b++)
    Te0_s[i][b] = val;

// staggered: each thread starts at a different column
//        -> 32 threads spread across 32 distinct banks
int lane = threadIdx.x & 31;
for (b = 0; b < 32; b++)
    Te0_s[i][(b + lane) & 31] = val;
```

Table 4: CUDA kernel throughput: Compact, CF naive, and CF staggered (256-bit key, RTX 4080, 128 thr/blk)

Mode	Compact (GiB/s)	CF naive (GiB/s)	CF staggered (GiB/s)
ECB (kernel)	79.93	52.84	81.16
CTR (kernel)	100.57	56.10	103.74
ECB (end-to-end)	5.22	5.26	5.41
CTR (end-to-end)	5.54	5.10	5.53

thereby eliminating bank conflicts during the *lookup* (load) phase. The remaining tables (`Te1`–`Te3`) are derived by applying byte rotations to the `Te0` lookup results. Per-block shared-memory usage is 32 KiB (Te0) + 4 KiB (last round) = 36 KiB, which limits each SM to at most 2 resident blocks and substantially reduces occupancy.

Hidden cost of CF: initialization bank conflicts. At kernel launch, the CF layout replicates the T-table into shared memory via a loop in which each thread writes the same value across all 32 columns. Because `Te0_s` is stored in row-major order, the bank number of `Te0_s[i][b]` depends solely on the column index b. When all 32 threads in a warp store to the same column b simultaneously, a 32-way bank conflict occurs. Nsight Compute measurements (Table 5) reveal **65.5 million** bank conflicts originating from this initialization phase alone.

The staggered store in Listing 4 writes the same value to all 32 entries, so the column order is immaterial to correctness; however, in any given cycle the 32 threads now access 32 distinct banks, completely eliminating the conflicts.

5.2 Performance Analysis

Table 4 compares the throughput of the Compact and CF layouts. Kernel throughput reflects the raw computational performance measured from GPU kernel execution time alone, while end-to-end throughput includes host–device PCIe transfers. All measurements use 100 MiB of data and report the average of 10 runs preceded by 3 warm-up iterations.

Tables 4 and 5 report only CUDA variants analyzed in this paper; no external baselines are included. These kernels are throughput-oriented shared-memory T-table implementations and are not constant-time under the definition of §2.5, because table indices and memory-access patterns depend on plaintext and key bytes. Table 5 presents kernel-level profiling data collected with NVIDIA Nsight Compute; all measurements use a 128 threads/block configuration, and "CF (staggered)" denotes results after applying the initialization bank-conflict elimination described above.

Table 5: Nsight Compute profiling of the same three CUDA kernels. 1M blocks (32 MiB), 256-bit key, 128 thr/blk.

Kernel	smem/blk	Occ. (%)	Bank Confl.	SM Cyc.	Exec. Instr.	IPC
Compact	8 KiB	86.9	37,303	1,109,657	66,027,520	59.5
CF (naive)	36 KiB	16.2	65,533,970	1,716,710	90,112,000	52.5
CF (staggered)	36 KiB	15.6	389,098	1,077,040	94,699,520	87.9

CF (naive) is free of bank conflicts during T-table lookups, yet incurs 65.5 million store-side bank conflicts while replicating the T-table into shared memory, driving SM cycles to 1.55× those of Compact. After applying the staggered store, bank conflicts drop by 99.4% (65.5M → 0.39M) and SM cycles improve by 1.59×, bringing CF *below* Compact (0.97×). In kernel throughput this reverses the naive layout's 1.5–1.8× deficit, lifting CF above Compact in both ECB and CTR (Table 4).

Notably, despite only 15.6% occupancy and ≈43% more instructions, resolving the bank conflicts alone causes the *instructions-per-cycle* rate (IPC; warp-level instructions retired per SM cycle, Table 5) to surge from 52.5 to 87.9, surpassing Compact's IPC of 59.5. This result demonstrates that **bank conflicts, not occupancy, were the dominant bottleneck**.

Table 4 also reports CTR-mode throughput. The CTR kernel generates counter blocks on the GPU and XORs the resulting ciphertext with the plaintext. The counter input to the round function is generated with register arithmetic, while the plaintext is still read for the final XOR. This decoupling is the source of the speedup: the cipher input is the register-resident counter rather than a global-memory load, so the plaintext fetch—needed only for the final XOR—overlaps with computation, whereas in ECB the plaintext block must be loaded before the round function can begin. Under this benchmark, CTR throughput exceeds ECB by 25–28% for Compact and CF staggered. For CF naive the gain is only 6%, because initialization bank conflicts remain the dominant bottleneck.

When host–device transfers are included, the end-to-end throughput gap narrows substantially: all configurations fall within the 5.1 GiB/s to 5.6 GiB/s range, as PCIe transfer time dominates overall execution time.

6 Evaluation

The optimized R256 implementations presented in §3–5 are valuable as standalone block-cipher libraries, but their practical impact on real cryptographic protocols must be quantified. As noted in §2.4, several NIST PQC signature candidates use R256 as an internal primitive; we therefore integrate our implementations into these algorithms and measure the end-to-end performance change.

Standalone R256 benchmarks and experimental setups for each platform are already described in the respective sections (§3–§5). The PQC integration experiments in this section were conducted only on Apple M2, using the same toolchain as §4.3. Accordingly, this section should be read as an AArch64 integration study, not as a cross-platform protocol benchmark.

6.1 PQC End-to-End Performance Impact

Target algorithms. Of the second-round candidates in the NIST signature call [Nat22], four use R256 as an internal symmetric primitive: SDitH [MFG+23], Mirath [AAB+25], RYDE [ABB+25], and MQOM [BFR24]. All four employ R256 at Level 3 (192-bit security) and above and follow the MPC-in-the-Head paradigm or closely related frameworks. We

exclude FAEST [BBB+25], which also uses R256 but invokes it only once per sign/verify operation (R256 fraction effectively 0%).

The baseline code for each benchmark is the reference implementation submitted to NIST by the respective algorithm team; the R256 cryptographic core was replaced with three implementation variants (Reference C, NEON, ARM Crypto), with minimal benchmark or interface adaptations where required by each codebase. The measurement units follow those adopted by each reference implementation: SDitH and MQOM report wall-clock time (ms), while Mirath and RYDE report CPU cycles. Because the units differ, direct cross-algorithm comparison of absolute figures is not meaningful; instead, we evaluate performance relative to the Reference baseline within each algorithm.

R256 fraction. The fraction of total execution time spent in R256 calls (R256%) was measured using the ARM cycle counter (`CNTVCT_EL0`). Under the same `-O3` optimization level used for all benchmarks, the `noinline` attribute was applied solely to the R256 encryption function (`rijndael256_encrypt`) to preserve the function boundary, while the rest of the code received full optimization including inlining. Cycle counters were inserted at function entry and exit to accumulate R256 cycles (excluding the key schedule), and R256% was computed as the mean over 10 repeated measurements. We therefore interpret R256% as the fraction attributable to block-encryption calls under the benchmarking harness, rather than as a complete decomposition of all symmetric-primitive costs. Because R256% varies with the parameter set, security level, and operation (sign vs. verify), per-row values are reported in Table 6. For SDitH, GF(2^{192}) field arithmetic dominates the overall execution, resulting in a low R256 fraction of 16–17%. The remaining three algorithms implement their internal hash and PRG almost entirely with R256, reaching above 96% in all configurations.

Integration and results. All four algorithms store round keys in a 480-byte array whose memory layout matches our key structure, so the existing key schedule output can be used directly without conversion. In this section, "NEON" and "ARM Crypto" (the `EOR` variant of §4) refer specifically to the two AArch64 implementation variants from §4. We use the `EOR` variant here rather than the folded one of §4: because these schemes rekey per seed-tree node, the folded path pays its one-time K^{pre} pre-shuffle on every key and cannot use block interleaving, which we measured to make it 8–29% slower on the Rijndael portion (up to ∼12% end-to-end for MQOM) — so the `EOR` path is the right default here. MQOM [BFR24] requires simultaneous encryption under four distinct keys during signing; we addressed this by extending the NEON 4-block parallel structure from §4 to a multi-key variant. For every algorithm, the KAT (Known Answer Test) outputs of the Reference, NEON, and ARM Crypto implementations were verified to be bit-identical.

Table 6 presents the signing and verification performance. "Reference" denotes the NIST-submitted implementation; "NEON" and "ARM Crypto" denote the results obtained by replacing the R256 cryptographic core in the reference implementations with our optimized versions, together with minimal integration adjustments where needed. For RYDE, the NEON column reports the single-block variant. Unlike MQOM, where a dedicated multi-key NEON variant handles the four simultaneous keys, RYDE uses a distinct key for each GGM-tree node, so our shared-key four-block NEON path is not applicable. Single-block NEON is about 3.5× slower than ARM Crypto on the Rijndael portion, so ARM Crypto remains the recommended path, although both far outpace the reference. For Mirath and RYDE verification, the number of R256 calls differs from signing, which is why the speedup ratios are higher than for signing.

On Apple M2, the end-to-end speedup is strongly governed by the R256 fraction. For MQOM, RYDE, and Mirath, where R256 accounts for over 96% of execution time, replacing the R256 cryptographic core in the reference implementations yields 30–114× speedup

Table 6: PQC signature algorithm performance (Apple M2). SDitH/MQOM are in ms; Mirath/RYDE are in Mcyc (10^6 cycles). ARM× is the speedup of ARM Crypto over Reference within each reference codebase; ratios are computed from unrounded measurements.

Algorithm	Lv.	R256%	Reference	NEON	ARM Crypto	ARM×
Sign						
SDitH	3	16.6%	263.6	235.9	223.6	1.18×
SDitH	5	16.3%	402.8	373.3	342.6	1.18×
Mirath	3	98.2%	46.6	1.18	0.91	51.4×
Mirath	5	96.8%	65.5	2.56	2.15	30.5×
RYDE	3	99.2%	57.5	0.88	0.54	106.2×
RYDE	5	98.3%	72.0	1.69	1.27	56.6×
MQOM	3	99.4%	2,861	26.9	25.0	114.3×
MQOM	5	99.1%	3,990	49.5	46.9	85.0×
Verify						
SDitH	3	17.0%	247.6	220.5	206.9	1.20×
SDitH	5	17.0%	372.7	347.4	314.2	1.19×
Mirath	3	99.3%	76.2	1.06	0.57	133.2×
Mirath	5	98.9%	105.8	1.91	1.24	85.5×
RYDE	3	99.3%	57.0	0.81	0.47	121.9×
RYDE	5	98.6%	70.1	1.47	1.04	67.2×
MQOM	3	99.6%	2,823	21.0	18.1	155.6×
MQOM	5	99.5%	3,953	35.2	31.5	125.5×

Table 7: Summary of platform-specific Rijndael-256 adaptation strategies. (CE: ARM Crypto Extensions; NEON: ARM NEON SIMD.)

Platform	Adaptation	Repr. Result	Limitation
Cortex-M4	Wider classical bitslicing	4,962 cyc/blk	No interleaving
AArch64 (CE)	Reuse + reshuffle	6,520 MB/s (N=4)	+2 `TBL`/round; folded ARK
AArch64 (NEON)	`TBL`/`TBX` S-box	480 MB/s (4PT)	16 vector regs
CUDA	Amortize mismatch	81.16 GiB/s	36 KiB smem; low occ.

for signing and up to 155.6× for verification. In contrast, SDitH achieves only about 1.2× because its R256 fraction is 16–17%. In SDitH, non-R256 computation (GF(2^{192}) field arithmetic, MPC simulation, hashing, etc.) accounts for roughly 83% of execution time, so by Amdahl's law the overall speedup is bounded at approximately 1.2× regardless of how fast R256 becomes. MQOM's Reference is especially slow because its constant-time core is bitsliced. Our ARM Crypto path reaches the same software-level constant-time guarantee through AES instructions while gaining two orders of magnitude.

For most algorithms the Level 5 speedup ratio is lower than Level 3, since larger parameters raise the non-R256 share, though the absolute time savings are larger at Level 5.

7 Discussion and Conclusion

Table 7 contrasts the three platform-specific adaptation strategies, together with the AArch64 NEON software fallback, their representative results, and their limitations. Performance units follow platform conventions and are not intended for cross-platform comparison.

Transferability and coupling. Across the platforms studied, AES optimizations carry over to R256 only when the target can cheaply repair the structural assumption they break. Fixslicing assumes uniform inter-row ShiftRows differences. R256's $(0, 1, 3, 4)$ offsets violate that assumption with no cheap repair, so on Cortex-M4 the robust choice is wider classical bitslicing (§3). Hardware AES rounds are coupled mainly to fixed ShiftRows wiring: because SubBytes is position-independent, a pre-shuffle restores the permutation while reusing the instruction—"Reuse with Reshuffling" on AArch64 (§4), where folded AddRoundKey reaches 6520 MB/s, and, with a single `VPERMB`, on the x86-64 VAES path below. On CUDA, the non-CT T-table path is only a throughput reference. Massive parallelism and conflict-aware shared memory amortize the wider state's doubled lookups—"Amortize the Mismatch" (§5). Thus the useful question is which structural assumption breaks and whether the platform can absorb the correction.

x86-64. For cross-platform context we reproduce the x86-64 approach of Drucker and Gueron [DG22]: a byte shuffle corrects the hardwired ShiftRows of `AESENC`, allowing reuse of the hardware AES round. Our VAES-512 version applies one `VPERMB` pre-shuffle per round and keeps 16 blocks in flight. Unlike 128-bit `PSHUFB`, AVX-512 VBMI `VPERMB` can express the correction as one wider byte permutation. On an Intel i7-1165G7 it runs at 0.255 cpb—a per-byte slowdown of 1.09× over round-matched AES-256 and 1.36× over AES-128 (Table 3)—consistent with the ∼0.27 cpb reported in [DG22] on different silicon. We reproduce only their method and make no x86 state-of-the-art claim. VAES-512 amortizes the correction across four 128-bit AES blocks, whereas 128-bit `AESE` processes one block and issues two `TBL` per round. Hence best-path AES-256→R256 slowdowns rise from 1.09× on x86 to 1.80× on M2 and 2.40× on Cortex-A76, indicating structural rather than Apple-specific overhead.

Masking. Our constant-time claim is at the software level (§2.5). Physical-side-channel masking is out of scope. A Boolean-masked Cortex-M4 variant would add eight bitplane registers per share, forcing substantial spilling because the unmasked state already occupies eight of the 14 usable registers (§3.2).

CUDA. As a non-constant-time throughput reference point, the GPU absorbs the wider state's doubled lookups through shared-memory layout: staggered stores remove 99.4% of replicated-table initialization bank conflicts, improving the conflict-free layout by 1.59× to just below Compact (0.97×), so bank conflicts—not occupancy—are the dominant bottleneck (§5).

PQC end-to-end impact. As §6 shows, these primitive-level gains translate end-to-end only when R256 dominates—near-linear for MQOM, RYDE, and Mirath (R256 > 96%; up to 155.6×), but Amdahl-bounded for SDitH (16–17%).

7.1 Conclusion

In short, R256 is not a drop-in widening of AES: portability depends on which AES-specific structural assumption breaks and how cheaply each platform repairs it. End-to-end, replacing only R256 in four PQC reference codebases on Apple M2 yields up to 155.6× speedup, scaling with each scheme's R256 fraction. Future work includes masking, Arm SVE back ends, RISC-V vector-crypto implementations and benchmarks building on Saarinen's instruction-level analysis [Saa25], and Acc256/Accordion support as standardization progresses. All implementations and measurement data from this paper are publicly available.[1]

[1] https://github.com/ShuRaAtum/rijndael-256-tches

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2025-02306395, Development and Demonstration of PQC-Based Joint Certificate PKI Technology, 40%) and this work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No. RS-2025-25394739, Development of Security Enhancement Technology for Industrial Control Systems Based on S/HBOM Supply Chain Protection, 40%) and this work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2026-25519543, Development of PQC Migration Technology for Quantum-Safe IC Chip, 20%).

References

[AAB+25] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, et al. Mirath signature scheme, 2025.

[ABB+25] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Romaric Neveu, Matthieu Rivain, et al. RYDE signature scheme. *Second Round Submission to the Additional NIST Post-Quantum Cryptography Call*, 2025.

[AP21] Alexandre Adomnicai and Thomas Peyrin. 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.

[Arm23] Arm Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, 2023. DDI 0487.

[BBB+25] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, et al. FAEST v2: Algorithm specifications, 2025.

[BFR24] Ryad Benadjila, Thibauld Feneuil, and Matthieu Rivain. MQ on my mind: Post-quantum signatures from the non-structured multivariate quadratic problem. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 468–485. IEEE, 2024.

[Bih97] Eli Biham. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer, 1997.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES S-box. In *IFIP International Information Security Conference*, pages 287–298. Springer, 2012.

[DBN+01] Morris J. Dworkin, Elaine Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. Advanced encryption standard (AES). NIST FIPS Publication 197, 2001.

[DG22] Nir Drucker and Shay Gueron. Software optimization of Rijndael for modern x86-64 platforms. In *ITNG 2022 19th International Conference on Information Technology-New Generations*, pages 147–153. Springer, 2022.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[Gue09] Shay Gueron. Intel's new AES instructions for enhanced performance and security. In *International Workshop on Fast Software Encryption*, pages 51–66. Springer, 2009.

[HW07] Owen Harrison and John Waldron. AES encryption implementation and analysis on commodity graphics processing units. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 209–226. Springer, 2007.

[LSR+26] Andriani Adi Lestari, Suryadi, Kalamullah Ramli, Teddy Surya Gunawan, Esti Rahmawati Agustina, and Susila Windarta. Constant-time bitsliced Rijndael-256 on ARM Cortex-M4: On the limitations of fixslicing beyond AES-128. *IIUM Engineering Journal*, 27(2):340–362, 2026.

[Man07] Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 65–68. IEEE, 2007.

[MFG+23] Carlos Aguilar Melchor, Thibauld Feneuil, Nicolas Gama, Shay Gueron, James Howe, David Joseph, Antoine Joux, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, et al. The syndrome decoding in the head (SD-in-the-Head) signature scheme. *NIST Round 1 Submission to the Additional Call for Signature Schemes*, 2023.

[Nat22] National Institute of Standards and Technology. Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process. Call for Proposals, 2022. Published August 2022; https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf.

[Nat24] National Institute of Standards and Technology. PRE-DRAFT call for comments: NIST proposes to standardize a wider variant of AES. NIST SP 800-197 (Initial Preliminary Draft; PRE-DRAFT call for comments), 2024. Published December 23, 2024; https://csrc.nist.gov/pubs/sp/800/197/iprd.

[Nat25] National Institute of Standards and Technology. PRE-DRAFT call for comments: NIST launches development of cryptographic accordions. NIST SP 800-197A (Initial Preliminary Draft; PRE-DRAFT call for comments), 2025. Published June 6, 2025; https://csrc.nist.gov/pubs/sp/800/197/a/iprd.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[Saa25] Markku-Juhani O. Saarinen. Brief comments on Rijndael-256 and the standard RISC-V cryptography extensions. Cryptology ePrint Archive, Paper 2025/1198, 2025.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *International Conference on Selected Areas in Cryptography*, pages 180–194. Springer, 2016.

[Tez21] Cihangir Tezcan. Optimization of advanced encryption standard on graphics processing units. *IEEE Access*, 9:67315–67326, 2021.