박사학위논문

# Bitslicing for Block Ciphers on Embedded Microcontrollers and GPUs

## Implementation Techniques and Trade-Offs

2026년


한 성 대 학 교 대 학 원

정 보 컴 퓨 터 공 학 과

정보시스템공학전공

김 현 준

박사학위논문
지도교수 서화정

# Bitslicing for Block Ciphers on Embedded Microcontrollers and GPUs

## Implementation Techniques and Trade-Offs

2025년 12월 일


한 성 대 학 교 대 학 원

정 보 컴 퓨 터 공 학 과

정보시스템공학전공

김 현 준

박사학위논문
지도교수 서화정

# Bitslicing for Block Ciphers on Embedded Microcontrollers and GPUs

## Implementation Techniques and Trade-Offs

위 논문을 공학 박사학위 논문으로 제출함

2025년 12월 일


한 성 대 학 교 대 학 원

정 보 컴 퓨 터 공 학 과

정보시스템공학전공

김 현 준

김현준의 공학 박사학위 논문을 인준함

2025년 12월  일

심사위원장 박 명 서 (인)

심 사 위 원 석 병 진 (인)

심 사 위 원 서 화 정 (인)

심 사 위 원 김 수 리 (인)

심 사 위 원 이 동 재 (인)

# ABSTRACT

## Bitslicing for Block Ciphers on Embedded Microcontrollers and GPUs Implementation Techniques and Trade-Offs

KIM, HYUN-JUN

Major in Information System Engineering

Dept. of Information and Computer Engineering

The Graduate School

Hansung University

This dissertation investigates bitslicing and fixslicing as table-free, timing-uniform software implementation paradigms for block ciphers across heterogeneous platforms, ranging from resource-constrained 32-bit microcontrollers to massively parallel NVIDIA GPUs. In such environments, overall security and performance depend not only on the cipher design but also on concrete implementation choices-especially state layout, packing/unpacking strategy, and the selected slice degree-under strict constraints on registers, memory behavior, and parallel execution efficiency.

We present three case studies. First, we implement SPEEDY-5/6/7-192 on ARM Cortex-M3 and RV32I-based RISC-V using a 6×32 bitsliced state representation. By combining SWAPMOVE-based packing, a Boolean-network realization of the 6-bit SubBox, and rotation-centric constant-time diffusion (including

rotation-XOR fusion where supported), the SPEEDY-7-192 implementation improves from 15,407/18,096 cycles per byte (byte-oriented reference) to 85.1/109.2 cycles per byte on Cortex-M3/RV32I, respectively, while maintaining a fixed instruction trace and secret-independent memory access.
Second, for AES-GCM on ARM Cortex-M4, we build a 2-way fixsliced AES-CTR core and integrate FACE-style caching directly in the fixsliced domain. We also evaluate two GHASH design points that expose a practical performance-assurance trade-off: a compact 4-bit table multiplier and a table-free Karatsuba-based routine for strict constant-time deployments. FACE yields up to 19.4% improvement for long-message AES-GCTR, and the 4-bit GHASH option is roughly twice as fast as the Karatsuba baseline, at the cost of secret-derived table indices.
Third, we design high-degree bitsliced CUDA implementations of PRESENT and GIFT on an RTX 3060. Using 32-way bitslicing per thread, branch-free Boolean S-boxes, efficient bit permutations, and device-side bitsliced counter generation, we achieve peak exhaustive-search throughput of 214-584 Gbit/s and bulk-encryption throughput up to 85 Gbit/s (including host-device transfers).
Across these studies, we distill actionable cross-platform guidelines for choosing state representations and slice degrees to balance throughput, resource footprint, and timing-uniform execution on embedded microcontrollers and GPUs.

Keywords - bitslicing, fixslicing, block-cipher implementation, constant-time implementation, timing side-channel resistance, embedded microcontrollers, GPU acceleration

# Contents

# List of Tables

# List of Figures

# Ⅰ. Introduction

## 1.1 Background and Motivation

Block ciphers serve as a fundamental building block of modern information security systems. They ensure confidentiality and, when integrated with authenticated encryption modes, provide data integrity. Consequently, they are extensively deployed across a wide spectrum of applications, including Transport Layer Security (TLS/DTLS), virtual private networks (VPNs), disk and file-system encryption, wireless communication protocols, and blockchains. Alongside the Advanced Encryption Standard (AES), various algorithms have been continuously proposed and standardized. These range from lightweight block ciphers tailored for the Internet of Things (IoT) and embedded environments to new designs optimized for high-throughput processing in server and cloud infrastructures. In this evolving landscape, implementation-level security on physical hardware has become as critical as algorithmic security in determining the overall performance and robustness of a system.

Modern deployment environments are increasingly heterogeneous, characterized by two distinct extremes. On one end are resource-constrained embedded devices, such as 32-bit microcontrollers (e.g., ARM Cortex-M series) and RV32I-based RISC-V cores. These platforms operate under stringent constraints regarding clock frequency, flash/SRAM capacity, and register availability. In environments such as sensor nodes, gateways, and industrial controllers, block-cipher-based mechanisms are widely utilized for local data protection and firmware encryption and authentication. In such settings, implementations must achieve sufficient throughput to maintain acceptable response latency

while minimizing code size and memory usage to reduce pressure on system resources and energy consumption.

On the other end are high-performance parallel devices, such as GPUs in cloud data centers and high-performance server environments. As the volume of data in transit and at rest grows exponentially, encryption and decryption processes can become significant system bottlenecks. GPUs—capable of executing thousands of threads in parallel—are increasingly employed not only for bulk data processing but also for research workloads such as large-scale key searches and the performance evaluation of cryptographic designs. However, GPU architectures differ fundamentally from those of CPUs and microcontrollers; they follow a warp-based execution model, impose strict constraints on registers and shared memory, and exhibit high sensitivity to memory-access patterns. Consequently, implementation strategies that are optimal for embedded microcontrollers often prove inefficient when applied to GPUs.

Despite these contrasting characteristics, block cipher implementations on both classes of platforms must satisfy several common requirements:

**Efficiency:** Achieving high throughput and low latency within the available hardware and energy budgets.

Resource Optimization: Minimizing the footprint of flash, RAM, and registers to reduce contention with other system components.

**Side-Channel Resilience:** Maintaining a constant-time execution pattern to mitigate timing side-channel attacks, where execution time may reveal sensitive data through cache states or branch outcomes.

To meet these requirements, traditional table-based techniques (e.g., T-tables) are often replaced by more robust paradigms. While

table-based approaches offer straightforward structures, they often introduce data-dependent memory accesses that are vulnerable to timing attacks. Furthermore, lookup tables increase memory footprints and can incur bus bottlenecks on microcontrollers or memory coalescing issues on GPUs. In response to these challenges, bitslicing and Fixslicing have emerged as influential implementation paradigms. Originally developed for SIMD-equipped CPUs, these techniques realize S-boxes and linear layers purely through word-level logical operations. This approach not only eliminates table-based vulnerabilities but also allows for efficient parallel processing tailored to the specific characteristics of the target platform.

Accordingly, this dissertation investigates bitsliced and Fixsliced software implementations of block ciphers across heterogeneous platforms, spanning resource-constrained microcontrollers and massively parallel GPUs. Through multiple case studies, it derives actionable guidelines for selecting state representations and bit-slice degrees under performance, resource, and strict constant-time constraints.

## 1.2 Research Gap and Research Questions

Despite the advancements in bitslicing techniques, existing literature exhibits several limitations when viewed from a cross-platform perspective:

Fragmentation of Optimization Efforts: Most studies focus on highly specific cipher-platform pairs (e.g., “AES on Cortex-M4”). While valuable, these point solutions offer limited insights into the general principles that apply across diverse architectures.

Narrow Focus on Throughput: Bitslicing is often treated primarily as an acceleration technique. There is a lack of systematic analysis regarding

how bitslicing decisions shape the trade-offs between performance, resource utilization, and timing-uniformity constraints.

Insufficient Heterogeneous Analysis: There are relatively few studies that apply comparable implementation paradigms across both embedded microcontrollers and GPUs to evaluate them under a common set of metrics.

Limited Side-Channel Evaluation: Quantitative analysis of how design choices—such as table-based versus bit-operation-based constructions—impact practical timing variability remains insufficient.

This dissertation addresses these gaps by exploring the following research questions:

1) Which bitsliced and Fixsliced design patterns recur effectively across different hardware platforms?

2) How should the state representation and bit-slice degree be selected under platform-specific resource constraints?

3) Under strict constant-time requirements, what performance and resource trade-offs arise among competing implementation options?

## 1.3 Contributions and Structure

This dissertation compares and analyzes bitsliced software implementations of block ciphers across heterogeneous platforms. It systematizes results for SPEEDY, AES-GCM, PRESENT, and GIFT on ARM Cortex-M3/M4, RV32I-based RISC-V, and NVIDIA GPUs, analyzing them under a unified framework.

The main contributions of this work are summarized as follows:

A Unified View of Bitslicing: This research provides a shared

perspective on bitslicing and Fixslicing across heterogeneous platforms, establishing a basis for systematic cross-platform comparison based on state representation and resource usage.

Comparative Analysis of SPEEDY on Embedded Cores: The dissertation evaluates single-block (intra-state) bitsliced SPEEDY implementations on ARM Cortex-M3 and RISC-V, identifying how architectural features like barrel shifters influence performance.

Constant-Time AES-GCM for Resource-Constrained Environments: By evaluating the FACE (Fixsliced AES-CTR) wrapper and comparing GHASH realizations, this work clarifies optimal design choices for secure AES-GCM implementations on the ARM Cortex-M4.

High-Degree Bitslicing Guidelines for GPUs: The study analyzes the relationship between bit-slice degree, register pressure, and warp occupancy on GPUs, providing practical guidelines for maximizing throughput in parallel environments.

Synthesized Design Guidelines: Finally, this dissertation synthesizes insights from all case studies to propose a set of general design guidelines for applying bitslicing to additional ciphers and emerging platforms.

Organization of the Dissertation: The remainder of this dissertation is organized as follows. Chapter 2 reviews the fundamental concepts of bitslicing and the target algorithms. Chapters 3, 4, and 5 present specific case studies on SPEEDY, AES-GCM, and PRESENT/GIFT, respectively. Chapter 6 conducts a comprehensive cross-platform comparison, and Chapter 7 concludes the work with directions for future research.

# Ⅱ. Related Work

## 2.1 Overview of Block Ciphers and Modes of Operation

### 2.1.1 The SPEEDY Block Cipher

SPEEDY is a family of ultra-low-latency block ciphers proposed by Leander et al. at CHES 2021 [35]. Related low-latency-oriented cipher families have also been proposed for performance-critical settings (e.g., QARMA), illustrating that latency-driven design goals often reshape implementation trade-offs [7]. Unlike traditional ciphers optimized for area or throughput in various environments, SPEEDY was specifically engineered for high-end CPU architectures to achieve single-cycle encryption in hardware. The cipher is denoted as SPEEDY-$r$-$6\ell$, where $6\ell$ represents the block and key sizes (both equal), and $r$ signifies the total number of rounds. We focus on the 192-bit instances ($\ell = 32$), specifically SPEEDY-$r$-192 with $r \in \{5,6,7\}$. The designers recommend $r = 6$ for 128-bit security and $r = 7$ for 192-bit security levels [35].

**State Representation** : SPEEDY represents its 192-bit internal state as 32 cells of 6 bits. We denote the state bits as $u[i,j]$, where $i \in 0,\cdots,31$ indexes the cell (row) and $j \in 0,\cdots,5$ indexes the bit position within the 6-bit cell. For a fixed j, the column $u[:,j]$ can be viewed as a 32-bit vector, which is well suited to word-level bitwise operations in software.

**Round Transformations. Each round consists of the following operations:**

One round of SPEEDY consists of five distinct transformations designed to minimize XOR depth and wire length:

SubBox ($SB$): A nonlinear layer applying a 6-bit S-box $S: \mathbb{F}_2^6 \to \mathbb{F}_2^6$ to each of the 32 rows independently. The S-box is realized as a two-level NAND tree, specifically optimized for CMOS latency rather than algebraic complexity [35]. A 6-bit S-box S is applied independently to each cell:

$$u[i,0..5] \leftarrow S(u[i,0..5]) \text{ for } i = 0..31.$$

ShiftColumns (SC): A bit-level permutation providing inter-row diffusion. For each bit position j, the corresponding 32-bit column is cyclically shifted by j positions:

$$v[i,j] = u[(i+j) \bmod 32,\ j]$$

MixColumns ($MC$): A linear diffusion layer acting independently on each of the six 32-bit columns. For each $j \in \{0,...,5\}$, the column vector $v_j = (v[0,j],...,v[31,j])^T \in \mathbb{F}_2^{32}$ is transformed via

$$w_j = M_{\alpha_{j+1}} \cdot v_j.$$

Here, $M$ is a $32 \times 32$ binary circulant matrix chosen for its high branch number (equal to 8), which ensures robust resistance against differential and linear cryptanalysis [35].

AddRoundKey ($ARK$) & AddRoundConstant ($ARC$): These operations XOR a 192-bit round key $k_r$ and a 192-bit round constant $c_r$ into the state. The round constants are derived from the binary expansion of $\pi - 3$ to ensure a "nothing-up-my-sleeve" design [35].

Round Structure and Software Challenges

For the first $r-1$ rounds, the transformations are applied in the following sequence:

$$\mathrm{SB} \rightarrow \mathrm{SC} \rightarrow \mathrm{MC} \rightarrow \mathrm{ARC} \rightarrow \mathrm{ARK}$$

The final round omits the linear layer and constant addition, concluding with an extra key addition to prevent the final S-box from being trivially inverted.

$$\mathrm{SB} \rightarrow \mathrm{SC} \rightarrow \mathrm{ARK}$$

Software Implications. SPEEDY's 6-bit cell structure and bit-oriented linear layer are hardware-friendly but create overhead in straightforward byte-oriented software implementations. In addition, LUT-based S-box realizations introduce secret-dependent memory accesses. These issues motivate constant-time bitsliced implementations, which are the focus of the case study in Chapter 3.

### 2.1.2 AES and GCM

The Advanced Encryption Standard (AES) is a symmetric-key block cipher based on a substitution-permutation network (SPN) structure, standardized by NIST as FIPS 197 [57, 64]. It operates on a fixed block size of 128 bits and supports multiple key lengths of 128, 192, and 256 bits, which require 10, 12, and 14 rounds, respectively. The internal 128-bit state is typically represented as a $4 \times 4$ matrix of bytes. Each round iterates a function consisting of four distinct algebraic transformations—SubBytes, ShiftRows, MixColumns, and AddRoundKey—with the final round omitting the MixColumns operation:

SubBytes ($SB$): A nonlinear byte−wise substitution that applies an 8−bit S−box to each byte of the state. The S−box is constructed by taking the multiplicative inverse in $GF(2^8)$ followed by an affine transformation over $GF(2)$, providing high nonlinearity and resistance to differential cryptanalysis.

ShiftRows ($SR$): A transposition step where the $i$−th row of the state matrix is cyclically shifted left by $i$ bytes ($i \in \{0,1,2,3\}$), ensuring inter−column diffusion.

MixColumns ($MC$): A linear diffusion layer where each column is treated as a polynomial over $GF(2^8)$ and multiplied by a fixed MDS (Maximum Distance Separable) matrix modulo $x^4+1$. This operation ensures that changes in a single byte propagate across the entire column.

AddRoundKey ($ARK$): A simple bitwise XOR operation between the state and the round key derived from the original cipher key via the AES key schedule.

Counter (CTR) mode is a representative mode of operation that effectively transforms a block cipher into a stream cipher. It offers significant advantages in high−performance environments because of its inherent parallelizability and the fact that it does not require padding. For each $i$−th plaintext block $P_i$, the corresponding keystream block $S_i$ is generated by encrypting an incrementing counter:

$$S_i = E_K(J \boxplus I)$$

where E_K denotes the AES encryption function, J is the initial counter block (nonce), and boxplus signifies addition modulo $2^{128}$. The ciphertext is then obtained via $C_i = P_i \oplus S_i$. Since each keystream block can be computed independently, CTR mode is ideally suited for architectures with SIMD or multi-threaded capabilities, such as modern CPUs and GPUs [36, 56].

**Galois/Counter Mode (GCM) :** GCM is an authenticated encryption with associated data (AEAD) mode that combines the confidentiality of AES-CTR with the integrity provided by a universal hash function, GHASH [42, 65]. GCM has been widely adopted in high-security protocols, including TLS 1.2/1.3, IPsec, and secure data logging frameworks [10, 43, 61].

The authentication mechanism in GCM is centered around the GHASH function, which operates in the binary finite field $GF(2^{128})$ defined by the irreducible polynomial[65]:

$$P(x) = x^{128} + x^7 + x^2 + x + 1$$

The process begins by deriving a hash subkey $H = E_K(0^{128})$ For a sequence of $n$ input blocks (comprising associated data $A$ and ciphertext $C$), the GHASH is computed recursively:

$$S_0 = 0,\ S_i = (S_{i-1} \oplus Y_i) \cdot H \in GF(2^{128})$$

where $Y_i$ represents the padded input blocks. The final authentication tag $T$ is then produced by encrypting the result of GHASH with a masked

counter block.

The primary challenge in implementing AES-GCM on resource-constrained platforms, such as the ARM Cortex-M4, is the high computational cost of $GF(2^{128})$ multiplication in the absence of hardware-accelerated carry-less multiplication instructions [32, 60]. Consequently, optimizing the performance and side-channel resilience of GHASH has remained a critical research objective in the field of embedded cryptography [29, 34, 49].

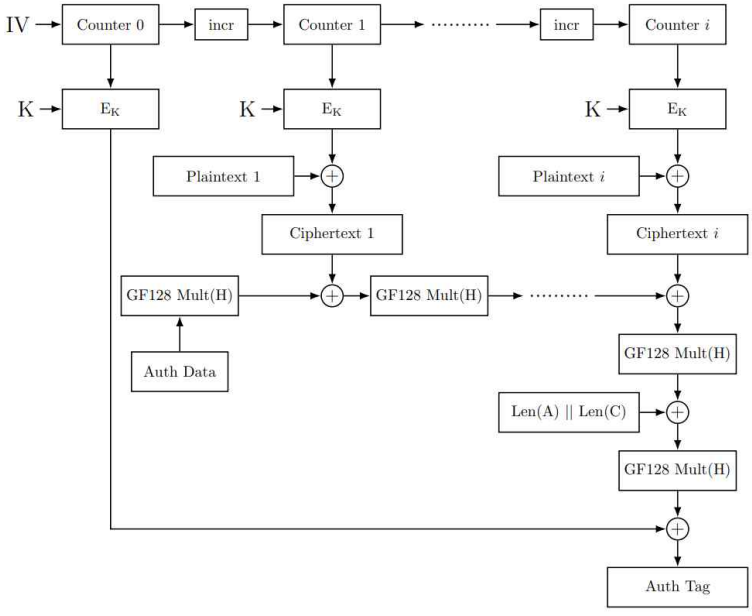


Figure 2-1. Schematic of AES-GCM encryption (AES-CTR + GHASH).

### 2.1.3 Lightweight Block Ciphers PRESENT and GIFT

PRESENT is an ultra-lightweight block cipher designed by Bogdanov, standardized under ISO/IEC 29192-2 for resource-constrained

environments such as RFID tags and sensor nodes [15]. It adopts a substitution-permutation network (SPN) structure with a 64-bit block size and supports key lengths of 80 and 128 bits. The cipher consists of 31 rounds, where each round incorporates three distinct layers:

**AddRoundKey:** A 64-bit round key is XORed with the current state.

**SubCells:** A nonlinear layer where the 64-bit state is divided into 16 nibbles, and a single 4-bit S-box is applied to each nibble in parallel. The S-box is chosen for its low hardware gate count and optimal resistance against linear and differential cryptanalysis.

**pLayer:** A bit-level permutation that provides diffusion by rearranging the 64 bits of the state according to a fixed pattern.

In hardware, the pLayer is "cost-free" as it is implemented solely through wiring. However, in traditional byte-oriented software, this bit-level shuffle is extremely inefficient, requiring numerous masking and shifting operations. This architectural mismatch was a primary motivator for the development of bitsliced software implementations, which can realize the pLayer using a sequence of word-level logical operations [45, 53].

GIFT was proposed at CHES 2017 as a refined successor to PRESENT, aiming to improve both security margins and implementation efficiency [8]. It features two main variants: GIFT-64 (64-bit block, 28 rounds) and GIFT-128 (128-bit block, 40 rounds), both utilizing a 128-bit key. GIFT revisits the SPN design of PRESENT to address vulnerabilities against linear hulls and differential characteristics that emerged after PRESENT's standardization. The overall structures of PRESENT and GIFT are illustrated in Figures 2-2 and 2-3, respectively.

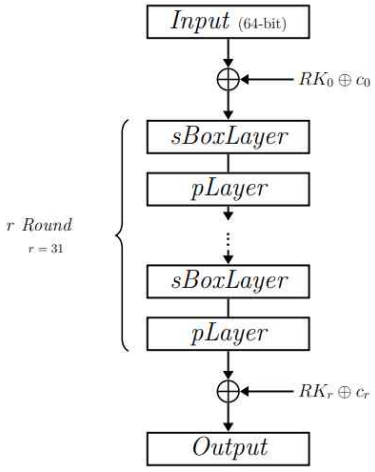


Figure 2-2. Overall structure of the PRESENT block cipher.

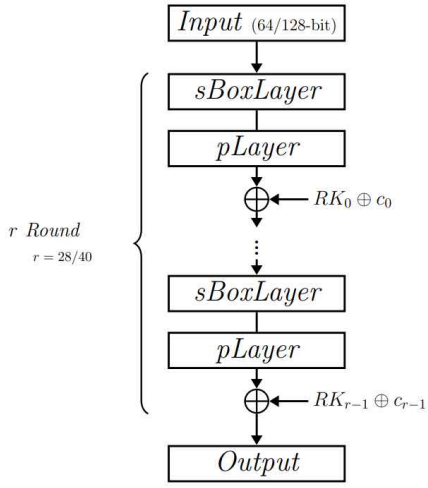


Figure 2-3. Overall structure of the GIFT block cipher.

The round function of GIFT is streamlined into three steps:

**SubCells:** Similar to PRESENT, it applies a 4-bit S-box to each nibble. However, GIFT's S-box and its placement are specifically tuned to provide higher security with fewer gates.

**PermBits:** A structured bit-permutation layer that provides diffusion without the need for XOR gates in hardware.

**AddRoundKey:** To further reduce hardware area, GIFT XORs round keys into only half of the state bits (two bits per nibble), a technique known as "partial key addition" [8].

Software Implementation Paradigms for Lightweight Ciphers

While PRESENT and GIFT are optimized for hardware, recent research has demonstrated their potential for high-throughput software execution.

On microcontrollers, Fixslicing techniques have been applied to GIFT to maintain constant-time behavior while achieving record-breaking speeds on ARM Cortex-M and RISC-V architectures [1, 2].

Furthermore, the regular SPN structure of these ciphers makes them exceptionally well-suited for massively parallel platforms. Kim et al. [30] demonstrated that by using high-degree bitslicing on GPUs, PRESENT and GIFT can achieve throughputs in the hundreds of Gbit/s. This makes them attractive candidates not only for IoT edge devices but also for high-performance server-side workloads such as bulk data encryption and exhaustive key searches [30, 62]. This cross-platform applicability is a central theme explored in the subsequent chapters of this dissertation.

## 2.2 Software Implementation Techniques and Side-Channel Risks

### 2.2.1 Byte-Oriented and T-Table-Based Implementations

The most common approach to implementing block ciphers in software is to represent nonlinear components (most notably S-boxes) as lookup tables (LUTs). In byte-oriented designs, the internal state is stored as bytes or words, and each round repeatedly applies table lookups and XORs. For AES, a widely used acceleration technique is the T-table (or "T-box") construction, in which several round transformations—SubBytes, ShiftRows, and MixColumns—are precomputed and merged into four 1 KB tables [29, 57]. This design trades arithmetic operations for memory accesses and can provide high throughput on desktop- and server-class CPUs where tables are served efficiently by large and fast caches.

However, the same table-centric approach faces two practical

limitations in the environments targeted in this dissertation. First, on embedded microcontrollers such as ARM Cortex-M3/M4, large tables are typically stored in flash memory and accessed over relatively constrained buses. Flash wait states and bus contention can therefore erode or even negate the expected performance gains of LUT-based implementations [60]. Second, for ciphers whose round structure is inherently bit-oriented (e.g., SPEEDY with 6-bit cells), byte-oriented table lookups introduce additional overhead: the implementation must repeatedly extract non-byte-aligned bit fields and reassemble outputs, which increases instruction count and memory traffic [35]. These limitations motivate table-free software techniques that emphasize word-level logical operations and architecture-aware state layouts, as discussed in the following sections.

Figure 2-2 previews a key building block used by many such table-free designs. The SWAPMOVE primitive [41] enables efficient bit transposition between layouts and is commonly used to convert between scalar and bitsliced representations, or to update bitsliced layouts with a small number of structured operations.

$$SWAPMOVE(A,B,M,n): \quad T=(B\oplus(A\ll n))\wedge M \quad B=B\oplus T \quad A=A\oplus(T\gg n)$$

Figure 2-2. The SWAPMOVE primitive [41] is commonly used to convert between bitsliced layouts, and the representation can be updated with only a few SWAPMOVE operations.

### 2.2.2 Constant-Time Implementations and Microarchitectural Timing Side Channels

Beyond performance considerations, a more fundamental drawback of table-based implementations is their susceptibility to timing

side-channel attacks. In typical LUT-based AES implementations, the memory address accessed by an S-box or T-table lookup depends on secret data (e.g., key-dependent intermediate state). As a result, the execution time may vary with the microarchitectural state of the processor, such as cache hits/misses, contention on shared interconnects, or other forms of resource-dependent latency.

A line of research has shown that such fine-grained timing variability can be exploited to recover secret keys. Bernstein [12] and Bonneau et al. [18] demonstrated key recovery from timing traces of AES, and subsequent work expanded the attack surface to include cache-collision and branch-prediction effects on contemporary systems [20]. Importantly, the risk is not limited to high-end CPUs with deep cache hierarchies. Even on the ARM Cortex-M4—often described as having minimal or no data cache—observable timing leakage can still arise due to bus interconnect contention and flash-access variability [54]. Consequently, constant-time programming has become a standard requirement for high-assurance cryptographic software. While this dissertation focuses on timing-uniform software, stronger physical leakage models (e.g., power/EM) are often addressed via masking, which has a well-established formal security foundation [50].

In this dissertation, "constant-time" is understood in the common implementation-oriented sense: (i) control flow does not depend on secret data (no secret-dependent branches or early exits), and (ii) memory access patterns do not depend on secret data (no key- or state-indexed table lookups) [29, 56]. Under these constraints, implementations typically replace LUT-based S-boxes with fixed Boolean networks realized by word-level bitwise operations, and they design linear layers to avoid data-dependent indexing. This naturally motivates bitsliced and Fixsliced paradigms, which eliminate table lookups

by construction and provide a principled way to balance performance, resource usage, and constant-time behavior across heterogeneous platforms. The next section introduces these paradigms and summarizes their advantages and practical trade-offs.

## 2.3 Bitslicing Paradigms

Bitslicing and Fixslicing are implementation paradigms that restructure a block cipher's internal state to better match the word-level execution model of software platforms. By representing state bits as "bit planes" stored in processor words, these approaches replace table lookups with Boolean circuits composed of word-level logical instructions. This design is particularly attractive when constant-time behavior is required, because it avoids secret-dependent memory accesses and data-dependent control flow. At the same time, the achievable performance depends strongly on platform constraints (e.g., register availability on microcontrollers and occupancy on GPUs), making the choice of representation and parallelism degree a key design decision.

### 2.3.1 Principles, Advantages, and Limitations of Bitslicing

Bitslicing, originally proposed by Biham to accelerate DES [13], represents cipher computations as word-level Boolean operations by "transposing" multiple independent blocks into a bitsliced layout. Bitsliced techniques have been explored even on 8-bit microcontrollers (e.g., AVR) for lightweight block ciphers, showing that transposition and register/memory constraints remain central across word sizes [9]. On a processor with a word size of w bits (e.g., w = 32 on a 32-bit microcontroller), bitslicing can process up to w blocks in parallel: each

word holds one specific bit position across w different blocks. For example, if a register holds the i−th bit of w blocks, then a single logical instruction (AND, XOR, OR, NOT) updates that bit position for all w blocks simultaneously.

This representation offers two major advantages:

**Constant−time security** : In bitsliced implementations, nonlinear S−boxes are realized as Boolean circuits rather than lookup tables. Since the computation is expressed using fixed sequences of register−only logical operations, the implementation avoids secret−dependent memory addresses and typically reduces timing variability associated with cache/bus effects in table−based designs [29, 52].

**High throughput via software−level parallelism** : When enough independent blocks are available, bitslicing maximizes ALU utilization by turning one logical instruction into many parallel bit operations. This can yield substantial speedups, especially on architectures where logical operations are fast and memory accesses are comparatively expensive.

However, bitslicing also introduces important limitations:

**Packing/unpacking overhead** : Input blocks must be converted from the conventional layout into the bitsliced layout (packing), and the result must be converted back (unpacking). These transposition steps can be non−trivial, and for short messages the conversion overhead may dominate overall runtime [52, 56].

**Register pressure and reduced flexibility** : A bitsliced state often requires many live registers (bit planes plus temporaries). On

register-constrained microcontrollers, this can limit the practical bitslicing degree (often to one or two blocks) unless careful scheduling and assembly-level tuning are used.

**Costly permutations in the bitsliced domain** : Operations that are "cheap" as wiring in hardware (e.g., ShiftRows- or pLayer-like permutations) can become expensive in software when they require bit rearrangement across planes. Efficient packing/permutation frequently relies on structured bit-transposition primitives (e.g., SWAPMOVE-style operations), but the optimal strategy is highly platform- and cipher-dependent [52, 56].

### 2.3.2 Overview of Fixslicing Pattern Design

Fixslicing was introduced by Adomnicai et al. to reduce the cost of bit permutations that are expensive in conventional bitsliced implementations [1, 2]. The central idea is to keep state bits in fixed register positions throughout the rounds, and to redesign the linear layer so that it implicitly accounts for the "virtual" movement of bytes/bits that would otherwise be implemented explicitly.

In a Fixsliced AES implementation, for example, ShiftRows is not executed as a separate permutation step. Instead, its effect is absorbed into the linear layer by using round-dependent variants of MixColumns (or equivalent diffusion transformations). By carefully choosing the state layout and the set of linear-layer variants, Fixslicing can eliminate explicit shuffle-like operations while preserving the cipher's overall algebraic effect. Prior work has shown that this approach can significantly reduce instruction count and improve performance on ARM Cortex-M and RISC-V platforms while maintaining strict constant-time properties [1,

2, 34].

From a design perspective, Fixslicing can be viewed as a pattern-based method: once a fixed state layout is selected, the implementer derives a small set of linear-layer patterns (variants) that collectively realize the intended round transformations without physically permuting the state. The trade-off is that Fixslicing may increase code complexity (and sometimes code size) because multiple diffusion variants must be maintained and applied in a round schedule. Nevertheless, on architectures where bit permutations are expensive relative to logical operations, Fixslicing is often the most effective way to achieve both performance and constant-time behavior.

### 2.3.3 High-Degree Bitslicing on GPUs

On massively parallel platforms such as GPUs, bitslicing can be applied at much higher degrees than on microcontrollers. While low-end embedded cores are often limited by register availability and must keep the parallelism degree small, GPUs provide a large register file per thread and execute many threads concurrently. As a result, GPU implementations can adopt "high-degree bitslicing," where each thread processes 32 or 64 blocks simultaneously in a bitsliced form [25, 30].

This approach has two practical benefits. First, it reduces reliance on lookup tables in global or shared memory, thereby avoiding memory bottlenecks and access-pattern sensitivity that can degrade throughput in table-based GPU implementations [36, 38, 44]. Second, it maps well to the GPU's SIMT execution model when the cipher is implemented with branch-free Boolean circuits, helping to avoid warp divergence.

At the same time, GPU optimization requires balancing bitslicing degree against register pressure and occupancy. Increasing the number of

parallel blocks per thread raises the number of live registers needed for state planes, keys, and intermediates; if register usage grows too large, occupancy may drop or the compiler may spill registers to local memory, harming performance. Consequently, high-degree bitslicing on GPUs is most effective when the degree is chosen to maximize arithmetic intensity without exceeding the register budget that sustains high occupancy.

## 2.4 Platform-Specific Implementation Characteristics

The efficiency of bitsliced and Fixsliced implementations is tightly coupled to platform microarchitecture. Although the same cipher can be expressed in a table-free, constant-time form across many targets, the optimal design choices—such as state representation, bit-slice degree, and scheduling of Boolean networks—depend on instruction-set features, register availability, and memory behavior. This section summarizes the characteristics of the three target platforms considered in this dissertation: ARM Cortex-M microcontrollers, RV32I-based RISC-V microcontrollers, and NVIDIA CUDA GPUs.

### 2.4.1 ARM Cortex-M Microcontrollers

ARM Cortex-M3 and Cortex-M4 are 32-bit embedded processors designed for low-power, real-time applications. Both belong to the ARMv7-M family and execute the Thumb-2 instruction set [5]. Cortex-M3 typically employs a 3-stage pipeline, whereas Cortex-M4 adds DSP-oriented instructions (and, in many devices, an optional single-precision FPU) to accelerate signal-processing workloads.

**Register file and register pressure** : The architecture exposes 16 32-bit registers (r0-r15). Among them, r15 is the program counter (PC), r14 is the link register (LR), and r13 is the stack pointer (SP). After reserving these special registers and accounting for calling conventions and loop/temporary usage, the number of registers practically available for cryptographic state and intermediates is limited (often around 13 or fewer). This constraint makes register pressure a primary bottleneck for bitsliced implementations: overly aggressive parallelism or large Boolean networks can force spills to the stack, increasing both runtime and timing variability.

**Barrel shifter and fused operations** : A key advantage of ARMv7-M is the integrated barrel shifter, which allows many data-processing instructions to apply shifts or rotates "for free" as part of the instruction encoding. For example, XOR with a rotated operand can be expressed as a single instruction (e.g., EOR dst, dst, src, ROR #imm). This feature is particularly useful in bitsliced linear layers and diffusion steps where repeated fixed rotations are required. ARMv7-M also provides practical bitfield instructions (e.g., UBFX/BFI) that can reduce packing/unpacking overhead when constructing bitsliced state layouts.

**Cryptography-specific limitations** : Cortex-M3/M4 do not provide AES-NI-like instructions or dedicated carry-less multiplication instructions for $GF(2^{128})$ arithmetic. Consequently, high-performance AES-GCM on these cores requires table-free Boolean S-boxes and carefully engineered GHASH multiplications using integer operations rather than hardware acceleration. From a constant-time perspective, avoiding lookup tables is especially important because flash/SRAM access latency and bus/interconnect effects can introduce observable timing

variation even on microcontrollers with minimal caching.

### 2.4.2 RV32I-Based RISC-V Microcontrollers

RISC-V is an open and modular ISA. In this dissertation, we focus on implementations targeting the RV32I base integer ISA [6, 63], which provides a compact and portable baseline for microcontroller-class designs.

**Register richness** : RV32I exposes 32 general-purpose integer registers (x0-x31), where x0 is hardwired to zero. Compared to Cortex-M, this larger register file provides more headroom for holding bitsliced state planes, round keys, and intermediate Boolean terms simultaneously. In practice, this can reduce stack traffic and improve performance stability by keeping more of the computation strictly register-resident.

**Rotation and bit-manipulation costs** : A notable limitation of RV32I (as used here) is the absence of dedicated rotate instructions and the lack of a barrel-shifter-style "free shift" within general ALU operations. Fixed rotations must therefore be synthesized using multiple instructions (typically shift-left, shift-right, and OR/XOR). As a result, linear layers that rely heavily on rotations can become significantly more expensive than on ARM Cortex-M. This makes state representation and the scheduling of rotations critical: designs that are rotation-heavy may require additional optimization or alternative formulations to remain competitive on RV32I.

**Memory behavior and constant-time design** : Similar to other low-end

microcontrollers, RV32I systems frequently execute code from flash and store data in SRAM, often with limited caching. Constant-time implementations should therefore avoid secret-dependent memory access patterns (e.g., S-box tables). Table-free Boolean implementations not only harden against timing leakage but also reduce sensitivity to variable-latency memory operations.

### 2.4.3 Overview of CUDA GPU Architecture

NVIDIA GPUs provide a fundamentally different execution environment, emphasizing massive parallelism through the SIMT (Single Instruction, Multiple Threads) model. Beyond symmetric primitives, GPUs have also been used to accelerate compute-intensive post-quantum schemes (e.g., LWE-based constructions), reinforcing their suitability for cryptographic workloads at scale [3]. Recent studies also report substantial speedups for PQC key encapsulation mechanisms on GPUs, such as FrodoKEM, NewHope, and Kyber, highlighting GPU applicability beyond symmetric encryption [24]. High-throughput GPU implementations have also been studied for post-quantum KEM workloads targeting IoT-scale deployments, indicating that GPU cryptography spans both symmetric and asymmetric domains [37]. In CUDA, threads are scheduled in groups called warps (32 threads) that execute in lockstep. This dissertation evaluates GPU implementations on an Ampere-class device (RTX 3060) [30].

**Warp-based execution and divergence** : Because threads in a warp share an execution path, conditional branches that depend on per-thread data can cause warp divergence, leading to serialized execution and throughput loss. For cryptographic kernels, this motivates branch-free

designs. Bitsliced implementations naturally align with this requirement because S-boxes and round transformations can be expressed as fixed Boolean circuits without data-dependent branching.

**Memory hierarchy and throughput considerations** : GPUs feature a hierarchy of global memory, shared memory, and constant memory. While global memory has high bandwidth, it also has high latency; performance relies on maintaining enough active warps (high occupancy) so that the hardware can hide latency by switching to other warps. Consequently, designs that reduce arithmetic intensity or create irregular memory accesses can suffer. Table-based implementations may introduce non-coalesced or variable-latency accesses, whereas table-free bitsliced Boolean circuits can keep most operations in registers.

**Register pressure and occupancy trade-offs** : Each Streaming Multiprocessor (SM) has a fixed register file shared among resident threads. High-degree bitslicing (e.g., 32 or 64 blocks per thread) increases the number of live registers required for state planes, key material, and intermediate values. If register usage per thread becomes too high, the compiler may spill registers to local memory (which maps to global memory) or the hardware may reduce occupancy, both of which can harm performance. Therefore, GPU bitsliced designs must balance bit-slice degree against register pressure to maximize overall throughput.

In summary, while microcontrollers are primarily constrained by a small register set and relatively expensive memory accesses, GPUs are constrained by warp execution behavior and the occupancy-register trade-off. These platform-specific factors explain why the same "bitsliced" paradigm results in different optimal configurations across Cortex-M, RV32I, and CUDA GPUs.

## 2.5 Summary of Related Prior Work

This section summarizes prior work that is most closely related to this dissertation, focusing on (i) bitsliced implementations of SPEEDY. (ii) constant-time AES-GCM implementations on embedded microcontrollers, (iii) GPU implementations of lightweight block ciphers such as PRESENT and GIFT. Beyond listing results, we emphasize recurring implementation themes—state representation, packing/unpacking, table-free S-box realization, and platform-dependent trade-offs—that directly motivate the research scope and evaluation methodology adopted in later chapters.

### 2.5.1 SPEEDY and Other Bit-Sliced Implementations

SPEEDY is a representative example of a cipher whose design is highly optimized for hardware latency yet poses challenges for conventional software. Its 192-bit state organized as 32×6-bit cells and its 6-bit S-box structure lead to substantial bit-extraction overhead in naïve byte-oriented software implementations [35]. This mismatch has motivated bitsliced implementations that reshape the state into bit planes stored in machine words, thereby enabling register-only Boolean evaluation of the S-box and efficient realization of the linear layer. Kim et al. demonstrated that SPEEDY can be implemented efficiently on 32-bit microcontrollers by adopting a 6×32 bitsliced representation and using SWAPMOVE-based packing/unpacking to bridge the algorithmic state structure and the processor word model [31, 33]. This line of work illustrates a general principle: for bit-oriented ciphers, the dominant

factor in software performance is often the state representation and the cost of transposition, rather than the round function alone.

More broadly, bitslicing has been applied to a wide range of block ciphers to achieve table-free, constant-time behavior and high throughput. For AES, bitsliced and Fixsliced designs provide speed records on constrained microcontrollers by eliminating table lookups and by restructuring permutations within the linear layer [2, 56]. For lightweight ciphers, PRESENT and GIFT have also seen substantial gains from bitslicing and Fixslicing because their S-boxes and permutations map naturally to Boolean networks and structured bit-move primitives [1, 53]. PRINCE is a representative example of a low-latency block cipher whose design goals emphasize fast round evaluation, making implementation-oriented analysis particularly relevant [17]. Follow-up designs such as PRINCEv2 refine low-latency block cipher families with improved security margins while preserving the implementation-driven design philosophy [16]. Beyond the ciphers studied in this dissertation, lightweight families such as SKINNY (and low-latency variants such as MANTIS) further motivate reusable implementation patterns for table-free designs [11]. These results collectively suggest that bitslicing is not merely an "optimization trick," but a cross-cutting implementation paradigm whose effectiveness depends on (i) how state bits are mapped to registers, (ii) how permutations are realized or absorbed, and (iii) how platform instruction-set features (e.g., rotation support, register file size) shape the achievable trade-offs.

Research gap and positioning. While the above studies provide strong point solutions, three recurring gaps motivate this dissertation's scope. First, optimization efforts remain fragmented across cipher-platform pairs, making it difficult to extract reusable design rules that apply across heterogeneous targets. Second, cross-platform comparisons are often

performed with non-unified metrics or under different assumptions (e.g., short vs. long messages, encryption-only vs. AEAD, bulk encryption vs. key search), limiting interpretability. Third, constant-time requirements are sometimes treated as an optional constraint rather than an explicit axis of the design space, even though table-free designs can impose non-trivial performance and resource trade-offs. This dissertation addresses these gaps by applying bitslicing and Fixslicing consistently across embedded microcontrollers and GPUs, evaluating multiple ciphers under unified metrics, and synthesizing the observed trade-offs into practical implementation guidelines in later chapters.

### 2.5.2 AES-GCM Implementation Studies

A large body of work has established that table-based AES and AES-GCM implementations are vulnerable to microarchitectural timing side channels due to secret-dependent memory accesses, motivating constant-time implementations that avoid lookup tables [12, 18, 20, 29]. A seminal milestone is the work of Käsper and Schwabe [29], who showed that a bitsliced AES-GCM design on general-purpose CPUs can achieve both high performance and resistance to cache-timing attacks by expressing the AES S-box and linear layers as Boolean circuits over machine words. On modern x86 CPUs, packed AES-GCM constructions have been proposed to exploit dedicated AES and carry-less multiplication instructions for higher throughput, providing a complementary design point to embedded constant-time efforts [28]. On server-class x86 CPUs, AES-NI provides dedicated instructions for AES rounds, substantially reducing the need for large lookup tables and changing the performance/security trade-offs compared to microcontrollers [55].

In the embedded domain, constant-time AES (and AES-CTR) on ARM Cortex-M microcontrollers has been steadily optimized by adapting bitslicing to tight register and memory constraints. Schwabe and Stoffelen [56] demonstrated that a carefully engineered multi-block bitsliced approach can achieve strong performance on Cortex-M3/M4 while avoiding table lookups. Adomnicai and Peyrin subsequently introduced Fixslicing for AES-like ciphers [2], where the cost of expensive bit permutations (e.g., ShiftRows in a bitsliced setting) is reduced by adopting a fixed state layout and using round-dependent linear-layer variants. These studies collectively show that, on microcontrollers without AES-NI-like acceleration, table-free Boolean implementations can be competitive with (and often superior to) LUT-based implementations, while providing substantially improved timing robustness.

For AES-GCM specifically, prior work has emphasized that optimizing the AES-CTR core alone is insufficient, because GHASH can dominate total runtime on platforms that lack carry-less multiplication instructions. Multiple GHASH strategies have been explored: compact table-based approaches (e.g., 4-bit tables) that trade strict constant-time behavior for speed, and multiplication-based approaches such as Karatsuba decompositions that avoid secret-dependent table indices and are designed to be constant-time [48, 49]. Practical embedded frameworks (e.g., PAGE) also explore design points that combine performance-oriented engineering with realistic microcontroller constraints [32, 60]. In addition, FACE (Fast AES-CTR Encryption) exploits redundancy between consecutive counter blocks by caching intermediate AES state and reusing it across blocks [46]. While FACE is conceptually general, integrating it efficiently into a Fixsliced AES core requires redesign at the representation level, because Fixslicing stores

state bits across interleaved lanes rather than in a conventional byte-oriented layout.

Taken together, the literature provides strong building blocks—fast constant-time Fixsliced AES cores [2, 56], acceleration techniques for CTR mode such as FACE [46], and multiple GHASH realizations spanning table-based and table-free designs [32, 48, 49, 60]. However, relatively few studies evaluate these components under a unified framework on low-end microcontrollers, explicitly quantifying how the combined design choices affect cycles-per-byte, memory footprint, and constant-time integrity at the full AES-GCM level. This motivates the integrated design-space exploration carried out in Chapter 4.

### 2.5.3 GPU Implementations of PRESENT and GIFT

GPU acceleration of block ciphers has been studied extensively, particularly for AES and ARX-based designs, with early approaches often relying on lookup tables stored in global/texture/shared memory and focusing on memory coalescing and cache behavior to maximize throughput [36, 39]. These LUT-based implementations can achieve high raw speed, but they may exhibit data-dependent memory access patterns that are undesirable when timing uniformity is a concern, especially in shared or multi-tenant environments [38, 44]. As a result, bitsliced GPU implementations have been explored as an alternative, most notably for AES, where Boolean-circuit S-boxes and table-free round functions reduce sensitivity to memory behavior and can better align with SIMD/SIMT execution [25, 44]. GPU-oriented optimization has also been explored for lightweight AEAD and hashing primitives (e.g., Gimli), further motivating computation-centric, table-free design choices on SIMT devices [26].

For lightweight SPN ciphers such as PRESENT and GIFT, the case for bitslicing is particularly strong: their S-box layers are small (4-bit) and their diffusion is dominated by bit permutations (pLayer/PermBits), which are "free" in hardware but expensive in conventional software. In GPU settings, achieving high performance requires mapping these bit-oriented steps onto word-level operations while respecting the constraints of SIMT execution. Prior studies have shown that high-degree bitslicing can achieve very high throughput for lightweight ciphers on GPUs, demonstrating applicability to both bulk encryption and cryptanalytic workloads [30, 62].

Nevertheless, several limitations remain in the GPU-focused literature for PRESENT/GIFT. First, many implementations still rely on LUTs for convenience, which can introduce memory bottlenecks (non-coalesced accesses, cache effects) and complicate timing behavior. Second, the trade-off between bit-slice degree and GPU resource constraints is often under-specified: increasing per-thread parallelism improves arithmetic intensity but increases register pressure, potentially reducing occupancy or causing spills that degrade performance. Third, exhaustive key search imposes distinct design requirements (e.g., on-the-fly counter/key generation in bitsliced form, minimizing host-device transfers, avoiding early-exit divergence), which are frequently not addressed in bulk-encryption-focused studies.

These observations motivate the methodology in Chapter 5, which treats GPU optimization as a joint problem of (i) selecting a high-degree bitsliced representation, (ii) designing table-free S-box and permutation layers that avoid warp divergence, and (iii) balancing register pressure against occupancy while supporting both bulk encryption and exhaustive search workloads within a unified kernel structure.

# Ⅲ. Embedded Implementations of the SPEEDY Block Cipher

## 3.1 Problem Definition and System Model

### 3.1.1 Structural Inefficiency of SPEEDY in Software

SPEEDY is an ultra-low-latency block cipher family introduced at CHES 2021, originally engineered to minimize gate delay in hardware-oriented secure processor architectures [35]. In this dissertation we focus on the 192-bit instance, whose internal state is organized as 32 rows of 6-bit cells (i.e., a 32×6-bit structure). While this representation is highly suitable for hardware, it creates a structural mismatch in software environments, where computation is naturally expressed over word-sized operands (e.g., 32-bit registers on embedded microcontrollers). This mismatch leads to two major inefficiencies in conventional software implementations.

**Bit-manipulation and format-conversion overhead** : A straightforward C implementation typically stores the 192-bit state in a byte-oriented buffer (e.g., a 24-byte array). Because the algorithm's basic unit is a 6-bit cell, each SubBox invocation requires extracting 6-bit inputs from non-aligned bit positions and reinserting 6-bit outputs back into the packed state. This process repeatedly performs masking, shifting, and cross-byte composition, and it frequently touches memory during the round function. On 32-bit microcontrollers, where cycles are scarce and memory access can be relatively expensive, these bit-extraction and

reconstruction steps can dominate the overall runtime.

**Timing side-channel exposure from LUT-style S-box realizations** : To reduce the cost of evaluating a 6-bit S-box, many baseline software designs employ lookup tables (LUTs). However, LUT-based S-box evaluation involves data-dependent memory addresses derived from secret-dependent intermediate values. On systems with caches or non-trivial memory/bus behavior, such data-dependent access patterns can introduce measurable timing variation and thereby enable timing side-channel attacks [12, 14, 18, 20]. Even on microcontroller-class platforms, bus contention and memory-system effects can still make table lookups undesirable from a constant-time standpoint.

These observations motivate table-free, register-centric implementations that align SPEEDY's 32×6-bit state with the processor word model. In particular, a bitsliced formulation can map the state to bit planes and evaluate the S-box as a fixed Boolean network, simultaneously reducing format-conversion overhead and enabling constant-time execution [31, 33].

### 3.1.2 Research Objectives and System Model

Motivated by the software inefficiencies and timing-leakage risks discussed in Section 3.1.1, this chapter formalizes the implementation problem and specifies the evaluation setting. Our goal is to construct high-performance and timing-robust SPEEDY implementations on 32-bit embedded processors by adopting a state representation and instruction schedule that align with word-level execution.

1) Research objectives. We pursue the following three objectives.

**Performance objective :** We aim to achieve encryption performance (reported in cycles per byte, cpb) on 32-bit microcontrollers that is competitive with established constant-time software implementations of widely deployed block ciphers (e.g., AES-128 and GIFT-128) on the same class of platforms. Because SPEEDY's state naturally consists of 32 parallel 6-bit cells, our design specifically targets efficient evaluation of the 32 S-box instances and diffusion steps using 32-bit word operations.

**Security objective under a constant-time requirement :** We target a strict constant-time programming model for the block-cipher core. Concretely, the implementation must avoid (i) secret-dependent table lookups and other secret-indexed memory accesses, and (ii) secret-dependent branching. The round function is therefore expressed as a fixed sequence of register-level Boolean operations (AND/XOR/NOT/OR) and deterministic rotations/shifts, so that the instruction trace and memory-access pattern do not depend on secret data.

**Platform-aware optimization objective :** We seek to quantify how instruction-set features and register file constraints shape the achievable performance. In particular, we analyze how ARM's rotate/shift capabilities and fusion opportunities (e.g., rotate-with-ALU forms) compare against RV32I's simpler base ISA where rotations must be synthesized from shifts and Boolean operations. The chapter's implementation strategy is thus designed to be structurally identical across platforms while allowing low-level tuning where the ISA permits.

2) System model and scope. We consider the encryption of a 192-bit plaintext block under a fixed secret key, where round keys are assumed to be derived once per session (as in typical mode-of-operation use) and are then consumed by the encryption routine. Our focus is the steady-state per-block cost of the encryption function, including the cost of mapping the external input representation to the internal bitsliced state (packing) and returning the output to the conventional representation (unpacking), because these costs materially affect performance on embedded targets.

3) Threat model (timing side channels). We assume an adversary who can obtain fine-grained timing observations of many encryptions (e.g., via repeated queries and high-resolution timers, or via co-resident activity that amplifies latency variations on shared buses/memory systems). Under this model, table-based S-box evaluation and secret-dependent branches are considered unacceptable. Accordingly, the implementations in this chapter are designed to be table-free and branch-free with respect to secrets.

Target platforms and measurement methodology. We target two representative 32-bit embedded platforms:

ARM Cortex-M3 (ARMv7-M class). This platform provides a 32-bit datapath and a compact register set. In hand-written assembly, the link register (LR) can be saved at function entry and subsequently used as a temporary register if needed, but the practical number of simultaneously available temporaries is still limited; minimizing register pressure is therefore a key design constraint. The presence of efficient shift/rotate forms is advantageous for SPEEDY's diffusion steps.

**RV32I-based RISC-V**. This platform exposes a larger architectural register file but lacks dedicated rotate instructions in the RV32I base ISA, requiring rotations to be implemented using shift-and-merge sequences. This difference provides a clean setting to evaluate how rotation-heavy diffusion layers behave when mapped to a minimal load-store ISA.

Cycle counts are obtained using on-chip cycle counters on each platform and are reported as cycles per byte (cpb) to enable direct comparison across ciphers and implementations. In addition to performance, we qualitatively assess constant-time integrity by confirming the absence of secret-dependent memory access and secret-dependent control flow in the encryption routine.

## 3.2 Related Work

This section reviews prior work that directly informs the embedded implementation of SPEEDY in this chapter. We focus on three threads: (i) software implementations of SPEEDY and their performance bottlenecks, (ii) bitsliced and Fixsliced implementations of AES and lightweight ciphers on constrained platforms, and (iii) platform-dependent optimization techniques on ARM Cortex-M and RV32I RISC-V that shape the achievable design space.

### 3.2.1 Software Implementations of SPEEDY

The SPEEDY design paper and subsequent studies provide baseline

software implementations that highlight a core challenge: SPEEDY's 192-bit state is naturally organized as 32 rows of 6-bit cells, which does not align with byte- or word-oriented software representations [35]. A straightforward reference implementation typically stores the 192-bit state in a byte array (e.g., 24 bytes) and evaluates the 6-bit S-box using table lookups. Although this approach is simple and faithful to the specification, it introduces significant overhead in software because each S-box application requires repeated bit extraction and reinsertion across non-aligned bit positions, combined with frequent masking, shifting, and memory traffic.

To reduce bit-extraction overhead, an alternative "6×32-bit-style" representation has been explored, in which 6-bit values are stored in 8-bit locations. This reduces some conversion cost but expands the effective state footprint (representing 192 bits with a 256-bit container) and still incurs substantial bit-move overhead due to misalignment and frequent rearrangement operations. As a result, the improvement over the byte-array reference is limited when compared to more structural approaches.

A more decisive direction is bitslicing. Kim et al. proposed a bitsliced C implementation for SPEEDY-7-192 on an Intel Core i7-8850H, showing that reinterpreting SPEEDY's 32×6 structure as bit planes and evaluating the S-box as a fixed Boolean network yields substantial speedups over both the byte-array reference and the 6×32-bit-style implementation [31, 33]. Table 3-1 summarizes these results, indicating that SPEEDY becomes significantly more software-friendly when expressed in a bitsliced representation.

However, existing SPEEDY-oriented software studies primarily demonstrate feasibility and speed on desktop-class CPUs or in portable C, and they do not fully address the constraints that dominate on 32-bit

microcontrollers—tight register budgets, the high cost of stack spills, and instruction-set-specific opportunities such as rotation-XOR fusion. This chapter builds on the bitslicing insight but targets the embedded setting explicitly, with an emphasis on assembly-level structure and platform-dependent trade-offs.

| Intel 8th Core i7-8850H | Speed (cpb) |
|---|---|
| SPEEDY-7-192 encryption reference | 2983 |
| $6 \times 32$ reference | 1278 |
| bitslice(our) | 852 |

Table 3-1 SPEEDY-7-192 encryption speed on Intel Core i7-8850H (cpb).

### 3.2.2 Bit-Sliced Implementations of Lightweight Block Ciphers

Bitslicing and Fixslicing have been extensively studied as constant-time implementation paradigms for AES and lightweight block ciphers on constrained platforms. On ARM Cortex-M microcontrollers, bitsliced AES and its refinements demonstrate that table-free Boolean S-box realizations can achieve competitive cycles-per-byte (cpb) while removing secret-dependent memory accesses [56]. Fixslicing further improves efficiency by adopting a fixed state layout and absorbing permutation costs into round-dependent linear-layer variants, leading to speed records on ARM Cortex-M and RISC-V targets for AES-like and lightweight SPN ciphers [1, 2]. Similar trends appear in optimized implementations of lightweight ciphers such as GIFT, where the combination of table-free S-box circuits and structured bit permutations maps naturally to word-level operations and supports strict constant-time behavior [1, 2].

These studies provide two implementation lessons that are directly

relevant to SPEEDY on microcontrollers. First, the dominant performance factor is often the choice of state representation and the cost of transposition (packing/unpacking), rather than the round function alone. Second, constant-time behavior is most naturally achieved when S-boxes are expressed as Boolean circuits over registers, avoiding LUT-based designs that introduce data-dependent memory access patterns.

Nevertheless, SPEEDY differs from many lightweight SPN ciphers in two important ways: (i) it uses a 6-bit S-box (rather than 4-bit or 8-bit), and (ii) its diffusion is rotation/permutation heavy in a 32×6 organization. Consequently, applying "generic" bitslicing patterns requires SPEEDY-specific engineering—especially for packing/unpacking and for realizing the linear layer efficiently using the target ISA. This motivates the SPEEDY-focused state layout and rotation-centric diffusion strategy introduced in Section 3.3.

### 3.2.3 Block-Cipher Optimization on Cortex-M3 and RISC-V

A large portion of embedded cryptographic performance is determined not only by algorithmic structure but also by ISA-level features and register constraints. On ARM Cortex-M (ARMv7-M class), two aspects are particularly relevant. First, the barrel shifter enables many instructions to combine a shift/rotation with an ALU operation, which can reduce instruction count in diffusion layers that require multiple fixed rotations (e.g., implementing XORs of rotated planes). Second, ARM provides useful Boolean instruction forms (e.g., BIC/ORN-style patterns depending on the core and assembler support), which can lower the cost of Boolean S-box networks by reducing explicit NOT and temporary-register usage. These advantages are counterbalanced by

limited general-purpose register availability in practice, making careful register allocation and the avoidance of stack spills critical for sustaining throughput.

In contrast, RV32I-based RISC-V targets offer a larger architectural register file, which can reduce memory traffic by keeping state and temporaries in registers. However, the RV32I base ISA does not provide a dedicated rotate instruction; rotations must be synthesized from shifts and Boolean operations, which increases the instruction count for rotation-heavy diffusion layers. As a result, implementations that are rotation-dominant may exhibit a different performance profile on RV32I than on Cortex-M, even when the high-level algorithmic structure is identical.

Prior work on AES and lightweight ciphers has repeatedly shown that these ISA characteristics materially influence constant-time software performance on both Cortex-M and RISC-V, and that the best designs explicitly exploit platform features while remaining table-free [1, 2, 56]. However, for SPEEDY specifically, there remains limited analysis that isolates how its 32×6 organization interacts with (i) Cortex-M's rotation-friendly instruction forms under tight register pressure and (ii) RV32I's rotation synthesis overhead under a larger register file.

This chapter addresses this gap by implementing SPEEDY with a shared bitsliced design principle across both platforms while applying ISA-aware tuning at the instruction-scheduling level. The resulting comparison clarifies which costs are representation-driven (packing/unpacking), which are cipher-driven (S-box and diffusion structure), and which are platform-driven (rotation support and register pressure), setting up the proposed techniques in Section 3.3 and the evaluation in Section 3.3.6.

## 3.3 Proposed Technique

### 3.3.1 6×32 Bit-Sliced State Representation and Packing

In the bitsliced representation, the 192-bit state is stored across six 32-bit registers. Each register corresponds to a bit plane that holds one bit position of many 6-bit chunks, enabling 32 parallel S-box inputs of 6 bits each.

Because SPEEDY structures the state as 32 × 6-bit blocks, simple SWAPMOVE operations are insufficient to construct the desired bitsliced layout. Figures 3-1-3-3 visualize this three-step packing procedure, from the initial 6×32 state layout through the rearrangement of 6-bit blocks and the final SWAPMOVE operations that produce the desired bitsliced representation.

| | $Block0$ | $Block1$ | $Block2$ | $Block3$ | $\cdots$ | $Block28$ | $Block29$ | $Block30$ | $Block31$ |
|---|---|---|---|---|---|---|---|---|---|
| $R_0$ | $b_0^0$ | $b_0^1$ | $b_0^2$ | $b_0^3$ | $\cdots$ | $b_0^{28}$ | $b_0^{29}$ | $b_0^{30}$ | $b_0^{31}$ |
| $R_1$ | $b_1^0$ | $b_1^1$ | $b_1^2$ | $b_1^3$ | $\cdots$ | $b_1^{28}$ | $b_1^{29}$ | $b_1^{30}$ | $b_1^{31}$ |
| $R_2$ | $b_2^0$ | $b_2^1$ | $b_2^2$ | $b_2^3$ | $\cdots$ | $b_2^{28}$ | $b_2^{29}$ | $b_2^{30}$ | $b_2^{31}$ |
| $R_3$ | $b_3^0$ | $b_3^1$ | $b_3^2$ | $b_3^3$ | $\cdots$ | $b_3^{28}$ | $b_3^{29}$ | $b_3^{30}$ | $b_3^{31}$ |
| $R_4$ | $b_4^0$ | $b_4^1$ | $b_4^2$ | $b_4^3$ | $\cdots$ | $b_4^{28}$ | $b_4^{29}$ | $b_4^{30}$ | $b_4^{31}$ |
| $R_5$ | $b_5^0$ | $b_5^1$ | $b_5^2$ | $b_5^3$ | $\cdots$ | $b_5^{28}$ | $b_5^{29}$ | $b_5^{30}$ | $b_5^{31}$ |

Figure 3-1. Initial 6×32 state layout used in the packing procedure : the 192-bit SPEEDY state is loaded into six 32-bit registers $(R_0,\ldots,R_5)$. Column $i$ corresponds to the $i$-th 6-bit cell ($i = 0,\ldots,31$), and the entry $(b_j^i)$ denotes bit $j$ ($j = 0,\ldots,5$) of that cell stored in register $R_j$ (bit-plane form).

Step 0: Sequentially load the 192-bit plaintext into six 32-bit registers, yielding some 6-bit blocks split across two registers.

Step 1: Apply SWAPMOVE operations so that each register uses only

30 bits to store five intact 6−bit blocks (indices 0-29), leaving 2 free bits per register.

Step 2: Place the remaining 12 bits corresponding to the last two blocks (indices 30 and 31) into the free bit positions using UBFX/BFI/LSL/LSR on Cortex−M3 and SRLI/SLLI/AND on RV32I.

Step 3: Apply the SWAPMOVE patterns of Figure 2 to exchange bit positions across registers, completing the 6×32 bitsliced representation (35 SWAPMOVE operations in total).

Unpacking for decryption simply applies these steps in reverse to recover the 192−bit block representation.

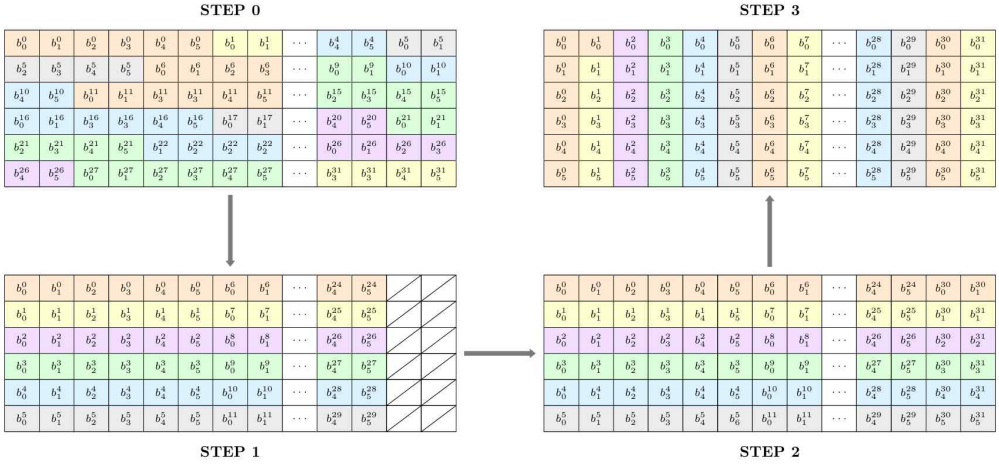


Figure 3−2. Plaintext made up of 32 six−bit blocks stored in six 32−bit registers is rearranged into a bitsliced form. Each 6−bit block is denoted $b_j^i$, where $i$ indexes the block and $j$ indicates the bit position.

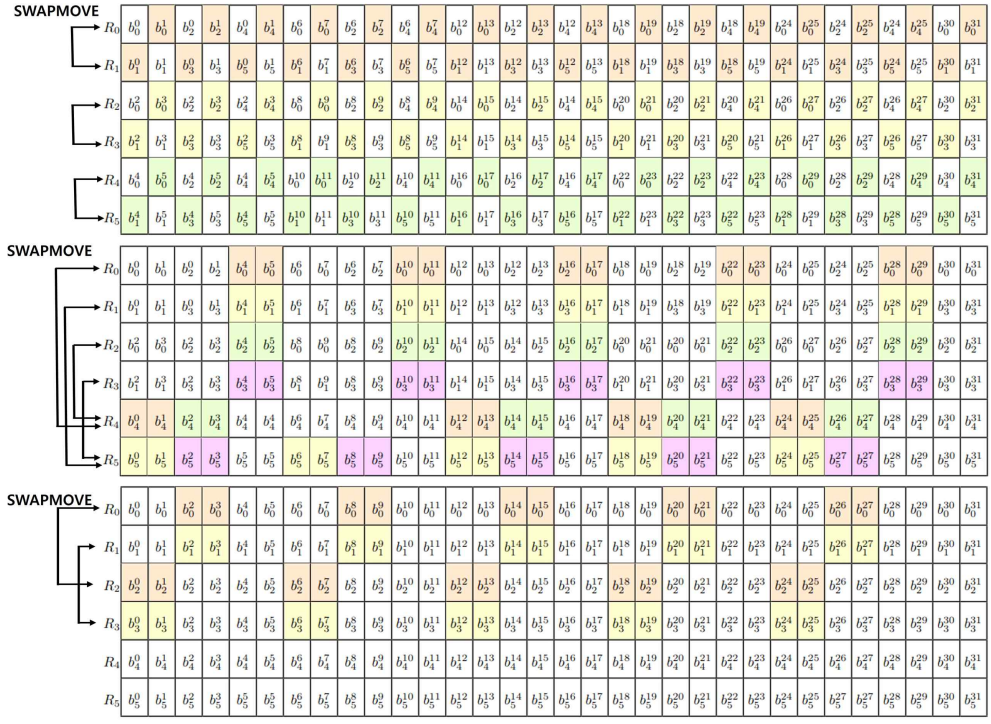


Figure 3-3. SWAPMOVE operation used as the final step of our method to obtain a bitsliced representation of SPEEDY. The bits in the highlighted regions of the two registers are exchanged. Each 6-bit block is denoted $b_j^i$, where $i$ indexes the block and $j$ indicates the bit position.

### 3.3.2 Bitwise SubBox Implementation

Since LUT-based S-boxes are unsuitable in a bitsliced setting, SPEEDY's 6-bit S-box is expressed in disjunctive normal form (DNF) and evaluated directly on the bit planes. Starting from the DNF given in the design paper, Kim et al. [31] reorganize and factor terms to reduce the number of AND/OR/NOT operations.

$$y_0 = (x_3 \wedge (\neg x_5 \vee (x_4 \wedge x_2))) \vee (x_1 \wedge ((\neg x_3 \wedge x_0) \vee (x_5 \wedge x_4)))$$
$$y_1 = (x_5 \wedge ((x_3 \wedge \neg x_2) \vee (x_2 \wedge x_0))) \vee (\neg x_5 \wedge x_3 \vee \neg x_4) \vee (\neg x_3 \wedge \neg x_0 \wedge x_1)$$
$$y_2 = (x_0 \wedge ((\neg x_3 \wedge x_4) \vee (x_3 \wedge x_1))) \vee (\neg x_3 \wedge \neg x_4 \wedge x_2) \vee (\neg x_0 \wedge \neg x_2 \wedge x_5)$$
$$y_3 = (x_2 \wedge ((\neg x_0 \wedge \neg x_3) \vee (x_0 \wedge x_4))) \vee (x_0 \wedge \neg x_2 \wedge x_5) \vee (\neg x_0 \wedge x_3 \wedge x_1)$$
$$y_4 = (x_0 \wedge \neg x_3) \vee (\neg x_0 \wedge x_4 \wedge x_5) \vee ((\neg x_4 \wedge \neg x_2) \wedge (x_0 \vee x_1))$$
$$y_5 = (x_2 \wedge (x_5 \vee (x_1 \wedge x_0))) \vee (\neg x_1 \wedge ((\neg x_2 \wedge x_4) \vee (x_0 \wedge x_3)))$$

On Cortex-M3, the ORN (OR-with-NOT) instruction is used to implement patterns such as ¬x and a ∧ ¬b efficiently, reducing the SubBox instruction count by about eight compared to a naive DNF realization. The ARMv6 assembly in Algorithm 1 maps input planes X0–X5 to output planes Y0–Y5 using only bitwise logical operations. On RV32I, the same Boolean expressions are implemented using XOR, AND, and OR, since ORN is not available. In both cases, SubBox contains no table lookups, eliminating cache-based timing leakage. Figure 3-4 shows the resulting ARMv6 assembly realization of the SubBox network, mapping the six input bit planes X0–X5 to the output planes Y0–Y5 using only bitwise logical operations.

```
Input: X0-X5 (r4-r9),
       temporal register T (r14)
Output: Y0-Y5 (r1-r3, r10-r12)

1: AND Y3, X2, X4
2: ORN Y3, Y3, X5
3: AND Y3, Y3, X3
4: AND Y4, X5, X4
5: ORN Y5, X3, X0
6: ORN Y4, Y4, Y5
7: AND Y4, X1, Y4
8: ORR Y0, Y4, Y3

9: AND Y3, X0, X2
10: ORN Y4, X2, X3
11: ORN Y3, Y3, Y4
12: AND Y3, Y3, X5
13: ORR Y4, X5, X4
14: ORN Y4, Y4, X3
15: ORN Y3, Y3, Y4
16: ORR Y4, X0, X3
17: ORN Y4, Y4, X1
18: ORN Y1, Y3, Y4

19: AND Y3, X1, X3
20: ORN Y4, X3, X4
21: ORN Y3, Y3, Y4
22: AND Y3, X0, Y3
23: ORR Y4, X3, X4
24: ORN Y4, Y4, X2
25: ORN Y3, Y3, Y4
26: ORR Y4, X0, X2
27: ORR Y4, Y4, X5
28: ORN Y2, Y3, Y4
29: AND Y3, X0, X4
30: ORR Y4, X0, X3
31: ORN Y3, Y3, Y4
32: AND Y3, Y3, X2
33: AND Y4, X0, X5
34: ORN Y4, X2, Y4
35: ORN Y3, Y3, Y4
36: AND Y4, X1, X3
37: ORN Y4, X0, Y4
38: ORN Y3, Y3, Y4

39: MOV T, #s0
40: ORR Y4, X4, X2
41: ORR Y5, X0, X1
42: ORN Y4, Y4, Y5
43: ORN Y4, T, Y4
44: ORN Y5, X3, X0
45: ORN Y4, Y4, Y5
46: AND Y5, X4, X5
47: ORN Y5, X0, Y5
48: ORN Y4, Y4, Y5

49: AND T, X0, X3
50: ORN Y5, X2, X4
51: ORN T, T, Y5
52: ORN T, X1, T
53: AND Y5, X1, X0
54: ORR Y5, Y5, X5
55: AND Y5, Y5, X2
56: ORN Y5, Y5, T
```

Figure 3-4. ARMv6 assembly realizations of the S-box using a bitsliced representation.

### 3.3.3 Rotation-Based ShiftColumns

In SPEEDY, ShiftColumns (SC) cyclically shifts each column j by j positions along the 32-row dimension:

$$v[i, j] = u[(i + j) \bmod 32, j].$$

Under the 6×32 bitsliced representation, each column j is stored as a 32−bit word S_j, where bit i corresponds to row i. Therefore, the above mapping is naturally realized as a word rotation:

$$S'_j = ROR_j(S_j), \text{ for } j = 0, \cdots, 5,$$

where ROR_j denotes a 32−bit right rotation by j positions. This realization avoids explicit bit swaps and replaces the original row−wise permutation with a small number of register−only rotate operations.

**On ARM Cortex−M3 (ARMv7−M)**, the barrel shifter enables a single−instruction realization per column using Thumb−2 forms such as MOV (or ROR) with an immediate rotate. The full ShiftColumns step thus requires one move for j = 0 (no rotation) and five rotate moves for j = 1, ⋯, 5, totaling six instructions in our assembly routine. The transformation touches only registers and contains no data−dependent memory access.

**On RV32I**, there is no dedicated rotate instruction. We synthesize ROR_j(x) using shift−and−merge operations of the form (x >> j) | (x << (32 − j)). In our implementation, the five non−trivial rotations (j = 1, ⋯, 5) are realized using fixed SRLI/SLLI and merge operations, leading to 19 instructions in total for ShiftColumns when accounting for the necessary register moves. Despite the increased instruction count, the computation remains register−only and constant−time.

Figures 3−5 and 3−6 illustrate the bitsliced and assembly−level realizations of ShiftColumns, respectively, highlighting that SC reduces to a deterministic set of word rotations in the bitsliced domain.

```
Input: state[0-5]
Output: state[0-5]
1: state[1] = (state[1] ≪ 1) | (state[1] ≫ 31)
2: state[2] = (state[2] ≪ 2) | (state[2] ≫ 30)
3: state[3] = (state[3] ≪ 3) | (state[3] ≫ 29)
4: state[4] = (state[4] ≪ 4) | (state[4] ≫ 28)
5: state[5] = (state[5] ≪ 5) | (state[5] ≫ 27)
```

Figure 3-5. Bitsliced implementations of the ShiftColumns transformation.

```
Input: X0-X5 (r1-r3, r10-r12)
Output: Y0-Y5 (r4-r9)
1: MOV Y0, X0
2: MOV Y1, X1, ROR #31
3: MOV Y2, X2, ROR #30
4: MOV Y3, X3, ROR #29
5: MOV Y4, X4, ROR #28
6: MOV Y5, X5, ROR #27
```

Figure 3-6. ARMv6 assembly implementations of the ShiftColumns operation in a bitsliced form.

### 3.3.4 Optimized MixColumns via Rotation-XOR Fusion

MixColumns (MC) applies, independently for each column j, a linear transformation defined by multiplication with a fixed 32×32 binary circulant matrix. In the 6×32 bitsliced representation, each column is a 32-bit word, and the circulant structure allows MC to be expressed as the XOR of a fixed set of rotated versions of that word. Following Kim et al. [31], each updated column word can be written as an XOR accumulation over six rotation offsets:

$$Y = ROR_1(X) \oplus ROR_5(X) \oplus ROR_9(X) \oplus ROR_{15}(X) \oplus ROR_{21}(X) \oplus ROR_{26}(X),$$

where X denotes the post-ShiftColumns column word and ROR_k is a 32-bit right rotation by k. Because the offsets are fixed constants, MixColumns is implemented as a straight-line sequence of rotate-and-XOR operations with no branches and no table lookups.

ARM Cortex-M3 (rotation-XOR fusion). The barrel shifter enables an efficient fusion of rotation and XOR. We accumulate the rotated terms using instruction forms such as:

EOR dst, dst, src, ROR #imm,

which rotates src by an immediate amount and XORs the result into dst in a single instruction. Concretely, for each column word, we initialize the accumulator with a rotated copy (one MOV-with-ROR), then XOR in the remaining five rotated terms using five EOR-with-ROR instructions. This yields six XOR-type instructions per column and 36 instructions across all six columns. Figure 3-7 shows the resulting Cortex-M3 assembly realization.

RV32I (rotation synthesis). RV32I lacks a rotate instruction, so each rotated term ROR_k(X) is synthesized using two shifts and one merge operation, and then XORed into the accumulator. Thus, each rotation term costs four instructions in total (SRLI, SLLI, merge, and XOR). With six rotation offsets per column, MixColumns requires 24 instructions per column and 144 instructions for all six columns. As with ShiftColumns, the implementation remains register-only, table-free, and constant-time; the performance gap relative to Cortex-M3 is primarily attributable to the absence of ISA-level rotate support.

```
Input: X0-X5 (r1-r3, r10-r12),
       Y0-Y5 (r4-r9)
Output: Y0-Y5 (r4-r9)

1: EOR Y0, Y0, X0, ROR #31
2: EOR Y0, Y0, X0, ROR #27
3: EOR Y0, Y0, X0, ROR #23
4: EOR Y0, Y0, X0, ROR #17
5: EOR Y0, Y0, X0, ROR #11
6: EOR Y0, Y0, X0, ROR #6

7: EOR Y1, Y1, X1, ROR #30
8: EOR Y1, Y1, X1, ROR #26
9: EOR Y1, Y1, X1, ROR #22
10: EOR Y1, Y1, X1, ROR #16
11: EOR Y1, Y1, X1, ROR #10
12: EOR Y1, Y1, X1, ROR #5

13: EOR Y2, Y2, X2, ROR #29
14: EOR Y2, Y2, X2, ROR #25
15: EOR Y2, Y2, X2, ROR #21
16: EOR Y2, Y2, X2, ROR #15
17: EOR Y2, Y2, X2, ROR #9
18: EOR Y2, Y2, X2, ROR #4

19: EOR Y3, Y3, X3, ROR #28
20: EOR Y3, Y3, X3, ROR #24
21: EOR Y3, Y3, X3, ROR #20
22: EOR Y3, Y3, X3, ROR #14
23: EOR Y3, Y3, X3, ROR #8
24: EOR Y3, Y3, X3, ROR #3

25: EOR Y4, Y4, X4, ROR #27
26: EOR Y4, Y4, X4, ROR #23
27: EOR Y4, Y4, X4, ROR #19
28: EOR Y4, Y4, X4, ROR #13
29: EOR Y4, Y4, X4, ROR #7
30: EOR Y4, Y4, X4, ROR #2

31: EOR Y5, Y5, X5, ROR #26
32: EOR Y5, Y5, X5, ROR #22
33: EOR Y5, Y5, X5, ROR #18
34: EOR Y5, Y5, X5, ROR #12
35: EOR Y5, Y5, X5, ROR #6
36: EOR Y5, Y5, X5, ROR #1
```

Figure 3-7. ARMv6 assembly realizations of the MixColumns step using a bitsliced representation.

### 3.3.5 Register and Memory Usage Strategies

Bitsliced implementations are highly sensitive to register pressure, since intermediate Boolean terms and rotated copies can quickly exceed the available register budget. Our assembly design therefore aims to keep the 6×32 state resident in registers throughout the round function and to minimize stack spills, as spills introduce additional load/store traffic and can dominate runtime on embedded cores.

ARM Cortex-M3. Although the architecture provides 16 architectural registers, the program counter (PC) and stack pointer (SP) are fixed, and the link register (LR) is typically reserved for control flow unless explicitly saved. In practice, only a limited subset of registers can be

dedicated to the cipher state and temporaries. We allocate six registers for the state planes S0–S5 and keep the round–key pointer in a dedicated register. The remaining registers are budgeted for temporaries required by SubBox and by the rotation/XOR–based linear layer. To avoid long–lived intermediates, we structure the Boolean network and linear–layer accumulation to reuse temporaries aggressively and to limit the number of simultaneously live rotated values. With this allocation, all transformations except AddRoundKey are executed in a register–only manner; any required pointer preservation (e.g., output pointer) is handled at function entry/exit to keep memory traffic off the critical path.

RV32I (RISC–V). RV32I provides a larger general–purpose register file, allowing the bitsliced state, the round–key pointer, and a larger set of temporaries to remain in registers throughout encryption. Consequently, stack spills are largely avoided and the runtime is dominated by the fixed instruction sequence of the Boolean S–box network and the synthesized rotations in the linear layer. This register–rich setting is particularly beneficial for SPEEDY–192, where multiple intermediate values are needed to implement the SubBox and MixColumns layers efficiently. Keeping the six state planes and the critical intermediates in registers is particularly important for the rotation–heavy linear layer (Sections 3.3.3–3.3.4), because spill/reload traffic would quickly negate the benefits of rotation–XOR fusion on Cortex–M3 and amplify the rotation–synthesis overhead on RV32I. Building on this register–resident design, Section 3.3.6 validates constant–time behavior and summarizes the measured cycles–per–byte results.

### 3.3.6 Constant–Time Properties and Performance

Building on the register-resident implementation strategy described in Section 3.3.5, we evaluate the constant-time properties of the proposed SPEEDY implementations and summarize the measured cycles-per-byte results on Cortex-M3 and RV32I.

Constant-time properties. The proposed SPEEDY implementations follow a strict constant-time execution model. First, the SubBox layer is evaluated as a Boolean circuit using only word-level logical operations rather than lookup tables, so there is no secret-dependent memory access. Second, SubBox, ShiftColumns, MixColumns, and AddRoundKey execute a fixed instruction sequence independent of the secret key and plaintext. Third, the memory access pattern is input-independent: packing/unpacking and AddRoundKey access plaintext/ciphertext and round keys in a sequential manner, and no data-dependent branches are used. The number of rounds and loop trip counts are fixed, making the execution path identical for all inputs [12, 18, 20, 40, 51, 54].

Performance. Table 3-2 reports encryption speed in cycles per byte (cpb) on ARM Cortex-M3 and RV32I, and compares our SPEEDY implementations with representative constant-time implementations of AES-128 and GIFT-128 on the same platforms. On Cortex-M3, SPEEDY-6-192 is approximately 1.6× faster than constant-time AES-128 (120.4 cpb) and about 1.3× faster than GIFT-128 (104.1 cpb). For SPEEDY-7-192 on Cortex-M3, the bitsliced implementation improves over the byte-oriented C reference by roughly 180× (15,407 cpb baseline). On RV32I, SPEEDY-6-192 remains competitive; compared to an 8-way constant-time AES-128 implementation (78.9 cpb), SPEEDY-6-192 is about 1.2× slower but still provides practical throughput. For SPEEDY-7-192 on RV32I, the speedup over the reference implementation is on the order of 165× (18,096 cpb baseline).

These results confirm that, although SPEEDY is optimized for

ultra-low gate latency in hardware, a carefully chosen 6×32 bitsliced representation combined with architecture-aware assembly optimization enables both high performance and constant-time behavior on resource-constrained 32-bit microcontrollers.

| Algorithm | Speed (cpb) | Block Size | Parallel Blocks |
|---|---|---|---|
| **ARM Cortex-M3** | | | |
| GIFT-128 encryption [9] | 104.1 | 128 | 1 |
| AES-128 encryption [19] | 120.4 | 128 | 2 |
| SPEEDY-7-192 encryption (reference) | 15,407 | 192 | 1 |
| SPEEDY-5-192 encryption (ours) | 65.7 | 192 | 1 |
| SPEEDY-6-192 encryption (ours) | 75.2 | 192 | 1 |
| SPEEDY-7-192 encryption (ours) | 85.1 | 192 | 1 |
| **RISC-V** | | | |
| AES-128 encryption [19] | 78.9 | 128 | 8 |
| SPEEDY-7-192 encryption (reference) | 18,096 | 192 | 1 |
| SPEEDY-5-192 encryption (ours) | 81.9 | 192 | 1 |
| SPEEDY-6-192 encryption (ours) | 95.5 | 192 | 1 |
| SPEEDY-7-192 encryption (ours) | 109.2 | 192 | 1 |

Table 3-2. Comparison of our SPEEDY implementations with other constant-time implementations on ARM Cortex-M3 and RISC-V; performance is reported in clock cycles per byte (cpb).

*Note: cpb = total cycles / processed bytes, including packing/unpacking and the encryption core; round keys are assumed precomputed once per key. "Reference" denotes a byte-oriented baseline on the conventional packed layout.*

# Ⅳ. AES-GCM Optimization on ARM Cortex-M4

## 4.1 Problem Definition and System Model

### 4.1.1 Problem Definition

The Advanced Encryption Standard in Galois/Counter Mode (AES-GCM) is widely deployed as a de facto standard authenticated-encryption-with-associated-data (AEAD) scheme in modern security protocols and embedded update mechanisms, including TLS/DTLS, SRTP, IPsec, and secure firmware updates for Internet-of-Things (IoT) devices [10, 42, 61]. AES-GCM provides confidentiality via AES in counter mode (CTR/GCTR) and integrity via the GHASH universal hash function over $GF(2^{128})$. Although lightweight AES variants have been proposed for resource-constrained IoT devices, many deployed stacks still rely on standardized AES-GCM, motivating continued optimization of the standard construction on microcontrollers [19]. In many embedded systems, AES-GCM is therefore executed frequently and often dominates the CPU budget for secure communication and storage.

Achieving both high performance and side-channel resilience on low-cost 32-bit microcontrollers remains challenging. In this chapter, we focus on ARM Cortex-M4-class devices, where AES-GCM must be implemented without dedicated cryptographic acceleration. The key challenges are summarized as follows.

(1) Limited computational resources and register pressure. Cortex-M4 cores are designed for low-power operation and provide only a small general-purpose register budget for simultaneously holding the

AES state, round keys, and GHASH intermediates. As a result, naïve or byte-oriented implementations incur heavy load/store traffic and tend to exhibit high cycles-per-byte (cpb), especially for message sizes of several kilobytes that are common in firmware transfers and buffered logging.

(2) Timing side-channel risks of table-based software. Conventional high-speed AES implementations often rely on S-box or T-table lookups. However, table indices depend on secret state and key material, and their access latency can vary due to flash wait states, bus contention, and microarchitectural effects even on microcontrollers with little or no data caching [12, 14, 18, 20, 54]. Similar concerns apply to GHASH if it is realized using secret-indexed tables derived from the hash subkey.

(3) The cost of GHASH without carry-less multiplication. GHASH requires multiplication in $GF(2^{128})$. On platforms lacking carry-less multiplication instructions (e.g., PMULL on high-end ARM cores), this must be emulated in software using either small tables or integer-multiplication-based methods (e.g., Karatsuba). In practice, GHASH can consume a large fraction of total AES-GCM time, so optimizing AES-CTR alone yields diminishing returns unless GHASH is addressed jointly [32, 49, 60].

Accordingly, this chapter studies a constant-time, fixsliced AES-CTR core on Cortex-M4, integrates FACE-style caching in the fixsliced domain, and evaluates two GHASH design points that span the performance-security trade-off space.

### 4.1.2 System and Threat Model

#### 4.1.2.1 Hardware platform

We target an ARMv7-M Cortex-M4 microcontroller platform, with STM32F4-class devices as a representative instance [5]. Cortex-M4 provides efficient 32-bit integer arithmetic, single-cycle multiplication, and a barrel shifter that enables low-cost shifts/rotations within data-processing instructions. However, it does not provide dedicated AES instructions or carry-less multiplication, so both AES and GHASH must be realized using base integer and bitwise operations.

The relevant architectural constraints are as follows.

Register constraints. Cortex-M4 exposes 16 architectural registers, but PC, SP, and LR are reserved for control flow and stack management in typical calling conventions. Hence, only a limited subset of registers can be dedicated to the active cryptographic state and temporaries, which makes register-aware implementation essential.

Memory hierarchy. Code is usually executed from flash and data is stored in SRAM. While the core may employ simple prefetch/buffer mechanisms, it generally lacks a large data cache, and memory-access latency can vary depending on bus arbitration and flash wait states. Consequently, reducing memory traffic and avoiding secret-dependent addressing are key for both performance and side-channel robustness.

Instruction efficiency. Word-level logical operations (AND, EOR, ORR, shifts) are fast and predictable on Cortex-M4. In contrast, frequent table loads from flash/SRAM can become a bottleneck and may introduce timing variability. This motivates a computation-centric (table-free) design whenever strict constant-time behavior is required.

#### 4.1.2.2 Software and protocol context

We evaluate AES-128-GCM and AES-256-GCM for message lengths ranging from 1 KB to 40 KB, reflecting common embedded workloads such as firmware-update chunks, sensor-log uploads, and SRTP-style packet aggregation [10, 42, 43, 61]. Benchmarks are reported for representative sizes (1 KB, 4 KB, 20 KB, and 40 KB) to capture both short-message behavior and the amortization effects that arise when initialization costs (e.g., key schedule and hash-subkey derivation) are spread over longer streams.

The intended usage model assumes that many consecutive blocks are processed under a fixed key and nonce within a session, which is typical for bulk data transfer in embedded secure-communication stacks.

#### 4.1.2.3 Threat model

We consider a timing side-channel adversary who can measure fine-grained execution-time variations of cryptographic routines on the target device. Concretely, the attacker may (i) run co-resident software tasks or DMA engines that influence/observe bus contention, or (ii) repeatedly invoke the cryptographic service and perform statistical timing analysis on the resulting measurements [20, 40, 54]. Under this model, secret-dependent table lookups and secret-dependent control flow are considered unsafe, as they can amplify observable timing differences.

In response, we adopt the following design policy in this chapter.

AES-CTR (encryption core). We enforce strict constant-time behavior by using a fixsliced, table-free AES realization with no secret-dependent branches or memory accesses.

GHASH (authentication core). We evaluate two design points: (a) a compact 4-bit table-based GHASH that prioritizes performance but does

not provide strict constant-time guarantees with respect to secret-indexed table accesses, and (b) a table-free Karatsuba-based GHASH that follows a fixed instruction sequence and is suitable for high-assurance constant-time deployments [29, 32, 48, 49].

This separation makes the security-performance trade-offs explicit and allows implementers to select an appropriate configuration depending on the threat model and resource constraints.

## 4.2 Related Work

This section reviews (i) constant-time AES and AES-GCM efforts on ARM Cortex-M processors, (ii) the FACE acceleration technique for AES-CTR, and (iii) software techniques for GHASH multiplication over $GF(2^{128})$ on platforms without carry-less multiplication. Based on this review, we position the contribution of this chapter in the context of prior embedded AEAD optimization studies.

### 4.2.1 AES and AES-GCM on ARM Cortex-M

Early AES software on Cortex-M3/M4 largely relied on byte-oriented code and lookup tables (S-box or T-tables) to achieve practical throughput. However, table-based designs involve data-dependent memory accesses and are therefore incompatible with strict constant-time requirements under realistic microarchitectural timing leakage models, including effects from flash wait states and bus/interconnect contention [12, 14, 18, 20, 54]. This motivated a shift toward table-free constant-time implementations based on bitslicing and fixslicing.

A major milestone was achieved by Schwabe and Stoffelen, who

reported an efficient constant-time AES-CTR implementation on Cortex-M3/M4 using 2-block bitslicing with a compact Boolean S-box [56]. Adomnicai and Peyrin subsequently introduced fixslicing for AES-like ciphers, reorganizing the state layout and absorbing ShiftRows into round-dependent MixColumns variants to reduce the cost of permutations and reach further speed records on ARM Cortex-M and RISC-V platforms [2]. These approaches share three essential properties: (i) the S-box is realized as a Boolean network rather than a table, (ii) multiple blocks are interleaved to exploit word-level parallelism, and (iii) packing/unpacking overhead is amortized over multi-block processing.

While constant-time AES-CTR has been extensively optimized, full AES-GCM implementations frequently remain dominated by GHASH on microcontrollers that lack carry-less multiplication. As a result, improving the AES core alone often yields diminishing end-to-end gains unless GHASH is addressed jointly within the same evaluation framework [22, 32, 60].

#### 4.2.2 FACE: Fast AES-CTR Encryption

FACE (Fast AES-CTR Encryption), proposed by Park et al. [46], exploits the observation that consecutive counter blocks in CTR mode differ only in a small portion of the input (typically the low counter bytes). Consequently, large parts of the intermediate AES state remain identical across successive counter values. FACE leverages this structure by caching invariant intermediate values and recomputing only the variant portion, thereby reducing redundant operations in AES-CTR.

FACE was originally evaluated mainly in byte-oriented or table-based AES settings (and in some cases in conjunction with hardware-accelerated instruction sets). Integrating FACE into a fixsliced AES core is non-trivial, because fixslicing uses a different internal state

layout and replaces the explicit ShiftRows operation with a schedule of MixColumns variants. Therefore, a direct reuse of the original FACE mapping is generally impossible; the cache representation and recombination logic must be redesigned to operate directly in the fixsliced domain. Table 4-1 summarizes the memory overhead and effective caching interval of representative FACE variants and serves as a reference point for the performance-memory trade-offs analyzed in this chapter.

Importantly, FACE caches are indexed by public counter values rather than secret-dependent state, so FACE can be compatible with strict constant-time requirements at the AES-CTR layer when implemented without secret-dependent branching.

| Variant | Caching Idea | Memory | Interval |
|---|---|---|---|
| $\text{FACE}_{rd0}$ | Cache 12 bytes from Round 0 AddRoundKey, reusing static counter bytes. | 12 | $2^{40}$ |
| $\text{FACE}_{rd1}$ | Extend cache to Round 1 intermediates post SubBytes/MixColumns, reusing as difference affects one column but spreads fully. | 12 | $2^{8}$ |
| $\text{FACE}_{rd1+}$ | Generate 256-entry table for Round 1, reusing via last-byte difference indexing without recomputation. | 1024 | $2^{40}$ |
| $\text{FACE}_{rd2}$ | Cache 16 bytes from Round 2 intermediates post ShiftRows/MixColumns, reusing as initial 4-byte difference (first column) spreads fully. | 16 | $2^{8}$ |
| $\text{FACE}_{rd2+}$ | Precompute Round 2 post-MixColumns with larger table, XORing precomputed value with counter difference. | 4096 | $2^{40}$ |

Table 4-1. Main parameters of the FACE variants. Memory overhead is given in bytes, and the interval denotes the number of blocks for which the caching mechanism remains effective.

### 4.2.3 GHASH and $GF(2^{128})$ Multiplication

The GHASH function in GCM repeatedly multiplies 128-bit field elements in $GF(2^{128})$. On x86 platforms, carry-less multiplication

instructions enable efficient GHASH evaluation for GCM, providing a contrasting acceleration path that is unavailable on Cortex-M4-class devices [23]. On microcontrollers without carry-less multiplication instructions (e.g., PMULL), this multiplication must be implemented in software. Prior work has broadly followed two approaches.

Table-based multiplication precomputes a set of multiples of the hash subkey H and updates the accumulator by scanning the input block in small chunks (typically 4, 8, or 16 bits). Larger chunk sizes reduce the number of iterations but require larger tables (kilobytes), which can be impractical on memory-constrained devices. As a result, 4-bit (nibble) tables are often used as a balanced compromise between memory overhead and throughput.

Multiplication-based constant-time designs avoid secret-indexed tables and instead emulate carry-less multiplication using integer operations, masks, and shifts. A widely used example is the constant-time GHASH routine in BearSSL, which employs a Karatsuba decomposition to compute the 128-bit product using a small number of 32-bit multiplications and fixed bit manipulations [48, 49]. These designs follow a fixed instruction sequence and do not perform secret-dependent memory accesses, making them suitable for high-assurance constant-time deployments. The trade-off is that they are typically slower than table-based methods on the same microcontroller.

Because table-based GHASH can introduce secret-dependent access patterns (through indices derived from H and the evolving hash state), its suitability depends on the assumed timing-leakage model and the deployment environment. Consequently, it is useful to evaluate both a fast table-based design and a table-free constant-time design within the same AES-GCM framework [20, 54].

#### 4.2.4 Positioning of This Study

To summarize, prior work provides: (i) high-speed constant-time bitsliced/fixsliced AES cores on Cortex-M [2, 56], (ii) FACE-based acceleration strategies for AES-CTR [46], and (iii) standalone GHASH implementations spanning table-based and constant-time multiplication-based techniques [32, 49]. However, an integrated evaluation that combines a fixsliced AES-CTR core, a FACE mechanism redesigned for the fixsliced state layout, and a controlled comparison of GHASH design points on an ARM Cortex-M4-class platform has been limited.

This chapter addresses that gap by integrating these components under a unified methodology and reporting (a) cycles-per-byte for AES-GCTR and full AES-GCM across realistic message lengths, (b) the memory overhead of FACE caches and GHASH tables, and (c) the constant-time implications of each design choice. This provides an explicit design space for implementers to select an appropriate AES-GCM configuration on Cortex-M4 depending on performance constraints and threat assumptions [32, 60].

## 4.3 Proposed Implementation Methodology

This section provides a comprehensive technical description of the proposed AES-GCM framework. The methodology focuses on integrating a Fixsliced AES-CTR core with a redesigned FACE caching mechanism and dual GHASH realizations to provide a flexible and high-performance AEAD solution for the ARM Cortex-M4.

### 4.3.1 Fixsliced Constant-Time AES-CTR Core

As the foundation of our encryption suite, we adopt and adapt the 2-way Fixsliced AES design proposed by Adomnicai and Peyrin [2]. This

core is tuned for the ARMv7-M (Cortex-M4) microarchitecture to provide both practical performance and robust constant-time behavior.

Two-block bitsliced state representation. In our implementation, two independent 128-bit blocks are interleaved into eight 32-bit registers (R0-R7), a configuration commonly referred to as 2-way bitslicing. Each register is partitioned into sixteen 2-bit lanes, where each lane corresponds to the same byte position across the two parallel blocks. As illustrated in Figure 4-1, this layout aligns the internal state with the 32-bit word-level datapath of the Cortex-M4 and allows bitwise operations to update two blocks simultaneously, while keeping register pressure within the practical limits of the ARM calling convention.

Round function optimization. The round function is engineered to avoid lookup tables and explicit bit-level permutations.

(1) Ark_Sub Boolean network. The SubBytes transformation is implemented as a Boolean circuit and is merged with AddRoundKey into a single optimized network, denoted Ark_Sub. It is realized using only word-level logical instructions (e.g., AND, EOR, ORR, and BIC), ensuring a fixed instruction sequence independent of secret data.

(2) Absorbed ShiftRows via MC variants. A defining characteristic of Fixslicing is that ShiftRows is not executed as an explicit permutation. Instead, its effect is absorbed into the linear layer by using four MixColumns variants (MC0-MC3) in a round-dependent schedule. This keeps the state bits in fixed register positions throughout the rounds and eliminates the overhead of implementing ShiftRows in the bitsliced domain.

CTR mode integration and baseline performance. AES-GCTR input blocks are constructed by concatenating a 96-bit IV and a 32-bit incrementing counter. Consecutive counter values are packed into the

Fixsliced representation, encrypted using the core described above, unpacked back to the normal layout, and finally XORed with the plaintext to produce ciphertext. On the STM32F4 platform, the resulting baseline Fixsliced AES-GCTR implementation achieves approximately 80 cycles per byte (cpb) for basic encryption.

Constant-time properties. The core satisfies strict constant-time requirements: it uses no secret-indexed lookup tables, contains no secret-dependent conditional branches, and executes a fixed number of rounds with a fixed instruction schedule. In addition, packing/unpacking and the XOR-with-plaintext step follow deterministic memory-access patterns. Table 4-2 summarizes the measured cycle counts for AES-GCTR across different message sizes and shows how FACE variants further improve throughput over this baseline.

| | 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 |
| $R_0$ | $b_{24}^0$ | $b_{24}^1$ | $b_{56}^0$ | $b_{56}^1$ | $b_{88}^0$ | $b_{88}^1$ | $b_{120}^0$ | $b_{120}^1$ | ⋯ | $b_0^0$ | $b_0^1$ | $b_{32}^0$ | $b_{32}^1$ | $b_{64}^0$ | $b_{64}^1$ | $b_{96}^0$ | $b_{96}^1$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋯ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $R_7$ | $b_{31}^0$ | $b_{31}^1$ | $b_{63}^0$ | $b_{63}^1$ | $b_{95}^0$ | $b_{95}^1$ | $b_{127}^0$ | $b_{127}^1$ | ⋯ | $b_7^0$ | $b_7^1$ | $b_{39}^0$ | $b_{39}^1$ | $b_{71}^0$ | $b_{71}^1$ | $b_{103}^0$ | $b_{103}^1$ |

Figure 4-1. Bitsliced layout following [2] that uses eight 32-bit registers $(R_0,\ldots,R_7)$ to process two blocks $(b^0,b^1)$ in parallel. The symbol $(b_j^i)$ denotes the $j$-th bit of the $i$-th block, illustrating how Fixslicing arranges bits across registers after the packing stage [2].

| 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%) |
| $FACE_{rd1}$ | 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%) |
| $FACE_{rd1+}$ | 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%) |
| $FACE_{rd2+}$ | 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%) |

Table 4-2. Cycle counts for AES-GCTR using different FACE variants (AES-128 and AES-256) on messages from 1 KB to 40 KB; percentages in parentheses indicate the improvement over the Basic configuration.

### 4.3.2 FACE Integration in the Fixsliced Domain

A primary technical challenge in this study is integrating FACE (Fast AES-CTR Encryption) into a 2-way Fixsliced AES-CTR core. In conventional byte-oriented AES, FACE can cache and reuse intermediate results by directly storing selected state bytes/columns. In the Fixsliced domain, however, bits belonging to a single byte (and thus a single column) are distributed across multiple registers and lanes. Therefore, applying FACE "as is" would incur expensive format conversions that would erase most of the potential speedup. To avoid this overhead, we redesign FACE so that both caching and reconstruction are performed directly in the Fixsliced representation [2, 46].

FACE relies on the CTR-mode property that consecutive counter blocks differ only in a small, public portion of the input (typically the low 8 bits of the 32-bit counter). Consequently, a large fraction of the AES intermediate state remains identical across successive counters. The following variants exploit this redundancy at different caching depths and memory budgets.

FACE(rd1): caching invariant components after round 1. After the

first−round MixColumns, the difference introduced by incrementing the low counter byte is confined to a single AES column, while the other three columns remain invariant. In a byte−oriented implementation, this enables straightforward caching of the invariant columns. In our Fixsliced setting, we achieve the same effect by applying a precomputed lane mask to the round−1 Fixsliced state: lanes corresponding to the variant column are zeroed, while lanes corresponding to the three invariant columns are preserved. The resulting eight 32−bit words (R0-R7) are stored as the FACE(rd1) cache. For subsequent counter values, only the variant column contribution is recomputed and XORed into this cached state before continuing with the remaining rounds. On STM32F4, FACE(rd1) yields an 8-10% speedup for AES−128 GCTR on messages of at least several kilobytes (Table 4−2).

FACE(rd1+): 1 KB table for eliminating round−1 recomputation. FACE(rd1+) further reduces work by precomputing the variant−column contribution for all 256 possible values of the low counter byte. The key observation is that our 2−way Fixsliced core exposes eight independent AES columns per evaluation (4 columns per block × 2 blocks = 8 columns). We exploit this structure by packing eight candidate counter values into a single Fixsliced state such that the varying column for each candidate occupies one of these eight column slots. After executing round 1 once, we obtain the corresponding eight round−1 variant−column results simultaneously and store them as a single 32−byte record (8 columns × 4 bytes/column). Repeating this procedure for all 32 groups of eight values covers the full 0-255 range, yielding a 1 KB table (32 records × 32 bytes = 1024 bytes). At runtime, the low counter byte b selects (i) a group index g = ⌊b/8⌋ and (ii) an in−group offset o = b mod 8. The implementation then loads the 32−byte record for group g and uses fixed word rotations and lane−alignment operations to extract the o−th column contribution and merge it with the FACE(rd1) cache. This effectively bypasses the entire round−1 computation for each block,

while keeping all indexing dependent only on the public counter value (Figure 4-2).

FACE(rd2+): 4 KB extended cache up to round 2. FACE(rd2+) increases the caching depth to the state after the second-round MixColumns, thereby skipping approximately two rounds of work in the online path. Because the 2-way Fixsliced core naturally processes counter pairs (c, c+1) together, we construct a table indexed by the low counter byte pair (2k, 2k+1). This results in 128 entries, each 32 bytes, for a total of 4 KB. For each pair, we precompute and store the Fixsliced contribution needed to reconstruct the round-3 input when combined with a precomputed FACE(rd2) cache. At runtime, the appropriate entry is selected using the public counter value and XORed with the cached invariant state before applying the Ark_Sub network, thereby reducing the effective round count. For AES-128 GCTR, this variant achieves up to a 19.4% throughput improvement for 40 KB messages (Table 4-2), at the cost of increased flash footprint (Table 4-1).

In summary, FACE(rd1), FACE(rd1+), and FACE(rd2+) form a clear performance-memory trade-off ladder. All FACE variants in this chapter remain compatible with strict constant-time requirements at the AES-CTR layer because the cache/table lookups are indexed solely by public counter values, and reconstruction is implemented using fixed instruction sequences without secret-dependent branching.

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

Figure 4-2. C implementation that rearranges the FACE rd1+ columns for each block, where ROR32 denotes a 32-bit right-rotation operation.

#### 4.3.3 GHASH: 4-Bit Table vs. Karatsuba

In AES-GCM, the GHASH component typically dominates the overall cost on Cortex-M4-class microcontrollers, because the platform lacks dedicated carry-less multiplication instructions. To characterize the practical design space, we implement and compare two GHASH realizations that represent opposite ends of the performance-assurance spectrum: (i) a compact 4-bit table-based multiplier for high throughput and (ii) a table-free constant-time multiplier based on Karatsuba decomposition.

##### 4.3.3.1 4-bit table-based GHASH

Our high-speed GHASH uses a 4-bit (nibble) table. For the hash subkey $H \in GF(2^{128})$, we precompute a table T of 16 field elements:

$$T[i] = i \cdot H, \text{ for } i \in \{0, 1, \cdots, 15\}.$$

Since each table entry is 128 bits, the total table size is $16 \times 16$ bytes = 256 bytes.

For each 128−bit input block X, GHASH updates the accumulator Y as:

$$Y \leftarrow (Y \oplus X) \cdot H,$$

where multiplication is performed by scanning X in 4−bit chunks. Concretely, the algorithm iterates over 32 nibbles, and in each iteration it performs (i) a fixed 4−bit shift of the intermediate value, (ii) an XOR with the precomputed table entry selected by the current nibble, and (iii) a fixed reduction step modulo the GCM polynomial. The loop count is constant, and the instruction sequence per iteration is fixed.

Performance and security implications. This 4−bit method offers a favorable balance on memory−constrained MCUs (small table, reduced arithmetic), and in our measurements it is approximately 2× faster than the Karatsuba−based baseline for long messages (Tables 4−3 and 4−4). However, the table index is derived from values that depend on the evolving GHASH state and H; therefore, memory accesses can be secret−dependent. Under timing−leakage models that consider bus contention or flash/SRAM latency variability, this implementation is not strictly constant−time.

FPU−register−based optimization. To reduce load/store pressure, we additionally evaluate an FPU−assisted variant that keeps selected 32−bit constants and frequently reused intermediate words (derived from H and the reduction process) in floating−point registers. This reduces repeated SRAM accesses and mitigates integer−register spills, leading to measurable cycle−count reductions as summarized in Table 4−3.

| Implementation | 1 KB | 4 KB | 20 KB | 40 KB |
|---|---|---|---|---|
| FPU-Optimized | 155,535 | 598,095 | 2,958,415 | 5,908,815 |
| Original | 158,749 | 611,293 | 3,024,861 | 6,041,821 |
| **Improvement (%)** | *2.02%* | *2.16%* | *2.20%* | *2.20%* |

Table 4-3. Performance comparison of GHASH implementations with and without FPU-register-based optimizations.

#### 4.3.3.2 Karatsuba-based constant-time GHASH

For high-assurance settings, we implement a strictly constant-time GHASH multiplication based on BearSSL's constant-time routine (ghash_ctmul) [49]. The implementation is table-free and avoids secret-dependent control flow.

Methodology. We represent 128-bit operands using fixed 32-bit limbs and compute the carry-less product via a Karatsuba decomposition. This reduces the number of required limb multiplications compared to a naive approach; in particular, our routine performs nine 32-bit multiplications per GHASH multiplication, combined with a fixed pattern of XORs, shifts, and masks. The resulting 256-bit intermediate product is then reduced modulo the standard GCM polynomial using a deterministic sequence of bit operations.

Constant-time properties. The routine performs no table lookups indexed by secret-dependent values and contains no secret-dependent branches. The number of iterations and the instruction schedule are invariant across all inputs, which makes this design suitable under strict constant-time requirements.

FPU-register-based optimization. Similar to the table-based variant, we consider an FPU-assisted implementation that stores frequently used 32-bit constants and selected intermediate values in FPU registers. On STM32F4, this yields an additional ~2% improvement by reducing

memory traffic and relieving register pressure. Table 4-4 reports GHASH cycle counts across message sizes and highlights the impact of the optimization.

| GHASH | 1 KB | 4 KB | 20 KB | 40 KB |
|---|---|---|---|---|
| Karatsuba | 155,535 *(0%)* | 598,095 *(0%)* | 2,958,415 *(0%)* | 5,908,815 *(0%)* |
| Table-based | 79,402 *(+48.9%)* | 302,390 *(+49.4%)* | 1,491,670 *(+49.6%)* | 2,978,281 *(+49.6%)* |

Table 4-4. Cycle counts for GHASH on 1-40 KB messages; values in parentheses show the performance improvement relative to the Karatsuba baseline.

*Note: the 4-bit table GHASH is faster but not strictly constant-time due to secret-derived table indices; the Karatsuba GHASH is table-free and designed for strict constant-time behavior.*

### 4.3.4 Summary

This chapter presented an integrated AES-GCM software framework for ARM Cortex-M4 that combines (i) a high-speed constant-time Fixsliced AES-CTR core, (ii) FACE caching redesigned to operate directly in the Fixsliced domain, and (iii) two GHASH realizations that expose a practical performance-assurance trade-off. The goal was to move beyond isolated optimizations of AES-CTR or GHASH and instead quantify their joint impact at the full AEAD level.

Fixsliced AES-CTR establishes the baseline: two 128-bit counter blocks are processed in parallel using a fixed 2-way bitsliced state layout, a Boolean Ark_Sub network, and round-dependent MixColumns variants that absorb ShiftRows. Building on this core, FACE(rd1), FACE(rd1+), and FACE(rd2+) exploit CTR-mode redundancy across consecutive counters while avoiding costly conversions back to a

byte-oriented representation. These variants form a clear ladder of trade-offs: increasing cache/table size yields progressively larger speedups in GCTR for long messages, with FACE(rd2+) providing the largest gain among the evaluated options.

For GHASH, we evaluated a compact 4-bit table-based multiplier and a table-free Karatsuba-based constant-time multiplier (adapted from BearSSL). The 4-bit table approach is designed for peak throughput with a small memory footprint, while the Karatsuba variant enforces strict constant-time behavior by avoiding secret-dependent table indices and branching. As discussed in Chapter 6, the 4-bit table method is roughly twice as fast as the Karatsuba baseline on long messages, whereas the Karatsuba design provides stronger timing-uniformity guarantees at a corresponding performance cost.

Table 4-5 summarizes the end-to-end AES-GCM cycle counts for 1-40 KB messages across FACE and GHASH combinations. Overall, the best-performing configuration for throughput-oriented settings is obtained by combining an aggressive FACE variant with the 4-bit table-based GHASH, whereas the recommended configuration for high-assurance settings is to pair FACE (indexed only by public counters) with the constant-time Karatsuba GHASH. This consolidated evaluation clarifies how to select an AES-GCM design point on Cortex-M4 depending on flash budget and side-channel requirements.

| 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 (+5.08%) | 915,445 (+3.33%) | 3,592,621 (+8.56%) | 4,425,658 (+5.02%) | 7,156,129 (+6.63%) | 8,813,752 (+5.24%) |
| | $FACE_{rd1+}$ | 206,894 (-0.25%) | 256,506 (-2.33%) | 721,076 (+7.58%) | 893,638 (+5.63%) | 3,470,230 (+11.68%) | 4,291,684 (+7.90%) | 6,904,704 (+9.90%) | 8,539,152 (+8.19%) |
| | $FACE_{rd2+}$ | 207,315 (-0.45%) | 256,511 (-2.33%) | 708,758 (+9.15%) | 880,339 (+7.04%) | 3,382,451 (+13.91%) | 4,206,896 (+9.72%) | 6,724,602 (+12.26%) | 8,365,109 (+10.06%) |
| Karatsuba | Basic | 282,902 (0%) | 327,931 (0%) | 1,076,438 (0%) | 1,247,227 (0%) | 5,311,098 (0%) | 6,148,219 (0%) | 10,603,898 (0%) | 12,274,939 (0%) |
| | $FACE_{rd1}$ | 284,351 (-0.51%) | 334,272 (-1.93%) | 1,039,879 (+3.40%) | 1,214,820 (+2.60%) | 5,073,356 (+4.48%) | 5,910,712 (+3.86%) | 10,114,091 (+4.62%) | 11,780,577 (+4.03%) |
| | $FACE_{rd1+}$ | 283,566 (-0.23%) | 333,490 (-1.70%) | 1,018,196 (+5.41%) | 1,193,067 (+4.34%) | 4,939,262 (+7.00%) | 5,777,138 (+6.03%) | 9,839,747 (+7.18%) | 11,507,228 (+6.26%) |
| | $FACE_{rd2+}$ | 284,542 (-0.58%) | 334,245 (-1.93%) | 1,008,037 (+6.35%) | 1,182,065 (+5.23%) | 4,869,485 (+8.31%) | 5,703,172 (+7.22%) | 9,695,385 (+8.66%) | 11,354,557 (+7.50%) |

Table 4-5. Cycle counts for AES-GCM on 1-40 KB messages under different GHASH implementations and FACE variants; percentages in parentheses are given relative to the Basic approach.

*Note: the 4-bit table GHASH is faster but not strictly constant-time due to secret-derived table indices; the Karatsuba GHASH is table-free and designed for strict constant-time behavior.*

# Ⅴ. High-Speed GPU Implementations of the Lightweight Block Ciphers PRESENT and GIFT

## 5.1 Problem Definition and System Model

### 5.1.1 The Server-Side Cryptographic Bottleneck

With the proliferation of Internet of Things (IoT) deployments and cloud computing infrastructures, data generated by a large number of edge devices is continuously aggregated at gateways and cloud servers [43, 59, 61]. Protecting this data requires not only secure transmission but also secure storage and processing, and block-cipher-based modes of operation are widely adopted for these purposes.

In large-scale server environments, however, cryptographic workloads can become a dominant performance bottleneck. For example, a server may be required to perform bulk encryption/decryption for incoming data streams from numerous gateways, periodic re-encryption of stored datasets, or integrity checks during backup and archival workflows. When such workloads scale, CPU-only execution can consume a substantial portion of the available compute budget, reducing system throughput and increasing end-to-end latency.

As discussed in Chapter 1, modern systems are increasingly heterogeneous. While resource-constrained microcontrollers (e.g., ARM Cortex-M and RV32I cores) perform local cryptographic processing at the edge, high-throughput processing is often required on the server side. This chapter addresses the problem of accelerating lightweight block ciphers on server-grade GPUs (Graphics Processing Units) and investigates GPU-oriented implementation strategies that improve throughput while preserving robust and predictable execution behavior.

### 5.1.2 System Model: Edge vs. Cloud

We assume a hierarchical system model consisting of three primary components:

**Edge/Device Side :** Edge nodes rely on low-power microcontrollers to execute lightweight cryptographic primitives such as PRESENT and GIFT [8, 15]. As shown in Chapters 3 and 4, these platforms often favor constant-time bitwise implementations (e.g., bitslicing/fixslicing) to avoid table-based timing leakage under tight register and memory constraints.

**Server Side :** Cloud servers and gateways must handle high-volume data processing, where throughput demands can reach the scale of tens to hundreds of gigabits per second in aggregate. In such settings, cryptographic processing can become a limiting factor if implemented solely on general-purpose CPU cores.

**GPU-Based Accelerator :** We target an NVIDIA RTX 3060 (Ampere architecture) as the server-side accelerator. Unlike register-constrained microcontrollers, GPUs execute thousands of threads using a SIMT (Single Instruction, Multiple Threads) model. Achieving high throughput therefore requires a GPU-aware re-design of the implementation strategy, typically shifting from low-degree bitslicing/fixslicing to high-degree bitslicing that maximizes arithmetic intensity while managing register pressure and occupancy.

### 5.1.3 CUDA Execution Hierarchy

Our PRESENT and GIFT implementations are designed to map

efficiently to CUDA's execution hierarchy (Figure 5-1). Threads are organized into warps of 32 threads, which execute in lockstep. To exploit this structure, we adopt a high-degree bitslicing strategy in which each thread processes 32 independent blocks in parallel in a bitsliced representation. Under this mapping, a single warp concurrently processes $32 \times 32 = 1024$ blocks during the round function, allowing the GPU to amortize control overhead and maximize the utilization of integer ALUs.

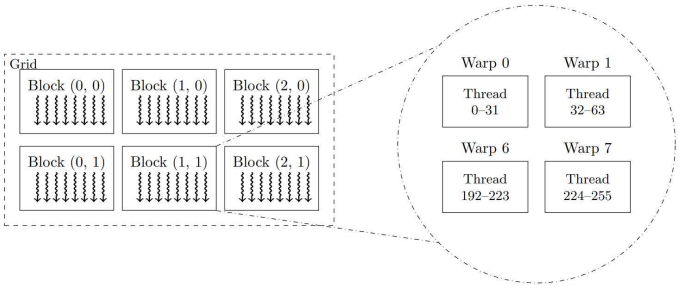


Figure 5-1. Hierarchy of CUDA's execution model, illustrating the arrangement of grids, blocks, warps, and threads.

Within this system model, this chapter aims to achieve three goals:

**High Throughput for Lightweight Ciphers** : We target throughput in the hundreds-of-Gbit/s range for cryptanalytic kernels (e.g., exhaustive key search) and competitive throughput for bulk encryption on an RTX 3060-class GPU, by expressing S-box and permutation layers as register-resident Boolean operations rather than lookup tables.

**Support for Multiple Workloads** : We support both (i) bulk encryption (e.g., ECB/CTR-style processing) and (ii) exhaustive key search. For key search, we emphasize on-the-fly generation of test inputs (e.g., counters/indices) directly in bitsliced form within the kernel to reduce host-to-device transfer overhead.

**Cross-Platform Consistency of the Implementation Paradigm** : By using bitslicing as the core design principle on both embedded processors (low-degree) and GPUs (high-degree), we maintain a coherent implementation perspective across heterogeneous platforms, enabling more systematic comparison and design guidelines in later chapters.

## 5.2 Related Work

The acceleration of block ciphers on Graphics Processing Units (GPUs) has been actively studied for both widely deployed standards (e.g., AES) and lightweight primitives designed for constrained environments [25, 36, 44]. The literature in this area broadly evolves from table-driven implementations that prioritize raw throughput to computation-centric designs (e.g., bitslicing) that mitigate memory bottlenecks and timing variability. This section reviews representative GPU implementation approaches and clarifies the research gap addressed by this chapter, with a particular focus on PRESENT and GIFT.

### 5.2.1 GPU-Based Block Cipher Acceleration Techniques

Early GPU implementations of block ciphers primarily relied on lookup tables (LUTs) for nonlinear layers and/or permutation layers. In these designs, S-boxes and precomputed round components are stored in global, constant, shared, or texture memory, and performance is improved by optimizing memory access patterns and reducing uncoalesced loads. Typical engineering techniques include replicating tables across suitable memory spaces, tuning thread/block configurations to increase Streaming Multiprocessor (SM) occupancy, and aligning data layouts to improve memory coalescing.

GPU parallelism has been exploited in two main granularities.

Fine-grained parallelism assigns one block (or one stream lane) to each thread, maximizing concurrency but often increasing the relative cost of memory traffic. Coarse-grained parallelism, including multi-block processing per thread and bitsliced processing, increases arithmetic intensity per thread and can reduce memory pressure when the cipher core is expressed using register-resident operations.

Beyond memory layout tuning, several studies have emphasized instruction-level optimization. For example, PTX-level tuning has been shown to reduce instruction count and improve performance for ARX-style ciphers by leveraging efficient rotate/bitwise sequences and avoiding compiler-introduced overhead [4, 36, 58]. PTX-level implementations have also been reported for lightweight ARX ciphers such as SPECK, SIMON, and SIMECK, demonstrating practical benefits of low-level tuning on CUDA GPUs [27]. These lines of work establish that GPU performance is largely determined by the balance among (i) arithmetic intensity, (ii) memory-access behavior, and (iii) occupancy under register and shared-memory constraints.

### 5.2.2 Bitslicing in Embedded and High-Performance Contexts

Bitslicing has a long history as a constant-time implementation paradigm, originally motivated by eliminating secret-dependent memory accesses and improving SIMD utilization. On embedded microcontrollers, bitslicing and fixslicing replace table lookups with Boolean networks composed of word-level operations, achieving both timing-uniform execution and competitive cycles-per-byte performance under tight memory and register constraints [1, 2, 53, 56].

On GPUs, bitslicing has also been explored to avoid LUT-related bottlenecks, particularly for AES [25, 44, 62]. However, high-degree bitslicing on GPUs introduces a platform-specific trade-off: increasing the number of parallel blocks per thread improves computational density,

but it also increases per-thread register usage. Since each SM has a fixed register file shared by active warps, excessive register pressure can reduce occupancy and, in the worst case, trigger register spilling to local memory—both of which can negate the expected throughput gains. As a result, some prior implementations either constrain the bitslicing degree or rely on less flexible key-handling strategies (e.g., hardcoded keys or pre-expanded keys with substantial storage overhead), limiting scalability across workloads.

### 5.2.3 Limitations of Current PRESENT/GIFT GPU Research

Compared to AES, explicit GPU implementations of PRESENT and GIFT are relatively less common in the literature, and many existing approaches remain LUT-centric [36, 39]. For lightweight SPN ciphers, this design choice can introduce several limitations.

First, LUT-based designs can suffer from memory bottlenecks, especially when bit-level permutations are emulated through table accesses. Even when tables are placed in cached memory spaces, divergent access patterns within a warp may reduce effective bandwidth and lead to throughput loss. Second, LUT-based implementations can exhibit timing variability due to memory-system effects and warp-level execution behavior. This is undesirable not only for stable performance measurement but also from a security perspective when constant-time behavior is a design goal.

Third, many GPU studies focus on bulk encryption throughput (e.g., ECB/CTR-style processing) and provide limited coverage of cryptanalytic workloads, such as exhaustive key search. In key search, the dominant overhead is often not only the cipher round function but also the end-to-end pipeline: generating candidate keys/counters, transferring data over the host-device interface, and converting to a representation suitable for the kernel. In particular, in a high-degree bitsliced setting,

generating large key arrays on the CPU and then repacking them on the GPU can become a substantial performance limiter. Existing work has not sufficiently systematized these overheads for PRESENT/GIFT in a GPU-oriented, bitsliced design space [30, 62].

### 5.2.4 Positioning of This Study

This dissertation addresses the above limitations by proposing a PRESENT/GIFT GPU framework that is table-free in the cipher core and explicitly designed for both bulk encryption and exhaustive key search.

**High-degree bitslicing with GPU-aware constraints** : We adopt a high-degree bitsliced representation and analyze it under the key GPU constraint: the trade-off between register usage and occupancy.

**Unified support for bulk encryption and exhaustive key search** : The same kernel design principles are applied to both workloads, avoiding separate "point solutions" for each scenario.

**On-the-fly generation in bitsliced form** : Keys/counters are generated directly within the kernel in a bitsliced representation to reduce host-device transfer overhead and avoid additional packing steps.

**Cross-platform consistency of the implementation paradigm** : By maintaining bitslicing as a unifying paradigm across microcontrollers (low-degree/fixslicing) and GPUs (high-degree bitslicing), we enable a coherent comparison framework across heterogeneous platforms, consistent with the dissertation's broader objective (Chapters 3-4).

Based on this positioning, the next section (Section 5.3) details the proposed bitsliced GPU design, including algorithm-level bitslicing, counter generation for exhaustive key search, and CUDA kernel/memory

organization.

## 5.3 Proposed Bit-Sliced GPU Implementation

We now present the CUDA techniques used to implement high-speed PRESENT-80/128 and GIFT-64/128 on an RTX 3060 GPU. For validation and interoperability, we cross-checked our CUDA implementation against public reference codebases for PRESENT[47] and GIFT[21].

The proposed design follows two principles.

**Algorithm-level bitslicing** : We express the SPN round functions using only word-level Boolean operations (AND, XOR, NOT, shifts, and masks) in a bitsliced state representation. This removes lookup tables from the cipher core and keeps most computations in registers, thereby reducing memory bottlenecks and timing variability.

**CUDA-aware kernel and memory design** : We tune the bitslicing degree (blocks per thread), threads per block, and blocks per grid to match the SIMT execution model and the register/occupancy constraints of Ampere GPUs. Read-only constants are stored in constant memory, while per-thread states and key materials are maintained in registers whenever possible to reduce global-memory traffic.

Sections 5.3.1-5.3.3 describe (1) algorithm-level bitslicing, (2) counter generation for exhaustive key search, and (3) CUDA kernel and memory design.

### 5.3.1 Algorithm-Level Bitslicing

PRESENT and GIFT are 64-bit SPN ciphers that repeatedly apply a

4-bit S-box layer and a bit-permutation layer (pLayer/PermBits). Our implementation restructures these operations to exploit GPU-friendly word-level parallelism.

**Bitsliced state and parallelism** : Since RTX 3060 cores naturally operate on 32-bit words, we configure each CUDA thread to process 32 independent 64-bit blocks in parallel using a high-degree bitsliced representation. Concretely, we transpose the 64×32-bit state into 64 word registers, where each 32-bit word corresponds to a single bit position across 32 blocks. Under this mapping, the i-th bit of a word represents the same state bit across the i-th block lane, so a single Boolean instruction updates that bit position for 32 blocks simultaneously.

Accordingly, Figure 5-2 depicts the 32-way bitsliced layout in which each 32-bit register holds one bit-plane over 32 blocks.

| | ***Block0*** | ***Block1*** | ***Block2*** | ***Block3*** | ⋯ | ⋯ | ***Block28*** | ***Block29*** | ***Block30*** | ***Block31*** |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_0$ | $b_0^0$ | $b_0^1$ | $b_0^2$ | $b_0^3$ | ⋯ | ⋯ | $b_0^{28}$ | $b_0^{29}$ | $b_0^{30}$ | $b_0^{31}$ |
| $R_1$ | $b_1^0$ | $b_1^1$ | $b_1^2$ | $b_1^3$ | ⋯ | ⋯ | $b_1^{28}$ | $b_1^{29}$ | $b_1^{30}$ | $b_1^{31}$ |
| $R_2$ | $b_2^0$ | $b_2^1$ | $b_2^2$ | $b_2^3$ | ⋯ | ⋯ | $b_2^{28}$ | $b_2^{29}$ | $b_2^{30}$ | $b_2^{31}$ |
| $R_3$ | $b_3^0$ | $b_3^1$ | $b_3^2$ | $b_3^3$ | ⋯ | ⋯ | $b_3^{28}$ | $b_3^{29}$ | $b_3^{30}$ | $b_3^{31}$ |
| $R_4$ | $b_4^0$ | $b_4^1$ | $b_4^2$ | $b_4^3$ | ⋯ | ⋯ | $b_4^{28}$ | $b_4^{29}$ | $b_4^{30}$ | $b_4^{31}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $R_{63}$ | $b_{63}^0$ | $b_{63}^1$ | $b_{63}^2$ | $b_{63}^3$ | ⋯ | ⋯ | $b_{63}^{28}$ | $b_{63}^{29}$ | $b_{63}^{30}$ | $b_{63}^{31}$ |

Figure 5-2. Bitsliced representation using 32-bit registers ($R_0,\ldots,R_{63}$) to process 32 blocks ($b^0,\ldots b^{31}$) in parallel; ($b_j^i$) denotes the $j$-th bit of the $i$-th block.

This representation is the basis for both bulk encryption and exhaustive key search, since the round function can be expressed as a fixed sequence of Boolean operations over these bit-planes.

**Branch-free S-box circuits** : The 4-bit S-box layers of PRESENT and GIFT are implemented as combinational Boolean circuits composed of AND/XOR/NOT operations rather than lookup tables. For PRESENT, we adopt a compact Boolean network (e.g., a small number of bitwise

operations per nibble) and apply it directly to the 32-bit bit-planes so that 32 S-box instances are computed in parallel. The Boolean description of the PRESENT S-box used in our implementation is shown in Figure 5-3. GIFT's S-box is implemented similarly as a Boolean network evaluated over bit-planes. The corresponding Boolean description of the GIFT S-box is shown in Figure 5-4.

This design has two benefits. First, it is naturally constant-time because it avoids secret-dependent memory accesses. Second, it reduces pressure on the GPU memory subsystem by keeping the nonlinear layer entirely register-resident.

**Input:** $x0, x1, x2, x3$
**Output:** $x0, x1, x2, x3$

| | | |
|---|---|---|
| 1: $T1 = x2 \oplus x1$ | 6: $T1 = T1 \oplus T5$ | 11: $T2 = T2 \oplus x3$ |
| 2: $T2 = x1 \wedge T1$ | 7: $T2 = T2 \oplus x1$ | 12: $x0 = x2 \oplus T2$ |
| 3: $T3 = x0 \oplus T2$ | 8: $T4 = x3 \vee T2$ | 13: $T2 = T2 \vee T1$ |
| 4: $T5 = x3 \oplus T3$ | 9: $x2 = T1 \oplus T4$ | 14: $x1 = T3 \oplus T2$ |
| 5: $T2 = T1 \wedge T3$ | 10: $x3 = \sim x3$ | 15: $x3 = T5$ |

Figure 5-3. PRESENT S-box

**Input:** x3, x2, x1, x0
**Output:** x3, x2, x1, x0

| | |
|---|---|
| 1: $x1 = (x0 \wedge x2) \oplus x1$ | 4: $x1 = x1 \oplus x3$ |
| 2: $x0 = (x1 \wedge x3) \oplus x2$ | 5: $x3 = \sim x3$ |
| 3: $x3 = x3 \oplus x2$ | 6: $x2 = (x0 \wedge x1) \oplus x2$ |

Figure 5-4. GIFT S-box

**Permutation layer (pLayer/PermBits) in bitsliced form** : The bit-permutation layers (PRESENT pLayer and GIFT PermBits) are costly in conventional software because they require fine-grained bit shuffles. In a bitsliced representation, however, each bit occupies a fixed position in a word register, and permutations can be expressed using fixed patterns of shifts, masks, and bit-exchange primitives (e.g., SWAPMOVE-style operations).

For PRESENT, we decompose the pLayer into two partial permutations, P0 and P1, and alternate between them across rounds. This reduces the effective cost of each permutation step by reusing structural regularity. For GIFT, we exploit the round-function structure to co-optimize the S-box and permutation steps by rearranging the computation order so that redundant bit moves are minimized.

**On-the-fly round-key generation** : In high-degree bitslicing, storing pre-expanded round keys for all blocks can significantly increase register usage and/or global-memory traffic, which reduces occupancy and can degrade throughput. We therefore avoid pre-expanding round keys and instead compute round keys on the fly within each thread.

For PRESENT-80, we maintain an 80-bit key state in bitsliced form and apply the key-update transformations (rotation, S-box application, and round-constant addition) directly to the bitsliced key state each round. GIFT is handled similarly by keeping the key state in bitsliced form and extracting the required round key material per round. Because lightweight key schedules are relatively simple, the additional arithmetic overhead is modest, while the reduction in storage overhead is substantial.

Figure 5-5 provides an example device function that updates the bitsliced key state and injects round constants.

```
__device__ void addRoundConstant(uint32_t* X, uint32_t r)
{
    uint32_t GIFT_RC[28] =
    {0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D,
     0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33,
     0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B, 0x16,
     0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B };

    X[ 3] ^= (GIFT_RC[r] & 0x1)*0xFFFFFFFF;
    X[ 7] ^= ((GIFT_RC[r] >> 1) & 0x1)*0xFFFFFFFF;
    X[11] ^= ((GIFT_RC[r] >> 2) & 0x1)*0xFFFFFFFF;
    X[15] ^= ((GIFT_RC[r] >> 3) & 0x1)*0xFFFFFFFF;
    X[19] ^= ((GIFT_RC[r] >> 4) & 0x1)*0xFFFFFFFF;
    X[23] ^= ((GIFT_RC[r] >> 5) & 0x1)*0xFFFFFFFF;
    X[63] ^= 0xFFFFFFFF;
}
```

Figure 5-5. CUDA C/C++ implementation of the modified key-update procedure for bitsliced multi-block encryption, where rotation operations are replaced by data-movement operations and `round_counter` is treated as a 32-bit value.

**Packing/unpacking overhead and choice of the bitslicing degree** : Bitslicing requires packing plaintext blocks into the bitsliced layout and unpacking the output back into the conventional representation. This transposition overhead must be amortized by sufficient intra-thread parallelism.

We therefore select 32 blocks per thread as a practical compromise: smaller degrees underutilize ALUs and make packing/unpacking dominant, whereas larger degrees increase register pressure and may reduce occupancy or induce register spilling. Section 5.4 empirically validates that 32 blocks per thread provides a favorable balance among packing cost, register usage, and throughput on RTX 3060-class GPUs.

### 5.3.2 Counter Generation for Exhaustive Key Search

Exhaustive key search differs from bulk encryption in that each thread repeatedly evaluates the cipher under a stream of candidate keys (or

candidate counters/indices) rather than encrypting an externally provided plaintext stream. A naïve GPU implementation would generate candidate keys on the CPU, transfer them to the GPU, and repack them into bitsliced form inside kernels. In a high-degree bitsliced setting, this approach is inefficient because: host-side key generation and PCIe transfer introduce nontrivial overhead, and repacking scalar keys into bitsliced form adds additional kernel-side cost and increases global-memory traffic.

To avoid these costs, we generate counters (or key indices) directly in bitsliced form inside the GPU kernel. Figure 5-6 shows the CUDA C/C++ implementation of the proposed bitsliced counter generation technique.

```
uint32_t tid = blockIdx.x * blockDim.x + threadIdx.x;
uint32_t subkeys[80] = {0,};
subkeys[0] = 0x55555555;
subkeys[1] = 0x33333333;
subkeys[2] = 0x0F0F0F0F;
subkeys[3] = 0x00FF00FF;
subkeys[4] = 0x0000FFFF;

for (uint32_t i = 0; i < 31; i++) {
    subkeys[i+5] = ((tid & (1<<i)) >> i) * 0xFFFFFFFF;
}
```

Figure 5-6. CUDA C/C++ implementation of our bitsliced counter generation technique, which directly produces counters in bitsliced form and avoids costly conversion from the normal representation.

**Thread indexing and assignment** : Each thread computes its global identifier as:

$$tid = blockIdx.x \cdot blockDim.x + threadIdx.x,$$

and uses tid to determine the starting point in the key space (or counter space) assigned to the thread. Under our 32-way bitsliced mapping, each thread constructs 32 candidate indices and evaluates them

in parallel, effectively testing a contiguous range per iteration.

**Bit-pattern templates for 0-31 within a bitsliced lane** : To construct the low bits corresponding to the integers 0, 1, 2, …, 31 across 32 lanes, we use fixed 32-bit masks with repeating bit patterns, such as:

0x55555555 (0101…): toggles every 1 bit
0x33333333 (0011…): toggles every 2 bits
0x0F0F0F0F (00001111…): toggles every 4 bits
0x00FF00FF: toggles every 8 bits
0x0000FFFF: toggles every 16 bits

**Constant-time and SIMT-friendly behavior** : These patterns encode the lane-wise values of the lowest 5 bits across the 32 candidate indices. Higher-order bits are derived from the thread's starting index (based on tid) and are injected by broadcasting the corresponding scalar bits into the relevant bit-planes. With this construction, the kernel produces candidate counters/indices in bitsliced form without an explicit scalar-to-bitsliced conversion stage.

The counter-generation procedure is composed only of fixed integer/bitwise operations and does not require secret-dependent branches or lookups. Moreover, even when a match condition is checked during key search, threads do not terminate early; instead, match flags/results are recorded in a divergence-minimizing manner so that warps execute uniform instruction sequences. This prevents warp divergence from dominating runtime and improves stability of throughput measurements.

As a result, key search workloads avoid both host-device key transfer overhead and unnecessary packing overhead, allowing the GPU kernels to

spend most cycles on the cipher's round function itself.

### 5.3.3 CUDA Kernel and Memory Design

This section summarizes how algorithm-level bitslicing (Section 5.3.1) and bitsliced counter generation (Section 5.3.2) are mapped to efficient CUDA kernels on RTX 3060.

**Grid/block configuration and occupancy considerations** : We fix the bitslicing degree at 32 blocks per thread and explore threads-per-block and blocks-per-grid configurations to maximize throughput. The optimal configuration depends on register usage per thread: higher register usage reduces the number of active warps per SM, decreasing occupancy and weakening latency hiding. We therefore tune launch parameters to maintain high occupancy without inducing register spilling.

**Use of the memory hierarchy** :

Registers - Bitsliced states, key states, and intermediate variables are kept in registers whenever possible to minimize memory traffic and keep the SPN computation ALU-bound.

Constant memory - Round constants and fixed bit-pattern templates used for bitsliced counter construction are stored in constant memory, enabling warp-uniform broadcast loads.

Shared memory - Although shared memory can be used to cache expanded keys or intermediate states, we minimize its usage in the final design because (i) on-the-fly key scheduling reduces the need for large shared tables, and (ii) excessive shared-memory allocation can reduce occupancy and harm throughput.

**Avoiding warp divergence** : PRESENT and GIFT have regular round structures. After bitslicing, all threads can execute the same instruction

sequence. We remove data-dependent control flow and implement:

S-box layers as fixed Boolean networks, permutation layers as fixed sequences of shifts/masks/bit exchanges, and key schedules as fixed update sequences.

For exhaustive key search, candidate evaluation and match recording are written in a SIMT-friendly way to avoid divergence from early termination.

**Host-device transfer considerations** : For bulk encryption, end-to-end throughput depends on both kernel execution and host-device transfers of plaintext/ciphertext. Our experimental methodology in Section 5.4 reports throughput in a way that distinguishes cryptanalytic kernels (where keys/counters are generated on-device) from bulk encryption (where input/output transfers must be included for realistic system evaluation).

## 5.4 Experimental Evaluation

This section evaluates the proposed bitsliced GPU implementations of PRESENT-80/128 and GIFT-64/128 on an NVIDIA RTX 3060 (Ampere architecture). The evaluation is designed to validate the key design choices in Section 5.3—most importantly, the selection of the bitslicing degree—and to quantify throughput under two representative workloads: exhaustive key search and bulk encryption.

Throughout this section, throughput is reported in gigabits per second (Gbit/s) and computed as:

$$\text{Throughput} = \text{total processed bits} / \text{measured runtime}$$

For bulk encryption, the reported runtime includes both kernel execution and host-device transfers (plaintext input and ciphertext output). For exhaustive key search, we r
eport kernel-only runtime because keys/counters are generated on the

GPU and there is no need to transfer key arrays from the host.

### 5.4.1 Evaluation Setup

All kernels were compiled using NVIDIA CUDA and executed on an RTX 3060 GPU. For each cipher and workload, we explored multiple CUDA launch configurations by varying: the number of threads per block, and the number of blocks per grid, while controlling the bitslicing degree (blocks processed per thread).

As discussed in Section 5.3.1, the bitslicing degree directly affects the performance trade-off between (i) packing/unpacking overhead and (ii) per-thread arithmetic density under register/occupancy constraints. Empirically, we observed the following general behavior:

With smaller bitslicing degrees, the round function does not sufficiently amortize packing/unpacking overhead, and ALU utilization is reduced.

With larger bitslicing degrees, register pressure increases and occupancy may drop, eventually degrading throughput due to reduced latency hiding and/or register spilling.

Based on this trade-off, all results reported in the remainder of this section use 32 blocks per thread as the default configuration, since it provides a favorable balance between transposition overhead, register usage, and occupancy on the RTX 3060.

### 5.4.2 Exhaustive Key Search

We first measure the exhaustive key search kernels, where each thread tests 32 candidate keys (or key indices) in parallel using the bitsliced counter-generation technique described in Section 5.3.2. Unlike a naïve design that generates a large key array on the host and transfers

it over PCIe, our approach constructs candidate counters/indices directly in bitsliced form on the device. This eliminates: host-side key generation for large batches, host-device transfer of key arrays, and an additional scalar-to-bitsliced conversion step inside the kernel.

Table 5-1 summarizes the peak throughputs achieved for PRESENT-80/128 and GIFT-64/128. Across the evaluated configurations, all four exhaustive-search kernels achieve throughputs in the range of several hundred Gbit/s, indicating that the proposed kernels can effectively utilize the GPU's parallel compute resources when the workload is compute-dominant and not constrained by PCIe transfers.

| Cipher | Blocks per Grid | Threads per Block | Throughput |
|---|---|---|---|
| PRESENT-80 | 131,072 | 192 | 553.932 |
| PRESENT-128 | 131,072 | 128 | 529.952 |
| GIFT-64 | 131,072 | 64 | 583.859 |
| GIFT-128 | 16,384 | 160 | 214.284 |

Table 5-1. Peak throughput achieved by the exhaustive-search variant of the proposed technique, expressed in gigabits per second (Gbps).

### 5.4.3 Bulk Encryption and Cross-Platform Comparison

Next, we evaluate the encryption-only variants of our kernels in ECB mode. In this bulk-encryption setting, plaintext and ciphertext must be transferred between the host and the GPU, so end-to-end throughput reflects both: GPU kernel execution time, and host-device transfer overhead.

Table 5-3 lists the maximum throughputs obtained for PRESENT-80/128 and GIFT-64/128 under the best-performing grid/block configurations for each cipher. Even when transfer costs are included, the GPU implementations still provide high throughput, confirming that the proposed bitsliced kernels remain practical not only

for cryptanalytic workloads (key search) but also for bulk encryption. To contextualize the results, Table 5-2 compares the GPU throughput with optimized CPU implementations reported in prior work and with representative GPU implementations in the literature[21, 47]. The comparison highlights two points:

Performance scaling via massive parallelism. GPU throughput substantially exceeds CPU throughput for these ciphers when the workload can be parallelized at scale.

Table-free cipher core. Unlike many LUT-centric GPU implementations, our PRESENT/GIFT kernels implement the S-box and permutation layers as fixed Boolean/bitwise sequences in a bitsliced representation, avoiding data-dependent lookup behavior in the cipher core.

Together, these results support the main claim of this chapter: by combining high-degree bitslicing (32 blocks per thread), on-the-fly key handling, and GPU-aware kernel design, PRESENT and GIFT can be implemented efficiently on RTX 3060-class GPUs for both bulk encryption and exhaustive key search.

| Cipher | Blocks per Grid | Threads per Block | Throughput |
|---|---|---|---|
| PRESENT-80 | 32,768 | 128 | 24.264 |
| PRESENT-128 | 32,768 | 64 | 24.522 |
| GIFT-64 | 32,768 | 32 | 85.283 |
| GIFT-128 | 32,768 | 96 | 10.723 |

Table 5-2. Maximum throughput achieved by the encryption-only variants of the proposed scheme, measured in gigabits per second (Gbps).

| Cipher | Ref. | Model | Throughput |
|---|---|---|---|
| PRESENT-80 ECB | [20] | intel i7-9750H | 0.001 |
| PRESENT-80 ECB | [12] | Tesla V100 | 14.15 |
| PRESENT-80 ECB using multiple stream | [12] | Tesla V100 | 24.525 |
| PRESENT-80 ECB | this work | RTX 3060 | 24.264 |
| PRESENT-80 CTR | [10] | RTX 3070 | 115.73 |
| PRESENT-128 ECB | [12] | Tesla V100 | 14.15 |
| PRESENT-128 ECB using multiple stream | [12] | Tesla V100 | 24.525 |
| PRESENT-128 ECB | this work | RTX 3060 | 24.522 |
| GIFT-64 ECB | [21] | intel i7-9750H | 0.003 |
| GIFT-64 ECB | this work | RTX 3060 | 85.283 |
| GIFT-128 ECB | [21] | intel i7-9750H | 0.014 |
| GIFT-128 ECB | this work | RTX 3060 | 10.723 |
| PRESENT-80 EXHAUSTIVE | [10] | RTX 3070 | 120.64 |
| PRESENT-80 EXHAUSTIVE | this work | RTX 3060 | 553.932 |

Table 5-3. Comparison of implementation performance on CPU and GPU platforms.

# Ⅵ. Discussion

This chapter synthesizes the results of the three case studies in Chapters 3–5 and revisits the research objectives introduced in Chapter 1. The overarching aim of this dissertation is to understand how bitslicing and Fixslicing behave across heterogeneous platforms and to distill design principles that remain valid under fundamentally different architectural constraints.

Three questions guide this discussion.

First, when bitslicing or Fixslicing is applied across different block ciphers and platforms, what recurring implementation patterns emerge?

Second, under platform-specific constraints (register file size, memory hierarchy, and instruction-set features), how should the state representation and bitslice degree be selected?

Third, under an explicit constant-time requirement, which implementation options remain practically viable, and what trade-offs do they impose?

To answer these questions, this dissertation examined SPEEDY on 32-bit microcontrollers (Chapter 3), AES-GCM on ARM Cortex-M4 (Chapter 4), and PRESENT/GIFT on an NVIDIA RTX 3060 GPU (Chapter 5). The remainder of this chapter summarizes the case studies (Section 6.1), identifies common design patterns (Section 6.2), analyzes platform-dependent trade-offs (Section 6.3), discusses constant-time behavior versus performance (Section 6.4), and distills cross-platform guidelines (Section 6.5).

## 6.1 Summary of the Case Studies

Case Study I: SPEEDY on Cortex-M3 and RV32I (Chapter 3).
Chapter 3 studied ultra-low-latency instances of SPEEDY-192 on

32-bit microcontrollers, specifically ARM Cortex-M3 and an RV32I-based RISC-V core. SPEEDY's design choices—192-bit blocks, 6-bit S-boxes, and a diffusion layer expressed through structured bit shuffles—are well matched to hardware but can be structurally inefficient in conventional byte-oriented software, where frequent bit extraction and recombination dominate the instruction mix. To address this mismatch, Chapter 3 introduced a 6×32 bitsliced state representation, decomposing the state into six bit-planes stored in 32-bit registers. A multi-stage packing procedure (based on SWAPMOVE and platform-specific bit-manipulation instructions) converts scalar inputs into the bitsliced domain, after which the round function is executed primarily as register-resident bitwise logic.

On Cortex-M3, the resulting SPEEDY implementation achieves competitive cycles-per-byte results compared to constant-time AES-128 and GIFT-128 baselines, while maintaining a constant-time structure by avoiding secret-dependent branches and table lookups. On RV32I, the same representation remains effective, although the lack of native rotate instructions changes the instruction-level cost profile for the linear layer.

Case Study II: AES-GCM on Cortex-M4 (Chapter 4).

Chapter 4 focused on AES-GCM on ARM Cortex-M4, a setting where high performance is challenging because the platform lacks AES-specific instructions and carry-less multiplication[23] for GHASH. Table-based AES and GHASH implementations can be either slow (due to flash/SRAM access costs) or prone to timing leakage (due to secret-dependent memory indices). The chapter therefore adopted a 2-way Fixsliced AES-CTR core that processes two blocks in parallel within a fixed register layout. In this design, SubBytes is realized as a Boolean network and ShiftRows is absorbed into a round-dependent set of MixColumns variants, thereby eliminating S-box/T-table lookups in the core AES

path.

Building on this constant-time AES-CTR core, the chapter adapted FACE caching techniques to the Fixsliced domain, exploiting redundancy among consecutive counter blocks to accelerate CTR encryption for long messages. For GHASH, two alternatives were evaluated: a compact 4-bit table method (faster but not strictly constant-time due to secret-derived indices) and a Karatsuba-based table-free method (strictly constant-time but with a clear performance penalty). The resulting chapter contribution is not only an optimized implementation, but an explicit mapping of the performance-security design space for AES-GCM on Cortex-M4.

Case Study III: PRESENT/GIFT on RTX 3060 GPU (Chapter 5).

Chapter 5 investigated high-throughput implementations of PRESENT-80/128 and GIFT-64/128 on an NVIDIA RTX 3060 GPU. Unlike microcontrollers, GPUs provide massive thread-level parallelism but impose constraints tied to per-thread register usage, occupancy, and warp-level execution. The core design choice is to use high-degree bitslicing, where each CUDA thread processes multiple independent blocks simultaneously. Specifically, the implementation maps many 64-bit blocks into bit-planes aligned with 32-bit words, enabling Boolean operations to update many blocks in parallel. The 4-bit S-box layers are implemented as branch-free Boolean circuits, and the bit-permutation layers are realized using fixed patterns of shifts and bit-move operations rather than lookup tables.

A key distinction in this chapter is the treatment of exhaustive key search. Instead of generating candidate keys/counters on the CPU and transferring them to the GPU, the kernel constructs candidates directly in a bitsliced form on the device. This removes host-device transfer overhead for key material and avoids additional packing stages, enabling substantially higher throughput in cryptanalytic workloads than in bulk

encryption workloads that must include I/O transfers.

Taken together, the three case studies cover different algorithms (SPEEDY, AES, PRESENT/GIFT), different platforms (Cortex-M3, Cortex-M4, RV32I, RTX-class GPU), and different workload types (encryption-focused vs. AEAD vs. key-search-oriented cryptanalysis). This breadth provides a basis for extracting cross-platform design patterns rather than cipher-specific "point solutions."

## 6.2 Common Design Patterns

Despite the diversity of algorithms and platforms, the case studies repeatedly converge on a small number of implementation patterns.

(1) State representation is the primary design decision.

In every case study, performance and constant-time properties are driven first by how the internal state is mapped onto machine words and registers. SPEEDY uses a 6×32 bitsliced mapping; Fixsliced AES uses a 2-way interleaving across eight 32-bit registers; PRESENT/GIFT on GPU uses bit-plane mappings aligned to 32-bit word operations. Once this mapping is fixed, the remaining design work becomes an exercise in expressing the round function as transformations that preserve (or systematically update) this mapping.

(2) Replace data-dependent tables with Boolean networks.

Across all platforms, lookup-table S-boxes are systematically replaced by Boolean circuits expressed via AND/XOR/NOT (and a small set of platform-friendly primitives). This has two benefits that appear consistently across the studies: it removes secret-dependent memory access patterns (a major source of timing leakage), and it shifts the bottleneck from memory access to register-resident ALU operations,

which is favorable on both microcontrollers (where flash/SRAM accesses are expensive) and GPUs (where table lookups can cause bandwidth pressure and uncoalesced accesses).

(3) Treat packing/unpacking as an amortization problem.

Bitslicing and Fixslicing introduce overhead for converting between the conventional layout and the bitsliced layout. In the microcontroller cases, the bitslice degree is constrained, so packing cost must be kept small and predictable. In the GPU case, the per-thread bitslice degree is selected precisely to amortize packing/unpacking while avoiding register pressure that would reduce occupancy. The common design lesson is that conversion overhead must be evaluated jointly with the intended workload size (short messages vs. long streams vs. repeated trials in key search).

(4) Prefer compact, on-the-fly key handling over large precomputation.

A recurring strategy is to avoid large key schedules stored as tables (or stored across too many registers). SPEEDY and PRESENT/GIFT maintain compact key states updated round-by-round, while AES-CTR integrates keys in a Fixsliced-friendly way and avoids AES T-tables. This pattern reduces both memory footprint and leakage risk, and it also helps manage register pressure on platforms where register allocation directly constrains throughput (notably GPUs).

## 6.3 Platform-Dependent Trade-offs

While the design patterns above recur, the optimal parameters and micro-optimizations are strongly platform-dependent.

**Microcontrollers (Cortex-M3/M4, RV32I):** registers and instruction set details dominate.

On microcontrollers, the limiting factors are the size of the register file, the cost of loads/stores, and the availability of rotate/shift fusion. Cortex-M devices benefit from barrel-shifter behavior and instruction forms that can fold shifts/rotations into arithmetic/logic operations, which reduces the instruction count in diffusion layers. RV32I, while often having a larger register file, lacks native rotate operations, increasing the cost of rotation-heavy linear layers. In this regime, bitslice degrees are typically bounded by spill avoidance: designs that increase parallelism but trigger stack traffic often lose both performance and predictability.

**Cortex-M4 AES-GCM:** flash budget becomes a first-class tuning axis.

For Fixsliced AES-GCM, the bitslice degree is largely fixed by the chosen Fixsliced layout (2-way), so the primary tuning dimensions shift to (i) cache/table footprint and (ii) GHASH strategy. FACE variants demonstrate an explicit flash/performance trade-off: larger caches can reduce cycles-per-byte for long messages but are not always justified in memory-constrained deployments. Similarly, GHASH exposes a clear security/performance choice between a faster table-based multiplier and a slower table-free constant-time multiplier.

**GPUs:** register pressure and occupancy shape the feasible bitslice degree.

On GPUs, increasing the bitslice degree per thread raises arithmetic intensity and reduces the relative impact of packing/unpacking. However, it also increases per-thread register usage, which can reduce occupancy and harm latency hiding. As a result, GPU optimization becomes a balancing act among: bitslice degree (blocks per thread), per-thread register count, and launch configuration (threads per block / blocks per grid).

The PRESENT/GIFT case study shows that a carefully chosen high-degree bitslicing strategy can yield excellent performance, particularly when combined with device-side counter generation for

exhaustive key search.

## 6.4 Constant-Time Behavior and Performance

A consistent objective across the dissertation is to approach constant-time behavior as closely as practical, but the case studies show that "constant-time" must be interpreted with platform-aware nuance.

**Avoiding secret-dependent memory access is the baseline requirement.** In SPEEDY (Chapter 3) and PRESENT/GIFT (Chapter 5), the implementations are table-free for S-box and core round operations, avoiding secret-derived indices entirely. In AES-GCM (Chapter 4), the AES-CTR core is table-free and the FACE caches are indexed by public counter values, but GHASH introduces a deliberate split: a fast table-based option and a slower table-free constant-time option. This division makes the security/performance trade-off explicit rather than implicit.

**Control-flow uniformity matters more on SIMT GPUs.** On scalar microcontrollers, eliminating secret-dependent branches is often sufficient to obtain stable execution behavior. On GPUs, warp-level execution creates an additional concern: divergence can serialize execution within a warp and may introduce timing variability. The GPU case study therefore emphasizes uniform control flow even in exhaustive key search: rather than early-exiting on a match, kernels record match flags while continuing a fixed execution pattern. This approach is consistent with SIMT efficiency goals and with timing-uniformity principles.

Overall, the studies suggest that bitslicing/Fixslicing can achieve

constant-time-friendly structures with acceptable overhead, but the remaining trade-offs—especially in GHASH and other auxiliary components—must be evaluated explicitly and not assumed away.

## 6.5 Design Implications and Guidelines

The combined evidence from Chapters 3-5 supports the following cross-platform guidelines.

1) **Start from the state layout, not from the code.** Choose a state representation that aligns with the platform's natural word size and register constraints. Then design S-box and diffusion implementations that preserve this representation with minimal bit movement.

2) **Treat the bitslice degree as a resource-constrained parameter.** On microcontrollers, the ceiling is often set by spill avoidance and code-size limits. On GPUs, the ceiling is set by register pressure and occupancy. The best degree is typically the smallest value that saturates ALU utilization without triggering spills or occupancy collapse.

3) **Prefer Boolean S-boxes and table-free diffusion under constant-time goals.** This eliminates secret-dependent memory indices and often improves performance on memory-constrained systems by shifting work to registers/ALUs.

4) **Use on-the-fly key updates when they reduce footprint and pressure.** Lightweight schedules (e.g., PRESENT/GIFT) are often cheaper to compute than to store at scale, especially in high-degree GPU bitslicing where per-thread storage is costly.

5) **Make security/performance trade-offs explicit for auxiliary components.** GHASH in AES-GCM is the canonical example: provide (and document) both a high-throughput table-based design and a strict constant-time table-free design, and clarify the threat model under

which each is appropriate.

## 6.6 Concluding Remarks

This chapter provided a cross-platform discussion of bitsliced and Fixsliced block cipher implementations spanning resource-constrained microcontrollers and high-throughput GPUs. Across the case studies, a consistent theme emerged: once the state representation is chosen to match the platform, core operations can be expressed as fixed sequences of bitwise instructions that are both efficient and well aligned with constant-time programming goals.

While concrete optimizations differ (e.g., rotate fusion on Cortex-M vs. occupancy constraints on GPUs), the underlying design methodology—state-centric layout, Boolean S-boxes, structured diffusion, and compact key handling—generalizes beyond the specific ciphers studied here. The next chapter concludes the dissertation by summarizing contributions and outlining directions for future work, including extensions to additional algorithms and platforms and deeper investigation of microarchitectural side channels in heterogeneous settings.

# Ⅶ. Conclusion

This dissertation examined how bitsliced and Fixsliced software implementations of block ciphers behave across heterogeneous platforms, spanning resource-constrained microcontrollers and high-throughput GPUs. The central motivation is that modern cryptographic systems are increasingly deployed in environments where low-end embedded devices and high-end parallel processors coexist, yet are often expected to share common primitives and modes. In such settings, overall security and performance are determined not only by algorithmic design but also by whether implementations can satisfy platform constraints (e.g., limited registers and memory on microcontrollers, occupancy and register pressure on GPUs) while preserving timing-uniform execution behavior to mitigate timing side-channel risks.

The study was organized around three case studies that collectively expose the design space of bitslicing across fundamentally different architectures. First, we investigated SPEEDY-5/6/7-192 on ARM Cortex-M3 and RV32I-based RISC-V microcontrollers. Although SPEEDY was designed for ultra-low-latency hardware with a 192-bit state arranged as 32 rows of 6-bit cells, we showed that this structure can be mapped efficiently to 32-bit software through a carefully chosen 6×32 bitsliced representation. By converting scalar inputs into six bit-planes via SWAPMOVE-based packing and by realizing SubBox, ShiftColumns, and MixColumns as fixed sequences of register-resident Boolean operations and rotations, the resulting implementations achieve constant-time behavior without lookup tables or secret-dependent branches, and attain cycles-per-byte performance that is competitive with (and in some settings superior to) established constant-time implementations of AES and GIFT on the same microcontrollers. Second,

we focused on AES-GCM on ARM Cortex-M4, where the absence of AES-specific instructions and carry-less multiplication makes traditional table-based implementations either slow or vulnerable to timing leakage. Using a 2-way Fixsliced AES-CTR core, we integrated FACE-style caching directly in the Fixsliced domain and evaluated two GHASH realizations: a compact 4-bit table approach emphasizing speed and a table-free Karatsuba-based approach emphasizing strict constant-time behavior. The results quantify how FACE improves throughput on long messages at the cost of additional flash, and how GHASH dominates end-to-end AEAD cost on platforms without carry-less multiplication, thereby making the performance-security trade-off explicit at the system level. Third, we investigated GPU implementations of PRESENT and GIFT on an NVIDIA RTX 3060, where throughput depends on the balance between high-degree bitslicing, per-thread register pressure, and occupancy. By adopting a high-order bitsliced state layout and expressing S-boxes and permutation layers as branch-free Boolean operations over bit-planes, we achieved high throughput while avoiding data-dependent table accesses. In addition, by generating counters (and key-search inputs) directly in bitsliced form on the device, exhaustive key-search workloads avoid host-side key-array generation and large host-device transfers, improving end-to-end efficiency in cryptanalytic settings.

Across these studies, several cross-platform conclusions emerge. Most importantly, the choice of state representation—how algorithmic state bits are mapped onto machine words and registers—acts as the primary design decision in bitsliced and Fixsliced implementations: once the mapping is fixed, the round function can be systematically expressed as word-level Boolean operations and fixed permutations, and the remaining optimization task reduces to controlling conversion overhead and platform-specific resource pressure. A second consistent theme is

that replacing lookup-table S-boxes and table-driven permutations with Boolean networks and fixed instruction sequences not only supports constant-time execution by construction (eliminating secret-dependent memory access and control flow), but can also deliver competitive performance when the bit-slice degree and register usage are chosen appropriately for the target architecture. Finally, the results highlight that the "right" parallelism differs by platform: microcontrollers are constrained primarily by the number of registers and the availability of efficient rotate/shift mechanisms, whereas GPUs are constrained by register pressure and occupancy; therefore, selecting the bit-slice degree is best treated as a platform-dependent tuning problem rather than a fixed design choice.

This dissertation also has limitations. The empirical evaluation is restricted to a specific set of platforms (Cortex-M3/M4, RV32I, and RTX 3060), and constant-time behavior is argued primarily through control-flow and memory-access structure rather than through direct measurements under timing, power, or electromagnetic leakage models. Extending the evaluation to additional architectures—such as CPUs with cryptographic extensions (e.g., AES/PMULL on ARMv8-A, AES-NI[55]/PCLMULQDQ on x86[28]) and emerging RISC-V cryptography extensions—would further test generality and clarify how the state-centric bitslicing methodology interacts with hardware acceleration. In addition, extending the approach to a broader range of primitives and AEAD constructions would help validate whether the derived guidelines transfer cleanly beyond the specific ciphers and mode studied here. These observations motivate several directions for future work: expanding platform coverage, conducting empirical side-channel studies that include subtle microarchitectural effects on both MCUs and GPUs, developing tool support for state-layout selection and

Boolean-network synthesis under register/occupancy constraints, and integrating robust bitsliced/Fixsliced implementations into real-world libraries and protocol stacks while preserving constant-time properties across compilation and deployment environments.

In conclusion, the dissertation demonstrates that bitslicing and Fixslicing form a robust and practical implementation paradigm for block ciphers in heterogeneous computing environments. By treating state representation as a first-class design parameter and by realizing nonlinear and linear layers as fixed sequences of word-level Boolean operations, it is possible to build implementations that are simultaneously efficient and aligned with constant-time programming goals on platforms as different as Cortex-M microcontrollers, RV32I-based RISC-V cores, and RTX-class GPUs. It is hoped that the case studies, measurements, and cross-platform design principles presented here will provide a useful foundation for further research and for secure, high-performance cryptographic software in increasingly heterogeneous deployment settings.

# 참 고 문 헌

[1] Adomnicai, A., Najm, Z. and Peyrin, T., "Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M," IACR Trans. Cryptogr. Hardw. Embed. Syst., 2020, pp. 402-427.

[2] Adomnicai, A. and Peyrin, T., "Fixslicing AES-Like Ciphers: New Bitsliced AES Speed Records on ARM-Cortex M and RISC-V," IACR Trans. Cryptogr. Hardw. Embed. Syst., 2021, pp. 402-425.

[3] An, S. and Seo, S. C., "Efficient Parallel Implementations of LWE-Based Post-Quantum Cryptosystems on Graphics Processing Units," Mathematics, vol. 8, 1781, 2020.

[4] An, S. and Seo, S. C., "Highly Efficient Implementation of Block Ciphers on Graphic Processing Units for Massively Large Data," Appl. Sci., vol. 10, 3711, 2020.

[5] Arm Ltd., ARMv7-M Architecture Reference Manual, ARM DDI 0403E.b, 2014.

[6] Asanovic, K. and Waterman, A., The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, RISC-V Foundation, 2019.

[7] Avanzi, R., "The QARMA Block Cipher Family: Almost MDS Matrices over Rings with Zero Divisors, Nearly Symmetric Even-Mansour Constructions with Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes," IACR Trans. Symmetric Cryptol., 2017, pp. 4-44.

[8] Banik, S. et al., "GIFT: A Small PRESENT," in Proc. CHES 2017, Taipei, Taiwan, 2017, pp. 321-345.

[9] Bao, Z., Luo, P. and Lin, D., "Bitsliced Implementations of the PRINCE, LED and RECTANGLE Block Ciphers on AVR 8-bit Microcontrollers," in Proc. ICICS 2015, Beijing, China, 2015, pp. 18-36.

[10] Baugher, M., McGrew, D., Naslund, M., Carrara, E. and Norrman, K., “The Secure Real-time Transport Protocol (SRTP),” IETF RFC 3711, 2004.
[11] Beierle, C. et al., “The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS,” in Advances in Cryptology - CRYPTO 2016, Santa Barbara, CA, USA, 2016, pp. 123-153.
[12] Bernstein, D. J., “Cache-Timing Attacks on AES,” Tech. Rep., 2005.
[13] Biham, E., “A Fast New DES Implementation in Software,” in Fast Software Encryption - FSE ’97, Haifa, Israel, 1997, pp. 260-272.
[14] Bogdanov, A., Eisenbarth, T., Paar, C. and Wienecke, M., “Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs,” in CT-RSA 2010, San Francisco, CA, USA, 2010, pp. 235-251.
[15] Bogdanov, A. et al., “PRESENT: An Ultra-Lightweight Block Cipher,” in Proc. CHES 2007, Vienna, Austria, 2007, pp. 450-466.
[16] Bozilov, D. et al., “PRINCEv2: More Security for (Almost) No Overhead,” IACR Cryptology ePrint Archive, 2021, Paper 483.
[17] Borghoff, J. et al., “PRINCE—A Low-Latency Block Cipher for Pervasive Computing Applications,” in ASIACRYPT 2012, Beijing, China, 2012, pp. 208-225.
[18] Bonneau, J. and Mironov, I., “Cache-Collision Timing Attacks Against AES,” in CHES 2006, Yokohama, Japan, 2006, pp. 201-215.
[19] Dhanda, S. S., Singh, B., Jindal, P., Kumar, V. and Gupta, S. K., “AES-8: A Lightweight AES for Resource-Constrained IoT Devices,” Trans. Emerg. Telecommun. Technol., vol. 36, no. 3, e70094, 2025.
[20] Ge, Q., Yarom, Y., Cock, D. and Heiser, G., “A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware,” J. Cryptogr. Eng., vol. 8, pp. 1-27, 2018.
[21] giftcipher, “GIFT GitHub Repository,” GitHub, 2020.

[22] Gouvêa, C. P. L. and López, J., “Implementing GCM on ARMv8,” in Topics in Cryptology - CT-RSA 2015, San Francisco, CA, USA, 2015, pp. 167-180.
[23] Gueron, S. and Kounavis, M. E., “Intel® Carry-Less Multiplication Instruction and Its Usage for Computing the GCM Mode,” Intel White Paper, 2010.
[24] Gupta, N., Jati, A., Chauhan, A. K. and Chattopadhyay, A., “PQC Acceleration Using GPUs: FrodoKEM, NewHope, and Kyber,” IEEE Trans. Parallel Distrib. Syst., vol. 32, pp. 575-586, 2021.
[25] Hajihassani, O., Monfared, S. K., Khasteh, S. H. and Gorgin, S., “Fast AES Implementation: A High-Throughput Bitsliced Approach,” IEEE Trans. Parallel Distrib. Syst., vol. 30, no. 10, pp. 2211-2222, 2019.
[26] Han, K., Lee, W.-K. and Hwang, S. O., “cuGimli: Optimized Implementation of the Gimli Authenticated Encryption and Hash Function on GPU for IoT Applications,” Cluster Comput., vol. 25, pp. 433-450, 2022.
[27] Jang, K. B., Kim, H. J., Lim, S. J. and Seo, H. J., “Parallel Implementation of SPECK, SIMON and SIMECK by Using NVIDIA CUDA PTX,” J. Korea Inst. Inf. Secur. Cryptol., vol. 31, pp. 423-431, 2021.
[28] Jankowski, K. and Laurent, P., “Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions,” IEEE Trans. Comput., vol. 60, no. 1, pp. 135-138, 2010.
[29] Käsper, E. and Schwabe, P., “Faster and Timing-Attack Resistant AES-GCM,” in CHES 2009, Lausanne, Switzerland, 2009, pp. 1-17.
[30] Kim, H., Eum, S., Lee, W.-K., Lee, S. and Seo, H., “Secure and Robust Internet of Things with High-Speed Implementation of PRESENT and GIFT Block Ciphers on GPU,” Appl. Sci., vol. 12, no. 20, 10192, 2022.
[31] Kim, H., Eum, S., Sim, M. and Seo, H., “Efficient Implementation of SPEEDY Block Cipher on Cortex-M3 and RISC-V Microcontrollers,”

Mathematics, vol. 10, no. 22, 4236, 2022.
[32] Kim, K., Choi, S., Kwon, H., Kim, H., Liu, Z. and Seo, H., "PAGE—Practical AES-GCM Encryption for Low-End Microcontrollers," Appl. Sci., vol. 10, no. 9, 3131, 2020.
[33] Kim, H., Eum, S., Sim, M. and Seo, H., "speedy_bitslice: Source Code for Bitsliced SPEEDY Implementations on 32-Bit Microcontrollers," GitHub repository.
[34] Kim, H. and Seo, H., "m4-fixslicing-face: Optimized AES-GCM Implementation (Fixslicing + FACE) for ARM Cortex-M4," GitHub repository.
[35] Leander, G., Moos, T., Moradi, A. and Rasoolzadeh, S., "The SPEEDY Family of Block Ciphers: Engineering an Ultra Low-Latency Cipher from Gate Level for Secure Processor Architectures," IACR Trans. Cryptogr. Hardw. Embed. Syst., 2021, pp. 510-545.
[36] Lee, W.-K., Goi, B. M. and Phan, R. C-W., "Terabit Encryption in a Second: Performance Evaluation of Block Ciphers in GPU with Kepler, Maxwell, and Pascal Architectures," Concurrency Comput.: Pract. Exper., vol. 31, e5048, 2019.
[37] Lee, W.-K. and Hwang, S. O., "High Throughput Implementation of Post-Quantum Key Encapsulation and Decapsulation on GPU for Internet of Things Applications," IEEE Trans. Serv. Comput., 2021.
[38] Lee, W.-K., Seo, H. J., Seo, S. C. and Hwang, S. O., "Efficient Implementation of AES-CTR and AES-ECB on GPUs With Applications for High-Speed FrodoKEM and Exhaustive Key Search," IEEE Trans. Circuits Syst. II: Express Briefs, vol. 69, pp. 2962-2966, 2022.
[39] Li, P., Zhou, S., Ren, B., Tang, S., Li, T., Xu, C. and Chen, J., "Efficient Implementation of Lightweight Block Ciphers on Volta and Pascal Architecture," J. Inf. Secur. Appl., vol. 47, pp. 235-245, 2019.
[40] Longo, J., De Mulder, E., Page, D. and Tunstall, M., "SoC It to EM:

Electromagnetic Side-Channel Attacks on a Complex System-on-Chip," in CHES 2015, Saint-Malo, France, 2015, pp. 620-640.
[41] May, L., Penna, L. and Clark, A., "An Implementation of Bitsliced DES on the Pentium MMX," in Proc. ACISP 2000, Brisbane, Australia, 2000, pp. 112-122.
[42] McGrew, D. and Viega, J., "The Galois/Counter Mode of Operation (GCM)," NIST Modes of Operation Proposal, 2004.
[43] Nguyen, H., Ivanov, R., Phan, L. T. X., Sokolsky, O., Weimer, J. and Lee, I., "LogSafe: Secure and Scalable Data Logger for IoT Devices," in Proc. IEEE/ACM Int. Conf. Internet-of-Things Design and Implementation (IoTDI), 2018, pp. 141-152.
[44] Nishikawa, N., Amano, H. and Iwai, K., "Implementation of Bitsliced AES Encryption on CUDA-Enabled GPU," in Proc. NSS 2017, 2017, pp. 273-287.
[45] Papapagiannopoulos, K., "High Throughput in Slices: The Case of PRESENT, PRINCE and KATAN64 Ciphers," in Radio Frequency Identification - Security and Privacy Issues (RFIDSec), Oxford, UK, 2014, pp. 137-155.
[46] Park, J. H. and Lee, D. H., "FACE: Fast AES-CTR Mode Encryption Techniques Based on the Reuse of Repetitive Data," IACR Trans. Cryptogr. Hardw. Embed. Syst., 2018, pp. 469-499.
[47] Pepton21, "PRESENT-Cipher GitHub Repository," GitHub, 2019.
[48] Pornin, T., "BearSSL," project website.
[49] Pornin, T., "GHASH Constant-Time Multiplication Implementation (ghash_ctmul.c)," BearSSL source code, 2016.
[50] Prouff, E. and Rivain, M., "Masking Against Side-Channel Attacks: A Formal Security Proof," in Advances in Cryptology - EUROCRYPT 2013, Athens, Greece, 2013, pp. 142-159.
[51] Randolph, M. and Diehl, W., "Power Side-Channel Attack Analysis:

A Review of 20 Years of Study for the Layman," Cryptography, vol. 4, no. 2, 15, 2020.
[52] Rebeiro, C., Selvakumar, D. and Devi, A. S. L., "Bitslice Implementation of AES," in Cryptology and Network Security (CANS 2006), 2006, pp. 203-212.
[53] Reis, T., Aranha, D. F. and López, J., "PRESENT Runs Fast," in CHES 2017, Taipei, Taiwan, 2017, pp. 644-664.
[54] Rodrigues, C., Oliveira, D. and Pinto, S., "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect," in Proc. 2024 IEEE Symp. Security and Privacy (SP), 2024, pp. 3679-3696.
[55] Rott, J. K., "Intel® Advanced Encryption Standard Instructions (AES-NI)," Intel Technical Article, 2012.
[56] Schwabe, P. and Stoffelen, K., "All the AES You Need on Cortex-M3 and M4," in Selected Areas in Cryptography - SAC 2016, St. John's, Canada, 2016, pp. 180-194.
[57] Selent, D., "Advanced Encryption Standard," Rivier Academic Journal, vol. 6, no. 2, 2010.
[58] Seo, H., Park, T., Heo, S., Seo, G., Bae, B., Hu, Z., Zhou, L., Nogami, Y., Zhu, Y. and Kim, H., "Parallel Implementations of LEA, Revisited," in Int. Workshop on Information Security Applications (WISA), Jeju, Korea, 2016, pp. 318-330.
[59] Singh, S., Sharma, P. K., Moon, S. Y. and Park, J. H., "Advanced Lightweight Encryption Algorithms for IoT Devices: Survey, Challenges and Solutions," J. Ambient Intell. Humaniz. Comput., 2017.
[60] Sovyn, Y., Khoma, V. and Podpora, M., "Comparison of Three CPU-Core Families for IoT Applications in Terms of Security and Performance of AES-GCM," IEEE Internet Things J., vol. 7, no. 1, pp. 339-348, 2020.

[61] Steger, M., Boano, C. A., Niedermayr, T., Karner, M., Hillebrand, J., Romer, K. and Rom, W., “An Efficient and Secure Automotive Wireless Software Update Framework,” IEEE Trans. Ind. Informatics, vol. 14, no. 5, pp. 2181-2193, 2017.
[62] Tezcan, C., “Key Lengths Revisited: GPU-Based Brute Force Cryptanalysis of DES, 3DES, and PRESENT,” J. Syst. Archit., vol. 124, 102402, 2022.
[63] Waterman, A., Lee, Y., Avizienis, R., Cook, H., Patterson, D. and Asanovic, K., “The RISC-V Instruction Set,” in Proc. 2013 IEEE Hot Chips 25 Symposium (HCS), 2013.
[64] National Institute of Standards and Technology, “FIPS PUB 197: Advanced Encryption Standard (AES),” 2001.
[65] National Institute of Standards and Technology, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,” NIST Special Publication 800-38D, 2007.

# 국 문 초 록

## Bitslicing for Block Ciphers on Embedded Microcontrollers and GPUs

### Implementation Techniques and Trade-Offs

한 성 대 학 교 대 학 원
정 보 컴 퓨 터 공 학 과
정 보 시 스 템 공 학 전 공
김 현 준

본 학위논문은 자원 제약이 큰 마이크로컨트롤러부터 고성능 GPU에 이르기까지, 이기종 플랫폼에서 블록암호의 비트슬라이싱(bitslicing) 및 픽스슬라이싱(fixslicing) 기반 소프트웨어 구현이 어떤 특성과 성능을 보이는지 분석하고, 상수시간 실행과 실용적 성능 사이의 트레이드오프를 체계적으로 정리한다. 현대 암호 시스템은 임베디드 장치와 병렬 가속기가 공존하는 환경에서 동일한 알고리즘·동작모드를 공유하는 경우가 많으며, 이때 전체 시스템의 보안성과 성능은 알고리즘 자체의 설계뿐 아니라 구체 하드웨어에서의 구현 선택—특히 상태(state) 레이아웃, 패킹/언패킹 전략, 그리고 비트슬라이스 차수(bitslice degree)—에 의해 크게 좌우된다. 본 논문은 세 가지 사례 연구를 제시한다. 첫째, ARM Cortex-M3 및 RV32I 기반 RISC-V 마이크로컨트롤러에서 SPEEDY-5/6/7-192를 대상으로 6×32 비트슬라이스 상태 표현을 도입하고, 어셈블리 수준 최적화를 통해 구현 효율을 개선한다. 구체

적으로 SWAPMOVE 기반 패킹을 통해 테이블 기반 구현을 제거하고, 6비트 SubBox를 불리언 네트워크로 구현하며, 확산층을 회전(rotate) 중심의 연산으로 매핑하고(가능한 경우 회전-XOR 결합까지 활용) 상수시간 실행 구조를 유지한다. 그 결과 SPEEDY-7-192는 바이트 지향 기준 구현(15,407/18,096 cpb) 대비 Cortex-M3/RV32I에서 각각 85.1/109.2 cpb로 크게 개선되며, 비밀정보에 의존하지 않는 메모리 접근과 고정된 명령 실행 흐름을 달성한다.

둘째, ARM Cortex-M4에서 AES-GCM을 대상으로 2-way 픽스슬라이스 AES-CTR 코어를 기반으로 FACE 계열 캐싱을 픽스슬라이스 도메인 내부에 직접 통합하고, GHASH 구현을 두 설계 지점으로 비교하여 성능-보안(상수시간 엄격성) 트레이드오프를 명확히 한다. 즉, (1) 4-bit 테이블 기반 곱셈(고성능)과 (2) 테이블을 사용하지 않는 Karatsuba 기반 곱셈(상수시간 지향)을 비교한다. FACE는 장문 AES-GCTR에서 최대 19.4%의 성능 향상을 제공하며, GHASH에서는 4-bit 테이블 방식이 Karatsuba 기준 대비 대략 2배 수준의 속도를 보이지만, 해시 서브키/상태에서 유도되는 테이블 인덱스 접근을 사용한다는 점에서 “엄밀한 상수시간” 요구와의 트레이드오프가 존재한다.

셋째, NVIDIA RTX 3060에서 PRESENT 및 GIFT의 고차(high-degree) 비트슬라이싱 GPU 구현을 설계한다. 스레드당 32-way 비트슬라이싱을 적용하고, 분기 없는 불리언 S-box와 효율적인 비트 순열을 구성하며, 카운터/인덱스를 비트슬라이스 형태로 커널 내부에서 직접 생성함으로써 대량 암호화와 전수(키) 탐색 모두를 효율적으로 지원한다. 제안 구현은 전수 탐색에서 214-584 Gbit/s의 피크 처리량을 달성하고, 호스트-디바이스 전송을 포함한 대량 암호화에서는 최대 85 Gbit/s 수준의 처리량을 보인다.

마지막으로 본 논문은 세 사례를 종합하여, 플랫폼 특성(레지스터 제약, 회전 지원 여부, GPU 점유율/레지스터 압박 등)에 따라 상태 표현과 비트슬라이스 차수를 선택하는 실천적 가이드라인을 제시하고, 패킹/언패킹 비용을 상쇄하는 설계 원칙과 상수시간 실행을 위한 구현 패턴을 정리한다.

주요어 – 비트슬라이싱, 픽스슬라이싱, 블록암호 소프트웨어 구현, 상수시간 구현, 타이밍 사이드채널 공격, 임베디드 마이크로컨트롤러, GPU 가속