Optimizing AES-GCM on ARM Cortex-M4: A Fixslicing and FACE-Based Approach

Hyunjun Kim[1][0000−0001−6757−6109] and Hwajeong Seo[2][0000−0003−0069−9061]

[1] Department of Information and Computer Engineering, Hansung University, Seoul, Republic of Korea
khj930704@gmail.com
[2] Department of Convergence Security, Hansung University, Seoul, Republic of Korea
hwajeong84@gmail.com

Abstract. The Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) delivers both confidentiality and integrity yet poses performance and security challenges on resource-limited microcontrollers. In this paper, we present an optimized AES-GCM implementation for the ARM Cortex-M4 that combines Fixslicing AES with the FACE (Fast AES-CTR Encryption) strategy, significantly reducing redundant computations in AES-CTR. We further examine two GHASH implementations—a 4-bit Table-based approach and a Karatsuba-based constant-time variant—to balance speed, memory usage, and resistance to timing attacks. Our evaluations on an STM32F4 microcontroller show that Fixslicing+FACE reduces AES-128 GCTR cycle counts by up to 19.41%, while the Table-based GHASH achieves nearly double the speed of its Karatsuba counterpart. These results confirm that, with the right mix of bitslicing optimizations, counter-mode caching, and lightweight polynomial multiplication, secure and efficient AES-GCM can be attained even on low-power embedded devices.

Keywords: AES-GCM · ARM Cortex-M4 · Optimization Implementation · Embedded Systems.

1 Introduction

In small embedded systems, such as Internet of Things (IoT) devices or wearable devices, ensuring secure communications is particularly difficult because of limited computational performance, memory, and energy. Because these devices exchange sensitive information over networks, cryptographic algorithms that guarantee confidentiality, integrity, and authentication are essential. It is standardized as an AEAD (Authenticated Encryption with Associated Data) mode in TLS (Transport Layer Security) and is currently widely used in various security protocols. However, a straightforward software implementation of AES-GCM on resource-constrained microcontrollers (e.g., ARM Cortex-M4) often leads to performance degradation and vulnerability to timing attacks.

One of the main challenges in implementing AES-GCM for embedded environments lies in the limited computational power and cache architecture of the microcontroller. For instance, an ARM Cortex-M4 lacks specialized cryptographic instructions and high-performance caches, causing block cipher operations to take much longer than on desktop or server platforms [26]. Furthermore, Table-based AES implementations are prone to side-channel attacks that exploit cache hit/miss patterns to infer internal round keys [4, 6]. Consequently, there has been extensive research into efficient, constant-time AES implementations that avoid table lookups, even on microcontroller unit (MCU) platforms [24, 14, 2]. Meanwhile, GHASH is the authentication tag generator in GCM. It relies on multiplication in $\mathrm{GF}(2^{128})$. Although x86/AMD64 architectures support instructions like `PCLMULQDQ` [10], and ARMv8 supports `PMULL` [7], the Cortex-M4 lacks such dedicated instructions, forcing GHASH to depend solely on general arithmetic and logical operations [3]. This limitation significantly increases the overhead of GHASH computations.

In this study, we propose techniques to optimize AES-GCM's two core components. Specifically, AES-CTR and GHASH are the targets, and we aim to make them both more secure and faster on embedded devices. For AES-CTR, we apply Fixslicing AES [2] to eliminate Table-based operations and integrate it with the FACE (Fast AES-CTR Encryption) technique [18] to minimize redundant computations. Fixslicing replaces AES S-box operations with bit-wise logical functions, thereby mitigating cache-based timing attacks [1] while achieving high efficiency. Consequently, AES-128 can operate at roughly 80 cycles/byte on an ARM Cortex-M4 [24]. Building on this, we incorporate the FACE idea of "reusing fixed segments of the CTR counter" to further reduce unnecessary operations for large-message processing [18].

For GHASH, various methods have been proposed to accelerate $\mathrm{GF}(2^{128})$ multiplication on resource-limited MCUs, such as 4/8/16-bit Table-based algorithms [16] and Karatsuba multiplication [8]. Although using 8-bit or 16-bit tables can theoretically cut down the number of multiplication operations, it requires a large amount of Flash/ROM, which is problematic in real-world embedded settings. Therefore, in this work, we focus on two methods that balance memory footprint and security. Specifically, we consider a 4-bit Table-based approach and a Karatsuba-based constant-time approach, and compare them in terms of performance, memory usage, and security. To address these challenges, we adopt a 4-bit (nibble) Table-based method that reduces the multiplication process to 32 iterations, and compare it against a Karatsuba-based constant-time implementation. The 4-bit table method can achieve a significant speed advantage on the Cortex-M4, which has minimal cache, but does not strictly achieve constant time. As a result, depending on security requirements (e.g., cache availability, memory constraints, and authentication speed), either method can be chosen. The main contributions of this paper are as follows:

1. **CTR Mode Optimization:** By combining Fixslicing AES with the FACE technique, we reduce the cycle count for AES-128 GCTR on ARM Cortex-M4

by up to 19.41% and for AES-256 GCTR by up to 14.63% when processing large messages (e.g., 40,KB).
2. **GHASH Acceleration:** We present a 4-bit Table-based GF(2^{128}) multiplication method and experimentally demonstrate that it can be up to twice as fast as a Karatsuba-based constant-time implementation.
3. **Comprehensive Embedded Evaluation:** We analyze the trade-offs in memory usage, performance, and security for various AES-CTR and GHASH configurations, providing AES-GCM solutions optimized for different embedded scenarios (e.g., cache architecture, memory constraints, and authentication requirements).
4. We open-source our code to encourage reproducibility and future development at [upon the completion of the review process].

The remainder of this paper is organized as follows. Section 2 introduces background concepts and related work on Fixslicing AES, FACE, and GF(2^{128}) multiplication. In Section 3, we describe how to integrate Fixslicing AES with the FACE technique, along with key implementation considerations. Section 4 compares the pros and cons of the 4-bit Table-based GHASH approach and the Karatsuba-based approach. Section 5 presents performance results for AES-GCM on a Cortex-M4 across different message lengths. Finally, Section 6 concludes the paper and suggests avenues for future research.

2 Background

2.1 AES and GCM

AES (Advanced Encryption Standard). AES is a symmetric-key block cipher standardized by the U.S. National Institute of Standards and Technology (NIST) [25]. It encrypts 128-bit blocks using 128-, 192-, or 256-bit keys, performing 10, 12, or 14 rounds, respectively. Each round applies four transformations:

- SubBytes: Non-linear byte substitution using a fixed S-box.
- ShiftRows: Cyclic row shifting in a 4×4 byte matrix.
- MixColumns: Polynomial-based mixing over GF(2^8) (omitted in the final round).
- AddRoundKey: XOR with the round key, derived from the main key.

CTR (Counter) Mode. CTR mode transforms a block cipher into a stream cipher by encrypting a counter combined with an a unique initialization vector (IV). For each block, an incremented counter is encrypted, and the result is XORed with the plaintext to produce the ciphertext:

$$C_i = P_i \oplus E_K(J_i).$$

Because the counter changes for each block and the IV is never reused, CTR provides confidentiality but requires an additional mechanism for integrity.

AES-GCM AES-GCM combines AES-CTR with the GHASH authentication function over GF(2^{128}) [16], as shown in Figure 1. A hash subkey H is derived by encrypting an all-zero block with AES. The additional data (AD) and ciphertext blocks are then processed by GHASH:

$$X_i \leftarrow (X_{i-1} \oplus B_i) \times H,$$

after which a final length block is appended before XORing the GHASH output with an AES encryption to form the authentication tag T. Typically, a 96-bit IV is recommended for GCM, and its parallel-friendly structure makes it well suited for high-speed protocols such as TLS.

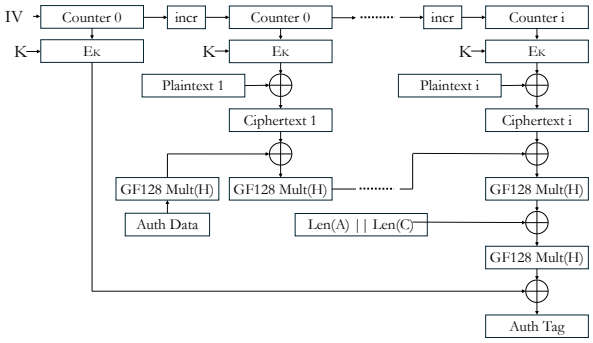


Fig. 1: Overview of AES-GCM encryption. The GHASH function processes the AD and ciphertext while AES-CTR provides confidentiality. The final authentication tag T is derived from the GHASH output and an AES-encrypted value.

2.2 Bitslicing Implementation

Bitslicing was originally introduced by Biham [5] to implement DES more efficiently. Rather than relying on S-box table lookups, it uses parallel, bitwise logical operations, thereby eliminating data-dependent memory accesses and reducing cache-based side-channel vulnerabilities. Instead of storing 8-bit values in a table, bitslicing distributes each bit across multiple registers so that a single logic instruction can process many blocks in parallel. Modern processors handle bitwise operations very efficiently [11, 17, 9], making bitslicing particularly advantageous for large data sets. Although this technique requires an initial bit interleaving step and may increase code size, it has been shown to be highly effective for AES and other ciphers on a wide range of platforms [23, 13, 12].

2.3 ARM Cortex-M4 Processor

The ARM Cortex-M4 is a 32-bit RISC processor based on the ARMv7-M architecture, widely used in low-power embedded systems such as IoT devices,

sensor nodes, and real-time control applications. While the Cortex-M4 includes digital signal processing (DSP) extensions—such as multiply-accumulate and barrel-shift instructions—it does not provide dedicated AES or cryptographic instructions [3]. Consequently, ciphers like AES and operations such as GHASH must be implemented via general-purpose arithmetic and logical operations.

Despite lacking cryptographic instructions, the Cortex-M4 includes several DSP-oriented features that can accelerate bitslicing cryptographic code. For example, the single-cycle `MUL` instruction (for certain operand sizes) and multiply-accumulate instructions can speed up polynomial multiplications in GHASH. The barrel shifter allows arithmetic and logical shifts or rotations in a single cycle with another operation, effectively combining two operations into one. These features prove useful in bitslicing implementations, where rotations and bit manipulations are frequent.

With only 16 general-purpose 32-bit registers (several of which are reserved for the program counter, stack pointer, and link register), the Cortex-M4 has fewer general-purpose register than desktop-class CPUs. bitslicing implementations that process multiple blocks in parallel must therefore carefully manage register usage, spilling values to the stack or memory. Even with these constraints, hand-optimized bitslicing AES on the Cortex-M4 can reach impressive speeds, 80 cycles/byte for AES-128 [2]. These results underscore that even resource-constrained microcontrollers can exploit bitslicing's parallelism, provided the code is carefully tuned to the pipeline, memory layout, and register limits of the ARMv7-M architecture.

2.4 Notation

Throughout this paper, we consistently use the notation presented in Table 1. We denote the i-th byte of the AES state as $S[i]$. Because AES operates on 16-byte blocks, $S[0]$ corresponds to the most significant byte (MSB), and $S[15]$ corresponds to the least significant byte (LSB). To clarify the handling of multiple data blocks, we write $S_i[j]$ for the j-th byte of the i-th block. Additionally, $X_i[j]$ represents the j-th column of the AES state (i.e., a round state) for the i-th block. In a bitsliced implementation (e.g., Fixslicing), the AES state is distributed across multiple 32-bit registers R_i. Within each 32-bit register R_i, we refer to its i-th bit specifically as b_i for fine-grained manipulation. Furthermore, we define four variants of the MixColumns transformation in Fixslicing AES, denoted by `MixColumns0` through `MixColumns3`. Lastly, `Ark_Sub` refers to a combined routine of `AddRoundKey` and `SubBytes`.

2.5 Overview of Fixslicing AES

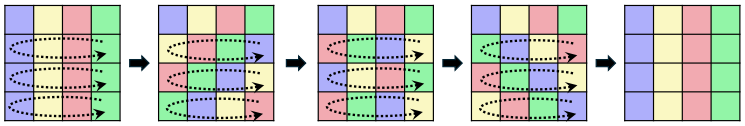
Fixslicing AES [2] is a specialized variant of bitslicing AES that aims to reduce overhead on 32-bit embedded processors by keeping each bit fixed in a register throughout the entire encryption process. Unlike conventional bitslicing, which frequently rearranges bits (particularly for `ShiftRows`), Fixslicing uses a single consistent representation across all rounds. This strategy significantly cuts

Table 1: Notation

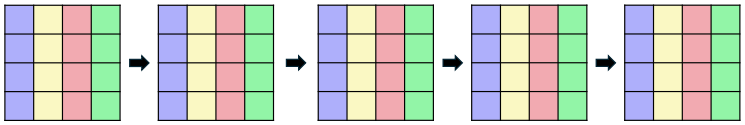
Notation	Description
$\mathbf{S[i]}$	i-th byte of the AES state ($i \in \{0, \ldots, 15\}$)
$\mathbf{S_i[j]}$	j-th byte ($j \in \{0, \ldots, 15\}$) of the i-th block's AES state
$\mathbf{X_i[j]}$	j-th column ($j \in \{0, 1, 2, 3\}$) of the AES round state in the i-th block
$\mathbf{R_i}$	i-th 32-bit register in a bitsliced implementation (e.g., Fixslicing)
$\mathbf{b_i}$	i-th bit of a 32-bit register R
MixColumns0 $\sim$ 3	Four variations of MixColumns in Fixslicing AES (one per round)
`Ark_Sub`	Combined routine of `AddRoundKey` and `SubBytes` in Fixslicing AES

down on data shuffling, preserves constant-time execution, and can yield faster performance on devices like the ARM Cortex-M4.

Eliminating Repeated Bit Shuffling. In standard bitslicing AES, each round may involve explicit row shifts or additional permutations to handle `ShiftRows`. By contrast, Fixslicing encodes the state so that `ShiftRows` is implicitly "merged" into a sequence of round-dependent `MixColumns` variants. After carefully choosing how bits map to registers, the code no longer requires rearranging the internal state to implement row shifting. This "fixes" each bit in place for all rounds, removing a major source of overhead in classical bitslicing AES (see Figure 2).



(a) Classical



(b) Fully-fixed

Fig. 2: Overview of the AES internal state over 4 rounds under different representations [2].

Merging `SubBytes` and `AddRoundKey`. Because Fixslicing distributes each byte's bits across multiple 32-bit registers, traditional byte-oriented S-box lookups are replaced by Boolean circuits or short bit-wise logic sequences. Furthermore, Fixslicing often merges `AddRoundKey` into `SubBytes` (sometimes referred to as `Ark_Sub`), since XOR operations can be absorbed into the bit-wise S-box at

negligible extra cost. This merging shortens the overall round sequence, thereby reducing instruction counts and memory references.

Modified `MixColumns` Variants. In standard AES, each round uses the same `MixColumns` polynomial on four columns, with `ShiftRows` providing the byte rotations. In Fixslicing, `ShiftRows` is omitted entirely, so each round applies a slightly different version of `MixColumns`—often referred to as `MixColumns0`, `MixColumns1`, `MixColumns2`, and `MixColumns3`. Over four rounds, these variants collectively achieve the same row-shifting effect without the runtime cost of additional shuffling. Although this approach expands the code base, it avoids the overhead of dynamic data rearrangement. After the final round, a small adjustment step realigns the output into the standard AES format.

Packing and Unpacking in Fixslicing. As with general bitslicing, Fixslicing requires an initial packing step in which plaintext bytes (and later, round keys) are loaded into registers in a bitsliced form. In many Cortex-M4 implementations, two 128-bit plaintext blocks (256 bits total) are distributed across eight 32-bit registers. This layout allows core bit-wise operations (`AND`, `OR`, `EOR`, etc.) to process multiple blocks in parallel. Figure 3 illustrates how the bits of two blocks can be "sliced" and placed into eight registers on a Cortex-M4.

	row 3								⋯	row 0							
	column 0		column 1		column 2		column 3		⋯	column 0		column 1		column 2		column 3	
	block 0	block 1	block 0	block 1	block 0	block 1	block 0	block 1		block 0	block 1	block 0	block 1	block 0	block 1	block 0	block 1
R_0	b^0_{24}	b^1_{24}	b^0_{56}	b^1_{56}	b^0_{88}	b^1_{88}	b^0_{120}	b^1_{120}	⋯	b^0_0	b^1_0	b^0_{32}	b^1_{32}	b^0_{64}	b^1_{64}	b^0_{96}	b^1_{96}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
R_7	b^0_{31}	b^1_{31}	b^0_{63}	b^1_{63}	b^0_{95}	b^1_{95}	b^0_{127}	b^1_{127}	⋯	b^0_7	b^1_7	b^0_{39}	b^1_{39}	b^0_{71}	b^1_{71}	b^0_{103}	b^1_{103}

Fig. 3: bitslicing representation from [24] using 8 32-bit registers $R_0, \ldots, R_7$ to process two blocks b^0, b^1 in parallel. Here b^i_j denotes the j-th bit of the i-th block, illustrating how Fixslicing distributes bits across registers after the packing step [2].

After encryption, an unpacking step reassembles the bitsliced ciphertext back into the standard byte-oriented format. While these packing and unpacking routines add a fixed overhead, the cost is significantly amortized when encrypting large messages (e.g., in `CTR` or GCM mode).

Fixslicing aligns well with the ARM Cortex-M4 instruction set. Logical operations (e.g., `AND`, `EOR`) and barrel shifts are highly efficient, and avoiding large S-box lookups eliminates potential timing channels or cache misses. Benchmarks show that Fixslicing AES-128 on a Cortex-M4 can run at 80 cycles/byte, roughly 20% faster than older constant-time bitslice approaches.

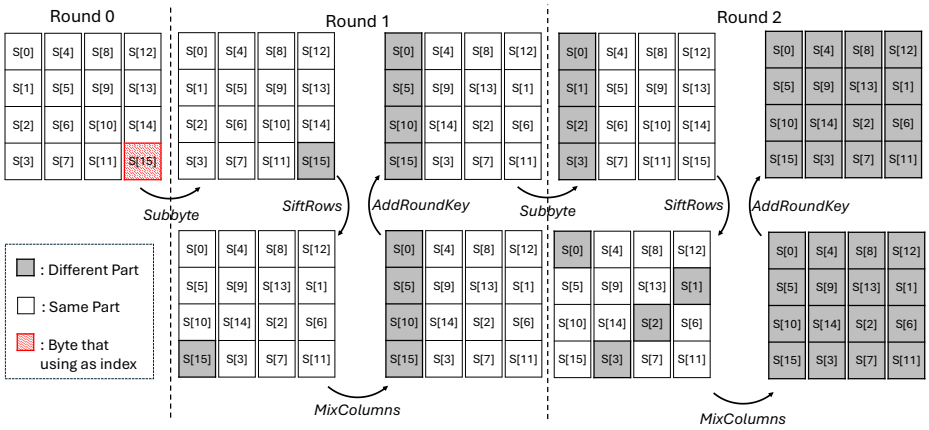


Fig. 4: Diffusion of the state difference between the first and second blocks in Rounds1 and Rounds2 of AES-CTR.

2.6 Overview of FACE

FACE (Fast AES-CTR Encryption) [18] is a set of techniques designed to reduce redundant computations when encrypting multiple blocks in `CTR` mode. In `CTR` mode, each block is formed by combining a nonce (IV) with an incrementing counter, which is then encrypted with AES. Typically, only a small portion of the counter bytes change from one block to the next, while the rest of the block input remains the same.

As shown in Figure 4, FACE exploits this property by caching partial AES round outputs—focusing on portions unaffected by the counter increment—so that large parts of the encryption process need not be recomputed for each subsequent block.

FACE provides five caching variants, each progressively capturing more partial-round data and thus offering different trade-offs between memory usage and performance. Table 2 summarizes these key variants.

$\texttt{FACE}_{rd0}$ (Round-0 caching) $\texttt{FACE}_{rd0}$ caches the result of the initial `AddRoundKey` (the "whitening" step) for all but the single counter byte that changes across blocks. This approach precomputes 12 of the 16 state bytes, using 12 bytes of extra memory. By avoiding about 75% of Round 0's repeated operations, it offers a modest speedup.

$\texttt{FACE}_{rd1}$ (Round-1 caching) $\texttt{FACE}_{rd1}$ extends caching into the first AES round, storing the three unchanged state columns after Round 1. This requires only 12 bytes of additional memory and yields a greater speedup than $\texttt{FACE}_{rd0}$, as three-quarters of the Round 1 operations are skipped for each new block.

FACE$_{rd1+}$ (Round-1 plus caching) FACE$_{rd1+}$ builds on FACE$_{rd1}$ by also caching the variable part of Round 1 through a precomputed lookup table (256 entries) for the first column. This table demands 1 KB of memory and significantly boosts performance, replacing the remaining Round 1 computations with simple table lookups and XOR merges.

FACE$_{rd2}$ (Round-2 caching) FACE$_{rd2}$ further extends caching into the second AES round by reusing previously cached intermediate values and storing partial Round 2 results. Thus, only the newly affected bytes must be processed in Round 2, incurring minimal extra memory while delivering a substantial performance gain. As shown below, the operational part of the unchanged state bytes in each column is cached. After ShiftRows, the 4-byte result of MixColumns+AddRoundKey for the column into which S[5], S[10], S[15] of the first column are placed is cached.

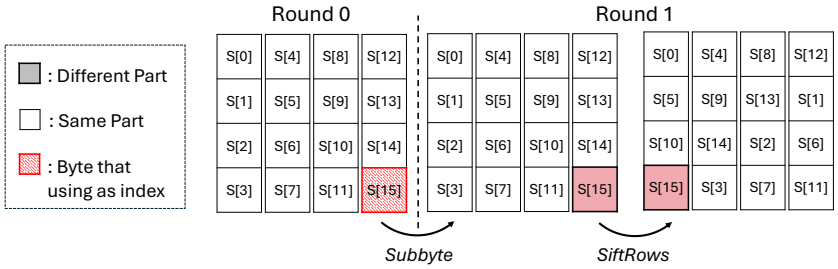
$$
\begin{aligned}
3 \cdot S[5] \oplus 1 \cdot S[10] \oplus 1 \cdot S[15] \oplus roundkey_{2,0},\\
2 \cdot S[5] \oplus 3 \cdot S[10] \oplus 1 \cdot S[15] \oplus roundkey_{2,1},\\
1 \cdot S[5] \oplus 2 \cdot S[10] \oplus 3 \cdot S[15] \oplus roundkey_{2,2},\\
1 \cdot S[5] \oplus 1 \cdot S[10] \oplus 2 \cdot S[15] \oplus roundkey_{2,3}.
\end{aligned}
$$

FACE$_{rd2+}$ (Round-2 plus caching) FACE$_{rd2+}$ builds on FACE$_{rd2}$, precomputing Round 2 outcomes for the changing byte using a 256-entry table that requires 4 KB of memory. With this table, the first two rounds can be performed via lookups and XOR operations alone, yielding the highest speedup among the FACE variants.

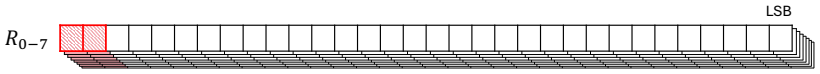
These caching strategies exploit the fact that consecutive CTR blocks differ only in a few bits of the incremented counter, leaving most of the round computations invariant. Consequently, any large segment of round operations that remains unchanged can be reused, accelerating the encryption of subsequent blocks.

Variant	Caching Idea	Memory Overhead	Reset Interval (Blocks)
FACE$_{rd0}$	Cache AddRoundKey for static bytes	12 bytes	2^8
FACE$_{rd1}$	Extend caching into Round 1	12 bytes	2^8
FACE$_{rd1+}$	256-entry lookup for Round 1	1 KB	2^{40}
FACE$_{rd2}$	Cache Round 2 intermediates	16 bytes	2^8
FACE$_{rd2+}$	Precompute Round 2 via table	4 KB	2^{40}

Table 2: Key characteristics of each FACE variant.



(a) Standard AES



(b) Fixslicing AES

Fig. 5: 1st round MixColumns The difference between the 1st and 2nd block states. Fixslicing AES does not change the position of the modified portion.

3 Integrating Fixslicing AES with FACE

By applying the FACE method to reuse any unchanged parts of the state, we can reduce redundant operations. At the same time, we preserve both constant-time behavior and the high efficiency of Fixslicing, ultimately boosting overall throughput. Indeed, Park et al. [18] have shown that FACE can be applied efficiently to various AES implementation strategies—such as Table-based, bitslicing, or AES-NI—without depending on a specific implementation style.

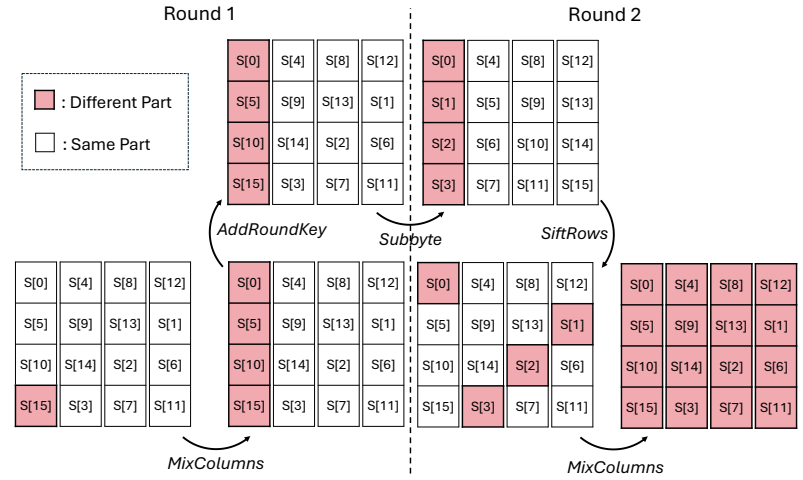
However, detailed explanations of FACE are primarily focused on conventional (Table-based) AES, with only brief performance comparisons given for bitslicing and AES-NI implementations. In other words, there is a lack of concrete examples demonstrating how to incorporate the five FACE caching methods into a bitslicing context. Furthermore, Fixslicing differs internally from standard bitslicing, requiring additional care in directly applying the five FACE caching methods. This section therefore proposes a concrete approach for integrating FACE into a Fixslice AES-based implementation.

3.1 Adapted FACE_{rd0}

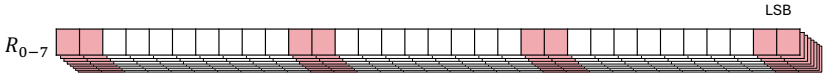
FACE_{rd0} caches the state immediately after the initial `AddRoundKey` in AES, thereby reducing redundant operations (e.g., the XOR involved in `AddRoundKey`) for subsequent blocks. Typically, a Fixslicing AES implementation follows the sequence:

$$\text{Packing} \rightarrow \texttt{AddRoundKey} \rightarrow \texttt{SubBytes}$$

Because `AddRoundKey` occurs after the bitslicing state has been packed, the main challenge in applying FACE_{rd0} lies in deciding which state (i.e., after which transformation) to cache.



(a) Standard AES



(b) Fixslicing AES

Fig. 6: The difference between the 1st and 2nd block before the 2nd round MixColumns. Fixslicing AES does not change the position of the Different Part.

One naive approach would be to unpack the bitslicing state back into a conventional AES-like format, apply caching, then re-pack. However, packing and unpacking can introduce a nontrivial cycle overhead, which may be significant compared to the rest of the encryption process. Hence, a strategy that applies FACE while preserving the bitslicing state is essential.

Depending on the implementation approach, there are two main methods: `(i)` maintaining the Fixslicing sequence by performing `Packing`, XOR with the packed round key, and then caching the resulting state, or `(ii)` performing XOR with an unpacked round key and then caching.

In the first method, the changing bytes are first `Packed`, then XORed with the packed round key, after which they are XORed with the cached value; finally, `SubBytes` is performed. In the second method, the changing bytes are XORed with the unpacked round key, followed by XOR with the cached value, and then `Packing` and `SubBytes` are performed.

In both methods, only the 32 bits that change due to the counter are XORed for the initial `AddRoundKey`, thereby omitting three load instructions and three XOR instructions. However, there is little practical difference between the two methods, and since the second method is effectively the same as the original approach, no separate performance evaluation was conducted.

3.2 Adapted FACE_{rd1}

In `CTR` mode, when the counter increases by 1, the state change that starts as a single byte eventually spreads to an entire column after the Round-1 `MixColumns` step (see Figure 5a. Hence, FACE_{rd1} is designed to isolate only the portion $X_i[0]$ that changes after Round 1 while caching the unchanged portions $X_0[1]$, $X_0[2]$, $X_0[3]$ thereby reducing redundant operations.

In a typical byte-oriented (Table-based) AES implementation, `MixColumns` processes each column independently, making it simple to handle a specific column by itself. However, in a Bitslicing or Fixslicing implementation, the state is rearranged at the bit level, such that the bits belonging to one column can be scattered across multiple registers.

Because `MixColumns` of Fixslicing performs the same matrix multiplication on all columns simultaneously, extracting only a single column for separate processing offers no tangible benefit. Moreover, to isolate just that column, one would have to undo parts of the already-performed bit transpose and then reassemble those bits, which increases the overall complexity without providing a meaningful performance gain.

Therefore, by performing a parallel operation using the `MixColumns0` step from Fixslicing AES, we apply a mask to the bitslicingd result so that only the relevant bits remain.

Additionally, as shown in Figure 6b, the byte affected by the counter does not undergo any additional position transformations until before `MixColumns` in Round 2. Thus, caching the state up to the Round-2 `Ark_Sub` step allows for a high degree of reuse and improved efficiency.

Below is the detailed FACE_{rd1} process.

$$\begin{aligned}
&\text{in} = \{S_0[0], S_0[1], \ldots, S_0[15],\ S_1[0], S_1[1], \ldots, S_1[15]\} \\
&\xrightarrow{\text{packing}} \xrightarrow{\text{Ark_Sub}} \xrightarrow{\text{MixColumns0}} \xrightarrow{\text{Ark_Sub}} \xrightarrow{\wedge 0xFCF3CF3F} \\
&\text{FACE}_{\text{rd1}} = \{b_0, b_1, b_2, b_3, b_4, b_5, 0, 0,\ b_8, b_9, b_{10}, b_{11}, 0, 0, b_{14}, b_{15}, \\
&\qquad b_{16}, b_{17}, 0, 0,\ b_{20}, b_{21}, b_{22}, b_{23}, 0, 0,\ b_{26}, b_{27}, b_{28}, b_{29}, b_{30}, b_{31}\}
\end{aligned}$$

After completing the round-2 `Ark_Sub` step, we apply an `AND` operation with mask value $R_i \wedge 0xFCF3CF3F$ to the bitslicingd state across all registers, thus caching the bit positions corresponding to two blocks. In a typical byte-oriented implementation, storing the columns $(X_0[1], X_0[2], X_0[3])$ for a single block require 12 bytes of memory.

However, Fixslicing AES stores the columns $(X_0[1], X_0[2], X_0[3])$ for two blocks in a bitslicing manner, consuming 64 bytes of storage.

3.3 Adapted $\text{FACE}_{\text{rd1+}}$

$\texttt{FACE}_{\texttt{rd1+}}$ precomputes (and stores in a lookup table) the varying portion $X_i[0]$ that emerges after Round 1. In alignment with FACE_{rd1}, it can be configured to handle only $X_i[0], X_{i+1}[0]$ separately. Below is the detailed process.

$$\text{in} = \{S_i[0], S_i[1], \ldots, S_i[15],\ S_{i+1}[0], S_{i+1}[1], \ldots, S_{i+1}[15]\}$$
$$\xrightarrow{\text{packing}} \xrightarrow{\text{Ark\_Sub}} \xrightarrow{\text{MixColumns0}} \xrightarrow{\text{Ark\_Sub}} \xrightarrow{\wedge 0x030c30c0}$$
$$\text{FACE}_{\text{rd1+}} = \{0,0,0,0,0,0,b_6,b_7,\ 0,0,0,0,b_{12},b_{13},0,0,$$
$$0,0,b_{18},b_{19},0,0,0,0,\ b_{24},b_{25},0,0,0,0,0,0\}$$

By combining the cached FACE_{rd1} values with the counter, we can call the $\text{FACE}_{\text{rd1+}}$ table and XOR them, effectively processing through Round 2 `SubBytes`. This method, however, requires storing a 4,096-byte table—four times larger than the 1,024 bytes needed in previous $\text{FACE}_{\text{rd1+}}$.

Parallel-Processing Method In the single-processing method, the operation $R_i \wedge 0x030c30c0$ sets 24 bytes, meaning some input bytes ($S_0[0]$, $S_0[5]$, $S_0[10]$, etc.) do not contribute to the result.

The $\text{FACE}_{\text{rd1+}}$ parallel processing method is optimized by filling the 'empty' spaces to process eight columns at once, as shown in 7. Because it processes four times more columns in parallel, the overall operation count and memory usage can be significantly reduced.

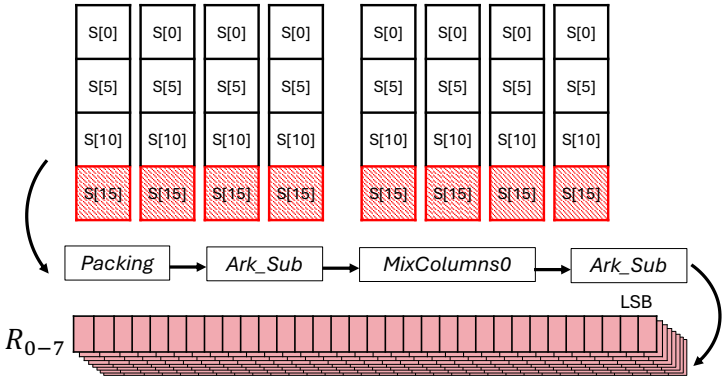


Fig. 7: Parallel processing method of $\text{FACE}_{\text{rd1+}}$. Shows that 8 columns are processed at once.

The first and second block columns remain in their original positions, and from the third block onward, each column is shifted one position to the right when stored. Since S0,0, S0,5, and S0,10 are effectively identical in each column, only the position of Si,15 needs to be considered.

In practice, assuming a little-endian environment, we sequentially store 32 bits in memory in the order of $S_0[15]$, $S_0[10]$, $S_0[5]$, $S_0[0]$ (eight bits each). Then, we increment the value by 1 using the ADD instruction. To align the position of Si,15, we stores the increased values in order, as shown below.

$$S_0[0], S_0[5], S_0[10], S_i[15] : R3 \quad S_0[0], S_0[5], S_0[10], S_{i+1}[15] : R7$$
$$S_0[0], S_0[5], S_0[10], S_{i+2}[15] : R2 \quad S_0[0], S_0[5], S_0[10], S_{i+3}[15] : R6$$
$$S_0[0], S_0[5], S_0[10], S_{i+4}[15] : R1 \quad S_0[0], S_0[5], S_0[10], S_{i+5}[15] : R5$$
$$S_0[0], S_0[5], S_0[10], S_{i+6}[15] : R0 \quad S_0[0], S_0[5], S_0[10], S_{i+7}[15] : R4$$

Finally, the REV instruction reverses the byte order of each register. After that, we perform operations up through the second round's Ark_Sub while preserving the bitslicingd state and storing the result.

With this approach, a total of 32 bytes (eight columns × 4 bytes) is used, and because 256 blocks are stored, the lookup table requires 1 KB of space.

After loading the first column (e.g., `R4`) from memory, each subsequent column is placed by incrementing the counter (via `ADD instruction`). As shown in Figure 7, we then apply

$$\texttt{packing} \rightarrow \texttt{Ark\_Sub} \rightarrow \texttt{MixColumns0} \rightarrow \texttt{Ark\_Sub},$$

When subsequent round operations proceed, the eight block columns stored in memory must be rearranged into the correct positions to combine with FACE_{rd1}. Because Fixslicing AES does not rearrange bits and maintains a consistent representation across all rounds, these positions correspond to the same S0, S5, S10, and S15 as the original input. The process of rearranging to maintain the Fixslice structure is shown in Fig 8. We apply masking and rotation operations to eight registers according to each block. Blocks 1 and 2 can directly extract bits using only the `0x030C30C0` mask, but Blocks 3 and 4, 5 and 6, and 7 and 8 partially use rotation and masking to move bits into the correct positions.

```
1 if (block < 2) // block 1,2
2     {out = x & 0x030C30C0;}
3 else if (block < 4) // block 3,4
4     {out = (ROR32(x, 2) & 0x030C3000) ^ ROR32(x & 3, 26); }
5 else if (block < 6) // block 5,6
6     {out = (ROR32(x, 4) & 0x030C0000) ^ (ROR32(x, 28) & 0
          x000030C0);}
7 else if (block < 8) // block 7,8
8     {t = ROR32(x, 30); out = (t & 0x000C30C0) ^ ROR32(t & 3,
          8);}
```

Fig. 8: C code for rearranging the FACErd1+ columns of each block, where ROR32 denotes a 32-bit right-rotation operation.

Round key modification In the $\texttt{FACE}_{\texttt{rd1+}}$ parallel-processing method, the Fixslice round key must be adjusted to align with the input layout. This adjustment is

done only once before encryption begins. Figure 9 shows the detailed procedure. From the packed round keys of Rounds 1 and 2, we extract the round key bits corresponding to S0,0, S0,5, S0,10, and S0,15, and then copy them to the matching positions.

```
for(int idx = 0; idx < 16; idx++){
    temp = (original_rk[idx] & 0x030c30c0);
    rk[idx] ^= temp;
    rk[idx] ^= ((temp & 0x3000000) << 6) |((temp & 0xC30C0)
        >> 2);
    rk[idx] ^= ((temp & 0x3000000) << 4) | ((temp & 0xC0000)
        << 4)
                | ((temp & 0x3000) >> 4) | ((temp & 0xC0) >>
                  4);
    rk[idx] ^= ((temp & 0x3000000) << 2) | ((temp & 0xC0000)
        << 2)
                | ((temp & 0x3000) << 2) | ((temp & 0xC0) >>
                  6);
}
```

Fig. 9: C code for round key modification

Table-Free Approach (FACE$_{rd1}$) We also introduce a variant of $\texttt{FACE}_{\texttt{rd1+}}$ that omits building any precomputation table. We also introduce a variant of $\texttt{FACE}_{\texttt{rd1+}}$ that omits building any precomputation table. In this scheme, we apply the parallel-processing version of $\texttt{FACE}_{\texttt{rd1+}}$ "on the fly," processing eight blocks in parallel, and then combine it with $\texttt{FACE}_{\texttt{rd1}}$ to handle computations up through the second round's `Ark_Sub`. As a result, by using only the $\texttt{FACE}_{\texttt{rd1}}$ cache without relying on $\texttt{FACE}_{\texttt{rd1+}}$, we avoid the cost and complexity of a large LUT.

3.4 Adapted $\texttt{FACE}_{\texttt{rd2}}$

As shown in Figure 6a, only $S[0], S[1], S[2], S[3]$ change before Round 2 `MixColumns`, while all bytes are affected later. Recognizing that the bytes outside $S[0]$, $S[1]$, $S[2]$, $S[3]$ remain constant, one can cache the intermediate results.

However, as with Round 1, splitting `MixColumns` column-by-column to handle only $S[0], S[1], S[2], S[3]$ is not well suited to Fixslicing. Consequently, we retain the standard Round 2 `MixColumns` step and then store the bitslicing state by masking only the changing bytes.

In a conventional FACE_{rd2} procedure, caches data up to the Round 2 `AddRoundKey` step. To accommodate the `Ark_Sub` structure in Fixslicing, however, we opt to store the state immediately after `MixColumns1`.

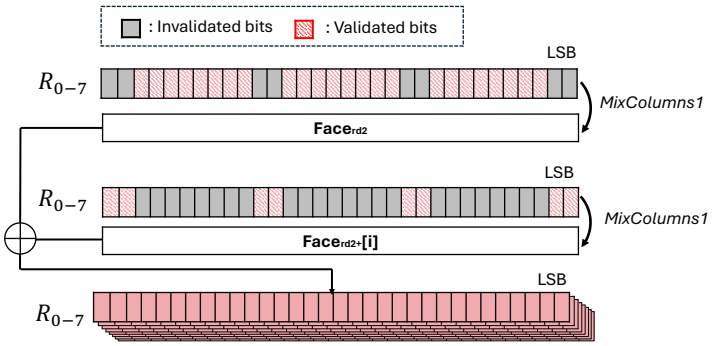


Fig. 10: Schematic of FACE_{rd2} and $\text{FACE}_{\text{rd2+}}$ on Fixslicing AES.

3.5 Adapted $\text{FACE}_{\text{rd2+}}$

$\text{FACE}_{\text{rd2+}}$ extends FACE_{rd2} by further precomputing the remaining intermediate values not covered in FACE_{rd2}. The approach parallels that in Section 3.4 FACE_{rd2}:

1. After `MixColumns1`, store the internal state in bitslicing form.
2. Mask (zero) bits outside of $(S[0], S[1], S[2], S[3])$.

Since this is essentially the same procedure as $\texttt{FACE}_{\texttt{rd1+}}$, one can implement it by feeding the $\texttt{FACE}_{\texttt{rd1+}}$ single-processing (or parallel-processing) output into this step. Hence, as depicted in Figure 10, the final operation simply XORs the cached FACE_{rd2} value with the $\text{FACE}_{\text{rd2+}}$ data to complete Round 2 (including `MixColumns`).

4 GF(2^{128}) Multiplication in GHASH

AES-GCM uses the GHASH function to authenticate 128-bit blocks by performing polynomial multiplication in GF(2^{128}). On x86 platforms, specialized instructions such as `PCLMULQDQ` allow fast carry-less multiplications. However, the ARM Cortex-M4 lacks such instructions, forcing GF(2^{128}) multiplication to rely on general-purpose arithmetic and logical operations. Our primary design goal is to directly compare a faster, small table-based GHASH implementation against a table-free, constant-time Karatsuba approach. Although both methods are well-known in the literature, we have integrated and evaluated them within the same AES-GCM framework on the ARM Cortex-M4. Below, we highlight the key details of each implementation.

4.1 Naive bit-wise Multiplication

The most straightforward method is to iterate over each bit of a 128-bit operand. To compute $X \times H$, for every set bit in X, we XOR the appropriately shifted H

into an accumulator. The procedure repeats for 128 iterations, as illustrated in the pseudo-code below:

```
Z = 0
for i in range(128):
    if ((X >> i) & 1) == 1:
        Z = Z ^ (H << i)
    # Modular reduction (if overflow occurs)
    Z = reduce(Z)
return Z
```

While simple to implement and virtually memory-free, this naive approach is computationally expensive. It involves 128 iterations, each with at least one shift and one conditional branch, creating a performance bottleneck on resource-constrained platforms.

4.2 4-Bit Table-Based Multiplication

An alternative method for performing GF(2) multiplication efficiently on resource-constrained devices is the 4-bit Table-based approach. This method utilizes precomputed values to accelerate carry-less multiplication by decomposing operands into 4-bit segments, significantly reducing the number of required arithmetic operations compared to direct bit-wise computation.

Precomputed Lookup Table The core idea behind this approach is to store the precomputed results of multiplying a fixed operand by all possible 4-bit values (0 through 15). Given an operand H, a table T is created such that:

$$T[i] = H \times i, \quad \forall i \in \{0, 1, 2, \ldots, 15\}$$

where $\times$ represents carry-less multiplication in GF(2). Since each operand is decomposed into 4-bit chunks, the full multiplication can be reconstructed by summing (via XOR) appropriately shifted table entries.

Efficient Computation To multiply a 128-bit operand X with H, X is split into 32 nibbles of 4 bits each: $X = (x_{31}, x_{30}, \ldots, x_0)$. The result is then computed as:

$$Z = \bigoplus_{i=0}^{31} T[x_i] \ll (4 \times i).$$

This approach substantially reduces the number of shift and XOR operations compared to bit-wise polynomial multiplication. Additionally, since all table lookups are independent, this method can be parallelized efficiently on architectures that support SIMD or word-wise operations.

Trade-offs and Performance Considerations While the 4-bit table method provides a significant speed improvement over direct bit-wise multiplication, it requires 256 bytes of storage for the precomputed table. In most embedded environments, including Cortex-M4, this overhead is relatively small and manageable.

4.3 GHASH Timing Attack Vulnerabilities

Because GHASH produces authentication tags, any timing variation dependent on secret data (e.g., the hash key H) may be exploitable. Conditional branches or table lookups that vary with the key can introduce observable timing patterns (e.g., memory access delays, branch mispredictions).

Although many Cortex-M4 systems have limited or no cache, making classical L1/L2 cache attacks less plausible, subtle timing differences in branch logic or flash-memory access may still be exploited. Consequently, libraries like BearSSL [19] strive for constant-time or near-constant-time GHASH implementations by avoiding data-dependent branches or, at minimum, masking accesses during table lookups.

4.4 Karatsuba-Based Multiplication

This method divides the 128-bit multiplication into several smaller multiplications and combines them through XOR. We adapted a constant-time implementation from BearSSL which avoids large lookup tables [20]. This process involves two major steps: first, preparing a carry-less multiplication routine for 32-bit integers, and second, using Karatsuba decomposition to handle the full 128-bit product efficiently. Below, we briefly introduce the 32-bit Karatsuba approach used by BearSSL and describe how it operates on the ARM Cortex-M4.

BearSSL adopts a 32-bit Karatsuba algorithm for GF(2^{128}) multiplication (`GHASH`), comprising two principal steps: `(i)` constructing a carry-less multiplication routine for 32-bit integers, and `(ii)` applying Karatsuba decomposition to achieve a full 128-bit product. The following discussion briefly introduces BearSSL's 32-bit Karatsuba approach and examines its operation on the ARM Cortex-M4.

32-bit Carry-Less Multiplication
In BearSSL's design, each 32-bit operand is subdivided so that ordinary integer multiplications yield carries only into "gaps," which are subsequently masked out to produce a pure carry-less result in GF(2). Concretely, by leveraging masks such as `0x11111111`, `0x22222222`, `0x44444444`, and `0x88888888`, the implementation isolates specific bit subsets, multiplies them separately as normal 32×32 integers, and merges the partial products through XOR. This procedure remains constant time by following a fixed sequence of bit-wise and arithmetic operations, avoiding any data-dependent branching.

Karatsuba for 128-Bit Operands
After establishing the 32-bit routine, BearSSL applies the Karatsuba algorithm to handle GF(2^{128}) multiplication. In particular, each 128-bit operand is divided into two 64-bit segments, $(A_{\text{hi}} \,\|\, A_{\text{lo}})$ and $(B_{\text{hi}} \,\|\, B_{\text{lo}})$. The method computes three partial products:

$$P = A_{\text{hi}} \otimes B_{\text{hi}}, \quad Q = A_{\text{lo}} \otimes B_{\text{lo}}, \quad R = (A_{\text{hi}} \oplus A_{\text{lo}}) \otimes (B_{\text{hi}} \oplus B_{\text{lo}}),$$

and combines them as:

$$C = (P \ll 128) \ \oplus \ Q \ \oplus \ \big((R \oplus P \oplus Q) \ll 64\big).$$

Here, $\otimes$ denotes carry-less multiplication in GF(2), while $\oplus$ is XOR. Each 64-bit multiplication (P, Q, and R) invokes the 32-bit routine multiple times, resulting in nine calls rather than the sixteen a naive schoolbook approach would require, thereby reducing the total count of partial multiplications.

On ARM Cortex-M4 and similar embedded architectures, the 4-bit table technique is a practical middle ground of speed, memory (256 bytes), and moderate design complexity. In scenarios with extreme memory constraints or heightened demand for timing-attack resistance, Karatsuba remains an appealing alternative—provided one can manage its more intricate implementation. We present benchmark results for both GHASH options in combination with our Fixslicing + FACE AES-CTR in Section 5, demonstrating AES-GCM performance on the ARM Cortex-M4 platform.

5 Evaluations

We conducted our experiments on an STM32F407G-DISC1 board featuring an ARM Cortex-M4 core. The source code are available at: [upon the completion of the review process]. For AES-128 and AES-256, we employed an optimized Fixslicing AES assembly code from prior work, and to optimize CTR mode, we implemented three FACE variants (i.e., FACE_{rd1}, $\text{FACE}_{\text{rd1+}}$, and $\text{FACE}_{\text{rd2+}}$) in assembly. For the GHASH computation, we compared two implementations: Karatsuba, which is a modified constant-time Karatsuba-based code derived from BearSSL, and Table-based, which uses a 4-bit lookup table. We measured cycle counts by sampling the Cortex-M4's DWT (Data Watchpoint and Trace) counter before and after each cryptographic routine, taking the average of 100 runs for messages of various sizes under the same key and nonce. These cycle counts include both round key generation and the precomputation overhead of FACE. We tested message sizes from 1 KB to 40 KB to observe performance trends. In all tables, 'Basic' refers to our Fixslicing AES implementation without any of the FACE optimizations.

Below, we present a comprehensive analysis of how the chosen CTR-mode optimizations (FACE) and GHASH implementations influence the overall speed and resource usage of AES-GCM.

5.1 Analysis of GCTR Measurement Results

We evaluated the cycle counts for AES-128 and AES-256 GCTR (counter mode encryption) on message sizes ranging from 1 KB to 40 KB, comparing our Basic GCTR implementation to three FACE variants: $\texttt{FACE}_{\texttt{rd1}}$, $\texttt{FACE}_{\texttt{rd1+}}$, and $\texttt{FACE}_{\texttt{rd2+}}$. The measurement results are summarized in Table 3. It is particularly noteworthy that small messages (1 KB) yield only limited benefits from FACE optimizations. For both AES-128 and AES-256, certain FACE variants even introduce

slight overheads at 1 KB, largely due to the initial precomputation required for partial round-key caching or lookup-table setup. These overheads are not fully amortized when relatively few blocks are processed.

However, significant performance improvements emerge as soon as the message size grows beyond 4 KB. In particular, $\texttt{FACE}_{\texttt{rd2+}}$ consistently delivers the largest cycle reduction, achieving savings of up to 19.4% for AES-128 and 14.6% for AES-256 at the 40 KB scale. Despite AES-256 having more rounds—and hence a higher baseline cycle count—FACE-based caching also provides meaningful relative gains for AES-256, reducing cycles by over 14% at 20 KB and 40 KB. From a practical standpoint, these results indicate that precomputation and caching strategies are most beneficial when a single key and nonce are used to encrypt many blocks.In strictly resource-constrained environments, $\texttt{FACE}_{\texttt{rd1}}$ may be appealing due to its smaller memory footprint, despite lower speedups compared to $\texttt{FACE}_{\texttt{rd2+}}$. Meanwhile, devices with sufficient RAM/flash headroom may prefer $\texttt{FACE}_{\texttt{rd2+}}$, which substantially reduces encryption latency at scale. Although AES-256 inherently demands more cycles than AES-128, FACE optimizations scale effectively to the extended round structure. Systems requiring 256-bit security can still gain considerable performance improvements—albeit at slightly smaller percentage reductions.

Table 3: Cycle counts (and in parentheses: percentage improvement over Basic) for AES-GCTR with various FACE variants (AES-128 and AES-256), from 1 KB to 40 KB messages.

FACE Variant	1 KB		4 KB		20 KB		40 KB	
	128	**256**	**128**	**256**	**128**	**256**	**128**	**256**
Basic	117,456 (0%)	158,453 (0%)	469,488 (0%)	633,461 (0%)	2,345,712 (0%)	3,166,837 (0%)	4,691,312 (0%)	6,333,557 (0%)
$\texttt{FACE}_{\texttt{rd1}}$	118,666 (-1.03%)	164,420 (-3.77%)	431,971 (+8.00%)	599,738 (+5.32%)	2,102,571 (+10.36%)	2,920,616 (+7.78%)	4,190,821 (+10.67%)	5,821,796 (+8.08%)
$\texttt{FACE}_{\texttt{rd1+}}$	118,259 (-0.68%)	164,366 (-3.73%)	411,574 (+12.34%)	580,511 (+8.36%)	1,980,866 (+15.55%)	2,799,003 (+11.62%)	3,941,041 (+16.00%)	5,572,118 (+12.03%)
$\texttt{FACE}_{\texttt{rd2+}}$	119,107 (-1.41%)	164,763 (-3.98%)	400,915 (+14.60%)	568,391 (+10.28%)	1,903,287 (+18.87%)	2,718,579 (+14.15%)	3,781,252 (+19.41%)	5,406,764 (+14.63%)

5.2 Analysis of GHASH Measurement Results

We compared two GHASH implementations, based on Karatsuba and based on tables, over message sizes ranging from 1 KB to 40 KB, as summarized in Table 4. Our measurements indicate that the Table-based approach consistently runs at roughly twice the speed of the Karatsuba variant across all tested message lengths. For instance, at 1 KB, the Table-based method requires 79,402 cycles, which is nearly half of Karatsuba's 158,749 cycles, and this performance gap remains similar at larger sizes (e.g., 2,978,281 vs. 6,041,821 cycles at 40 KB).

Both implementations scale linearly with message size, reflecting the fixed-block nature of GHASH. As a result, the absolute difference in cycle counts grows in proportion to the input length—reaching nearly three million additional cycles for Karatsuba at 40 KB. Although the Table-based variant offers clear throughput advantages, it may also demand a larger memory footprint and pose cache-based side-channel risks, especially if table lookups are not performed in constant time. By contrast, the Karatsuba-based version relies on a divide-and-conquer polynomial multiplication strategy that is inherently constant-time, potentially making it better suited for highly constrained or security-critical environments, despite its higher overall cycle count.

Table 4: Cycle counts (with improvements over Karatsuba in parentheses) for GHASH, tested on 1 KB to 40 KB messages.

GHASH	**1 KB**	**4 KB**	**20 KB**	**40 KB**
Karatsuba	158,749 (0%)	611,293 (0%)	3,024,861 (0%)	6,041,821 (0%)
Table-based	79,402 (+50.0%)	302,390 (+50.5%)	1,491,670 (+50.7%)	2,978,281 (+50.7%)

5.3 Analysis of AES-GCM Mode Measurement Results

We next examined the performance of AES-GCM encryption by using each of the four FACE variants with Table-based or Karatsuba GHASH. Table 5 summarizes the cycle counts for AES-GCM in messages 1 /,KB, 4 /,KB, 20 /,KB and 40 /,KB under both AES-128 and AES-256. It is particularly noteworthy that, while FACE primarily optimizes AES-CTR, its gains are partially offset once GHASH computation is included. Because GHASH can occupy a substantial portion of the total runtime, the net speedup from FACE over the Basic GCM typically remains in the 5–10% range—less than the larger improvements observed in GCTR alone. Nevertheless, Table-based GHASH (generally faster) underscores the effect of FACE more clearly, offering cycle reductions of up to 12–13% for AES-128 GCM at 40/,KB and around 10% for AES-256. By contrast, the relatively slower Karatsuba GHASH diminishes the net impact of FACE, although FACE-based implementations still provide measurable gains (e.g., 5–8%) when the message size is sufficiently large. As with the GCTR measurements, message size strongly influences the benefits of FACE. Larger messages ($\geq 4/, KB$) amortize the precomputation and caching overheads more effectively, resulting in higher net speedups. At 40/,KB, for instance, $\texttt{FACE}_{\text{rd2+}}$ can reduce cycle counts by roughly 12% under Table-based GHASH for AES-128, and around 10% for

AES-256. Conversely, 1/,KB inputs may yield less pronounced gains—or even minor slowdowns—since initialization overhead can dominate at smaller scales.

Table 5: Cycle counts (with percentages relative to the Basic approach) for AES-GCM (1 KB to 40 KB) under different GHASH implementations and FACE variants, , from 1 KB to 40 KB messages.

GHASH Technique	FACE Variant	Input Size (bytes)							
		1 KB		4 KB		20 KB		40 KB	
		128	256	128	256	128	256	128	256
Table-based	Basic	206,377 (0%)	250,657 (0%)	780,148 (0%)	946,962 (0%)	3,929,031 (0%)	4,659,716 (0%)	7,664,348 (0%)	9,301,295 (0%)
	FACE_{rd1}	207,123 -0.36%	257,089 -2.57%	740,507 (+5.07%)	915,445 (+3.32%)	3,592,621 (+8.56%)	4,425,658 (+5.02%)	7,156,129 (+6.62%)	8,813,752 (+5.23%)
	$\text{FACE}_{\text{rd1+}}$	206,894 -0.25%	256,506 -2.33%	721,076 (+7.56%)	893,638 (+5.63%)	3,470,230 (+11.69%)	4,291,684 (+7.88%)	6,904,704 (+9.90%)	8,539,152 (+8.20%)
	$\text{FACE}_{\text{rd2+}}$	207,315 -0.45%	256,511 -2.33%	708,758 (+9.16%)	880,339 (+7.03%)	3,382,451 (+13.92%)	4,206,896 (+9.72%)	6,724,602 (+12.26%)	8,365,109 (+10.05%)
Karatsuba	Basic	286,425 (0%)	330,735 (0%)	1,091,001 (0%)	1,257,903 (0%)	5,381,427 (0%)	6,201,513 (0%)	10,744,627 (0%)	12,381,353 (0%)
	FACE_{rd1}	287,956 -0.53%	337,339 -1.99%	1,054,620 (+3.33%)	1,226,429 (+2.50%)	5,143,006 (+4.42%)	5,967,503 (+3.77%)	10,253,496 (+4.57%)	11,893,853 (+3.94%)
	$\text{FACE}_{\text{rd1+}}$	287,320 -0.31%	336,580 -1.76%	1,033,309 (+5.28%)	1,204,553 (+4.24%)	5,011,815 (+6.87%)	5,833,415 (+5.93%)	9,984,955 (+7.06%)	11,619,500 (+6.16%)
	$\text{FACE}_{\text{rd2+}}$	288,077 -0.58%	337,182 -1.95%	1,022,429 (+6.28%)	1,193,035 (+5.16%)	4,938,363 (+8.24%)	5,756,741 (+7.16%)	9,833,288 (+8.49%)	11,461,381 (+7.45%)

6 Conclusion

In this paper, we proposed methods to optimize both the AES-CTR mode and the GHASH phase in order to efficiently and securely implement AES-GCM in an ARM Cortex-M4 environment. First, for the AES-CTR component, we removed or replaced Table-based S-box lookups by employing Fixslicing AES, and integrated the caching concept from the FACE (Fast AES-CTR Encryption) scheme to minimize redundant operations when processing large messages. Experiments on an ARM Cortex-M4 showed that our approach reduces the cycle count by up to 19.41% for AES-128 GCTR and 14.63% for AES-256 GCTR, offering a significant performance improvement over approaches that rely solely on Fixslicing.

Meanwhile, for GHASH operations, we compared a 4-bit Table-based multiplication method with a Karatsuba-based constant-time approach. While the 4-bit Table-based method can be nearly twice as fast as Karatsuba, it cannot guarantee fully constant-time execution due to its reliance on cache-based memory accesses. Hence, although the risk of cache-timing attacks may be lower in MCU environments with limited or no caches, certain security requirements or attack models might still favor table-free implementations like Karatsuba.

Based on these findings, future work could explore applying the proposed optimization techniques to a broader range of embedded processors and cryptographic algorithms beyond the ARM Cortex-M4. For example, one could leverage instructions from ARMv8-M or RISC-V, or extend these methods. Additionally, side-channel countermeasures such as masking [21] could be integrated into our design to further enhance security against power [22] or electromagnetic analysis [15].

References

1. Adomnicai, A., Najm, Z., Peyrin, T.: Fixslicing: A new gift representation. Cryptology ePrint Archive (2020)
2. Adomnicai, A., Peyrin, T.: Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 402–425 (2021)
3. Arm Limited: ARMv7-M Architecture Reference Manual. Tech. Rep. ARM DDI 0403E.b, Arm Ltd., Cambridge, UK (2014), https://developer.arm.com/documentation/ddi0403/latest
4. Bernstein, D.J.: Cache-timing attacks on aes (2005)
5. Biham, E.: A fast new des implementation in software. In: Fast Software Encryption: 4th International Workshop, FSE'97 Haifa, Israel, January 20–22 1997 Proceedings 4. pp. 260–272. Springer (1997)
6. Bonneau, J., Mironov, I.: Cache-collision timing attacks against aes. In: Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings. pp. 201–215. Springer (2006)
7. Gouvêa, C.P., López, J.: Implementing gcm on armv8. In: Topics in Cryptology—CT-RSA 2015: The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015, Proceedings. pp. 167–180. Springer (2015)
8. Gueron, S., Kounavis, M.E.: Intel® carry-less multiplication instruction and its usage for computing the gcm mode. White Paper p. 10 (2010)
9. Hajihassani, O., Monfared, S.K., Khasteh, S.H., Gorgin, S.: Fast aes implementation: A high-throughput bitsliced approach. IEEE Transactions on parallel and distributed systems **30**(10), 2211–2222 (2019)
10. Jankowski, K., Laurent, P.: Packed aes-gcm algorithm suitable for aes/pclmulqdq instructions. IEEE transactions on computers **60**(1), 135–138 (2010)
11. Käsper, E., Schwabe, P.: Faster and timing-attack resistant aes-gcm. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 1–17. Springer (2009)
12. Kim, H., Eum, S., Lee, W.K., Lee, S., Seo, H.: Secure and robust internet of things with high-speed implementation of present and gift block ciphers on gpu. Applied Sciences **12**(20), 10192 (2022)
13. Kim, H., Eum, S., Sim, M., Seo, H.: Efficient implementation of speedy block cipher on cortex-m3 and risc-v microcontrollers. Mathematics **10**(22), 4236 (2022)
14. Kim, K., Choi, S., Kwon, H., Kim, H., Liu, Z., Seo, H.: Page—practical aes-gcm encryption for low-end microcontrollers. Applied Sciences **10**(9), 3131 (2020)
15. Longo, J., De Mulder, E., Page, D., Tunstall, M.: Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. In: Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17. pp. 620–640. Springer (2015)

16. McGrew, D., Viega, J.: The galois/counter mode of operation (gcm). submission to NIST Modes of Operation Process **20**, 0278–0070 (2004)
17. Nishikawa, N., Amano, H., Iwai, K.: Implementation of bitsliced aes encryption on cuda-enabled gpu. In: Network and System Security: 11th International Conference, NSS 2017, Helsinki, Finland, August 21–23, 2017, Proceedings 11. pp. 273–287. Springer (2017)
18. Park, J.H., Lee, D.H.: Face: Fast aes ctr mode encryption techniques based on the reuse of repetitive data. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 469–499 (2018)
19. Pornin, T.: Bearssl. https://bearssl.org, accessed: 2025-03-16
20. Pornin, T.: BearSSL: GHASH constant-time multiplication implementation ($ghash\_ctmul.c$) (2016), https://bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/hash/$ghash_ctmul.c$, accessed: 2025-03-16
21. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 142–159. Springer (2013)
22. Randolph, M., Diehl, W.: Power side-channel attack analysis: A review of 20 years of study for the layman. Cryptography **4**(2), 15 (2020)
23. Rebeiro, C., Selvakumar, D., Devi, A.: Bitslice implementation of aes. In: International Conference on Cryptology and Network Security. pp. 203–212. Springer (2006)
24. Schwabe, P., Stoffelen, K.: All the aes you need on cortex-m3 and m4. In: International Conference on Selected Areas in Cryptography. pp. 180–194. Springer (2016)
25. Selent, D.: Advanced encryption standard. Rivier Academic Journal **6**(2), 1–14 (2010)
26. Sovyn, Y., Khoma, V., Podpora, M.: Comparison of three cpu-core families for iot applications in terms of security and performance of AES-GCM. IEEE Internet of Things Journal **7**(1), 339–348 (2020). https://doi.org/10.1109/JIOT.2019.2953230