ADDIS ABABA UNIVERSITY

COLLEGE OF TECHNOLOGY AND BUILT ENVIRONMENT

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Design of Memory Encryption for RISC-V CPU

BY
HAIMANOT TIZAZU

ADVISORS

Dr. FITSUM ASSAMNEW
Dr. SALESSAWI FEREDE

A thesis submitted to the School of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

June, 2025
ADDIS ABABA, ETHIOPIA

Addis Ababa University

College of Technology and Built Environment

School of Electrical and Computer Engineering

Design of Memory Encryption for RISC-V CPU

Haimanot Tizazu

ADVISORS

Dr. Fitsum Asamnew
Dr. Salessawi Ferede

Approval by Boards of Examiners

Dr. Sosina Mengistu	_____	_____
Dean, SECE, CTBE	Date	Signature
Dr. Fitsum Assamnew	_____	_____
Advisor	Date	Signature
Dr. Menore Tekeba	_____	_____
Internal Examiner	Date	Signature
Dr. Bisrat Derebssa	_____	_____
External Examiner	Date	Signature

June, 2025
ADDIS ABABA, ETHIOPIA

Declaration

I, Haimanot Tizazu Bitew, declare that this thesis is my original work. All sources of information in this study have been appropriately acknowledged. I further confirm that this thesis has not been submitted either in part or in full for any other requirements to any other learning institution.

Haimanot Tizazu	_____	_____
Student Name	Signature	Date

June, 2025
ADDIS ABABA, ETHIOPIA

Acknowledgments

I would like to express my sincere gratitude to my advisor, **Dr. Fitsum Asaminew**, for his continuous support, guidance, and encouragement throughout the course of this research. Dr. Fitsum was always generous with his time and provided insightful feedback, whether on technical matters related to my M.Sc. work or broader academic guidance. His patient mentorship has been invaluable to the successful completion of this thesis.

I am also deeply grateful to my co-advisor, **Dr. Salsawi Ferede**,for his invaluable guidance and support throughout this thesis. His expertise in finite fields and his assistance in constructing transformation matrices for basis changes significantly shaped the technical foundation of this work. His support has been both intellectually enriching and transformative to the direction and depth of this thesis.

Abstract

The increasing demand for secure computing in embedded and general-purpose systems has heightened the importance of hardware-based memory protection mechanisms. This thesis investigates the design and implementation of memory encryption techniques tailored for RISC-V processors, with a focus on architectural design choices and performance trade-offs.

This work proposes a lightweight Memory Encryption Unit (MEU) integrated into the memory controller of a RISC-V architecture to ensure the confidentiality of external memory transactions. The MEU is designed to handle six lightweight encryption algorithms: block ciphers (QARMA, PRINCE, SIMON) and stream ciphers (ChaCha,Grain,Trivium), selected for their low latency and hardware efficiency. By embedding the encryption engines directly within the memory controller, the system performs inline encryption and decryption with minimal performance overhead. The design is implemented and tested on Xilinx Arty-7 FPGAs, enabling detailed evaluation across key metrics including throughput, area utilization, throughput-to-area ratio (TP/A), power consumption, and energy efficiency.

The memory encryption architecture was seamlessly integrated into the RISC-V system, introducing a modest storage overhead of 1.5%–4% and an execution overhead of 3%–6%, depending on the encryption algorithm. These results demonstrate that lightweight ciphers can effectively secure memory with minimal impact on system performance and resource utilization.

Keywords: RISC-V, lowRISC, Rockcore, DRAM, Memory Encryption, Memory Encryption Unit, Block Cipher, Stream Cipher,FPGA.

Contents

List of Figures

List of Tables

Listing

List of Acronyms and Abbreviations

AES Advanced Encryption Standard

API Application Programming Interface

BRAM Block Random-Access Memory

CPU Central Processing Unit

CSR Control and Status Register

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

FPGA Field-Programmable Gate Array

HPME High-Performance Memory Encryption

ISA Instruction Set Architecture

IV Initialization Vector

LFSR Linear Feedback Shift Register

MAC Message Authentication Code

MC Memory Controller

MEU Memory Encryption Unit

MITM Meet-In-The-Middle

NFSR Nonlinear Feedback Shift Register

NVDIMM Non-Volatile Dual In-line Memory Module

OS Operating System

PUF Physically Unclonable Function

QoS Quality of Service

RAM Random-Access Memory

RISC Reduced Instruction Set Computer

RISC-V Reduced Instruction Set Computer – Version Five

RTL Register Transfer Level

SoC System-on-Chip

SRAM Static Random-Access Memory

TP/A Throughput-to-Area Ratio

UART Universal Asynchronous Receiver-Transmitter

Chapter 1

Introduction

1.1 Background

The demand for robust security in modern computing systems has reached unprecedented levels. Escalating system complexity, the widespread adoption of cloud computing, and the introduction of emerging technologies have created a computing landscape that is both intricate and increasingly vital to safeguard. Typically, runtime data in computer systems resides in dynamic random-access memory (DRAM), interfaced with the CPU via data and address buses. While this configuration supports efficient operation, it poses significant challenges for systems requiring stringent physical security. Adversaries with physical access to a device can exploit this setup in multiple ways, such as intercepting sensitive data through bus snooping or extracting DRAM contents using external tools, thereby compromising system integrity and confidentiality.

Memory encryption offers a powerful defense against such threats by securing main memory contents. Although data is commonly encrypted when stored on persistent media like disks, it remains unencrypted in DRAM during runtime, leaving it exposed to unauthorized access. This vulnerability can be exploited by malicious software, rogue administrators, or hardware-based probing techniques. The advent of non-volatile dual in-line memory modules (NVDIMMs) further amplifies this risk, as these chips retain data even when removed from a system—akin to a hard drive—making unencrypted sensitive information, passwords, or cryptographic keys easily recoverable without protective measures.

In the context of the RISC-V architecture, an open-source ISA gaining traction across diverse applications, the need for memory encryption becomes even more pressing. RISC-V systems, often deployed in lightweight and embedded environments, lack native encryption mechanisms in their standard designs, rendering them susceptible to these evolving threats. Implementing memory encryption tailored to RISC-V's modular and resource-efficient framework presents a critical opportunity to enhance its security posture, ensuring it meets the demands of modern, attack-prone computing environments while preserving its core advantages.

The development of a Memory Encryption Engine (MEE) for general-purpose processors represents a significant advancement in securing memory against eavesdropping and tampering. This technology, exemplified by Intel's Software Guard Extensions (SGX), integrates cryptographic

protection directly into the processor's architecture, ensuring the confidentiality, integrity, and freshness of data in external memory. The MEE is embedded within the memory controller of general-purpose processors, allowing it to manage memory traffic securely. It employs a combination of cryptographic primitives and an integrity tree structure, which primarily resides in DRAM, to maintain security with minimal internal storage requirements. The MEE has demonstrated effective performance, enabling real-time encryption and decryption processes without significantly impacting overall system efficiency[1].

Modern processors include an internal cache that stores a small amount of data for faster access than main memory. During normal execution, the processor core issues memory transactions, and those that miss the cache are handled by the Memory Controller (MC). To ensure confidentiality and integrity, a Memory Encryption Engine (MEE) is integrated with the MC, managing traffic between the cache and DRAM for a designated protected memory region. An additional seized memory region is allocated to store the MEE's integrity tree. Together, these form the MEE region, a fixed physical address range defined securely at boot time. Read and write operations targeting the protected region are routed through the MEE, which performs encryption and decryption before accessing DRAM. The MEE also autonomously issues transactions to verify or update the integrity tree, which is built using counters and message authentication codes (MACs). These operations involve both the seized DRAM region and an on-chip SRAM array that stores the tree's root node [1].

As computing systems become more advanced and increasingly depend on open hardware platforms, safeguarding data confidentiality and integrity has become a top priority. One critical area of concern is memory security, especially against physical threats like memory snooping and tampering. Memory encryption serves as an effective defense strategy to prevent unauthorized access to sensitive information stored in memory.

RISC-V, a widely adopted open-source Instruction Set Architecture (ISA)[2, 3], is gaining momentum in both academic and industrial sectors due to its flexibility and customizable nature. However, this openness also makes it more vulnerable to security threats, particularly in environments where physical access to the hardware cannot be controlled. To address this, implementing memory encryption within the RISC-V architecture becomes vital for protecting data during operation. Driving its generic nature is the desire to serve all markets, including microcontrollers as well as image, graphics, and server processors. Thus, the ISA must be consistent across microarchitectures, from an in-order scalar design to a heavily out-of-order design. Similarly, it's intended to be suitable for nearly any implementation, from FPGAs to synthesized macros to fully custom layouts. This study focuses on the development and integration of memory encryption mechanisms within RISC-V systems to enhance security with introducing major performance bottlenecks. The process involves encrypting data during memory writes and decrypting it during reads, often using lightweight cryptographic algorithms suited for hardware

implementation. Recent advancements in attack techniques, such as the Cold Boot attacks[4] Rowhammer RAM exploit[5]and timing-based vulnerabilities like Meltdown[6]and Spectre[7], have introduced significant security risks to contemporary computing platforms. These attacks reveal that an adversary with adequate computational resources and physical access can extract confidential data from memory or undermine a system's security controls. For instance, Rowhammer manipulates DRAM to flip bits, while Meltdown and Spectre exploit speculative execution to breach isolation boundaries, collectively jeopardizing memory integrity and confidentiality. Such vulnerabilities are particularly concerning in environments where external memory (e.g., DRAM) is exposed to attackers who can monitor or alter its contents, regardless of the software's privilege level. To counter these threats, cryptographic memory protection has become a cornerstone of secure processor and System-on-Chip (SoC) architectures. Encryption enables systems to execute software reliably, protect sensitive data, and resist malicious interference from external components susceptible to eavesdropping or tampering. This is critical under the premise that off-chip regions are untrusted, granting attackers complete control over DRAM and the executing software. By incorporating encryption, integrity verification, and anti-replay measures, systems can maintain a secure, isolated computing environment. The RISC-V architecture, an open-source ISA increasingly adopted in lightweight SoCs for domains like IoT and embedded applications, offers a unique context for addressing these challenges. While its flexibility and simplicity are strengths, RISC-V lacks built-in memory encryption, leaving it exposed to modern threats. Developing a security solution that aligns with RISC-V's low-cost, low-overhead ethos is thus a pressing need. Memory encryption for RISC-V is an emerging area of research aimed at enhancing security in various applications, particularly in resource-constrained environments like IoT devices. This technology leverages the RISC-V architecture's capabilities to implement robust memory protection mechanisms, ensuring data confidentiality and integrity. The following sections outline key aspects of memory encryption in RISC-V systems.

A memory encryption unit designed for FPGA-based RISC-V systems utilizes the ChaCha stream cipher, providing a flexible and efficient solution for trusted execution environments[8]. The HPME (High-Performance Memory Encryption Engine)[9] integrated into RISC-V SoCs achieves a 41.8% reduction in cycles compared to traditional designs, enhancing safety mechanisms for trusted execution environments like Keystone. A programmable PCI-E encryption system architecture[10] allows for high-speed storage encryption with a throughput of 18.8 Gbps. This system can adapt to various encryption algorithms and secure communication protocols, showcasing the flexibility of RISC-V designs. The design of a memory protection unit (MPU) for RISC-V SoCs focuses on lightweight yet robust security measures, providing integrity and confidentiality while allowing partial encryption based on application needs[11].

While memory encryption significantly enhances security, it also introduces complexity in system design and may impact performance if not optimized effectively. Balancing these factors is crucial for the successful implementation of memory encryption in RISC-V architectures.

1.2 Statement of Problem

Modern RISC-V systems, from embedded devices to high-performance servers, are increasingly vulnerable to hardware-level attacks such as cold boot, memory probing, Meltdown, Spectre, and Rowhammer, which compromise data confidentiality and integrity. While memory scrambling is commonly used in commercial processors [4], it offers little cryptographic protection, leaving sensitive data exposed. The open nature of RISC-V enables the implementation of hardware-based memory encryption, but this requires addressing challenges in performance, key management, and integration with existing security mechanisms.

Existing efforts, such as the SMARTS architecture [11], demonstrate the feasibility of secure memory encryption in RISC-V, but they lack comprehensive defenses against physical attacks and robust evaluation of security features. This underscores the need for an open, efficient, and verifiable memory encryption scheme for RISC-V that ensures strong protection against hardware threats while maintaining performance and scalability.

Therefore, the problem lies in the absence of an open, efficient, and transparent memory encryption framework for RISC-V that simultaneously protects sensitive data and encryption keys in large random access memories against physical, microarchitectural, and fault-injection attacks, while maintaining system performance and scalability.

Refer to the objectives and the problem statement; the following research question will be addressed:

1. **RQ1:** Which implementation and evaluation strategies can enable effective memory encryption for RISC-V, ensuring confidentiality while preserving performance and compatibility?
2. **RQ2:** What techniques can securely and efficiently integrate key management into the RISC-V execution environment without introducing performance bottlenecks?
3. **RQ3:** Which lightweight encryption algorithms offer the best trade-off between hardware efficiency and security for memory protection on constrained devices?

1.3 Objective

1.3.1 General Objective

To design and implement a secure and efficient memory encryption scheme for RISC-V architectures, mitigating hardware-based security vulnerabilities and ensuring data confidentiality.

1.3.2 Specific Objective

- To evaluate and adapt a lightweight block and stream cipher encryption algorithm for optimal security, performance, and hardware efficiency.
- To study real-world attacks (e.g. Rowhammer, cold boot, DMA-based memory scraping) relevant to unprotected memory systems.
- To identify vulnerabilities in the memory subsystem of RISC-V that can be mitigated through encryption.
- To implement RISC-V memory encryption by integrating a lightweight block and stream cipher into the memory controller on FPGA
- To analyze trade-offs between security, performance, area overhead, and power consumption in the RISC-V memory encryption, with a focus on protecting data in DRAM and data transfer between CPU and memory.

1.4 Methodology

In order to accomplish our objectives, we conduct the following steps:

- Literature Review: Overview of RISC-V architecture, memory security threats, existing memory encryption techniques, and their application to RISC-V.
- Algorithm Selection: Choosing lightweight block and stream ciphers suited for memory encryption.
- Design and Hardware Implementation: Develop and implement encryption/decryption modules in Verilog and integrate with the RISC-V pipeline.
- Analysis and Evaluation: Conduct security analysis and evaluate performance metrics including latency, throughput, and resource utilization.

1.5 Contributions

The main contribution in this work is a memory encryption for a RISC-V CPU that implement lightweight block and stream ciphers that are well suited for hardware implementations.

- Design of a lightweight memory encryption architecture for RISC-V.
- Hardware Implementation and Evaluation on FPGA.
- Security Analysis Against Real-World Threats
- Performance and Overhead Trade-off Analysis.

1.6 Thesis Outline

The rest of the paper is organized as follows. Chapter two establishes theoretical foundation for lightweight cryptography block and stream cipher algorithms used in memory encryption and RISC-V architecture overview. Chapter three reviews on memory security, protection overview of RISC-V architecture and lightweight block and stream ciphers.

Chapter four details the methodology of the design of the hardware architecture for block and stream ciphers. Section five presents the security analysis and performance evaluation. Finally, chapter six concludes with contribution and future directions of key findings from analysis and evaluation.

Chapter 2

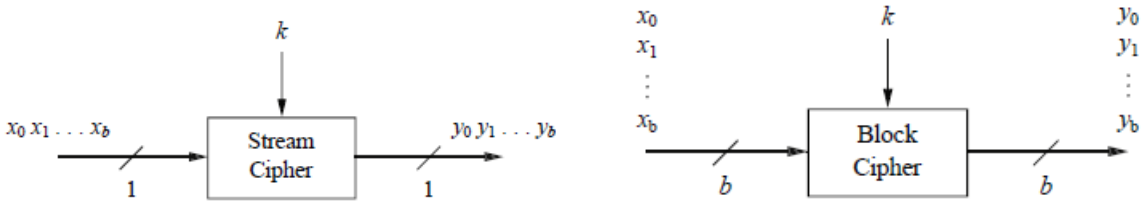
Background

This chapter provides a comprehensive theoretical overview of the fundamental concepts and technologies relevant to memory encryption in RISC-V-based systems. It begins with an introduction to the RISC-V instruction set architecture and its modularity, which makes it ideal for implementing custom security extensions. The discussion then focuses on cryptographic algorithms suitable for hardware-based memory protection, including lightweight block ciphers such as QARMA, PRINCE, and SIMON, as well as stream ciphers like ChaCha8, Grain128, and Trivium. These ciphers are analyzed in terms of their structure, security properties, and hardware efficiency, particularly in resource-constrained environments. Additionally, the chapter covers key components such as encryption engines, memory controllers, and key management mechanisms, establishing the groundwork for the architectural design and implementation approaches presented in later chapters.

In modern computing systems, memory encryption has emerged as a critical technique to ensure data confidentiality and integrity against physical and side-channel attacks. As processors increasingly operate in untrusted or shared environments, the protection of sensitive data in main memory becomes essential. The RISC-V architecture, with its open-source and modular design, offers a flexible platform for integrating custom security features, including memory encryption. Unlike proprietary architectures, RISC-V enables researchers and designers to explore fine-grained hardware-level enhancements tailored for secure computation. This thesis focuses on the analysis and design of memory encryption mechanisms for RISC-V CPUs, aiming to provide hardware-based confidentiality guarantees while maintaining performance efficiency and low resource overhead.

2.1 Stream Ciphers vs. Block Ciphers

Symmetric cryptography is broadly categorized into block ciphers and stream ciphers, which differ in how they process data. As shown in Figure 2.1, stream ciphers (Figure 2.1a) encrypt data in a continuous stream, typically operating on individual bits or small groups of bits in sequence. In contrast, block ciphers (Figure 2.1b) work on fixed-size chunks of data, encrypting one block (of b bits) at a time. The value b represents the block size, which is a defining feature of block ciphers[12].



(a) Stream Cipher **(b)** Block Cipher

Figure 2.1: Comparison of symmetric encryption: (a) Stream Cipher, (b) Block Cipher[12].

Stream ciphers are well-suited for real-time applications where data is transmitted continuously, as they generate ciphertext on the fly. Block ciphers, however, require complete blocks before encryption can begin, making them better suited for fixed-length data processing. Both methods aim to secure b bits of data per operation, but their underlying mechanisms and use cases differ significantly. This distinction highlights the flexibility and efficiency of symmetric cryptography in various encryption scenarios.Below is an explanation of how the two kinds of symmetric ciphers work.

2.2 Simon Cipher

The SIMON block cipher, developed by the NSA, is a lightweight encryption algorithm designed for resource-constrained environments. Ege Gulcan and etal.[13] have proposed a flexible and compact hardware architecture for SIMON, which is particularly suited for implementation on FPGAs. This architecture supports all configurations of SIMON, offering adaptability in security levels through variable key sizes and block sizes, while maintaining compactness and efficiency. The architecture is notable for its minimal area usage on FPGAs, making it a competitive choice for secure applications in constrained environments.While the proposed architecture offers significant advantages in terms of flexibility and compactness, it is important to consider the trade-offs involved. The balance between security, area, and throughput must be carefully managed to meet specific application requirements. Additionally, while the architecture is optimized for FPGAs, other platforms may require different considerations for optimal performance.

The area and throughput of a hardware implementation of a block cipher are significantly influenced by the levels of parallelism employed—specifically, bit-level, round-level, and encryption-level parallelism. In bit-level parallelism, the operand width can range from a single bit to n bits, where n denotes the block size. Round-level parallelism enables the execution of multiple rounds per clock cycle, from a single round up to r rounds, where r represents the total number of rounds in the cipher.

Table 2.1: Simon Parameters

Security Configuration	Block Size (2n)	Key Size	Word Size (n)	Key Words (m)	Rounds
1	32	64	16	4	32
2	48	72	24	3	36
3	48	96	24	4	36
4	64	96	32	3	42
5	64	128	32	4	44
6	96	96	48	2	52
7	96	144	48	3	54
8	128	128	64	2	68
9	128	192	64	3	69
10	128	256	64	4	72

Table 2.1 shows the parameters for all SIMON configurations. The word size, n, defines the bit length of each Feistel word, resulting in a block size of 2n. The key length is a multiple of the word size, with m representing the number of words in the key. A new parameter, the security configuration, is introduced to specify the desired SIMON setup.

Figure 2.2 illustrates the round function used in all configurations of SIMON. The variables X_{i+1} and X_i represent the two n-bit words of the block. These words initially hold the plaintext input and are updated with the output after each round. The round function employs bitwise AND, bitwise XOR, and circular left shift operations. In each round, X_{i+1} undergoes shift and bitwise AND operations, and the result is XORed with X_i and the round key. This computed value replaces X_{i+1}, while its original content is transferred to X_i. The process repeats for the specified number of rounds[13]. The SIMON round encryption function, F, is represented in Equation (2.1).

Figure 2.3 shows the key schedule function of SIMON for a configuration with four master key words (i.e., $m = 4$), each n bits wide. These initial four sub-keys are generated in the first iteration and used as the round keys for the first four rounds. In each subsequent iteration, a new round key k_i is produced for round i, where $0 \leq i < (T - m)$. The most significant word, k_{i+3}, is circularly shifted right by three bits (S^{-3}) and XORed with k_{i+1}. The result is then circularly shifted right by one bit (S^{-1}), and XORed with the least significant word k_i and the round constant $(c \oplus z_{ij})$[14].

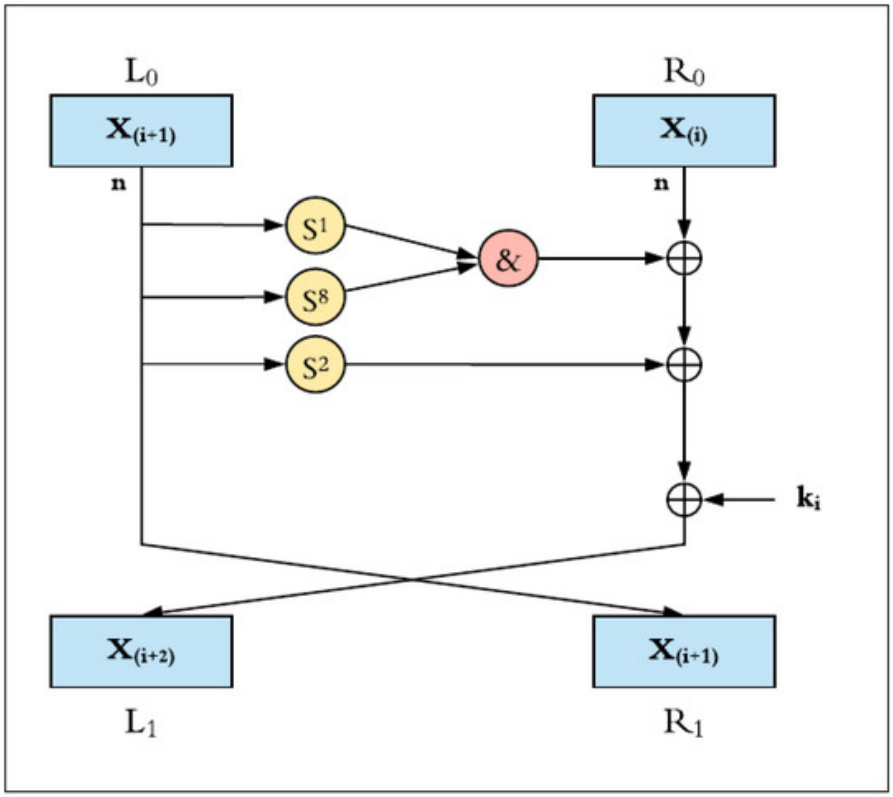


Figure 2.2: SIMON Round Function[14].

The constant c is defined as $(2^n-1)\oplus 3$, which yields a value with $(n-2)$ ones followed by two zero bits (i.e., $c = 2^n - 4 =$ `0xff ... fc`). The sequence z_{ij} represents the i-th bit of a predefined sequence z_j, selected based on the configuration, where i is calculated as $(i-m)$ mod 62 for $m \leq i \leq T-1$.

The research on FPGA modeling and optimization of the SIMON lightweight block cipher highlights its suitability for low-resource devices (LRDs) by balancing security and resource efficiency. SIMON is designed for hardware implementations, particularly in environments where power and area are critical. The studies reveal various implementation strategies, including scalar and pipelined designs, each with distinct advantages in terms of resource utilization and

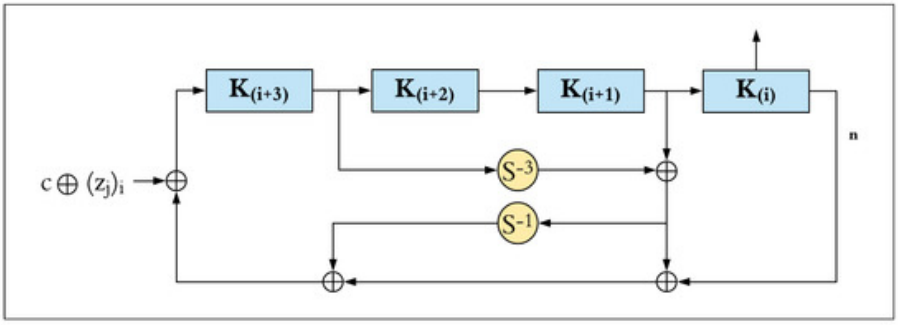


Figure 2.3: SIMON four-word key schedule[14].

energy consumption.Sa'ed Abed et al.[14] focus on implementing, optimizing, and modeling the SIMON lightweight block cipher for low-resource devices using FPGA technology, emphasizing energy and power efficiency across various design implementations, including scalar and pipelined approaches.

$$F(x, y, k) = (y \oplus ((S^1 x \,\&\, S^8 x) \oplus S^2 x) \oplus k) \tag{2.1}$$

$$k_{i+m} = c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1})(S^{-3} k_{i+3} \oplus k_{i+1}) \tag{2.2}$$

where k is the round key, x is the leftmost word of the cipher block, and y is the rightmost word.

2.3 Prince Cipher

PRINCE is a low-latency block cipher designed for efficient encryption in resource-constrained environments. It employs a unique features a 12-round core, utilizes α-reflection, which allows decryption to be performed as an encryption operation with a related key, thus minimizing overhead. This cipher is particularly suitable for applications requiring real-time security, as it can encrypt data within a single clock cycle, making it competitive in terms of chip area and performance[15]. It features a 12-round core, utilizes α-reflection for efficient decryption, and employs whitening keys for enhanced security in pervasive computing applications.Nicolai Müller and et al.[16] presents low-latency implementations of the PRINCE cipher using first-order secure hardware masking. It demonstrates a masked variant with a single register stage per round, optimizing trade-offs between circuit area, latency, and randomness requirements for specific use cases.

PRINCE [15] is a 64-bit block cipher that uses a 128-bit secret key k. The key expansion first divides k into two parts of 64 bits each, $k = (k_0 \| k_1)$, where $\|$ denotes concatenation. The key material is then extended to 192 bits:

$$k = (k_0 \| k_1) \rightarrow (k_0 \| k_0' \| k_1) = (k_0 \| L(k_0) \| k_1), \quad \text{with } L(x) = (x \gg 1) \oplus (x \lll 63) \tag{2.3}$$

Figure 2.4 shows the 64-bit subkeys, k_0 and k_0' are used as input and output whitening keys, respectively, while k_1 is used as the internal key for the core block cipher `PRINCEcore`.

The overall architecture of the PRINCE cipher is illustrated in Figure 2.4. Its main components are outlined as follows:

Key Schedule: The PRINCE cipher uses a 128-bit secret key, which is divided into two equal parts: k_0 and k_1. The first part, k_0, serves as the pre-whitening key, applied before the main cipher operations. For post-whitening, a slightly modified version of k_0 is used, defined as $k_0' = (k_0 \ggg 1) \oplus (k_0 \gg 63)$, where $\ggg$ denotes a circular right shift and $\gg$ indicates a logical right shift. Pre- and post-whitening refer to the addition of key material before and after the core cipher function. In PRINCE, this core is called `PRINCEcore`. The second half of the key, k_1, is used directly during the key addition steps of both the forward (R) and inverse (R^{-1}) round functions.

The round functions R and R^{-1}: consist of a series of operations: an XOR with the fixed key k_1, an XOR with a round-dependent constant RC, followed by a substitution layer using the S-box S (or its inverse S^{-1}), and a linear diffusion layer M (or its inverse M^{-1}).

Round-Dependent Constant: The round constants RC_i are defined such that $\text{RC}_i \oplus \text{RC}_{11-i} = \alpha$ for $0 \leq i \leq 1$, where $\alpha =$ `coac29b7c97c50dd` (in hexadecimal

S-box Layer S and its Inverse S^{-1}: The S-box layer of PRINCE operates on 4-bit input values, providing a nonlinear substitution through a 4-bit to 4-bit mapping. Table 2.2 shows both the forward S-box $S(X)$ and its inverse $S^{-1}(X)$.

Table 2.2: S-box Layer of PRINCE.

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(X)	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4
X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S^{-1}(X)	B	7	3	2	F	D	8	9	A	6	4	0	5	E	C	1

Linear Diffusion Layer M and its Inverse M^{-1}: The linear diffusion layer M is designed to enhance diffusion within the cipher by combining specific input bits through XOR operations. Each output bit is computed as the XOR of three distinct input bits, ensuring that each bit in the output depends on multiple bits from the input. This structure increases the cipher's resistance to differential and linear cryptanalysis. The inverse layer M^{-1} is constructed to reverse the effect of M, allowing for proper decryption.

Middle Involution: The middle involution is a key structural component of the PRINCE cipher. It consists of three functions applied in sequence: SR^{-1}, M', and SR. The operations SR and SR^{-1} perform row-wise permutations similar to the ShiftRows transformation in AES. The function M' is a linear diffusion operation that further enhances the spread of input differences

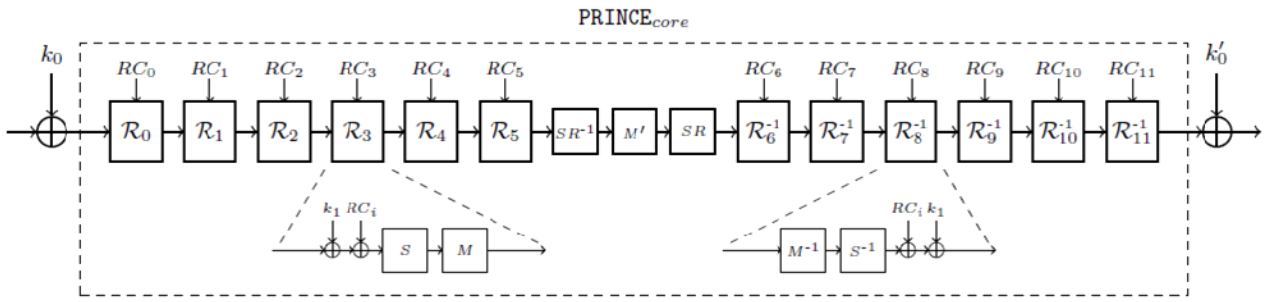


Figure 2.4: A schematic view of the PRINCE cipher[15].

across the output. This combination ensures that the cipher remains secure while maintaining a low-latency design due to its involutive nature.

Middle Involution and Key Symmetry: The middle involution functions as a transitional layer, facilitating the structural symmetry between the forward and inverse round transformations. This structure enables a unique symmetry property of the cipher: encryption using a key k is equivalent to decryption using the key $k \oplus \alpha$, where α is a fixed constant. This design contributes to the cipher's low-latency characteristics and simplifies the hardware implementation of both encryption and decryption.

2.4 QARMA Cipher

AVANZI Roberto[17] presented QARMA, a new family of lightweight tweakable block ciphers that are targeted at applications such as memory encryption, the generation of very short tags for hardware-assisted prevention of software exploitation, and the construction of keyed hash functions. QARMA is inspired by reflection ciphers such as PRINCE, to which it adds a tweaking input, and MANTIS. However, QARMA differs from previous reflector constructions in that it is a three-round Even-Mansour scheme instead of an FX-construction, and its middle permutation is noninvolutory and keyed. QARMA exists in 64- and 128-bit block sizes, where block and tweak size are equal, and keys are twice as long as the blocks.

In [18], the authors investigate the security of QARMA-64 and QARMA-128 block ciphers against truncated differential attacks utilizing the meet-in-the-middle (MITM) attack methodology. By employing differential enumeration in combination with a key-dependent sieve technique, they construct an MITM distinguisher for QARMA-64. This distinguisher is subsequently extended to facilitate a 10-round attack on both QARMA-64 and QARMA-128. Notably, this work presents the first known analysis of QARMA's resistance to MITM attacks.

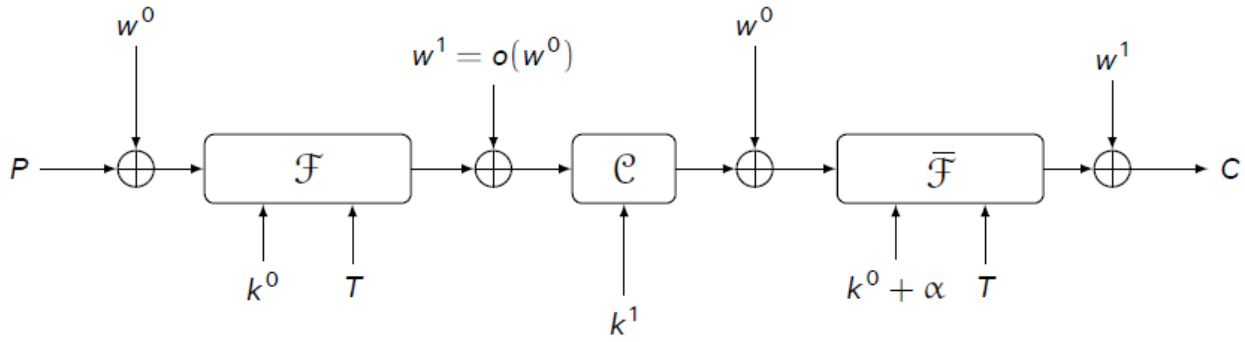


Figure 2.5: The Overall Scheme[17].

Figure 2.5 illustrates the high-level structure of the TBC QARMA cipher. In this design, an overline on a function (e.g., $\overline{F}$) denotes its inverse. QARMA uses a three-round Even-Mansour construction in which the permutations are parameterized by a core key, and inter-round key mixing is derived from a whitening key. The first and third permutations are inverses of each other and are further influenced by a tweak. The central permutation is designed to be easily inverted through a simple transformation of the key.

Figure 2.6 presents a more detailed view of the cipher, which follows a bricklayer Substitution-Permutation Network (SPN) structure. While QARMA incorporates concepts from ciphers such as PRINCE, MIDORI, and MANTIS, its unique keying mechanism and central permutation distinguish it with its own security properties. The keys k_0, k_1, w_0, and w_1 are derived from a master key K through a simple specialization process. The plaintext P and tweak T are each expressed as arrays of sixteen m-bit cells:

$$P = p_0 \parallel p_1 \parallel \cdots \parallel p_{15}, \quad T = t_0 \parallel t_1 \parallel \cdots \parallel t_{15}$$

where $m = 4$ or 8, making $n = 16m$. These arrays are viewed as 4×4 matrices processed column-wise. Bits inside each cell are ordered in little-endian, while the cells themselves follow big-endian ordering. Cipher components include S-box layers S, permutations h and π, MixColumns-like transformations M and Q (with Q being involutive), and a linear feedback shift register (!). All additions use bitwise XOR, and the term "tweakey" (tk) refers to a value derived from the key, tweak, and round constants[17].

$$IS = s_0 \| s_1 \| \cdots \| s_{14} \| s_{15} = \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix} \tag{2.4}$$

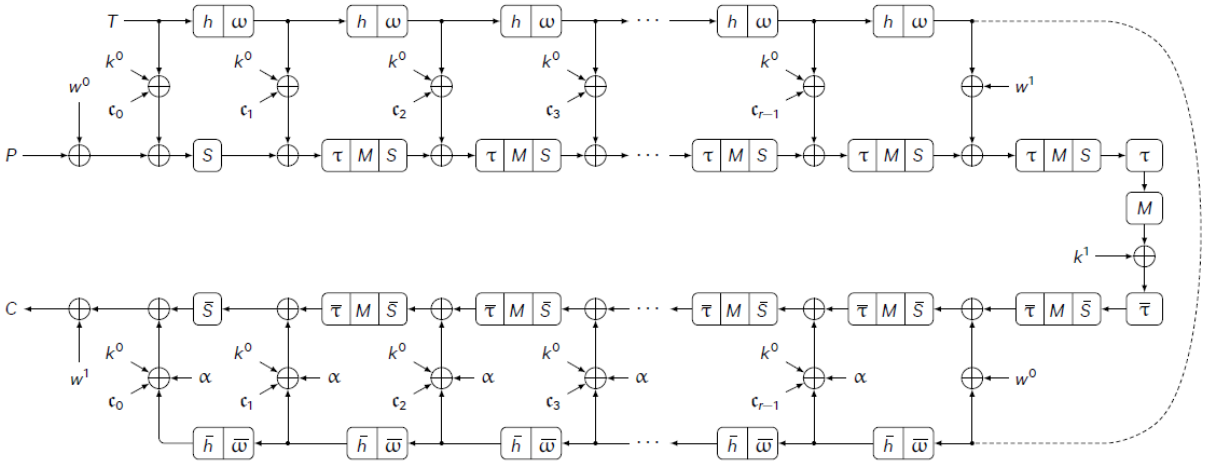


Figure 2.6: The structure of QARMA[17].

2.5 ChaCha

ChaCha, similar to Salsa20, updates four 32-bit state words using a combination of four additions, four XORs, and four bitwise rotations, all in a reversible manner. However, ChaCha differs in the sequence of these operations and notably performs two updates on each word instead of just one. In particular, the values of a, b, c, and d are updated through this altered process.

This sequence of operations appears to be part of a cryptographic mixing function, specifically resembling the quarter-round function from the ChaCha cipher. Let me break it down step by step:

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

In ChaCha, this quarter-round is applied to different columns and diagonals of the state matrix in alternating rounds. A full ChaCha round consists of applying this operation to all four columns, then all four diagonals.

Figure 2.7 illustrates the overall structure of ChaCha. The matrix M is permuted through r rounds. After permutation, the output of the round function R is added modulo 2^{32} to M, resulting in the key stream K. This key stream is then XORed with the plaintext (PT) or ciphertext (CT) to perform encryption or decryption, respectively.

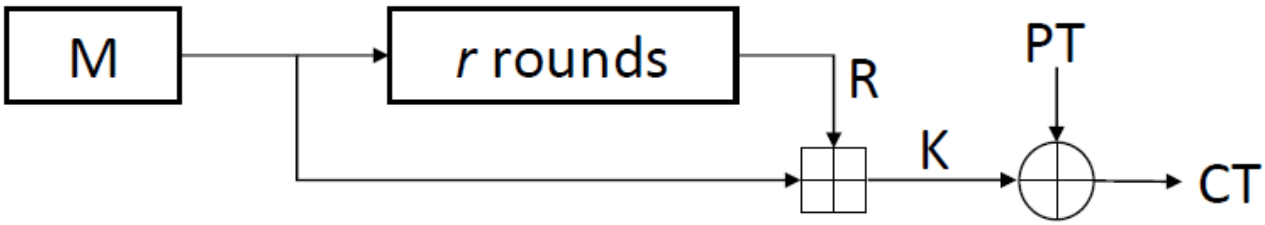


Figure 2.7: The structure of ChaCha Stream Cipher[19].

ChaCha, much like Salsa20/r, constructs a 4×4 matrix and applies an invertible transformation over r rounds. The resulting matrix is then added to the original matrix to generate a 16-word (64-byte) output block.

There are three primary differences in its design. The first is that ChaCha permutes the order of the words in the output block according to a specific permutation. This rearrangement does not compromise security but improves performance on SIMD platforms and has no measurable impact on performance on other architectures. The second distinction lies in the initialization of the matrix: ChaCha places all attacker-controlled input words in the lower rows of the matrix.

Table 2.3: Initial ChaCha State Matrix

constant	constant	constant	constant
key	key	key	key
key	key	key	key
input	input	input	input

The key words are inserted into the matrix in sequential order, while the input words consist of the block counter followed by the nonce. The constants used are identical to those in Salsa20.

In the first round of ChaCha, the key values are added to the constants. The resulting values are then XORed with the input words, followed by a rotation operation. These rotated values are subsequently added back into the key words, continuing the transformation process.*(Note: Starting with modifications to the constants is beneficial when designing compression functions that utilize the same core structure.)*

The third key difference is in the way ChaCha processes its matrix during each round. It consistently follows the same order when sweeping through the rows. In the first round, it modifies the

first, fourth, third, and second columns in that sequence—repeating this order twice. In the second round, the same sequence is applied, but the operations are performed along the southeast diagonals of the matrix instead of the columns.

```
QUARTERROUND(x0, x4, x8, x12)
QUARTERROUND(x1, x5, x9, x13)
QUARTERROUND(x2, x6, x10, x14)
QUARTERROUND(x3, x7, x11, x15)
QUARTERROUND(x0, x5, x10, x15)
QUARTERROUND(x1, x6, x11, x12)
QUARTERROUND(x2, x7, x8, x13)
QUARTERROUND(x3, x4, x9, x14)
```

The four words processed in each quarter-round are consistently ordered from top to bottom within the matrix[20].

Arthur Beckers and et al.[19] investigate the vulnerability of the ChaCha and Salsa families of stream ciphers to fault injection attacks in their work on fault analysis. Their study highlights how carefully induced faults can compromise the internal state of these ciphers, enabling adversaries to recover secret keys or internal variables. By analyzing the structural similarities and differences between ChaCha and Salsa, they demonstrate practical fault attack strategies and propose countermeasures to enhance resilience. This research contributes to the understanding of security risks in modern stream ciphers, especially in hardware implementations vulnerable to physical attacks.

2.6 Grain128 and Trivium Cipher

Grain-128 is a variant within the Grain stream cipher family that supports 128-bit keys. In this version, the initialization vector (IV) has been extended to 96 bits, and both the linear and nonlinear feedback shift registers have lengths of 128 bits. The cipher undergoes an initialization process that lasts for 256 clock cycles. During this phase, specific functions are used to update the internal state[21].

An overview of the various components used in the cipher is provided in Figure 2.8, which serves as a reference throughout this specification. The cipher comprises three primary building blocks: a Linear Feedback Shift Register (LFSR), a Nonlinear Feedback Shift Register (NFSR), and an output function. The state of the LFSR is represented by $s_i, s_{i+1}, \ldots, s_{i+127}$, while the

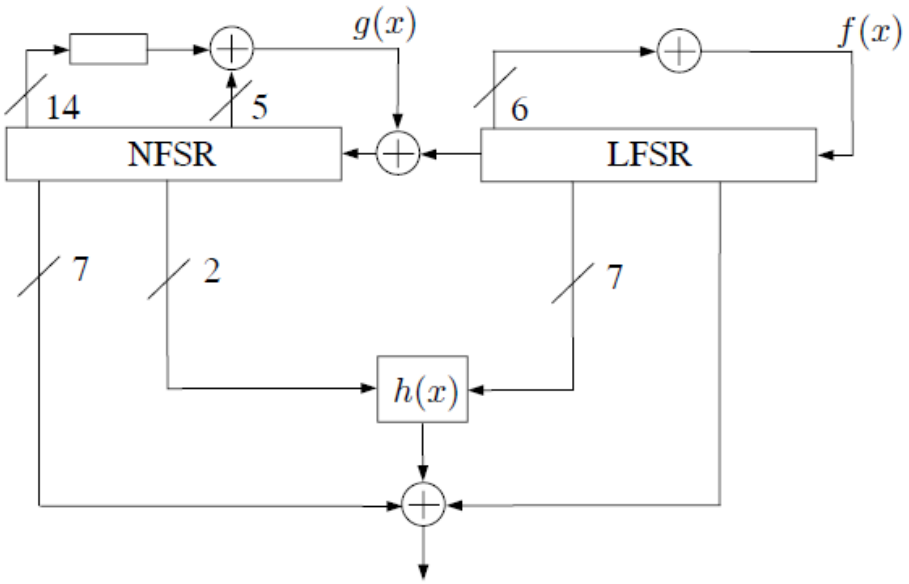


Figure 2.8: An overview of the cipher[22].

state of the NFSR is represented by $b_i, b_{i+1}, \ldots, b_{i+127}$. The feedback polynomial of the LFSR, denoted as $f(x)$, is a primitive polynomial of degree 128. It is defined as:

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128} \tag{2.5}$$

To remove any possible ambiguity, we also provide the corresponding update function of the LFSR:

$$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96} \tag{2.6}$$

The nonlinear feedback polynomial of the NFSR, denoted by $g(x)$, is the sum of a linear function and a bent function. It is defined as:

$$\begin{aligned} g(x) = 1 &+ x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} + x^{61}x^{125} \\ &+ x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117} \end{aligned} \tag{2.7}$$

To ensure clarity, we also present the corresponding update function of the NFSR. In the expression below, it is important to note that the bit s_i, which is used to mask the input to the NFSR, is explicitly included in the update function, although it is not part of the feedback poly-

nomial.

$$\begin{aligned} b_{i+128} = s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} \\ + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} \\ + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} \\ + b_{i+68}b_{i+84} \end{aligned} \tag{2.8}$$

The cipher's internal state is composed of 256 memory elements distributed across the two shift registers. From this state, nine specific bits are extracted to serve as inputs to a Boolean function denoted by $h(x)$. Of these, two inputs are derived from the NFSR, while the remaining seven are obtained from the LFSR. The function $h(x)$ is a relatively simple Boolean function with an algebraic degree of 3.The function is defined as follows:

$$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8 \tag{2.9}$$

The variables $x_0, x_1, \ldots, x_8$ correspond to the following tap positions:

$$x_0 = b_{i+12}, \quad x_1 = s_{i+8}, \quad x_2 = s_{i+13}, \quad x_3 = s_{i+20}, \quad x_4 = b_{i+95},$$

$$x_5 = s_{i+42}, \quad x_6 = s_{i+60}, \quad x_7 = s_{i+79}, \quad x_8 = s_{i+95}$$

The final output function of the cipher is defined as:

$$z_i = \sum_{j \in A} b_{i+j} + h(x) + s_{i+93} \tag{2.10}$$

where the set $A = \{2, 15, 36, 45, 64, 73, 89\}$ specifies the tap positions from the NFSR used in the linear combination.

Before keystream generation can begin, the cipher requires initialization using the key and the initialization vector (IV). Let the key bits be represented by k_i for $0 \leq i \leq 127$, and the IV bits by IV_i for $0 \leq i \leq 95$. The initialization process proceeds as follows: the 128 elements of the NFSR are loaded with the key bits, i.e., $b_i = k_i$ for $0 \leq i \leq 127$, while the first 96 elements of the LFSR are loaded with the IV bits, $s_i = \text{IV}_i$ for $0 \leq i \leq 95$. The remaining 32 bits of the LFSR, corresponding to positions 96 through 127, are set to ones, i.e., $s_i = 1$ for $96 \leq i \leq 127$. Following this loading phase, the cipher is clocked 256 times without producing any output keystream. Instead, the output function is fed back and XORed with the inputs to both the LFSR and the NFSR, as illustrated in Figure 2.9.

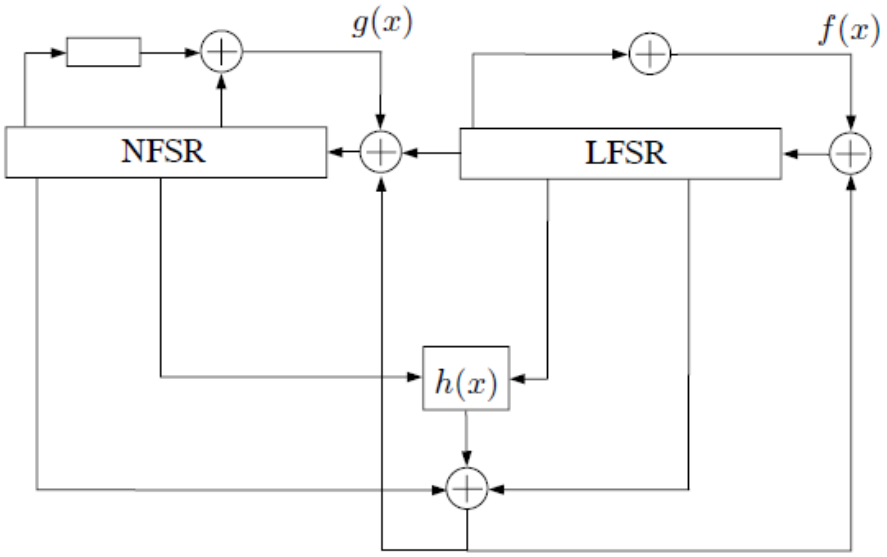


Figure 2.9: The key initialization[22].

Trivium is a lightweight stream cipher that draws inspiration from block cipher design principles, prioritizing both simplicity and performance.An extensive study examines Trivium's internal architecture, confirming its resistance to various cryptanalytic techniques and supporting its suitability for real-world applications. From a hardware implementation standpoint introduce energy-efficient masking strategies to defend Trivium against side-channel attacks, tackling key security issues in resource-limited systems. Nonetheless, concerns persist, as recent studies have identified possible vulnerabilities in FPGA-based implementations of Trivium, especially when lacking adequate protections against fault injection and side-channel threats[23],[24],[25],[26].

Figure 2.10 illustrates the structure of the algorithm; the initialization phase functions identically to the keystream generation phase, with the exception that no keystream is produced during this stage. After loading the key and initialization vector (IV), the internal state undergoes 4 complete rotations, equivalent to $4 \times 288 = 1152$ clock cycles. Key and IV are loaded as following:

$$(s_1, s_2, \dots, s_{93}) \leftarrow (K_1, \dots, K_{80}, 0, \dots, 0)$$
$$(s_{94}, s_{95}, \dots, s_{177}) \leftarrow (IV_1, \dots, IV_{80}, 0, \dots, 0)$$
$$(s_{178}, s_{179}, \dots, s_{288}) \leftarrow (0, \dots, 0, 1, 1, 1)$$

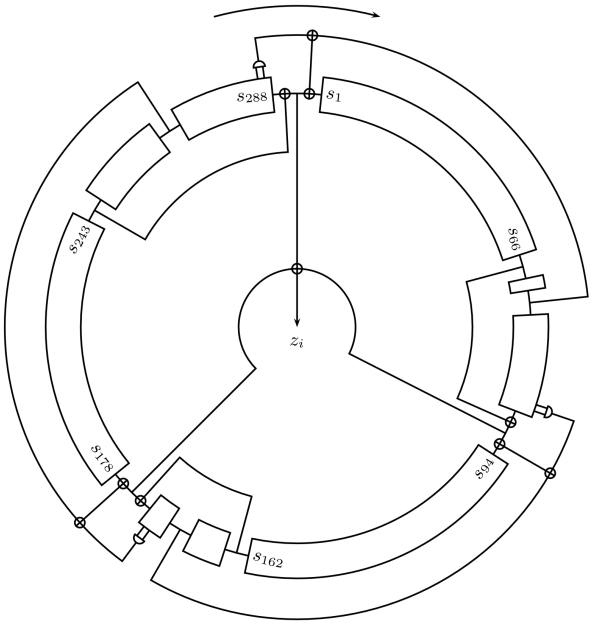


Figure 2.10: Structure of Trivium[24].

2.7 DRAM Architecture

Dynamic Random Access Memory (DRAM) serves as the primary form of volatile memory in modern computing systems. Its critical role stems from its high-speed data access capabilities and substantial storage capacity, both of which are essential for the performance and responsiveness of contemporary digital devices. As computing tasks become increasingly data-driven and resource-intensive, DRAM has emerged as a foundational component in enabling efficient and effective system operations[27].

One of the most significant attributes of DRAM is its speed. DRAM provides low-latency access to data, allowing processors to quickly retrieve and modify information during execution. This rapid access is crucial for high-performance tasks such as running complex algorithms, real-time processing of multimedia content, and maintaining smooth and responsive user interfaces. The user experience in today's digital environments—whether in personal computing, mobile devices, or embedded systems—is heavily dependent on the speed at which DRAM can deliver data to the processor.

In addition to speed, the large capacity offered by modern DRAM modules is a key factor in supporting the memory requirements of contemporary applications. As software becomes more sophisticated and datasets grow in size, systems must be equipped with ample memory to manage multiple processes simultaneously. DRAM enables efficient multitasking, supports virtual machines, and handles large-scale data analysis tasks, all of which are essential for both personal computing and enterprise-level operations[27].

Moreover, the rise of data-intensive applications such as artificial intelligence, big data analytics, scientific simulations, and high-definition media processing has amplified the demand for high-performance memory systems. These applications require frequent and rapid access to large volumes of data, a requirement that DRAM is uniquely suited to meet. As AI and deep learning models become more widespread, the need for large, fast-access memory continues to increase, making DRAM indispensable in modern computational workflows.

Cloud computing and virtualization further highlight the importance of DRAM. In cloud environments, servers often host numerous virtual machines and must manage concurrent access from multiple users. DRAM's fast access times and scalability are vital in these scenarios, ensuring that system resources are allocated dynamically and that performance remains consistent under varying workloads.

In summary, DRAM is a cornerstone of modern computing, enabling fast, reliable, and scalable memory operations across a broad range of applications. Its role is essential not only for traditional computing tasks but also for the growing demands of future technologies such as AI, cloud computing, and real-time data processing.

DRAM is organized in a hierarchical manner, comprising multiple levels such as channels, ranks, bank groups, and individual banks. This structured layout is designed to enhance memory access efficiency and streamline data retrieval processes[27].

At the top of this hierarchy are channels, which serve as the primary pathways for data transfer within the DRAM system. Each channel connects to one or more ranks, forming the backbone of communication between the memory and other system components.

While each rank can operate independently, full parallelism across ranks is limited. This limitation stems from the fact that ranks sharing the same channel also share the associated data lines. As a result, although ranks may perform operations concurrently, they must coordinate their access to the shared lines, which can lead to minor delays in data transmission.

A rank typically constitutes a 64-bit wide memory module made up of several DRAM chips. These chips are the core elements responsible for data storage and processing. For example, in an x8 configuration, eight individual DRAM chips are combined—each with an 8-bit width—forming a complete 64-bit data path. This structure is depicted in Figure 2.11, demonstrating the interconnection of the eight chips.

Other common configurations include x4, which uses 16 chips, and x16, which employs just 4 chips. Each configuration presents different trade-offs in terms of density, performance, and power efficiency, and they are chosen based on specific application and system requirements.

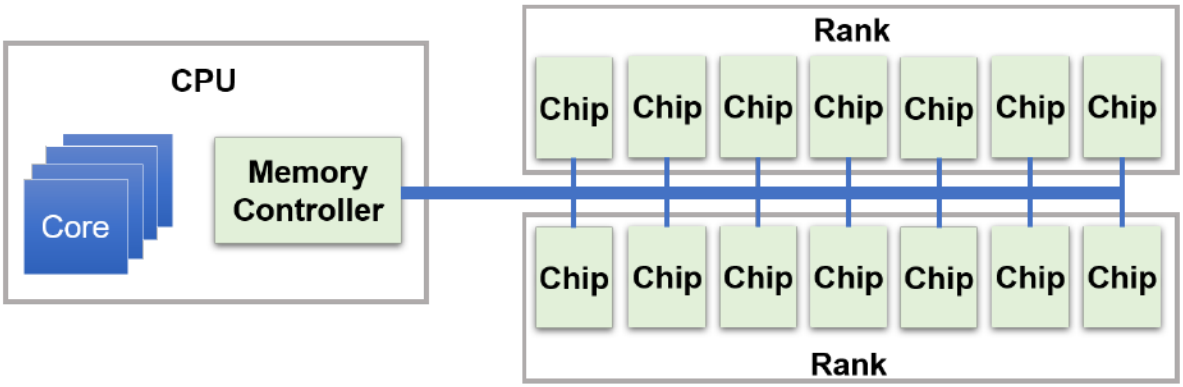


Figure 2.11: Memory System Organization.[27]

2.8 RISC-V

The RISC-V Instruction Set Manual is divided into two primary volumes: Volume I focuses on the unprivileged instruction set architecture (ISA), while Volume II addresses the privileged architecture. The unprivileged ISA outlines the basic instructions, operands, and formats necessary for user-level programming, whereas the privileged architecture encompasses additional functionalities essential for operating systems and hardware management.[2],[3].

The RISC-V Instruction Set Manual Volume I[2]: Unprivileged ISA provides a comprehensive specification of the base integer instruction set and standard extensions that are accessible at the user level. This volume serves as the foundation for application development on RISC-V platforms, detailing the syntax, semantics, and behavior of instructions used in user-mode. It is designed to be modular and extensible, allowing for a flexible architecture that supports a wide range of implementations, from small embedded systems to high-performance processors. The clarity and simplicity of the unprivileged ISA make it suitable for both academic use and commercial development, promoting portability and long-term architectural stability.

Volume II: Privileged Architecture builds upon the unprivileged specification by defining the mechanisms that support operating systems, hypervisors, and other system-level software. This volume introduces the privilege modes, control and status registers (CSRs), trap handling, memory management, and other features required for secure and efficient system control. It provides the necessary architectural support for isolation, resource management, and exception handling, enabling robust multitasking and system protection. Together with Volume I, this document offers a complete view of the RISC-V architecture, enabling both hardware designers and software developers to build and manage full-stack RISC-V systems[3].

At any given moment, a RISC-V hardware thread (hart) operates at a specific privilege level,

Table 2.4: RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	–
3	11	Machine	M

which is indicated by a mode field within one or more control and status registers (CSRs). Currently, the RISC-V architecture defines three privilege levels, as outlined in Table 2.4.

These privilege levels serve to enforce isolation between different layers of the software stack. If a program attempts to execute an operation that is not permitted under the current privilege level, an exception is triggered. Such exceptions typically result in a trap to a lower-level execution environment that has the authority to handle them.

The machine privilege level (M-mode) holds the highest level of authority in the RISC-V architecture and is the only privilege level required by all RISC-V hardware implementations. Code executed in M-mode is generally considered highly trusted due to its unrestricted access to the underlying hardware. M-mode is also responsible for managing secure execution environments. In contrast, user mode (U-mode) and supervisor mode (S-mode) are designed for typical application-level and operating system-level operations, respectively.

Each privilege level is associated with a core set of privileged ISA (Instruction Set Architecture) extensions, along with optional extensions and variants. For instance, M-mode includes an optional standard extension for memory protection. Depending on the design trade-offs between isolation and implementation complexity, a RISC-V system may support between one and three privilege levels, as illustrated in Table 2.5.

Table 2.5: Supported combinations of privilege modes.

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

2.9 RISC-V Architecture

RISC-V architecture represents a significant advancement in processor design, emphasizing openness and flexibility. It serves as a bridge between hardware and software, allowing for diverse implementations tailored to specific performance needs. The architecture's modularity and extensibility facilitate innovations in various applications, including edge computing and cryptography. Below are key aspects of RISC-V architecture and interfaces.

Moritz Nöltner[28]offers an insightful overview of the RISC-V instruction set architecture with a focus on its practical realization through the RocketChip generator. The work explores the modular and open-source nature of the RISC-V ecosystem, emphasizing how the RocketChip SoC generator, developed at UC Berkeley, facilitates scalable and customizable hardware design. By detailing the interaction between core components such as the processor pipeline, tilelink interconnect, and memory hierarchy, the document provides a strong foundation for understanding how abstract architectural principles translate into concrete, synthesizable hardware.

The author also examines the RocketChip's interface layers and its integration with system-level components, highlighting its compatibility with standard toolchains and its support for various privilege levels defined in the RISC-V specification. Through this analysis, the paper underscores the strengths of RISC-V in academic and industrial research, especially in terms of flexibility, verification, and experimentation. This makes it particularly valuable for students, researchers, and engineers interested in learning about SoC design within the RISC-V framework.

Earlier versions of RocketChip used a custom bus interface called NASTI, but newer versions adopt ARM's widely supported Advanced eXtensible Interface (AXI). AXI features multiple independent, unidirectional channels with handshake signals, supporting burst transfers and various topologies. Its design allows flexible insertion of register stages to optimize latency and frequency. RocketChip's default AXI configuration uses 64-bit data widths for both read and write channels. The AXI interface also efficiently connects to memory controllers, providing robust support for high-performance memory transactions.

To support synchronization in multi-core systems, AXI includes exclusive access for atomic read-modify-write operations, while simpler AXI4-Lite offers a reduced feature set for low-performance peripherals. AXI4 dropped the locked transactions feature found in AXI3 due to complexity and quality-of-service issues. Interfacing between AXI and AXI4-Lite requires additional logic or defined conversion mechanisms when needed[28]

Figure 2.12 illustrates an instance of a Rocket Chip/lowRISC system[29]. It consists of two tiles connected to a four-bank L2 cache, which in turn links to external I/O and memory through an AXI interconnect. Tile 1 contains an out-of-order BOOM core equipped with an FPU, separate L1 instruction and data caches, and an accelerator that utilizes the RoCC interface. Tile 2 features a different core, Rocket, with distinct L1 data cache configurations. Overall, Rocket Chip functions as a library of configurable generators that can be customized and combined to create a wide range of system-on-chip (SoC) designs. Below is a summary of the current capabilities of these generators and interfaces:

- **Core:** Generators for the Rocket scalar core and the BOOM out-of-order superscalar core, both optionally equipped with FPUs, customizable functional unit pipelines, and configurable branch predictors.
- **Caches:** A suite of cache and TLB generators offering flexible configurations for size, associativity, and replacement policies.
- **RoCC:** The Rocket Custom Coprocessor interface, serving as a framework for designing application-specific coprocessors with customizable parameters.
- **Tile:** A tile generator template for cache-coherent tiles, allowing configuration of the number and types of cores and accelerators, as well as the layout of private caches.
- **TileLink:** A generator for networks of cache-coherent agents and their corresponding cache controllers, with configurable options such as tile count, coherence policy, shared backing storage, and physical network implementation.
- **Peripherals:** Generators for AMBA-compatible buses, including AXI, AHB-Lite, and APB, along with various converters and controllers such as the Z-scale processor.

The paper by Viktor V. Prutyanov et al.[30] investigates how the organization of memory systems affects the efficiency of memory encryption techniques. The authors focus on the additional latency and performance penalties introduced by cryptographic operations, especially when metadata needed for encryption is stored in DRAM. To address this challenge, the paper proposes the use of a metadata (MD) cache, which stores frequently accessed metadata closer to the processor, thus reducing the need for repeated DRAM accesses.

The study includes a thorough experimental setup using an FPGA-based platform with DDR3 DRAM and a RISC-V RocketChip soft-core processor. The authors analyze performance using encryption schemes such as Intel SGX-style memory encryption and the ELM (Encryption for Large Memory) integrity tree approach. Through simulation and empirical testing, the paper shows that the inclusion of a metadata cache significantly reduces memory access latency and

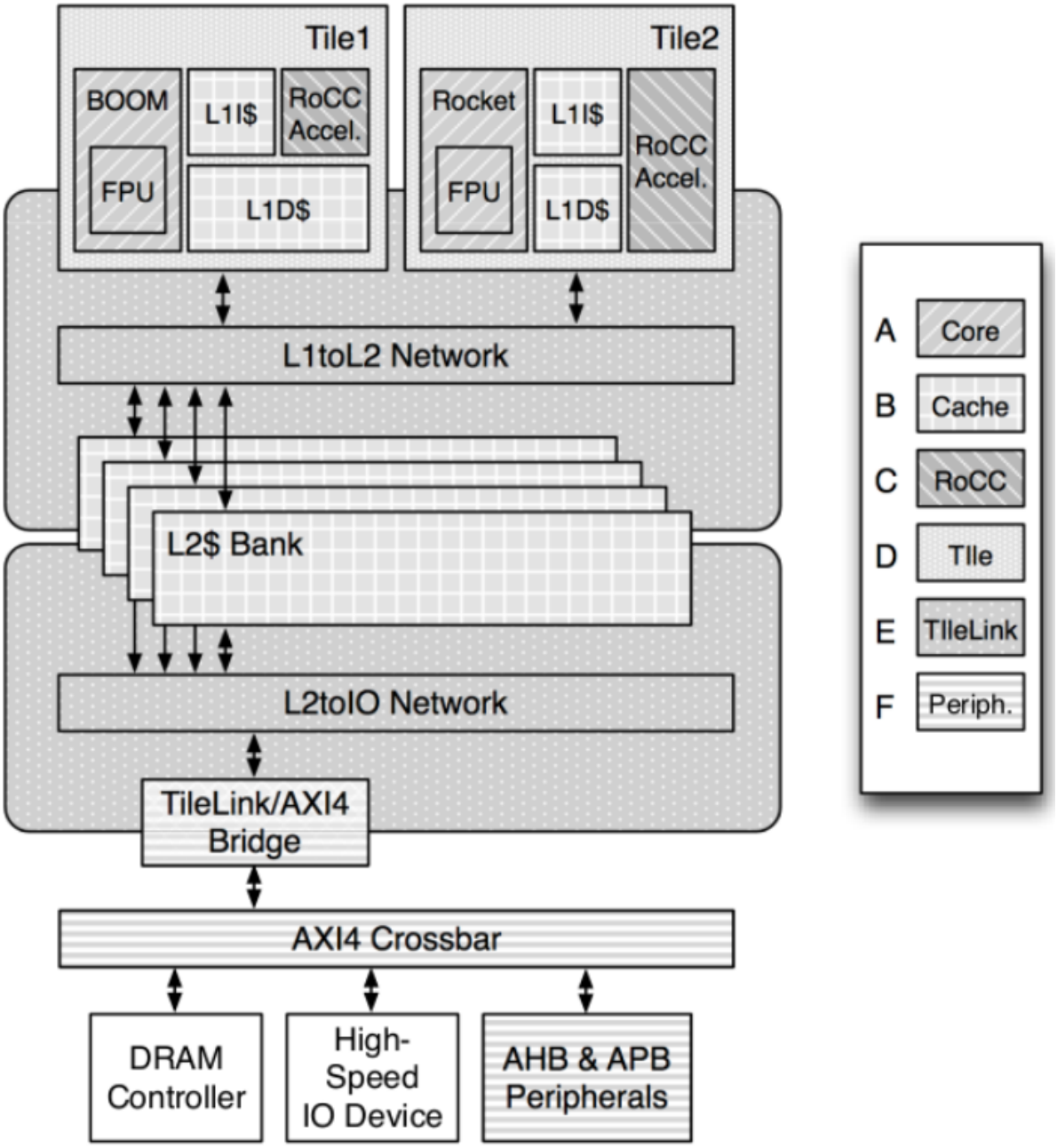


Figure 2.12: The Rocket Chip generator consists of the following sub-components: A) Core generator B) Cache generator C) RoCC-compatible coprocessor generator D) Tile generator E) TileLink generator F) Peripherals[29].

improves overall system performance. These results are particularly relevant for secure embedded systems, where balancing encryption overhead and performance is critical. Overall, the research makes a strong case for integrating metadata-aware caching strategies into memory hierarchies to support efficient hardware-based memory encryption.

This research provides valuable insights into optimizing memory encryption performance by

strategically managing the memory hierarchy, particularly through the implementation of metadata caches. The findings are pertinent for the design of secure systems where memory encryption is essential, such as in embedded systems and environments requiring protection against physical attacks.

The RISC-V open instruction set architecture (ISA) has gained substantial traction due to its flexibility, extensibility, and open-source model. However, with this growth comes significant security challenges. In his comprehensive survey, Lu[31] presents a multi-faceted examination of the current state of RISC-V security, addressing threats and solutions from both hardware and architectural perspectives.

Lu[31] begins by contextualizing the unique security landscape of RISC-V, noting that its open-source nature, while enabling innovation and customization, also exposes it to a broader attack surface. This differentiates RISC-V from proprietary ISAs like ARM and x86, where source-level access is restricted.

One of the core areas reviewed is hardware and physical security, including threats such as hardware Trojans, fault injection attacks, and reverse engineering. These attacks exploit the physical properties of integrated circuits, and the survey highlights countermeasures like obfuscation techniques and tamper-resistant designs.

The survey also explores hardware-assisted security mechanisms, such as the integration of secure enclaves and Hardware Security Modules (HSMs). These components serve as trusted execution environments, providing isolation and protection for sensitive operations—a concept popularized by technologies like Intel SGX.

Lu dedicates significant attention to ISA-level security extensions, a key advantage of RISC-V due to its modular design. He discusses proposals for adding cryptographic instructions and memory protection features, such as Physical Memory Protection (PMP) and the more sophisticated Supervisor-mode PMP (sPMP). These mechanisms are crucial for enforcing memory isolation, particularly in systems with multiple privilege levels or executing untrusted code.

Side-channel attack (SCA) mitigation is another critical topic. RISC-V, like other architectures, is vulnerable to SCAs that exploit timing, power, or electromagnetic emissions to leak sensitive data. The paper surveys software and hardware approaches to mitigating these risks, including constant-time programming, masking techniques, and hardware randomization.

The paper also provides a comparative analysis with other architectures, notably ARM, to highlight both the progress and limitations of current RISC-V security solutions. Lu notes that while RISC-V offers a flexible platform for implementing custom security features, the ecosystem lacks mature, standardized solutions, especially in areas such as secure boot and remote attestation.

In conclusion, the paper identifies future research directions that include formal verification of hardware modules, development of standard security extensions, and enhancements in secure compilation and toolchains. These areas are essential for ensuring the long-term viability of RISC-V in security-critical applications, particularly in IoT and embedded systems.

2.10 Summary

This section provides a theoretical foundation for memory encryption in RISC-V-based systems by examining the RISC-V instruction set architecture alongside cryptographic primitives suitable for hardware implementation. It distinguishes between block and stream ciphers, analyzing lightweight algorithms such as SIMON, PRINCE, and QARMA for block ciphers, and ChaCha, Grain128, and Trivium for stream ciphers, with a focus on their structural efficiency and applicability in resource-constrained environments. The chapter also reviews DRAM architecture, emphasizing the critical need to protect memory contents against increasing physical and side-channel threats. Furthermore, it outlines the privilege levels and modularity inherent in the RISC-V architecture, highlighting its suitability for integrating customized security features like memory encryption. This groundwork supports the development of secure, high-performance encryption engines tailored specifically for RISC-V systems.

Chapter 3

Literature Review

In this chapter, a number of key concepts and previous work related to memory encryption for RISC-V processors are reviewed. The focus is on analyzing existing lightweight cryptographic algorithms and hardware architectures that enable secure memory access. This review provides a foundation for the design and implementation of efficient encryption mechanisms tailored to the RISC-V platform.

3.1 Block Cipher

Block ciphers are a fundamental component of cryptographic systems, designed to encrypt data in fixed-size blocks using a symmetric key. They are widely used in various applications to ensure data confidentiality and integrity. Block ciphers operate by dividing plaintext into blocks, encrypting each block independently, and then concatenating the resulting ciphertext blocks to form the complete encrypted message. This process is reversed during decryption to retrieve the original plaintext. The design and implementation of block ciphers involve several key aspects, including the choice of structure, key schedule, and security considerations[32, 33].

The PRINCE cipher, introduced by Julia Borghoff et al.[15], addressed a lightweight block cipher specifically designed for applications requiring low-latency encryption, such as hardware-based memory protection. It features a 64-bit block size and a 128-bit key, with an innovative structure that allows for encryption and decryption to be performed with minimal delay. One of its key advantages is the property 'alpha reflection', which enables almost symmetric encryption and decryption operations, thus reducing the complexity of hardware implementations. The compact design of the cipher allows it to fit within constrained hardware environments, and its low critical path delay supports single-cycle encryption when used with pipelined architectures. Yasir Amer Abbas et al.[34] presents a practical hardware implementation of the PRINCE cipher in FPGA, demonstrating its suitability for resource-constrained environments. The study emphasizes the low latency and compact design of PRINCE, which highlights its performance advantages in terms of area and speed when synthesized on FPGA platforms.

Jean J and et al.[35]conducted an in-depth security analysis of the PRINCE block cipher, evaluating its resistance to various cryptanalytic attacks, including differential, linear, and biclique cryptanalysis. Their work confirmed that PRINCE maintains a strong security margin in relation to its lightweight design goals. Despite its simplified structure and low latency, the cipher resists known attacks up to near-full rounds, making it a viable candidate for hardware-level encryption in embedded platforms.

AVANZI Roberto[17] presented QARMA, a new family of lightweight tweakable block ciphers that are targeted at applications such as memory encryption, the generation of very short tags for hardware-assisted prevention of software exploitation, and the construction of keyed hash functions. QARMA is inspired by reflection ciphers such as PRINCE, to which it adds a tweaking input, and MANTIS. However, QARMA differs from previous reflector constructions in that it is a three-round Even-Mansour scheme instead of an FX-construction, and its middle permutation is noninvolutory and keyed. QARMA exists in 64- and 128-bit block sizes, where block and tweak size are equal, and keys are twice as long as the blocks. In[18], the authors explore the security of QARMA-64 and QARMA-128 against truncated differential attacks using MITM methodology. Using the differential enumeration and key-dependent sieve technique, the authors propose an MITM distinguisher on QARMA-64. After that, they extend the distinguisher to achieve a 10-round attack on both versions of QARMA. This is the first result to analyze the security of QARMA against MITM attack.

Studies in[13, 14] explores SIMON as a lightweight block cipher optimized for low-resource devices, balancing security with hardware efficiency. Its configurable design supports multiple security levels and encryption requirements. FPGA implementations demonstrate reduced resource usage compared to similar ciphers, with pipelined designs offering higher throughput and scalar implementations favoring minimal area and energy efficient.

This[36]explores multiple hardware architecture designs for implementing the SIMON64/128 block cipher on Xilinx Spartan-6 FPGAs, with a particular focus on encryption, decryption, and key scheduling functionalities. It also presents a detailed analysis of the design trade-offs and performance outcomes associated with each implementation.

3.2 Lightweight Stream Cipher

ChaCha, a variant of the Salsa20 stream cipher proposed by Daniel J. Bernstein[20], was designed to improve diffusion per round while maintaining high performance in software. It operates on 512-bit state blocks using simple operations such as addition, XOR, and bit rotation, making it highly efficient on general-purpose processors. ChaCha offers excellent resistance

to cryptanalysis and has been widely adopted in security protocols such as TLS and SSH. Its simplicity and speed, combined with strong security guarantees, make it a practical choice for lightweight encryption, especially in constrained environments.

Pfau[37] presents scalable hardware implementations of ChaCha8/12/20, demonstrating that the cipher can be efficiently deployed across a wide performance range—from ultra-compact designs using just 476 FPGA slices to high-throughput variants exceeding 175 Gbit/s. These implementations showcase ChaCha's flexibility and suitability for both constrained and high-performance environments. Similarly, at[38]evaluates compact FPGA designs for ChaCha alongside other cryptographic primitives, confirming its strong performance in area-constrained settings. Both studies highlight ChaCha's efficiency, scalability, and competitive throughput in hardware compared to other stream ciphers.

The Grain-128 cipher, a lightweight stream cipher designed for resource-constrained environments, has been widely analyzed for its security and efficiency in low-power applications. Proposed by Hell et al. [22] and later refined as Grain-128a and Grain-128AEAD, it uses a 128-bit key, 96-bit IV, and a 256-bit internal state with linear and nonlinear feedback shift registers (LFSR and NLFSR). Recent studies,[39], emphasize its hardware efficiency, achieving low gate and high throughput through parallelization, ideal for IoT devices. However, cryptanalysis by Todo et al.[40] revealed vulnerabilities to fast correlation attacks, while Liu and Gao[41] highlighted quantum attack risks using the HHL algorithm, prompting enhancements in Grain-128AEADv2. Research by Sönnerup et al.[42] and Maximov and Hell[43] underscores its role in NIST's lightweight cryptography standardization, though its simplicity poses challenges against differential and fault attacks, balancing performance with security.

Trivium is a lightweight stream cipher inspired by block cipher design principles, emphasizing simplicity and efficiency[23]. Tian et al.[24] provide an in-depth analysis of Trivium's internal structure and demonstrate its resilience against several cryptanalytic attacks, affirming its robustness for practical use. From a hardware perspective, Montoya et al.[25] propose energy-efficient masking techniques to protect Trivium implementations against side-channel attacks, addressing critical security challenges in constrained environments. However, vulnerabilities remain a concern, as highlighted by recent analyses revealing potential weaknesses in FPGA implementations of Trivium, particularly when unprotected against fault and side-channel attacks[26].

3.3 RISC-V Memory Encryption

The RISC-V Instruction Set Manual Volume I: Unprivileged ISA defines the foundational instruction sets for RISC-V, including RV32I, RV64I, and optional extensions such as M (integer multiplication/division), A (atomic operations), F/D (floating-point), and C (compressed instructions). Designed with a focus on simplicity and modularity, the unprivileged ISA provides a compact, orthogonal set of instructions that enable efficient implementation in both low-power embedded devices and high-performance processors. The modular nature allows system designers to include only the necessary features, promoting scalability and reducing hardware overhead. Its open-source specification and clean design have contributed to widespread adoption in both academia and industry, serving as a flexible foundation for custom extensions and secure hardware design[2].

The RISC-V Instruction Set Manual Volume II: Privileged Architecture outlines the essential features required for operating system support, including modes of execution, memory protection, exception and interrupt handling, and control and status registers (CSRs). It defines three privilege levels—machine, supervisor, and user—enabling secure and flexible system software design. The specification also introduces mechanisms such as Physical Memory Protection (PMP) and support for virtual memory, which are critical for isolation and multitasking in modern computing environments. Its modular and open design allows for implementation across a wide range of platforms, from embedded systems to full-featured operating systems, making it a foundational component for secure and extensible RISC-V-based processor development[3].

The base integer instruction set, RV32I/RV64I, forms the core of the RISC-V architecture and is designed for simplicity, portability, and hardware efficiency. The specification defines a small set of orthogonal instructions that support basic arithmetic, logic, control flow, and memory operations, making it well-suited for both academic research and commercial use [2]. Its clean design enables straightforward hardware implementation and compiler support, facilitating adoption in various domains ranging from microcontrollers to high-performance systems. Complementing the base ISA, the RISC-V privileged architecture specification defines the control and status registers (CSRs), exception handling, interrupt mechanisms, and memory protection schemes necessary for supporting operating systems and virtualization[3]. Together, these foundational specifications enable a modular and extensible platform, allowing designers to build both minimalist and complex computing systems while maintaining compatibility and software portability.

The paper [44] by Enfang Cui, Tianzheng Li, and Qian Wei, provides a comprehensive overview of the RISC-V ISA extensions landscape. It categorizes the extensions into standard and non-

standard types, detailing their functionalities and applications. The survey emphasizes the modularity and openness of RISC-V, which facilitate the development of custom extensions tailored to specific application domains such as artificial intelligence, Internet of Things (IoT), and high-performance computing. Additionally, the authors discuss the challenges and future directions in the evolution of RISC-V ISA extensions, highlighting the need for standardized toolchain support and the potential for further innovation in this area.

The paper by Asanović et al.[29] introduces a parameterizable system-on-chip (SoC) generator based on the RISC-V architecture. It provides a flexible and reusable framework for designing and customizing processor cores and memory systems using Chisel, a hardware construction language. The Rocket Chip integrates components such as the Rocket core (a 64-bit in-order RISC-V processor), memory hierarchies, and peripheral interfaces, enabling rapid prototyping and evaluation of architectural extensions. Its modularity makes it a widely adopted platform for academic and industrial research on RISC-V-based systems, including security features like memory encryption. The generator's scalability and open-source availability have significantly accelerated innovation in RISC-V development. Recent advancements in RISC-V processor design leverage the flexibility of the Rocket Chip generator to enhance hardware and software integration as well as computational capabilities. Savas et al.[45] propose an automated framework for generating both hardware descriptions and corresponding software toolchains for RISC-V cores created with Rocket Chip, streamlining development and ensuring hardware-software consistency. Complementing this, Arunkumar et al.[46] present PERC (Posit Enhanced Rocket Chip), which integrates the posit numerical format into the Rocket Chip architecture to improve arithmetic precision and efficiency. Together, these works demonstrate the adaptability of the Rocket Chip generator in supporting rapid customization and novel architectural extensions suited for diverse computing applications.

Mofrad et al.[47] present a comparative study of two major hardware-based memory protection mechanisms: Intel Software Guard Extensions (SGX) and AMD Secure Memory Encryption (SME). The paper evaluates their architectural differences, security guarantees, and performance overheads. While SGX provides fine-grained enclave-based protection with support for secure computation, SME offers full memory encryption with minimal software modifications. The authors highlight that SGX is more suited for secure computation use cases, whereas SME delivers better system-level transparency and performance. This comparison provides valuable insights into the trade-offs between isolated execution and broad memory encryption strategies in modern processor architectures.

Elbaz et al.[48]addressed a comprehensive survey of hardware mechanisms for memory authentication, focusing on techniques designed to ensure data integrity and prevent tampering in secure computing systems. The paper reviews existing approaches, including Merkle trees,

block-level integrity trees, and hardware-based engines that support secure memory architectures. The authors analyze each method in terms of performance, storage overhead, and security guarantees, offering a valuable framework for understanding the trade-offs involved in deploying memory authentication solutions. This work serves as a foundational reference for hardware designers seeking to integrate robust integrity-checking mechanisms into modern secure processors.

Gueron[1] proposes a memory encryption engine (MEE) optimized for general-purpose processors, focusing on low-latency and high-throughput secure memory access. The engine is based on AES in counter (CTR) mode and includes additional components to mitigate side-channel attacks and protect against replay and tampering. The design emphasizes minimal performance overhead while preserving strong confidentiality guarantees, making it suitable for commercial CPUs. Gueron also evaluates implementation challenges, such as key management and metadata storage, providing practical considerations for integrating encryption at the memory controller level.

Pessl et al.[49] introduce DRAMA, a novel class of side-channel and covert-channel attacks that exploit DRAM row buffer behavior and address mapping to perform cross-CPU attacks without requiring shared memory. By reverse-engineering DRAM addressing functions, the authors demonstrate that attackers can leverage timing differences caused by row conflicts to infer sensitive information or establish high-bandwidth communication channels across processor boundaries. The study highlights the risks of microarchitectural leakage in shared memory subsystems and underscores the need for stronger memory isolation techniques, especially in multi-tenant and cloud environments.

Recent studies have exposed critical weaknesses in modern memory subsystems through both physical and microarchitectural attacks. Cold Boot Attacks, as revisited by Yitbarek et al.[4], demonstrate that residual data in DRAM remains accessible after power-off, enabling key recovery even in systems with memory scrambling. Spectre, introduced by Kocher et al.[7], exploits speculative execution in CPUs to leak data across isolation boundaries, posing significant challenges for software and hardware mitigations. Rowhammer, as described by Seaborn and Dullien[5], leverages frequent DRAM row accesses to induce bit flips in adjacent rows, allowing attackers to corrupt memory without direct access.Lipp et al.[6] introduce Meltdown, a microarchitectural vulnerability that allows user-level applications to access privileged kernel memory by exploiting out-of-order execution. The attack bypasses conventional memory protection by transiently executing unauthorized memory loads and then leaking the accessed data through cache-based side channels. Meltdown is particularly impactful on processors that do not enforce proper privilege checks before speculative execution, notably affecting Intel CPUs. The study demonstrates that speculative behaviors, once considered benign performance opti-

mizations, pose serious threats to system confidentiality and require both hardware and software mitigations to ensure isolation between user and kernel space.

Recent studies have significantly contributed to enhancing secure memory systems in RISC-V-based platforms. Wong et al.[11] introduce SMARTS, a memory assurance framework that enforces trusted access and runtime integrity verification in RISC-V SoCs. Cilardo[8] presents an FPGA-based RISC-V architecture with integrated memory encryption, emphasizing practicality and configurability. Yin et al.[9] propose HPME, a high-performance encryption engine integrated with Trusted Execution Environments (TEEs) in RISC-V, delivering strong protection with minimal latency. Lu designs a programmable PCIe encryption system leveraging RISC-V cores for secure high-speed communication. Meanwhile, Zhang et al.[50] develop Sealer, an in-SRAM AES-based encryption engine that offers high throughput and reduced area and power consumption, making it ideal for lightweight, secure embedded systems. Collectively, these works underscore the versatility of RISC-V in enabling efficient and scalable hardware-based memory protection mechanisms.

3.4 Summary

Overall, this review covers critical concepts and prior work related to memory encryption for RISC-V CPUs, with a focus on lightweight cryptographic algorithms and secure hardware architectures. It examines block ciphers such as PRINCE, QARMA, and SIMON for their low-latency and area-efficient designs suited to resource-constrained environments. Stream ciphers like ChaCha, Grain-128, and Trivium are analyzed for their energy-efficient, high-throughput, and resilient implementations. The modular and extensible nature of the RISC-V architecture is emphasized, particularly through frameworks like Rocket Chip, which support secure and customizable processor development. Memory encryption approaches are explored in depth, including Intel SGX, AMD SME, and RISC-V-based solutions such as SMARTS, HPME, and Sealer, highlighting the trade-offs between performance, data integrity, and resistance to physical and side-channel attacks. Collectively, this survey establishes a foundation for the design of efficient and secure memory encryption mechanisms tailored to the flexibility and scalability of RISC-V platforms.

In general, a comprehensive review is conducted on key concepts and previous work related to memory encryption for RISC-V processors. The focus is on analyzing existing lightweight cryptographic algorithms and hardware architectures that enable secure memory access. This review provides a foundation for the design and implementation of efficient encryption mechanisms tailored to the RISC-V platform.

Chapter 4

Methodology

This chapter outlines the systematic approach undertaken to analyze and design a memory encryption mechanism tailored for the RISC-V CPU architecture. The methodology begins with threat modeling and requirements definition to identify security vulnerabilities in unprotected memory systems and establish clear design objectives. It then proceeds to review and evaluate existing memory encryption techniques, including cryptographic primitives, encryption granularities, and hardware/software trade-offs in modern processors. Next, architectural design and integration analysis focus on effectively embedding memory encryption within the RISC-V memory hierarchy, such as caches, memory encryption units (MEUs), and bus interfaces. Finally, implementation and performance evaluation use simulation tools and FPGA-based platforms to assess the impact of encryption on system performance, area utilization, and security resilience.

4.1 The Proposed System

The proposed system of memory encryption RISC-V CPU figure 4.1 illustrates a structured methodology for designing memory encryption in a RISC-V CPU and a systematic approach for integrating encryption within a processing system, focusing on selecting and applying different cipher algorithms based on system needs.

4.1.1 Algorithm Analysis

The flow starts with Algorithm Analysis, where the requirements and constraints of the system (such as security needs, performance trade-offs, and hardware constraints) are assessed. This evaluation determines the type of cipher most suitable for the intended application. Evaluates the suitability of different cryptographic algorithms for memory encryption. The analysis of cryptographic algorithms are considers both block ciphers and stream ciphers, focusing on lightweight implementations suitable for memory security systems.

- **Block Ciphers**: Suitable for scenarios requiring secure bulk data encryption, often providing robust security at the cost of additional overhead.

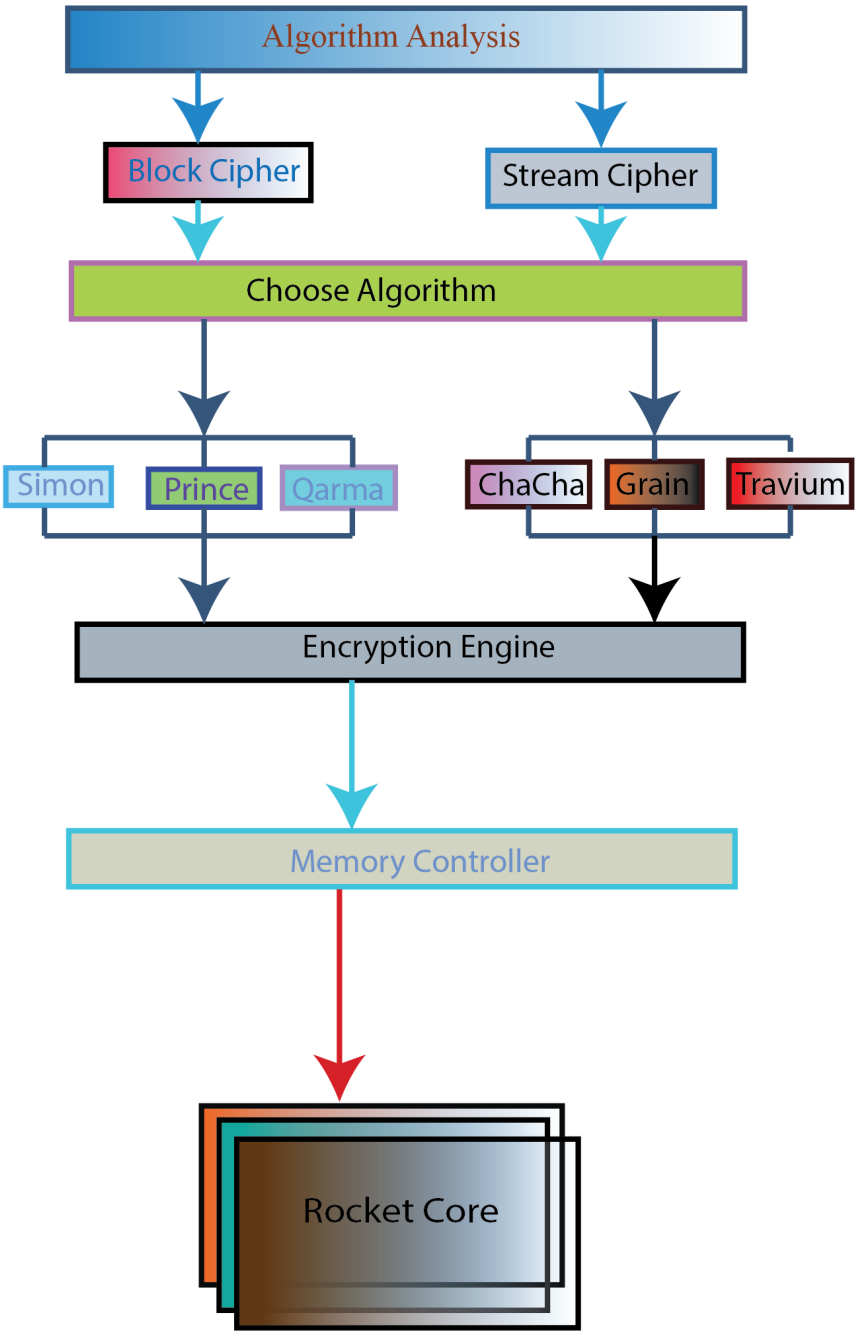


Figure 4.1: Proposed Memory Encryption Methodology for RISC-V.

- **Stream Ciphers**:Stream Cipher: More suitable for environments demanding lower latency and efficient real-time data encryption.

The next step involves selecting an appropriate algorithm based on factors such as security strength, performance requirements, and hardware constraints. Once the algorithm is chosen, it is implemented within the Encryption Engine, which serves as the core component responsible for encrypting and decrypting memory data.

Each type of cipher leads to the next critical stage: Choosing the Specific Algorithm. Based on the initial analysis, an appropriate algorithm is selected from a curated set of lightweight, hardware-efficient encryption options.

The decision depends on Security requirements Performance constraints Hardware resources available in the RISC-V implementation.

Block Cipher

- Lightweight block ciphers like **Simon**, **Prince**, and **Qarma** are evaluated for their efficiency and security properties.
- These algorithms operate on fixed-size blocks of data, making them suitable for scenarios requiring deterministic encryption.
- **Simon**: A lightweight block cipher designed for optimal performance in hardware-constrained environments.
- **Prince**: Known for its low latency and efficient hardware implementation.
- **Qarma**: A tweakable block cipher supporting more flexible encryption parameters.

The security and performance of block ciphers are heavily influenced by their key size and the number of rounds in their encryption process. A larger key size generally provides stronger resistance against brute-force attacks, while the number of rounds determines how thoroughly the plaintext is diffused and confused to resist cryptanalysis. Lightweight block ciphers, designed for resource-constrained environments, often balance these factors by selecting moderate key sizes—typically 64 to 128 bits—and a carefully optimized number of rounds that ensures sufficient security without excessive computational overhead. For example, SIMON, PRINCE, and QARMA use different iteration counts tailored to their structural design and security goals, aiming to maintain both efficiency and robustness in hardware implementations.

Table 4.1: Key Size and Number of Rounds for Block Ciphers

Cipher	Block Size (bits)	Key Size (bits)	Rounds
SIMON-32/64	32	64	32
SIMON-64/128	64	128	44
PRINCE-64/128	64	128	12
QARMA-64/128	64	128	20

Table 4.1: shows Key Size and Number of Rounds for Block Ciphers summarizes the key lengths and the corresponding number of rounds used in various common block cipher algorithms.

Stream Cipher

- Lightweight stream ciphers like **ChaCha**, **Grain**, and **Trivium** are assessed for their speed and resource utilization.
- Stream ciphers generate a continuous keystream that can be XORed with plaintext, offering flexibility for variable-length data.
- **ChaCha**: Recognized for high-speed performance and resistance to cryptographic attacks.
- **Grain**: Suitable for very low-power applications, often used in IoT devices.
- **Trivium**: Offers simple hardware implementation with a strong security margin.

ChaCha-8 is a stream cipher that uses a 256-bit key and a 512-bit internal state organized as sixteen 32-bit words. Its hardware implementation relies on simple arithmetic operations such as addition, XOR, and bitwise rotation, enabling high throughput and low latency. The 32-bit parallelism fits well with common hardware word sizes, making ChaCha-8 suitable for FPGA and ASIC designs with pipelined or fully unrolled architectures that maximize speed.

In contrast, Trivium and Grain-128 prioritize hardware simplicity and efficiency. Trivium uses an 80-bit key and a 288-bit internal state composed of three nonlinear feedback shift registers, producing one bit per clock cycle with minimal logic using only XOR and AND gates. Grain-128 employs a 128-bit key with a 160-bit internal state, combining a linear feedback shift register (LFSR) and a nonlinear feedback shift register (NFSR). It also outputs one bit per cycle and relies on simple shift register logic and bitwise operations, offering a balance between throughput and resource usage, making both ciphers ideal for lightweight hardware and low-power embedded applications.

Table 4.2: Key Size and Bit in Block (Stream Cipher) presents the key lengths and block sizes for various stream ciphers.

Table 4.2: Key Size and Bit in Block (Stream Cipher)

Cipher	Bit in Block	Key Size (bits)
ChaCha-8	32	256
Trivium	32	80
Grain128	32	128

4.1.2 Implementation Encryption Engine

Once the algorithm is selected, the data is processed through the Encryption Engine. This engine is responsible for implementing the algorithm's core cryptographic operations, ensuring data confidentiality as it passes through the system. Develop the Encryption Engine based on the selected algorithm to perform encryption and decryption operations. Integrated with the Memory Controller to ensure that all memory transactions are secured. The encryption engine abstracts the complexity of each cipher, providing a uniform interface for the subsequent hardware components to interact with encrypted data efficiently. The Encryption Engine is integrated with the Memory Controller, which manages the flow of data between the CPU and memory. The Memory Controller ensures that all memory accesses (both reads and writes) are routed through the Encryption Engine for secure processing. Finally, the encrypted or decrypted data is communicated to the Rocket Core, which represents the central processing unit of the RISC-V system.

The designed Encryption Engine is implemented in Verilog and supports both block ciphers (PRINCE, Simon, Qarma) and stream ciphers (ChaCha, Grain, Trivium). A random key generator, based on a Pseudo-Random Number Generator (PRNG), is used to dynamically produce encryption keys. These keys are securely distributed to the cipher modules within the engine. The engine is directly integrated with the memory controller, ensuring that all data written to or read from external memory is encrypted or decrypted transparently. It interfaces seamlessly with a RISC-V Rocket Core, operating between the L2 cache and DRAM, thereby securing off-chip memory while maintaining system compatibility and performance.

The SIMON, PRINCE, and QARMA ciphers can be efficiently implemented in Verilog using modular design principles. For SIMON variants, the Feistel network structure is realized through round modules containing the key schedule, rotation operations, and XOR logic, with the 32/64 version requiring 32 rounds and 64/128 requiring 44 rounds. PRINCE's α-*reflection* property allows encryption and decryption to share most logic, with the core implementation focusing on its 4-bit S-boxes and MDS matrix multiplication. QARMA's tweakable structure requires additional key/tweak mixing logic alongside its substitution-permutation network.All designs benefit from pipelining for throughput-critical applications, with SIMON's simple operations enabling particularly compact implementations. PRINCE's low-latency design makes it suitable for unrolled implementations when area isn't constrained. QARMA's S-boxes and linear layers can be optimized using shared combinatorial logic.

The lightweight block ciphers SIMON 32/64, SIMON 64/128, PRINCE 64/128, and QARMA 64/128 are well-suited for hardware implementation using Verilog due to their compact and efficient design. In Verilog, each cipher can be modularized into components such as key scheduling, round functions, and control units. For example, SIMON's structure primarily uses simple

operations like AND, XOR, and left/right shifts, making it highly optimized for FPGA and ASIC environments. PRINCE and QARMA, which use S-box substitutions and linear layers, require slightly more logic resources but maintain low latency, especially in pipelined or unrolled configurations.

The Verilog implementation focuses on balancing area, throughput, and power consumption. Each cipher's round transformation is implemented as a combinational logic block, and a finite state machine (FSM) controls the round iteration. Key expansion modules are also coded to preload round keys efficiently. These implementations are typically synthesized and tested on FPGA platforms (Artix®-7 35T FPGA Arty Evaluation Kit), where performance metrics like area utilization, clock frequency, and power usage are evaluated. Overall, these ciphers demonstrate strong compatibility with Verilog and are effective choices for secure, low-resource embedded systems.

Listing 4.1 depicts the Verilog module interface and internal signals for the Simon 64/128 encryption core. The module defines input ports for the clock (`clk`), reset (`rst`), start signal (`start`), a 64-bit plaintext input, and a 128-bit key. It also includes output ports for the 64-bit ciphertext and a done signal to indicate the completion of encryption.

The internal logic includes two 64-bit registers (`clk` and `rst`) used to store intermediate values during the encryption rounds. A 6-bit round counter keeps track of the iteration process, and the Simon-specific round function `f` performs the necessary bitwise transformations. The `always` block implements the sequential encryption logic, executing 44 rounds of operations. Once all rounds are completed, the final ciphertext is assigned, and the done flag is asserted. This listing shows the structure and control flow of a hardware-implemented Simon 64/128 cipher, suitable for integration with secure RISC-V memory encryption systems.

Listing 4.1: Simon 64/128 module interface and internal signals

```
module simon_64_128 (
    input wire clk,
    input wire rst,
    input wire [63:0] plaintext,  // 64-bit input bus where the unencrypted data
        block is provided.
    input wire [127:0] key,       // 128-bit key
    input wire encrypt,           // 1=encrypt, 0=decrypt
    output reg [63:0] ciphertext  // 64-bit output block
);

    // Internal state and key schedule
    reg [63:0] left_reg, right_reg;
    reg [127:0] key_schedule [0:43]; // 44 round keys (for 44-round SIMON 64/128)
    integer i;

    // SIMON round function constants
    parameter [31:0] z_sequence = 32'h7369F885; // Z-value for key expansion
    wire [63:0] round_constant = 64'hFFFFFFFFFFFFFFFC; // Round constant

endmodule
```

Listing 4.2 module interface and key expansion signals describes the module interface and key expansion signals for the Prince 64/128 encryption core. The module includes input ports for clock (`clk`), reset (`rst`), start signal (`start`), 64-bit plaintext, and a 128-bit key. The output ports provide the 64-bit ciphertext and a done signal indicating encryption completion. Internal signals include an array of 128-bit registers used for storing round keys during the key expansion process. A 4-bit round counter tracks the progress through encryption rounds. The design implements the key schedule and encryption rounds consistent with the Prince cipher specification.

Listing 4.2: Prince 64/128 module interface and key expansion signals

```
module prince_64_128 (
    input wire clk,
    input wire rst,
    input wire [63:0] plaintext,  // 64-bit input block
    input wire [127:0] key,       // 128-bit key (k0 || k1)
    input wire encrypt,           // 1=encrypt, 0=decrypt
    output reg [63:0] ciphertext  // 64-bit output block
);
    // Key expansion
    wire [63:0] k0 = key[127:64];  // First 64 bits of key
    wire [63:0] k1 = key[63:0];    // Last 64 bits of key
    wire [63:0] k0_prime = {k0[62:0], k0[63]} ^ {k0[30:0], k0[63:31]}; //  (k0)
    wire [63:0] round_constants [0:11]; // Round constants (RC0-RC11)
endmodule
```

Listing 4.3 outlines the top-level Verilog module interface for the QARMA-64/128 block cipher, including input/output definitions, key splitting into k_0 and k_1, tweak input, and the use of a fixed constant α used in the cipher's internal transformation.

Listing 4.3: QARMA 64/128 module interface and key/tweak expansion

```
module qarma_64_128 (
    input wire clk,
    input wire rst,
    input wire [63:0] plaintext,  // 64-bit input block
    input wire [127:0] key,       // 128-bit key (k0 || k1)
    input wire [63:0] tweak,      // 64-bit tweak (for tweakable mode)
    input wire encrypt,           // 1=encrypt, 0=decrypt
    output reg [63:0] ciphertext  // 64-bit output block
);

    // Key and tweak expansion
    wire [63:0] k0 = key[127:64];    // First 64 bits of key
    wire [63:0] k1 = key[63:0];      // Last 64 bits of key
    wire [63:0] alpha = 64'hC0AC29B7C97C50DD; // QARMA constant (similar to PRINCE)

endmodule
```

The ChaCha8 stream cipher can be efficiently implemented in Verilog using a 512-bit state register composed of 16 words and unrolled round logic. Each round performs 16 quarter-round operations based on ARX primitives—modular addition, XOR, and fixed rotations—on 32-bit words. The design includes a state initialization module to load the key, nonce, and counter, followed by eight rounds of transformation, and a final state addition stage. Quarter-round operations can be parallelized, enabling a throughput of one 64-byte block every 8–16 cycles in typical FPGA implementations, with optional pipelining to further increase performance.

Listing 4.4: ChaCha module interface and signals

```
module chacha(
    input wire clk,
    input wire reset_n,
    input wire cs,
    input wire we,
    input wire [7 : 0] addr,
    input wire [31 : 0] write_data,
    output wire [31 : 0] read_data
);

wire [255 : 0] core_key;
wire [63 : 0]  core_iv;
wire           core_ready;
wire [511 : 0] core_data_in;
wire [511 : 0] core_data_out;
wire           core_data_out_valid;

reg [31 : 0]   tmp_read_data;

endmodule
```

Listing 4.4 illustrates the ChaCha module interface and internal signal declarations, including the input control signals (`clk`, `reset_n`, `cs`, `we`, `addr`, and `write_data`) and output signal `read_data`, as well as internal wires for the 256-bit key, 64-bit IV, 512-bit input/output data paths, and associated control signals for data readiness and validity.

Trivium's 288-bit shift register structure maps naturally to Verilog, using three interconnected feedback loops of 94, 91, and 111 bits. The implementation includes logic for key and IV loading, a warm-up phase of 1152 clock cycles managed by a counter, and continuous keystream

generation through a nonlinear combination function. Each clock cycle produces 1 bit of output, though the design can be parallelized for 32- or 64-bit outputs. With critical paths involving only 2–3 XOR gates, the design enables high-frequency operation exceeding 300 MHz on modern FPGAs.

Listing 4.5: Trivium module definition and signals

```
module trivium (
    input wire clk,
    input wire rst,
    input wire [79:0] key,     // 80-bit key
    input wire [79:0] iv,      // 80-bit initialization vector (IV)
    output wire keystream_bit // Output keystream bit
);

// Trivium's 288-bit internal state (registers A, B, C)
reg [92:0] A;  // 93 bits (s1 to s93)
reg [83:0] B;  // 84 bits (s94 to s177)
reg [110:0] C; // 111 bits (s178 to s288)

// Feedback and update wires
wire A_feedback, B_feedback, C_feedback;
wire A_input, B_input, C_input;
wire t1, t2, t3;

endmodule
```

Listing 4.5 shows the top-level Verilog module definition for the Trivium stream cipher, including the 80-bit key and IV inputs, a single-bit keystream output, and internal 288-bit state registers split across three shift registers (A, B, and C). It also defines the intermediate wires used for feedback and update logic necessary for the cipher's nonlinear state transitions.

Grain-128a's implementation combines a 128-bit LFSR and 128-bit NFSR with nonlinear output filtering. The Verilog design features parallel loading of a 128-bit key and 96-bit IV, 256 initialization clocks using a counter, and keystream generation with optional MAC computation. The feedback functions (LFSR: g(x), NFSR: f(x)) can be implemented as combinatorial XOR networks. For area optimization, the design can process 1 bit per cycle, requiring 800 LUT. Throughput-optimized versions implement 8- or 32-bit parallel output and require 1500 to 2000 LUTs. The authentication feature requires additional accumulation logic for the 64-bit MAC.

Listing 4.6: Grain-128 module definition and signals

```
module grain128 (
    input wire clk,
    input wire rst,
    input wire [127:0] key,
    input wire [95:0] iv,
    output wire keystream_bit
);

// Registers for LFSR (128-bit) and NFSR (128-bit)
reg [127:0] lfsr;
reg [127:0] nfsr;

// Feedback taps for LFSR and NFSR
wire lfsr_feedback;
wire nfsr_feedback;

// Output function wires
wire h_func;
wire y_func;

endmodule
```

Listing 4.6 shows the Verilog module definition for the Grain-128 cipher, including the inputs, outputs, and internal registers for the 128-bit LFSR and NFSR. It also declares wires for the feedback functions and output logic, outlining the basic signal structure without the internal combinational or sequential logic.

4.1.3 Memory Controller

This component manages the interface between the encrypted data path and system memory. It ensures all read/write operations are securely routed through the Encryption Engine and maintains proper synchronization with the CPU. After encryption, the output needs to be managed appropriately. Here, the Memory Controller takes charge. The memory controller interfaces with encrypted data, ensuring that it is securely stored or transmitted to the next processing unit without leakage or tampering. The memory controller manages the communication between the CPU and memory subsystems, ensuring that all memory accesses—both reads and writes—are

routed through the Encryption Engine. This guarantees end-to-end confidentiality of data during transmission and storage.

The Figure 4.2 depicts the memory hierarchy of a System on Chip (SoC) with an integrated Memory Encryption (ME) module. The SoC consists of multiple layers of memory, starting from the Core, which includes the L1 Cache as the closest and fastest memory component. Data flows between the L1 Cache and the L2 Cache, which serves as a larger but slower cache layer. Beyond the L2 Cache lies the Memory Controller, where the Memory Encryption (ME) module is embedded. This ME module ensures that data is encrypted before being sent to the DRAM (Dynamic Random Access Memory), providing confidentiality for sensitive information stored in main memory. By placing the encryption logic within the Memory Controller, the system achieves efficient protection against physical attacks such as cold-boot or DMA-based snooping, while maintaining performance by encrypting data at the memory interface. Memory encryption is strategically integrated within or tightly coupled with the memory controller in modern SoCs. This placement offers a centralized and efficient mechanism to encrypt all data destined for external DRAM and decrypt data retrieved from it. By operating at this level, the memory controller ensures transparent data protection without requiring explicit involvement from the processing core, thereby enhancing system security against physical attacks and unauthorized data access while maintaining performance. The encryption engine uses secure keys, often tied to a hardware root-of-trust (e.g., AMD's SME/SEV or Intel's MKTME), protecting against physical attacks like bus snooping or cold-boot exploits. By embedding encryption directly in the memory controller, the SoC maintains low latency, minimizes software overhead, and enforces robust security without requiring modifications to existing software or CPU architectures. We propose a lowRISC-compatible lightweight Memory Encryption Unit system to enable secure memory guarantee. In the subsections that follow, a possible memory access protection strategy is defined, considering the lightweight design of RISC-V SoC, as well as the outlined model threat and needs, and reasoning from security issues in existing technologies. A modern processor has an internal cache that obliges a little sum of memory, and can be become significantly faster than the system memory. Amid typical operation, memory transactions are persistently issued by the processor's Core, while transactions that miss the cache

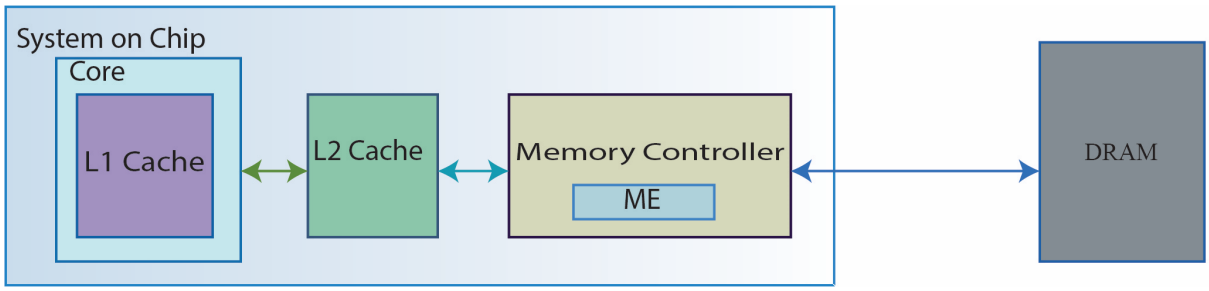


Figure 4.2: Schematic illustration of the Memory Encryption Unit (MEU) operation.

are dealt with by the Memory Controller (MC). The Memory Encryption (ME) functions as an extension/integration with memory controller (MC), taking over the cache-DRAM traffic that points to what is called the Protected data region. The memory controller routes read/write requests to the protected region to the memory encryption (ME), which encrypts (decrypts) the data before delivering it to (from) the DRAM[1]. The Memory Encryption Unit operations are depicted schematically in Figure.

As a result, data memory will be mapped to the allotted region based on its address. Prior to and after the encapsulation, data designated in the trusted region will be transmitted forward to the MPU for authorized encryption. This function protects the secrecy of sensitive data in the DRAM while also ensuring minimum leakage of the memory data's address/location mapping. Furthermore, PME, in general, is effective in reducing memory overhead to a minimum. Because other software is prevented from changing the area allocation, sensitive data is protected from being maliciously written or read to an untrusted zone[11].

The deployed cryptographic primitives are authorized to partition the data memory address spaces into blocks of preset size within the trusted region, based on the deployed encryption scheme and the hardware specifications (such as the cache line size). The block size chosen has a significant impact on the trade-off between metadata overhead, access granularity, and speed. As a result, the aforementioned qualities have imposed restrictions on software access to the DRAM, preventing side-channel software leakage indirectly[11].

Figure 4.3 illustrates the architecture of memory access in a Rocket Chip-based SoC with integrated encryption. The Rocket Tile interfaces with the L2 cache bus via `io_cached` and `io_uncached` ports for cached and uncached memory operations, respectively. The L2 cache bus manages memory requests and responses, serving as a central point for all traffic between the processor core and the main memory subsystem.

In this design, the memory encryption unit is tightly integrated within the memory controller itself, rather than being a standalone component. When the L2 cache sends an Acquire request, it is passed to the memory controller, where the embedded encryption unit performs encryption or decryption as needed. Encrypted data is written to memory through nasti.w, and decrypted data is returned through nasti.r along with the Grant signal. This integration simplifies the hardware interface and reduces latency while maintaining data confidentiality, using TileLink/NASTI protocol standards implemented in Chisel and Verilog.

In this design, the memory encryption unit is tightly integrated within the memory controller itself, rather than being a standalone component. When the L2 cache sends an `Acquire` request, it is passed to the memory controller, where the embedded encryption unit performs encryption or decryption as needed. Encrypted data is written to memory through `nasti.w`, and decrypted data is returned through `nasti.r` along with the `Grant` signal. This integration simplifies the

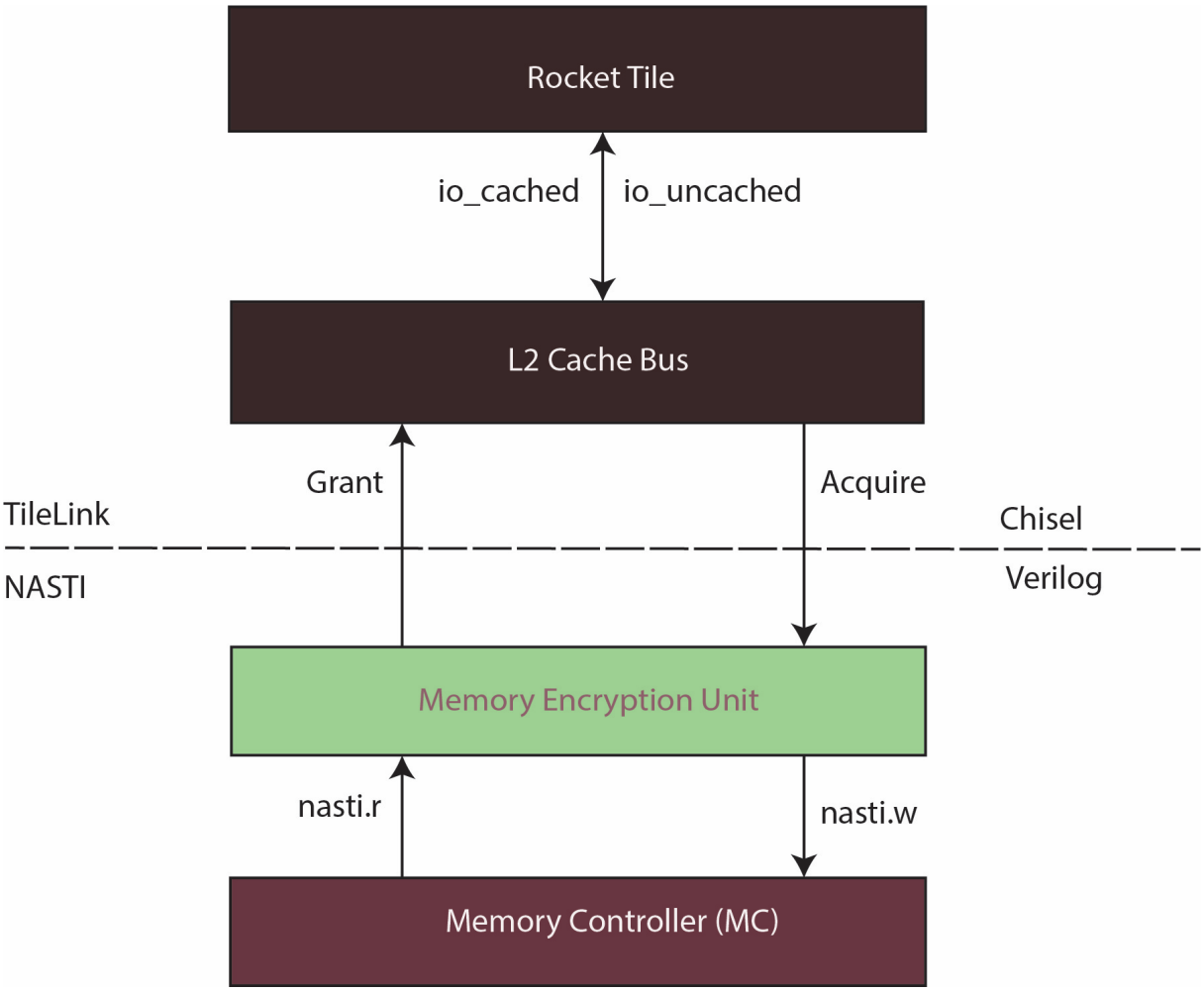


Figure 4.3: Bus Interface for Memory Encryption Unit in RISC-V Rocket.

hardware interface and reduces latency while maintaining data confidentiality, using TileLink/-NASTI protocol standards implemented in `Chisel` and `Verilog`.

4.1.4 Rocket Core

The final stage of the flow leads to the Rocket Core, a RISC-V based processor core. The Rocket Core retrieves the encrypted (and later decrypted) data seamlessly, without needing to handle the complexities of the encryption process itself. This is the RISC-V based Rocket Core CPU, which processes all instructions and interacts with memory through the controller. It is the central computing element relying on secure data transfer provided by the encryption and memory layers. The Rocket Core represents the RISC-V CPU core and relies on the Memory Controller to securely access encrypted memory. It interacts with the Memory Controller to execute instructions and retrieve data, without directly managing the encryption or decryption processes. This separation ensures that cryptographic operations are abstracted away from the

core, maintaining both performance and security.

The figure 4.4 illustrates a high-level architecture of a System-on-Chip (SoC) designed with integrated memory encryption. The SoC includes multiple Rocket cores, each equipped with private L1 caches. These cores are connected via a coherent interconnect fabric that facilitates consistent memory access and data sharing across cores. The coherent fabric interfaces with a shared L2 cache, which acts as a unified buffer before reaching the main memory subsystem.

A key feature of this design is the integration of a memory encryption unit within the memory controller (MC), positioned between the L2 cache and DRAM. This unit ensures that data written to and read from DRAM is transparently encrypted and decrypted, enhancing system security. In parallel, the I/O fabric connects the processor cores to various peripherals including ROM, BRAM, SPI, UART, Ethernet, and Debug modules, enabling robust communication and peripheral management without compromising the encrypted memory flow.

The figure 4.4 presents a high-level architectural overview of a System-on-Chip (SoC) that incorporates secure memory access through a tightly integrated memory encryption mechanism. At its core, the system consists of multiple Rocket cores, each with private L1 caches, interconnected via a coherent fabric that maintains data consistency across processors. This fabric connects to a shared L2 cache, which serves as an intermediate storage layer between the processor cores and the main memory subsystem. The coherent design ensures low-latency communication and synchronization between the cores, which is critical for parallel processing.

A notable security enhancement in this architecture is the integration of a Memory Encryption Unit (MEU) within the Memory Controller (MC). Positioned between the L2 cache and DRAM, the MEU ensures that all data leaving the processor subsystem is encrypted before being stored in external memory, and decrypted upon retrieval. This transparent encryption mechanism protects sensitive data against physical attacks such as cold boot attacks or direct memory probing. The memory controller handles address translation, memory scheduling, and bandwidth management, while also collaborating closely with the MEU to ensure that cryptographic operations do not introduce significant performance penalties. Together, the MEU and MC form a trusted computing base for memory access, balancing both security and system efficiency. The SoC also includes an I/O fabric that connects peripherals such as ROM, BRAM, SPI, UART, Ethernet, and Debug modules, supporting a wide range of embedded applications.

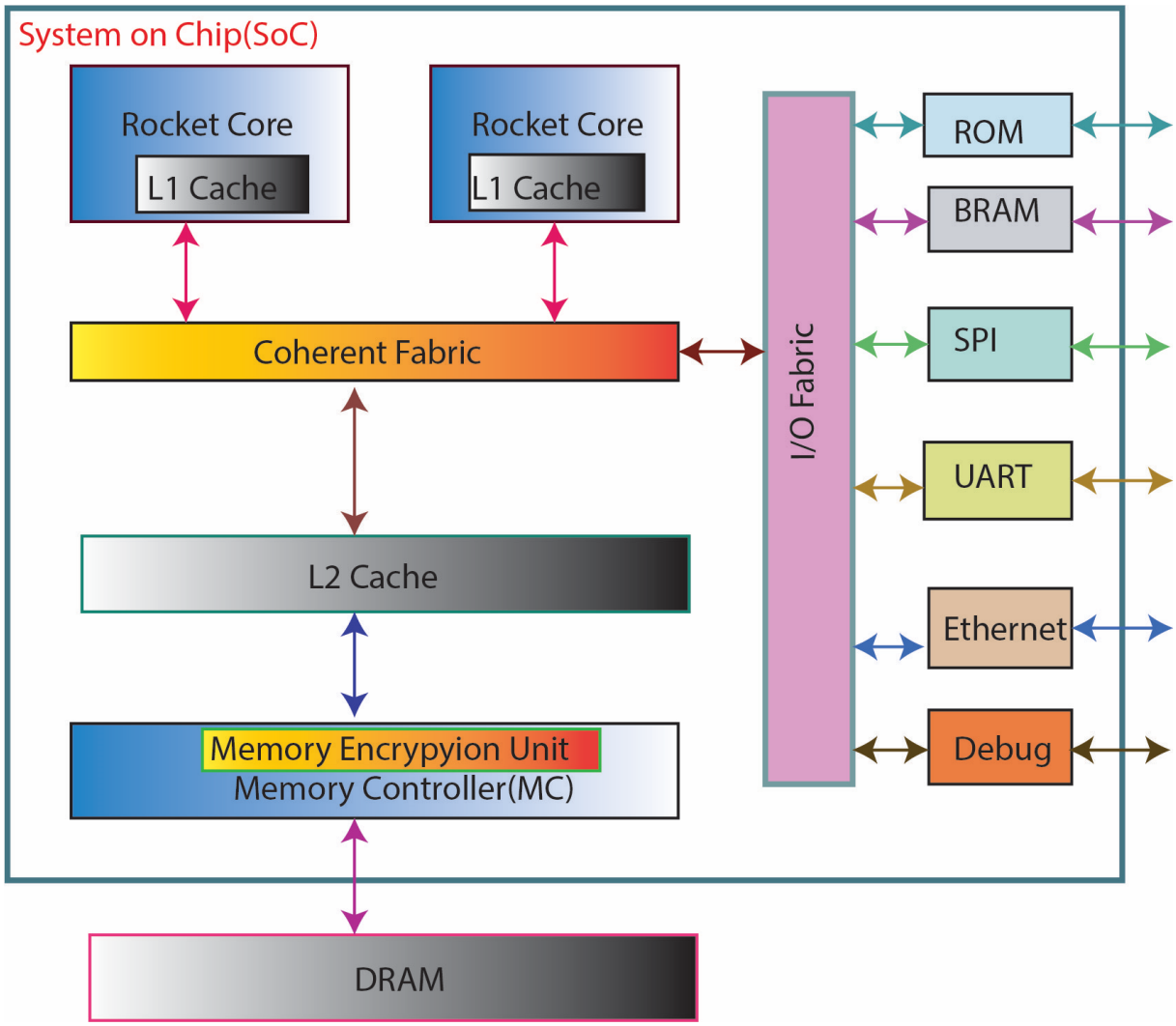


Figure 4.4: Schematic lowRISC Architecture MEU is integrated to the Memory Controller.

4.2 Summary

The methodology focuses on designing and implementing a secure memory encryption scheme for RISC-V systems using lightweight cryptographic algorithms. It begins with selecting suitable block and stream ciphers—specifically QARMA, PRINCE, SIMON, ChaCha, Grain128, and Trivium—based on their efficiency and hardware suitability. These algorithms are implemented in Verilog and integrated into the memory controller of a RISC-V Rocket Core. The encryption engine is inserted between the L2 cache and DRAM, ensuring on-the-fly data encryption and decryption. Hardware modules for each cipher are developed and synthesized on an FPGA platform to assess performance, area utilization, and power consumption. The methodology also involves security analysis and evaluation of trade-offs between encryption strength and system overhead, aiming to provide a practical, low-cost solution for enhancing memory confidentiality in RISC-V-based architectures.

Chapter 5

Result and Discussion

In this chapter, we discuss the experiments carried out to evaluate the proposed memory encryption module for a RISC-V CPU and present the obtained results. Specifically, we will detail the impact of the encryption scheme on key hardware metrics, including the resulting area overhead, power consumption, and introduced delay. The subsequent discussion will then analyze these findings to quantify the overall overhead and explore the implications of these trade-offs for practical implementations. We evaluates the effectiveness of the proposed FPGA-based approach in mitigating attacks targeting sensitive data and encryption keys in DRAM. It presents experimental results, a comparative analysis, and a discussion of the key findings.

5.1 Experimental Setup

The implementation and evaluation of the proposed memory encryption system for the RISC-V CPU Figure 4.1 was conducted using a Lenovo laptop with an Intel Core i5 7th Generation processor running at 2.5 GHz, 8 GB of RAM, and an NVIDIA GeForce 920MX GPU. The system operated on Ubuntu 18.04 LTS, providing a stable Linux-based development environment. Software tools utilized included Xilinx ISE 14.6 for legacy IP integration and Vivado Design Suite 2017.4 for design synthesis, implementation, and bitstream generation.

The hardware implementation was carried out on the Digilent Arty Evaluation Kit, equipped with a Xilinx Artix-7 XC7A35T FPGA. This FPGA features 33,280 logic cells distributed across 5,200 configurable logic slices, with each slice containing four 6-input LUTs and eight flip-flops. The device also integrates 1,800 Kbits of block RAM, supporting efficient on-chip data buffering. The FPGA fabric is capable of operating at internal clock frequencies exceeding 450 MHz, allowing high-speed operation of the designed cryptographic modules.

The evaluation board includes 256 MB of DDR3L SDRAM connected via a 16-bit data bus running at 667 MHz, and 16 MB of Quad-SPI flash memory used for configuration and persistent storage. This setup provided a robust and resource-rich environment for implementing and evaluating the proposed memory encryption mechanisms under realistic operating conditions on a physical platform.

The integration of a memory encryption module into the RISC-V CPU introduces both security benefits and hardware overhead. The primary trade-offs observed include increased area utilization, higher power consumption, and additional delay in the data path. These costs are balanced against the gain in data confidentiality and resistance to memory snooping attacks. From synthesis results, it was observed that the encryption module consumes a measurable portion of FPGA resources, including LUTs and flip-flops, and slightly lowers the maximum achievable clock frequency.

Despite these overheads, the design remains within acceptable operational limits for embedded and low-power systems. The added encryption ensures data protection without significantly compromising performance. This trade-off highlights a critical design consideration: while encryption enhances security, its implementation must be carefully optimized to minimize hardware penalties. In this case, the proposed approach achieves a favorable balance between security and efficiency, making it suitable for security-sensitive applications in resource-constrained environments.

5.2 Memory security Thread-off

Implementing memory encryption in RISC-V CPUs presents a significant security-cost trade-off, where the level of protection achieved must be carefully balanced against the performance, hardware resource (area), and power consumption overheads. This is particularly critical given RISC-V's diverse applications, from tiny embedded systems to high-performance computing, each with unique constraints.

Security Benefits: Memory encryption directly addresses critical vulnerabilities such as cold boot attacks, where data might be retrieved from DRAM after power-off, and direct memory access (DMA) attacks, where malicious peripherals can bypass CPU security mechanisms to read or modify memory. It also protects against bus snooping and certain software-based memory exploits by ensuring that data stored off-chip is unintelligible to unauthorized parties. A robust memory encryption scheme typically includes secure key management, covering generation, storage, and distribution, which are foundational to overall system security. The strength of the chosen encryption algorithm (e.g., AES, or lightweight ciphers like SIMON, PRINCE, QARMA, ChaCha, Trivium, Grain) and its key size directly determine the cryptographic strength.

Performance: Encryption and decryption introduce latency in memory access, as data must be processed before being written to or after being read from DRAM. This can impact CPU

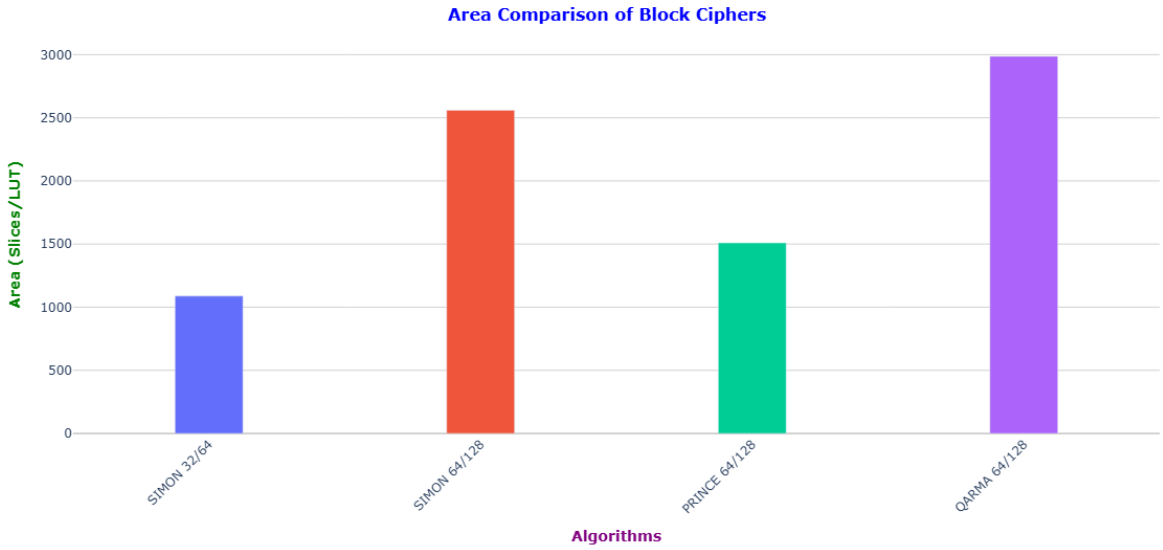


Figure 5.1: Block Cipher.

throughput, potentially creating a bottleneck if the encryption engine cannot keep pace with the CPU's memory demands.

Hardware Area: The encryption/decryption engine, along with associated control logic and key storage, consumes chip area (measured in Slices/LUTs). As shown in the "Area Comparison of Block Ciphers," AES 128/128, while strong, is significantly larger than lightweight ciphers like SIMON 32/64 or PRINCE 64/128. For RISC-V designs targeting low-cost, minimal-footprint embedded devices, selecting a lightweight cipher is crucial to minimize area overhead.

Power Consumption: The additional logic and data movement involved in encryption/decryption increase both dynamic power consumption (due to active operations) and static power (due to increased leakage from more transistors), which is a significant concern for battery-powered or energy-efficient RISC-V systems.

The core challenge for memory encryption in RISC-V is optimizing these trade-offs to deliver sufficient security for the target application without making the system prohibitively expensive, slow, or power-hungry. This often involves choosing ciphers with appropriate area-performance profiles and carefully integrating the encryption engine with the memory hierarchy (e.g., between the memory controller and DRAM)

5.3 Lightweight Block Ciphers

Figure 5.1 illustrates an area comparison of lightweight block ciphers, illustrating the hardware implementation area required for four distinct lightweight block cipher algorithms, SIMON 32/64, SIMON 64/128, PRINCE 64/128, and QARMA 64/128.

We examined the data; the SIMON 32/64 algorithm stands out as the most compact, requiring the least amount of hardware area among the four ciphers evaluated. This makes it a highly attractive option for deeply embedded systems or applications where silicon real estate is extremely limited. Following SIMON 32/64 in terms of efficiency is PRINCE 64/128, which also demonstrates a relatively low area consumption, suggesting it too is suitable for constrained environments, albeit slightly less efficient than SIMON 32/64.

SIMON 32/64 is the least resource-intensive, while PRINCE 64/128 follows with a moderate area requirement. SIMON 64/128 shows a significantly higher area consumption, and QARMA 64/128 demands the most resources, making it the largest in terms of hardware footprint. This comparison is crucial for designers selecting cryptographic algorithms, particularly in resource-constrained environments where hardware area directly impacts cost and efficiency.

In contrast, the SIMON 64/128 algorithm exhibits a significantly larger hardware footprint when compared to its 32/64 counterpart and PRINCE 64/128. This increase in area might be attributed to the larger key size (128 bits vs 64 bits) and/or block size (64 bits vs 32 bits) requiring more complex logic. The QARMA 64/128 algorithm is the most resource-intensive of the group, demanding the largest hardware area. While larger area often correlates with higher security or performance characteristics, this graph specifically highlights the direct hardware cost, which is a critical trade-off consideration for hardware designers when selecting a cipher for a given application's constraints.

In summary, the graph highlights the varying hardware costs associated with different block cipher implementations. SIMON 32/64 and PRINCE 64/128 are more compact, while SIMON 64/128 and especially QARMA 64/128 require substantially more logic resources. This comparison is crucial for designers selecting cryptographic algorithms, particularly in resource-constrained environments where hardware area directly impacts cost and efficiency.

5.4 Lightweight Stream Ciphers

Figure 5.2 compares the area usage of three stream cipher algorithms: ChaCha8, Trivium, and Grain128. ChaCha8 consumes the largest area, significantly outpacing the others. Trivium is the

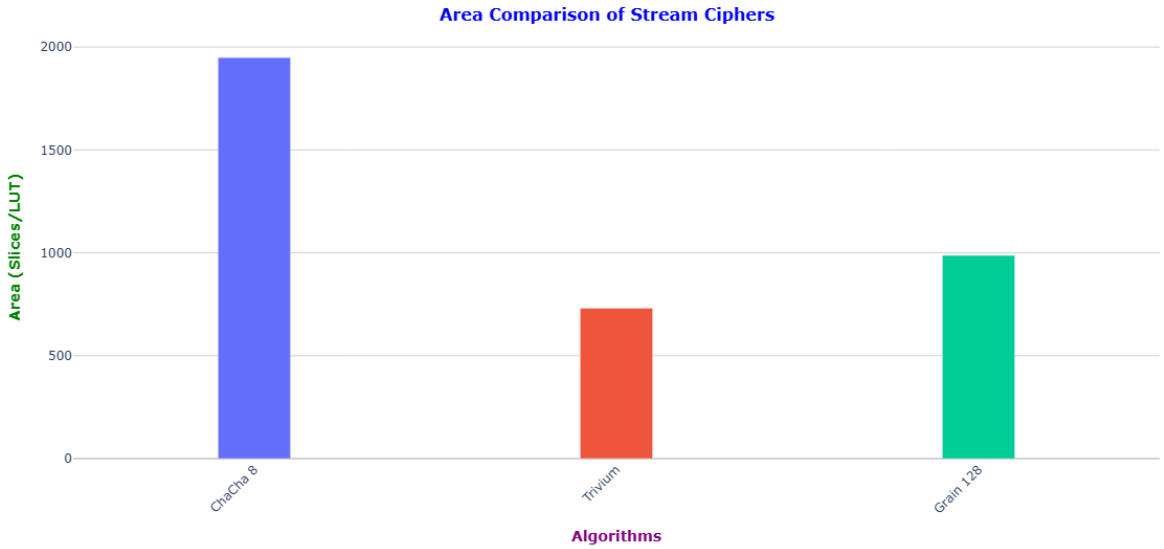


Figure 5.2: Stream Cipher.

most efficient, requiring the least area, while grain128 falls between the two. This visualization highlights the varying resource demands of these cryptographic algorithms.

Figure 5.3 observing the data, AES 128/128 demands significantly more hardware area than any of the other listed ciphers, with its bar towering above the rest, indicating it is the most resource-intensive. In contrast, algorithms like SIMON 32/64, PRINCE 64/128, ChaCha 8, Trivium, and Grain 128 all show substantially lower area footprints compared to AES 128/128, highlighting their ”lightweight” nature. Even SIMON 64/128 and QARMA 64/128, while larger than other lightweight options, consume considerably less area than the AES baseline.

This comparison emphasizes the design trade-offs between security levels (often associated with AES) and hardware resource consumption. For applications in resource-constrained environments, where area is a primary concern, the lightweight ciphers offer a viable alternative to the more demanding AES, even with AES serving as a standard for many security protocols.

The area comparison of various block ciphers shows that AES requires significantly more hardware resources than other algorithms, making it the most demanding in terms of implementation area. In contrast, lightweight ciphers such as SIMON, Trivium, and Grain are much more efficient, occupying only a small fraction of the area needed by AES. PRINCE, QARMA, and ChaCha strike a balance between performance and hardware cost, offering moderate area usage. This highlights the trade-offs between security strength, performance, and resource constraints, with lightweight ciphers being more suitable for constrained environments where minimizing hardware area is a key concern.

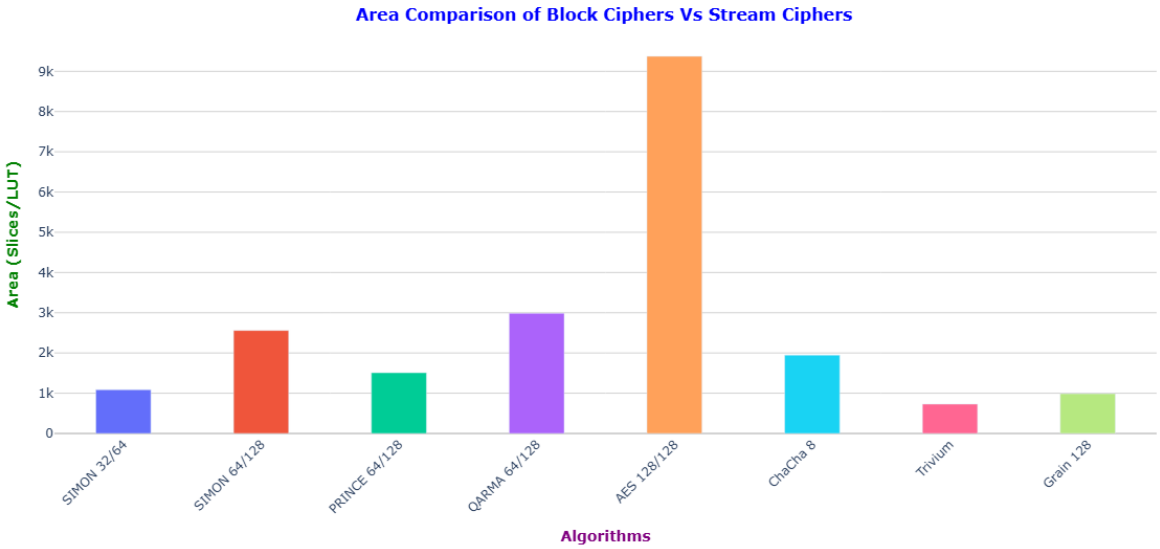


Figure 5.3: Block Vs Stream Cipher.

5.5 Power

Table 5.1 presents a comparison of power consumption and delay among various block and stream ciphers, using AES-128 as the baseline. The data highlights the relative efficiency of lightweight ciphers in terms of both power and latency, offering insight into their suitability for energy-constrained and real-time applications.

AES-128 is widely regarded as a standard benchmark in cryptographic performance due to its strong security guarantees and broad adoption. Although it does not have specific power consumption data listed here, its delay of 29.645 nanoseconds establishes a performance baseline against which other lightweight ciphers can be compared. This latency reflects the computational complexity of AES-128's robust encryption algorithm, which is designed for high security but often at the cost of increased processing time and energy consumption.

When compared to AES-128, the other ciphers in the table demonstrate varying trade-offs between power efficiency and speed. For instance, lightweight ciphers like Grain-128 and Trivium exhibit significantly lower delays (8.607 ns and 9.182 ns respectively) and much lower power consumption, highlighting their suitability for resource-constrained environments. Meanwhile, ciphers such as SIMON and PRINCE offer moderate delays and power usage, balancing security with efficiency. Overall, AES-128 serves as a critical reference point, illustrating the performance and energy costs associated with a traditional, well-established encryption standard.

Table 5.1: Power and Delay Comparison of Block and Stream Ciphers

Cipher	Power (W)	Delay (ns)
SIMON 32/64	0.413	21.258
SIMON 64/128	0.459	24.546
PRINCE 64/128	0.144	16.961
QARMA 64/128	0.215	18.900
ChaCha-8	0.337	15.015
Trivium	0.122	9.182
Grain-128	0.089	8.607
AES-128	–	29.645

5.6 Storage and Execution Overhead

Execution overhead is quantified by the total cycles required to perform encryption and authentication operations, impacting system performance. Storage overhead accounts for the additional memory allocated to metadata in both internal cache and external RAM, which affects overall memory utilization. Minimizing these overheads is crucial to achieving efficient and secure memory encryption without significantly degrading system responsiveness or increasing resource consumption.

Table 5.2 compares block and stream lightweight ciphers in terms of storage and execution overhead. SIMON exhibits the highest overhead, suggesting a greater resource demand. PRINCE, QARMA, and ChaCha provide a balance between performance and resource usage, with moderate overheads. Trivium and Grain are the most efficient, offering minimal impact on storage and execution, making them ideal for low-resource environments. Overall, the data highlights trade-offs between security, performance, and system efficiency across different cipher designs.

Table 5.2: Storage and Execution Overhead for Lightweight Ciphers

Cipher	Storage Overhead (%)	Execution Overhead (%)
SIMON 64/128	4.0%	6%
PRINCE 64/128	2.5%	4.50%
QARMA 64/128	3.0%	5.50%
ChaCha 8	2.5%	4%
Trivium	1.5%	3.50%
Grain 128	1.5%	3%

Table 5.3: Comparison of Memory Encryption Techniques

Characteristic	Our Work	SMARTS[11]	AEGIS [51]	Split Counter [52]	SecureMe [53]	AISE [54]
Execution Overhead	3–6%	3.0%	4.5%	2.0%	5.2%	1.6%
Storage Overhead	1.5–4%	0.43%	6%	1.5%	1.6%	1.6%
Maturity	FPGA	Simulation	P-FPGA	Simulation	Simulation	Simulation
Confidentiality / Integrity	C + I	C + I	C + I	C + I	C + I	C + I
Encryption Algorithm	SIMON, PRINCE, QARMA, ChaCha, Trivium, Grain128	AES-GCM	AES	AES-GCM	AES	AES
Full/Partial Mem Encryption	FME	PME	FME	FME	FME	FME

5.7 Comparison with Previous Works

Table 5.3 provides a comparative overview of the proposed Memory Encryption Unit (MEU) framework alongside several existing technologies. The analysis focuses on the overheads introduced by cryptographic operations, measured in execution time and memory consumption. It also summarizes general security features such as protection coverage, operational modes, cipher types, and coverage regions. The term maturity reflects the extent to which each framework has been evaluated and tested.

This work presents a memory encryption framework for RISC-V-based CPUs, emphasizing low-overhead and hardware-friendly cryptographic integration. Implemented and validated on a Xilinx Artix-7 FPGA, the design leverages lightweight block and stream ciphers including *SIMON*, *PRINCE*, *QARMA*, *ChaCha*, *Trivium*, and *Grain-128* to enable full memory encryption (FME) with execution overhead ranging from 3% to 6% and storage overhead between 1.5% and 4%.

Unlike prior approaches such as *SMARTS*, *AEGIS*, *Split Counter*, *SecureMe*, and *AISE*—which mostly rely on AES or AES-GCM and are limited to simulation or partial FPGA prototyping—our system demonstrates practical feasibility and flexibility. SMARTS offers low storage overhead (0.43%) but only supports partial memory encryption (PME) and lacks FPGA implementation. AEGIS and SecureMe incur higher overheads while relying on more resource-intensive AES variants. In contrast, our solution achieves confidentiality and integrity with a broader cipher selection and moderate hardware cost, making it highly suitable for area- and power-constrained RISC-V embedded systems.

5.8 Security Implications

Incorporating hardware-based cryptographic extensions significantly strengthened the security of the memory system. By delegating essential cryptographic tasks to dedicated hardware components, the design effectively reduced vulnerabilities and threats targeting DRAM. This

hardware-focused strategy improves both data integrity and confidentiality, while also creating a safer execution environment for applications with stringent security requirements. The move toward hardware-driven security solutions reflects the increasing demand for resilient and robust systems in today's highly connected landscape.

The three research questions posed in Section 1.2 are addressed as follows:

1. **RQ1:** Which implementation and evaluation strategies can enable effective memory encryption for RISC-V, ensuring confidentiality while preserving performance and compatibility?

   Lightweight memory encryption for the RISC-V architecture can be effectively realized by embedding compact cryptographic cores directly within the memory access path of the processor, specifically at the memory controller level. In this work, a hardware-based encryption engine supporting a range of lightweight ciphers (SIMON, PRINCE, QARMA, ChaCha, Trivium, and Grain 128) was designed and integrated with the memory interface of a RISC-V CPU. The encryption engine was implemented on the Digilent Arty A7 FPGA featuring a Xilinx Artix-7 XC7A35T device.

   To maintain performance, stream ciphers with simple datapaths and low-latency operation were prioritized for high-throughput scenarios, while block ciphers with small round footprints were selected to minimize area overhead. Evaluations showed that even the most complex ciphers consumed a moderate portion of the available logic resources and incurred only negligible latency overhead. Furthermore, all encryption modules operated within a constrained internal memory budget and remained compatible with existing RISC-V memory operations without requiring ISA modifications. This demonstrates that secure and efficient memory encryption can be achieved on RISC-V platforms without compromising performance or hardware constraints.

2. **RQ2:** What methods can securely and efficiently integrate key management into the RISC-V execution environment without introducing performance bottlenecks?

   Secure and efficient key management in the RISC-V execution environment can be achieved by combining hardware-based key generation using a random number generator (RNG). In this design, a lightweight cryptographically secure pseudo-random number generator (PRNG) is embedded in hardware to autonomously generate encryption keys during the system's initialization or secure boot phase. The generated keys are stored in protected on-chip memory or dedicated key registers and are accessible only by the memory encryption engine.

3. **RQ3:** Which lightweight encryption algorithms offer the best trade-off between hardware efficiency and security for memory protection on constrained devices?

Among the evaluated ciphers, stream ciphers like Trivium and Grain 128 demonstrated superior efficiency in terms of area and storage usage, making them ideal for highly constrained environments. However, block ciphers like SIMON 64/128 and QARMA 64/128 offer stronger structural resistance to certain attacks and remain competitive in area efficiency. The selection of an appropriate cipher thus depends on the specific security and performance requirements of the system, with Trivium favored for ultra-lightweight cases and SIMON or QARMA for balanced security-performance scenarios.

Overall, memory encryption can be implemented on RISC-V platforms with minimal performance penalties, especially when lightweight ciphers and pipelined hardware architectures are used. The results demonstrate a practical trade-off between security and performance, enabling confidentiality without compromising the responsiveness or efficiency of typical RISC-V applications.

Chapter 6

Conclusion and Future Work

This paper demonstrates the feasibility of integrating a memory encryption unit into the memory controller of a RISC-V CPU architecture, utilizing lightweight block and stream ciphers to achieve promising results compared to state-of-the-art solutions. The implementation was carried out on an Artix-7 evaluation board, and the design was evaluated in terms of power, area, and storage overheads. The experimental results show that stream ciphers offer significant area and storage efficiency compared to block ciphers. Based on these findings, we conclude that stream ciphers are well-suited for applications where minimizing area is a higher priority than achieving maximum throughput. Conversely, block ciphers are more appropriate for applications that demand high throughput and can tolerate increased area usage.

While this work has been implemented and evaluated exclusively on a RISC-V CPU architecture, future implementations may explore adaptation to other instruction set architectures such as x86, including AMD platforms, to evaluate compatibility, performance, and broader applicability across diverse computing systems.

It is important to investigate and incorporate additional countermeasures to improve the resistance of the encryption hardware against side-channel attacks. These attacks, which include power analysis, electromagnetic leakage, and timing-based techniques, can exploit physical characteristics of the system to extract sensitive information such as cryptographic keys. Since the current implementation assumes a secure internal execution environment, it does not address these vulnerabilities. Techniques such as power balancing, clock jittering, noise injection, and algorithmic masking can be explored to mitigate these risks. Incorporating such protections will significantly enhance the robustness of the system, especially in scenarios where physical access to the hardware cannot be fully restricted.

References

[1] Shay Gueron. “Memory encryption for general-purpose processors”. In: *IEEE Security & Privacy* 14.6 (2016), pp. 54–62.

[2] Andrew Waterman et al. *The RISC-V Instruction Set ManualVolume I: Unprivileged ISA, Version 20191213*. Tech. rep. RISC-V, 12/2019.

[3] Andrew Waterman et al. *The RISC-V Instruction Set ManualVolume II: Privileged Architecture,Version 20190608-Priv-MSU-Ratified*. Tech. rep. RISC-V, 06/2019.

[4] Salessawi Ferede Yitbarek et al. “Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 313–324. DOI: 10.1109/HPCA.2017.10.

[5] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: *Black Hat* 15.71 (2015), p. 2.

[6] Moritz Lipp et al. “Meltdown”. In: *arXiv preprint arXiv:1801.01207* (2018).

[7] Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*. 2018. arXiv: 1801.01203 [cs.CR]. <https://arxiv.org/abs/1801.01203>.

[8] Alessandro Cilardo. “Memory Encryption Support for an FPGA-based RISC-V Implementation”. In: *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2021, pp. 1–5.

[9] Tianyu Yin, Guozhu Xin, and Jun Han. “Hpme: A high-performance hardware memory encryption engine based on risc-v tee”. In: *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*. IEEE. 2020, pp. 1–3.

[10] Liu Lu. “Design of a programmable pci-e encryption system based on risc-v”. In: *2022 6th International Symposium on Computer Science and Intelligent Control (ISCSIC)*. IEEE. 2022, pp. 368–373.

[11] Ming Ming Wong, Jawad Haj-Yahya, and Anupam Chattopadhyay. “Smarts: Secure memory assurance of risc-v trusted soc”. In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2018, pp. 1–8.

[12] Christof Paar, Jan Pelzl, and Tim Güneysu. *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. Springer, 2024.

[13] Ege Gulcan, Aydin Aysu, and Patrick Schaumont. “A flexible and compact hardware architecture for the SIMON block cipher”. In: *International workshop on lightweight cryptography for security and Privacy*. Springer. 2014, pp. 34–50.

[14] Sa’ed Abed et al. “FPGA modeling and optimization of a SIMON lightweight block cipher”. In: *Sensors* 19.4 (2019), p. 913.

[15] Julia Borghoff et al. “PRINCE–A low-latency block cipher for pervasive computing applications”. In: *Advances in Cryptology–ASIACRYPT 2012*. Springer, 2012, pp. 208–225.

[16] Nicolai Müller, Thorben Moos, and Amir Moradi. “Low-latency hardware masking of PRINCE”. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2021, pp. 148–167.

[17] Roberto Avanzi. “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”. In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 4–44.

[18] Rui Zong and Xiaoyang Dong. “Milp-aided related-tweak/key impossible differential attack and its applications to qarma, joltik-bc”. In: *IEEE Access* 7 (2019), pp. 153683–153693.

[19] Arthur Beckers, Benedikt Gierlichs, and Ingrid Verbauwhede. “Fault analysis of the chacha and salsa families of stream ciphers”. In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2017, pp. 196–212.

[20] Daniel J Bernstein et al. “ChaCha, a variant of Salsa20”. In: *Workshop record of SASC*. Vol. 8. 1. Citeseer. 2008, pp. 3–5.

[21] Christophe De Canniere, Özgül Küçük, and Bart Preneel. “Analysis of Grain’s initialization algorithm”. In: *International Conference on Cryptology in Africa*. Springer. 2008, pp. 276–289.

[22] Martin Hell et al. “A stream cipher proposal: Grain-128”. In: *2006 IEEE International Symposium on Information Theory*. IEEE. 2006, pp. 1614–1618.

[23] Christophe Cannière and Bart Preneel. “Trivium”. In: *New Stream Cipher Designs: The ESTREAM Finalists*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 244–266. <https://doi.org/10.1007/978-3-540-68351-3_18>.

[24] Yun Tian, Gongliang Chen, and Jianhua Li. *On the Design of Trivium*. Cryptology ePrint Archive, Paper 2009/431. 2009. <https://eprint.iacr.org/2009/431>.

[25] Maxime Montoya et al. “Energy-efficient Masking of the Trivium Stream Cipher”. In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2018, pp. 393–396. DOI: 10.1109/ICECS.2018.8617892.

[26] F. E. Potestad-Ordóñez, C. J. Jiménez-Fernández, and M. Valencia-Barrero. “Vulnerability Analysis of Trivium FPGA Implementations”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.12 (2017), pp. 3380–3389. DOI: 10.1109/TVLSI.2017.2751151.

[27] Tesfamichael Gebregziabher Gebrehiwot, Fitsum Assamnew Andargie, and Mohammed Ismail. “DEACT: Hardware Solution to Rowhammer Attacks”. In: *Journal of Computer Science* 19.7 (06/2023), pp. 861–876. DOI: 10.3844/jcssp.2023.861.876. <https://thescipub.com/abstract/jcssp.2023.861.876>.

[28] Moritz Nöltner-Augustin. *RISC-V — Architecture and Interfaces: The RocketChip*. Technical Report. Available at University of Heidelberg. University of Heidelberg, ZITI, 2017.

[29] Krste Asanovic et al. “The rocket chip generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[30] Viktor V Prutyanov et al. “Impact of Memory Hierarchy on Memory Encryption Performance”. In: *IEEE Access* (2024).

[31] Tao Lu. “A survey on risc-v security: Hardware and architecture”. In: *arXiv preprint arXiv:2107.04175* (2021).

[32] Orr DUNKELMAN. “Block Ciphers”. In: *Symmetric Cryptography, Volume 1: Design and Security Proofs* (2023), p. 39.

[33] Richard E Blahut. *Cryptography and secure communication*. Cambridge University Press, 2014.

[34] Yasir Amer Abbas et al. “Implementation of PRINCE algorithm in FPGA”. In: *Proceedings of the 6th International Conference on Information Technology and Multimedia*. IEEE. 2014, pp. 1–4.

[35] Jérémy Jean et al. “Security analysis of PRINCE”. In: *International Workshop on Fast Software Encryption*. Springer. 2013, pp. 92–111.

[36] Jos Wetzels and Wouter Bokslag. “Simple SIMON: FPGA implementations of the SIMON 64/128 Block Cipher”. In: *arXiv preprint arXiv:1507.06368* (2015).

[37] Johannes Pfau et al. “A hardware perspective on the ChaCha ciphers: Scalable Chacha8/12/20 implementations ranging from 476 slices to bitrates of 175 Gbit/s”. In: *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. IEEE. 2019, pp. 294–299.

[38] Nuray At et al. “Compact hardware implementations of chacha, blake, threefish, and skein on fpga”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.2 (2013), pp. 485–498.

[39] Shohreh Sharif Mansouri and Elena Dubrova. “An Improved Hardware Implementation of the Grain Stream Cipher”. In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, pp. 433–440. DOI: 10.1109/DSD.2010.49.

[40] Yosuke Todo et al. “Fast correlation attack revisited: cryptanalysis on full grain-128a, grain-128, and grain-v1”. In: *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*. Springer. 2018, pp. 129–159.

[41] Weijie Liu and Juntao Gao. “Quantum security of Grain-128/Grain-128a stream cipher against HHL algorithm”. In: *Quantum Information Processing* 20 (2021), pp. 1–22.

[42] Jonathan Sönnerup et al. “Efficient hardware implementations of grain-128aead”. In: *International Conference on Cryptology in India*. Springer. 2019, pp. 495–513.

[43] Alexander Maximov and Martin Hell. “Software evaluation of grain-128aead for embedded platforms”. In: *Cryptology ePrint Archive* (2020).

[44] Enfang Cui, Tianzheng Li, and Qian Wei. “Risc-v instruction set architecture extensions: A survey”. In: *IEEE Access* 11 (2023), pp. 24696–24711.

[45] Süleyman Savas, Endri Bezati, and Jörn W Janneck. “Generating hardware and software for RISC-V cores generated with Rocket Chip generator”. In: *2021 IEEE 34th International System-on-Chip Conference (SOCC)*. IEEE. 2021, pp. 89–94.

[46] MV Arunkumar, S Ganesh Bhairathi, and Harshal G Hayatnagarkar. “Perc: Posit enhanced rocket chip”. In: *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. 2020.

[47] Saeid Mofrad et al. “A comparison study of intel SGX and AMD memory encryption technology”. In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2018, pp. 1–8.

[48] Reouven Elbaz et al. “Hardware mechanisms for memory authentication: A survey of existing techniques and engines”. In: *Transactions on Computational Science IV: Special Issue on Security in Computing* (2009), pp. 1–22.

[49] Peter Pessl et al. “{DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks”. In: *25th USENIX security symposium (USENIX security 16)*. 2016, pp. 565–581.

[50] Jingyao Zhang, Hoda Naghibijouybari, and Elaheh Sadredini. “Sealer: In-sram aes for high-performance and low-overhead memory encryption”. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 2022, pp. 1–6.

[51] G Edward Suh et al. “AEGIS: Architecture for tamper-evident and tamper-resistant processing”. In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. 2003, pp. 357–368.

[52] Chenyu Yan et al. “Improving cost, performance, and security of memory encryption and authentication”. In: *ACM SIGARCH Computer Architecture News* 34.2 (2006), pp. 179–190.

[53] Siddhartha Chhabra et al. “SecureME: a hardware-software approach to full system security”. In: ICS ’11. Tucson, Arizona, USA: Association for Computing Machinery, 2011, pp. 108–119. DOI: 10.1145/1995896.1995914. <https://doi.org/10.1145/1995896.1995914>.

[54] Brian Rogers et al. “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 183–196. DOI: 10.1109/MICRO.2007.16.