Regular Article

Performance and Memory Trade-offs in SM4 Implementation on Embedded ARM Cortex-M Microcontrollers

Hoang-Gia Vu[1], Khanh-Tuong Tran[1], Dinh-Tuan Nguyen[1], Tuan-Khang Nguyen[2], Khanh-Nghia Truong[3], Hoai-Luan Pham[4]

[1] Falcuty of Radio-Electronics, Le Quy Don Technical University, Hanoi, Vietnam
[2] Centre 80, Department of Electronic Warfare, Hanoi, Vietnam
[3] Institute of Simulation Technology, Le Quy Don Technical University, Hanoi, Vietnam
[4] University of Information Technology, Vietnam National University - Ho Chi Minh city, Ho Chi Minh city, Vietnam

Correspondence: Hoang-Gia Vu, giavh@lqdtu.edu.vn
Communication: received 25 October 2025, revised 29 March 2026, accepted 11 May 2026
Online publication: 30 May 2026, Digital Object Identifier: 10.21553/rev-jec.421

Abstract– **The SM4 block cipher has been widely adopted in security applications across embedded systems, particularly as part of China's national cryptographic standards. However, its practical deployment on resource-constrained microcontrollers remains challenging due to limited processing power and memory. This paper investigates the trade-offs between performance and memory usage in various SM4 software implementations on ARM Cortex-M microcontrollers. We evaluate and compare three implementation strategies: (1) S-box lookup tables stored in Flash and SRAM, (2) T-table optimization that combines substitution and transformation operations, and (3) direct computation of the S-box using Galois Field (GF) logic. Each implementation is benchmarked on a 32-bit STM32 microcontroller to measure encryption latency, SRAM and Flash memory usage, and code complexity. The results reveal that the T-table implementation in SRAM provides the best performance with the lowest encryption latency, albeit at the cost of high SRAM consumption. Conversely, the GF logic implementation minimizes memory usage but suffers from the slowest execution time. This study provides important insights for selecting the most suitable SM4 implementation strategy for embedded systems with varying resource constraints and real-time requirements.**

Keywords– **SM4, block cipher, embedded, ARM Cortex-M, performance, memory, trade-offs.**

1 Introduction

The proliferation of Internet-of-Things (IoT) and embedded applications has created a growing demand for lightweight cryptography tailored to devices with limited compute and memory resources. For example, NIST's new lightweight cryptography standard [1] explicitly targets "tiny devices that have limited resources and thus demand a compact security implementation". ARM Cortex-M microcontrollers (such as the 72 MHz Cortex-M3 in the STM32F103 [2]) are widely deployed in such systems. These MCUs offer only modest computational power and memory (e.g. up to 512 KB flash and 64 KB SRAM in the STM32F103 family). As a result, implementing symmetric ciphers in firmware requires careful balancing of execution speed against code and data footprint.

SM4 is a 128-bit block cipher standardized by China (GB/T 32907-2016) [3] and now adopted in ISO/IEC 18033 [4]. It operates on 128-bit blocks with a 128-bit key over 32 rounds. As with AES [5], SM4's nonlinear component (the S-box) is defined by a multiplicative inverse in $GF(2^8)$ followed by an affine transform. SM4 is used in Chinese national security standards (e.g. WAPI wireless LAN and TLS cipher suites) and is effectively mandated for many applications. Given its official status and growing adoption, efficient implementations of SM4 on low-power embedded platforms are of practical interest.

Implementing SM4 on a memory-constrained MCU entails trade-offs. A straightforward software approach is a 256-byte S-box lookup table. This yields very fast substitutions, but 256 bytes of table (plus code) can consume precious memory. Prior work notes that LUT-based S-box implementations "achieve high throughput but acquire large area", implying substantial memory usage. An alternative is to compute the S-box on-the-fly via Galois-field arithmetic (inversion and affine transforms). This GF-based method uses negligible table storage, but incurs more CPU cycles – lookup tables are known to produce simpler, faster code than fully algebraic S-box computations. An intermediate strategy (common in AES implementations) is to use T-tables: precomputed 32-bit tables that combine S-box substitution with linear diffusion. The T-table approach can reduce the number of operations per round but requires several kilobytes of table data. In sum, different SM4 implementations trade off RAM usage, flash usage, and execution time.

In this work, we systematically evaluate these trade-offs on an STM32F103VET6 microcontroller (72 MHz ARM Cortex-M3, 512 KB flash, 64 KB SRAM). We

compare three software strategies for realizing the SM4 S-box: (1) Direct lookup tables (stored either in SRAM or embedded in Flash), (2) T-tables (32-bit tables in SRAM or Flash), and (3) GF arithmetic (computing the S-box without precomputed tables). For each approach we measure the SM4 encryption throughput (cycles or time per block) as well as the resulting SRAM and Flash footprints.

Our contributions are as follows:

1) Systematic multi-strategy implementation and analytical formulation of SM4 on Cortex-M3. *We present a comprehensive software implementation study of the SM4 block cipher on an ARM Cortex-M3 microcontroller, covering three S-box realization strategies: plain lookup-table, T-table, and GF-based computation. Each strategy is implemented with variants that store precomputed tables either in SRAM or in Flash memory. In addition to the implementation itself, we provide a detailed mathematical formulation of the transformation from the S-box-based round function to the T-table structure, explicitly analyzing the interaction between the nonlinear substitution and the linear transformation L. This establishes a theoretical foundation for understanding implementation-level optimization in SM4.*

2) Quantitative and experimentally validated performance–memory trade-off analysis. *In the bare-metal firmware environment, we systematically measure encryption latency, SRAM usage, and Flash footprint for all implementation variants. Unlike prior studies that report isolated cycle counts, this work jointly evaluates time and memory metrics and reveals how the architectural characteristics of Cortex-M3 influence substitution-diffusion optimization strategies. The results provide new experimental insights into the practical trade-offs between speed, code size, and data memory in resource-constrained embedded systems.*

3) Hybrid implementation strategies enabling fine-grained resource tuning. *To further explore the design space, we propose and evaluate two hybrid SM4 implementations. The first selectively partitions precomputed S-box tables between SRAM and Flash to balance access latency and memory footprint. The second combines GF-based computation with partial SRAM-based S-box storage to reduce SRAM consumption while maintaining competitive performance. These hybrid schemes demonstrate how implementation-level flexibility can be leveraged to tailor SM4 deployments to diverse embedded constraints, providing practical design guidelines for IoT-oriented systems.*

The remainder of the manuscript is structured as follows. Section 2 reviews the related work. Section 3 briefly introduces the SM4 algorithm and the STM32 microprocessor. Section 4 presents the implementation methods. Section 5 discusses the evaluation results with some discussion, and Section 6 concludes with a summary of the findings.

2 Related Work

In this section, we will review SM4 implementations on microcontrollers, general CPU, and the lightweight implementation of other block ciphers.

2.1 SM4 Implementations on Embedded Microcontrollers

Implementing the SM4 algorithm on resource-constrained microcontrollers, especially 32-bit ARM Cortex-M cores, involves tight memory–performance trade-offs. Kwon *et al.* [6] explicitly study such trade-offs on 8-bit AVR and 32-bit RISC-V cores. They find that placing the 256-byte SM4 S-box in fast SRAM yields 2-cycle lookups, whereas moving it to Flash (using the *LPM* instruction) saves RAM at the cost of a 3-cycle lookup. They also contrast fully-unrolled and looped code: unrolling eliminates branches (benefitting throughput) but increases code size, whereas looping cuts code footprint at the expense of extra cycles. As a result, for the speed-optimization, memory-optimization, and code size-optimization, they achieved 205.2 cycles per byte, 213.3 cycles per byte, and 207.4 cycles per byte, respectively.

Pu *et al.* [7] addressed the growing need for compact and high-throughput cryptographic engines in IoT devices by proposing three lightweight hardware implementations of the SM4 algorithm: *area-priority*, *speed-priority*, and *area-speed trade-off* designs. The *area-priority* scheme optimized the linear transformation functions L/L' to minimize register and XOR resource usage, achieving a very small footprint of only 2371 GE. The *speed-priority* design introduced two modified S-boxes that integrate the nonlinear substitution and linear transformation operations, effectively eliminating the delay caused by the linear layer. The *area-speed trade-off* scheme further merged the S-box linear mapping, its inverse, and the L/L' function into a composite-field implementation to reduce the encryption latency and to enhance the performance. Experimental results showed that the area-priority version reduced hardware area by 5.5 - 44.8% and consumed only 0.88 mW, while the speed-priority version reached a maximum frequency of 549 MHz and a throughput of 439.2 Mbps. These designs demonstrate that SM4 can be effectively optimized for hardware-constrained IoT platforms through judicious balancing of circuit area, power, and speed.

Similarly, Niu and Jiang [8] designed a MUX-based S-box architecture that minimizes hardware switching power, suggesting practical relevance for ultra-low-power microcontrollers and ASICs. These contributions underline the continued evolution of SM4 toward energy-efficient and memory-compact operation in embedded contexts.

2.2 SM4 Implementations on general-purpose CPU/GPU

On general-purpose CPUs, Guo [9] demonstrated highly efficient constant-time implementations leveraging Intel GFNI and ARM NEON instruction sets, achieving substantial speedups by parallelizing field arithmetic. Although these architectures differ from embedded microcontrollers, their underlying optimization concepts, exploiting data-level parallelism and minimizing memory stalls, are instructive for MCU firmware design. Similarly, Miao *et al.* [10] explored

bit-sliced implementations, treating multiple plaintext blocks as bit-vectors to maximize logical parallelism. While bit-slicing is not practical for single-block embedded encryption, it provides a theoretical upper bound for instruction-level efficiency.

Liu [11] implemented SM4 in Python for educational and prototyping purposes, providing an open reference for algorithm validation and intermediate testing. Though not performance-oriented, such high-level implementations support verification of embedded designs. In contrast, Li *et al.* [12] implemented SM4 using CUDA parallelism to explore throughput scaling on GPUs. This shares optimization ideas, such as loop unrolling and substitution merging, that can inform MCU-level firmware optimization.

2.3 Lightweight Implementation of Other Block Ciphers

The trade-offs between performance and memory footprint were also investigated for other block ciphers. Schwabe *et al.* report that fully unrolling AES on Cortex-M3 dramatically removes branch overhead at only modest code inflation [13]. This paper presents highly-optimized AES-128/192/256 assembly implementations tailored for ARM Cortex-M3 and M4, achieving roughly twice the speed of previous implementations. The authors emphasize architecture-specific scheduling and register allocation to minimize loads, providing a strong benchmark (cycle count, code size) for embedded cipher performance. Likewise, Seo *et al.* [14] implement the ARIA cipher on a Cortex-M3 using an 8×32-bit lookup table (combining S-box and linear layer) and carefully schedule it to match the 3-stage pipeline. Altogether, these microcontroller implementations underscore that SM4 on MCUs must judiciously choose Flash-based tables vs. SRAM calculations, trading a few CPU cycles against tens or hundreds of bytes of memory.

The present work extends these findings by providing a quantitative performance-memory evaluation of all three strategies S-box, T-table, and GF-based computation on a Cortex-M3 STM32 microcontroller. Moreover, it introduces two hybrid implementations that combine SRAM and Flash storage or integrate GF-logic with lookup tables, offering configurable trade-offs between speed and memory. Compared to prior SM4 implementations that focused primarily on speed optimization or hardware acceleration, this study contributes a balanced empirical analysis that reflects the practical constraints of embedded systems where both Flash and SRAM capacities are strictly limited.

3 Background

3.1 SM4 Block Cipher

SM4 operates on 128-bit data blocks using a 128-bit key over 32 encryption or decryption rounds. Let the input plaintext be divided into four 32-bit words: (X_0, X_1, X_2, X_3). Each encryption round computes

$$X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i), \quad (1)$$

where RK_i is the round key for round i, and the T function can be defined as

$$T(a) = L(\tau(a)), \quad (2)$$

Here, $\tau(a)$ is a nonlinear byte-wise substitution using the SM4 S-box, applied to each byte a_0, a_1, a_2, a_3.

$$\tau(a) = (S(a_0), S(a_1), S(a_2), S(a_3)), \quad (3)$$

and $L()$ is a linear transformation:

$$\begin{aligned} L(B) = B \oplus B \lll 2 \oplus B \lll 10 \\ \oplus B \lll 18 \oplus B \lll 24, \end{aligned} \quad (4)$$

where $\lll$ denotes left rotation. The 32 round keys RK_0 to RK_{31} are derived from the master key using a key expansion algorithm that also applies the S-box and linear transforms. SM4's performance in embedded systems depends heavily on how the S-box is implemented via lookup tables (in Flash or SRAM) or computed on-the-fly using finite field arithmetic, impacting both execution time and memory usage.

3.2 STM32F103 Processors

The STM32F103 microcontroller family is based on the ARM Cortex-M3 core and is widely used in embedded and real-time applications. It has Harvard architecture with separate buses for instructions stored in Flash and data in SRAM. The Cortex-M3 core supports a three-stage pipeline (fetch, decode, execute) but does not feature out-of-order execution, cache, or advanced branch prediction mechanisms. Due to these architectural constraints, software performance heavily depends on memory access patterns and instruction efficiency.

In the STM32F103, Flash memory has significantly slower access latency compared to SRAM. Although Flash reads are prefetch-buffered, table-heavy cryptographic routines stored in Flash may still suffer noticeable performance degradation, especially under tight loops. In contrast, SRAM, offers faster and more deterministic access but is much more limited in size.

These architectural features make the STM32F103 an ideal testbed for studying trade-offs in cryptographic algorithm implementations, where small memory sizes, lack of hardware acceleration, and predictable performance are central concerns. Optimizing cryptographic primitives such as SM4 on this platform requires careful balancing between execution speed and memory utilization, both Flash and SRAM.

4 Implementation Methods

In this section we present three implementation methods for the SM4 algorithm: S-box lookup table; T-table implementation; GF-based S-box computation.

4.1 S-box Lookup Table

The most straightforward method to implement the non-linear transformation of the SM4 algorithm on an embedded platform is through a direct S-box lookup.

In the SM4 round function, the intermediate 32-bit word is partitioned into four 8-bit sub-blocks, each of which is substituted independently using the fixed 8-bit S-box defined in the SM4 specification. This S-box is a bijective mapping over the finite field GF(2^8), constructed using an affine transformation combined with the multiplicative inverse in the field. It provides the non-linearity that is essential for cryptographic security. In practice, the S-box is realized as a 256-entry lookup table, where each entry corresponds to an 8-bit input value and stores the corresponding 8-bit output. When executing the SM4 algorithm, each byte of the input word (a_0, a_1, a_2, a_3) is used as an index to retrieve the substituted value $(S(a_0), S(a_1), S(a_2), S(a_3))$. The outputs are then concatenated to form a 32-bit word B, which subsequently undergoes the linear transformation as Equation (4).

The advantage of this approach lies in its conceptual simplicity and close correspondence to the algorithm's specification. However, on STM32 microcontrollers, the efficiency of the S-box implementation depends heavily on the memory location in which the lookup table is stored. Therefore, there are two variants evaluated:

- **LUT-SRAM:** The S-box table is placed in SRAM during initialization.
- **LUT-Flash:** The S-box table is kept in Flash and accessed directly during encryption.

If the table resides in Flash memory, each access requires wait states due to the relatively slower access latency, thereby increasing the overall encryption time. Conversely, placing the S-box table in SRAM allows for single-cycle access, reducing substitution latency and improving performance at the expense of limited SRAM resources. Thus, the S-box implementation represents a baseline method that is memory-efficient in Flash-based form but can achieve higher throughput when allocated in SRAM. Its performance and memory trade-offs provide a meaningful reference point for comparison with more optimized techniques, such as the T-table approach.

4.2 T-table Implementation

The T-table optimization moves the S-box and the linear $L()$ operations into precomputed 32-bit lookups. Since $L()$ is a linear (bitwise XOR and rotate) function over GF(2), it satisfies the following property

$$L(u \oplus v) = L(u) \oplus L(v). \tag{5}$$

We exploit this by precomputing, for each possible byte value x in $[0, 255]$ and for each byte position j, the 32-bit result of applying $S(x)$ in that byte position followed by the function $L()$. We now define four tables T_0, T_1, T_2, and T_3, each of 256 entries, as follows:

$$T_0[x] = L(S(x) * 2^{24}), \tag{6}$$
$$T_1[x] = L(S(x) * 2^{16}), \tag{7}$$
$$T_2[x] = L(S(x) * 2^{8}), \tag{8}$$
$$T_3[x] = L(S(x)). \tag{9}$$

It is noted that each entry is a 32-bit word. In words, $T_0[x]$ is the result of putting $S(x)$ in the most-significant byte and zero elsewhere, then applying $L()$ function; $T_1[x]$ puts $S(x)$ in the next byte position, etc. These tables encode the combined $\tau()$ and $L()$ functions for a single byte. In particular, if $a = (a_0, a_1, a_2, a_3)$, then by linearity we have

$$\begin{aligned} T(a) = L(\tau(a)) &= L\left(\sum_{j=0}^{3} S(a_j) \cdot 2^{8(3-j)}\right) \\ &= \big(L(S(a_0) \cdot 2^{24})\big) \oplus \big(L(S(a_1) \cdot 2^{16})\big) \\ &\oplus \big(L(S(a_2) \cdot 2^{8})\big) \oplus \big(L(S(a_3))\big) \\ &= T_0[a_0] \oplus T_1[a_1] \oplus T_2[a_2] \oplus T_3[a_3]. \end{aligned} \tag{10}$$

Thus the round update can be computed by table lookups

$$X_{i+4} = X_i \oplus T_0[a_0] \oplus T_1[a_1] \oplus T_2[a_2] \oplus T_3[a_3], \tag{11}$$

with (a_0, a_1, a_2, a_3) the bytes of $a = X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i$. This replaces four S-box lookups and five word-rotations by four 32-bit table loads and three XOR operations. In effect, each table T_j corresponds to one input byte position (with T_0 for the most significant byte and T_3 for the least significant).

To derive these tables, note first that for a given byte value x, the S-box produces an 8-bit output $S(x)$. When $S(x)$ is placed in the j-th byte of a 32-bit word (i.e. multiplied by $2^{8(3-j)}$) and then fed into $L()$ function, the result is a fixed 32-bit value. When j goes from 0 to 3, we have four formulas (6-9). Each table entry can be computed offline by performing the S-box lookup and then applying the bitwise rotations of the $L()$ function to that single 32-bit word. For example, $T_0[x]$ is computed by taking the byte value x, computing $s = S(x)$, forming the 32-bit word $B = s * 2^{24}$ (for example, s in the top byte), and then computing $L(B)$ using formula (4).

It is noted that L is defined only in terms of rotations and XOR operations, it commutes with rotations:

$$L(B \lll k) = L(B) \lll k, \tag{12}$$
$$L(B \ggg k) = L(B) \ggg k \tag{13}$$

When the input word is rotated, then this rotates all its shifted versions. As a consequence, the result is simply rotated at the output. We now consider $W_0 = S(x) * 2^{24}$. This places $S(x)$ in the most significant byte (big-endian). If W_0 is rotated right by 8 bits, then $S(x)$ moves to the next byte position

$$W_1 = W_0 \ggg 8 = S(x) * 2^{16}. \tag{14}$$

So W_1 is exactly the word used in the definition of $T_1[x]$

$$T_1[x] = L(W_1) = L(W_0 \ggg 8). \tag{15}$$

Using the commutativity property of L with rotations

$$L(W_0 \ggg 8) = L(W_0) \ggg 8. \tag{16}$$

But by definition $L(W_0) = T_0[x]$. Therefore, combining

formulas (14) and (15), we have

$$T_1[x] = T_0[x] \ggg 8. \tag{17}$$

The same reasoning shows the chain:

$$T_2[x] = T_0[x] \ggg 16, \tag{18}$$

$$T_3[x] = T_0[x] \ggg 24. \tag{19}$$

To derive the three lookup tables $T_1[x], T_2[x]$, and $T_3[x]$, we now have two options. First, these tables can be also stored in memory. This option will consume a large amount of memory footprint. Second, only the table $T_0[x]$ is stored in the memory. The three others are derived by shifting each entry of $T_0[x]$. This option will save the memory footprint, but increase the overall execution time due to the additional shifting operations.

Compared to the S-box lookup table method, the T-table implementation allows each encryption round to use fewer instructions and memory accesses. The T-table method accelerates processing but demands significant memory: 4 KB for the tables alone. Four storage variants will be evaluated:

- **T-Table-SRAM:** All tables stored in SRAM for faster access.
- **T-Table-Flash:** Tables stored in Flash to conserve SRAM.
- **T0-Table-SRAM:** Only T_0 table is stored in SRAM. T_1, T_2, and T_3 are derived by shifting T_0 right 8 bit, 16 bit, and 24 bit, respectively.
- **T0-Table-Flash:** Only T_0 table is stored in Flash memory. T_1, T_2, and T_3 are derived by shifting T_0 right 8 bit, 16 bit, and 24 bit, respectively.

4.3 GF-based S-box Computation

The non-linear substitution box (S-box) in the SM4 block cipher is mathematically defined as the composition of a multiplicative inversion in the finite field $\mathrm{GF}(2^8)$ and a subsequent affine transformation over the vector space $\mathrm{GF}(2)^8$. This SM4 algorithm design, inherited from principles used in AES, ensures that the S-box exhibits high non-linearity, resistance against linear and differential cryptanalysis, and efficient diffusion of input differences. Unlike a simple lookup table, the S-box can be rigorously derived from field-theoretic properties, which makes it possible to compute it directly from algebraic operations in software-constrained environments such as embedded microprocessors.

Each input byte $a \in \{0,1\}^8$ is interpreted as an element of the finite field $\mathrm{GF}(2^8)$. The field is constructed as

$$\mathrm{GF}(2^8) = \mathrm{GF}(2)[x]/p(x). \tag{20}$$

Where $p(x)$ is the irreducible polynomial

$$p(x) = x^8 + x^7 + x^6 + x + 1. \tag{21}$$

Thus, the bit string $a = (a_7, a_6, \ldots, a_0)$ corresponds to the polynomial

$$a(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0. \tag{22}$$

Addition in this field is simply bitwise XOR, while multiplication involves polynomial multiplication followed by reduction modulo $p(x)$. The core non-linearity arises from inversion in $\mathrm{GF}(2^8)$. For nonzero a, its multiplicative inverse b is defined by

$$a(x) * b(x) \equiv 1 \pmod{p(x)}. \tag{23}$$

Because $\mathrm{GF}(2^8)$ is a finite field, every nonzero element has a unique inverse. The inversion can be computed using the extended Euclidean algorithm applied to polynomials over $\mathrm{GF}(2)$

$$\exists u(x), v(x) : a(x) * u(x) + p(x) * v(x) = 1. \tag{24}$$

Reducing modulo $p(x)$, we obtain

$$a(x) * u(x) \equiv 1 \pmod{p(x)}. \tag{25}$$

Hence $u(x)$ is the multiplicative inverse of $a(x)$. For the special case $a = 0$, inversion is undefined, and the S-box maps this input to 0. After inversion, the result b is expressed as an 8-bit vector

$$b = (b_7, b_6, \ldots, b_0)^T. \tag{26}$$

The affine transformation is defined as

$$S(a) = A * b \oplus c, \tag{27}$$

where A is a fixed 8×8 binary matrix and c is a fixed binary vector. Multiplication is over $\mathrm{GF}(2)$, so each output bit is the XOR of selected input bits plus a constant. The explicit matrix A and constant vector c for SM4 are:

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}, \tag{28}$$

$$c = (1,1,0,0,0,1,1,0)^T. \tag{29}$$

This transformation introduces linear diffusion, mixing the bits of the inverse and removing structural weaknesses. This construction of the S-box ensures strong cryptographic properties. The inversion function provides a highly non-linear mapping, which is difficult to approximate with linear functions. The subsequent affine transformation prevents symmetries in the field structure from leaking into the cipher's operation.

This method minimizes memory usage, requiring no S-box or T-table storage. However, it introduces substantial computational overhead due to the field inversion operation. Despite its lower performance, this approach is well suited for platforms where SRAM and Flash space are severely constrained.

4.4 Hybrid Methods

In this section, we propose two hybrid implementations of the SM4 algorithm. The purpose of the hybrid methods is to balance the SRAM usage and the encryption latency. The first implementation combines SRAM

and Flash, called SRAM-Flash, for precomputed S-box storage as follow

$$S(x) = \begin{cases} S_{\text{SRAM}}(x) & \text{for } x \leq q,\ 1 \leq q < 255 \\ S_{\text{FLASH}}(x) & \text{for } x > q \end{cases}. \quad (30)$$

In this scheme, we store the first $(q+1)$ elements of the S-box table in SRAM memory and the rest of the table in Flash memory. The value of q is selected from 0 to 244, depending on the amount of SRAM memory available in the design. The $S_{\text{SRAM}}(x)$ function accesses the $(q+1)$ element S-box table stored in the SRAM memory, while $S_{\text{FLASH}}(x)$ accesses the $(255-q)$ element S-box table stored in the Flash memory.

The other hybrid implementation combines the GF-based computation and the precomputed S-box on SRAM, called SRAM-GF, as follows

$$S(x) = \begin{cases} S_{\text{SRAM}}(x) & \text{for } x \leq q,\ 1 \leq q < 255 \\ S_{\text{GF}}(x) & \text{for } x > q \end{cases}. \quad (31)$$

In this scheme, we also store the first $(q+1)$ elements of the S-box table in SRAM memory. The rest of the table is not stored in memory, but calculated via GF-based computation. The value of q is selected from 0 to 254, depending on the amount of SRAM memory available in the design. The $S_{\text{SRAM}}(x)$ function accesses the $(q+1)$ element S-box table stored in the SRAM memory, while $S_{\text{GF}}(x)$ calculates the $S(x)$ function based on the finite field GF(2^8) for the remaining $(255-q)$ elements of the S-box table.

5 Experimental Results and Discussion

5.1 Experimental Results

All implementations were written in C with critical functions optionally optimized for the STM32F103VET6. The encryption routine was benchmarked on a bare-metal system without operating system overhead, using 128-bit plaintext inputs and fixed 128-bit keys. The processor ran at 72 MHz. Execution time, SRAM usage, and Flash usage were recorded for each variant.

5.1.1 Encryption Latency: To evaluate the encryption latency, we measure the encryption execution time for each SM4 implementation: LUT-SRAM, LUT-Flash, T-Table-SRAM, T-Table-Flash, T0-Table-SRAM, T0-Table-Flash, and GF-based computation. The evaluation was performed using a timer integrated in the microprocessor to count the number of clock cycles. Table I shows the number of clock cycles for each SM4 implementation. Since the processor core ran at 72 MHz, the encryption throughput can be calculated for each implementation as in Figure 1.

Table I
Clock Cycles for Different SM4 Implementations

LUT-SRAM	LUT-Flash	T-SRAM	T-Flash	T0-SRAM	T0-Flash	GF-based
2738	3102	2080	2812	2444	2506	707608

As can be seen from Table I, T-Table-SRAM has the lowest encryption latency at 2080 clock cycles, 364 cycles faster than T0-Table-SRAM. T-table-based implementations are much better than S-box-based implementations in both SRAM and Flash memory in terms of encryption latency and throughput. As a result, T-table-based implementations achieve higher encryption throughput, up to 4.5 MBps compared to 3.3 Mbps for LUT-SRAM implementation as in Figure 1. The table also reveals that the latency for the GF-based implementation is 707608 clock cycles, far longer than that for S-box or T-table lookup methods. Its encryption throughput is the lowest one at 13 Kbps, around more than 3000 times lower than the encryption throughput of the T-Table-SRAM implementation. However, this implementation do not require storing any lookup table on SRAM, but consumes more Flash memory used for the S-box calculation function.

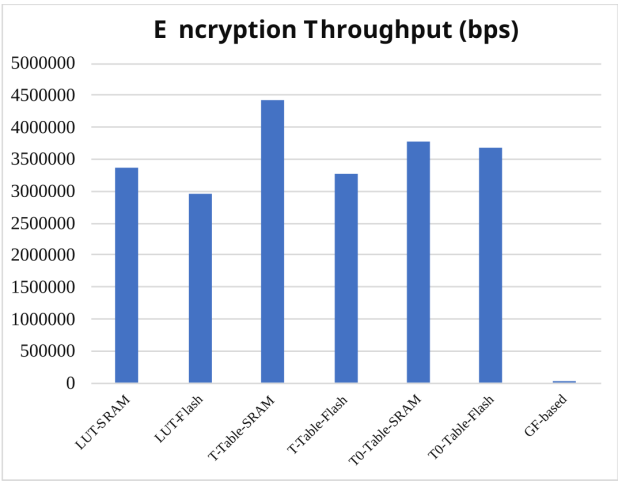


Figure 1. Encryption throughput for each SM4 implementation.

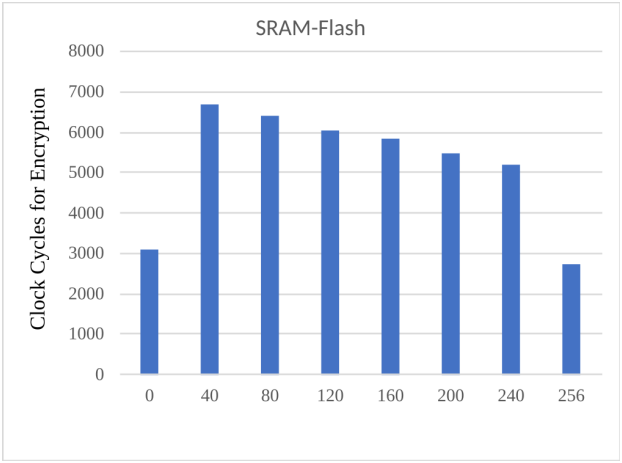


Figure 2. Encryption latency for hybrid SRAM-Flash S-box.

Figure 2 and Figure 3 reveal the encryption latency for the two hybrid implementations SRAM-Flash and SRAM-GF, respectively. As can be seen from Figure 2, the hybrid latencies are even longer than that of the pure SRAM-LUT or Flash-LUT implementation. These increases in encryption latency are caused by

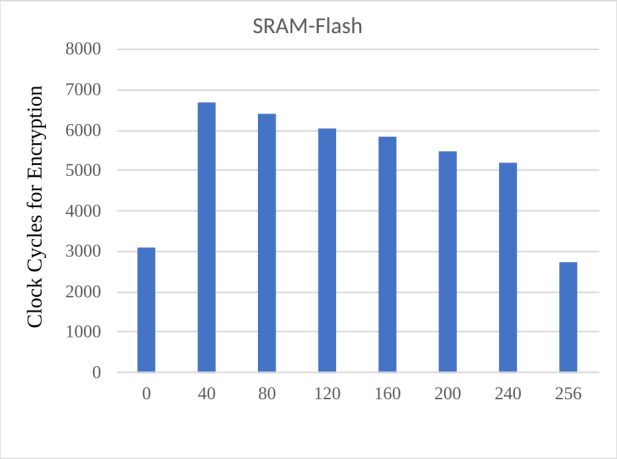


Figure 3. Encryption latency for hybrid SRAM-GF S-box.

the comparison and branch instructions to identify reading the S-box table from SRAM or Flash memory. In contrast, the hybrid SRAM-GF implementation shows the decrease in latency when scaling up the number of S-box elements on SRAM. However, the latencies are still much higher than that for the SRAM-LUT implementation.

5.2 Memory usage and energy consumption discussion

In this section we evaluate the memory footprint for each implementation in both SRAM and Flash memory. For the Flash memory footprint, we measure the size of output hex file used to flashed to the microprocessor. It is noted that this size of footprint also includes the source code for other functions, such as the LCD driver, the SPI functions, the Timer functions, and so on. For the SRAM memory, we only count the footprint for the lookup tables such as the S-box table or T-tables. It is noted that except the footprint for lookup tables, SRAM memory is also used to store the expanded key table, local variables, and other data arrays.

Table II
Memory Usage

Implementation	SRAM usage (bytes)	Flash usage (bytes)
SRAM-LUT	256	53045
Flash-LUT	0	53135
T-SRAM	4096	75312
T-Flash	0	75201
T0-SRAM	1024	58077
T0-Flash	0	57950
GF-based	0	53773
SRAM-Flash	X	53196
SRAM-GF	X	54538

Table II shows the SRAM and Flash memory usage for each SM4 encryption implementation. As can be seen from the table, T-Table-SRAM consumed the largest number of bytes in SRAM, while all Flash implementations did not use any SRAM byte for lookup tables. The T0-Table-SRAM implementation consumed less both SRAM and Flash memory, compared to T-Table-SRAM. However, it had a longer encryption latency at 2444 clock cycles comared to 2080 cycles for T-Table-SRAM. Table I and Table II also indicate that Flash-LUT was better than GF-based implementation in terms of Flash usage and encryption latency, while their SRAM usage are the same at zero byte. Table II also reveals that SRAM-Flash and SRAM-GF hybrid implementation consumed more Flash and SRAM footprint, while its encryption latency is much longer, than Flash-LUT. The 'X' value in the "SRAM usage" column is the number of bytes stored in SRAM for the S-box table ($256 > X > 0$), thus providing finer granularity in balancing memory allocation and performance trade-offs.

Table III
Latency and Memory Usage Comparison

Algorithm	MCU	Method	Latency (Clocks)
AES128 ECB [13]	ARM Cortex-M3 (72 MHz)	Lookup-table	659
ARIA-128 [15]	ARM Cortex-M3 (48 MHz)	Lookup-table	3584
SM4 [6]	8-bit AVR (16 MHz)	Lookup-table	3280
SM4 (Ours)	ARM Cortex-M3 (72 MHz)	Lookup-table	2738

5.3 Comparison

In order to evaluate the efficiency of different cryptographic algorithms on resource-constrained embedded platforms, a comparison of representative software implementations reported in previous studies is presented in Table III. The table summarizes the key performance indicator, which is the number of clock cycles required per encryption block. This metric provides an overview of the computational cost of widely used symmetric encryption algorithms when implemented on typical microcontroller architectures such as the ARM Cortex-M3 and AVR platforms.

As shown in Table III, different cryptographic algorithms exhibit distinct performance characteristics depending on both the algorithmic structure and the underlying hardware architecture. The AES-128 implementation on the Cortex-M3 [13] achieves relatively high efficiency due to the use of lookup-table optimizations, resulting in a low number of clock cycles per processed byte. In contrast, the ARIA-128 implementation [15] demonstrates a significantly higher computational cost, reflecting the complexity of its internal transformations when implemented in software.

The SM4 implementation on an 8-bit AVR microcontroller [6] presents a different scenario. Although the reported cycle count per byte appears competitive, the overall throughput is inherently limited by the lower computational capability and narrower data width of the AVR architecture compared to the 32-bit ARM Cortex-M3 in our implementation. Nevertheless, the results indicate that SM4 can still be implemented effectively on highly constrained devices when appropriate optimization techniques are applied.

Overall, the comparative evaluation highlights the importance of considering both algorithm design and

hardware characteristics when implementing cryptographic primitives in embedded systems. Achieving an optimal balance between computational efficiency and memory consumption is essential, particularly for Internet-of-Things (IoT) devices and other resource-limited platforms where both processing power and memory capacity are restricted.

5.4 Discussion

Since the energy consumption of a microcontroller can be approximated as $E = P \times T$, and the power P remains relatively stable under fixed operating conditions, the energy consumption is largely proportional to execution time T (or cycle count). Therefore, the measured encryption latency provides a reasonable proxy for comparing the relative energy efficiency of different implementations. Based on this analysis, implementations with lower execution time (e.g., T-table in SRAM) are expected to consume less energy, while computation-heavy approaches such as GF-based S-box incur higher energy cost despite reduced memory usage.

As can be seen from Figure 2, hybrid implementations do not achieve the lowest latency compared to the pure LUT-SRAM or LUT-Flash approaches. However, the primary objective of the hybrid designs is not to maximize performance, but to provide a flexible trade-off between memory usage and execution time for resource-constrained embedded systems. Specifically, the SRAM-Flash hybrid reduces SRAM consumption by partially storing lookup tables in Flash, at the cost of a significant increase in latency. For instance, a hybrid S-box configuration that allocates 120 bytes in SRAM and 136 bytes in Flash results in approximately a 100% increase in encryption latency compared to the LUT-SRAM implementation. Despite this performance degradation, such a configuration is advantageous for microcontrollers with highly constrained memory resources. In these scenarios, fully storing lookup tables entirely in SRAM or entirely in Flash may be impractical, and the hybrid approach provides a viable compromise between memory footprint and execution time.

Similarly, the SRAM-GF hybrid significantly reduces memory footprint by replacing part of the lookup operations with on-the-fly GF-based computation. Although this introduces much computational overhead as in Figure 3, it provides an alternative design point for systems where memory constraints are more critical than execution speed.

We can see from Table I that the GF-based implementation exhibits significantly higher latency compared to LUT-based and T-table approaches, and therefore is not suitable for performance-critical applications. However, the purpose of including the GF-based method in this study is not to compete with high-performance implementations, but to provide a memory-minimal baseline for SM4 realization on resource-constrained platforms.

In particular, the GF-based approach eliminates the need for lookup tables, thereby achieving near-zero SRAM usage and reduced memory footprint. This characteristic is valuable for ultra-constrained embedded systems, where even small lookup tables may not be affordable, or in scenarios where minimizing memory usage or avoiding table-based implementations is desirable (e.g., for improved resistance to certain memory-based side-channel attacks). Furthermore, the GF-based implementation serves as a key component in the proposed hybrid SRAM-GF design, where partial substitution is computed on-the-fly to reduce memory requirements while maintaining acceptable performance.

After the evaluation for the SM4 implementations on the STM32 microprocessor, we make the following suggestions:

- Using T-Table-SRAM implementation when the application requires very high SM4 encryption throughput and there is much SRAM and Flash memory footprint available.
- Using T0-Table-Flash implementation when the application requires high SM4 encryption throughput and there is much Flash but limited SRAM memory footprint available.
- Using SRAM-LUT implementation when the application requires high SM4 encryption throughput and there is limited SRAM and Flash memory footprint available.
- Using Flash-LUT implementation when the application does not requires high SM4 encryption throughput and there is limited SRAM and Flash memory footprint available.
- Not using GF-based implementation when the application requires low SM4 encryption latency.

6 Conclusion

This paper presented a comprehensive performance and memory evaluation of the SM4 block cipher on ARM Cortex-M microcontrollers, focusing on the trade-offs among three different implementation methods: lookup-table–based substitution, T-table optimization, and Galois Field (GF)-based S-box computation. Experimental results obtained from the STM32F103 platform demonstrated that the T-table implementation stored in SRAM provides the highest encryption throughput, owing to reduced memory access latency and merged substitution–transformation operations. However, this method incurs substantial SRAM usage, making it less suitable for systems with tight memory budgets.In contrast, the GF-based implementation achieves the smallest SRAM and Flash footprint, yet its runtime performance is limited by the computational overhead of field arithmetic. The standard S-box lookup approach offers a middle ground, achieving moderate performance with predictable memory use, depending on whether the table is placed in Flash or SRAM.

Through systematic benchmarking and analysis, we have highlighted the intrinsic trade-off between computational speed and memory utilization when implementing SM4 in resource-constrained embedded systems.This study provides practical design guidelines for selecting appropriate implementation strategies according to hardware resources and real-time encryption

requirements. In particular, applications prioritizing low latency, such as secure communication protocols or real-time data protection, benefit from SRAM-based T-tables, while memory-critical environments, such as sensor nodes or IoT endpoints, should consider Flash-LUT, GF-based computation or hybrid approaches.

Future work will extend this investigation along several directions. First, power consumption and energy-per-encryption metrics will be incorporated into the evaluation framework to quantify efficiency for battery-powered devices. Second, side-channel resistance and timing-attack mitigation techniques will be investigated, particularly for table-based and hybrid implementations. These extensions will contribute to a deeper understanding of the trade-offs among performance, memory, and security in practical SM4 deployments for embedded cryptographic systems.

References

[1] NIST, "Lightweight Cryptography." [Online]. Available: https://csrc.nist.gov/projects/lightweight-cryptography

[2] STMicroelectronics, "STM32F103 documentation." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f103/documentation.html

[3] Standardization Administration of China, "GB/T 32907-2016: Information security technology - SM4 block cipher algorithm," Tech. Rep., 2016.

[4] ISO/IEC, "ISO/IEC 18033: Information security - Encryption algorithms." [Online]. Available: https://www.iso.org/standard/81564.html

[5] NIST, "Advanced Encryption Standard (AES)," Tech. Rep. FIPS PUB 197, 2001.

[6] H. Kwon, H. Kim, S. Eum, M. Sim, H. Kim, W.-K. Lee, Z. Hu, and H. Seo, "Optimized implementation of SM4 on AVR microcontrollers, RISC-V processors, and ARM processors," *IEEE Access*, vol. 10, pp. 80 225–80 233, 2022.

[7] J. Pu, Y. Teng, Q. Gao, X. Zheng, and Y. Xu, "Internet of Things Oriented SM4 Lightweight Optimization Implementation," *Acta Electronica Sinica*, vol. 52, no. 6, pp. 1888–1895, 2024.

[8] Y. B. Niu and A. P. Jiang, "A Low Power Design of SM4 Cipher Based on MUX S-Box Architecture," *Applied Mechanics and Materials*, vol. 411, pp. 125–130, 2013.

[9] W. Guo, "Efficient Constant-Time Implementation of SM4 with Intel GFNI instruction set extension and Arm NEON coprocessor," *Cryptology ePrint Archive*, 2022.

[10] X. Miao, L. Li, C. Guo, M. Wang, and W. Wang, "Bit-Sliced Implementation of SM4 and New Performance Records," *IET Information Security*, vol. 2023, no. 1, p. 1821499, 2023.

[11] L. Liu, "Software Implementation of SM4 in Python Programing Language," in *Proceedings of the International Conference on Modern Science and Scientific Studies*, vol. 3, no. 4, 2024, pp. 215–224.

[12] J. Li, W. Xie, L. Li, and X. Wu, "Parallel Implementation and Optimization of SM4 Based on CUDA," in *Proceedings of the Applied Cryptography in Computer and Communications*, ser. Lecture Notes, vol. 386. Springer, 2021, pp. 93–104.

[13] P. Schwabe and K. Stoffelen, "All the AES You Need on Cortex-M3 and M4," in *Proceedings of the Selected Areas in Cryptography – SAC*, ser. LNCS, vol. 10532. Springer, 2016, pp. 180–194.

[14] S. Eum, H. Kim, H. Kwon, M. Sim, G. Song, and H. Seo, "Parallel Implementations of ARIA on ARM Processors and Graphics Processing Unit," *Applied Sciences*, vol. 12, no. 23, p. 12246, 2022.

[15] H. Seo, H. Kim, K. Jang, H. Kwon, M. Sim, G. Song, and S. Uhm, "Compact Implementation of ARIA on 16-Bit MSP430 and 32-Bit ARM Cortex-M3 Microcontrollers," *Electronics*, vol. 10, no. 8, p. 908, 2021.

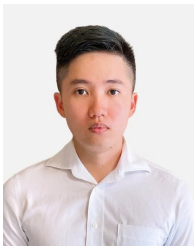
Hoang-Gia Vu received the B.E. and M.E. degrees in Electrical Engineering from Le Quy Don Technical University , Vietnam in 2007 and 2010, respectively. He received the PhD degree in Computer Engineering from Nara Institute of Science and Technology, Japan in 2018. He is currently a lecturer and a researcher at the Faculty of Radio-Electronics, Le Quy Don Technical University. His research interests include embedded systems, reconfigurable computing, parallel computing, and processor architecture.

Khanh-Tuong Tran is currently a fourth-year undergraduate student pursuing the B.E. degree in Electronic Warfare Engineering at Le Quy Don Technical University, Vietnam, where he has been studying since 2022. His academic and research interests include electronic warfare systems, signal processing, embedded systems, and military communication technologies.

Dinh-Tuan Nguyen received the B.S. and M.S. degrees in Applied Mathematics and Physics from Moscow Institute of Physics and Technology, Russia in 2010 and 2012, respectively. During 2013-2018, and from 2023 until now, he has been a lecturer in Le Quy Don Technical University, Hanoi, Vietnam. Since 2018, he has been a PhD candidate in Department of Electrical Engineering and Information Technology, TU Wien, Austria. His research interests include THz electronics, sub-THz/THz resonant-tunneling-diode (RTD) oscillators, and slot antennas.

Tuan-Khang Nguyen received the B.E. and M.E. degrees in Electrical Engineering and Rada Engineering from Le Quy Don Technical University , Vietnam in 2007 and 2010, respectively. He is currently a researcher and developer at the Centre 80, Department of Electronic Warfare, Hanoi, Vietnam. His research interests include embedded systems, digital design, signal processing in electronic warfare.

Khanh-Nghia Truong received the B.E. and M.E. degrees in Computer Science from Le Quy Don Technical University , Vietnam in 2007 and 2012, respectively. From 2012 to the present, He has been involved in research and development of training simulation products at the Institute of Simulation Technology, Le Quy Don University.



Hoai-Luan Pham received a bachelor's degree in computer engineering from Vietnam National University Ho Chi Minh City—University of Information Technology (UIT), Vietnam, in 2018, and a master's degree and Ph.D. degree in information science from the Nara Institute of Science and Technology (NAIST), Japan, in 2020 and 2022, respectively. Since October 2022, he has been with NAIST as an Assistant Professor and with UIT as a Visiting Lecture. His research interests include blockchain technology, cryptography, computer architecture, circuit design, and accelerators.