https://doi.org/10.31436/iiumej.v27i2.4405

Constant-Time Bitsliced Rijndael-256 on ARM Cortex-M4: On the Limitations of Fixslicing Beyond AES-128

ANDRIANI ADI LESTARI[1], SURYADI MT[2], KALAMULLAH RAMLI[1*], TEDDY SURYA GUNAWAN[3], ESTI RAHMAWATI AGUSTINA[4], SUSILA WINDARTA[5]

[1]*Department of Electrical Engineering, Faculty of Engineering, Universitas Indonesia, Depok 16424, Indonesia*
[2]*Department of Mathematics, Faculty of Mathematics and Natural Sciences, Universitas Indonesia, Depok 16424, Indonesia*
[3]*Department of Electrical and Computer Engineering, Kulliyyah of Engineering, International Islamic University Malaysia (IIUM), Kuala Lumpur 50728, Malaysia*
[4]*National Cyber and Crypto Agency, Depok 16516, Indonesia*
[5]*Department of Cybersecurity, Politeknik Siber dan Sandi Negara, Bogor 16120, Indonesia*

**Corresponding author: kalamullah.ramli@ui.ac.id*

(Received: 15 March 2026; Accepted: 5 May 2026; Published online: 10 May 2026)

ABSTRACT: Wider-block ciphers are increasingly needed in high-volume applications, because 128-bit blocks in modes such as Galois/Counter Mode (GCM) limit each invocation to roughly 64 GiB of plaintext per key-nonce pair, forcing complex re-keying strategies. Rijndael-256, the 256-bit-block variant of Rijndael with a 256-bit key, has therefore attracted renewed interest as a natural wider-block companion to Advanced Encryption Standard (AES). At the same time, 32-bit ARM Cortex-M microcontrollers dominate the IoT and embedded landscape, yet, to the best of our knowledge, no constant-time software implementation of Rijndael-256 targeting this platform has been published. This paper addresses that gap. We present a constant-time bitsliced implementation of Rijndael-256 on the ARM Cortex-M4 and provide a systematic structural analysis explaining why fixslicing, the technique that achieves the best-known AES-128 performance on this platform, becomes suboptimal when applied to Rijndael-256. Specifically, the irregular ShiftRows offsets (0, 1, 3, 4) of Rijndael-256 break the uniform register rotation exploited by fixslicing, requiring eight distinct MixColumns compensation variants instead of four. We demonstrate that these compensation variants cost 3.00× as much as executing an explicit, in-place ShiftRows routing using ARM's bitfield instructions. Our macro-inlined assembly variant achieves 6,199 cycles (193.7 cycles/byte) at `-O2`, including packing and unpacking. We provide benchmarks across five compiler optimization levels, constant-time verification over 10^6 samples via DUDECT (maximum t-statistic well below the vulnerability threshold), and per-component cycle breakdowns, showing that the optimal bitslicing strategy is inherently cipher-specific and architecture-dependent.

ABSTRAK: Sifer blok yang lebih lebar semakin diperlukan dalam aplikasi berisipadu tinggi kerana blok 128-bit dalam mod seperti Galois/Counter Mode (GCM) mengehadkan setiap invokasi kepada kira-kira 64 GiB teks biasa bagi setiap pasangan kunci-nonce, sekali gus memerlukan strategi penukaran kunci yang lebih kompleks. Rijndael-256, iaitu varian Rijndael dengan blok 256-bit dan kunci 256-bit, telah menarik semula perhatian sebagai alternatif pelengkap berblok lebih lebar kepada Advanced Encryption Standard (AES). Pada masa yang sama, mikropengawal ARM Cortex-M 32-bit mendominasi landskap Internet of Things (IoT) dan sistem terbenam. Namun, setakat pengetahuan kami, tiada pelaksanaan perisian Rijndael-256 secara bitslicing dan masa-tetap yang menyasarkan platform ini telah diterbitkan. Makalah ini menangani jurang tersebut. Kami membentangkan pelaksanaan

Rijndael-256 secara bitslicing dan masa-tetap pada ARM Cortex-M4 serta menyediakan analisis struktur yang sistematik bagi menjelaskan mengapa fixslicing, iaitu teknik yang mencapai prestasi terbaik yang diketahui bagi AES-128 pada platform ini, menjadi kurang optimum apabila diterapkan pada Rijndael-256. Secara khusus, anjakan ShiftRows Rijndael-256 yang tidak seragam, iaitu (0, 1, 3, 4), mengganggu putaran daftar seragam yang dimanfaatkan oleh fixslicing, lalu memerlukan lapan varian pampasan MixColumns yang berbeza berbanding hanya empat. Kami menunjukkan bahawa varian pampasan ini memerlukan kos 3.00× lebih tinggi berbanding pelaksanaan penghalaan ShiftRows secara eksplisit di tempat menggunakan arahan bitfield ARM. Varian kod himpunan tersisip-makro kami mencapai 6,199 kitaran, bersamaan 193.7 kitaran/bait, pada tahap pengoptimuman `-O2`, termasuk proses pembungkusan dan penyahbungkusan. Kami turut menyediakan penanda aras merentasi lima tahap pengoptimuman pengkompil, pengesahan masa-tetap ke atas 10^6 sampel menggunakan DUDECT dengan statistik-t maksimum yang jauh di bawah ambang kerentanan, serta pecahan kitaran bagi setiap komponen. Dapatan ini menunjukkan bahawa strategi bitslicing yang optimum adalah khusus kepada sifer dan sangat bergantung pada seni bina perkakasan.

KEYWORDS: *ARM Cortex-M4; bitslicing; constant-time implementation; fixslicing; Rijndael-256.*

1. INTRODUCTION

The AES, based on the Rijndael cipher with a fixed 128-bit block size, has served as the foundation of symmetric cryptography since its standardization by NIST in 2001 [1]. However, the 128-bit block size imposes constraints that have become increasingly relevant for high-volume data processing. Modes of operation such as GCM are subject to per-invocation plaintext length constraints: GCM restricts secure plaintext processing to approximately $2^{39} - 256$ bits (roughly 64 GiB) per key-nonce pair [2, 3].

Beyond this limit, the security guarantees of the mode degrade, necessitating re-keying procedures that add complexity to protocol design. Wider-block ciphers address this by enabling birthday-bound modes such as Synthetic IV and Accordion [4]. These constraints have motivated renewed interest in wider-block ciphers. In August 2024, NIST indicated its intent to vet Rijndael with 256-bit blocks and a single 256-bit key for potential standardization, and in December 2024, formally requested public comments on its plan to develop a draft standard (SP 800-197) [5]. The comment period closed on June 25, 2025, and received submissions from both academic researchers and industry stakeholders [6]. This represents the first significant consideration of wider-block Rijndael variants for standardization.

Throughout this paper, we use "Rijndael-256" as shorthand for the specific Rijndael variant with a 256-bit block and a 256-bit key, which corresponds to $N_b = 8$ and $N_k = 8$, which is the configuration under consideration by NIST. Here, N_b denotes the number of 32-bit words in the block (state), and N_k denotes the number of 32-bit words in the key.

1.1. The 32-bit IoT Landscape

A central question for the practical deployment of Rijndael-256 is whether it can be implemented efficiently on resource-constrained processors. The ARM Cortex-M family of 32-bit microcontrollers is the dominant architecture in Internet of Things (IoT) and embedded systems. According to recent industry analyses, ARM Cortex-M cores command approximately 66% of the overall MCU instruction-set architecture share, and the global installed base of IoT-connected devices is forecast to surpass 40 billion units by 2030 [7, 8].

https://doi.org/10.31436/iiumej.v27i2.4405

These devices serve critical functions across industrial automation, smart grid infrastructure, medical monitoring, and automotive systems. Many operate on batteries for extended periods and process sensitive data that requires strong confidentiality guarantees. If Rijndael-256 is standardized, these devices will need constant-time implementations to resist timing side-channel attacks in protocols such as DTLS, secure firmware update, and sensor-to-gateway encryption.

While 64-bit application processors and emerging RISC-V cores with vector extensions are gaining traction in edge computing, the 32-bit Cortex-M class is expected to remain the dominant platform for constrained IoT endpoints for the foreseeable future [7]. The timing of this work is therefore important: the NIST consideration of Rijndael-256 creates a practical need for implementers to understand whether AES-optimized techniques transfer to wider-block Rijndael on microcontrollers. Existing AES/fixslicing studies primarily target AES-128 or AES-256 and do not quantify the structural cost introduced by Rijndael-256's 4×8 state and irregular ShiftRows offsets on scalar 32-bit ARM hardware. Despite this practical need, the published literature contains no constant-time Rijndael-256 implementation for any 32-bit ARM Cortex-M device. This paper addresses that gap.

1.2. Performance Challenges on 32-bit Hardware

Implementing Rijndael-256 on 32-bit hardware presents challenges beyond those encountered with AES-128. The cipher operates on a 4×8 byte state matrix, doubling the state size relative to AES. On the Cortex-M4, which provides 13 general-purpose registers, the 256-bit state consumes eight registers, leaving minimal headroom for intermediate computations.

The ShiftRows transformation uses offsets $(0, 1, 3, 4)$ for $N_b = 8$, which differ from AES-128's uniform $(0, 1, 2, 3)$ offsets. Recent work by Saarinen [6], submitted as a comment to the NIST standardization process, evaluated Rijndael-256 using the RISC-V Vector Cryptography extension (Zvkn). Saarinen reports that the instruction count for 1-kilobyte encryption is $2.66 \times$ higher than for AES-256, primarily because ShiftRows requires additional byte-shuffle operations that cross 128-bit lane boundaries in each round. Our results confirm that this bottleneck persists on scalar 32-bit hardware, but can be addressed effectively using the Cortex-M4's `BFI` (Bit Field Insert) and `UBFX` (Unsigned Bit Field Extract) bit-field instructions.

1.3. Contributions

Our contributions are as follows:

(1) To the best of our knowledge, we present the first constant-time bitsliced implementation of Rijndael-256 on the ARM Cortex-M4 microcontroller, scoped to single-block, encryption-only, software-only execution without a hardware cryptographic accelerator. Our macro-inlined assembly variant achieves 193.7 cycles/byte (6,199 cycles total) at `-O2`, while our function-call assembly variant achieves 203.7 cycles/byte (6,518 cycles), and our portable C variant achieves 311.6 cycles/byte. All assembly cycle counts include bitsliced state packing and unpacking.

(2) We provide a systematic structural analysis demonstrating why fixslicing [9], the state-of-the-art technique for AES-128, becomes suboptimal for Rijndael-256. We show that the irregular ShiftRows offsets $(0, 1, 3, 4)$ double the fixslicing period from $T = 4$ to $T = 8$, breaking the uniform register rotation that makes fixslicing efficient, and requiring eight distinct MixColumns compensation variants that cost on average $3.00 \times$ more than explicit ShiftRows with `BFI`/`UBFX`.

https://doi.org/10.31436/iiumej.v27i2.4405

(3) We provide comprehensive benchmarks across five compiler optimization levels (`-O0` through `-Os`), including a theoretical instruction-level cost analysis. This analysis reveals that ShiftRows achieves a stall factor of $1.02\times$ (near-theoretical), while MixColumns carries a $3.04\times$ stall penalty due to chained data dependencies. We further provide code size measurements, memory footprint comparisons, and empirical constant-time verification over 10^6 samples using Welford's online algorithm, passing the DUDECT statistical framework [10] with a maximum t-statistic well below the vulnerability threshold.

1.4. Paper Organization

The remainder of this paper is organized as follows. Section 2 reviews the Rijndael-256 specification, bitslicing techniques, and related work. Section 3 presents the analysis of why fixslicing becomes inefficient for Rijndael-256. Section 4 describes our implementation. Section 5 presents experimental results. Sections 6 and 7 provide the comparison with AES-128 and a broader discussion, respectively. Section 8 concludes the paper.

2. Background and Related Work

2.1. Rijndael-256 Specification

Rijndael-256 operates on a 4×8 byte state matrix, with 14 rounds, $N_b = 8$, and $N_k = 8$ [11]. Each round applies four transformations: SubBytes (identical S-box to AES), ShiftRows with row offsets $(0, 1, 3, 4)$, MixColumns over $GF(2^8)$ with the same polynomial as AES, and AddRoundKey. The final round omits MixColumns.

The key schedule for $N_k = 8$ applies SubBytes with RotWord and Rcon when $i \equiv 0 \pmod{N_k}$, where i denotes the word index in the expanded key, and additionally applies SubBytes (without rotation) when $i \equiv 4 \pmod{N_k}$. The expanded key comprises 15 256-bit round keys, totaling 480 bytes.

2.2. Bitslicing

Bitslicing, introduced by Biham for DES [12], represents an n-bit cipher state across n registers, where each register holds one bit position from the data. For Rijndael-256, the 256-bit state maps to eight 32-bit ARM registers. All operations are implemented as Boolean circuits, thereby inherently eliminating data-dependent memory accesses and their associated timing leakage.

The S-box is implemented as a Boolean circuit following Boyar and Peralta's depth-16 minimized design [13], with further optimization for ARM Cortex-M by Schwabe and Stoffelen [14].

2.3. Constant-Time Execution Model

In this paper, “constant-time” means that the implementation contains no data-dependent branches or memory accesses, addressing the cache-timing threat model demonstrated by Bernstein [15] and following the formal constant-time definition of Almeida *et al.* [16]. This work addresses timing side-channels only; power, electromagnetic, and fault attacks are outside the scope of the present implementation. All control flow and memory access patterns are independent of the plaintext, key, and intermediate state values. Residual cycle-count variation (as reported in Section 5) arises from microarchitectural effects, such as instruction-cache line alignment and flash prefetch behavior, rather than from data-dependent operations. The T-table

https://doi.org/10.31436/iiumej.v27i2.4405

implementations execute a fixed sequence of instructions (constant instruction count), but perform table lookups indexed by secret-dependent values; the resulting cache access patterns make their execution time data-dependent, i.e., not constant-time in the sense defined above. DUDECT [10] provides empirical evidence of no detectable timing leakage under the tested conditions; it is not a formal proof of constant-time execution.

2.4. Fixslicing for AES-128

Fixslicing, introduced by Adomnicai *et al.* [9, 17], absorbs the ShiftRows permutation into MixColumns by pre-rotating round keys to compensate for the accumulated ShiftRows offset. For AES-128 ($N_b = 4$, offsets $(0, 1, 2, 3)$), ShiftRows has period $T = 4$, meaning four distinct MixColumns variants suffice. On the Cortex-M4, each variant reduces to word-level rotations handled at zero cost by the barrel shifter. This approach achieves approximately 80 cycles/byte for two-block parallel AES-128 encryption [9].

2.5. ARM Cortex-M4 Architecture

The ARM Cortex-M4 is a 32-bit processor based on the ARMv7E-M architecture, featuring a single-cycle barrel shifter [18]. The `BFI` and `UBFX` instructions enable byte extraction and insertion in 1–2 cycles [18]. The STM32F407 variant used in our experiments features 1 MB of flash, 192 KB of SRAM, and an ART accelerator for zero-wait-state flash access at 168 MHz [19].

2.6. Related Work

Constant-time AES implementations on Cortex-M4 have been studied extensively. Schwabe and Stoffelen [14] presented an optimized bitsliced implementation achieving approximately 100 cycles/byte. Adomnicai and Peyrin improved this to approximately 80 cycles/byte using fixslicing [9], processing two 128-bit blocks in parallel.

For Rijndael variants beyond AES-128, the published literature is sparse. Saarinen's evaluation of RISC-V Vector architectures [6] represents the most relevant contemporary work, but targets a different hardware class. Constant-time software implementations for embedded 32-bit processors have not been previously reported.

3. WHY FIXSLICING BECOMES INEFFICIENT FOR RIJNDAEL-256

3.1. The Period Theorem

Fixslicing defers ShiftRows and compensates within MixColumns. After k rounds without explicit ShiftRows, the accumulated row offsets are $(0, k \cdot s_1, k \cdot s_2, k \cdot s_3) \bmod N_b$, where s_1, s_2, s_3 denote the non-zero ShiftRows offsets. The technique requires these offsets to return to $(0, 0, 0, 0)$, establishing a period T.

Theorem 1 (ShiftRows Period). For Rijndael with block size N_b and non-zero ShiftRows offsets s_1, s_2, s_3, the ShiftRows period is

$$T = \operatorname{lcm}\left(\frac{N_b}{\gcd(N_b, s_1)}, \frac{N_b}{\gcd(N_b, s_2)}, \frac{N_b}{\gcd(N_b, s_3)}\right). \quad (1)$$

Proof. Row i, with offset s_i, returns to its original position after k rounds when $k \cdot s_i \equiv 0 \pmod{N_b}$. Thus k must be a multiple of $N_b/\gcd(N_b, s_i)$ for each non-zero row. The period T is the smallest k satisfying all three congruences simultaneously, giving the least common multiple above. ■

https://doi.org/10.31436/iiumej.v27i2.4405

Definition 1 (Bitsliced State Packing). Let $W = 32$ denote the register width in bits and N_b the number of state columns, and $p = 8/N_b$ the number of parallel blocks in the layout. A bitsliced Rijndael state packs each bit-plane into a W-bit register. The register is partitioned into four $(W/4)$-bit row groups, and each column occupies a field of width $g = 4 \cdot p = W/N_b$ bits, distributed across the four row groups and not contiguous in the full register. For AES-128, $p = 2$ and $g = 8$ bits (byte-aligned). For Rijndael-256, $p = 1$ and $g = 4$ bits (sub-byte).

For AES-128 ($N_b = 4$, offsets $1, 2, 3$), $T = \mathrm{lcm}(4, 2, 4) = 4$. For Rijndael-256 ($N_b = 8$, offsets $1, 3, 4$), $T = \mathrm{lcm}(8, 8, 2) = 8$. To illustrate the practical difference, after two rounds, the accumulated offsets for AES-128 are $(0, 2, 0, 2)$, whereas those for Rijndael-256 are $(0, 2, 6, 0)$. Because each column in the two-block AES-128 layout spans $g = 8$ bits, the shifts are byte-aligned and can be performed by the ARM barrel shifter at zero cost. For Rijndael-256, each column spans only $g = 4$ bits, so column shifts are sub-byte and require per-row bit-field manipulation via `UBFX/BFI` rather than barrel-shifted rotation.

Lemma 1 (Rotation Granularity Constraint). On the ARMv7-M architecture, the barrel shifter applies a uniform rotation to an entire 32-bit word. The accumulated ShiftRows compensation for row i at phase k requires a rotation by $r_i(k) = (k \cdot s_i \bmod N_b) \cdot p$ bits within that row's $(W/4)$-bit row group. This per-row rotation can be absorbed into a single barrel-shifted data-processing instruction if and only if all the required rotations are compatible with a whole-word, byte-aligned rotation; otherwise, the rows require independent sub-word manipulation.

$$\text{Condition (2): } r_1(k) = r_2(k) = r_3(k), \text{ with byte-aligned } r_i(k). \tag{2}$$

When Condition (2) does not hold, each non-aligned row requires an extract–shift–insert sequence (`UBFX, LSL/LSR, BFI`), costing at least 3 instructions per row.

Proof. The barrel shifter in `EOR Rd, Rn, Rm, ROR #n` applies a uniform rotation to the entire 32-bit operand Rm. For this to implement the correct compensation, the rotation amount must match the required row-wise rotations without splitting the $(W/4)$-bit row groups. When the amounts differ, or when the rotation is sub-byte as in Rijndael-256, the rows occupy disjoint $(W/4)$-bit row-groups within the register, and no single word-level operation can independently rotate sub-word fields on ARMv7-M. Each misaligned row, therefore, requires at minimum one `UBFX` to extract the $(W/4)$-bit row-group, one shift to rotate it, and one `BFI` to insert it back. Conversely, when Condition (2) holds and $r_i(k)$ is byte-aligned, the uniform rotation by $r_i(k)$ bits correctly compensates all rows simultaneously via a single barrel-shifted instruction, as the rotation amount is a multiple of $(W/4)$ and preserves row-group boundaries. ■

Theorem 2 (Phase-wise Compensation Cost). Let $a_i(k) = k \cdot s_i \bmod N_b$ denote the accumulated column offset for row i at phase k. Define the number of rows requiring compensation at the phase k as

$$\mu(k) = |\{i \in \{1,2,3\} : a_i(k) \neq 0\}|. \tag{3}$$

By Lemma 1, in the row-grouped layout, each non-zero row resides in a distinct row-group within the register and therefore requires its own extract-shift-insert sequence; rows that happen to share the same offset value cannot share a single `UBFX/BFI` pair because the row-groups are not contiguous. Each such sequence costs at least $C_{ins} = 3$ instructions. For Rijndael-256 with offsets (1,3,4), we have $\mu(k) \geq 1$ for all $k \in \{1, \ldots, 7\}$, since $k < T$

https://doi.org/10.31436/iiumej.v27i2.4405

guarantees at least one $a_i(k) \neq 0$. Over one full period $T = 8$, the total additional instruction count is at least

$$C_{ins} \cdot \sum_{k=0}^{T-1} \mu(k) \geq C_{ins}(T-1). \quad (4)$$

Proof. For any $k \in \{1, \ldots, T-1\}$, since $k < T$ and $s_i \neq 0$, we have $k \cdot s_i \not\equiv 0 \pmod{N_b}$ for at least one $i \in \{1, 2, 3\}$, hence $\mu(k) \geq 1$. Summing over all phases gives $\sum_{k=0}^{T-1} \mu(k) \geq T - 1$, and therefore the total additional instruction count is at least $C_{ins}(T-1)$. For Rijndael-256 specifically, we enumerate the offsets $(a_1(k), a_2(k), a_3(k)) \bmod 8$ for $k = 0, \ldots, 7$ with $(s_1, s_2, s_3) = (1, 3, 4)$ and $N_b = 8$:

k	$(\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3})$	$\mu(k)$	Min. Extra instr.
0	(0, 0, 0)	0	0
1	(1, 3, 4)	3	9
2	(2, 6, 0)	2	6
3	(3, 1, 4)	3	9
4	(4, 4, 0)	2	6
5	(5, 7, 4)	3	9
6	(6, 2, 0)	2	6
7	(7, 5, 4)	3	9

Summing the last column gives 54 instructions, which is the tight enumeration-based lower bound for this row-grouped layout. The coarser bound $C_{ins}(T-1) = 21$ instructions are only a conservative universal bound and is not used for the Rijndael-256 cost claim. For even k, the third offset is zero because $4k \equiv 0 \ (mod\ 8)$, but the first two offsets remain non-zero, so $\mu(k) = 2$; for odd k, all three offsets are non-zero, so $\mu(k) = 3$. ■

Corollary 1 (Asymptotic Lower Bound). For any Rijndael variant with period $T > 1$, the fixsliced MixColumns overhead is at least $C_{ins} \cdot (T-1) \cdot \lfloor r/T \rfloor = \Omega(r)$ instructions over the full cipher, where C_{ins} denotes the minimum instruction cost of one extract-shift-insert compensation and r denotes the number of rounds. This overhead is absent in the canonical bitsliced implementation, which realizes ShiftRows through explicit `BFI/UBFX` byte routing rather than phase-dependent MixColumns compensation.

Proof. By Theorem 2, phase k contributes at least $C_{ins} \cdot \mu(k)$ instructions, so a full period contributes at least $C_{ins} \cdot \sum_{k=0}^{T-1} \mu(k)$. Since every non-identity phase admits at least one non-zero offset, we have $\mu(k) \geq 1$ for $k \in \{1, \ldots, T-1\}$, yielding the conservative per-period lower bound $C_{ins} \cdot (T-1)$. The r rounds of the cipher decompose into $\lfloor r/T \rfloor$ disjoint periods of length T, together with at most $T-1$ residual rounds whose contribution we discard. Summing the per-period bound over these $\lfloor r/T \rfloor$ disjoint periods, therefore gives a full-cipher lower bound of $C_{ins} \cdot (T-1) \cdot \lfloor r/T \rfloor$, and since T is fixed this quantity is $\Omega(r)$. For Rijndael-256, the explicit enumeration gives $\sum_{k=0}^{T-1} \mu(k) = 18$, hence 54 instructions per period and $54 \cdot \lfloor r/T \rfloor$ instructions over the full cipher. In the canonical bitsliced approach, ShiftRows is implemented by explicit `BFI/UBFX` routing in a single semi-fused routine, so no phase-dependent MixColumns compensation is emitted. ■

Proposition 1 (Semi-fused vs. Fixsliced Comparison). The semi-fused canonical approach, which performs ShiftRows and MixColumns explicitly in a single routine, costs 297 cycles per round at `-O2` on the STM32F407. The fixsliced approach eliminates ShiftRows but replaces the canonical MixColumns with phase-dependent compensated variants averaging 891 cycles per round (Table 1), a ratio of 3.00 ×. The net balance is negative.

https://doi.org/10.31436/iiumej.v27i2.4405

Proof. On the STM32F407 at `-O2`, we measure each approach as a single per-round unit (Table 1): the semi-fused canonical routine costs C_{SF} = 297 cycles, while the fixsliced compensated MixColumns averages $\bar{C}_{FS}$ = 891 cycles across all eight phases. Since $\bar{C}_{FS}/C_{SF} = 891/297 = 3.00 > 1$, the fixsliced approach is strictly more expensive per round. Over the full 14-round cipher, the measured total encryption cost corroborates the 3.00 × overhead ratio: 11,633 cycles (fixsliced hardcoded) versus 9,971 cycles (semi-fused), a difference of 1,662 cycles or 16.7%. ■

Table 1. Per-phase fixsliced MixColumns cost for Rijndael-256 on STM32F407 (cycles, `-O2`).

Phase*	Cycles	vs Semi-fused
Semi-fused	297	1.00× (base)
Phase 0	949	3.19×
Phase 1	805	2.71×
Phase 2	949	3.19×
Phase 3	853	2.87×
Phase 4	949	3.19×
Phase 5	853	2.87×
Phase 6	949	3.19×
Phase 7	821	2.76×
Average	891	3.00×

*Phase i in this table, corresponds to the mathematical phase $k = (i + 1) \bmod 8$ from Theorem 2. The identity phase ($k = 0$, zero accumulated offset) appears as Phase 7.

Note on the 3.00 × *ratio.* Both implementations are compiled in C at `-O2`; the fixsliced variant decomposes each compensated MixColumns into three explicit steps, undoing the accumulated offset (Step A), performing canonical MixColumns (Step B), applying the next-phase offset (Step C), rather than fusing the compensation into the MixColumns XOR tree. For Rijndael-256, such fusion would not reduce the instruction count: the sub-byte compensation ($g = 4$) cannot be absorbed by the barrel shifter, so each non-zero row still requires an `UBFX/BFI` sequence regardless of whether it is performed inline or as a separate pass. The 3.00 × ratio therefore reflects both the algorithmic overhead (two rotation passes versus one) and compiler overhead (generic C rotation versus hand-scheduled assembly). At the instruction level, the hand-tuned assembly ShiftRows costs 96 instructions per call (Table 8), a fixsliced round would require two such passes plus MixColumns, giving a theoretical lower bound of $96 + 96 + 27 = 219$ instructions versus $96 + 27 = 123$ for the canonical round, a ratio of 1.78 ×, confirming that the overhead is structural, not an artifact of the C implementation.

Remark 1 (Decryption Symmetry). The preceding results apply equally to decryption. InvShiftRows uses offsets $(0, N_b - s_1, N_b - s_2, N_b - s_3)$; for Rijndael-256 these yield $(0, 7, 5, 4)$. The inverse period is $T_{inv} = \mathrm{lcm}\left(\frac{8}{\gcd(8,7)}, \frac{8}{\gcd(8,5)}, \frac{8}{\gcd(8,4)}\right) = \mathrm{lcm}(8, 8, 2) = 8 = T$. The column field width g is unchanged, and the same byte-alignment condition determines efficiency. Therefore, separate proofs for InvShiftRows are unnecessary.

3.2. General Condition for Fixslicing Inefficiency

The preceding analysis of Rijndael-256 is a special case of a broader structural limitation. We now characterize, for the general Rijndael family, the conditions under which fixslicing becomes more expensive than the canonical approach. The primary bottleneck is not the

https://doi.org/10.31436/iiumej.v27i2.4405

specific ShiftRows offset values: all Rijndael variants with $N_b \in \{4, 5, 6\}$ share the same ShiftRows offsets $(1, 2, 3)$; $N_b = 7$ uses $(1, 2, 4)$; only $N_b = 8$ uses $(1, 3, 4)$. Rather, the decisive factor is the column field width g. When $g < 8$, column shifts correspond to sub-byte rotations within each $(W/4)$-bit row-group, which no single ARMv7-M data-processing instruction can absorb for free.

Theorem 3 (Fixslicing Efficiency Threshold). Let C_{SR} be the cost (in instructions) of an explicit ShiftRows per round, and let C_{ins} be the cost of a single extract-shift-insert operation. Fixslicing incurs zero ShiftRows compensation cost within MixColumns if

(C1) Byte-alignment: $g = W/N_b \geq 8$, i.e., each column occupies at least one full byte in the 32-bit register layout, so a column shift corresponds to a byte-aligned rotation that can be absorbed by the barrel shifter's second operand at zero additional instruction cost.

In particular, AES-128 satisfies (C1) because $N_b = 4$ and $g = 8$. Its fixsliced MixColumns variants absorb all compensation rotations into barrel-shifted operands of existing `EOR` instructions, despite the four rows sharing each register.

When (C1) is violated (g < 8), the per-row compensation of $(k \cdot s_i \bmod N_b) \cdot p$ bits is sub-byte and requires extract–shift–insert sequences. A sufficient (but not necessary) condition for fixslicing to be more expensive than the canonical approach is:

$$\frac{C_{ins}}{T} \sum_{k=0}^{T-1} \mu(k) > C_{SR}\,, \tag{5}$$

where $\mu(k)$ is the number of rows requiring compensation at phase k (Theorem 2). The left-hand side is a lower bound on the average per-round compensation cost; the actual overhead may be substantially larger due to factors not captured by the instruction count (compiler-generated bit-manipulation code expanding beyond the minimum 3-instruction pattern, pipeline stalls from chained data dependencies, function-dispatch, and instruction-cache pressure from the T-variant phase table). Hence, failure to satisfy (5) does not imply that fixslicing wins. For borderline variants, the verdict requires empirical measurement.

Proof. Sufficiency of (C1). When $g \geq 8$, a column shift corresponds to a register-level rotation. Adomnicai and Peyrin [9] exploit this property by pre-permuting round keys offline (during key expansion) so that, after the accumulated ShiftRows shift, the bits required at each register position are already laid out correctly in the round key. At runtime, the rotation is applied to the pre-permuted round key as the second operand of an `EOR Rd, Rn, Rm, ROR #sh` instruction, not to the state itself, where $sh = g \cdot a_i(k) = g \cdot (k \cdot s_i \bmod N_b)$ denotes the rotation amount in bits for each row $i \in \{1, 2, 3\}$. The ARMv7-M barrel shifter absorbs this rotation at zero instruction cost. Each fixsliced phase k emits zero additional compensation instructions.

Cost lower bound when (C1) fails. If $g < 8$, the corresponding rotation amount is sub-byte and can split bits across row-group boundaries, corrupting the per-row data. Since ARMv7-M lacks sub-word rotation instructions that operate independently on g-bit fields, each non-zero row compensation requires an extract-shift-insert sequence `(UBFX, shift, BFI)`, costing at least $C_{ins} = 3$ instructions.

Sufficient condition for Canonical to win. When compensation is required, by Lemma 1 each non-zero row contributes at least C_{ins} instructions, so phase k adds at least $C_{ins} \cdot \mu(k)$ instructions to MixColumns. Averaged over one period T, the per-round overhead is therefore at least $(C_{ins}/T) \cdot \sum_k \mu(k)$. Fixslicing saves C_{SR} per round by eliminating explicit ShiftRows. Therefore, if $C_{SR} < (C_{ins}/T) \cdot \sum_k \mu(k)$, the canonical approach is provably cheaper. The

converse does not hold, because real implementations may incur overhead far above the lower bound. ■

Remark 2 (Rijndael-256 Lies in the Borderline Regime). For Rijndael-256, $\sum_k \mu(k) = 18$ (Theorem 2), giving a formal lower bound of $3 \cdot 18/8 = 6.75$ instructions per round. The C ShiftRows implementation costs approximately 219 cycles per round at `-O2` (Table 7), while the hand-tuned assembly routing costs 96 instructions and measures 98–168 cycles depending on optimization context (Tables 7 and 8). Because these values substantially exceed the formal lower bound, Theorem 3 must be read as a conservative sufficient condition only. It does not predict Rijndael-256's actual outcome. The operational verdict for Rijndael-256 is delivered by Proposition 1, which shows by direct measurement that the canonical semi-fused approach is faster by 1,662 cycles in the C-only comparison.

Remark 3 (Rijndael Variant Analysis). Table 2 applies this criterion to all Rijndael block lengths. All variants with $N_b \in \{4, 5, 6\}$ share the same ShiftRows offsets (1,2,3); $N_b = 7$ uses $(1, 2, 4)$; and $N_b = 8$ uses $(1, 3, 4)$. Despite identical offsets, fixslicing is efficient only for AES-128 ($N_b = 4$), because it is the only variant where $g \geq 8$. For $N_b \in \{5, 6, 7\}$, the column field width g is not even an integer, meaning that state columns cannot be packed into byte-aligned fields within 32-bit registers. This represents a more severe obstacle than the sub-byte fields of Rijndael-256 ($g = 4$).

Table 2. Fixslicing feasibility for all Rijndael variants ($W = 32$, p parallel blocks as defined).

Variant	N_b	(s_1, s_2, s_3)	T	p	g	Verdict
Rijndael-128 (AES)	4	(1,2,3)	4	2.0	8.0	Efficient
Rijndael-160	5	(1,2,3)	5	1.6†	6.4†	Not applicable
Rijndael-192	6	(1,2,3)	6	1.3†	5.3†	Not applicable
Rijndael-224	7	(1,2,4)	7	1.1†	4.6†	Not applicable
Rijndael-256	8	(1,3,4)	8	1.0	4.0	Inefficient

† Variants with non-integer g and p cannot be evenly packed into 32-bit row-groups, so the standard bitsliced layout is not realizable for these variants.

3.3. Measured Per-Phase Costs

Figure 1 illustrates why fixslicing succeeds for AES-128 but fails for Rijndael-256. Table 1 uses dispatch labels in which the Phase i maps to the mathematical phase $k = (i + 1) \bmod 8$. Therefore, phases requiring compensation for all three rows, $k \in \{1, 3, 5, 7\}$, correspond to Phases 0, 2, 4, and 6 in Table 1 and are the most expensive at 949 cycles each, consistent with Theorem 2's prediction. Phase 7 corresponds to $k = 0$, the identity case, and costs 821 cycles, essentially representing canonical MixColumns plus dispatch overhead with no compensation. Phases with $\mu(k) = 2$ range from 805 to 853 cycles depending on the non-zero shift amounts. Even the cheapest compensated phase costs 2.71 × the semi-fused baseline.

https://doi.org/10.31436/iiumej.v27i2.4405

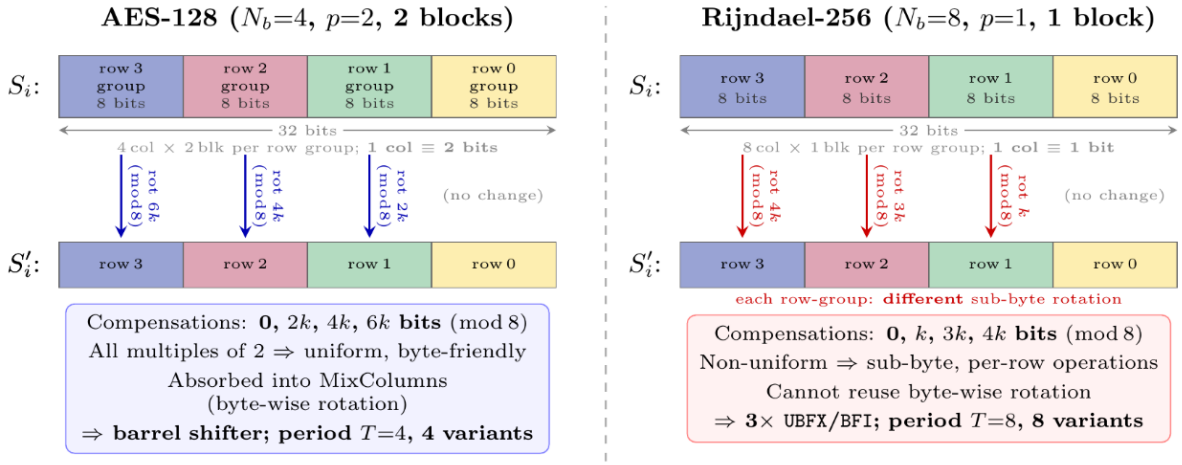

Figure 1. Fixslicing ShiftRows compensation for AES-128 ($g = 8$, byte-aligned) vs. Rijndael-256 ($g = 4$, sub-byte `UBFX/BFI`).

3.4. Key Schedule Overhead

The fixsliced key schedule must pre-rotate round keys to match each phase's accumulated offset. Table 3 shows that this costs 6,370 cycles versus 3,285 cycles for the canonical bitsliced schedule at `-O2`, a ratio of 1.94 × . At `-Os`, the ratio increases to 3.06 × because the rotation code sequences resist size optimization, causing instruction cache pressure.

Table 3. Key schedule cost for Rijndael-256 on STM32F407 (cycles).

Key Schedule	-O0	-O1	-O2	-O3	-Os
BS canonical	21,231	5,085	3,285	3,267	3,234
FS fixsliced	40,570	9,829	6,370	6,189	9,894
FS/BS ratio	1.91	1.93	1.94	1.89	3.06

4. IMPLEMENTATION

We describe the key components of our implementation. Table 4 summarizes the notation used for the implementation variants discussed throughout this paper.

Table 4. Implementation variants and abbreviations.

Abbreviation	Description
Full ASM (macro)	Hand-tuned assembly, macro-inlined (S-box + MC + SR)
Full ASM (func)	Hand-tuned assembly, function-call (S-box + MC + SR)
Inline ASM SR	C bitsliced with inline ASM ShiftRows
BS C (semi-f.)	Pure C bitsliced, explicit ShiftRows
FS hardcoded	Fixsliced with 8 hardcoded MC variants
FS generic	Fixsliced with generic MC compensation
T-table 4T	Four lookup tables (4×1 KB)
T-table 1T	One lookup table (1 KB) + barrel shifter

4.1. Bitsliced State Layout

We adopt a byte-based packing in which each 32-bit register byte holds one state row across eight columns, with reversed column order (column 0 in the MSB, column 7 in the LSB). In this representation, `ROR32(x,8)` shifts data between rows, and ShiftRows requires left rotation per byte within each register. Figure 2 illustrates this mapping.

https://doi.org/10.31436/iiumej.v27i2.4405

	row 3 [31:24]					row 0 [7:0]			
	c_0	c_1	$\cdots$	c_7	$\cdots$	c_0	c_1	$\cdots$	c_7
S_0	$b_{3,0}[0]$	$b_{3,1}[0]$	$\cdots$	$b_{3,7}[0]$	$\cdots$	$b_{0,0}[0]$	$b_{0,1}[0]$	$\cdots$	$b_{0,7}[0]$
S_1	$b_{3,0}[1]$	$b_{3,1}[1]$	$\cdots$	$b_{3,7}[1]$	$\cdots$	$b_{0,0}[1]$	$b_{0,1}[1]$	$\cdots$	$b_{0,7}[1]$
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$		$\vdots$
S_7	$b_{3,0}[7]$	$b_{3,1}[7]$	$\cdots$	$b_{3,7}[7]$	$\cdots$	$b_{0,0}[7]$	$b_{0,1}[7]$	$\cdots$	$b_{0,7}[7]$

Figure 2. Rijndael-256 bitsliced register layout. Register S_i holds bit i of each state byte. Cell $b_{r,c}[i]$ equals plaintext bit $(4c + r) \cdot 8 + i$.

Figure 3 shows the exact bit-level mapping. The 3-stage `swpmv` butterfly establishes a reversed column order: column 0 maps to bit 7 (MSB) of each row group, and column 7 to bit 0 (LSB), so that a left rotation `(ROL)` within an 8-bit group directly implements ShiftRows. The precise packing rule is $S_i[8r + (7 - c)] = b_{r,c}[i]$.

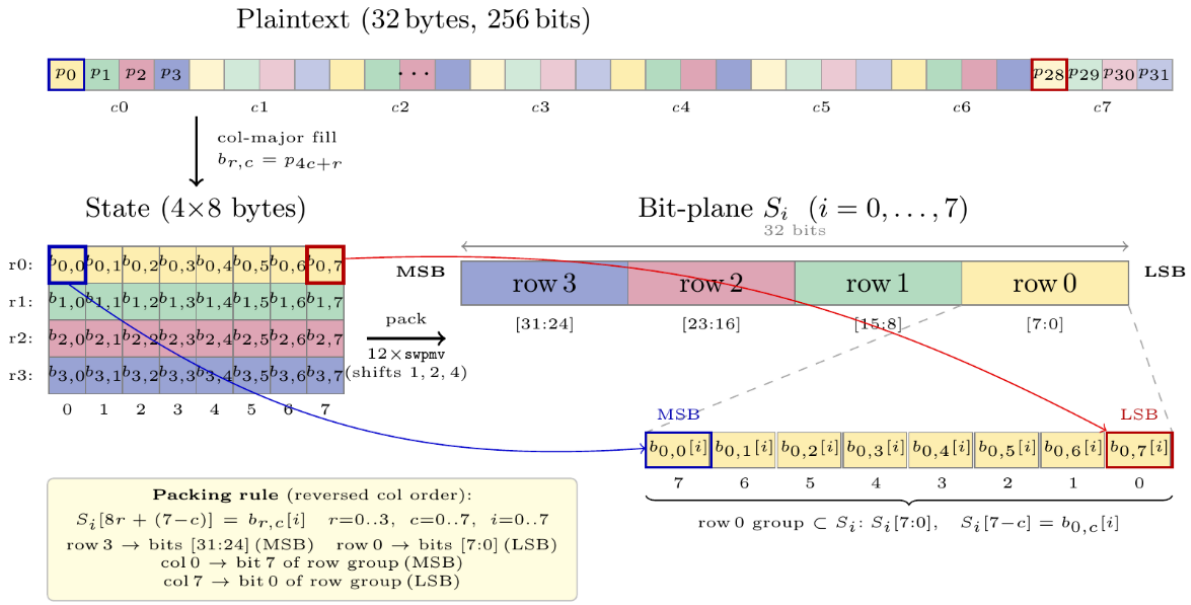

Figure 3. Bit-level packing layout. The 3-stage `swpmv` butterfly maps each state byte $b_{r,c}$ into bit-plane S_i at position $8r + (7 - c)$. Row 0 occupies the LSB byte [7:0]; row 3 the MSB byte [31:24].

4.2. SubBytes

The S-box uses the Schwabe–Stoffelen Boolean circuit [14], which operates on eight bit-plane registers via Boolean operations without memory accesses. Care must be taken not to duplicate NOT operations in the key schedule, as the Boyar–Peralta circuit [13] already incorporates them internally.

4.3. ShiftRows

We implement ShiftRows in two ways: pure C with byte rotation macros, and inline ARM assembly using `BFI/UBFX` instructions. The inline assembly variant extracts bytes using `UBFX`, rotates them according to the offsets $(0, 1, 3, 4)$, and reinserts them using `BFI`. At `-O3`,

https://doi.org/10.31436/iiumej.v27i2.4405

the assembly variant costs 98 cycles versus 260 for C (2.65 × improvement). This approach addresses the byte-shuffle bottleneck identified by Saarinen [6] in the RISC-V context, using the Cortex-M4's bit-field instructions rather than vector permutations.

4.4. MixColumns

Multiplication by 2 in $GF(2^8)$ at MixColumns is a left shift with conditional XOR of `0x1B` (denoted `xtime`), and multiplication by 3 is `xtime` XOR with the original. In our bitsliced design, the MixColumns step follows the implementation approach originally proposed by Käsper and Schwabe [20] for x86, adapted for ARM by Schwabe and Stoffelen [14], and extended to the eight-column Rijndael-256 state. Multiplication by 2 in $GF(2^8)$ is realized by rotating the eight bit-plane registers and XORing the high-bit plane into the appropriate reduction planes; multiplication by 3 adds the original plane to the doubled result. All operations are unconditional register-level `EOR` and `ROR` instructions with no data-dependent branches or memory accesses, preserving the constant-time property.

For the fixsliced variants (FS hardcoded, FS generic), the same clean MixColumns circuit is used internally, but wrapped with per-byte compensation rotations. Each phase-k variant applies three steps: (A) rotate each byte of the eight-bit plane registers to undo the accumulated ShiftRows offset, (B) execute the clean MixColumns, and (C) rotate bytes again to impose the next phase's offset. The rotation amounts are pre-computed per phase in the hardcoded variant; the generic variant computes them at runtime. Two phases incur reduced compensation cost: phase 1 requires no Step A, and phase 7 requires no Step C, as their respective rotation amounts are all zero. The additional cost relative to the canonical circuit arises entirely from Steps A and C, which require approximately $3 \cdot \mu(k)$ instructions per step, giving $6 \cdot \mu(k)$ total across Steps A and C for $\mu(k)$ non-zero rows, consistent with the overhead analyzed in Section 3.

4.5. Full Assembly Variants

Our highest-performance variants combine the Schwabe–Stoffelen S-box [14], the canonical bitsliced MixColumns described in Section 4.4, and custom `BFI/UBFX` ShiftRows for the $(0, 1, 3, 4)$ offsets. We provide two versions that expose a trade-off between code size and performance.

The *function-call variant* (Full ASM (func)) wraps the round function in a standard ARM procedure. It achieves 6,518 cycles (203.7 cycles/byte) at `-O2`, is compiler-independent, and requires 658 bytes of flash for the ASM cipher core.

The *macro-inlined variant* (Full ASM (macro)) expands the round function at each of the 14 call sites, eliminating all `BL/BX` branching and register save/restore overhead. This reduces the per-round overhead by approximately 23 cycles, yielding 6,199 cycles (193.7 cycles/byte) at the cost of a larger flash footprint. Both variants produce cycle counts that are essentially invariant across compiler optimization flags, since all inner-loop code is hand-written assembly.

4.6. T-table Baseline

For comparison, we implement 4-table and 1-table T-table variants (not in constant time). The standard relationship $T_j[x] = ROL32(T_0[x], 8j)$ requires left rotation in practice. Concretely, $T_1 = ROL32(T_0, 8)$, $T_2 = ROL32(T_0, 16)$, and $T_3 = ROL32(T_0, 24)$. On the Cortex-M4, the barrel shifter absorbs this at zero cost. On RISC-V RV32I, which lacks a native rotation instruction, this would add 2–3 cycles per derivation, potentially affecting the relative advantage of 1-table versus 4-table strategies.

https://doi.org/10.31436/iiumej.v27i2.4405

5. EXPERIMENTAL RESULTS

All benchmarks were conducted on the STM32F407VGT6 Discovery Board, with the ART Accelerator enabled and the core clock set to 168 MHz. Cycle counts were measured using the DWT CYCCNT register with interrupts disabled. The evaluation firmware was compiled with the GNU Arm Embedded GCC toolchain (arm-none-eabi-gcc; exact release recorded in the build artifact) using the optimization flags reported in the tables (`-O0`, `-O1`, `-O2`, `-O3`, and `-Os`) and targeting ARMv7E-M/Thumb-2. For the assembly variants, all reported cycle counts include bitsliced state packing and unpacking, as these operations are integrated into `rj256_encrypt_asm`. Source code, build scripts, and raw measurement logs are available from the corresponding author upon reasonable request and will be archived publicly upon acceptance.

5.1. Encryption Performance

Table 5 presents encryption cycle counts across all optimization levels. The macro-inlined assembly variant achieves 6,199 cycles (193.7 cycles/byte), while the function-call variant achieves 6,518 cycles (203.7 cycles/byte). Both are essentially invariant across compiler optimization flags. The semi-fused C variant achieves 9,971 cycles (311.6 cycles/byte) at `-O2`, outperforming the fixsliced hardcoded variant (11,633 cycles, 363.5 cycles/byte) by 16.7%.

Among non-constant-time implementations, the 1-table variant achieves 3,252 cycles (101.6 cycles/byte) at `-O2`, outperforming the 4-table variant by 20%. The barrel shifter makes `ROR` at runtime effectively free, while the single 1 KB table is more cache-friendly than four 1 KB tables. The `-O2` level produces the best results for both bitsliced implementations and the T-table 1T variant; for the T-table 4T variant, `-O1` is optimal. At `-O3`, the bitsliced C variant regresses by 6% due to aggressive inlining causing code bloat.

Table 5. Encryption performance for Rijndael-256 on STM32F407 at 168 MHz (cycles).

Implementation	-O0	-O1	-O2	-O3	-Os
Constant-time:					
Full ASM (macro)	6,199	6,199	6,199	6,199	6,199
Full ASM (func)	6,524	6,521	6,518	6,519	6,517
Inline ASM SR	42,064	9,044	9,102	8,295	9,635
BS C (semi-f.)	48,028	10,129	9,971	10,559	11,829
FS hardcoded	66,972	11,719	11,633	12,322	20,527
FS generic	76,004	19,308	17,736	15,084	23,384
Non-constant-time:					
T-table 4T	8,624	3,800	4,036	4,300	3,968
T-table 1T	8,925	3,891	3,252	3,842	3,801

5.2. Timing Variability Analysis

Table 6 reports the timing range (max-min) over 10^6 iterations. The constant-time implementations range from 46–140 cycles, while T-table variants range from 132–180 cycles. The constant-time implementations contain no data-dependent memory accesses; the residual variability arises solely from instruction-fetch effects (cache line alignment, flash prefetch). The T-table variability is systematically higher because different plaintext values access different cache lines, constituting exploitable timing leakage.

https://doi.org/10.31436/iiumej.v27i2.4405

Table 6. Timing variability for Rijndael-256 on STM32F407 (cycle range over 10^6 iterations).

Implementation	-O0	-O1	-O2	-O3	-Os
Full ASM (func)	79	62	49	46	48
BS C (semi-f.)	140	132	70	57	118
FS hardcoded	101	86	93	68	121
FS generic	91	68	61	60	76
T-table 4T	172	148	151	174	132
T-table 1T	180	145	150	172	135

5.3. Component Breakdown

Table 7 presents the per-component cycle costs across all optimization levels. SubBytes dominates at 293 cycles per call at `-O2`, consistent with the complexity of the Boolean S-box circuit. ShiftRows is the second most expensive component. The inline assembly variant provides a 23% improvement over C (168 vs. 219 cycles). MixColumns is efficient at 82 cycles. Packing and unpacking together consume 332 cycles, a one-time overhead amortized in modes processing multiple blocks.

Table 7. Per-component cycle costs for the bitsliced Rijndael-256 implementation on STM32F407 (cycles per call).

Component	-O0	-O1	-O2	-O3	-Os
Packing	990	209	109	109	108
SubBytes	1,500	297	293	293	304
ShiftRows (C)	997	220	219	260	332
ShiftRows (ASM)	381	156	168	98	155
MixColumns	357	83	82	82	85
Unpacking	1,115	217	223	222	219

5.4. Theoretical Instruction-Level Cost Analysis

We analyze the theoretical cycle cost of the macro-inlined assembly variant by categorizing instruction sequences by cryptographic role and computing the minimum cycle counts under the assumption of zero pipeline stalls. Comparing these theoretical minima against measured results quantifies the pipeline stall overhead per component, revealing which operations are the primary performance bottlenecks on the Cortex-M4.

5.4.1. Cycle Model

We apply the following Cortex-M4 single-issue cycle model: data-processing instructions (`EOR, AND, ORR, MOV, ROR,` with or without an inline barrel shift) execute in 1 cycle; `UBFX` and `BFI` execute in 1 cycle; `LDR/STR` cost 2 cycles; `LDMIA/STMIA` for n registers cost $n + 1$ cycles; `PUSH/POP` for n registers cost $n + 1$ cycles. Load-use hazards add 1 stall cycle each.

5.4.2. Per-Component Analysis

Packing: The 12 `swpmv` macro invocations (each expanding to 4 data-processing instructions) plus the function prologue (`PUSH` {14 regs}, `SUB`, `STR`, `MOV`, `LDM` {8 regs}) and mask-setup (`MOVW`, `MOVT`, 2×`EOR`) contribute 80 theoretical cycles.

AddRoundKey (ARK): Each round loads two groups of four round-key words (`LDR` + 2×`LDMIA` {4}) and (8×`EOR`) with two pointer-save/restore instructions (`STR.W` + `STR`): 24 theoretical cycles per call × 14 rounds = 336 cycles total.

https://doi.org/10.31436/iiumej.v27i2.4405

SubBytes: The Boyar–Peralta depth-16 Boolean circuit comprises 146 instructions (81 `EOR`, 32 `AND`, 16 `STR`, and 17 `LDR`), yielding 179 theoretical cycles per call. The measured value of 269 cycles gives a stall factor of 1.50 ×, attributable to frequent read-after-write hazards in the chained Boolean operations and load-use stalls on the stack-spilled intermediates.

ShiftRows: Each `shiftrows_one` invocation consists of 12 instructions (4 ×`UBFX`, 2 ×`ORR`, 2 ×`BFI` for rows 1–2, plus 2 ×`UBFX`, 2 ×`BFI` for the row-3 nibble swap) whose data dependencies are resolved by result forwarding on the Cortex-M4, giving 96 theoretical cycles for 8 registers per call. The measured value at `-O3` is 98 cycles, yielding a stall factor of 1.02 ×, essentially pipeline-stall-free. This confirms that the `UBFX/BFI` routing approach is not only correct but also architecturally efficient.

MixColumns: The Käsper–Schwabe circuit uses 27 `EOR/ROR` instructions (27 theoretical cycles), yet measures 82 cycles, a stall factor of 3.04 ×. This overhead is caused by the chained `EOR` sequence: many instructions read registers written 1–2 instructions earlier, causing systematic 1–2 cycle stalls on each dependency.

Unpacking: The reverse transposition comprises 12 swpmv invocations and mask-setup (`MOVW`, `MOVT`, 2 ×`EOR`), plus the function epilogue (`STMIA{8}`, `ADD.W`, `POP{14}`), yielding 77 theoretical cycles.

5.4.3. Summary

As summarized in Table 8, the theoretical minimum is 4,694 cycles; the measured result is 6,199 cycles, giving an overall pipeline efficiency of 4,694/6,199 ≈ 76%. Two findings are architecturally significant. First, ShiftRows achieves a stall factor of 1.02 ×, near-theoretical performance, because data dependencies are resolved by result forwarding on the Cortex-M4, causing no effective pipeline stalls. This result validates the explicit `UBFX/BFI` routing strategy as both functionally correct and pipeline-friendly. Second, MixColumns carries the highest stall penalty (3.04×) despite using only simple `EOR/ROR` instructions, because the Käsper–Schwabe diffusion circuit is inherently sequential with chained data dependencies. This suggests that reordering the MixColumns instruction stream to interleave independent operations across columns represents the most promising target for future micro-optimization.

Table 8. Theoretical instruction-level cost analysis for the macro-inlined assembly variant at `-O2`.

Component	Instr./call	Theory cyc/call	Calls	Total (theory)	Stall factor
Packing	57	80	1	80	≈1.1×
AddRoundKey	13	24	14	336	≈1.3×
SubBytes	146	179	14	2,506	≈1.5×
ShiftRows	96	96	14	1,344	1.02×
MixColumns	27	27	13	351	3.04×
Unpacking	56	77	1	77	≈1.6×
Total				**4,694**	**1.32×**

5.5. Code Size and Memory

The function-call assembly variant requires only 658 bytes of flash for the ASM cipher core with zero RAM overhead, while the macro-inlined variant requires 15,898 bytes due to full round-function expansion across all 14 rounds. The constant-time implementations require no data RAM beyond the stack and round keys. All implementations share 480 bytes of round key storage (15 keys × 32 bytes).

Table 9. Memory requirements for Rijndael-256 implementations on STM32F407 (`-O2`).

Implementation	Flash (B)	RAM (B)	Stack (B)	Round Keys (B)
Full ASM (macro)	15,898	0	112	480
Full ASM (func)	658	0	112	480
BS C (semi-f.)	2,204	0	96	480
T-table 1T	1,528	1,024	64	480
T-table 4T	5,680	4,096	64	480
Fixsliced HC	3,816	0	96	480

5.6. Constant-Time Verification

5.6.1. Methodology and Experimental Setup

To empirically verify the constant-time execution of the proposed bitsliced Rijndael-256 implementation, we adopted the DUDECT methodology proposed by Reparaz *et al.* [10]. Unlike static code analysis, DUDECT performs a leakage assessment using Welch's t-test by measuring actual execution times on the target hardware. The assessment evaluates whether the execution duration exhibits a statistically significant dependency on the processed input data. DUDECT therefore provides empirical evidence under the tested hardware and testbench conditions, not a formal proof of constant-time behavior.

The evaluation was conducted on an ARM Cortex-M4 microcontroller. Cycle-accurate measurements used the Data Watchpoint and Trace (DWT) cycle counter native to the Cortex-M architecture. The testing procedure was divided into two distinct classes: fixed data (Class 0) and randomized data (Class 1). The t-statistic was continuously computed using Welford's online algorithm for running variance, enabling the collection of 10^6 samples without requiring $O(N)$ memory on the target device. A $|t|$-statistic value exceeding 4.5 indicates detectable timing leakage with a confidence level greater than 99.999%.

A critical aspect of hardware-level timing analysis is isolating software-induced noise and architectural artifacts. During our initial setup, the standard C library pseudorandom number generator (PRNG) was found inadequate because its internal state management introduced artificial memory-alignment stalls. Consequently, we integrated the microcontroller's hardware true random number generator (TRNG) to supply the testbench with high-entropy random values, ensuring unbiased class distribution and input generation. Furthermore, to mitigate pipeline state carry-over caused by the differing data preparation routines of Class 0 and Class 1, Data Synchronization Barrier (`DSB`) and Instruction Synchronization Barrier (`ISB`) instructions were injected immediately preceding the DWT cycle capture. This guarantees a uniform micro-architectural state prior to execution [18].

5.6.2. Results and Discussion

The implementation was compiled using the GCC toolchain and evaluated across all primary optimization levels (`-O0`, `-O1`, `-O2`, `-O3`, and `-Os`). For each configuration, we collected $N = 10^6$ samples. The progression of the experimental results, summarized in Table 10, highlights the crucial role of environmental sanitization in micro-architectural timing evaluations.

Initial baseline testing with the software PRNG reported false-positive leakages at the -O1 optimization level ($t = 0.09$). This anomaly was traced to memory alignment discrepancies introduced by the compiler when handling the PRNG's static variables alongside the testbench buffers, resulting in an artificial penalty of approximately 2 cycles. Replacing the PRNG with the hardware TRNG eliminated this artifact, dropping the t-statistic to 0.11 and confirming the absence of algorithmic leakage at `-O1`.

https://doi.org/10.31436/iiumej.v27i2.4405

Table 10. DUDECT Evaluation Results for Full ASM (macro) Variant Across Compiler Optimization Levels ($N = 10^6$ Samples).

Opt. Level	Testbench Environment	Mean Class 0	Mean Class 1	t-Statistic / Verdict
`-O0`	Baseline (PRNG)	6208.64	6208.61	1.1173 / PASS
`-O0`	TRNG Only (No Flush)	6208.67	6208.68	0.3151 / PASS
`-O0`	Final (TRNG + Pipeline Flush)	6209.61	6209.61	0.0285 / PASS
`-O1`	Baseline (PRNG)	6202.15	6204.11	90.09 / FAIL (False Positive)
`-O1`	TRNG Only (No Flush)	6205.08	6205.08	0.1054 / PASS
`-O1`	Final (TRNG + Pipeline Flush)	6202.07	6202.07	0.1912 / PASS
`-O2`	Baseline (PRNG)	6197.64	6197.64	0.3425 / PASS
`-O2`	TRNG Only (No Flush)	6198.72	6199.75	57.45 / FAIL (1-Cycle Stall)
`-O2`	Final (TRNG + Pipeline Flush)	6198.71	6198.71	0.2385 / PASS
`-O3`	Baseline (PRNG)	6197.83	6197.84	0.8650 / PASS
`-O3`	TRNG Only (No Flush)	6198.64	6199.63	59.16 / FAIL (1-Cycle Stall)
`-O3`	Final (TRNG + Pipeline Flush)	6198.73	6198.77	2.0229 / PASS
`-Os`	Baseline (PRNG)	6197.88	6197.85	1.8792 / PASS
`-Os`	TRNG Only (No Flush)	6197.93	6197.94	0.5411 / PASS
`-Os`	Final (TRNG + Pipeline Flush)	6197.86	6197.87	0.3785 / PASS

However, isolating the input generation with the TRNG revealed a secondary micro-architectural artifact at higher optimization tiers (`-O2` and `-O3`). As shown in the intermediate TRNG tests, the evaluation yielded t-statistics exceeding 55. A detailed trace of the cycle counts showed a microscopic, highly stable difference of exactly 1 cycle between the two classes (e.g., 6198.72 vs. 6199.75 cycles at `-O2`).

This 1-cycle discrepancy is a classic manifestation of branch predictor and pipeline state leakage. The testbench prepares Class 0 using a highly predictable `memset` operation, whereas Class 1 requires iterative polling of the TRNG peripheral. At `-O2` and `-O3`, the aggressive compiler loop unrolling and branch generation for the TRNG polling left the Cortex-M4's pipeline in an asymmetric state upon entering the encryption function. The 1-cycle delay was due to the processor flushing its pipeline after a loop-termination misprediction.

By implementing the `DSB` and `ISB` pipeline flushing mechanism prior to the measurement block, the processor state was successfully synchronized. As demonstrated in Table 10, the t-statistics for all optimization levels, including `-O2` ($t = 0.23$) and -O3 ($t = 2.02$), fell well below the 4.5 threshold. The execution cycles became well-aligned across both classes, providing strong statistical evidence that the macro-inlined assembly variant exhibits no detectable data-dependent timing variation across all tested compiler optimization levels.

Table 11. DUDECT Evaluation Results for Full ASM (func) Variant ($N = 10^6$ Samples, TRNG + Pipeline Flush).

Opt. Level	Mean Class 0	Mean Class 1	t-Statistic / Verdict
`-O0`	6534.72	6534.68	1.7204 / PASS
`-O1`	6527.14	6527.10	1.7555 / PASS
`-O2`	6523.77	6523.73	2.1849 / PASS
`-O3`	6523.79	6523.80	0.7758 / PASS
`-Os`	6522.90	6522.90	0.3333 / PASS

The function-call variant (Full ASM (func)) shares an identical assembly core with the macro-inlined variant; the sole architectural difference is the addition of data-independent function-call overhead (BL/BX and PUSH/POP instructions), which, by definition, introduces

https://doi.org/10.31436/iiumej.v27i2.4405

no secret-dependent timing variation. Full DUDECT evaluation of this variant with $N = 10^6$ samples using the same TRNG and pipeline-flush methodology is presented in Table 11. All optimization levels yield t-statistics well below the threshold of 4.5, with a maximum t-statistic of 2.18 at `-O2`, confirming that the constant-time property is preserved across both variants.

6. COMPARISON WITH AES-128 AND AES-256

Table 12 summarizes the constant-time performance comparison. First, we compare Rijndael-256 with AES-128 using the public fully-fixsliced AES implementation of Adomnicai and Peyrin [9], re-measured on the same STM32F407VG platform and under the same DWT-based timing protocol used for our Rijndael-256 implementation. Under this same-platform measurement, AES-128 achieves 105.1 cycles/byte. Therefore, the rolled Rijndael-256 variant yields $203.7/105.1 = 1.94\times$, while the unrolled variant yields $193.7/105.1 = 1.84\times$. The originally reported AES-128 figure in [9] is 80.0~cycles/byte; using that published value instead would give ratios of $203.7/80 = 2.55\times$ for the rolled variant and $193.7/80 = 2.42\times$ for the unrolled variant. We therefore use the same-platform reproduced value as the primary normalization point, while reporting the published-reference ratios for transparency.

The expected scaling from AES-128 to Rijndael-256 is affected by two factors. Rijndael-256 uses 14 rounds instead of 10 (factor $1.4\times$), while the fixsliced AES-128 implementation processes two 128-bit blocks in parallel ($2\times$ throughput advantage). A simple scaling estimate suggests $1.4\times 2.0 = 2.8\times$. Both the same-platform ratio of $(1.84\times - 1.94\times)$ and the published-reference ratio of $(2.42\times - 2.55\times)$ are below this estimate, indicating that the proposed semi-fused Rijndael-256 strategy is efficient relative to the naive scaling expectation, despite the wider state and irregular ShiftRows structure.

Table 12. Comparison of constant-time performance on STM32F407VG at 168 MHz. All cycle counts include packing and unpacking.

Cipher	Key (bit)	Blocks	Rounds	Cycles	Cycles/byte
AES-128†	128	2	10	3,362	105.1
AES-256†	256	2	14	4,590	143.4
Rijndael-256 (rolled)	256	1	14	6,518	203.7
Rijndael-256 (unrolled)	256	1	14	6,199	193.7

† AES-128 and AES-256 were measured using the public fully-fixsliced implementation of Adomnicai and Peyrin [9] on the same STM32F407VG platform and under the same interrupt-disabled DWT measurement protocol. Under this same-platform setup, AES-128 measured 105.1 cycles/byte. The originally reported AES-128 figure in [9] is 80.0 cycles/byte; we use the reproduced value on the same platform for the primary comparison.

Second, we compare Rijndael-256 with AES-256. This comparison is structurally cleaner because both ciphers use a 256-bit key, perform 14 rounds, and process 32 bytes per call (AES-256 through two parallel 128-bit blocks, and Rijndael-256 through one 256-bit block). Using the same platform AES-256 measurement of 143.4 cycles/byte, the rolled Rijndael-256 variant yields $203.7/143.4 = 1.42\times$, while the unrolled variant yields $193.7/143.4 = 1.35\times$. This ratio more directly isolates the cost of Rijndael-256's wider state ($N_b = 8$) and irregular ShiftRows offsets relative to AES-256 ($N_b = 4$), with key size and round count held constant.

For context, Saarinen reports a $2.66\times$ instruction count ratio for Rijndael-256 versus AES-256 on RISC-V Vector hardware [6]. That metric targets a different architecture and measures instructions rather than cycles; our $1.42\times$ cycle ratio on Cortex-M4 reflects a different trade-off space but confirms that Rijndael-256 imposes a moderate, not prohibitive, overhead relative to AES-256.

https://doi.org/10.31436/iiumej.v27i2.4405

7. DISCUSSION

7.1. Fixslicing Optimality is Cipher-Specific

Our central finding is that fixslicing is suboptimal for Rijndael-256 while being optimal for AES-128, and this has implications for the implementation of other Rijndael variants. For AES-128 ($N_b = 4$), the column field width $g = 8$ bits ensures that all column shifts are byte-aligned, allowing the barrel shifter to absorb them at zero instruction cost. For Rijndael-256 ($N_b = 8$), $g = 4$ bits makes column shifts sub-byte, requiring per-row `UBFX/BFI` manipulation. This distinction is a property of the interaction between the cipher's block size N_b and the target register width W, not an inherent limitation of the fixslicing technique itself.

7.2. The 1-Table Advantage on ARM

The 20% performance advantage of 1-table over 4-table on Cortex-M4 at `-O2` is architecturally significant. On RISC-V RV32I, the additional cycles for software rotation would reduce or eliminate this advantage. Both this result and Saarinen's findings [6] demonstrate that the relative cost of implementation strategies depends heavily on available ISA primitives.

7.3. Compiler Sensitivity

A 6% performance loss occurs when transitioning from `-O2` to `-O3` in the bitsliced implementation, primarily due to code size expansion (from 2,204 to 3,732 bytes) that exceeding the instruction cache working set. This results in increased cache misses, negating the advantages of aggressive optimization. Conversely, the full assembly variants remain unaffected due to their compact code size.

7.4. Macro Inlining: Throughput vs. Code Size Trade-off

The two assembly variants expose a fundamental trade-off between throughput and flash footprint that is particularly consequential in constrained IoT deployments. The macro-inlined variant (Full ASM (macro)) expands all 14 round functions, each comprising `ark_sbox`, `shiftrows_rj256`, and `mixcolumns_rj256`, at their respective call sites, eliminating `BL/BX` branching and `PUSH/POP` register save/restore overhead. This yields a 319-cycle reduction per encryption (6,518 to 6,199 cycles, or 4.9%), at the cost of 15,240 additional bytes of flash (658 to 15,898 bytes, a 24.2 × increase).

To contextualize this trade-off, a device with 16 KB of flash can accommodate the function-call variant alongside a minimal protocol stack, whereas the macro-inlined variant alone consumes nearly the entire flash of a 16 KB device and is therefore only suitable for targets with at least 32 KB of flash. For high-throughput applications on flash-unconstrained devices, the macro variant offers a measurable throughput advantage. For typical IoT sensor nodes, the function-call variant is the practical choice.

This disproportion, large code size increase for a modest cycle reduction, arises because the Rijndael-256 round function is substantially larger than its AES-128 counterpart (4 × 8 vs. 4 × 4 state). On AES-128, fixsliced macro inlining yields a more favorable ratio because the round function is smaller. This further illustrates that the implementation strategy must be tailored to the specific cipher and target constraints.

https://doi.org/10.31436/iiumej.v27i2.4405

7.5. Practical Implications for IoT Deployment

At 203.7 cycles/byte, the function-call assembly variant achieves approximately 825 KB/s at 168 MHz, which is sufficient for typical IoT workloads; a 256 KB firmware image would encrypt in approximately 0.318 seconds. Using the STM32F407 active-mode current of approximately 87 mA at 168 MHz and a 3.3 V supply, the estimated active energy is about 11.1 µJ per 32-byte block, 0.348 µJ/byte, or 91 mJ for a 256 KiB firmware image. In CTR mode, a split pack/encrypt-loop/unpack implementation could reduce the per-block cost by amortizing packing and unpacking across consecutive blocks, though such restructuring is beyond the scope of this work.

For DTLS-secured sensor data with typical payloads of 64–256 bytes, the encryption cost is 2–8 Rijndael-256 block operations, completing in approximately 78–310 µs on the function-call assembly variant and consuming roughly 22–89 µJ of active energy under the estimate above. The 658-byte ASM core footprint of the function-call variant allows the cipher to coexist with protocol stacks even on devices with as little as 16 KB of flash. The macro-inlined variant (15,898 bytes) is more appropriate for targets with larger flash budgets where maximum throughput is the priority.

For deployments where timing-side-channel resistance is not required (e.g., isolated sensor nodes without physical access), the T-table 1T variant at 101.6 cycles/byte offers higher throughput at 1 KB RAM cost, but it should not be used when cache-timing leakage is within the threat model.

7.6. Limitations and Future Work

This work focuses on the encryption direction, which suffices for the dominant use cases of Rijndael-256 in practice: CTR, GCM, and CCM modes require only the block cipher's forward permutation. The implementation is single-block and software-only on ARM Cortex-M4. Decryption requires InvSubBytes, InvShiftRows, and InvMixColumns; the bitsliced InvMixColumns can be decomposed as $\mathrm{InvMC}(x) = \mathrm{MC}(M_{extra}(x))$ [11], enabling the reuse of the existing MixColumns circuit with a preprocessing step. A complete decrypt implementation is left as future work.

The constant-time property established here addresses the timing side-channel threat model. Power and electromagnetic side-channel attacks constitute a distinct threat class requiring masking countermeasures, and fault attacks require separate redundancy or detection mechanisms. The bitsliced representation is structurally amenable to first-order Boolean masking; the linear layers are maskable at negligible cost, and the Boolean-domain operations eliminate the data-dependent memory accesses that complicate masking of T-table implementations. A complete first-order masked implementation would require a DOM-based masked S-Box circuit and a masked key schedule. This remains future work rather than a claim of present resistance.

Extending this analysis to Rijndael-192 and to other 32-bit architectures (RISC-V RV32IM, Cortex-M33) would further validate our findings. Direct power-trace measurements would complement the cycle-count analysis and the back-of-the-envelope energy estimate reported in Section 7.5.

8. CONCLUSION

We have presented, to the best of our knowledge, the first constant-time bitsliced implementation of Rijndael-256 on the ARM Cortex-M4. This work addresses a practical need arising from NIST's ongoing consideration of Rijndael-256 for standardization (SP 800-197).

Our principal technical contribution is a quantitative analysis demonstrating that fixslicing becomes suboptimal for Rijndael-256 on this platform. The ShiftRows period $T = 8$ for $N_b = 8$ requires eight MixColumns compensation variants, costing on average $3.00\times$ more than explicit ShiftRows, making the semi-fused approach 17% faster, inverting the relationship established for AES-128.

The macro-inlined assembly variant achieves 193.7 cycles/byte, and the function-call variant achieves 203.7 cycles/byte, both with zero RAM overhead. The function-call variant requires only 658 bytes of flash, while the macro-inlined variant trades 15,898 bytes of flash for a 319-cycle savings per block. The constant-time overhead of approximately 1.91× relative to the T-table baseline (6,199 vs. 3,252 cycles) represents a practical cost for timing side-channel resistance on constrained devices.

These results demonstrate that the optimal bitslicing strategy depends on the interaction between cipher-specific ShiftRows offsets, block size, and the target architecture's instruction set. As Rijndael-256 moves toward potential standardization, implementers should evaluate rather than assume that techniques optimal for AES-128 will transfer to other Rijndael variants. The source code and measurement artifacts are available from the corresponding author upon reasonable request and will be archived publicly upon acceptance.

ACKNOWLEDGEMENT

The work of Andriani Adi Lestari was supported by Lembaga Pengelola Dana Pendidikan (LPDP), Ministry of Finance of the Republic of Indonesia, under Contract SKPB-4400/LPDP/LPDP.3/2023.

REFERENCES

[1] National Institute of Standards and Technology. (2023) Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS) 197-upd1. doi:10.6028/nist.fips.197-upd1

[2] Dworkin MJ. (2007) Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. doi:10.6028/NIST.SP.800-38D

[3] McGrew DA, Viega J. (2005) The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In: Progress in Cryptology -- INDOCRYPT 2004. Springer Berlin Heidelberg: 343-355. doi:10.1007/978-3-540-30556-9_27

[4] Avanzi R, Chakraborti A, Chakraborty B, List E. (2025) The Large Block Cipher Vistrutah. IACR Transactions on Symmetric Cryptology. 2025(3): 1-150. doi:10.46586/tosc.v2025.i3.1-150

[5] National Institute of Standards and Technology. (2024) Initial Pre-Draft Call for Comments on SP 800-197. NIST Cybersecurity and Privacy Reference Tool. URL: https://csrc.nist.gov/pubs/sp/800/197/iprd

[6] Saarinen MJO. (2025) Brief Comments on Rijndael-256 and the Standard RISC-V Cryptography Extensions. Cryptology ePrint Archive, Paper 2025/1198. URL: https://eprint.iacr.org/2025/1198

[7] Mordor Intelligence. (2025) IoT Microcontroller Market Size & Share Analysis–Growth Trends and Forecast (2025–2030). Online; accessed Apr. 2026. URL: https://www.mordorintelligence.com/industry-reports/iot-microcontroller-market

[8] IoT Analytics. (2025) IoT Microcontrollers Market Report 2025–2030. Market Report, Oct. 2025. URL: https://iot-analytics.com/product/iot-microcontroller-market-report-2025-2030/

[9] Adomnicai A, Peyrin T. (2021) 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. doi:10.46586/tches.v2021.i1.402-425

[10] Reparaz O, Balasch J, Verbauwhede I. (2017) Dude, is my code constant time? In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017: 1697-1702. doi:10.23919/DATE.2017.7927267

[11] Daemen J, Rijmen V. (2002) The Design of Rijndael: AES — The Advanced Encryption Standard. Springer-Verlag, Berlin, Germany.

[12] Biham E. (1997) A Fast New DES Implementation in Software. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 1267. Springer Berlin Heidelberg: 260-272. doi:10.1007/BFb0052352

[13] Boyar J, Peralta R. (2012) A Small Depth-16 Circuit for the AES S-Box. In: Information Security and Privacy Research, IFIP Advances in Information and Communication Technology, vol. 376. Springer Berlin Heidelberg: 287-298. doi:10.1007/978-3-642-30436-1_24

[14] Schwabe P, Stoffelen K. (2017) All the AES You Need on Cortex-M3 and M4. In: Selected Areas in Cryptography -- SAC 2016, Lecture Notes in Computer Science, vol. 10532. Springer: 180-194. doi:10.1007/978-3-319-69453-5_10

[15] Bernstein DJ. (2005) Cache-Timing Attacks on AES. Technical report. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[16] Almeida JB, Barbosa M, Barthe G, Dupressoir F, Emmi M. (2016) Verifying Constant-Time Implementations. In: 25th USENIX Security Symposium. USENIX Association: 53-70.

[17] Adomnicai A, Najm Z, Peyrin T. (2020) Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2020(3): 402-427. doi:10.13154/tches.v2020.i3.402-427

[18] ARM Limited. (2010) Cortex-M4 Technical Reference Manual. DDI 0439B. URL: https://developer.arm.com/documentation/ddi0439/b

[19] STMicroelectronics. (2024) RM0090 Reference Manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 Advanced Arm-based 32-bit MCUs. RM0090, Rev. 21.

[20] Käsper E, Schwabe P. (2009) Faster and Timing-Attack Resistant AES-GCM. In: Cryptographic Hardware and Embedded Systems -- CHES 2009, Lecture Notes in Computer Science, vol. 5747. Springer Berlin Heidelberg: 1-17. doi:10.1007/978-3-642-04138-9_1