Fast implementation of $\mathrm{GF}(2^8)$ inversion with applications to the AES S-box using ARM Helium MVE extensions

Eric Azevedo de Oliveira[1], **Décio Luiz Gazzoni Filho**[1,2], **Felix Carvalho Rodrigues**[1], **Julio López**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

[2]Departamento de Engenharia Elétrica – Universidade Estadual de Londrina

e291073@dac.unicamp.br,
{decio.gazzoni,felix.rodrigues,jlopez}@ic.unicamp.br

Abstract. *The AES block cipher is widely used in embedded systems. Efficient realizations of its S-box, which consists of a costly inversion in* $\mathrm{GF}(2^8)$ *and an affine transformation, are usually done with lookup tables, which may be vulnerable to timing side-channel attacks. We develop a constant-time inversion algorithm in* $\mathrm{GF}(2^8)$ *optimized with Arm Helium MVE extensions and a fully vectorized suite of field-arithmetic primitives, from which we derive three Cortex-M S-box implementations: two table-free exponentiation schedules requiring only three field multiplications, and a third variant employing a compact, vectorized 16-byte lookup table. We integrate these methods into a constant-time AES-128 implementation on a Cortex-M85, demonstrating that our proposal is competitive with state-of-the-art implementations.*

1. Introduction

The Advanced Encryption Standard (AES), a block cipher standardized by NIST, is widely used across a broad range of applications, including those involving embedded, resource-constrained devices. Its design heavily relies on operations in the finite field $\mathrm{GF}(2^8)$, and in particular its S-box is defined in terms of a costly inversion operation in this field, which can be implemented using squarings and multiplications. Thus, secure implementation in embedded systems can be challenging.

To address these challenges, we leverage Helium, the M-Profile Vector Extension (MVE) available on recent ARM Cortex-M cores, to specifically improve the performance of the inversion computation in $\mathrm{GF}(2^8)$, a fundamental operation of the AES S-Box. Helium introduces 128-bit SIMD vector instructions and provides native support for polynomial multiplication over $\mathrm{GF}(2^8)$, an operation not previously available in the ARMv7-M architecture, enabling efficient implementation of binary field arithmetic.

In this work, we present efficient algorithms for the finite field operations in $\mathrm{GF}(2^8)$, including a new algorithm for inversion in $\mathrm{GF}(2^8)$. We also provide an efficient constant-time implementation of AES using Helium's SIMD capabilities on a Cortex M-85. Our experimental results demonstrate the effectiveness of the proposed algorithms and emphasize how hardware-aware algorithmic optimizations can enhance cryptographic efficiency in resource-constrained environments.

2. Preliminaries

The following section introduces key concepts underlying the computational nature of modern embedded systems, beginning with an overview of the M-Profile Vector Ex-

tension (MVE) and Arm Helium Technology, which enhance processing performance through SIMD operations. The discussion then shifts to the Advanced Encryption Standard (AES), a widely used symmetric encryption algorithm.

2.1. M-Profile Vector Extension (MVE) and Helium Technology

The M-Profile Vector Extension (MVE) is a feature of the Armv8.1-M architecture that aims to improve the computational performance of Cortex-M processors in embedded systems. This extension integrates Single Instruction Multiple Data (SIMD) capabilities, thus allowing multiple elements to be processed in the same cycle. Taking advantage of the 128-bit vector registers, MVE increases the processing rate of instructions, thus reducing execution time.

Arm Helium technology [Marsh 2020] is the specific implementation of MVE in Cortex-M processors. It provides a extensive set of SIMD vector instructions capable of operating on various data types, including signed and unsigned 8-, 16-, 32- and 64-bit integers, as well as floating-point numbers. Particularly relevant for our work are the `VMULLB.P8` and `VMULLT.P8` instructions, which perform carry-less multiplication on two vectors of sixteen 8-bit polynomials, returning the lower and upper halves of the 128-bit interleaved polynomial products. These specialized instructions are effective for computationally intensive tasks.

One of Helium's advantages is its memory-handling mechanisms, such as scatter-gather instructions, which allow rapid access to non-contiguous memory locations. Additionally, the Helium instruction set is designed to allow pipelined execution of its instructions, although some restrictions apply: for example, in the Cortex-M85, two `VMULL` operations cannot be pipelined.

2.2. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm, specified by the U.S. National Institute of Standards and Technology (NIST) for encryption of digital data, as described in their official document [National Institute of Standards and Technology 2001]. It supports three key lengths: 128 bits, 192 bits, and 256 bits. The number of rounds varies with key length: 10, 12, and 14 rounds for AES-128, AES-192, and AES-256, respectively. The key size is directly related to the security level provided by each variant.

The AES algorithm represents 128-bit blocks as a 4×4 matrix of bytes, called the state, taken as elements of $\mathrm{GF}(2^8)$ modulo the irreducible polynomial $x^8+x^4+x^3+x+1$. It applies the following four transformations in each round:

- `AddRoundKey`: In this step, the round-specific sub-key derived from the AES key schedule is combined with the state matrix using an XOR operation. This operation introduces the key material directly into the encryption process.
- `SubBytes`: Each byte of the state is substituted by another byte according to the non-linear transformation known as the S-box. The function S-box(a) is defined by the multiplicative inversion of $a \in \mathrm{GF}(2^8)$ followed by an affine transformation. This is the sole non-linear operation of the cipher.

- `MixColumns`: Each column of the state matrix undergoes a linear transformation, represented by multiplication with a fixed polynomial matrix over the finite field $\mathrm{GF}(2^8)$. This transformation provides vertical diffusion, ensuring that changes in individual bytes affect the entire column.
- `ShiftRows`: In this layer, the last three rows of the state are cyclically shifted to the left: the second row is rotated by one byte, the third row by two bytes, and the fourth row by three bytes. The first row remains unchanged.

The round keys are generated using a key expansion algorithm, which transforms its master key into distinct round keys for each encryption round.

Several works explore AES implementations on ARM devices, from high end ARMv8 and ARMv7 based approaches [Babu et al. 2011, Fujii et al. 2019] to embedded devices such as Cortex-M4 and others [Adomnicai and Peyrin 2020, Schwabe and Stoffelen 2017, Kim and Seo 2025], however there is still a lack of works focusing on Helium enabled devices.

2.3. Implementation Challenges

Implementing AES securely on constrained devices poses notable challenges. In particular, most table-based optimizations target the computation of the S-box, although tables can also accelerate operations such as `xtime` ($x \cdot a$ for a in $GF(2^8)$). Schwabe and Stoffelen [Schwabe and Stoffelen 2017] report that AES-128 in CTR mode requires 554.4 cycles per block on the Cortex-M4 cores when using table-based unprotected implementations. In contrast, their constant-time implementations increase the cost to 1617.6 cycles per block. This trade-off motivates the development of efficient constant-time routines for operations such as multiplicative inversion in $\mathrm{GF}(2^8)$.

One of the fastest methods for computing multiplicative inverses in $\mathrm{GF}(2^8)$ was proposed by Rivain and Prouff [Rivain and Prouff 2010], using the formula:

$$a^{-1} = a^{254} = a^{14} \cdot (a^{15})^{16}.$$

This method requires four field multiplications and seven squarings at minimum, as shown in [Rivain and Prouff 2010, Section 3.1]. Other works revisit this exponentiation-based strategy: [Itoh and Tsujii 1988, Gentry et al. 2012, Dimitrov and Järvinen 2013, Fan 2020, Morita et al. 2024]. All these methods realise the inverse with four general multiplications and six or seven squarings. A different line avoids field arithmetic altogether: Fixslicing by [Adomnicai and Peyrin 2020] embeds the Boyar–Peralta Boolean circuit directly into a bit-sliced AES, replacing the inverse by gate-level logic rather than reducing the multiplication count.

This work presents an improved method for computing a^{-1} in $\mathrm{GF}(2^8)$, requiring only three multiplications and seven squarings. We also provide an optimized implementation for the Cortex-M85, leveraging 128-bit vector instructions such as `VMULLB.P8` and `VMULLT.P8` to accelerate polynomial arithmetic.

3. $\mathrm{GF}(2^8)$ Arithmetic

This section introduces our new constant-time routines for multiplication, squaring, and multi-squaring in $\mathrm{GF}(2^8)$. These operations are themselves a key contribution and will later serve as the core of the inverse algorithms presented in Section 4.

Algorithm 1. $a \cdot b$ in $\mathrm{GF}(2^8)$ (also computes a^2 when $a = b$)

Require: $a, b \in \mathrm{GF}(2^8)$ ▷ a, b: 128 bits
Ensure: $a \cdot b$
1: $b_0 \leftarrow \texttt{VMULLB.P8}(a, b)$
2: $t_0 \leftarrow \texttt{VMULLT.P8}(a, b)$
3: $b_1 \leftarrow \texttt{VMULLT.P8}(t_0, 0\text{x}1\text{b})$
4: $t_1 \leftarrow \texttt{VMULLT.P8}(b_0, 0\text{x}1\text{b})$
5: $b_2 \leftarrow \texttt{VMULLT.P8}(t_1, 0\text{x}1\text{b})$
6: $t_2 \leftarrow \texttt{VMULLT.P8}(b_1, 0\text{x}1\text{b})$
7: $b_0 \leftarrow b_0 \oplus b_1$
8: $t_0 \leftarrow t_0 \oplus t_1$
9: $b_0 \leftarrow b_0 \oplus b_2$
10: $t_0 \leftarrow t_0 \oplus t_2$
11: $r_0 \leftarrow \texttt{VSLI.16}(b_0, t_0, 8)$
12: **return** r_0

Algorithm 2. Affine transformation

Require: $a \in \mathrm{GF}(2^8)$ ▷ a: 128 bits
Ensure: AffineTransform(a)
1: $l \leftarrow \texttt{VMULLB.P8}(a, 0\text{x}1\text{f})$
2: $h \leftarrow \texttt{VMULLT.P8}(a, 0\text{x}1\text{f})$
3: $s_1 \leftarrow \texttt{VSRI.16}(h, l, 8)$
4: $s_2 \leftarrow \texttt{VSLI.16}(l, h, 8)$
5: $s_2 \leftarrow s_2 \oplus s_1$
6: $r_0 \leftarrow s_2 \oplus 0\text{x}63$ ▷ 0x63 . . . 0x63 (16 bytes)
7: **return** r_0

Algorithm 3. a^{16} in $\mathrm{GF}(2^8)$

Require: $a \in \mathrm{GF}(2^8)$ ▷ a: 128 bits
Ensure: a^{16}
1: $r_1 \leftarrow a \ll 6$
2: $r_2 \leftarrow a \ll 5$
3: $r_3 \leftarrow a \ll 4$
4: $r_5 \leftarrow a \ll 2$
5: $m_2 \leftarrow r_5 \oplus a$
6: $t_1 \leftarrow m_2 \oplus r_2$
7: $m_0 \leftarrow t_1 \oplus r_3$
8: $m_0 \leftarrow m_0 \gg 7$
9: $m_2 \leftarrow m_2 \gg 7$
10: $m_1 \leftarrow (t_1 \ll 1) \oplus a$
11: $m_1 \leftarrow m_1 \gg 7$
12: $r_0 \leftarrow r_1 \gg 7$
13: $r_0 \leftarrow r_0 \oplus (m_0 \cdot 0\text{xe}0) \oplus (m_1 \cdot 0\text{x}5\text{d}) \oplus (m_2 \cdot 0\text{x}51)$
14: $r_0 \leftarrow r_0 \oplus a$
15: **return** r_0

Algorithm 4. MixColumns over $\mathrm{GF}(2^8)$

Require: $s = a|b|c|d\,; a, b, c, d$: (32 bit each)
Ensure: $r_1 = \textsc{MixColumns}(s)$
1: $r_0 \leftarrow \texttt{VREV16}(s)$ ▷ $r_0 = b|a|d|c$
2: $r_1 \leftarrow \texttt{VREV32}(r_0)$ ▷ $r_1 = c|d|a|b$
3: $r_2 \leftarrow \texttt{VREV16}(r_1)$ ▷ $r_2 = d|c|b|a$
4: $r_3 \leftarrow \texttt{VPSEL}(r_2, r_0, 0\text{xaaaa})$ ▷ $r_3 = b|c|d|a$
5: $r_3 \leftarrow r_3 \oplus s$
6: $r_0 \leftarrow r_0 \oplus r_1$
7: $r_0 \leftarrow r_0 \oplus r_2$
8: $r_1 \leftarrow (r_3 \ll 1) \oplus ((r_3 \gg 7) \cdot 0\text{x}1\text{b})$
9: $r_0 \leftarrow r_1 \oplus r_0$
10: **return** r_0

Algorithm 1 implements multiplication (and squaring) in $\mathrm{GF}(2^8)$, exploiting Arm Helium vector instructions `VMULLB.P8` and `VMULLT.P8`. Given inputs a and b, the algorithm first computes the lower and upper halves of the polynomial product using the carry-less multiplication instructions (cf. Appendix A). Subsequently, modulo reduction by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ (represented by $0\text{x}1\text{b}$) is performed through iterative multiplications of these upper halve by the modulo $0x1b$ using the `VMULLT.P8` instruction. Intermediate results are combined using XOR operations, and the final polynomial result is obtained by merging these intermediate values using the `VSLI.16` instruction.

Algorithm 2 implements the affine transformation in $\mathrm{GF}(2^8)$, taking an input a, and computes $M \cdot a \oplus b$, where M is a circulant matrix whose first row is $0\text{x}1\text{f}$ and b is the constant polynomial $0\text{x}63$; as shown in [Aguilar et al. 2016], this operation can be written as $T(x) = a \cdot 0\text{x}1\text{f} \oplus 0\text{x}63 \bmod (x^8 + 1)$, and can be implemented using the specialized instructions `VMULLB.P8`, `VMULLT.P8`, and `VSLI.16`, whose design is illustrated in Appendix A.

All algorithms are branch-free and therefore execute in constant time. Besides the base multiplication/squaring routine, we use Algorithm 3 for four consecutive squarings (a^{16}) and Algorithm 4 for the `MixColumns` step. The latter remains table-free: byte

shuffles `VREV16` and `VREV32` create the required rotations, `VPSEL` (with mask 0xaaaa) assembles the rotation $b|c|d|a$, and the final multiplication by x is a left shift combined with a conditional XOR with 0x1b.

4. Novel Techniques for Multiplicative Inverses

We present an algorithm for computing the multiplicative inverse of an element $a \in GF(2^8)$, which is presented in Algorithm 5 and and it is defined by the following:

$$a^{-1} = \begin{cases} r, & \text{if } r \neq 0, \\ a^{84}, & \text{if } r = 0. \end{cases} \quad \text{where } r = a \cdot (a^{84})^2 \oplus a^{84}, \tag{1}$$

Its correctness follows from properties of $\mathrm{GF}(2^8)$. Let $a \in \mathrm{GF}(2^8)$, with $a \neq 0$. Since $a^{255} = (a^{85})^3 = 1$, it follows that a^{85} is an element of order either 1 or 3 in $GF2^8$; thus, $a^{85} \in \{0\text{x01}, 0\text{xbc}, 0\text{xbd}\}$. If $a^{85} = 0\text{x01}$, it follows directly that $a^{-1} = a^{84}$. Otherwise, since $a^{85} \in \{0\text{xbc}, 0\text{xbd}\}$, $0\text{xbc} \cdot 0\text{xbd} = 1$ and $0\text{xbc} = 0\text{xbd} \oplus 1$, we have $(a^{85})^{-1} = a^{85} \oplus 1$. Since $a^{-1} = a^{84} \cdot (a^{85})^{-1}$, we obtain: $a^{-1} = a^{84}(a^{85} \oplus 1) = a \cdot (a^{84})^2 \oplus a^{84}$. Let $b = a^{17}$; then $a^{84} = a^{16} \cdot b^4$ and $r = a^{16} \cdot (b^4 \oplus b \cdot b^8)$. Therefore, a^{-1} can be computed as:

$$a^{-1} = a^{16} \cdot \begin{cases} b^4, & r = 0, \\ (b^4 \oplus b \cdot b^8), & r \neq 0. \end{cases} \tag{2}$$

The inversion of a can also be computed in terms of the inversion of b as $a^{-1} = a^{16} \cdot b^{-1}$. Hence, the new method requires three finite field multiplications, one multi-squaring of a and two multi-squarings of b.

Optimized computation of b^4 and b^8. Since $b^{15} = a^{17 \cdot 15} = a^{255} = 1$, the representation of b in $GF((2^4)^2)$, using the isomorphism between $GF(2^8)$ and $GF((2^4)^2)$ given in [Morita et al. 2024], is (b_H, b_L), where b_H is zero. Hence, b_L can be computed using only the bits b_0, b_3, b_4, b_6 of b, as $b_L = (b_3 \oplus b_4, b_4 \oplus b_6, b_3, b_0)$. After computing b_L^4 and b_L^8 in $GF(2^4)$, one can use the isomorphism to get the corresponding results in $GF(2^8)$. Therefore, $b^4 = b_3 \cdot 0\text{xb0} \oplus b_6 \cdot 0\text{xe1} \oplus b_4 \cdot 0\text{x0c} \oplus b_0$, and $b^8 = b_3 \cdot 0\text{xed} \oplus b_6 \cdot 0\text{x5c} \oplus b_4 \cdot 0\text{x50} \oplus b_0$, each one requiring three constant multiplications and three XORs. The final value $r = b^4 \oplus b \cdot b^8$ is obtained with a single multiplication instruction, after which `VPSELQ_u8` selects $a^{16} \cdot b^4$ or $a^{16} \cdot (b^4 \oplus b \cdot b^8)$ based on the predicate $b^4 \oplus b \cdot b^8 = 0$, completing the inversion in constant time with only three finite-field multiplications, as shown in Algorithm 6.

Another way to compute the inverse of b is to build a 4-bit index via $\mathrm{idx} = b_6 \cdot 2^3 \oplus b_4 \cdot 2^2 \oplus b_3 \cdot 2^1 \oplus b_0 \cdot 2^0$, and then retrieve b^{-1} from the 16-byte lookup table $\mathrm{LUT}_{\mathrm{inv}} = \{0\text{x00}, 0\text{x01}, 0\text{x0c}, 0\text{xe0}, 0\text{xed}, 0\text{x5c}, 0\text{xb1}, 0\text{x0d}, 0\text{xb0}, 0\text{xe1}, 0\text{xbd}, 0\text{xbc}, 0\text{x51}, 0\text{xec}, 0\text{x5d}, 0\text{x50}\}$. This approach uses only three vector bitwise operations and one gather-load, occupying exactly 16 bytes of memory (see Algorithm 7); the gather-load instruction remains timing-invariant [Arm Ltd. 2023].

5. Results

The execution-cycle counts were obtained on a Renesas EK-RA8M1 evaluation board (Cortex-M85 core) using `gcc` version 13.3.1 (target `arm-none-eabi`) with the `O3`

Algorithm 5. a^{-1} using Eq. (1)

Require: $a \in GF(2^8)^{16}$
Ensure: a^{-1}
1: $r_0 \leftarrow a^{16}$
2: $r_1 \leftarrow a \cdot r_0$
3: $r_2 \leftarrow r_0 \cdot r_1^4$
4: $r_3 \leftarrow r_2^2 \oplus r_2$
5: **if** $r_3 \neq 0$ **then return** r_3, **else return** r_2

Algorithm 6. a^{-1} using Eq. (2)

Require: $a \in GF(2^8)^{16}$
Ensure: a^{-1}
1: $r_0 \leftarrow a^{16}$
2: $b \leftarrow r_0 \cdot a$
3: $mask \leftarrow$ `0x01` ▷ 0x01 . . . 0x01 (16 bytes)
4: $b_0 \leftarrow b \wedge mask$
5: $b_3 \leftarrow (b \gg 3) \wedge mask$
6: $b_4 \leftarrow (b \gg 4) \wedge mask$
7: $b_6 \leftarrow (b \gg 6) \wedge mask$
8: $r_0 \leftarrow (b_3 * 0\text{xb0}) \oplus (b_6 * 0\text{xe1}) \oplus (b_4 * 0\text{x0c}) \oplus b_0$
9: $r_1 \leftarrow (b_3 * 0\text{xed}) \oplus (b_6 * 0\text{x5c}) \oplus (b_4 * 0\text{x50}) \oplus b_0$
10: $t_0 \leftarrow b \cdot r_1$ ▷ Using (Algorithm 1)
11: $t_0 \leftarrow t_0 \oplus r_0$
12: `isZero` $\leftarrow$ VCMPEQQ_N_U8$(t_0, 0)$
13: $b \leftarrow$ VPSELQ_U8$(r_0, t_0,$ `isZero`$)$
14: $r_0 \leftarrow r_0 \cdot b$
15: **return** r_0

Algorithm 7. b^{-1} via lookup table

Require: $b \in GF(2^8)^{16}$
Ensure: b^{-1}
1: $mask \leftarrow$ `0x69` ▷ 0x69 . . . 0x69 (16 bytes)
2: $r_0 \leftarrow b \wedge mask$
3: $r_1 \leftarrow r_0 \oplus (r_0 \ll 4)$
4: $idx \leftarrow r_1 \gg 4$
5: **return** LUT$[idx]$

optimization flag. We benchmarked three inversion routines (Algorithms 5, 6 and 7) alongside our implementation of the exponentiation-based method of Rivain and Prouff [Rivain and Prouff 2010]. Since their article does not provide a direct link to a public implementation, we implemented the method directly from its algebraic description; all three versions rely on identical auxiliary functions and control flow, so the only difference lies in the inversion core itself, enabling an accurate cycle-level comparison under identical conditions. In Table 1 we show the costs for the operations discussed in Section 3.

Table 1. Cycle Cost per Operation.

Operation	Cycles
Multiplication (Alg. 1)	18
One squaring (Alg. 1)	18
Four squarings (Alg. 3)	24
Affine transformation (Alg. 2)	10
MixColumns (Alg. 4)	14
ShiftRows (Permutation)	10

Table 2. Cycle counts on Cortex-M85.

	S-box primitives			AES-128 enc.	
Method	a^{-1}	1×S-Box	2×S-Box	1-block	2-blocks
Algorithm 5	137	129	201	1532	2563
Algorithm 6	108	112	190	1457	2485
Algorithm 7	74	82	160	1086	2109
Rivain–Prouff	150	152	219	1690	2746
Fixslicing	—	—	—	—	2114

We also evaluate full AES-128 encryption by reporting cycle counts for four variants: AES with Algorithm 5, AES with Algorithm 6, AES with our implementation of Rivain-Prouff inverse [Rivain and Prouff 2010], and the state-of-art fully-fixsliced implementation by Adomnicai and Peyrin [Adomnicai and Peyrin 2020]. We report timings for encrypting one or two blocks using AES-128, whereas the implementation in [Adomnicai and Peyrin 2020] supports only the two-blocks (128×2-bit) computation.

Table 2 summarises the micro-benchmarks for the isolated finite-field operations. All our implementations exploit GCC's `inline` [Free Software Foundation 2025] expansion to merge Algorithm 2 with inversion into single code block, eliminating redundant operations; in particular, Algorithm 5 combines both steps so effectively that even with the extra affine step the combined S-box is approximately 6% faster than the standalone inversion. Replacing the Rivain–Prouff core with Algorithm 6 yields a 28% reduc-

tion in inversion cost and a 26% reduction in S-box cost. Algorithm 7 further accelerates inversion and S-box evaluation by 51% and 46%, respectively.

The impact on full AES-128 encryption is also shown in Table 2. For the single-block mode, the 16 byte LUT-based S-box (Algorithm 7) accelerates encryption by 36% over Rivain and Prouff's implementation [Rivain and Prouff 2010], Algorithm 6 achieves a 14% speed-up; Adomnicai and Peyrin's fixsliced approach [Adomnicai and Peyrin 2020] does not support single-block mode. In two-block mode, the LUT-based variant improves AES-128 by 23% over Rivain and Prouff, slightly outperforming the fixsliced reference, while the table-free schedule yields a 15% gain. While Algorithm 7, utilizing a 16-byte lookup table, achieves competitive performance with the fixsliced approach, future research should focus on enhancing Algorithm 6 without relying on lookup tables.

Algorithm 6 also supports a single block AES computation, while the fixsliced code of Adomnicai and Peyrin [Adomnicai and Peyrin 2020] was designed for processing two blocks. This capability is valuable in contexts requiring exactly one AES call, such as in the FAEST key generation phase[1], and the same inverse routine can be reused during FAEST verification and signing, where the inverse is evaluated separately. Accordingly, Algorithm 6 offers an efficient option for memory constrained Helium enabled devices.

Acknowledgments

The authors gratefully acknowledge the financial support provided by the Technology Innovation Institute (TII) during the development of this work.

References

Adomnicai, A. and Peyrin, T. (2020). Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-cortex m and RISC-v. Cryptology ePrint Archive, Paper 2020/1123.

Aguilar, C., Blazy, O., Deneuville, J.-C., Gaborit, P., and Zémor, G. (2016). Efficient encryption from random quasi-cyclic codes. *arXiv preprint arXiv:1612.05572*.

Arm Ltd. (2023). Armv8-M architecture reference manual
mainline and helium extension. Technical Report DDI0553B, Revision B, Arm Ltd., Cambridge, UK. Accessed: 2025-08-02.

Babu, T. R., Murthy, K. V. V. S., and Sunil, G. (2011). Implementation of AES algorithm on ARM. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, page 1211–1213, New York, NY, USA. Association for Computing Machinery.

Dimitrov, V. and Järvinen, K. (2013). Another look at inversions over binary fields. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 211–218.

Fan, H. (2020). A trace based $GF(2^n)$ inversion algorithm. Cryptology ePrint Archive, Paper 2020/482.

[1] `https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/faest-spec-round2-web.pdf`

Free Software Foundation (2025). *An Inline Function is As Fast As a Macro*. GNU Compiler Collection (GCC). Section “Inline” (An Inline Function is As Fast As a Macro), accessed on 03 August 2025.

Fujii, H., Rodrigues, F. C., and López, J. (2019). Fast AES implementation using ARMv8 ASIMD without cryptography extension. In *International Conference on Information Security and Cryptology*, pages 84–101. Springer.

Gentry, C., Halevi, S., and Smart, N. P. (2012). Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Paper 2012/099.

Itoh, T. and Tsujii, S. (1988). A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases. *Inf. Comput.*, 78(3):171–177.

Kim, H. and Seo, H. (2025). Optimizing AES-GCM on ARM Cortex-M4: A fixslicing and FACE-based approach. Cryptology ePrint Archive, Paper 2025/512.

Marsh, J. (2020). *Arm Helium Technology M-Profile Vector Extension (MVE) for Arm Cortex-M Processors Reference Book*. Arm Education Media.

Morita, H., Pohle, E., Sadakane, K., Scholl, P., Tozawa, K., and Tschudi, D. (2024). MAESTRO: Multi-party AES using lookup tables. Cryptology ePrint Archive, Paper 2024/1317.

National Institute of Standards and Technology (2001). Advanced Encryption Standard. *NIST FIPS PUB 197*.

Rivain, M. and Prouff, E. (2010). Provably secure higher-order masking of AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer.

Schwabe, P. and Stoffelen, K. (2017). All the AES You Need on Cortex-M3 and M4. In Avanzi, R. and Heys, H., editors, *Selected Areas in Cryptography – SAC 2016*, pages 180–194, Cham. Springer International Publishing.

A. Illustration of polynomial multiplication using VMULL

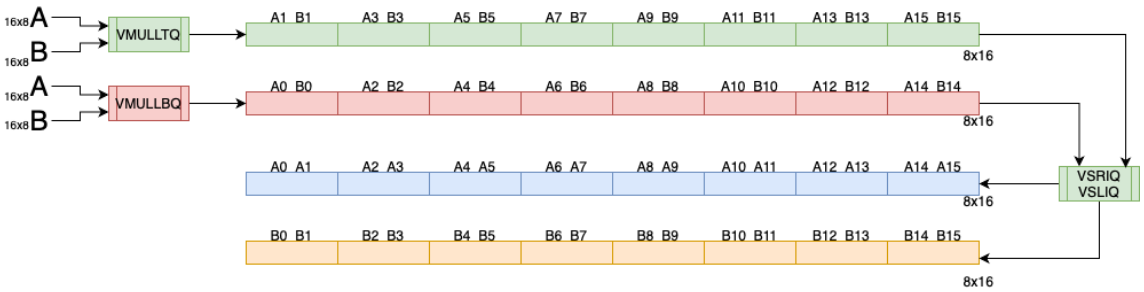

Figure 1. Illustration of the `VMULLT.P8` and `VMULLB.P8` instructions, which perform carry-less multiplication of two vectors composed of sixteen 8-bit polynomials each, simultaneously interleaving 128-bit results. The polynomial multiplication itself, described in Algorithm 1, corresponds directly to the carry-less multiplication portion illustrated here. Additionally, the logic of the `VSRI.16` instruction from Algorithm 2 is visually represented by the interleaving operation depicted in this figure.