박사학위논문

Structural Optimization of Cryptographic Algorithms Based on Embedded Processor Characteristics

2025년

한 성 대 학 교 대 학 원

정보컴퓨터공학과

정보시스템공학전공

권 혁 동

박사학위논문
지도교수 서화정

Structural Optimization of Cryptographic Algorithms Based on Embedded Processor Characteristics

임베디드 프로세서 특성 기반 암호 알고리즘의
구조적 최적화

2024년 12월 일

한성대학교 대학원

정보컴퓨터공학과

정보시스템공학전공

권 혁 동

박 사 학 위 논 문
지도교수 서화정

Structural Optimization of Cryptographic Algorithms Based on Embedded Processor Characteristics

임베디드 프로세서 특성 기반 암호 알고리즘의 구조적 최적화

위 논문을 공학 박사학위 논문으로 제출함

2024년 12월 일

한 성 대 학 교 대 학 원

정 보 컴 퓨 터 공 학 과

정보시스템공학전공

권 혁 동

권혁동의 공학 박사학위 논문을 인준함

2024년 12월 일

심사위원장 박 명 서 (인)

심사위원 김 수 리 (인)

심사위원 이 웅 희 (인)

심사위원 유 지 현 (인)

심사위원 서 화 정 (인)

ABSTRACT

Structural Optimization of Cryptographic Algorithms Based on Embedded Processor Characteristics

Kwon, HyeokDong

Major in Information System Engineering

Dept. of Information and Computer Engineering

The Graduate School

Hansung University

This dissertation investigates and presents the results of an optimized implementation technique achieved through modifications to the internal structure of cryptographic algorithms. Among the various aspects of optimal implementation, speed optimization is crucial in improving inefficient computational performance by accelerating the algorithm's processing speed. Parallel implementation is commonly employed for optimization; however, inherent limitations exist when relying solely on parallelizing internal operations. A method for enhancing the performance of cryptographic algorithms is proposed through modifications to their internal structure. These structural

modifications may involve precomputing specific values, utilizing precomputation tables for large-scale calculations, or leveraging processor features to reverse original operations.

The cryptographic algorithms targeted for implementation in this research include the domestic lightweight block cipher CHAM, the lightweight block cipher candidate TinyJAMBU, and the post-quantum cryptography candidate Rainbow. The implementation platforms selected for this study are the 8-bit AVR processor, commonly used in low-end Internet of Things (IoT) environments, and the 64-bit ARM processor, which, though relatively high-end compared to AVR, has recently expanded its application from smartphones to laptops. The proposed technique involves redesigning the internal structure of each algorithm, considering the unique characteristics of the algorithms and the processor environments, to enhance overall algorithm performance.

【Keywords】 Block cipher, Post Quantum Cryptography, Optimized implementation, IoT Processor

Table of Contents

1. Introduction ····· 1

1.1 Main Contribution ····· 1

2. Preliminaries ····· 3

2.1 Symmetric-Key Cryptography ····· 3

2.2 Public-Key Cryptography ····· 5

2.3 Post-Quantum Cryptography ····· 6

2.4 Target Cryptographic Algorithms ····· 8

2.4.1 Block Cipher CHAM ····· 8

2.4.2 Lightweight Cipher TinyJAMBU ····· 9

2.4.3 Post-Quantum Cryptography Rainbow ····· 10

2.5 Target Processors ····· 12

2.5.1 8-bit AVR Microcontroller ····· 12

2.5.2 64-bit ARM Processor ····· 13

2.6 Previous Works ····· 13

3. Optimized Implementation of Target Cipher ····· 16

3.1 CHAM with Precomputation ····· 16

3.1.1 Skip Rounds by Precomputation ····· 16

3.1.2 Logical Block Rotation ····· 18

3.1.3 Register Scheduling and Instructions Used ····· 20

3.1.4 Alternative Implementation: Furious CHAM ····· 27

3.2 TinyJAMBU with Reverse Bitwise Shift ····· 29

3.2.1 Reverse Bitwise Shift(RBS) ····· 29

3.2.2 Register Scheduling and Instructions Used ····· 36

3.2.3 Alternative Implementation: Initialization Skip ····· 42

3.3 Rainbow with Look-Up Table Based Multiplication ····· 43
3.3.1 Tower-Field Based Multiplication ····· 43
3.3.2 Look-Up Table Based Multiplication in Rainbow I ····· 46
3.3.3 Resolve of LUT Size Problem in Rainbow III and V ····· 50
3.3.4 Register Scheduling and Instructions Used ····· 50
3.3.5 Alternative Implementation: Avoiding Cache Side Attack 54

4. Performance Evaluation ····· 57
4.1 Evaluation CHAM Block Cipher ····· 57
4.2 Evaluation TinyJAMBU Lightweight Cipher ····· 59
4.3 Evaluation Rainbow Post-Quantum Cryptography ····· 63

5. Conclusion ····· 70

Appendix: Look-Up Table for Rainbow ····· 72

Appendix: Performance evaluation result for Rainbow on A13 processors ····· 80

Appendix: Performance evaluation result for Cache side attack resistance implementation of Rainbow III and V ····· 87

Abbreviation: Abbreviation ····· 90

Bibliography ····· 91

국문초록 ····· 95

List of Tables

[Table 2-1] List of CHAM parameters (n: block size, k: key size, ω: word size, r: number of rounds) ····· 9
[Table 2-2] Key and signature size of Rainbow signature. (Key size unit: KB, Signature size unit: bit) ····· 11
[Table 3-1] List of instructions used in implementation for CHAM in alphabetical order ····· 22
[Table 3-2] Implementation code of proposed CHAM (RC: round counter, RK: round key, X00~X31: plaintext, XT: temporary register, Zero: zero register) ····· 22
[Table 3-3] Number of keyed permutations each step in tinyJAMBU ····· 30
[Table 3-4] Pseudocode for keyed permutation(<<n: bitwise left shift n times, >>n: bitwise right shift n times, |: bitwise OR, &: bitwise AND, ^: bitwise XOR, ~: bitwise NOT) ····· 30
[Table 3-5] Number of shifts for each implementation ····· 35
[Table 3-6] List of instructions used in implementation for TinyJAMBU in alphabetical order ····· 37
[Table 3-7] Implementation code of proposed TinyJAMBU (RC: round counter, RK: round key, X00~X31: plaintext, XT: temporary register, Zero: zero register) ····· 38
[Table 3-8] Pseudocode of tower-field based polynomial multiplication for Rainbow signature (^: bitwise XOR) ····· 43
[Table 3-9] Pseudocode of look-up table based polynomial multiplication for Rainbow I (<<n: bitwise left shift n times, >>n: bitwise right shift n times, &: bitwise AND, ^: bitwise XOR) ····· 46
[Table 3-10] Pseudocode of look-up table based polynomial multiplication for Rainbow III and V(<<n: bitwise left shift n times, >>n: bitwise right shift n times, &: bitwise AND, ^: bitwise XOR,

A: additional) ····· 48
[Table 3-11] List of instructions used to implement Rainbow signatures in alphabetical order ····· 52
[Table 3-12] Implementation code of proposed multiplication (x0: output address, x1: operand address, x2(w2): constant) ····· 53
[Table 3-13] Implementation code of cache side attack resistance implementation. (x2(w2): constant) ····· 54
[Table Appendix-1] Precomputation look-up table of tower-field based polynomial multiplication results on GF16 expressed in hexadecimal (A: additional table for Rainbow III and V) ····· 72
[Table Appendix-2] Implementation code of proposed multiplication for Rainbow III and V (x0: output address, x1: operand address, x2(w2): constant) ····· 73
[Table Appendix-3] Implementation code of constant-time implementation. (x2(w2): constant) ····· 81

List of Figures

[Figure 2-1] Symmetric-key cryptography structure ····· 3
[Figure 2-2] Block cipher framework (Encryption only) ····· 4
[Figure 2-3] Stream cipher framework (Encryption only) ····· 5
[Figure 2-4] Public-key cryptography architecture ····· 5
[Figure 2-5] Round function structure of CHAM ····· 9
[Figure 2-6] NLFSR for Keyed permutation of TinyJAMBU ····· 10
[Figure 2-7] Whole structure of TinyJAMBU ····· 10
[Figure 2-8] Structure of AVR registers ····· 12
[Figure 2-9] Controlling vector registers via arrangement specifiers ····· 13
[Figure 3-1] Flow of counter values in CHAM CTR mode of operation ····· 17
[Figure 3-2] Optimized CHAM structure with CTR mode of operation ····· 19
[Figure 3-3] Optimized 32-bit counter CHAM-64/128 structure 20
[Figure 3-4] Register allocation plan for proposed CHAM ····· 20
[Figure 3-5] Two implementation scenarios for the variable key model ····· 26
[Figure 3-6] Register allocation plan for furious CHAM ····· 28
[Figure 3-7] S2 state computation structure using AVR assembly instructions ····· 31
[Figure 3-8] Proposed RBS applied to s2 state calculation ····· 32
[Figure 3-9] S3 state flow with AVR assembly implementation ·· 33
[Figure 3-10] S3 state calculation with RBS technique applied ···· 33
[Figure 3-11] Operation process with the previous s1 state block operation and RBS applied ····· 34
[Figure 3-12] Operate s2 state block with single shift ····· 35
[Figure 3-13] Register allocation plan for RBS TinyJAMBU ····· 36

[Figure 3-14] Table loading process in proposed Rainbow signature ···· 46
[Figure 3-15] Register allocation plan for Look-up table based Rainbow signature ····· 52
[Figure 4-1] Performance Measurement Results for CHAM (Unit: clock cycles per byte, 32-bit: 32-bit counter of CHAM-64/128) ··· 57
[Figure 4-2] Performance Measurement Results for Furious CHAM-64/128 (Unit: clock cycles per byte) ····· 59
[Figure 4-3] Performance Measurement Results for Keyed permutation of TinyJAMBU (Unit: clock cycles) ····· 60
[Figure 4-4] Performance Measurement Results for TinyJAMBU (Unit: clock cycles, I: Initialization skip implementation) ····· 62
[Figure 4-5] Performance Measurement Results for table based multiplier of proposed Rainbow signature in log scale (Unit: clock cycles) ····· 64
[Figure 4-6] Performance Measurement Results for Rainbow I on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 65
[Figure 4-7] Performance Measurement Results for Rainbow III on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 66
[Figure 4-8] Performance Measurement Results for Rainbow V on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 67
[Figure 4-9] Performance Measurement Results for Rainbow I cache side attack resistance implementation and constant-time implementation on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 68
[Figure Appendix-1] Performance Measurement Results for Rainbow I on Apple A13 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 80
[Figure Appendix-2] Performance Measurement Results for Rainbow III on Apple A13 processors expressed in log scale (Unit: 10^6 clock

cycles) ····· 81
[Figure Appendix-3] Performance Measurement Results for Rainbow V on Apple A13 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 82
[Figure Appendix-4] Performance Measurement Results for Rainbow I on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 83
[Figure Appendix-5] Performance Measurement Results for Rainbow III on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 84
[Figure Appendix-6] Performance Measurement Results for Rainbow V on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles) ····· 85
[Figure Appendix-7] Performance Measurement Results for Rainbow III cache side attack resistance implementation on Apple M1 processors (Unit: 10^6 clock cycles) ····· 87
[Figure Appendix-8] Performance Measurement Results for Rainbow V cache side attack resistance implementation on Apple M1 processors (Unit: 10^6 clock cycles) ····· 88

1. Introduction

1.1 Main Contribution

Cryptographic algorithms, which provide security based on complex mathematical principles, typically require significant computational resources. With recent advancements in hardware, performing cryptographic operations is no longer a major challenge. However, for small electronic devices such as sensor nodes, available resources are limited, and executing cryptographic algorithms can consume considerable time. This dissertation presents optimal implementation methods to efficiently execute cryptographic algorithms. Although there are various perspectives on optimal implementation, the focus is primarily on speed optimization, which aims to enhance processing speed. The main contributions of this dissertation are as follows.

1. Proposal of optimized implementation through modifications to the internal operational structure of the algorithm. Parallel implementation is one of the most powerful methods used in optimization; however, it is limited to environments that support parallel instructions. Moreover, parallel implementation is applicable only to algorithms that benefit from parallel operations. Therefore, various methods for modifying the internal structure of algorithms are proposed. By utilizing the unique characteristics of each algorithm, the computational process is redesigned, and a more advanced design is proposed by taking into account the characteristics of the processor.

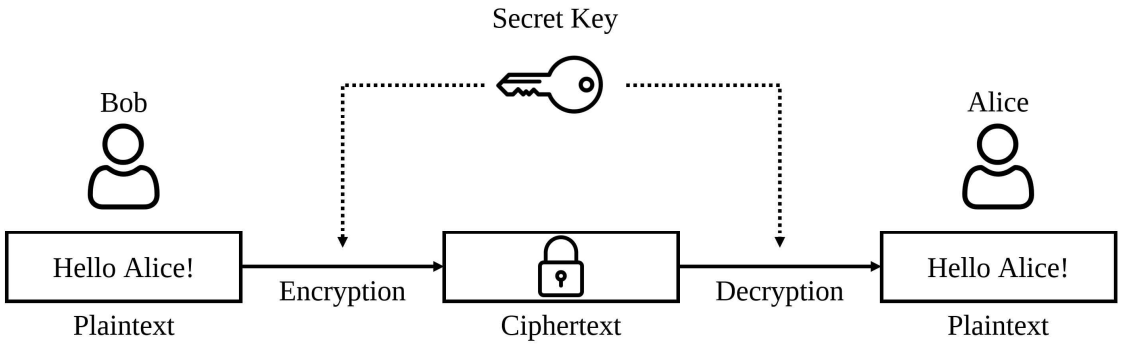
2. Presentation of additional implementations for special purposes. The proposed optimized implementations are designed based on general use cases. In addition, specialized implementations tailored to specific scenarios are presented. While these additional implementations may require certain assumptions not present in the general implementations, they offer more optimized performance or exhibit resistance to certain attacks, thereby possessing distinctive features compared to the general optimal implementations.

The remainder of this dissertation is organized as follows. Chapter 2 introduces various cryptographic algorithm structures and examines the algorithms selected for optimal implementation, along with an introduction to the target processors for optimization. Additionally, prior research on optimal implementations is reviewed. Chapter 3 discusses the redesign for optimal implementation, the methods for algorithm implementation, and additional implementations. Chapter 4 evaluates the performance of the proposed implementations. Chapter 5 concludes the dissertation.

2. Preliminaries

2.1 Symmetric-Key Cryptography

A symmetric-key encryption system is an algorithm in which encryption and decryption are performed using a single shared secret key, also referred to as a secret-key encryption algorithm. The basic structure of symmetric-key cryptography is shown in [Figure 2-1].

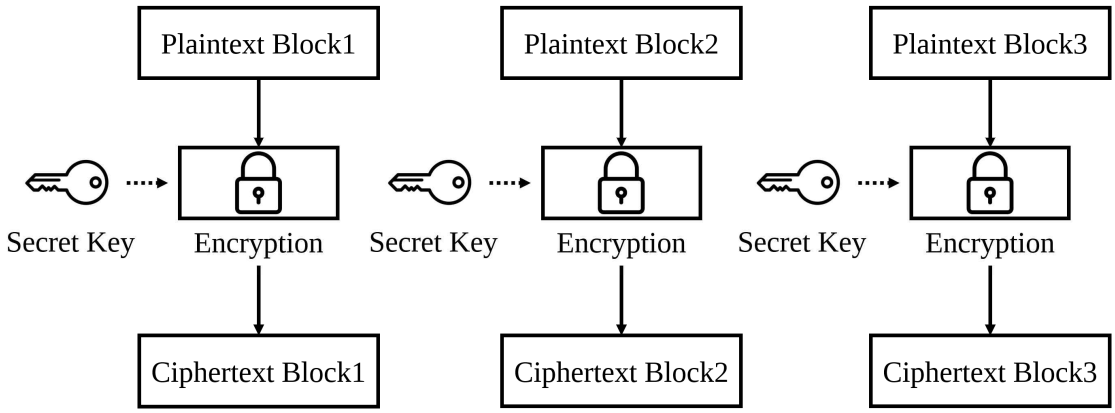


[Figure 2-1] Symmetric-key cryptography structure

It is defined by the use of the same key for both encryption and decryption. However, symmetric-key cryptography has the drawback of difficulty in securely sharing the secret key. Since the secret key is used for both encryption and decryption, if the key is compromised, the encrypted confidential information can be immediately restored. Therefore, securely sharing the secret key is critical, and this remained a significant challenge until the development of public-key cryptography. Algorithms that belong to symmetric key encryption include DES, AES, LEA, ARIA, and CHAM.

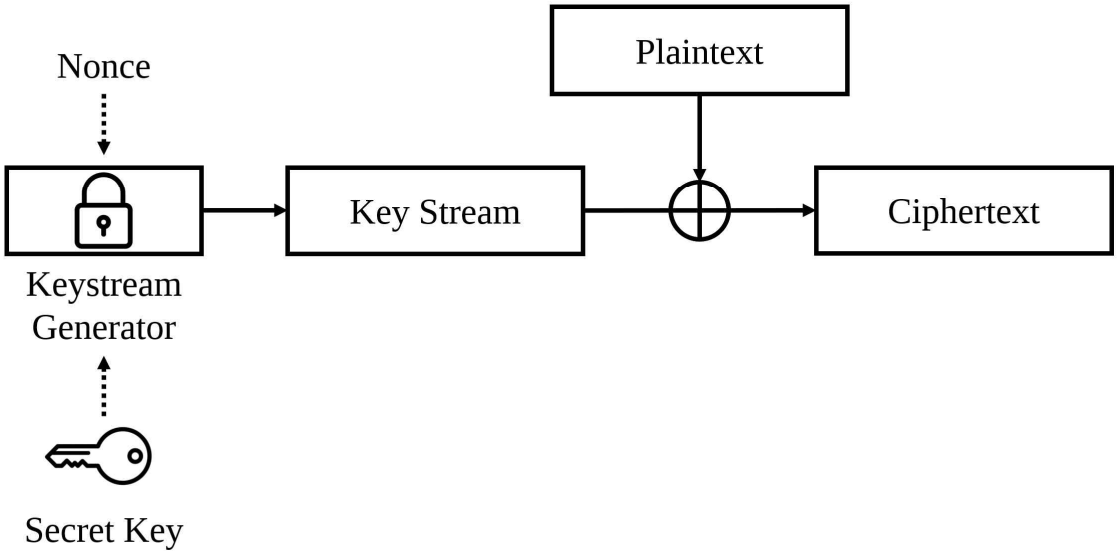
Symmetric-key cryptography is divided into block ciphers and stream ciphers.

- A block cipher encrypts data in fixed-size blocks, and its structure is illustrated in [Figure 2-2]. Since input messages may exceed the block size, block cipher operation modes are provided to encrypt messages larger than a single block. While block ciphers offer high diffusion and versatility, they tend to have slower encryption speeds and the propagation of errors in case of transmission issues. Common operation modes include ECB, CBC, CFB, OFB, and CTR.



[Figure 2-2] Block cipher framework (Encryption only)

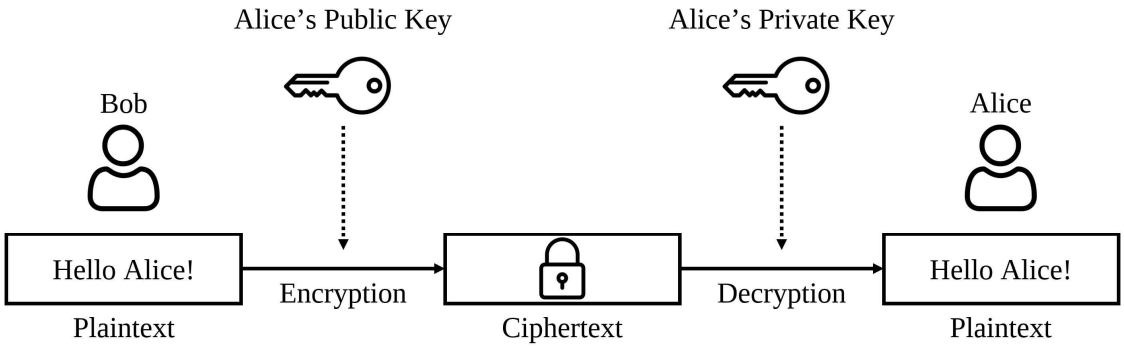
- Stream ciphers, on the other hand, do not encrypt the input message directly. Instead, they generate pseudorandom numbers, which are combined with the input message to produce the ciphertext, as shown in [Figure 2-3]. Typically, XOR operations are used to combine the pseudorandom numbers with the input message. Stream ciphers are known for their fast encryption speed and the non-propagation of errors, but they suffer from lower diffusion.



[Figure 2-3] Stream cipher framework (Encryption only)

2.2 Public-Key Cryptography

A public-key encryption system uses different keys for encryption and decryption, as shown in [Figure 2-4], and is therefore also referred to as an asymmetric-key encryption algorithm.



[Figure 2-4] Public-key cryptography architecture

In this cryptography, encryption is performed using a public key, while decryption is carried out using a private key, also known as a secret key. The public key is known to all participants in the

network, but only the key owner knows the private key. As long as the private key remains secure, decrypting the ciphertext is impossible, making public-key encryption highly secure for transmitting information or messages. Due to these characteristics, public-key encryption can also provide additional functionalities such as authentication, integrity, and non-repudiation. However, compared to symmetric-key encryption, it is significantly slower and requires larger key sizes. For this reason, public-key encryption is not commonly used for general message transmission but is instead employed for tasks such as key exchange and authentication. Examples of public-key cryptography include RSA.

2.3 Post-Quantum Cryptography

With the advancement of quantum computing, cryptographic systems based on traditional mathematical challenges have begun to face threats. Grover's algorithm, which can be implemented on quantum computers, is an optimized search algorithm capable of performing attacks such as brute force effectively. While Grover's algorithm poses a threat to symmetric-key cryptography and hash functions, increasing the key length can mitigate its impact. Shor's algorithm, on the other hand, solves problems such as integer factorization and discrete logarithms. Among public-key cryptography, those based on integer factorization, such as RSA, are highly vulnerable to Shor's algorithm, and no effective countermeasures are currently known.

In response to the threat posed by quantum computers, the U.S. National Institute of Standards and Technology (NIST) launched a competition to standardize new cryptographic algorithms resistant to quantum attacks, known as post-quantum cryptography (PQC). As a

result, Kyber was selected as the standard for PKE/KEM algorithms, while Dilithium, Falcon, and SPHINCS+ were chosen for digital signature algorithms. In Republic of Korea, a competition named KpqC is being held to select a PQC standard. Currently, the competition is in its second round, with NTRU+, PALOMA, REDOG, and SMAUG-T competing as candidates for PKE/KEM, and AIMer, HAETAE, MQ-Sign, and NCC-Sign competing in the digital signature category. Post-quantum cryptography involves more fundamental problems than traditional symmetric-key and public-key cryptography.

- Lattice-based cryptography: It is based on the Shortest Vector Problem (SVP) and Closest Vector Problem (CVP), which involve finding the shortest vector in a lattice when two integer vectors exist. Lattice-based cryptography is known for its fast computation speed and relatively small key and signature sizes. Due to these advantages, many algorithms in the post-quantum cryptography competition are based on lattice problems. Examples of lattice-based cryptographic algorithms include Kyber, Dilithium, Falcon, NCC-Sign, and HAETAE.
- Code-based cryptography: This problem generates public/private key pairs using error correction codes, which control errors in signals. Code-based cryptography has been subject to security analysis for a longer time compared to other hard problems, thus earning a high level of trust. Classic McEliece is a representative example of code-based cryptography.
- Hash-based cryptography: It relies on the collision resistance of hash functions. Although quantum algorithms can compromise hash functions, security can be maintained by extending the hash output length. Additionally, if a security vulnerability is discovered in a

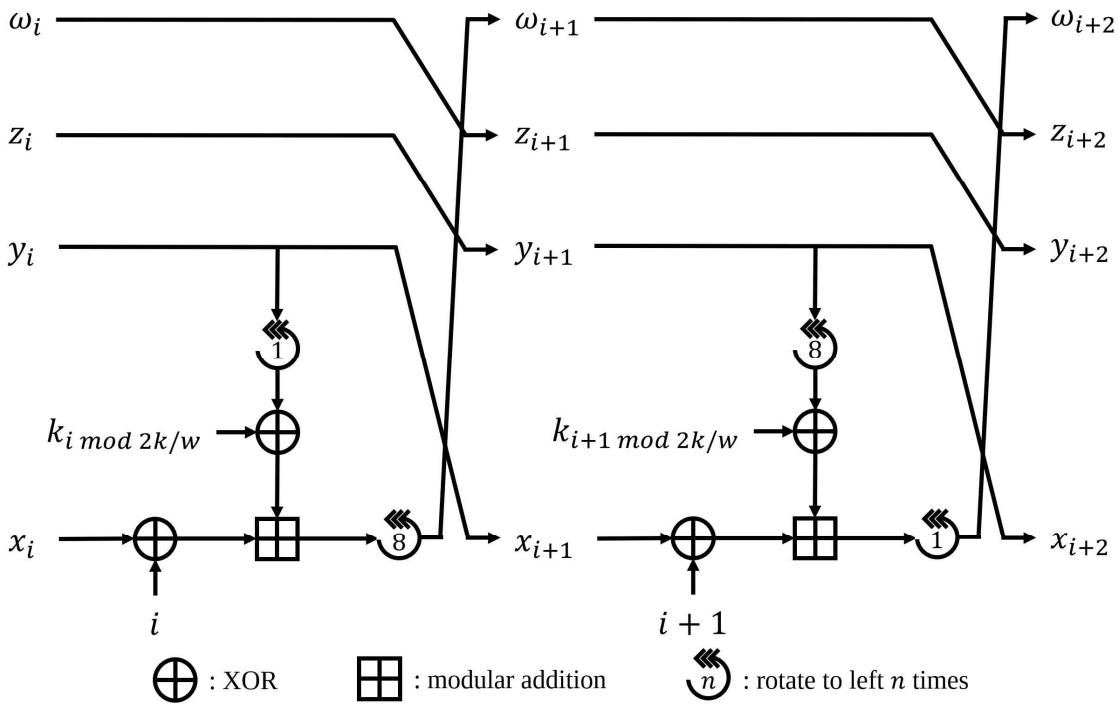
hash function used in the cryptographic scheme, it can be replaced with another hash function to maintain security. SPHINCS+ is an example of hash-based cryptography.

- Multivariate based cryptography: This cryptographic method is based on the difficulty of solving systems of multivariate polynomial equations over finite fields. Compared to other problems, proving security mathematically is relatively straightforward. Since the primary operation involves solving matrices of polynomials, effective implementations can be achieved using the Gaussian Elimination algorithm. Rainbow and MQ-Sign are examples of multivariate polynomial cryptography.

2.4 Target Cryptographic Algorithm

2.4.1 Block Cipher CHAM

CHAM is a block cipher introduced in South Korea in 2017, designed with low-end processors in mind. In 2019, revised CHAM was published, with the only difference between the original and revised versions being the number of rounds, while the core design remains the same. CHAM is an ARX-based algorithm that divides the input data into four blocks, as shown in [Figure 2-5]. Although the operations per round are identical, the number of left shifts differs between odd and even rounds. [Table 2-1] summarizes the parameters of CHAM.



[Figure 2-5] Round function structure of CHAM

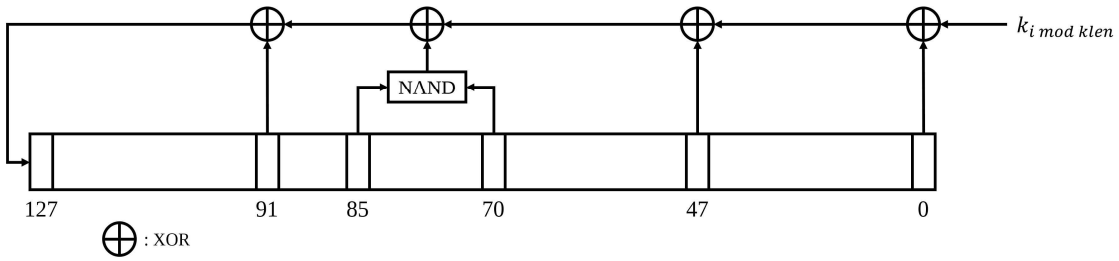
[Table 2-1] List of CHAM parameters (n: block size, k: key size, ω: word size, r: number of rounds)

Cipher	n	k	ω	r
CHAM-64/128	64	128	16	88
CHAM-128/128	128	128	32	112
CHAM-128/256	128	256	32	120

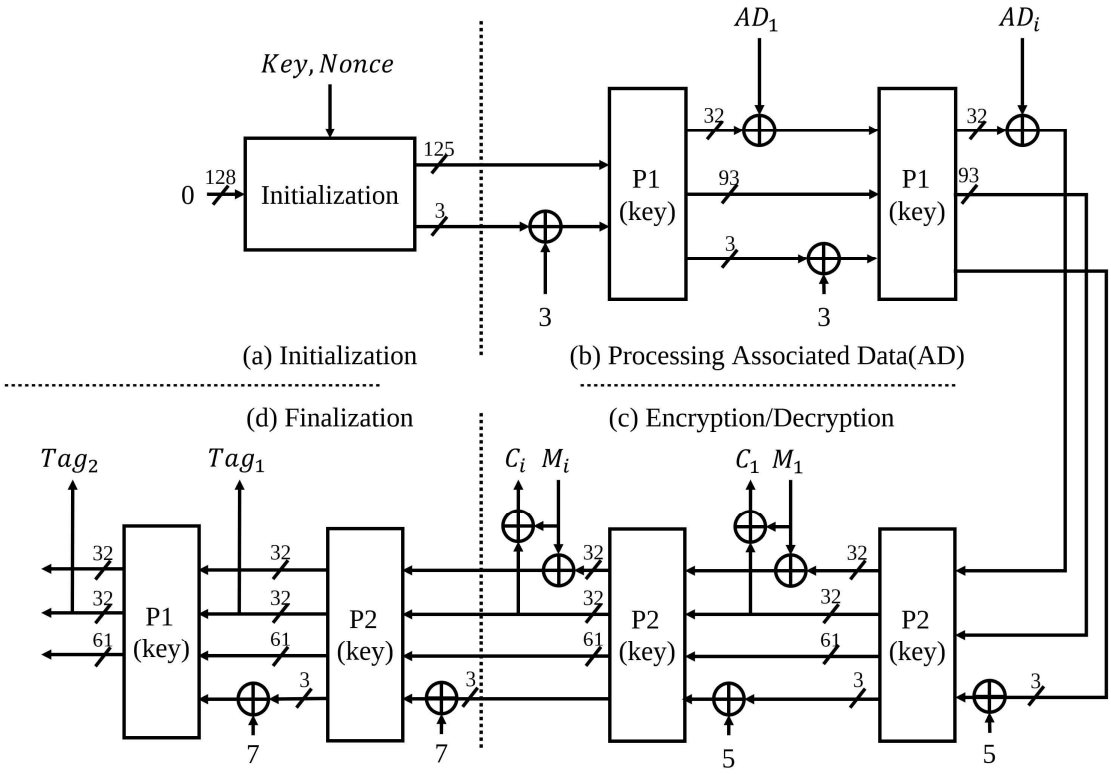
2.4.2 Lightweight Cipher TinyJAMBU

TinyJAMBU is a permutation-based variant of the block cipher JAMBU. The encryption and decryption processes consist of five stages: Initialization, Processing Associated Data, Encryption/Decryption, Finalization, and Verification. TinyJAMBU employs a keyed permutation structure with NLFSR, as shown in

[Figure 2-6], and [Figure 2-7] shows the TinyJAMBU mode. [Table 2-2] provides its pseudocode representation.



[Figure 2-6] NLFSR for Keyed permutation of TinyJAMBU



[Figure 2-7] Whole structure of TinyJAMBU

2.4.3. Post Quantum Cryptography Rainbow

Rainbow, a finalist in the third round of the NIST post-quantum cryptography competition, is a multivariate polynomial-based digital signature algorithm. It leverages the UOV problem and offers faster

signing and verification speeds compared to other algorithms, along with smaller signature sizes. However, Rainbow has the disadvantage of slow key generation and significantly larger key sizes than other post-quantum algorithms, especially lattice-based cryptography. Variants of Rainbow, such as the circumzenithal and compressed versions, are available to reduce key size, and the parameters are summarized in [Table 2-2].

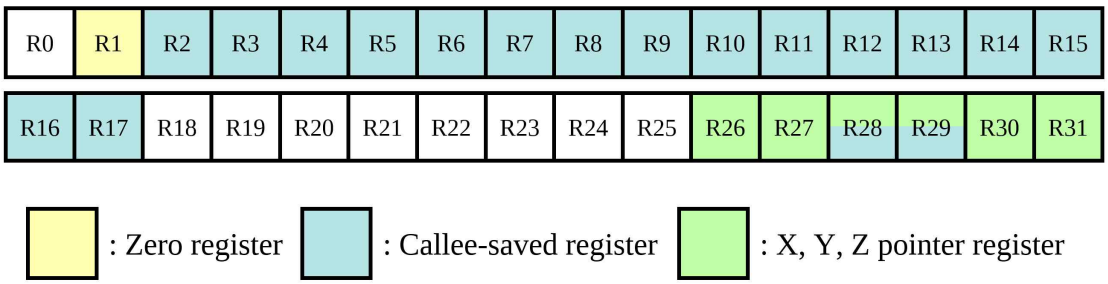
[Table 2-2] Key and signature size of Rainbow signature. (Key size unit: KB, Signature size unit: bit)

Security Level	Parameters	Public key size	Private key size	Signature size
Standard Rainbow				
I	(GF(16),36,32,32)	157.8	101.2	528
III	(GF(256),68,32,48)	681.4	611.3	1,312
V	(GF(256),96,36,64)	1,885.4	1,375.7	1,632
Circumzenital Rainbow				
I	(GF(16),36,32,32)	58.8	101.2 (99.0)	528
III	(GF(256),68,32,48)	258.4	611.3 (603.0)	1,312
V	(GF(256),96,36,64)	523.5	1,357.7 (1,631.8)	1,696

2.5 Target Processor

2.5.1 8-bit AVR Microcontroller

The 8-bit AVR processor, first introduced in 1996 with the ATmega series, is a RISC-based processor. It features 32 general-purpose 8-bit registers, and for any operation, values must first be loaded from memory into the registers. Similarly, the results of operations must be stored back from the registers to memory, with each process requiring 2 cycles. [Figure 2-8] illustrates the structure of AVR registers.



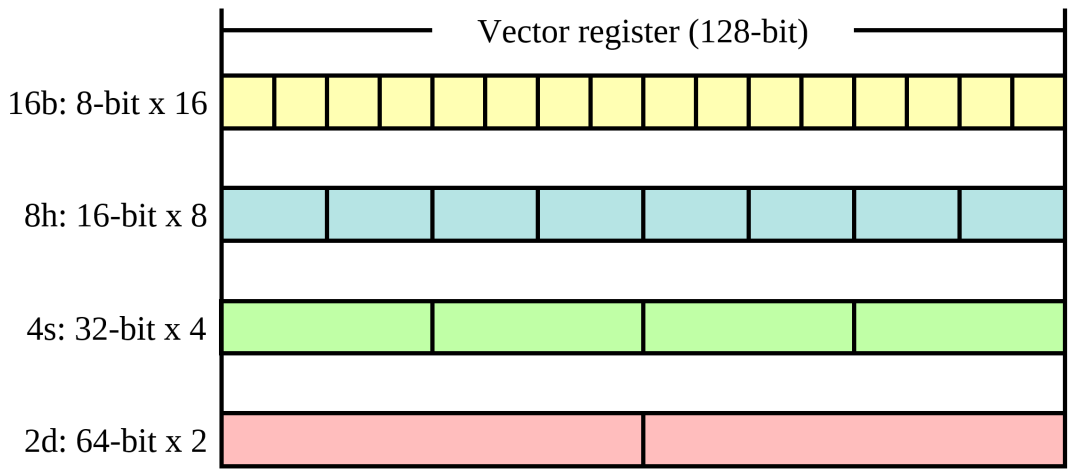
[Figure 2-8] Structure of AVR registers

The R1 register serves as the zero register, allowing flexible use, but it must always hold the value zero when the operation is complete. Therefore, if the R1 register is used, it is recommended to clear the register with the CLR instruction before the operation finishes. Registers R2 through R17, as well as R28 and R29, are callee-saved registers. These registers might hold important values necessary for computations prior to function calls, so their contents should be temporarily saved before usage and restored after the operation. Lastly, registers R26 through R31 are pointer registers, with R26 and R27, R28 and R29, and R30 and R31 paired to form the X, Y, and Z pointer registers. These pointer registers are used to access memory, either to load or store values via pointers. It is

important to note that since the Y pointer register is callee-saved, its value needs to be preserved before use.

2.5.2 64-bit ARM Processor

The ARMv8 processor is a high-performance processor within the embedded processor category, commonly used in devices such as smartphones and laptops. Its register configuration consists of 64-bit general-purpose registers and 128-bit vector registers. The vector registers, which support parallel computation, have a maximum size of 128 bits, though the effective size for storage is limited to 64 bits, allowing for up to two values to be stored. The arrangement specifier determines how the internal data is treated in terms of bit size, which can be set when utilizing vector instructions (also referred to as NEON). [Figure 2-9] illustrates how data is handled in vector registers based on the arrangement specifier.



[Figure 2-9] Controlling vector registers via arrangement specifiers

2.6 Previous Work

Seo et al. conducted optimized implementations of the LEA and HIGHT block ciphers on an 8-bit AVR processor. For this purpose, they optimized the rotation operations, particularly the right rotation, by utilizing BST and BLD instructions to reduce the number of instructions used. Additionally, they addressed the limited number of registers available on the AVR processor by optimizing register usage strategies. As a result, the C implementation of LEA required 326 cpb for key generation, 263 cpb for encryption, and 236 cpb for decryption, whereas the AVR-optimized implementation achieved 235, 168, and 176 cpb for the same processes, respectively. For HIGHT, the C implementation required 156, 537, and 525 cpb for key generation, encryption, and decryption, while the AVR-optimized implementation achieved 58, 160, and 161 cpb, respectively.

Kim et al. proposed the FACE-LIGHT algorithm, a lightweight implementation of AES-CTR mode tailored for low-resource processors like AVR. Their approach involved designing a new cache table to enable partial precomputation, extending the precomputation capability from two rounds in the original FACE algorithm to three rounds in FACE-LIGHT. The resulting implementation achieved optimal performance, with AES-128, AES-192, and AES-256 requiring 1,967, 2,449, and 2,931 cpb, respectively.

Kwon et al. optimized the block cipher SIMON for the AVR processor. Their work leveraged the characteristics of 8-bit processors to demonstrate that specific registers could be precomputed. They also calculated operational parts based on the plaintext length. The implementation resulted in performance improvements ranging from 1.5% to 5.3%.

Seo et al. also optimized the post-quantum cryptographic algorithms SIDH and SIKE on the ARMv8 processor. Their approach

focused on accelerating the Montgomery multiplier and extensively utilized 64-bit operations. For SIDH, the C implementation required 643.8 million or 574.3 million clock cycles, depending on the processor, while the proposed method reduced this to 133.3 million and 90.3 million clock cycles, respectively. For SIKE, the C implementation required 626.3 million or 558.5 million clock cycles, but the optimized implementation reduced this to 129.6 million and 87.8 million clock cycles, respectively.

Kim et al. optimized the post-quantum signature scheme CRYSTALS-Dilithium for ARM processors. Their proposed method utilized ARM NEON parallel instructions to optimize the NTT multiplier and employed layer merging to reduce memory access frequency. This approach resulted in performance improvements of 49%, 113%, and 41% in the key generation, signature generation, and verification processes, respectively.

3. Optimized Implementation of Target Cipher

3.1 CHAM with Precomputation

3.1.1 Skip Rounds by Precomputation

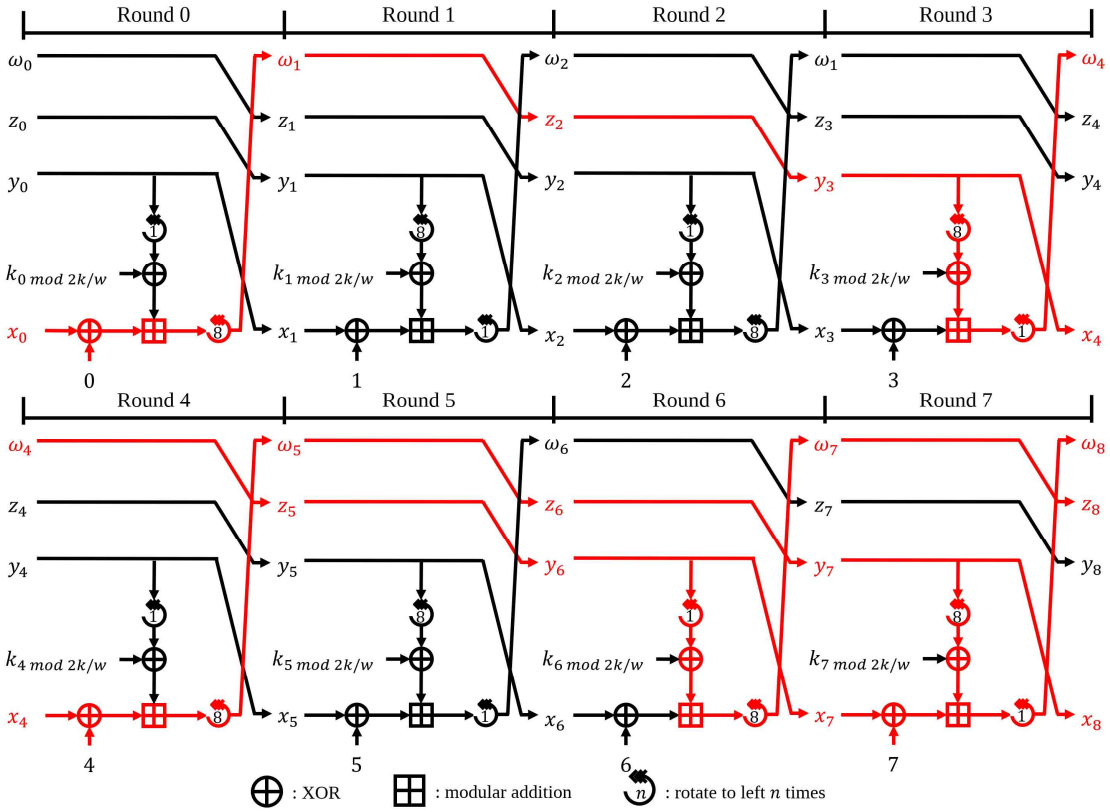
In the Counter (CTR) mode of operation, rather than encrypting the plaintext directly, a fixed nonce is first encrypted, and the result is then XORed with the plaintext to perform the encryption. The nonce is divided into a fixed part, generated randomly, and a counter that represents the block number. Since the fixed part remains the same for all blocks, it always produces the same result, making precomputation possible. Due to the structure of CHAM's round function, as the encryption progresses, the block storing the counter begins to affect other blocks, making it necessary to analyze the flow of the counter block. [Figure 3-1] highlights the flow of counter values in each round, marked in red.

In round 0, only one block is affected by the counter, but by round 9, all blocks are influenced by the counter. This implies that certain computations during the first nine rounds can be omitted. However, for the sake of implementation efficiency, the actual precomputations are conducted up to the first eight rounds. This corresponds to approximately 9.09%, 7.14%, and 6.67% of the total rounds for CHAM's 88, 112, and 120-round variants, respectively. The specific computations that can be omitted for each round are as follows:

- Round 0: Addition of the round key to the second block.
- Round 1: Entire round.
- Round 2: Entire round.
- Round 3: Addition of the round counter to the first block.
- Round 4: Addition of the round key to the second block (same as

round 0).

- Round 5: Entire round.
- Round 6: Addition of the round counter to the first block (same as round 3).
- Round 7: No precomputable operations.



[Figure 3-1] Flow of counter values in CHAM CTR mode of operation

Although [Figure 3-1] shows that precomputations are possible in round 8, they were not implemented. Implementing round 8 results in less than 1 cpb performance improvement. However, due to the paired round structure of CHAM, implementing round 9 separately would lead to inefficiency, and thus, round 8 is not implemented.

3.1.2 Logical Block Rotation

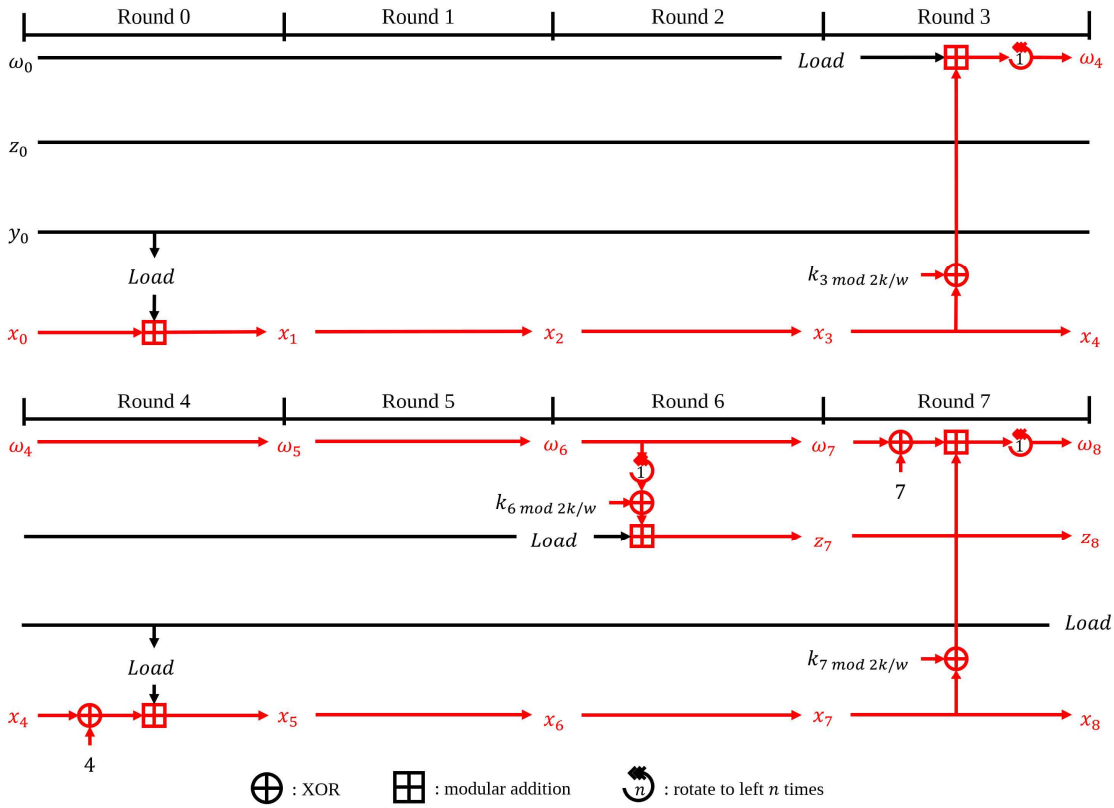
At the end of each round in CHAM, the blocks undergo a word-wise rotation. In CHAM-64/128, the rotation occurs in 16-bit units, while for the other CHAM variants, the rotation occurs in 32-bit units. Although the block rotation can be implemented in AVR assembly using the MOV instruction, a more efficient implementation can be achieved using the MOVW instruction, which moves data in 16-bit word units. However, the implementation can be further optimized by applying a logical block rotation, thereby omitting the actual rotation.

Also in CHAM, the blocks used for computation in each round are the first and second blocks. Since the block rotation occurs at the end of every round, the blocks involved in computations do not change. Without performing the rotation, the first and second blocks are used in the first round, the second and third blocks in the second round, the third and fourth blocks in the third round, and the fourth and first blocks in the fourth round. This process repeats starting from the fifth round. Thus, the original implementation grouped odd and even rounds in pairs, but by applying logical block rotation, the implementation can be optimized into 4-round units. This is also the reason why round 8 was not implemented in Section 3.1.1; implementing round 8 would require separate implementation for rounds 9, 10, and 11, making efficient implementation more difficult.

A similar approach can be applied to the 8-bit left rotation operation used in each round. Since the AVR registers store 8 bits, the 8-bit left rotation can be implemented using register shifting instructions instead of a rotation instruction. However, instead of actually shifting the registers, the operation can be logically treated as if the value has been rotated. In the following round, operations

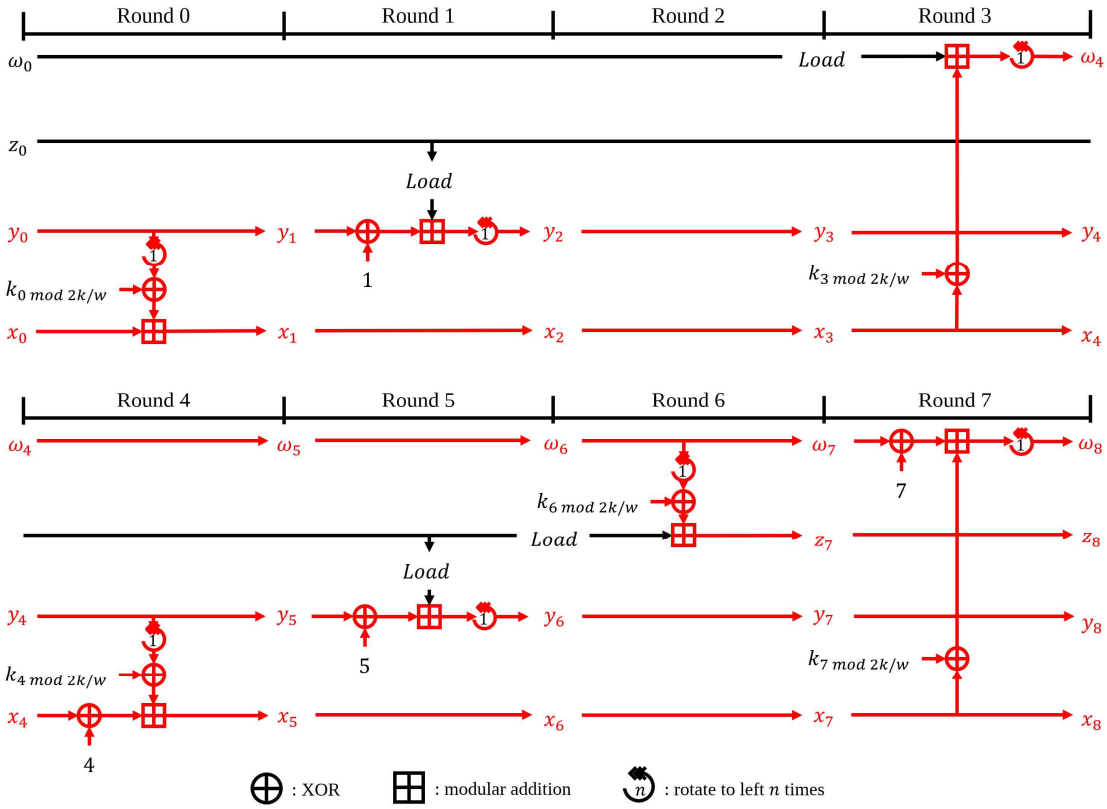
that would normally apply to the lower register can be applied to the upper register, effectively bypassing the rotation operation.

[Figure 3-2] illustrates the structure of the first eight rounds of CHAM with all proposed techniques applied.



[Figure 3-2] Optimized CHAM structure with CTR mode of operation

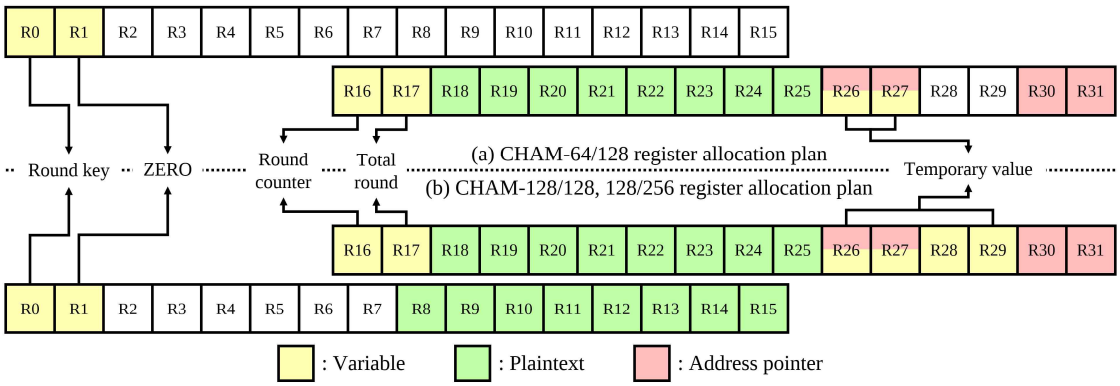
In block cipher counter mode, a 16-bit counter can be used, but a 32-bit counter is generally preferred. In CHAM-64/128, where internal blocks are processed in 16-bit units, the structure of a 16-bit counter is depicted in [Figure 3-2]. If a 32-bit counter is used, two blocks are required to store the counter value, resulting in a slightly different structure as shown in [Figure 3-3]. No additional implementation is required for CHAM-128/128 and CHAM-128/256, as they operate with 32-bit units.



[Figure 3-3] Optimized 32-bit counter CHAM-64/128 structure

3.1.3 Register Scheduling, Instructions Used and Implementation

To implement the proposed CHAM cipher, the register allocation is planned as shown in [Figure 3-4].



[Figure 3-4] Register allocation plan for proposed CHAM

Register R0 is used to store the round key value, while R1 serves

as a zero register. In AVR architecture, R1 is conventionally assigned as the zero register by default, so there is no need to use another register. Register R16 holds the total round count for CHAM, with 88 rounds for CHAM-64/128, 112 for CHAM-128/128, and 120 for CHAM-128/256. Register R17 is used to store the present round counter value, which increments with each round. Registers R18 through R25 store the plaintext composed of the nonce and counter. Lastly, registers R26 and R27, R30 and R31 are used both as the X, Z register, respectively. And also R26 and R27 are used for temporary registers to store intermediate values during computations. In case of CHAM-128/128 and CHAM-128/256, these are required more plaintext registers and temporary registers. So in this cases, R8 to R15 used to store plaintext and R28, R29 used for additional temporary registers.

[Table 3-1] summarizes the instructions used in the implementation, while [Table 3-2] shows the code for the first 8 rounds where round skipping is applied, specifically for CHAM-64/128.

[Table 3-1] List of instructions used in implementation for CHAM in alphabetical order

Mnemonic	Operands	Description	Operation
ADC	Rd, Rr	Add with Carry	Rd←Rd+Rr+C
ADD	Rd, Rr	Add without Carry	Rd←Rd+Rr
EOR	Rd, Rr	Exclusive OR	Rd←Rd⊕Rr
LD	Rd, X+	Load Indirect and Post Increment	Rd←(X) X←X+1

LD	Rd, Y+	Load Indirect and Post Increment	Rd←(Y) Y←Y+1
LD	Rd, Z+	Load Indirect and Post Increment	Rd←(Z) Z←Z+1
LDI	Rd, K	Load Immediate	Rd←K
LPM	Rd, Z	Load Program Memory	Rd←(Z)
LSL	Rd	Logical Shift Left	Rd(n+1)←Rd(n) Rd(0)←0 C←Rd(7)
MOV	Rd, Rr	Copy Register	Rd←Rr
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd←Rr+1:Rr
POP	Rd	Pop Register from Stack	Rd←STACK
PUSH	Rr	Push Register on Stack	STACK←Rr
ROL	Rd	Rotate Left Through Carry	Rd(n)←Rd(n+1) Rd(7)←0 C←Rd(0)
ST	X+, Rr	Store Indirect and Post Increment	(X)←Rr (X)←X+1
ST	Y+, Rr	Store Indirect and Post Increment	(Y)←Rr (Y)←Y+1
ST	Y+, Rr	Store Indirect and Post Increment	(Z)←Rr (Z)←Z+1

Lines 1−4 represent round 0, where the LDI instruction is used to retrieve precomputed values from the second block and add them to the first block containing the counter value. Rounds 1 and 2 are skipped and do not appear in the code. Lines 5−18 correspond to

[Table 3-2] Implementation code of proposed CHAM (RC: round counter, RK: round key, X00~X31: plaintext, XT: temporary register, Zero: zero register)

Line	Code	Comment
1:	LDI XT0, 0x45	Round 0 start
2:	LDI XT1, 0x65	
3:	ADD X00, XT0	
4:	ADC X01, XT1	ROL8 skipped
5:	ADIW R30, 6	Round 3 start
6:	MOVW XT0, X00	
7:	LD RK, Z+	
8:	EOR XT0, RK	
9:	LD RK, Z+	
10:	EOR XT1, RK	
11:	LDI X30, 0x65	
12:	LDI X31, 0x77	
13:	ADD X30, XT0	
14:	ADC X31, XT1	
15:	LSL X30	
16:	ROL X31	
17:	ADC X30, ZERO	
18:	LDI RC, 4	
19:	EOR X01, RC	Round 4 start, XOR on upper register
20:	LDI XT0, 0xDC	
21:	LDI XT1, 0xCA	
22:	ADD X01, XT0	XOR in reverse order of registers
23:	ADC X00, XT1	XOR in reverse order of registers
24:	LDI X10, 0x02	Round 5 start

```
25:    LDI  X11,  0X32
26:   ADIW  R30,  4      Round 6 start
27:   MOVW  XT0,  X30
28:    LSL  XT0
29:    ROL  XT1
30:    ADC  XT0,  ZERO
31:     LD  RK,   Z+
32:    EOR  XT0,  RK
33:     LD  RK,   Z+
34:    EOR  XT1,  RK
35:    LDI  X20,  0x0B   Load in reverse order of registers
36:    LDI  X21,  0x3D   Load in reverse order of registers
37:    ADD  X21,  XT0
38:    ADC  X20,  XT1
39:    LDI  RC,   7
40:   MOVW  XT0,  X00    Round 7 start
41:    EOR  X30,  RC
42:     LD  RK,   Z+
43:    EOR  XT1,  RK
44:     LD  RK,   Z+
45:    EOR  XT0,  RK
46:    ADD  X30,  XT1
47:    ADC  X31,  XT0
48:    LSL  X30
49:    ROL  X31
50:    ADC  X30,  ZERO
51:    INC  RC
```

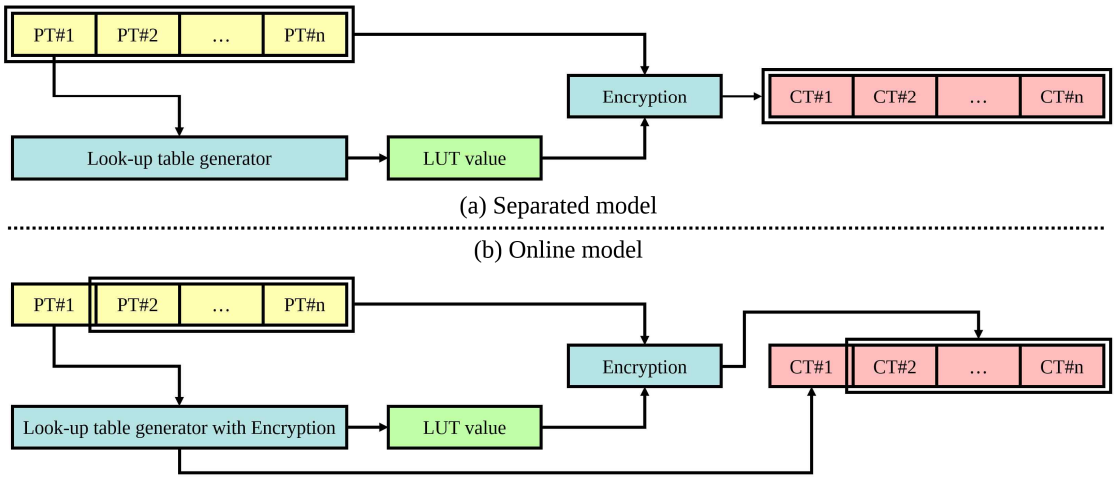
round 3, where the ADIW instruction first shifts the lower address of the skipped round key, followed by normal operations, with lines

11-12 fetching precomputed values. The round counter, which had not been modified until this point, is updated at line 18. Lines 19-23 implement round 4, which is nearly identical to round 0, except that an XOR operation is performed with the round counter before the computation begins. In round 0, XOR with 0 would yield the same result, so it was omitted, but round 4 requires XOR due to the round counter.

Lines 24-25 implement round 5, which, like rounds 1 and 2, can be fully skipped. However, a load operation is added, as the values are needed for round 6. Lines 26-39 represent round 6, structured similarly to round 3. Finally, the implementation of round 7 in lines 40-51 completes the process, as no further parts can be skipped.

Since logical block rotation, described in section 3.1.2, is applied, no word-level rotation occurs at the end of each round. The eight left-rotate operations are also omitted in lines 4, 19, 22, 23, 35, and 36. These rotations would have required the MOV instruction and an additional temporary register, necessitating a total of three MOV instructions, which were successfully avoided. CHAM-128/128 and CHAM-128/256 can be implemented in a similar manner.

The implementation outlined in [Table 3-2] pertains to a fixed-key scenario, assuming that the secret key remains unchanged. The implementation utilizes the LDI instruction to load predefined values, meaning that if the key varies, these values must also be adjusted. Therefore, in environments where the key changes, this implementation is not feasible. Additional modifications are required to accommodate key variability, as illustrated in [Figure 3-5].



[Figure 3-5] Two implementation scenarios for the variable key model

The scenario of generating precomputed values while processing the first block can be divided into two cases. The first case involves generating only the precomputed values without encrypting the first block, corresponding to [Figure 3-5] (a). In this case, only the first 8 rounds are executed to generate the precomputed values, and then the computation halts. Once these values are generated, all blocks, including the first, can be precomputed. This approach is referred to as the "separated model."

The second case involves generating the precomputed values while simultaneously processing the first block to completion, as shown in [Figure 3-5] (b). This is referred to as the "online model." The advantage of the online model is that it avoids repeating the initial 8 rounds for the first block. However, it has the drawback of increased code complexity. Even in the variable-key scenario, the register allocation remains largely unchanged from [Figure 3-4]. However, due to the need to load the precomputed table, the address of the table is stored in registers R28 and R29, which are then used as the Y pointer register. In the case of CHAM-128/128 and

CHAM-128/256, the temporary values that were previously stored in R28 and R29 are moved to R2 and R3, allowing R28 and R29 to function as the Y pointer register for table access.

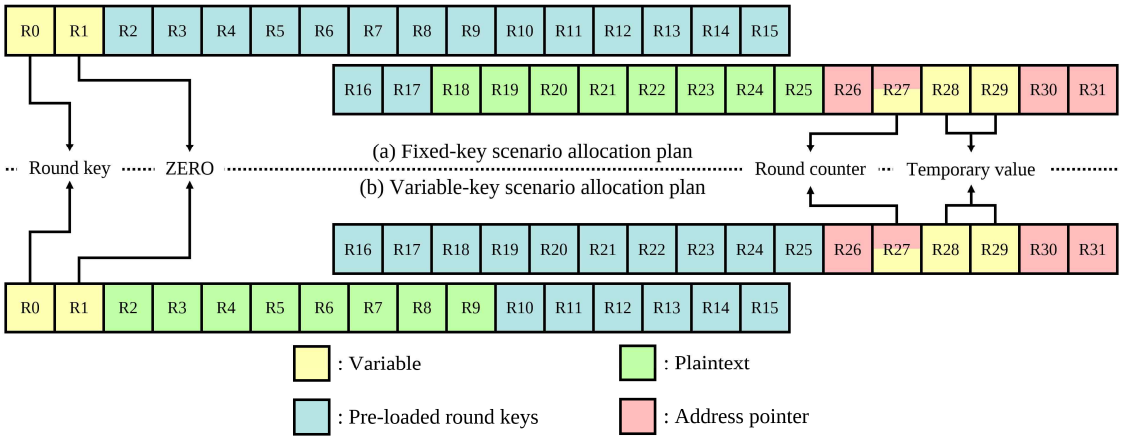
Assuming that the key changes, the first block of the input plaintext cannot undergo precomputation. Consequently, standard encryption must be performed for the first block. During this process, there is a need to store precomputed values, which will later be used for precomputation. In the proposed CHAM scheme, there are five points at which precomputed values are called. As a result, when encrypting the first block, five values must be stored to enable precomputation for subsequent blocks. Since CHAM-64/128 uses 16-bit words, five values require storing 10 bytes. In the case of CHAM-128/128 and CHAM-128/256, which use 32-bit words, double the storage—20 bytes—is required.

Once the encryption of the first block is complete, precomputed values can be used for subsequent blocks. From this point forward, the initial eight rounds can be rapidly processed using the precomputation technique outlined in the code provided in [Table 3-2].

3.1.4 Alternative Implementation: Furious CHAM

The total round keys for CHAM-64/128 amount to 32 bytes, and they are reused over 88 rounds. Upon reviewing the register allocation plans in [Figure 3-4] (a), it is evident that several registers remain unused during the implementation of CHAM-64/128. Given the structural characteristic of CHAM, where round keys are reused across rounds, preloading the round keys could reduce the number of load operations, thereby enhancing computational efficiency. This optimized approach is termed "Furious CHAM."

With 16 registers available, it becomes possible to preload half of the round keys (16 bytes out of the total 32 bytes). This allows the omission of round key loads for 40 of the 80 rounds, excluding the initial 8 rounds. Since each round involves the loading of 2 bytes of round keys, omitting 80 LD instructions corresponds to a savings of 160 cycles.



[Figure 3-6] Register allocation plan for furious CHAM

To implement Furious CHAM, a new register allocation plan is proposed, as outlined in [Figure 3-5] (a) and (b). In the fixed-key scenario at [Figure 3-5] (a), only the plaintext and round key pointers are required, allowing the free use of the Y register (R28, R29). Additionally, since R28 and R29 are not callee-saved registers, the need for PUSH and POP operations is eliminated, providing a further efficiency advantage.

In the variable-key scenario at [Figure 3-5] (b), a similar register allocation is applied, but the storage location of the plaintext differs. In the fixed-key scenario, the precomputed values remain constant, allowing the use of the LDI instruction, which is one cycle faster than the LD instruction. However, LDI can only be used with

registers R16 to R31, so plaintext was stored in R18 to R25. In the variable-key scenario, since the precomputed values cannot be fetched via LDI and must be accessed via LD, this constraint does not apply.

Nonetheless, even when using LD, it is possible to maintain the same register allocation. The reason for altering the plaintext registers more precisely lies in the need to preserve the pointer value passed as a parameter. Unlike the fixed-key scenario, the variable-key scenario requires an additional pointer to store the precomputed values (table pointer). Since the parameter pointers are stored starting at R24 and R25, the third pointer is stored in R20 and R21. If the plaintext pointer is moved to X or Z and values are loaded as in the fixed-key scenario, there is a risk of losing the table pointer stored in R20 and R21. While the use of MOVW, PUSH, and POP instructions could prevent this, it would result in additional cycle costs, making the alternative register allocation plan more efficient.

Although it is possible to load the round keys before the plaintext, this approach is less efficient because round keys require more registers than plaintext. Therefore, using MOVW or PUSH and POP instructions would be unavoidable in that case.

3.2 TinyJAMBU with Reverse Bitwise Shift

3.2.1 Reverse Bitwise Shift

TinyJAMBU exhibits a high dependency on keyed permutations, as demonstrated in [Table 3-3], where the number of keyed permutations performed at each stage is indicated.

[Table 3-4] presents the pseudocode of the keyed permutations. It can be observed that when generating t2, t3, and t4, the same

[Table 3-3] Number of keyed permutations each step in tinyJAMBU

Step	Key length		
	128-bit	192-bit	256-bit
Initialization: Key setup	1,024	1,152	1,280
Initialization: Nonce setup	640	640	640
Processing associated data	640	640	640
Encryption/Decryption	1,024	1,152	1,280
Finalization	1,024 / 640	1,152 / 640	1,280 / 640

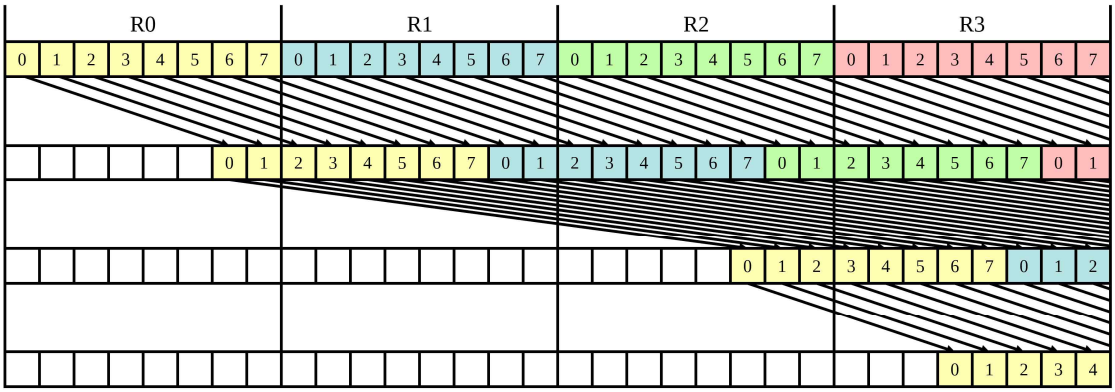
state block is used, and it is shifted in the same direction. Since each state block of TinyJAMBU consists of 32 bits, it occupies four registers on the AVR processor. In the pseudocode of [Table 3-4], the s_2 state block undergoes shifts of 6, 21, and 27 times, respectively.

[Table 3-4] Pseudocode for keyed permutation(<<n: bitwise left shift n times, >>n: bitwise right shift n times, |: bitwise OR, &: bitwise AND, ^: bitwise XOR, ~: bitwise NOT)

Input: State s_0, s_1, s_2, s_3(32-bit each), Key k, Round n	
Output: State s_0, s_1, s_2, s_3(32-bit each)	
1:	StateUpdate(s_0, s_1, s_2, s_3, k, n)
2:	for i = 0 to n
3:	t_1 = (s_1>>15)|(s_2<<17)
4:	t_2 = (s_2>>6)|(s_3<<26)

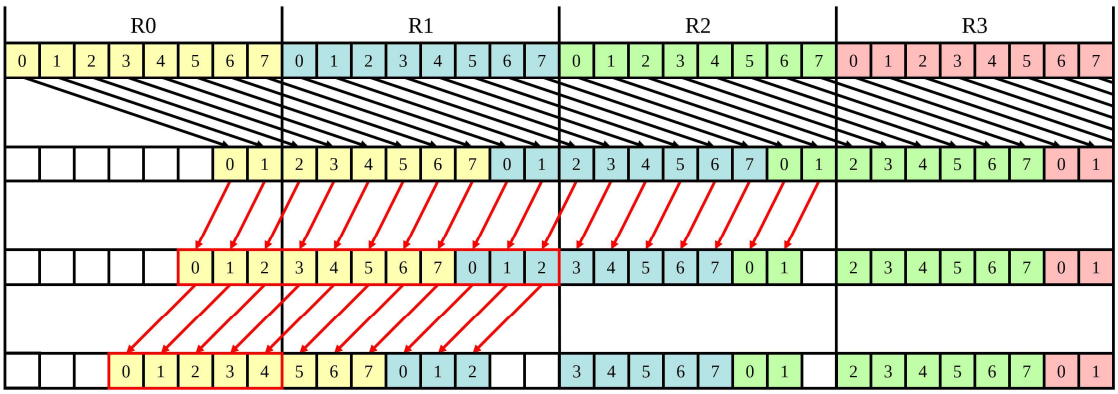
5:	$t_3 = (s_2 >> 21) \mid (s_3 << 11)$
6:	$t_4 = (s_2 >> 27) \mid (s_3 << 5)$
7:	feedback = $s_0 \hat{} t_1 \hat{} (\sim(t_2 \& t_3)) \hat{} t_4 \hat{} k$
8:	$s_0 = s_1$
9:	$s_1 = s_2$
10:	$s_2 = s_3$
11:	s_3 = feedback
12:	end for

However, by utilizing the values stored in the registers, as shown in [Figure 3-4], a total of 27 shifts can be reduced to 6, 15, and 6 shifts, respectively. Consequently, the required size for s_2 during computation is 26 bits, 11 bits, and 5 bits, respectively. By reusing the values left in the registers, it becomes possible to reduce the number of shifts compared to the reference implementation, leading to a more efficient computation.



[Figure 3-7] S_2 state computation structure using AVR assembly instructions

Additionally, if the 8-bit operations of the AVR processor's registers are utilized, the number of shifts can be further minimized. In the second stage of [Figure 3-4], the remaining 11 bits are stored across R0, R1, and R2, with only the most significant bit of R2 being used. Thus, instead of shifting 15 times to obtain 11 bits, shifting once in the opposite direction allows the required 11 bits to be retained in R0 and R1. Similarly, the final 5 bits can be found by shifting twice in the opposite direction, as only the second bit of R1 is needed. Therefore, by reversing the direction of the shifts, as illustrated in [Figure 3-7], all necessary values can be obtained with 6 shifts in the first stage, 1 shift in the second, and 2 shifts in the third, resulting in a total of 9 shifts to complete the computation of the s_2 block. [Figure 3-7] also shows that the red outline values are the same as the original results.

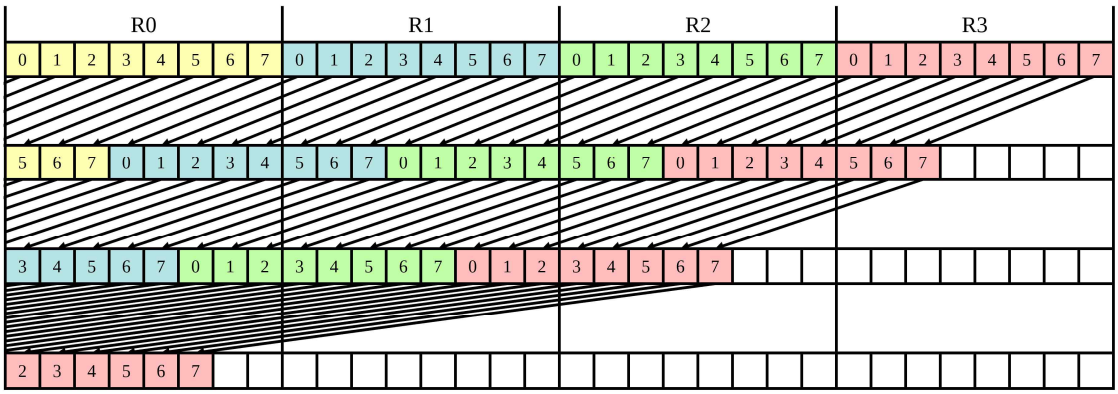


[Figure 3-8] Proposed RBS applied to s_2 state calculation

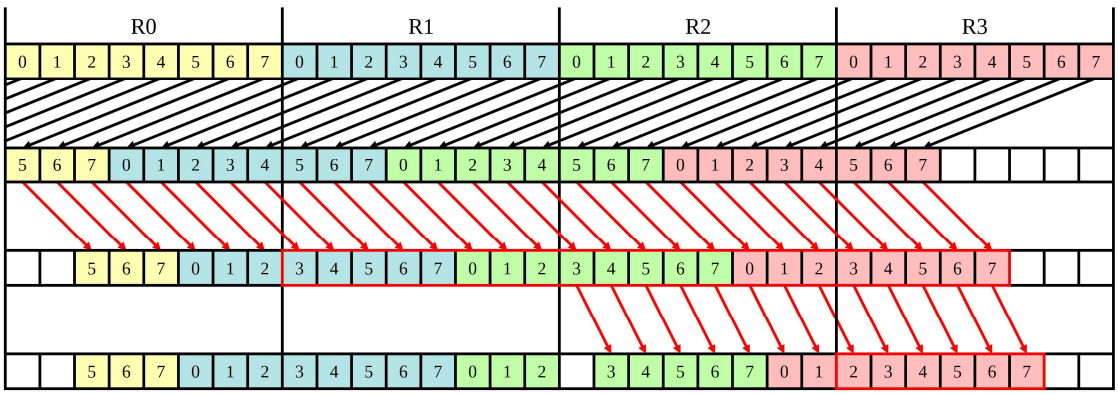
The s_3 state block can be processed similarly. However, unlike s_2, s_3 is shifted to the right, with the shift count decreasing in later stages. Since some values may already be lost in the early stages due to the large number of shifts, the computation proceeds in reverse order. Typically, as shown in [Figure 3-8], 5, 6, and 15

shifts are required, totaling 26 shifts.

By reversing the direction of the shifts, as done with the s_2 block, the number of shifts can be reduced which described at [Figure 3-9]. Specifically, the first stage proceeds with 5 shifts as usual, while the second stage, which requires 21 bits, can be obtained by shifting twice in the opposite direction and utilizing the values from R1, R2, and R3. In the final stage, which requires 6 bits, the values in R2 and R3 are shifted once in the opposite direction, and only the value from R3 is used. Similarly, the values indicated by the red line in [Figure 3-9] are the same as the existing results at [Figure 3-8].



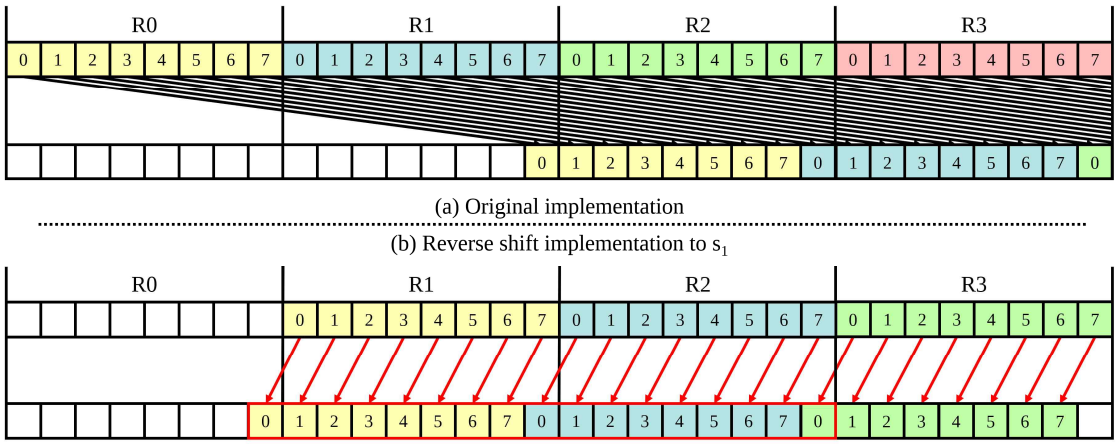
[Figure 3-9] S_3 state flow with AVR assembly implementation



[Figure 3-10] S_3 state calculation with RBS technique applied

The temporary value t_1 is generated using the s_1 and s_2 state

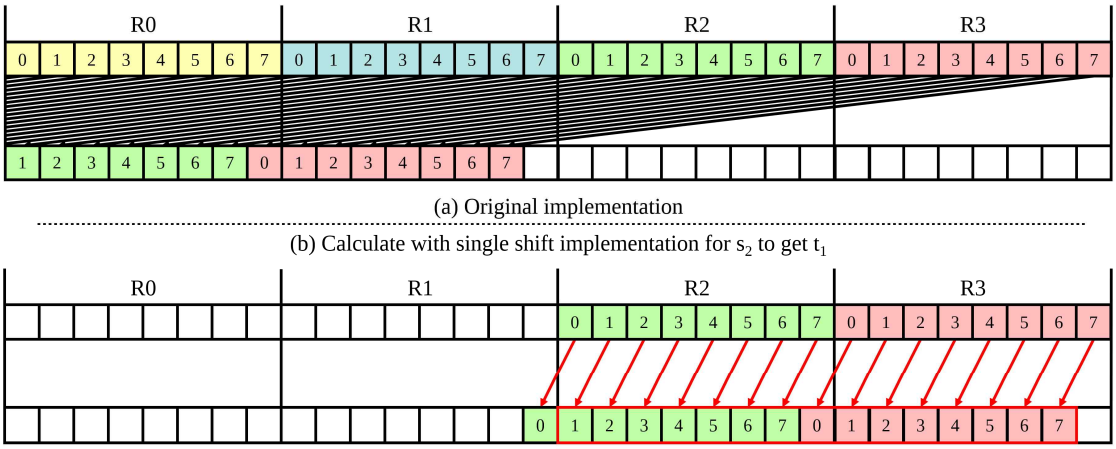
blocks, which are not involved in the generation of other temporary values. Thus, a separate implementation is performed to generate this value. The s_1 block undergoes a 15-bit right shift, leaving the upper 17 bits intact. By using a RBS implementation, 24 bits from s_1 can be loaded, and a single RBS will yield the required value, as illustrated in [Figure 3-10] (a) and (b). [Figure 3-10] (a) represents the implementation using the conventional method, while [Figure 3-10] (b) shows the implementation with the RBS applied. The section marked with the red line in [Figure 3-10] (b) demonstrates that the result matches the output of [Figure 3-10] (a).



[Figure 3-11] Operation process with the previous s_1 state block operation and RBS applied

For s_2, a 17-bit left shift is performed to retain the lower 15 bits. This can be implemented without Reverse Bitwise Shifting by loading only the lower 16 bits and shifting once to the left to obtain the desired value, as shown in [Figure 3-11] (a) and (b). In [Figure 3-11] (a), the operation is performed by shifting s2 15 times, whereas in [Figure 3-11] (b), the result is obtained by shifting in the same direction but only once. Although the number of shifts

differs, it can be observed that both yield identical computation results.



[Figure 3-12] Operate s_2 state block with single shift

By employing reverse bitwise shifts, the number of required shifts in the original TinyJAMBU can be drastically reduced, even more than the method of reusing accumulated values. The differences in shift counts between the various implementations are summarized in [Table

[Table 3-5] Number of shifts for each implementation

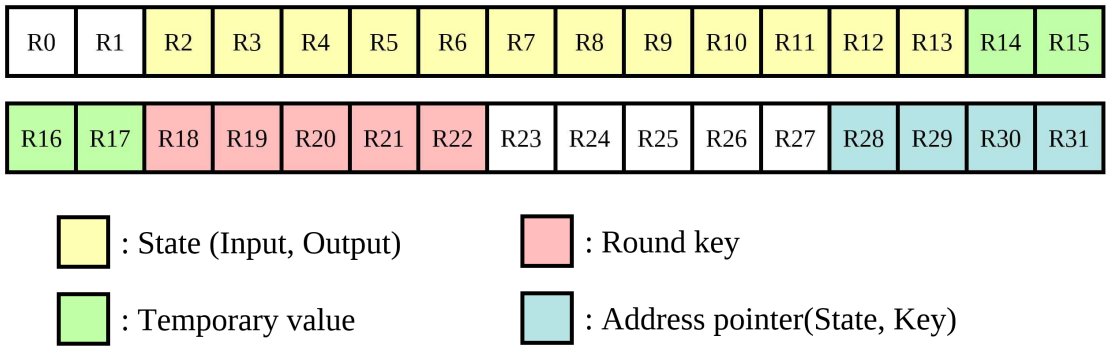
Case	Reference	Assembly	RBS
s_1>>15	15	15	6
s_2<<17	17	17	1
s_2>>6	6	6	2
s_2>>21	21	15	5
s_2>>27	27	6	2
s_3>>5	5	5	1

s_3>>11	11	6	1
s_3>>26	26	15	1

3-5].

3.2.2 Register Scheduling, Instructions Used and Implementation

To implement the proposed TinyJAMBU, a register allocation plan, as illustrated in [Figure 3-13], is devised.



[Figure 3-13] Register allocation plan for RBS TinyJAMBU

Registers R2, R3, R4, and R5 are assigned to store the computation result state variables, while registers R6 through R13 are allocated for storing the input state variables. Registers R14, R15, R16, and R17 are designated as temporary registers to hold intermediate values during the computation. Additionally, registers R18 through R22 are used for storing key values, and registers R28 through R31 are allocated for storing pointer addresses. The types of instructions used in the implementation can be found in [Table 3-6].

[Table 3-7] presents the source code implementation of a keyed permutation that incorporates reverse bitwise shifts. The code includes only the portions where t_1, t_2, t_3, and t_4 are generated during

[Table 3-6] List of instructions used in implementation for TinyJAMBU in alphabetical order

Mnemonic	Operands	Description	Operation
ADD	Rd, Rr	Add without Carry	Rd←Rd+Rr
CLR	Rd	Clear Register	Rd←Rd⊕Rd
COM	Rd	One's Comeplement	Rd←$FF−Rd
EOR	Rd, Rr	Exclusive OR	Rd←Rd⊕Rr
LDD	Rd, Y+q	Load Indirect with Displacement	Rd←(Y+q)
LDD	Rd, Z+q	Load Indirect with Displacement	Rd←(Z+q)
LSL	Rd	Logical Shift Left	Rd(n+1)←Rd(n) Rd(0)←0 C←Rd(7)
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd←Rr+1:Rr
OR	Rd, Rr	Logical OR	Rd←Rd v Rr
POP	Rd	Pop Register from Stack	Rd←STACK
PUSH	Rr	Push Register on Stack	STACK←Rr
ROL	Rd	Rotate Left Through Carry	Rd(n)←Rd(n+1) Rd(7)←0 C←Rd(0)
STD	Y+q, Rr	Store Indirect with Displacement	(Y+q)←Rr
STD	Z+q, Rr	Store Indirect with Displacement	(Z+q)←Rr

the overall keyed permutation process. Lines 1−8 cover the loading

of required values. In lines 9−12, the left shift operation for s1 is used to calculate the first part of t_1, replacing the original 15 right shifts with a single left shift. Lines 13−17 prepare for subsequent operations, a process that appears intermittently after shifting each state block.

Lines 18−19 handle the computation of the second part of t_1, where s_2 is shifted left once instead of the 17 left shifts in the original method. Lines 20−25 are further preparation steps for the next operations. From lines 26−49, the front part of t_2 is computed by shifting s_2 six times to the right, following the original procedure.

Lines 56−75 cover the calculation of the back part of t_4, where s_3 is shifted left five times. Since the initial shift of each state block remains unchanged, this step mirrors the original process. Lines 80−87 simplify the calculation of the back part of t_3 by replacing the original 11 left shifts with two right shifts.

Lines 91−92 compute the back part of t_2 from s_3, where the original 26 left shifts are replaced by a single right shift. Lines 94−96 compute the front part of t_3, shifting s_2 left once instead of the original 21 right shifts. Finally, in lines 101−104, the front part

[Table 3−7] Implementation code of proposed TinyJAMBU (RC: round counter, RK: round key, X00~X31: plaintext, XT: temporary register, Zero: zero register)

Line	Code	Comment
1:	LDD R2, Y+0	Load state
2:	LDD R3, Y+1	
3:	LDD R4, Y+2	
4:	LDD R5, Y+3	

```
5:    LDD  R6,   Y+5
6:    LDD  R7,   Y+6
7:    LDD  R8,   Y+7
8:    CLR  R9
9:    LSL  R6          Front t1: s1>>15 → s1<<1
10:   ROL  R7
11:   ROL  R8
12:   ROL  R9
13:   EOR  R2,   R7
14:   EOR  R3,   R8
15:   EOR  R4,   R9
16:   LDD  R6,   Y+8
17:   LDD  R7,   Y+9
18:   LSL  R6          Rear t1: s2<<17 → s2<<1
19:   ROL  R7
20:   EOR  R4,   R6
21:   EOR  R5,   R7
22:   LDD  R6,   Y+8
23:   LDD  R7,   Y+9
24:   LDD  R8,   Y+10
25:   LDD  R9,   Y+11
26:   LSR  R9          Front t2: s2>>6 (same as original)
27:   ROR  R8
28:   ROR  R7
29:   ROR  R6
30:   LSR  R9
31:   ROR  R8
32:   ROR  R7
```

33:	ROR R6	
34:	LSR R9	
35:	ROR R8	
36:	ROR R7	
37:	ROR R6	
38:	LSR R9	
39:	ROR R8	
40:	ROR R7	
41:	ROR R6	
42:	LSR R9	
43:	ROR R8	
44:	ROR R7	
45:	ROR R6	
46:	LSR R9	
47:	ROR R8	
48:	ROR R7	
49:	ROR R6	
50:	MOVW R14, R6	
51:	MOVW R16, R8	
52:	LDD R10, Y+12	
53:	LDD R11, Y+13	
54:	LDD R12, Y+14	
55:	LDD R13, Y+15	
56:	LSL R10	Rear t_4: s_3<<5 (same as original)
57:	ROL R11	
58:	ROL R12	
59:	ROL R13	
60:	LSL R10	

61:	ROL R11	
62:	ROL R12	
63:	ROL R13	
64:	LSL R10	
65:	ROL R11	
66:	ROL R12	
67:	ROL R13	
68:	LSL R10	
69:	ROL R11	
70:	ROL R12	
71:	ROL R13	
72:	LSL R10	
73:	ROL R11	
74:	ROL R12	
75:	ROL R13	
76:	PUSH R10	
77:	PUSH R11	
78:	PUSH R12	
79:	PUSH R13	
80:	LSR R13	Rear t_3: s_3<<11 → s_3>>2
81:	ROR R12	
82:	ROR R11	
83:	ROR R10	
84:	LSR R13	
85:	ROR R12	
86:	ROR R11	
87:	ROR R10	
88:	PUSH R10	

89:	PUSH	R11		
90:	PUSH	R12		
91:	LSR	R11		Rear t_2: s_3<<26 → s_3>>1
92:	ROR	R10		
93:	OR	R9,	R10	
94:	LSL	R15		Front t_3: s_2>>21 → s_2<<1
95:	ROL	R16		
96:	ROL	R17		
97:	POP	R12		
98:	POP	R11		
99:	POP	R10		
100:	OR	R10,	R17	
101:	LSL	R16		Front: t_4: s_2>>27 → s_2<<2
102:	ROL	R17		
103:	LSL	R16		
104:	ROL	R17		
105:	POP	R13		
106:	POP	R12		
107:	POP	R11		
108:	POP	R10		

of t_4 is calculated from s_2, reducing the original 27 left shifts to just two, enabled by the proposed RBS technique.

3.2.3 Alternative Implementation: Initialization Skip

During the initialization step of TinyJAMBU, the key and nonce are set. If the key has changed, this process must be repeated; however, in environments such as the Internet of Things (IoT), the key and nonce may not be updated frequently. In such cases, the previously

used key and nonce settings can be retained, eliminating the need to repeat the initialization process. Skipping this initialization step allows for faster computation, providing a simple yet effective method for increasing efficiency. Notably, the number of keyed permutations required for key and nonce settings during initialization is significantly higher compared to other stages, so bypassing this step can enhance computational efficiency.

3.3 Rainbow with Look-Up Table Based Multiplication

3.3.1 Tower-Field Based Multiplication

Rainbow signature uses the Karatsuba polynomial multiplication algorithm based on tower fields. While Rainbow signature operates over GF16, it transitions to the subfield GF4 for tower field operations, and GF4 further transitions to another subfield, GF2. This hierarchical process is illustrated in [Expression 3-1].

$$F_{16} := F_4[y]/(y^2+y+x), F_4 := F_2[y]/(x^2+x+1)$$

[Expression 1] Tower-field based multiplication of Rainbow I signature

The multiplication process for Rainbow signature can be represented in pseudocode, as shown in [Table 3-8]. In the first to fourth lines, two 4-bit input values are split into 2-bit segments. Lines five and six perform multiplications on the lower and upper 2-bit segments, respectively. Line seven involves XORing the upper and lower 2-bit segments derived from the same values, then multiplying them to obtain the intermediate value. Line eight generates the square of the upper 2-bit value, termed as "square." Lastly, the algorithm XORs the intermediate value with the result of

[Table 3-8] Pseudocode of tower-field based polynomial multiplication for Rainbow signature (^: bitwise XOR)

Input: 4-bit array A, 4-bit constant B	
Output: 4-bit accumulated calculation result C	
1:	$a_0 \leftarrow$ low 2-bit from A
2:	$a_1 \leftarrow$ high 2-bit from A
3:	$b_0 \leftarrow$ low 2-bit from B
4:	$b_1 \leftarrow$ high 2-bit from B
5:	$a_0b_0 \leftarrow a_0 \times b_0$
6:	$a_1b_1 \leftarrow a_1 \times b_1$
7:	intermediate $\leftarrow a_0 \hat{} a_1 \times b_0 \hat{} b_1$
8:	square $\leftarrow a_1b_1 \times a_1b_1$
9:	C $\leftarrow$ ((intermediate $\hat{}$ a_1b_1)<<2) $\hat{}$ $a_0b_0 \hat{}$ square
10:	return C

multiplying the upper 2-bit values, shifts this result by two bits to the upper segment, and then XORs it with the lower bit multiplication result and the square, completing the operation.

For the Rainbow signature multiplication, the initial value is decomposed down to the lowest subfield, GF2, where the computation takes place. Afterward, during the reconstruction of the result, modular reduction is applied. ARMv8 architecture includes PMUL and PMULL instructions that can perform polynomial multiplication in parallel. However, these instructions cannot be directly applied to the polynomial multiplication required for Rainbow signature. While PMUL and PMULL perform modular reduction, they can only do so for values with a minimum size of 8 bits. Due to the nature of tower field-based operations in Rainbow signature, these instructions cannot

handle the modular reduction of carry values generated in the subfields. To use PMUL and PMULL for Rainbow signature, additional custom code would be needed to handle the modular reduction in the subfields, which would reduce efficiency. Therefore, a different type of multiplication method is required.

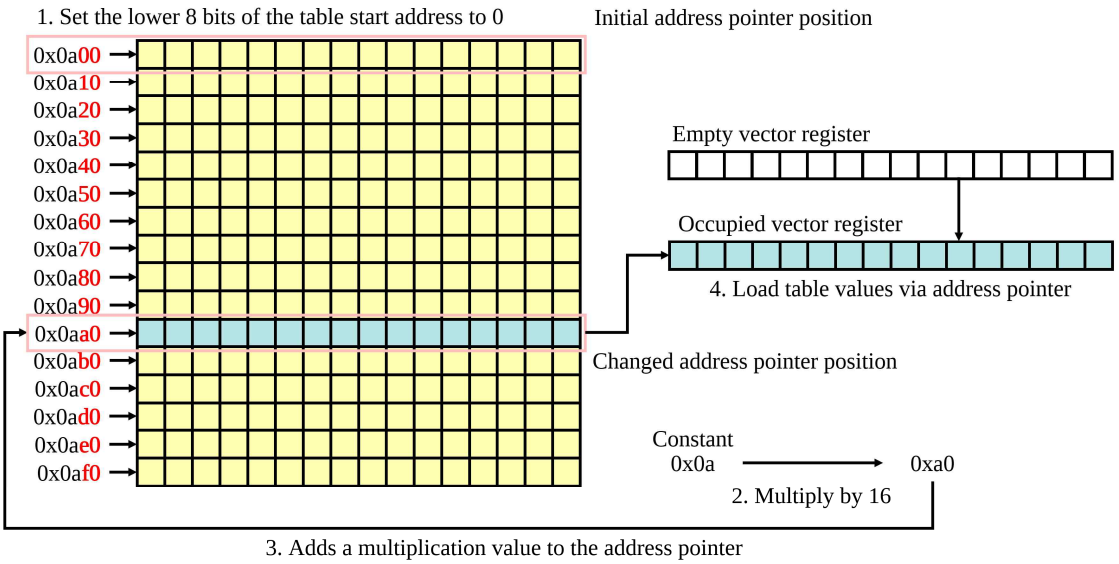
3.3.2 Look Up Table Based Multiplication in Rainbow I

In the Rainbow signature scheme, multiplication can be optimized using a look-up table (LUT). Since the Rainbow scheme involves performing multiple multiplications with a single value, loading a look-up table once can yield multiple results, making it well-suited for parallel implementation. The proposed LUT stores multiplication results for 4-bit values, as the Rainbow I scheme operates in GF(16), where operations are performed on 4-bit units. A 4-bit unit can represent 16 different values, and a 4-bit by 4-bit multiplication yields 256 possible results. Therefore, the LUT would require 1024 bits, or 128 bytes of storage. However, given that variables are stored in 8-bit units, the actual size of the LUT becomes 256 bytes. The full contents of the LUT are detailed in the Appendix's [Table Appendix-1].

During the multiplication process, a constant value is multiplied with a variable array. This means that not all values from the LUT need to be loaded; only the specific 16-byte segment corresponding to the constant value is required. While the LUT could be loaded using conditional statements, this introduces variability in load speeds depending on the constant value. Thus, an alternative method is employed to avoid such discrepancies.

[Figure 3-14] illustrates the process of loading the table. To implement the proposed method, the lower 8 bits of the starting

address of the LUT are set to 0x00. Each row in the table contains 16 bytes, so if the starting address is 0x00, the subsequent rows increment by 0x10, 0x20, and so on. A pointer variable is initialized to point to the starting address of the LUT. When a constant value is provided for multiplication, the pointer is adjusted by adding 16 times the constant value. For instance, if the constant is 0x03, 0x30 is added to the pointer. The pointer then references the 16-byte row in the LUT corresponding to the constant, completing the table load needed for the multiplication operation.



[Figure 3-14] Table loading process in proposed Rainbow signature

This process can be represented in pseudocode as shown in [Table 3-9]. In the first four lines, the table is set up. From lines 5 to 13, the multiplication operations based on the table values are performed. Line 14 returns the result.

3.3.3 Resolve of LUT Size Problem in Rainbow III and V

In Rainbow I, computations were performed over GF16, but

[Table 3-9] Pseudocode of look-up table based polynomial multiplication for Rainbow I (<<n: bitwise left shift n times, >>n: bitwise right shift n times, &: bitwise AND, ^: bitwise XOR)

Input: 8-bit(4-bit||4-bit) operand array A, 4-bit constant C	
Output: 8-bit(4-bit||4-bit) accumulated result R	
1:	Table address pointer P initialized to first address of LUT
2:	$C \leftarrow C \times 16$
3:	$P \leftarrow P + C$
4:	Table[16] ← Load table values via P
5:	Loop counter $LC \leftarrow \|A\|/16$
6:	**for** i from 0 until to LC **do**
7:	j ← i×16
8:	A_{low}[j:j+15] ← A_{low}[j:j+15]&0x0f
9:	A_{high}[j:j+15] ← A_{high}[j:j+15>>4
10:	A_{low}[j:j+15] ← Table[A_{low}[j:j+15]]
11:	A_{high}[j:j+15] ← Table[A_{high}[j:j+15]]
12:	R[j:j+15] ← R[j:j+15]^(A_{low}[j:j+15]&(A_{high}[j:j+15]<<4))
13:	**end for**
14:	return R

Rainbow III and V perform calculations over GF256. Since GF256 is represented by 8 bits and can express 256 values, the results of 8-bit multiplication yield 65,536 possible values, requiring a lookup table size of 65,536 bytes. This poses a problem not only due to the table size increasing by 256 times from the previous 256 bytes but also because a single lookup now requires loading 256 bytes. Since vector registers can only hold 16 bytes, loading a 256-byte lookup

table would require 16 vector registers. This stands in stark contrast to the previous design, where only one vector register was needed.

Considering that Rainbow III and V also use tower-field-based multiplication, it is possible to use the original 4-bit multiplication table instead of an 8-bit multiplication table. However, an additional 16-byte table for precomputed intermediate values is required. Thus, the total size for the Rainbow signature becomes 256 bytes + 16 bytes, resulting in a total of 272 bytes.

Implementing multiplication with an 8-bit lookup table simplifies the entire computation, except for the table loading process, but it comes with the drawback of requiring a large number of vector registers and an excessively large table size. On the other hand, implementing 8-bit multiplication by breaking it down into 4-bit units reduces the required vector register allocation and decreases the table size by approximately 99.58%. However, the computation process becomes slightly more complex, and when expressed in pseudocode, it takes the form shown in [Table 3-10]. In lines 1-13, the necessary table is called, preparing for the calculations and setting the number of iterations. From lines 14-31, the required values are sequentially fetched, and the table-based computation is performed. Since the operands are expressed in 8-bit form, they are first divided into 4-bit segments before calculation. Notably, there is an additional step where a 16-byte table is loaded. If the table loading is included within the loop, it could slow down the computation, as the table would be reloaded in each iteration. However, in the actual implementation, the loop is not applied, and this approach is used only in the pseudocode. The result of the computation is returned in line 32.

[Table 3-10] Pseudocode of look-up table based polynomial multiplication for Rainbow III and V(<<n: bitwise left shift n times, >>n: bitwise right shift n times, &: bitwise AND, ^: bitwise XOR, $_A$: additional)

Input: 8-bit operand array A, 8-bit constant C	
Output: 8-bit accumulated result R	
1:	Table address pointer P initialized to first address of LUT
2:	$C_{low} \leftarrow C$&0x0f
3:	$C_{low} \leftarrow C_{low} \times 16$
4:	$P \leftarrow P + C_{low}$
5:	$Table_{low}$[16] ← Load table values via P
6:	Table address pointer P initialized to first address of LUT
7:	$C_{high} \leftarrow C>>4$
8:	$C_{high} \leftarrow C_{high} \times 16$
9:	$P \leftarrow P + C_{high}$
10:	$Table_{high}$[16] ← Load table values via P
11:	Table address pointer P initialized to first address of LUT_A
12:	$Table_A$[16] ← Load table values via P
13:	Loop counter $LC \leftarrow \|A\|/16$
14:	**for** i from 0 until to LC **do**
15:	j ← i×16
16:	A_{low}[j:j+15] ← A_{low}[j:j+15]&0x0f
17:	A_{high}[j:j+15] ← A_{high}[j:j+15]>>4
18:	A_{middle}[j:j+15] ← A_{low}[j:j+15]^A_{high}[j:j+15]
19:	A_{low}[j:j+15] ← Table[A_{low}[j:j+15]]

20: A_{high}[j:j+15] ← Table[A_{high}[j:j+15]]

21: C_{low} ← C&0x0f

22: C_{high} ← C>>4

23: C_{middle} ← C_{low} ^ C_{high}

24: Table address pointer P initialized to first address of LUT

25: P ← P+C_{middle}

26: $Table_{middle}$[16] ← Load table values via P

27: A_{middle}[j:j+15] ← $Table_{middle}$[A_{middle}[j:j+15]]

28: A_{low}[j:j+15] ← A_{low}[j:j+15] ^ A_{middle}[j:j+15]

29: A_{high}[j:j+15] ← $Table_A$[A_{high}[j:j+15]]

30: R[j:j+15] ← R[j:j+15] ^ (A_{low}[j:j+15]&(A_{high}[j:j+15]<<4))

31: **end for**

32: return R

3.3.4 Register Scheduling and Instructions Used

The instructions in [Table 3-11] are the commands used to implement the Rainbow signature. Since the ARMv8 processor supports both general instructions and vector instructions, the vector

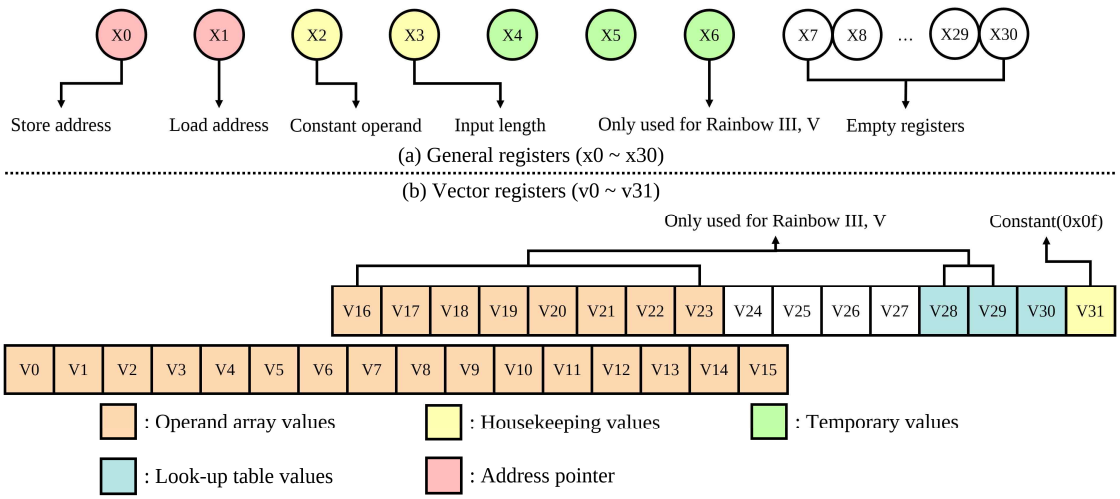
[Table 3-11] List of instructions used to implement Rainbow signatures in alphabetical order

Mnemonic	Operands	Description	Operation
ADD	Xd, Xn, #imm	Add registers immediate	Xd←Xn+#imm
ADR	Xd, (Label)	Form PC-relative address	Xd←address

B	(Label)	Branch	Go to label
BEQ	(Label)	Branch if it is equal	Go to label
CBNZ	Xt, (Label)	Compare and branch on nonzero	Go to label
CMP	Xd, #imm	Compare	Flags←result
LSL	Xd, Xn, #shift	Logical shift left immeidate	Xd←Xn<<#shift
MOV	Xd, #imm	Move immediate	Xd←#imm
RET	{Xn}	Return from subroutine	Return
SUB	Xd, Xn, #imm	Subtract immediate	Xd←Xn−#imm
AND	Vd.T, Vn.T, Vm.T	Bitwise AND	Vd←Vn&Vm
EOR	Vd.T, Vn.T, Vm.T	Bitwise XOR	Vd←Vn⊕Vm
LD1	Vt.T, [Xn]	Load multiple single−element structures	Vt←[Xn]
MOVI	Vt.T, #imm	Move immediate	Vt←#imm
SHL	Vd.T, Vn.T, #shift	Shift left immediate	Vd←Vn<<#shift
ST1	Vt.T, [Xn]	Store multiple single−element structures	[Xn]←Vt
TBL	Vd.T, {Vn.16B},	Table vector	Vd←Vn[Vm]

	Vm.T	Lookup	
USHR	Vd.T, Vn.T, #shift	Unsigned shift right immediate	Vd←Vn>>#shift

instructions that utilize vector registers are listed separately. [Figure 3-] shows the register allocation plan for the proposed implementation. As with the instructions, the general registers and vector registers are separated and represented distinctly. In the general registers, x0 and x1 are used to store address values, while x2 holds operand constants, and x3 records the size of the input array. The x4 and x5 registers are used for temporary variables. In the vector registers, v0-v15 store the operand arrays, and v16-v23 are additionally used for storing operand arrays in Rainbow III and V. Registers v28 and v29 store additional lookup table values, while v30 is used for the general lookup table. The v31 register holds the constant value 0x0f for specific AND operations.



[Figure 3-15] Register allocation plan for Look-up table based Rainbow signature

The actual implementation code for Rainbow I is shown in [Table 3-12], and it operates in the same way as the pseudocode. The multiplier design operates on 4-bit units, but the actual computation is carried out in 8-bit units, with the minimum size of a vector register also being 8 bits. Consequently, although the vector register is 128 bits in size, it can fetch 16 values in parallel from the lookup

[Table 3-12] Implementation code of proposed multiplication (x0: output address, x1: operand address, x2(w2): constant)

Line	Code				Comment
1:	MOVI	v31.16b,	#15		
2:	ADR	x4,	MUL_TABLE		Initial address
3:	LSL	w2,	w2,	#4	Multiplied by 16
4:	ADD	x4,	x4,	x2	Get table address
5:	LD1.16b	{v30},	[x4]		Load table values
6:	LD1.16b	{v1},	[x1]		
7:	AND.16b	v0,	v1,	v31	Divide into 4-bit
8:	USHR.16b	v1,	v1,	#4	
9:	TBL.16b	v0,	{v30},	v0	Table look-up
10:	TBL.16b	v1,	{v30},	v1	
11:	SHL.16b	v1,	v1,	#4	
12:	EOR.16b	v0,	v0,	v1	
13:	LD1.16b	{v1},	[x0]		
14:	EOR.16b	v1,	v1,	v0	
15:	ST1.16b	{v1},	[x0]		Return

table at once. The implementation source code for Rainbow III and V can be found in [Table Appendix-2]. Rainbow III and V perform multiplication over GF256, making their source code slightly longer and more complex compared to Rainbow I.

3.3.5 Alternative Implementation: Avoiding Cache Side Attack

The proposed optimized implementation of the Rainbow signature has a potential vulnerability where timing information could be leaked during the lookup table access process. To mitigate this risk, an additional timing attack-resistant implementation is presented. This implementation introduces two approaches, the first being a cache side attack-resistant method.

This approach is inspired by the characteristics of the M1 processor, which has a 128-byte cache line size, the 256-byte size of the full lookup table, and the aligned memory addresses of the table. In the proposed method, when 16 bytes of a specific table are loaded, the M1 processor's cache stores an additional 128 bytes from the adjacent table in the cache line. By proactively loading the remaining 128 bytes into the cache, cache hits are always ensured, effectively obfuscating cache timing information. The implementation is structured as shown in [Table 3-13].

[Table 3-13] Implementation code of cache side attack resistance implementation. (x2(w2): constant)

Line	Code				Comment
1:	ADR	x4,	MUL_TABLE		Initial address
2:	LSL	w2,	w2,	#4	Multiplied by 16
3:	ADD	x4,	x4,	x2	Get table address
4:	LD1.16b	{v30},	[x4]		Load table values
5:	SUB	x4,	x4,	x2	Address recovery
6:	ROR	w2,	w2,	#4	
7:	XOR	w2,	w2,	#80	Offset move

8:	LSL	w2,	w2,	#4	
9:	ADD	x4,	x4,	x2	Get other address
10:	LD1.16b	{v27},	[x4]		Load other table

In lines 1-4, a 16-byte table is called, following the same process as in the original [Table 3-12]. During this step, the 128 bytes adjacent to the currently called lookup table are also loaded into the cache. In lines 5-6, the table is reset to its initial address. Line 7 modifies the offset value used to specify the table by XORing it with 0x80, ensuring that the offset always points to the opposite 128-byte segment, regardless of the original offset. Lines 8-10 call the new lookup table, but the loaded data are not used directly. Instead, this step ensures that the remaining 128 bytes are stored in the cache.

Through this process, the entire lookup table is stored in the cache, ensuring that all accesses result in cache hits, thereby eliminating timing information leaks related to cache access.

Another implementation is the constant-time version. The process of loading table values typically involves conditional statements, such as if-else constructs, which can cause variations in execution time depending on the condition. If execution time depends on secret information, it introduces the risk of timing attacks, where secret values can be inferred from timing variations. The constant-time implementation ensures that all operations are executed in a uniform amount of time. In the proposed method, the constant-time implementation guarantees identical execution times when loading values into registers.

To achieve this, the entire 256-byte lookup table is preloaded into

registers. According to [Figure 3-15], when implementing Rainbow I, registers v16 to v29 are unused, while v30 is used to store lookup table values and v31 is used for the constant value required to split data into 4-bit segments. These 16 registers, capable of holding 256 bytes in total, are used to preload all table values.

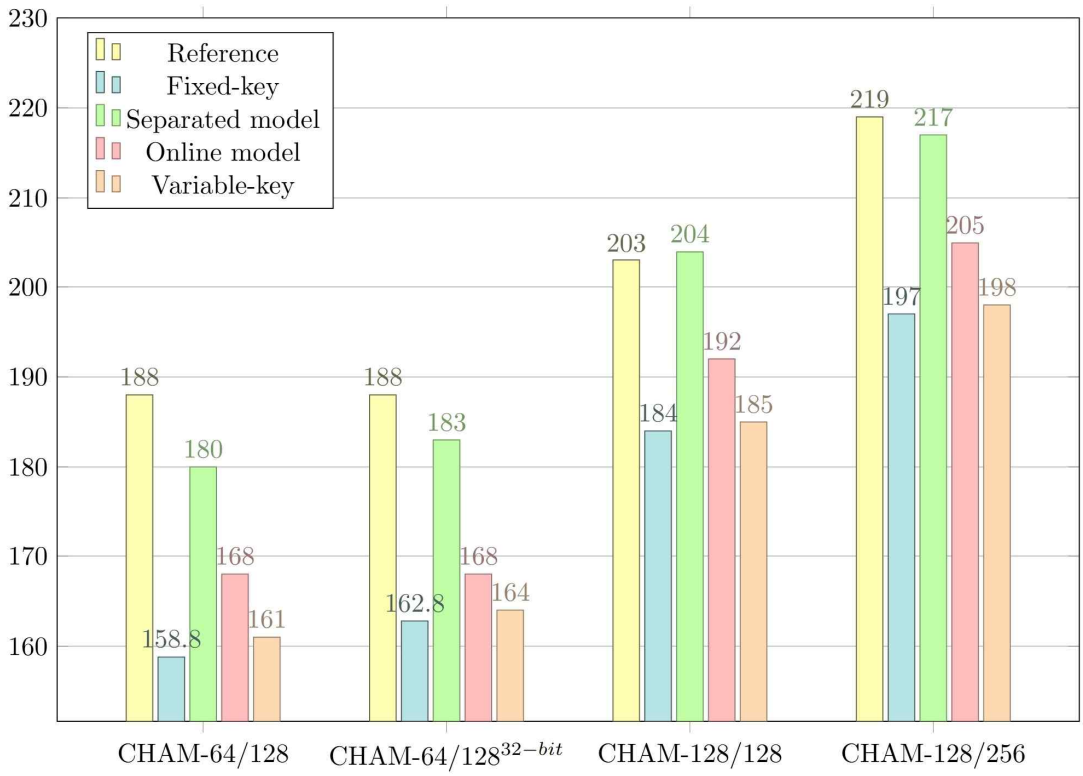
The offset value determining the table load ranges from 0 to 15. The implementation divides the table into two branches based on the midpoint value of 8. In the second stage, each branch is further divided into two sub-branches using midpoint values of 4 and 12. This process continues iteratively until the final branch is reached, at which point the value in the corresponding register is moved to the actual usage register, v30.

To ensure uniform execution time across all paths, dummy branch instructions are inserted in cases where the condition is not met. This ensures that every branch has the same number of instructions, maintaining consistent execution times regardless of the input. The complete implementation can be found in [Table Appendix-3].

4. Performance Evaluation

4.1 Evaluation CHAM Block Cipher

The performance evaluation of the optimized CHAM block cipher implementation is conducted on the AVR processor, specifically the ATmega128, using the Microchip Studio IDE. The performance metric used is cycles per byte (cpb), with an overall performance summary provided in [Figure 4-1].



[Figure 4-1] Performance Measurement Results for CHAM (Unit: clock cycles per byte, 32-bit: 32-bit counter of CHAM-64/128)

The original CHAM algorithm yields 188, 203, and 219 cpb for CHAM-64/128, 128/128, and 128/256, respectively. The proposed

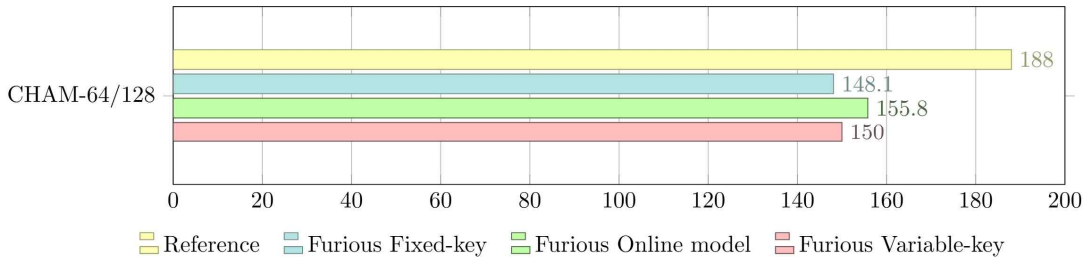
CHAM optimization techniques include various versions, starting with the fixed-key scenario implementation.

In the fixed-key scenario, the implementations achieve 158.8, 162.8, 184, and 197 cpb for CHAM-64/128 (16-bit counter), 64/128 (32-bit counter), 128/128, and 128/256, respectively. This corresponds to performance improvements of 18.4%, 15.5%, 10.3%, and 11.2%. The proposed methods optimize operations by omitting a significant number of computations in the initial 8 rounds and employing techniques like logical block rotation to maintain efficient processing in subsequent rounds.

For the variable-key scenario, a look-up table must be generated, and two models are provided: the separated model and the online model. The separated model is less efficient, as it effectively encrypts the first plaintext block twice. In contrast, the online model avoids redundant encryption and achieves performance improvements of 11.9%, 11.9%, 5.7%, and 6.8% for CHAM-64/128 (16-bit counter), 64/128 (32-bit counter), 128/128, and 128/256, respectively. In the actual variable-key scenario, after generating the look-up table, the encryption proceeds using the precomputed table, resulting in performance gains of 16.8%, 14.6%, 9.7%, and 10.6% compared to the original implementation.

[Figure 4-cf] shows the performance measurement results for Furious CHAM. Due to the characteristics of the implementation environment, only CHAM-64/128 was implemented, and the 32-bit counter was not considered. The performance results indicate that the fixed-key scenario implementation achieves 148.1 cpb. In the variable-key scenario, the process of generating the look-up table and performing encryption takes 155.8 cpb, while the encryption process alone in the variable-key scenario requires 150 cpb. These

results represent performance improvements of 26.9%, 20.7%, and 25.3%, respectively, compared to the original implementation.

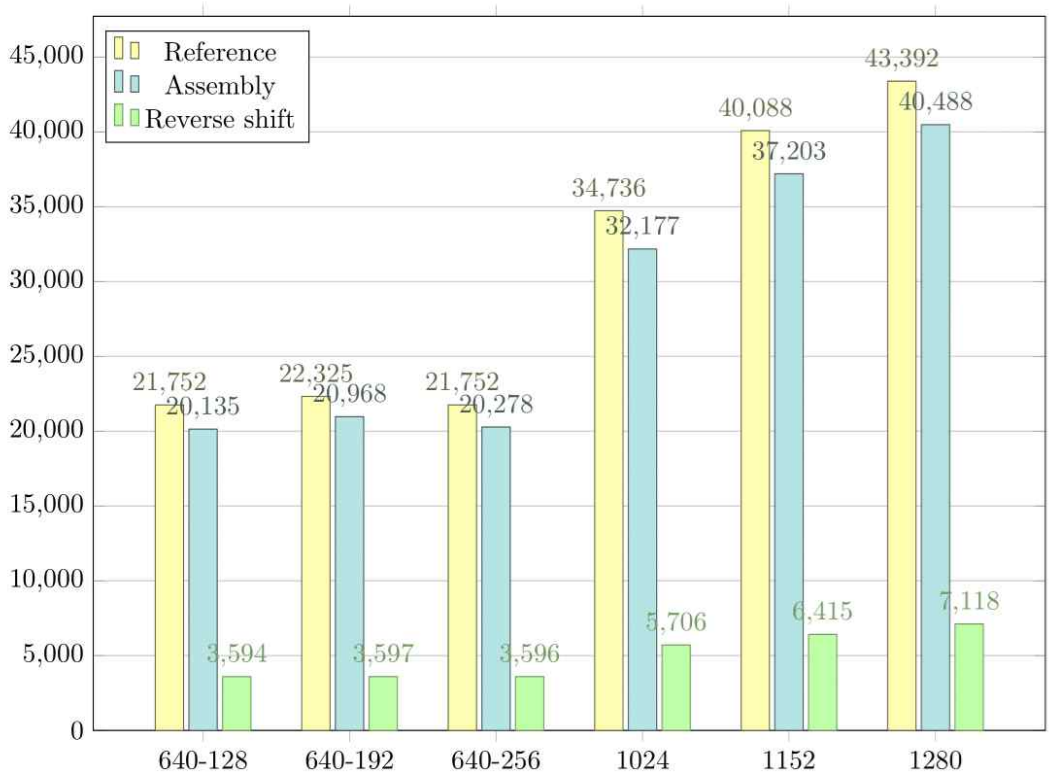


[Figure 4-2] Performance Measurement Results for Furious CHAM-64/128 (Unit: clock cycles per byte)

4.2 Evaluation TinyJAMBU Lightweight Cipher

The performance evaluation of the TinyJAMBU implementation was conducted using the Microchip Studio IDE on the ATmega128 processor. Clock cycles were used as the performance metric, and an 8-byte plaintext input was selected for the tests. The performance of TinyJAMBU's keyed permutation was compared across three implementations: the original TinyJAMBU reference, an assembly-optimized version, and the proposed implementation that incorporates RBS optimization.

Performance measurements were carried out for 640 keyed permutations with various key lengths, and for 1024, 1152, and 1280 keyed permutations, which correspond to key lengths of 128-bit, 192-bit, and 256-bit, respectively. The results are summarized in the graph in [Figure 4-3].



[Figure 4-3] Performance Measurement Results for Keyed permutation of TinyJAMBU (Unit: clock cycles)

For the reference implementation, the cycles recorded for 640/128, 640/192, 640/256, 1024, 1152, and 1280 keyed permutations were 21,752, 22,325, 21,752, 34,736, 40,088, and 43,392, respectively. The 640 keyed permutation showed little variation in performance across different key lengths because the number of repetitions remained constant. However, as the number of keyed permutations increased like 1024, 1152, 1280, the cycle count also increased.

The assembly-optimized version recorded 20,135, 20,968, 20,278, 32,177, 37,203, and 40,488 cycles, respectively, representing performance improvements of 8.0%, 6.5%, 7.3%, 8.0%, 7.8%, and 7.2% compared to the reference implementation. Although no special optimization techniques were applied, the assembly version showed

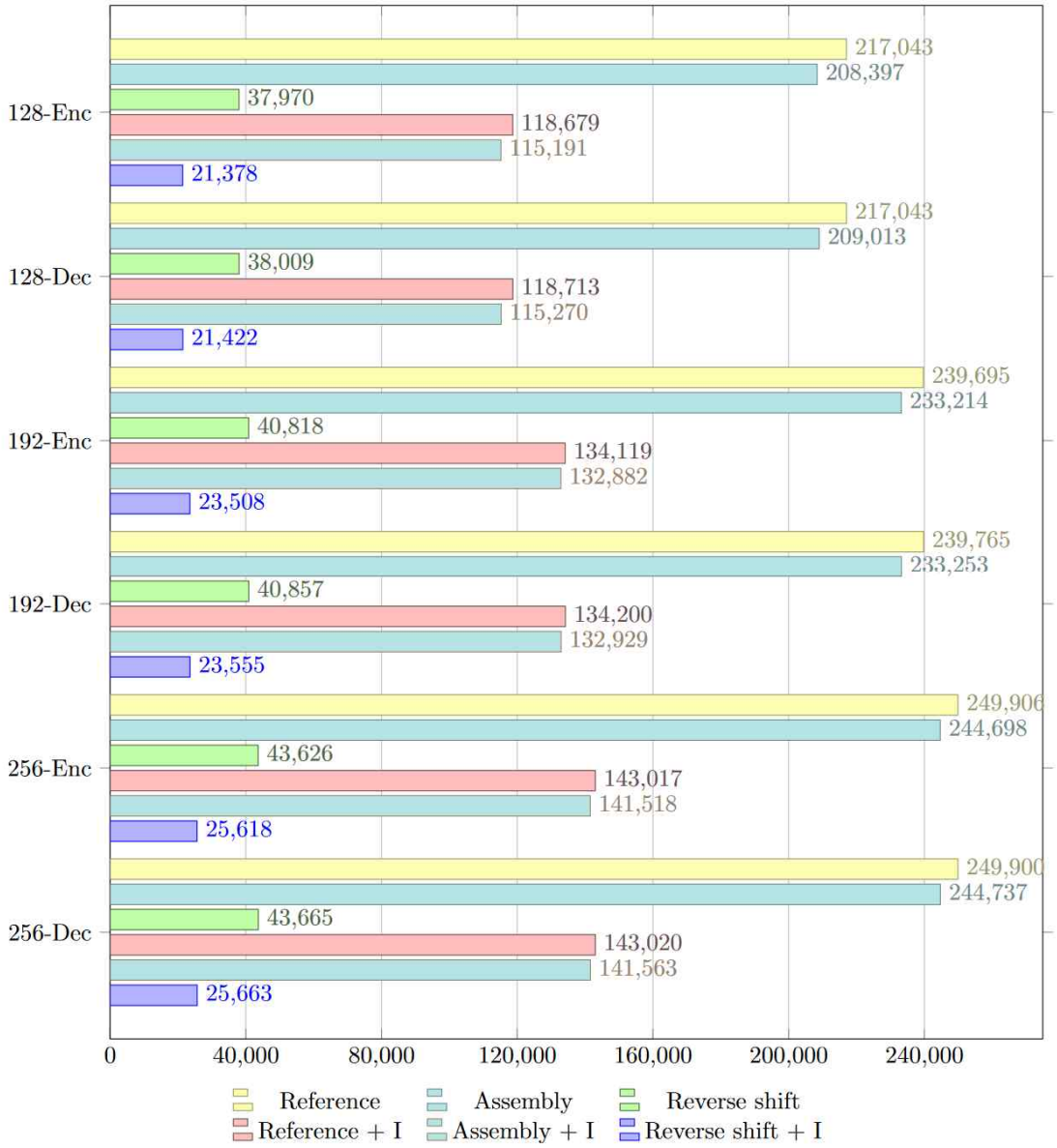
slightly better performance than the reference C implementation.

The RBS optimization method recorded 3,594, 3,597, 3,596, 5,706, 6,415, and 7,118 cycles, achieving significant performance improvements of 505.2%, 520.7%, 504.9%, 508.8%, 524.9%, and 509.6%, respectively, compared to the reference implementation. This remarkable performance enhancement is due to the effective use of RBS and optimization techniques tailored to the AVR processor, which significantly improved the efficiency of the keyed permutation.

The performance evaluation of the encryption and decryption processes in TinyJAMBU, each employing different keyed permutations, is presented. The results are summarized in the graph in [Figure 4-4]. An analysis of the encryption and decryption performance reveals that there is minimal difference between the two, as TinyJAMBU employs the same structure for both operations. For ease of comparison, a detailed analysis is performed on the encryption process only.

For the reference implementation, the cycle counts for key lengths of 128-bit, 192-bit, and 256-bit are 217,043, 239,765, and 249,900, respectively. In contrast, the assembly implementation records 208,397, 233,214, and 244,698 cycles for the respective key lengths, showing performance improvements of 4.1%, 2.8%, and 2.1% over the reference implementation. The implementation using the RBS technique, however, achieves cycle counts of 37,970, 40,818, and 43,626 for the respective key lengths. Compared to the reference implementation, these results represent performance enhancements of 471.6%, 487.2%, and 472.8%.

The RBS technique not only reduces the number of shifts but also decreases the number of registers involved in shift



[Figure 4-4] Performance Measurement Results for TinyJAMBU
(Unit: clock cycles, I: Initialization skip implementation)

operations, thereby significantly reducing the number of instructions required, making it even more efficient than the proposed structure.

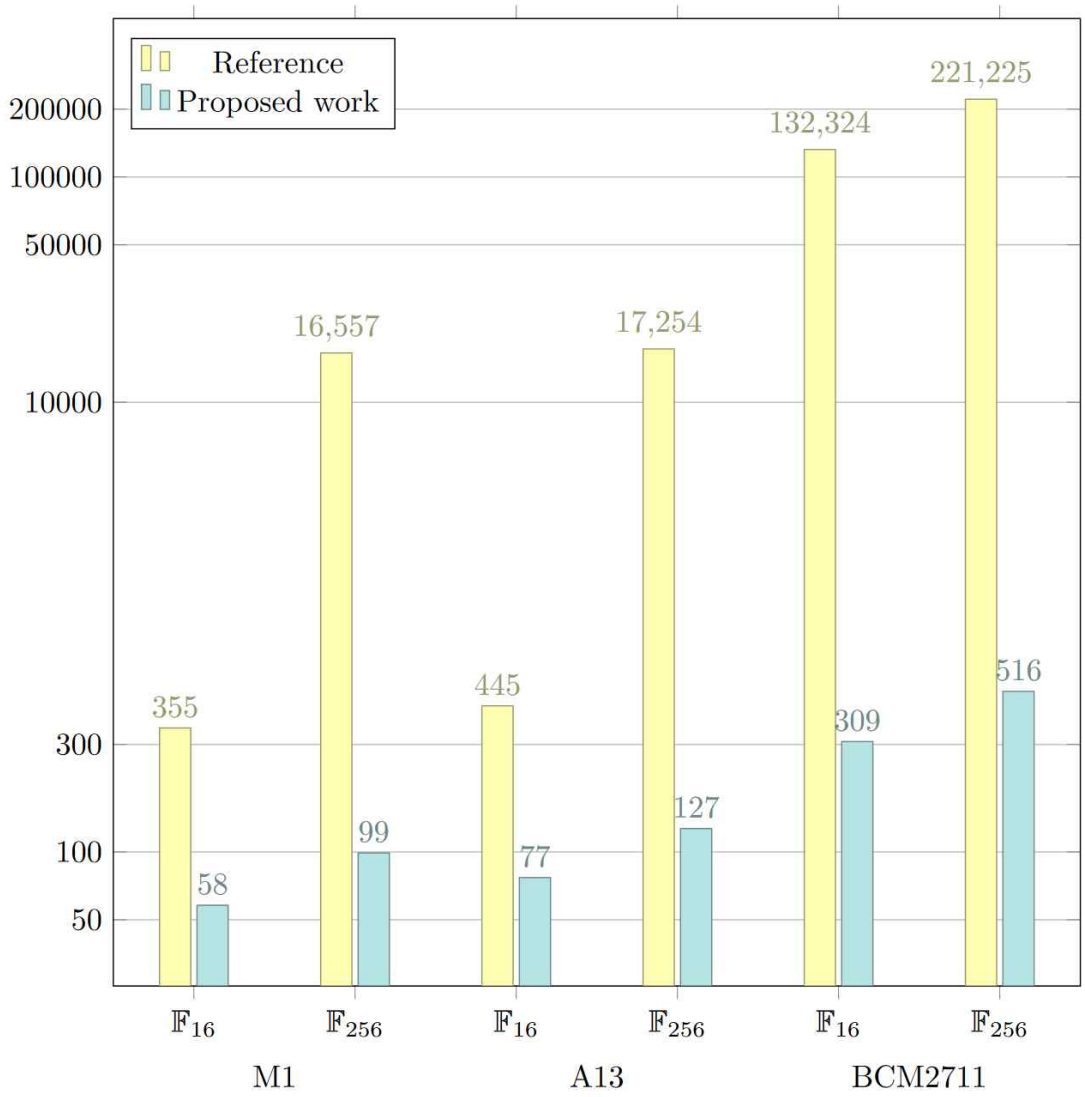
The performance of an additional implementation incorporating the initialization-skipping technique is also compared. This technique, which is beneficial in environments where the key and nonce are

reused, can be applied universally to all implementation types. When the initialization-skipping technique is used, performance improvements of up to 82.9% for the reference implementation, 80.9% for the assembly-optimized implementation, and 77.6% for the RBS implementation are observed. The combined effect of the RBS and initialization-skipping techniques yields up to a 915.3% performance improvement over the reference implementation.

4.3 Evaluation Rainbow Post-Quantum Cryptography

In this section, the performance evaluation of the Rainbow signature algorithm is presented. The implementation was worked using Xcode IDE. The first focus is on assessing the performance of the proposed multiplier. The evaluation results are shown in the graph in [Figure 4-5]. For the reference implementation, the F16 and F256 multipliers required 355 and 16,557 cycles, respectively. In contrast, the proposed method required only 58 and 99 cycles, corresponding to performance improvements of 512.1% and 16,624.2%, respectively. Similarly, on the A13 processor, the F16 and F256 multipliers demonstrated performance enhancements of 477.9% and 13,485.8%, respectively. On the BCM2711, the improvements were even more significant, with performance gains of 42,723.3% and 42,773.1% for the F16 and F256 multipliers, respectively.

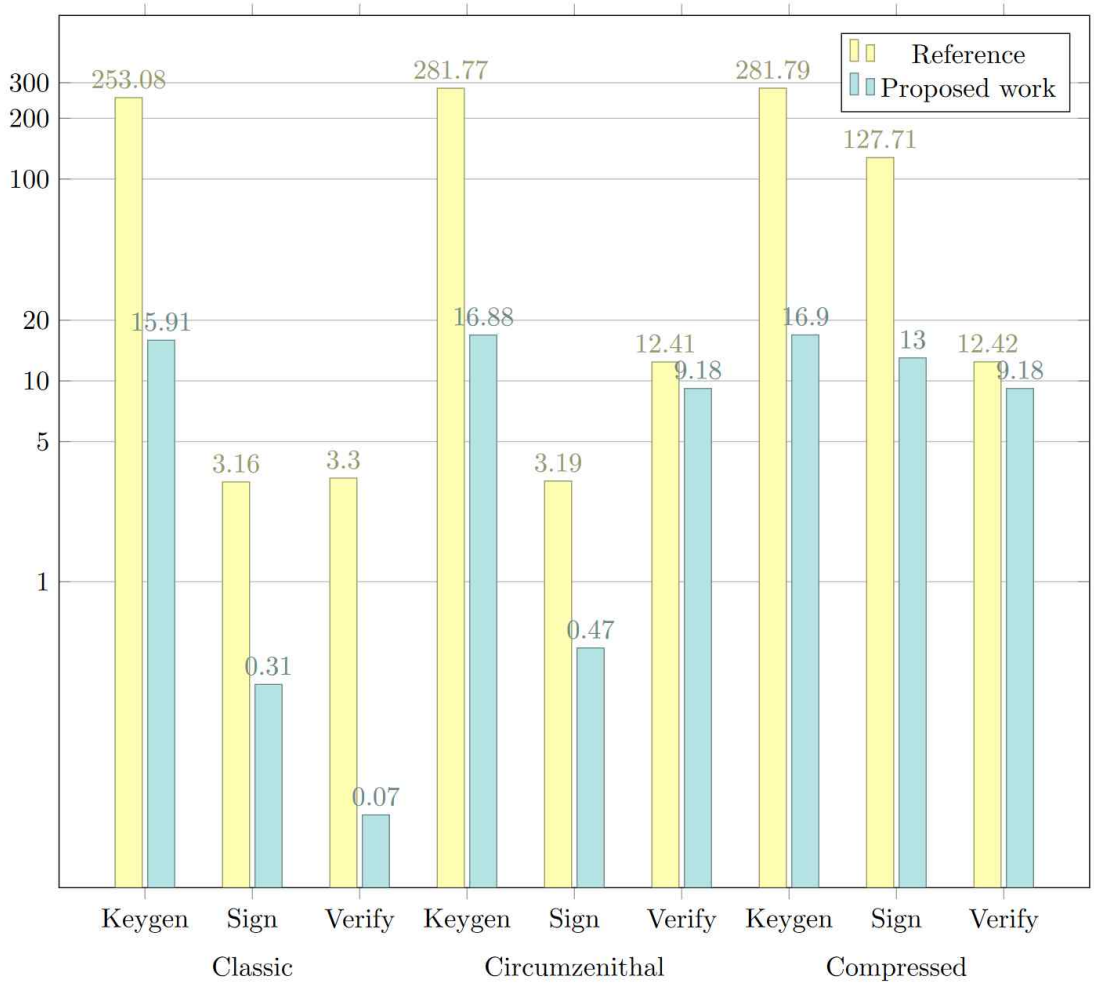
The Rainbow algorithm has several variant implementations, and performance measurements have been conducted on different processors. Here, the performance evaluation focuses on the Apple M1 processor, while data for other processors are provided in the Appendix.



[Figure 4-5] Performance Measurement Results for table based multiplier of proposed Rainbow signature in log scale (Unit: clock cycles)

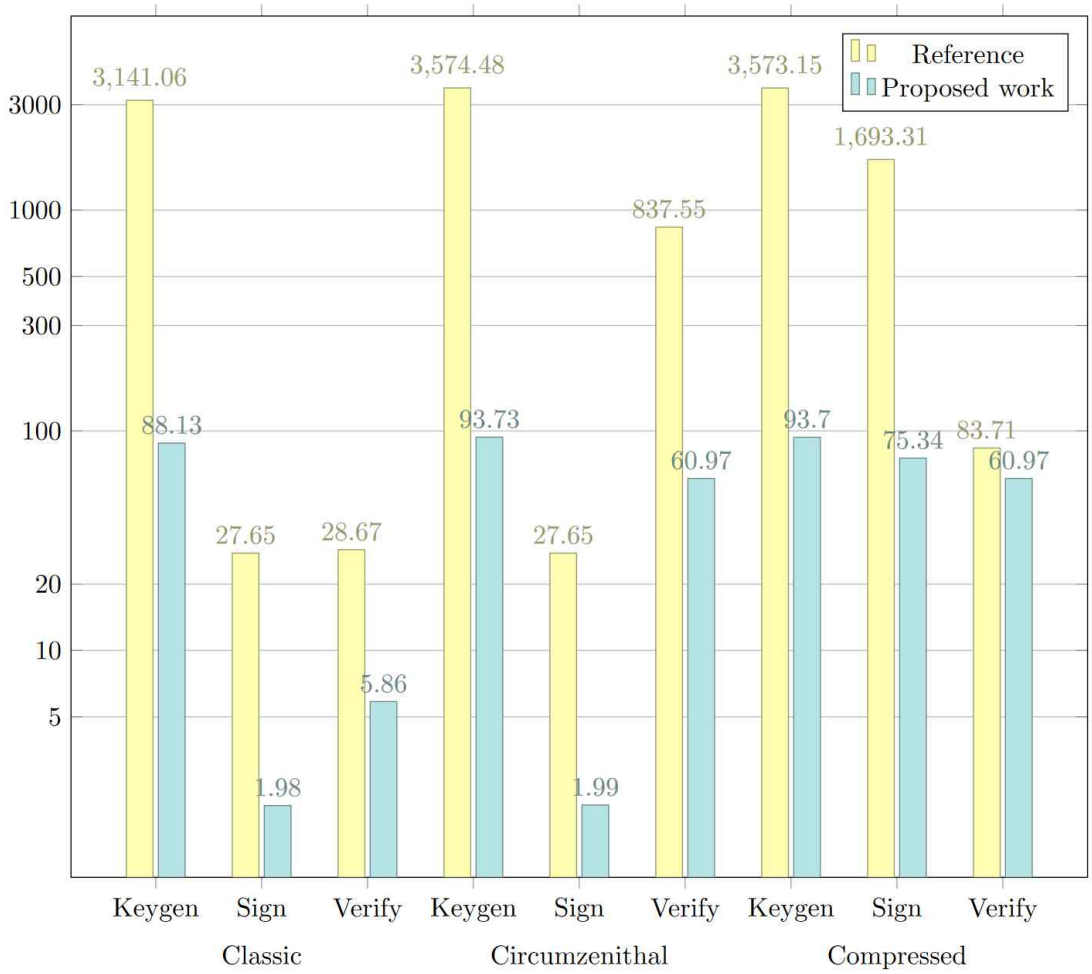
The performance results for Rainbow I are presented in [Figure 4-6]. When comparing the reference implementation of Rainbow I Classic to the proposed method, the improvements for key generation, signing, and verification processes were 1490.7%, 919.4%, and 4614.3%, respectively. Similarly, for the Rainbow I Circumzenithal version, performance improvements of 1569.3%, 578.7%, and 35.2%

were observed. The Rainbow I Compressed version showed enhancements of 1567.4%, 882.4%, and 35.3% for keygen, sign, and verify, respectively.



[Figure 4-6] Performance Measurement Results for Rainbow I on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles)

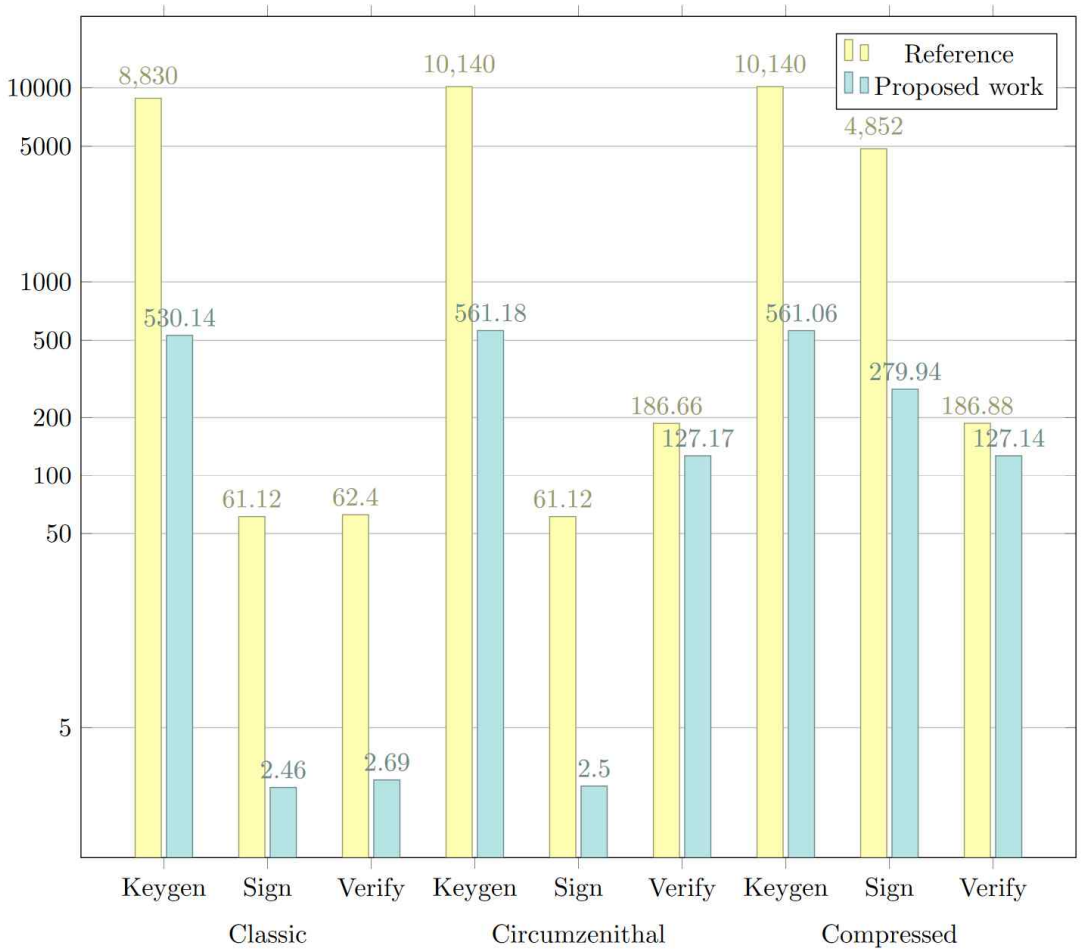
Next, the performance evaluation results for Rainbow III on the M1 processor are shown in [Figure 4-7]. The Rainbow III Classic version demonstrated performance improvements of 3464.1%, 1296.5%, and 389.2% for key generation, signing, and verification, respectively. The Rainbow III Circumzenithal version yielded



[Figure 4-7] Performance Measurement Results for Rainbow III on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles)

improvements of 3713.6%, 1289.4%, and 1273.7%, while the Rainbow III Compressed version achieved 3713.4%, 2147.6%, and 37.3% enhancements in the same processes.

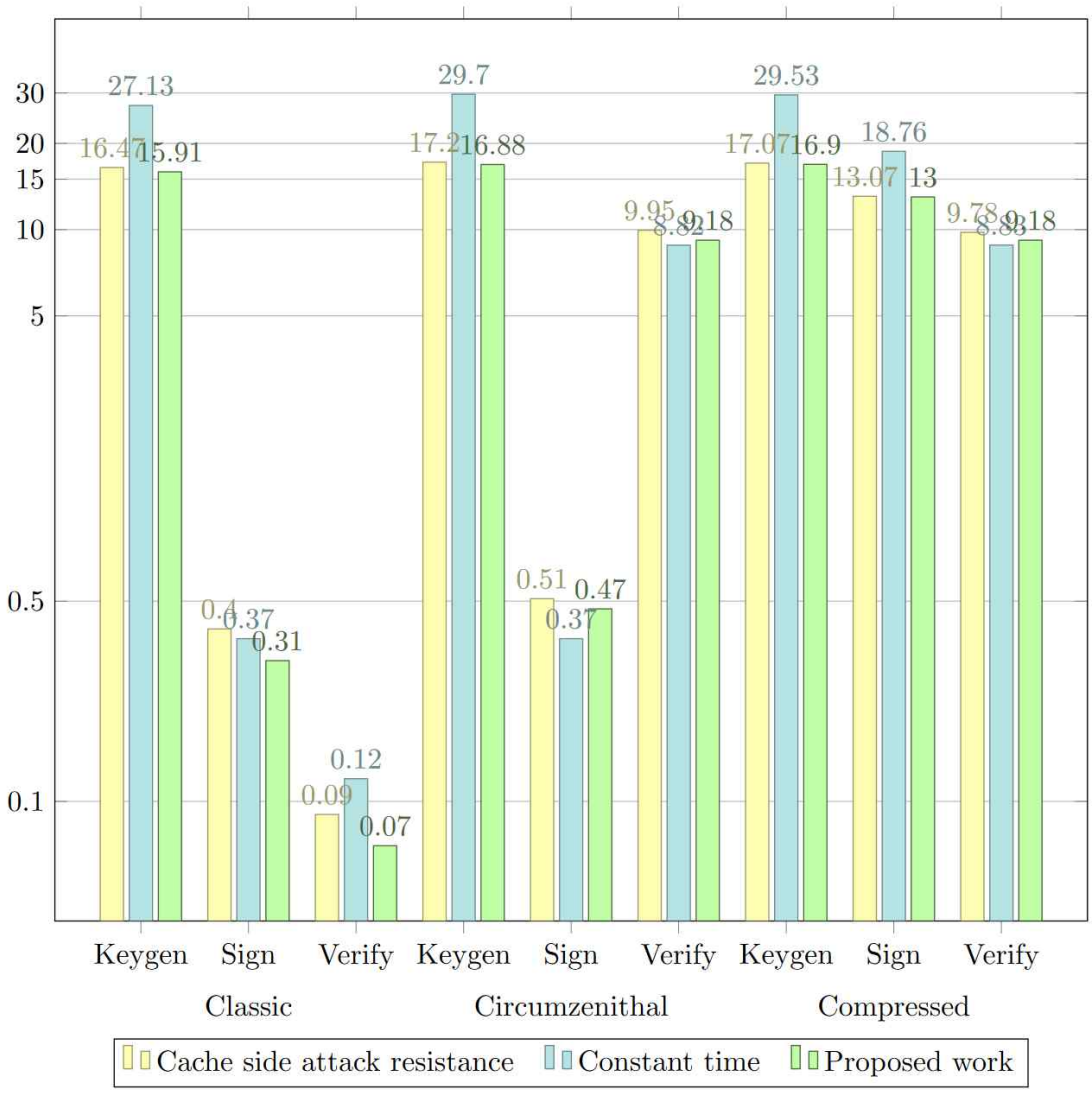
Lastly, the performance measurements for Rainbow V on the M1 processor are summarized in [Figure 4-8]. The Rainbow V Classic version showed performance gains of 1565.6%, 2384.6%, and 2219.7% for keygen, sign, and verify, respectively. The Rainbow V Circumzenithal version recorded improvements of 1706.9%, 2344.8%, and 46.8%. Finally, the Rainbow V Compressed version exhibited



[Figure 4-8] Performance Measurement Results for Rainbow V on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles)

enhancements of 1707.3%, 1633.2%, and 47.0% in the respective processes.

The performance evaluation results of the Cache side attack resistance and Constant time implementations are presented, as shown in the graph in [Figure 4-9], which focuses on Rainbow I. Data for Rainbow III and V's Cache side attack resistance implementations are provided in the appendix.



[Figure 4-9] Performance Measurement Results for Rainbow I cache side attack resistance implementation and constant-time implementation on Apple M1 processors expressed in log scale (Unit: 10^6 clock cycles)

When comparing the Cache side attack resistance implementation with the proposed optimized implementation, Rainbow I Classic exhibited performance decreases of 3.4%, 22.5%, and 22.2% for key generation, signing, and verification, respectively. In the Circumzenithal version, the performance decreases were 1.9%, 7.8%,

and 7.7% for keygen, sign, and verify. The Compressed version showed the smallest performance impact, with reductions of 1.0%, 0.5%, and 6.1%, respectively.

Despite the larger performance drops observed in the Classic version's signing and verification processes, the actual increase in cycle counts was relatively small, at 0.09 million cycles for sign and 0.02 million cycles for verify. This apparent discrepancy arises because the original optimized implementation already had very low cycle counts, making the percentage decrease appear more significant.

For the Constant time implementation, Rainbow I Classic experienced performance drops of 41.4%, 16.2%, and 41.7% for keygen, sign, and verify, respectively. In the Circumzenithal version, key generation experienced a performance decrease of 43.2%, while sign and verify showed performance improvements of 27.0% and 4.1%, respectively. The Compressed version saw reductions of 42.8% and 30.7% for keygen and sign, respectively, while verification showed a 4.0% performance improvement. The observed performance increases in some cases are due to measurement variations, and under typical conditions, the Constant time implementation is generally slower than the optimized implementation.

Overall, the Constant time implementation exhibited greater performance degradation compared to the Cache side attack resistance implementation. This is because ensuring constant-time execution significantly increases the operation time. Additionally, the Constant time implementation is limited to Rainbow I due to register constraints, though it offers the crucial advantage of guaranteeing constant-time behavior.

5. Conclusion

This dissertation presents optimized implementations of the lightweight cipher CHAM, TinyJAMBU, and the post-quantum cryptography Rainbow signature. The implementations were carried out on 8-bit AVR and 64-bit ARMv8 processors, which are commonly used in embedded hardware. For the CHAM block cipher, a technique that skips certain operations in the initial 8 rounds was applied, leveraging the characteristics of block cipher counter mode. As a result, performance improvements ranging from 9.7% to 18.4% were achieved, with a maximum improvement of 26.9% for the specialized Furious CHAM variant.

TinyJAMBU utilized reverse bitwise shifts that take advantage of AVR register storage to optimize the keyed permutation. This optimization led to performance enhancements exceeding 500% for the keyed permutation alone. When the optimized permutation was applied to TinyJAMBU, overall performance improved by approximately 470%. Additionally, applying an initialization-skipping technique resulted in a maximum performance gain of 915.3%.

Finally, for the Rainbow signature, the tower field-based multiplication, which could not be fully supported by ARMv8 assembly instructions, was converted into a lookup table format for more efficient multiplication. This modification yielded performance improvements between 477.9% and 42,773.1% across various processors. When the proposed multiplication method was applied to Rainbow signature on the M1 processor, the maximum performance gains were observed as follows: 3464.1% for key generation in Rainbow III, 2384.6% for signing in Rainbow V, and 4614.3% for

verification in Rainbow I.

The specialized implementations of Rainbow signature, such as the Cache side attack resistance and Constant time versions, exhibited slightly lower performance compared to the optimized implementation but provided the advantage of resistance to side-channel attacks. The optimization techniques proposed in this dissertation demonstrate effective performance on embedded processors and can be applied to other cryptographic algorithms with similar structures to achieve optimized implementations.

Appendix: Look-Up Table for Rainbow

[Table Appendix-1] Precomputation look-up table of tower-field based polynomial multiplication results on GF16 expressed in hexadecimal (A: additional table for Rainbow III and V)

×	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x1	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x2	0x0	0x2	0x3	0x1	0x8	0xa	0xb	0x9	0xc	0xe	0xf	0xd	0x4	0x6	0x7	0x5
0x3	0x0	0x3	0x1	0x2	0xc	0xf	0xd	0xe	0x4	0x7	0x5	0x6	0x8	0xb	0x9	0xa
0x4	0x0	0x4	0x8	0xc	0x6	0x2	0xe	0xa	0xb	0xf	0x3	0x7	0xd	0x9	0x5	0x1
0x5	0x0	0x5	0xa	0xf	0x2	0x7	0x8	0xd	0x3	0x6	0x9	0xc	0x1	0x4	0xb	0xe
0x6	0x0	0x6	0xb	0xd	0xe	0x8	0x5	0x3	0x7	0x1	0xc	0xa	0x9	0xf	0x2	0x4
0x7	0x0	0x7	0x9	0xe	0xa	0xd	0x3	0x4	0xf	0x8	0x6	0x1	0x5	0x2	0xc	0xb
0x8	0x0	0x8	0xc	0x4	0xb	0x3	0x7	0xf	0xd	0x5	0x1	0x9	0x6	0xe	0xa	0x2
0x9	0x0	0x9	0xe	0x7	0xf	0x6	0x1	0x8	0x5	0xc	0xb	0x2	0xa	0x3	0x4	0xd
0xa	0x0	0xa	0xf	0x5	0x3	0x9	0xc	0x6	0x1	0xb	0xe	0x4	0x2	0x8	0xd	0x7
0xb	0x0	0xb	0xd	0x6	0x7	0xc	0xa	0x1	0x9	0x2	0x4	0xf	0xe	0x5	0x3	0x8
0xc	0x0	0xc	0x4	0x8	0xd	0x1	0x9	0x5	0x6	0xa	0x2	0xe	0xb	0x7	0xf	0x3
0xd	0x0	0xd	0x6	0xb	0x9	0x4	0xf	0x2	0xe	0x3	0x8	0x5	0x7	0xa	0x1	0xc
0xe	0x0	0xe	0x7	0x9	0x5	0xb	0x2	0xc	0xa	0x4	0xd	0x3	0xf	0x1	0x8	0x6
0xf	0x0	0xf	0x5	0xa	0x1	0xe	0x4	0xb	0x2	0xd	0x7	0x8	0x3	0xc	0x6	0x9
A	0x0	0x8	0xc	0x4	0xb	0x3	0x7	0xf	0xd	0x5	0x1	0x9	0x6	0xe	0xa	0x2

[Table Appendix-2] Implementation code of proposed multiplication for Rainbow III and V (x0: output address, x1: operand address, x2(w2): constant)

Line	Code				Comment
1:	MOVI	v31.16b,	#15		
2:	AND	w4,	w2,	#15	
3:	LSR	w5,	w2,	#4	
4:	ADR	x6,	MUL_TABLE		Initial address1
5:	LSL	w4,	w4,	#4	Multiplied by 16
6:	ADD	x6,	x6,	x4	Get table1 address
7:	ADR	x7,	MUL_TABLE		Initial address2
8:	LSL	w5,	w5,	#4	Multiplied by 16
9:	ADD	x7,	x7,	x5	Get table2 address
10:	LD1.16b	{v30},	[x6]		Load table1 values
11:	LD1.16b	{v29},	[x7]		Load table2 values
12:	ADR	x6,	ADDI_TABLE		Initial addressA
13:	LD1.16b	{v27},	[x6]		Load tableA values
14:	LD1.16b	{v1},	[x1],	#16	
15:	LD1.16b	{v5},	[x1],	#16	
16:	AND.16b	v0,	v1,	v31	Divide into 4-bit
17:	USHR.16b	v1,	v1,	#4	
18:	AND.16b	v4,	v5,	v31	
19:	USHR.16b	v5,	v5,	#4	
20:	TBL.16b	v2,	{v30},	v0	Table look-up
21:	TBL.16b	v3,	{v29},	v1	
22:	TBL.16b	v6,	{v30},	v4	
23:	TBL.16b	v7,	{v29},	v5	
24:	EOR.16b	v0,	v0,	v1	

```
25:   EOR.16b   v4,   v4,   v5
26:   AND       w4,   w2,   #15
27:   LSR       w5,   w2,   #4
28:   EOR       w4,   w4,   w5
29:   ADR       x6,   MUL_TABLE   Initial address3
30:   LSL       w4,   w4,   #4    Multiplied by 16
31:   ADD       x6,   x6,   x4    Get table3 address
32:   LD1.16b   {v28}, [x6]       Load table3 values
33:   TBL.16b   v0,   {v28}, v0   Table look-up
34:   EOR.16b   v0,   v0,   v2
35:   TBL.16b   v4,   {v28}, v4
36:   EOR.16b   v4,   v4,   v6
37:   TBL.16b   v3,   {v27}, v3
38:   TBL.16b   v7,   {v27}, v7
39:   SHL.16b   v0,   v0,   #4
40:   EOR.16b   v0,   v0,   v2
41:   EOR.16b   v0,   v0,   v3
42:   SHL.16b   v4,   v4,   #4
43:   EOR.16b   v4,   v4,   v6
44:   EOR.16b   v4,   v4,   v7
45:   LD1.16b   {v1},  [x0], #16
46:   LD1.16b   {v5},  [x0], #16
47:   SUB       x0,   x0,   #32
48:   EOR.16b   v1,   v1,   v0
49:   EOR.16b   v5,   v5,   v4
50:   ST1.16b   {v1},  [x0], #16
51:   ST1.16b   {v5},  [x0], #16   Return
```

[Table Appendix-3] Implementation code of constant-time implementation. (x2(w2): constant)

Line	Code	Comment
1:	ADR x4, MUL_TABLE	
2:	LD1.16b {v16}, [x4],	Load offset0 table
3:	(All table values load)	
4:	LD1.16b {v31}, [x4],	Load offset15 table
5:	CMP w2, #8	
6:	BLT MUL_07	
7:	B MUL_815	
8:	MUL_BACK:	
9:	(The multiplication code section)	
10:	RET	Return
11:	MUL_07:	
12:	B	Dummy branch
13:	CMP w2, #4	
14:	BLT MUL_03	
15:	B MUL_47	
16:	MUL_03:	
17:	B	Dummy branch
18:	CMP w2, #2	
19:	BLT MUL_01	
20:	B MUL_23	
21:	MUL_01:	
22:	B	Dummy branch
23:	CMP w2, #1	
24:	BEQ MUL_1	
25:	B MUL_0	

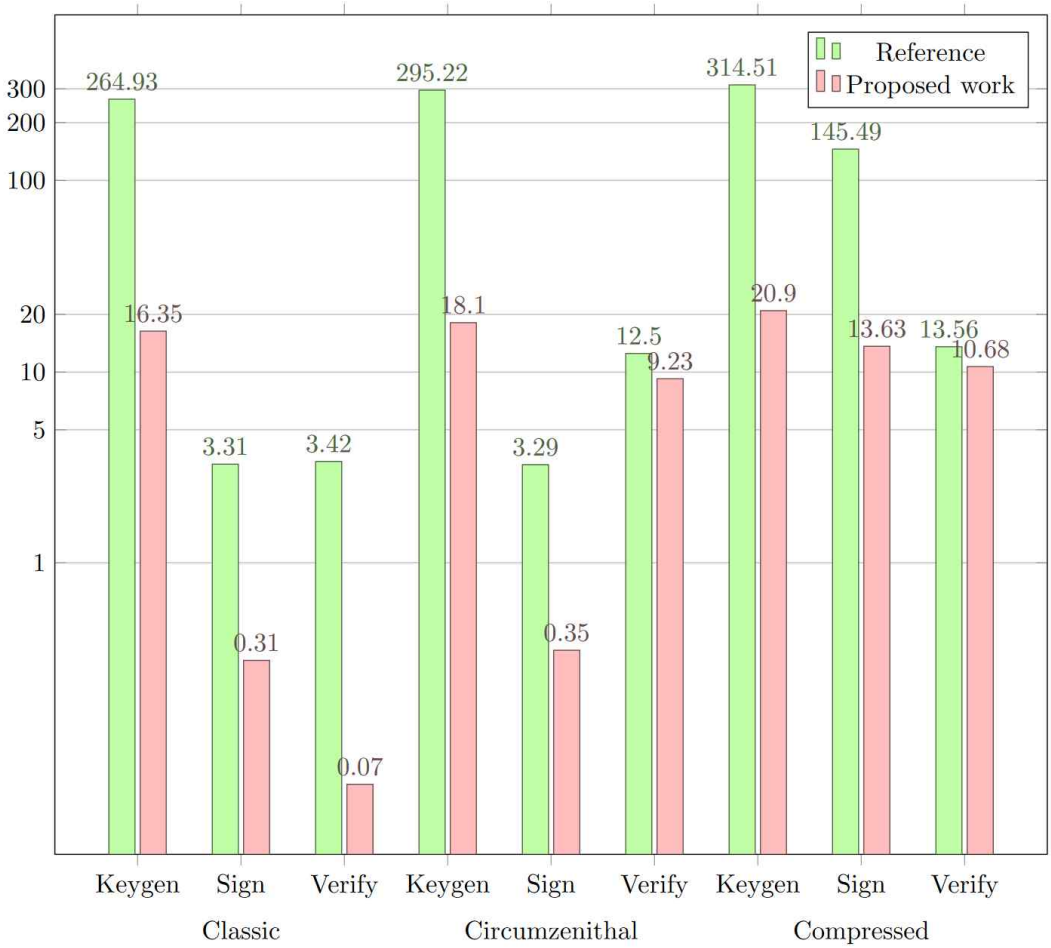
```
26:  MUL_23:
27:  CMP     w2,     #3
28:  BEQ     MUL_3
29:  B       MUL_2
30:  MUL_47:
31:  CMP     w2,     #6
32:  BLT     MUL_45
33:  B       MUL_67
34:  MUL_45:
35:  B                                  Dummy branch
36:  CMP     w2,     #5
37:  BEQ     MUL_5
38:  B       MUL_4
39:  MUL_67:
40:  CMP     w0,     #7
41:  BLT     MUL_811
42:  B       MUL_1215
43:  MUL_815:
44:  BLT     MUL_811
45:  B       MUL_1215
46:  MUL_811:
47:  B                                  Dummy branch
48:  CMP     w2,     #10
49:  BLT     MUL_89
50:  B       MUL_1011
51:  MUL_89:
52:  B                                  Dummy branch
53:  CMP     w2,     #9
```

```
54:   BEQ       MUL_9                    
55:   B         MUL_8                    
56:   MUL_1011:                           
57:   CMP       w2,      #11             
58:   BEQ       MUL_11                   
59:   B         MUL_10                   
60:   MUL_1215:                           
61:   CMP       w2,      #14             
62:   BLT       MUL_1213                 
63:   B         MUL_1415                 
64:   MUL_1213:                           
65:   B                                  Dummy branch
66:   CMP       w2,      #13             
67:   BEQ       MUL_13                   
68:   B         MUL_12                   
69:   MUL_1415:                           
70:   CMP       w2,      #15             
71:   BEQ       MUL_15                   
72:   B         MUL_14                   
73:   MUL_0:                             
74:   MOV       v30.16b, v16.16b         Get actual table value
75:   B         MUL_BACK                 
76:   MUL_1:                             
77:   B                                  Dummy branch
78:   MOV       v30.16b, v17.16b         Get actual table value
79:   B         MUL_BACK                 
80:   MUL_2:                             
81:   MOV       v30.16b, v18.16b         Get actual table value
82:   B         MUL_BACK                 
```

```
83:   MUL_3:
84:   B                                   Dummy branch
85:   MOV     v30.16b, v19.16b            Get actual table value
86:   B       MUL_BACK
87:   MUL_4:
88:   MOV     v30.16b, v20.16b            Get actual table value
89:   B       MUL_BACK
90:   MUL_5:
91:   B                                   Dummy branch
92:   MOV     v30.16b, v21.16b            Get actual table value
93:   B       MUL_BACK
94:   MUL_6:
95:   MOV     v30.16b, v22.16b            Get actual table value
96:   B       MUL_BACK
97:   MUL_7:
98:   B                                   Dummy branch
99:   MOV     v30.16b, v23.16b            Get actual table value
100:  B       MUL_BACK
101:  MUL_8:
102:  MOV     v30.16b, v24.16b            Get actual table value
103:  B       MUL_BACK
104:  MUL_9:
105:  B                                   Dummy branch
106:  MOV     v30.16b, v25.16b            Get actual table value
107:  B       MUL_BACK
108:  MUL_10:
109:  MOV     v30.16b, v26.16b            Get actual table value
110:  B       MUL_BACK
111:  MUL_11:
```

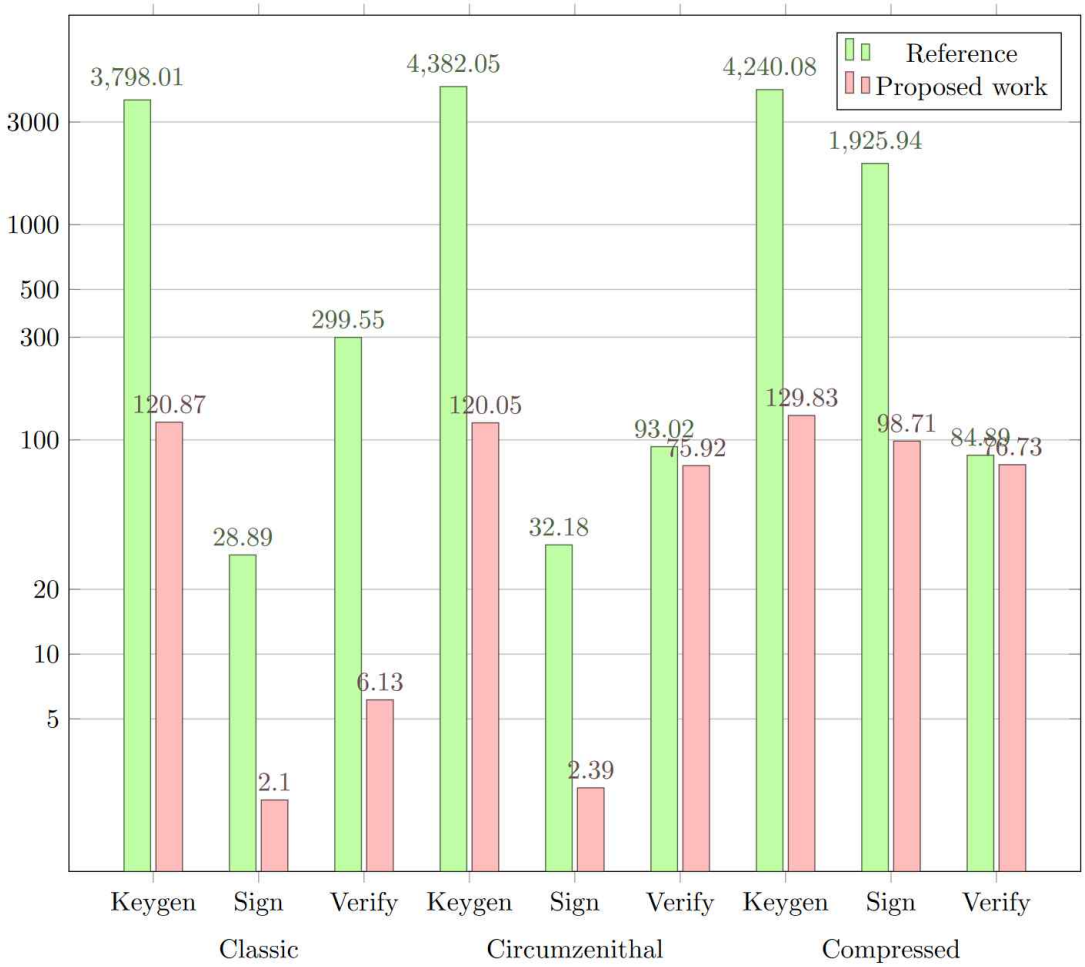
```
112:    B                                   Dummy branch
113:    MOV     v30.16b,  v27.16b           Get actual table value
114:    B       MUL_BACK
115:    MUL_12:
116:    MOV     v30.16b,  v28.16b           Get actual table value
117:    B       MUL_BACK
118:    MUL_13:
119:    B                                   Dummy branch
120:    MOV     v30.16b,  v29.16b           Get actual table value
121:    B       MUL_BACK
122:    MUL_14:
123:    MOV     v30.16b,  v30.16b           Get actual table value
124:    B       MUL_BACK
125:    MUL_15:
126:    B                                   Dummy branch
127:    MOV     v30.16b,  v31.16b           Get actual table value
128:    B       MUL_BACK
```

Appendix: Performance evaluation result for Rainbow on A13 processors



[Figure Appendix-1] Performance Measurement Results for Rainbow I on Apple A13 processors expressed in log scale (Unit: 10^6 clock cycles)

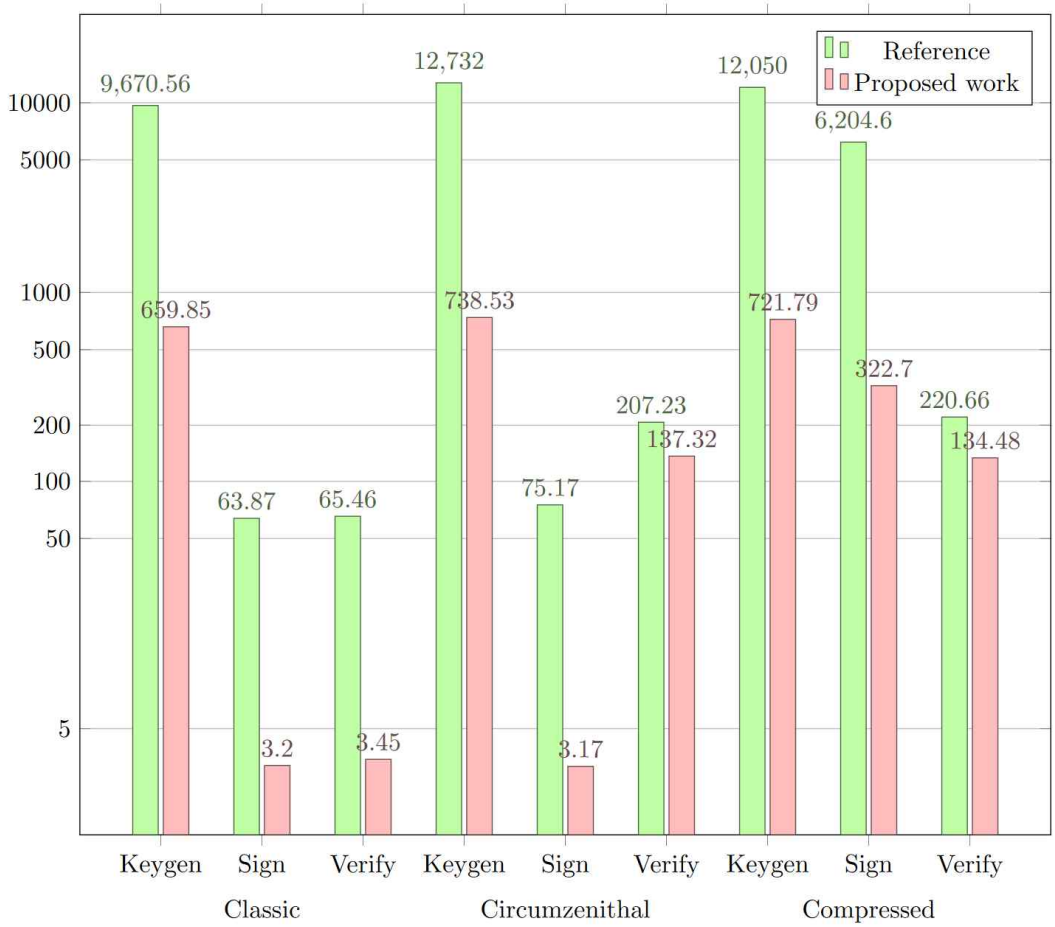
The performance evaluation results of Rainbow I on the A13 processor are shown in [Figure Appendix-1], with the graph represented on a logarithmic scale. For the Classic version, the performance improvements were 1520.4% for key generation, 967.7% for signing, and 4785.7% for verification. In the Circumzenithal version, the improvements for keygen, sign, and verify were 1531.0%, 840.0%, and 35.4%, respectively. The Compressed version achieved



[Figure Appendix-2] Performance Measurement Results for Rainbow III on Apple A13 processors expressed in log scale (Unit: 10^6 clock cycles)

enhancements of 1404.8%, 967.4%, and 27.0%, respectively.

The performance evaluation results of Rainbow III are illustrated in the graph in [Figure Appendix-2]. For the Classic Rainbow version, performance improvements were observed at 3042.2% for key generation, 1275.7% for signing, and 4786.6% for verification. In the Circumzenithal version, the improvements for keygen, sign, and verify were 3550.2%, 1246.4%, and 22.5%, respectively. Lastly, the Compressed version showed performance gains of 3165.9%, 1851.1%,

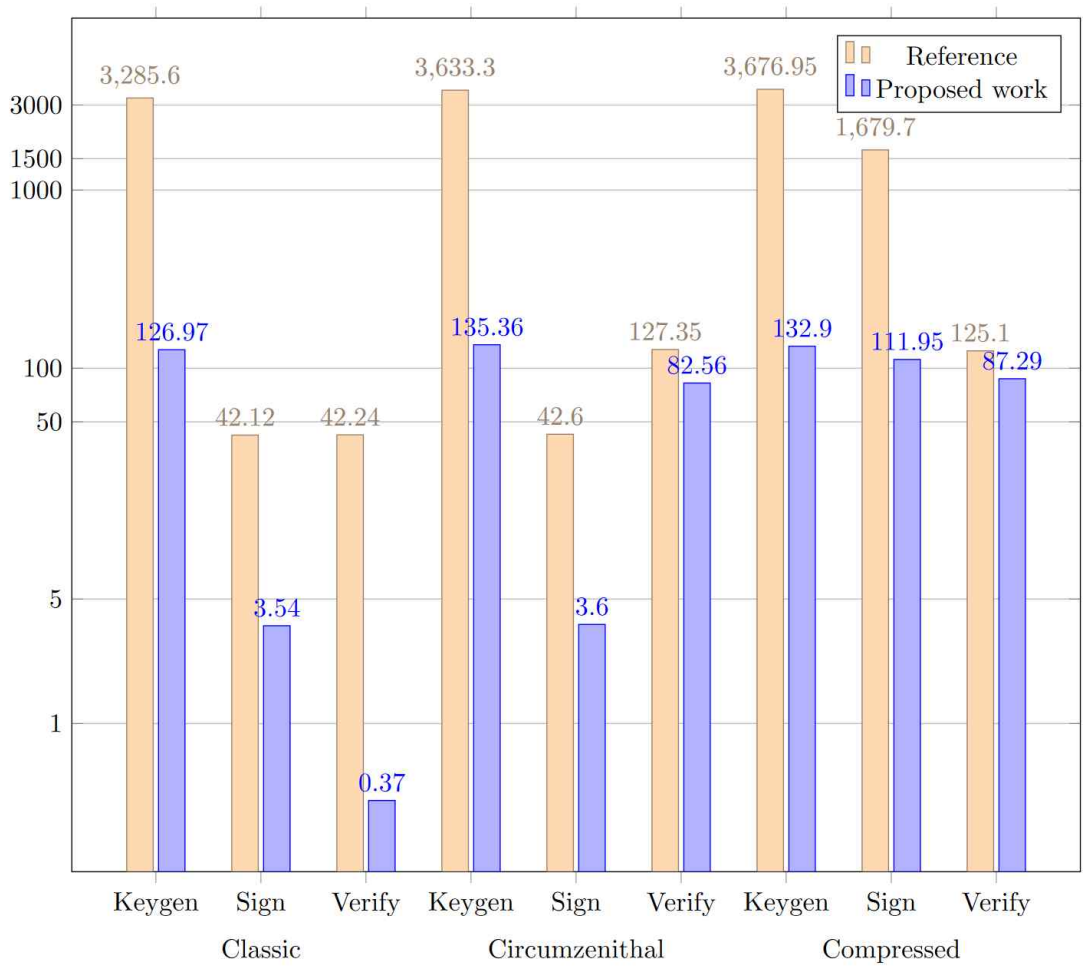


[Figure Appendix-3] Performance Measurement Results for Rainbow V on Apple A13 processors expressed in log scale (Unit: 10^6 clock cycles)

and 14.5% for key generation, signing, and verification, respectively.

The performance evaluation results of Rainbow V are summarized in the graph in [Figure Appendix-3]. For the Rainbow V Classic version, performance improvements of 1365.6% for key generation, 1895.9% for signing, and 1797.4% for verification were observed. In the Circumzenithal version, keygen showed an improvement of 1624.0%, sign improved by 2271.3%, and verify improved by 50.9%. For the Compressed version, the performance gains were 1569.5% for key generation, 1822.7% for signing, and 64.1% for verification.

Appendix: Performance evaluation result for Rainbow on BCM2711 processors

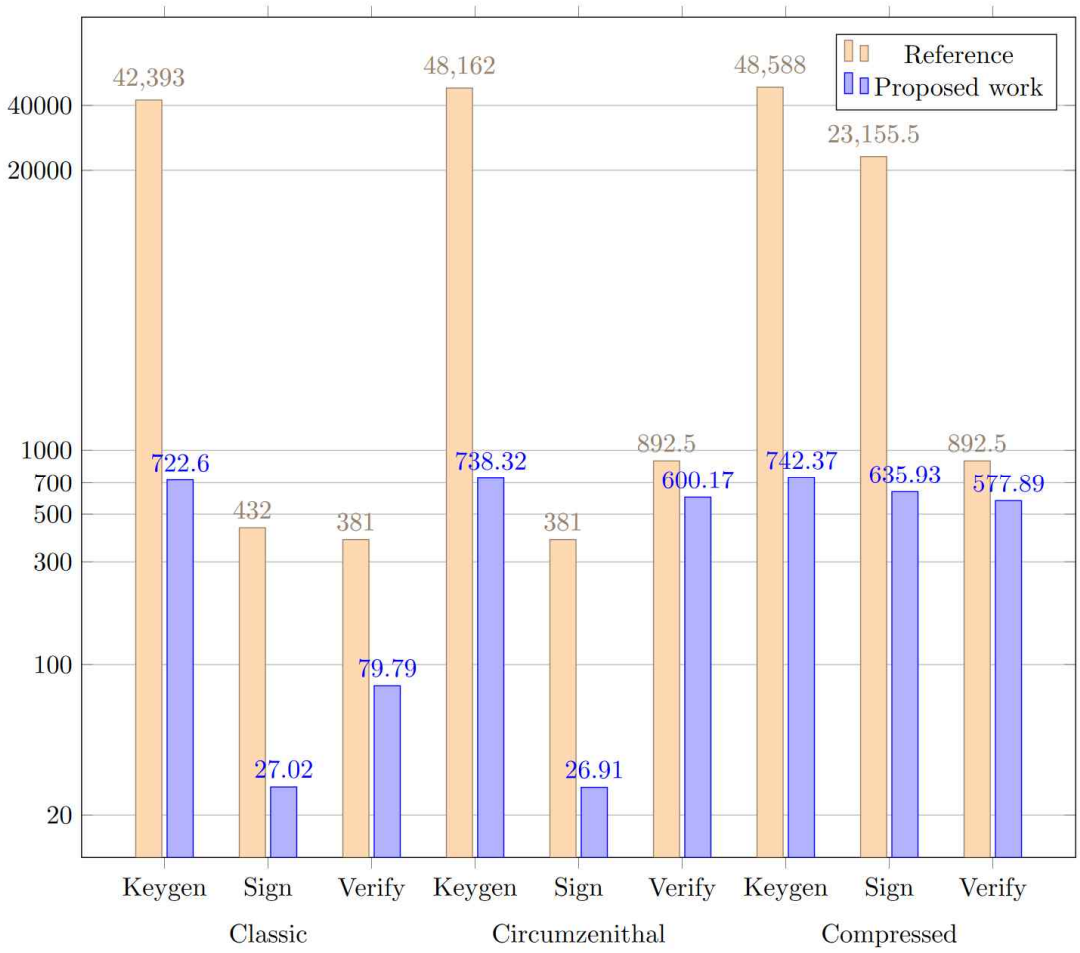


[Figure Appendix-4] Performance Measurement Results for Rainbow I on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles)

The implementation results of Rainbow on the BCM2711 processor are evaluated. The performance measurements for Rainbow I are shown in [Figure Appendix-4], with the graph presented on a logarithmic scale. For Rainbow I Classic, performance improvements of 2487.7% for key generation, 1089.8% for signing, and 11316.2%

for verification were observed. The significant improvement in verification is attributed to the original implementation being exceptionally slow on the BCM2711 processor.

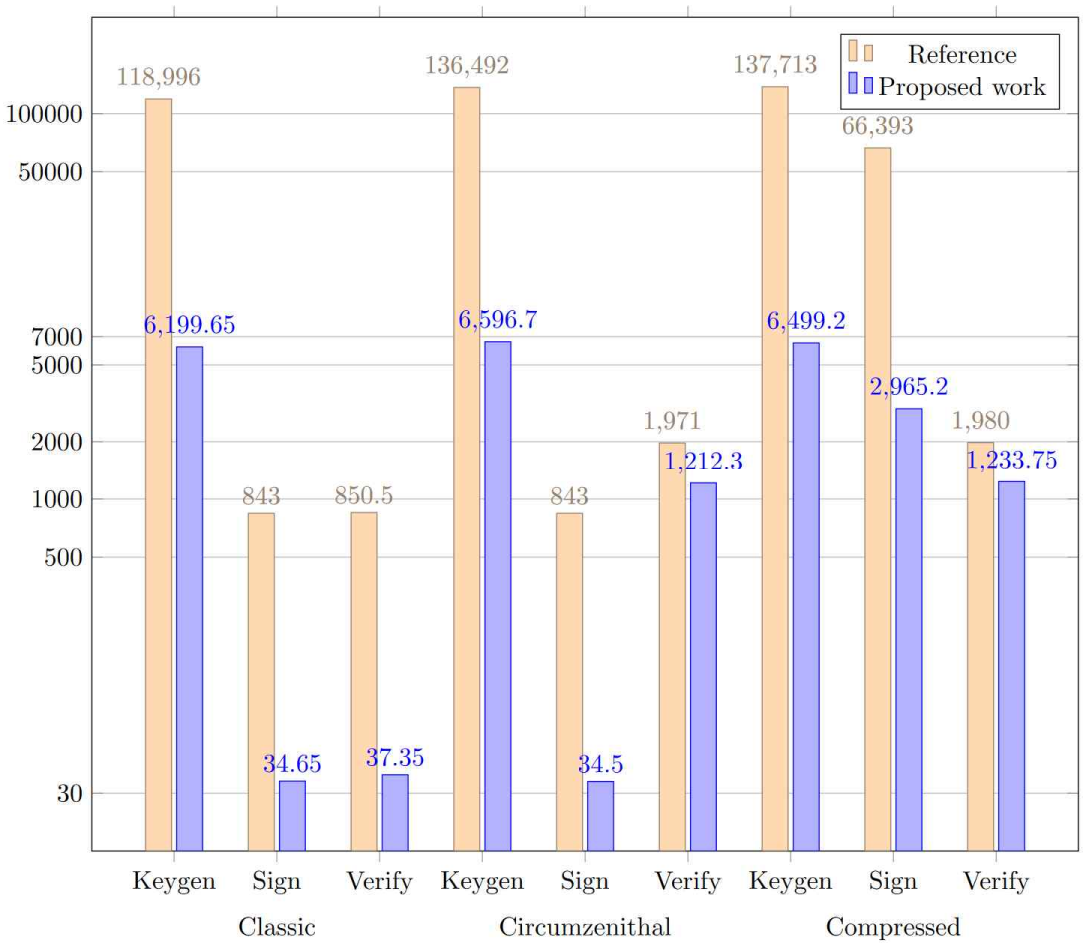
In the Circumzenithal version, the performance gains were 2584.2% for keygen, 1083.3% for sign, and 54.3% for verify. For the Compressed version, keygen improved by 2666.7%, sign by 1400.4%, and verify by 43.3%.



[Figure Appendix-5] Performance Measurement Results for Rainbow III on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles)

The performance evaluation results for Rainbow III are summarized

in [Figure Appendix-5]. For the Classic version, key generation showed a performance improvement of 5766.7%, while signing and verification improved by 1498.8% and 377.5%, respectively. In the Circumzenithal version, keygen improved by 6423.2%, sign by 1315.8%, and verify by 48.7%. Lastly, in the Compressed version, keygen demonstrated a 6445.0% improvement, sign improved by 3541.2%, and verify improved by 54.4%.

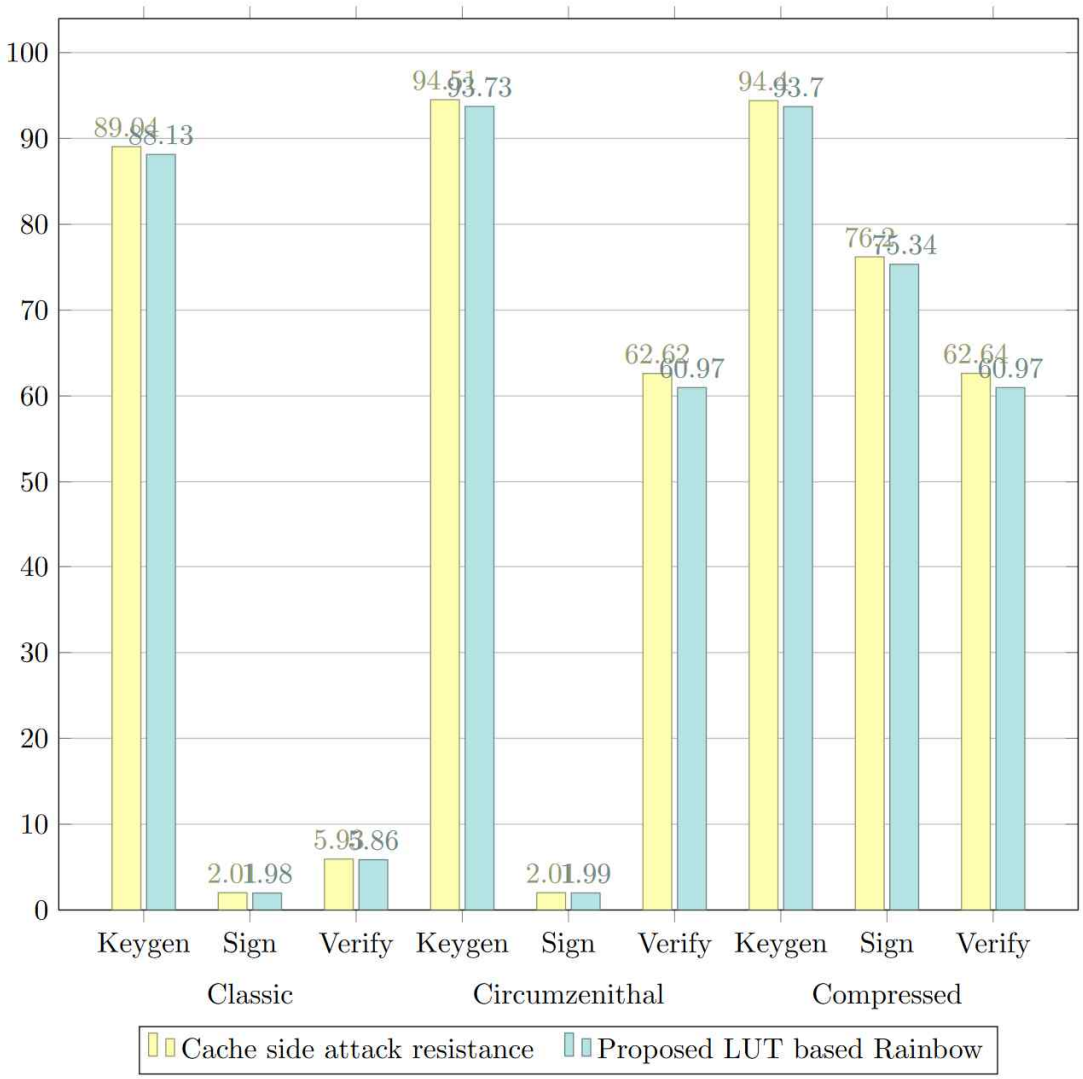


[Figure Appendix-6] Performance Measurement Results for Rainbow V on BCM2711 processors expressed in log scale (Unit: 10^6 clock cycles)

The performance evaluation results for Rainbow III are shown in

[Figure Appendix-6]. For Rainbow V Classic, performance improvements of 1819.4%, 2332.9%, and 2771.1% were observed for key generation, signing, and verification, respectively. In the Circumzenithal version, the improvements were 1969.1% for keygen, 2343.5% for sign, and 62.6% for verify. For the Compressed version, performance gains of 2018.9%, 2139.1%, and 60.5% were recorded for key generation, signing, and verification, respectively.

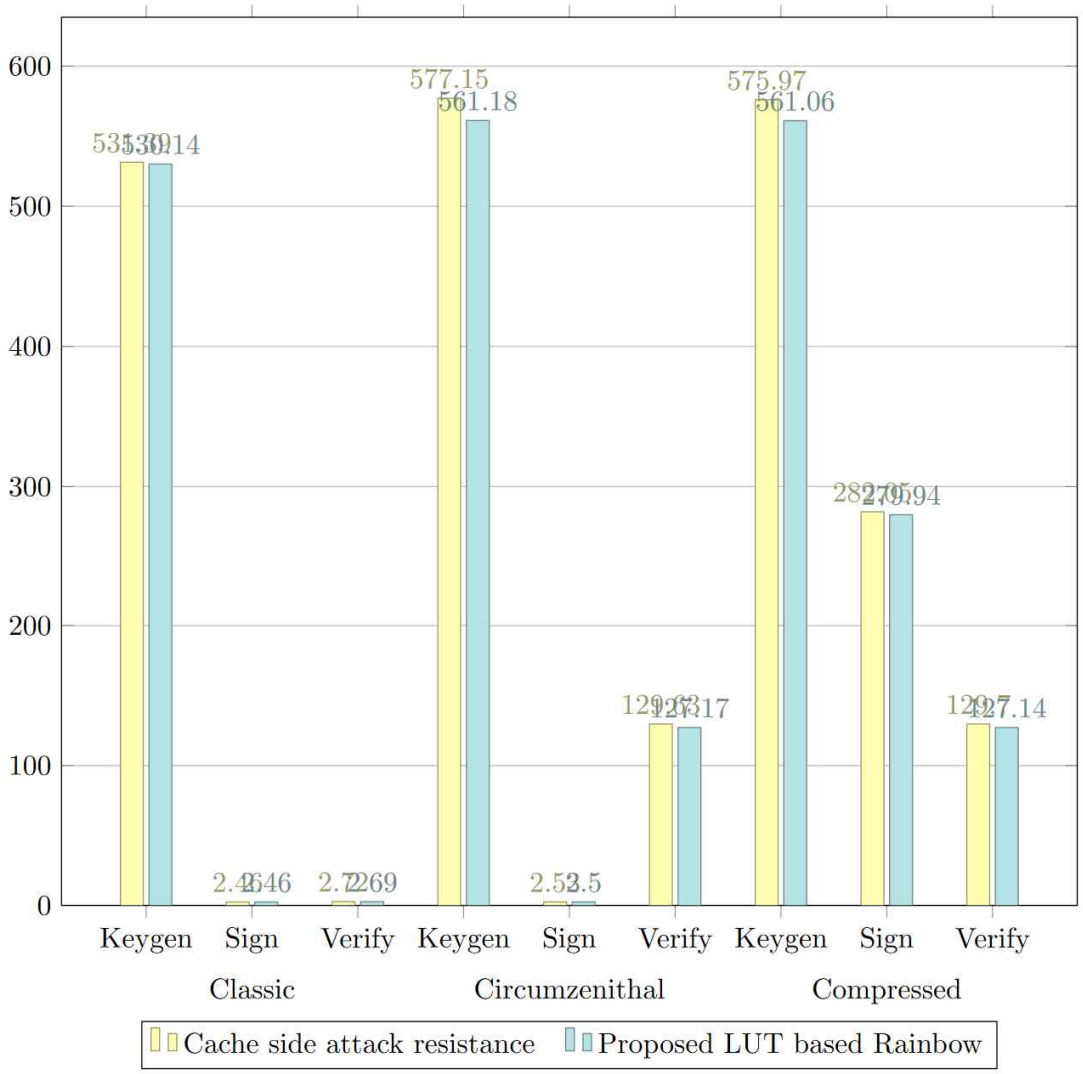
Appendix: Performance evaluation result for Cache side attack resistance implementation of Rainbow III and V



[Figure Appendix-7] Performance Measurement Results for Rainbow III cache side attack resistance implementation on Apple M1 processors (Unit: 10^6 clock cycles)

The performance of the cache side attack resistant implementation

of Rainbow III is summarized in [Figure Appendix-7]. Overall, there is a slight performance decrease, with the maximum degradation being 2.7% and the minimum being 0.7%, showing results that are nearly comparable to the original implementation.



[Figure Appendix-8] Performance Measurement Results for Rainbow V cache side attack resistance implementation on Apple M1 processors (Unit: 10^6 clock cycles)

The performance evaluation results for the cache side attack

resistant implementation of Rainbow V are also presented in [Figure Appendix-8]. Similarly, there is a minor performance drop overall, with a maximum decrease of 2.8% and a minimum of 0%.

Appendix: Abbreviation

- AES: Advanced Encryption Standard
- ARX: Add-Rotate-XOR
- AVR: Advanced Virtual RISC
- CBC: Cipher Block Chaining
- CFB: Cipher Feedback
- CPB: Cycles Per Byte
- CTR: Counter Mode
- CVP: Closest Vector Problem
- DES: Data Encryption Standard
- ECB: Electronic Codebook
- GF: Galois Field
- IoT: Internet of Things
- KEM: Key Encapsulation Mechanism
- LEA: Lightweight Encryption Algorithm
- NIST: National Institute of Standards and Technology
- NLFSR: Non-Linear Feedback Shift Register
- OFB: Output Feedback
- PKE: Public-Key Encryption
- PQC: Post-Quantum Cryptography
- RBS: Reverse Bitwise Shift
- RSA: Rivest-Shamir-Adleman
- SIMD: Single Instruction, Multiple Data
- SPHINCS+: Secure Hash-Based Signature Scheme
- SVP: Shortest Vector Problem
- XOR: Exclusive-OR

Bibliography

1. Domestic Literature

Kwon, H., Sim, M., Lim, S., Kang, Y., & Seo, H. (2022). Technological Trends in Quantum-Resistant Blockchain. *Review of KIISC*, 32(1), 7-17.

Kwon, H., Jang, K., Kim, H., & Seo, H. (2021). The fast implementation of block cipher SIMON using pre-computation with counter mode of operation, *Journal of the Korea Institute Of Information and Communication Engineering(JKIICE)*, 25(4), 588-594.

Kim, K., Choi, S., Kwon, H., Liu, Z., & Seo, H. (2020). FACE-LIGHT: Fast AES-CTR mode encryption for low-end microcontrollers. *In Information Security and Cryptology-ICISC 2019: 22nd International Conference, Seoul, South Korea*, December 4-6, 2019, Revised Selected Papers 22 (pp. 102-114). Springer International Publishing.

2. International Literature

Delfs, H., Knebl, H., Delfs, H., & Knebl, H. (2015). Symmetric-key cryptography. *Introduction to Cryptography: Principles and Applications*, 11-48.

Hellman, M. E. (2002). An overview of public key cryptography. *IEEE Communications Magazine*, 40(5), 42-49.

Standard, D. E. (1999). Data encryption standard. *Federal Information Processing Standards Publication*, 112, 3.

Rijmen, V., & Daemen, J. (2001). Advanced encryption standard. *Proceedings of federal information processing standards publications, national institute of standards and technology*, 19, 22.

Lee, D., Kim, D. C., Kwon, D., & Kim, H. (2014). Efficient hardware implementation of the lightweight block encryption algorithm LEA. *Sensors*, 14(1), 975-994.

Kwon, D., Kim, J., Park, S., Sung, S. H., Sohn, Y., Song, J. H., ... & Hong, J. (2003, November). New block cipher: ARIA. *In International conference on information security and cryptology* (pp. 432-445). Berlin, Heidelberg: Springer Berlin Heidelberg.

Roh, D., Koo, B., Jung, Y., Jeong, I. W., Lee, D. G., Kwon, D., & Kim, W. H. (2020). Revised version of block cipher CHAM. *In Information Security and Cryptology-ICISC 2019: 22nd International Conference, Seoul, South Korea, December 4-6, 2019*, Revised Selected Papers 22 (pp. 1-19). Springer International Publishing.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 120-126.

Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. *In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (pp. 212-219).

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2), 303-332.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., ... & Stehlé, D. (2018, April). CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. *In 2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (pp. 353-367). IEEE.

Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., ... & Bai, S. (2020). Crystals-dilithium. Algorithm Specifications and Supporting Documentation.

Prest, T., Fouque, P. A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., ... & Zhang, Z. (2020). Falcon. Post-Quantum Cryptography Project of NIST.

Bernstein, D. J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., & Schwabe, P. (2019, November). The SPHINCS+ signature framework. *In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security* (pp. 2129-2146).

Kim, J., & Park, J. H. (2023). NTRU+: Compact construction of NTRU using simple encoding method. *IEEE Transactions on Information Forensics and Security*.

Kim, D. C., Jeon, C. Y., Kim, Y., & Kim, M. (2023, April). PALOMA: binary separable Goppa-based KEM. *In Code-Based Cryptography Workshop* (pp. 144-173). Cham: Springer Nature Switzerland.

Kim, J. L., Hong, J., Lau, T. S. C., Lim, Y., & Won, B. S. (2022). REDOG and its performance analysis. *Cryptology ePrint Archive*.

Cheon, J. H., Choe, H., Hong, D., Hong, J., Seong, H., Shin, J., & Yi, M. (2024). SMAUG: the Key Exchange Algorithm based on Module-LWE and Module-LWR. Algorithm Specifications Version, 3.

Kim, S., Ha, J., Son, M., Lee, B., Moon, D., Lee, J., ... & Lee, J. (2023, November). AIM: symmetric primitive for shorter sig-natures with stronger security. *In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (pp. 401-415).

Cheon, J. H., Choe, H., Devevey, J., Güneysu, T., Hong, D., Krausz, M., ... & Yi, M. (2023). Haetae: Shorter lattice-based fiat-shamir

signatures. *Cryptology ePrint Archive*.

Shim, K. A., Kim, J., & An, Y. (2022). MQ-sign: A new post-quantum signature scheme based on multivariate quadratic equations: Shorter and faster. KpqC Round, 1.

Shim, K. A., Kim, J., & Kwon, H. (2024). NCC-Sign: A New Lattice-based Signature Scheme using Non-Cyclotomic Polynomials and Trinomials. KpqC Round, 1.

Wu, H., & Huang, T. (2019). TinyJAMBU: A family of lightweight authenticated encryption algorithms. Submission to the NIST Lightweight Cryptography Standardization Process (March 2019).

Wu, H., & Huang, T. (2014). JAMBU lightweight authenticated encryption mode and AES-JAMBU. *CAESAR competition proposal*.

Ding, J., Chen, M. S., Kannwischer, M., Patarin, J., Petzoldt, A., Schmidt, D., & Yang, B. Y. (2020). Rainbow—Algorithm Specification and Documentation, The 3rd Round Proposal. NIST Post-Quantum Cryptography Standardization Round, 3.

Karatsuba, A. A. (1995). The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 211, 169-183.

Bernstein, D. J., & Chou, T. (2014, August). Faster binary-field multiplication and faster binary-field macs. *In International Conference on Selected Areas in Cryptography* (pp. 92-111). Cham: Springer International Publishing.

Seo, H., Jeong, I., Lee, J., & Kim, W. H. (2018). Compact implementations of ARX-based block ciphers on IoT processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3), 1-16.

Seo, H., Liu, Z., Longa, P., & Hu, Z. (2018). SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 1-20.

Kim, Y., Song, J., Youn, T. Y., & Seo, S. C. (2022). Crystals-Dilithium on ARMv8. Security and Communication Networks, 2022(1), 5226390.

국 문 초 록

임베디드 프로세서 특성 기반
암호 알고리즘의 구조적 최적화

한 성 대 학 교 대 학 원
정 보 컴 퓨 터 공 학 과
정 보 시 스 템 공 학 전 공
권 혁 동

본 논문에서는 암호 알고리즘의 내부 구조 변경을 통한 최적 구현 기법에 대해 연구 및 그 결과를 제시한다. 암호 알고리즘의 최적 구현 관점 중에서 속도 최적화는 알고리즘의 연산 속도를 빠르게 하여 비효율적인 연산 성능을 개선하는 것이다. 최적 구현에서는 주로 병렬 구현이 많이 사용되는데, 알고리즘 내부 연산을 병렬 구현하는 것으로는 최적 구현의 한계점이 존재한다. 제안하는 기법은 알고리즘의 내부 구조를 변경하는 것으로 암호 알고리즘의 성능을 향상시키는 방법에 대해서 제안한다. 구조 변경은 특정 값을 사전 연산하거나 또는 대규모로 연산할 때는 사전 연산 테이블을 활용하는 방법, 프로세서의 특성을 활용하여 원래 연산의 반대로 연산하는 방법 등이 있다. 구현 대상 알고리즘은 국산 경량 블록암호 CHAM, 경량 블록암호 후보 TinyJAMBU, 양자내성암호 후보 Rainbow를 대상으로 한다. 구현 대상 프로세서는 저사양 사물 인터넷 환경에서 주로 활용되는 8-bit AVR 프로세서와 AVR에 비해서는 상대적으로 고사양이며 주로 스마트폰과 최근에는 노트북까지 사용처가 넓어진 64-bit

ARM 프로세서이다. 제안하는 기법은 각 알고리즘의 특성과 프로세서의 환경을 고려하여 알고리즘의 성능을 향상시킬 수 있는 내부 구조 재설계를 진행한다.

【주요어】 블록암호, 양자내성암호, 최적 구현, 사물 인터넷 프로세서