Secure and Fast Implementation of ARX-Based Block Ciphers Using ASIMD Instructions in ARMv8 Platforms

JINGYO SONG\(^1\) AND SEOG CHUNG SEO\(^2\), (Member, IEEE) \(^1\)\(^2\)
\(^1\)Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea
\(^2\)Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, South Korea
Corresponding author: Seog Chung Seo (scseo@kookmin.ac.kr)
This work was supported by the Research Project on Efficient Implementation of Approved Algorithm, Korean Cryptographic Module Validation Program, through the Affiliated Institute of the Electronics and Telecommunications Research Institute (ETRI), under Grant 2020-060.

ABSTRACT In Internet of Things services, various types of embedded devices are employed. Among them, ARM-based devices have been widely used as clients. Since these devices communicate with each other in wirelessly, transmitted data needs to be protected with secure block ciphers. Recently, several Add-Rotate-XOR (ARX)-based block ciphers, such as HIGHT and revised CHAM, have been developed for efficient encryption on embedded devices. In this paper, we present secure and fast implementations of ARX-based block ciphers HIGHT and revised CHAM in ARMv8 platforms. For performance efficiency, we basically apply task and data parallel processing mechanism by fully utilizing NEON architecture embedded in ARMv8 platforms. Typically, it is required to duplicate round key in NEON register to utilize the NEON architecture to process multiple data blocks simultaneously. In our implementations, we propose an optimal approach minimizing the cost of round key duplication and efficient key scheduling for task parallelism. For secure implementation, we develop efficient software countermeasures against realistic fault attack models. Thus, we present efficient software countermeasure based on intra-instruction redundancy. Especially, we propose enhanced random shuffling method which is the core operation for the proposed countermeasure. With the proposed random shuffling method, we can significantly reduce the overhead for preventing fault attacks. We present two versions of the software: a version providing highly fast (HF) performance without fault attack countermeasures and a version providing highly secure (HS) against fault attacks. Compared with referenced software, HF with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 provides about 8 times, 38 times, 13 times and 13 times of enhanced performance, respectively. Compared with previous best results having fault attack countermeasure, HS with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 provides about 50%, 30%, 80%, and 70% of enhanced performance, respectively. Both our HS and HF achieve better performance and higher security compared with related works.

INDEX TERMS HIGHT, revised CHAM, lightweight cryptography, fault attack resistance, ARMv8, ASIMD, NEON, optimization, Internet of Things.

I. INTRODUCTION Owing to the development of Internet of Things (IoT) technology, more types of embedded devices are employed than ever before, and these devices communicate with each other to provide convenient services to users. With an increasing scale of information communicated by IoT devices, the importance of technology to prevent accidents caused by exposure of information has been emphasized. Thus, it is necessary to encrypt data with a cryptographic algorithm when communicating between IoT devices to protect data. However, as the size of embedded devices is reduced, CPU resources, memory, power, etc. are limited unlike high-end computers, so it is difficult to apply the existing cryptographic algorithm in embedded devices. For this reason, several lightweight block ciphers, such as LEA, HIGHT, CHAM, SPECK, and SIMON, have been developed. HIGHT [1] was adopted as an ISO/IEC international standard in
2010 [2], so its effective was verified as suitable lightweight cryptography for limited environment. Also, revised CHAM [3] is not only similar in performance to SPECK, one of the best-performing lightweight block ciphers in software, but was verified to the security against differential attacks by increasing the number of rounds to CHAM block cipher. Thus, we select two lightweight block ciphers, HIGHT and revised CHAM. However, these algorithms still require large amount of computations compared to other tasks. So it is necessary to enhance the performance of the block cipher algorithms considering the characteristics of target devices in which the cryptographic algorithms are implemented.

When the cryptographic algorithm is performed, embedded devices unintentionally emit additional information such as power consumption, electro-magnetic waves, and timing information. Side channel analysis can effectively obtain cryptographic keys in the devices by exploiting these emitted information. There are mainly two kinds of side channel analysis: passive one and active one. Passive analysis makes use of the additional information unintentionally emitted from the devices. Attackers with active analysis can obtain cryptographic keys by injecting faults into the device and analyzing the faulty results from the device. Recently, fault injection attacks have become practical attack methods in which an attacker physically injects faults into the embedded devices to find the secret key. Since 1997, Biham and Shamir analyzed DES algorithm by applying the fault injection attack for the first time [4], various fault attacks have been applied to various block ciphers such as AES [5], [6], ARIA [7], SEED [8], LEA [9], HIGHT [10], and CHAM [11]. These literatures show that attackers can recover the secret key by injecting faults during cipher execution on embedded devices. Therefore, it is essential to add fault attack countermeasures when implementing cryptographic algorithms on embedded devices.

Until now, several fault attack countermeasures have been proposed. In SAC’16, C. patrick et al. proposed fault attack countermeasure for secure AES implementation on a simulator of a 32-bit SPARC processor called LEON3 [12]. Proposed countermeasure was fault detection mechanism based on intra-instruction-redundancy (IIR), known ciphertext (KC) slices and random-shuffling. In WISA’17 [13], Seo et al. presented countermeasure against fault attack for secure LEA implementation on NEON architecture. That work detected efficiently faults through IIR, and proposed a random-shuffling method that the previous work failed to describe how to generate the random-shuffling using a random bit. In ICISC’19, Seo et al. introduced an optimized implementation of HIGHT equipped with a fault attack countermeasure on ARM Cortex-M4 [14]. They presented a optimization technique with task and data parallelism using single instruction multiple data (SIMD) instruction set and proposed a fault attack countermeasure using IIR and random-shuffling method.

The aforementioned efficient implementations of block ciphers with fault attack countermeasures have focused only on the ARM Cortex-M4 and ARMv7. Recently, since the ARMv8-based IoT devices have been being widely used, it is necessary to develop efficient block cipher implementations providing resistance against fault attacks on ARMv8-based devices. In this paper, we provide two versions of HIGHT and revised CHAM software implementations, the first version is a highly fast (HF) version without fault attack countermeasure focusing on only the efficiency, and the second version is a highly secure (HS) version with efficient software countermeasures against fault attacks. Our proposed HF and HS show much improved performance and security than previous works. Thus, this work achieves the best performance and security for protecting sensitive data when communicating wirelessly in widely used ARMv8-based IoT devices.

The contributions of this work can be summarized as follows.

- **First secure and efficient implementations of HIGHT and revised CHAM on ARMv8 platforms**

Until now, there exists only on implementations of lightweight cryptography for ARMv7 or ARM Cortex-M4 environment. ARMv8 processor has become popular MCU in IoT devices. Therefore, we present the first optimized implementation of HIGHT and revised CHAM ciphers, and efficient software countermeasure against fault attacks for lightweight block ciphers in the ARMv8 platforms.

- **Optimized implementation of HIGHT and revised CHAM**

We present optimized implementation of ARX-based block ciphers (HIGHT and revised CHAM) through advanced single instruction multiple data (ASIMD) instruction set in ARMv8 platforms. The following techniques are used to optimized implementation. First, we apply task and data parallelism in encryption process, so multiple data and operation are performed simultaneously. Second, we make it possible task parallelism to be applied automatically without overheads. For this technique, we use LDN and STN instructions when loading and storing the words of the plaintext into the vector register. Finally, we reduce the number of memory accesses by accessing the memory with the maximum number of registers. Also, since multiple data is performed at once in the encryption process, the round key should be duplicated. So we present minimizing the cost of round key duplication and efficient key scheduling for task parallelism. As a result, our optimized implementations for HIGHT, revised CHAM-64/128, CHAM-128/256, and CHAM-128/256 are shown to be about 8 times, 36 times, 13 times, and 13 times better than the reference implementations.

- **Countermeasure against fault attacks for HIGHT and revised CHAM**

Until now, no effective fault attack countermeasures have been proposed on ARMv8 even though some countermeasures such as [13] were proposed on ARMv7.
Thus, we present efficient fault attack countermeasure by using IIR, KC slices, and a fault checking method on ARMv8. In addition, to make it difficult for an adversary to inject faults, random-shuffling is applied to each round. However, the existing random-shuffling method from [13] is costly, so we propose a new random-shuffling which can be efficiently performed on the target device. The proposed random-shuffling is faster and requires fewer registers than the existing random-shuffling. Our fault attack countermeasure with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 is about 50%, 30%, 80%, and 70% faster than that the previous approach, respectively. Furthermore, even though the fault attack countermeasures are applied, our implementation of fault attack countermeasure for HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 shows an improved performance of about 100%, 130%, 230%, and 190%, respectively, over the reference implementations without the fault attack countermeasures.

The rest of this paper is organized as follows. Section II gives an overview of the target algorithm: HIGHT and revised CHAM with respect to key scheduling process and encryption process, and ASIMD instruction sets used in our proposed implementation. Section III introduces previous works about HIGHT and revised CHAM implementations on ARM processor, fault attacks on block ciphers with fault attack model, and fault attack countermeasures. Section IV proposes optimized implementations of HIGHT and revised CHAM with a novel fault attack countermeasure based on IIR and a new random shuffling method on ARMv8 NEON architecture. Section V compares the performance of the proposed implementations with previous works. Finally, section VI concludes this paper with future works.

II. BACKGROUND

A. ARX-BASED KOREAN BLOCK CIPHERS

1) HIGHT

HIGHT is a lightweight block cipher developed jointly by the Korea Internet & Security Agency (KISA), Electronics and Telecommunications Research Institute (ETRI) and Korea University in 2005 [1]. Since it does not use a substitute like S-box, it is suitable for mobile environments requiring low power and simple arithmetics such as byte-wise arithmetic operations to be applied in limited environments. In addition, it was adopted as an ISO/IEC international standard in 2010 [2], so its effectiveness was verified as a suitable lightweight cryptography for limited environments.

HIGHT is a transformed Feistel Add-Rotate-XOR (ARX) structure that encrypts 64-bit plaintext with 128-bit secret key. The total round comprises the initial conversion, round function, and final conversion. The round function is repeated 32 times in total. The key scheduling generates a 136-byte round key to be used in the encryption process with a 128-bit secret key. In the key scheduling, through a linear feedback shift register (LFSR), a variable called delta is constantly updated to generate the each round key. The initial concatenated polynomial of the LFSR is \( x^7 + x^3 + 1 \), and since it is a primitive polynomial at \( F_2 \), the period of the LFSR is 127 in total. In addition, the internal state value of the initial LFSR is set as \( (1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0) \). All round key are generated to modular addition by 2^8 of the updated delta value through the LFSR and master key, and the remaining eight round keys use the master key as it is.

HIGHT round function consists of modular addition by 2^8, F0 function, F1 function, and an XOR operation. The F functions consist of a rotate shift left and an XOR operation. In the HIGHT round function, only 4-byte out of 8-byte perform the F functions. When the round function is completed, it circularly rotates to the left 1-byte. The above process is repeated 31 times in total. In the last round, no rotate shift left is applied to the result values. When all 32 round functions are completed, the final conversion is performed, and then the ciphertext is output. Fig. 1 shows one round of the HIGHT encryption.

2) REVISED CHAM

In ICISC’17, Koo. et al. published CHAM lightweight crypto family [15]. CHAM family is based on ARX structure and is divided into CHAM-64/128, CHAM-128/128, and
CHAM-128/256 according to their parameters. Table 1 shows the parameters of the CHAM family. CHAM is a stateless on the fly structure that does not store the key state. In addition, the operation is an ARX structure, so it is a lightweight cryptography suitable for limited environments.

In ICISC’19, revised CHAM, a version executing the increased number of rounds, was proposed to ensure sufficient security about differential characteristics detected to the previous version of CHAM [3]. Revised CHAM family are increased to the number of rounds from 80 to 88, from 80 to 112, and from 96 to 120 for revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 to ensure security against differential attacks. Even though the number of rounds has increased, revised CHAM shows fast performance in hardware and software. On the hardware, it shows no performance difference from original CHAM. In addition, it performs better than SIMON in software and is similar to SPECK [16], which shows one of the best performance in software.

In the key scheduling of revised CHAM, the secret key, $ROL_1$, $ROL_8$, $ROL_{11}$, and $XOR$ operations generate $2 \times k/w$ round keys ($ROL_i$ means bitwise rotate left as $i$ bits). The $2 \times k/w$ round keys are repeatedly used to encryption process. CHAM encryption consists of odd and even rounds. Fig. 2 shows the structure of the odd and even round functions. The each round function of CHAM consists of $ROL_1$, $ROL_8$, modular addition by $2^8$, and $XOR$ operations. When a round is completed, a rotate left shift is applied to the result in word units.

B. ARMV8 MICROCONTROLLERS WITH ASIMD INSTRUCTIONS SETS

1) ARMV8 ARCHITECTURE

ARM Cortex-A series is a high-performance MCU typically employed in tablets and mobile phones. ARMv8 provides a 64-bit environment, but ARM series prior to ARMv8 provided a 32-bit environment [18]. ARMv8 was released in 2011 and it provides AArch32 execution state, which is a 32-bit operating system environment compatible with previous versions, and AArch64 execution state, which supports a new 64-bit operating system environment. In the instruction set, AArch32 execution state provides A32, Thumb, and Thumb2 previously provided for backward compatibility, and AArch64 execution state provides A64 and ASIMD that can be processed in parallel called NEON architecture. As the version was upgraded to ARMv8, the size of 32-bit register supported by ARMv7 was extended to 64-bit register. In addition, the number of registers in ARMv8 was increased from 16 to 31 compared to ARMv7. Also, it supports optional Cryptography Extension, which improves performance over ARMv7 when encrypting and decrypting. The specifications of our board are ARM-Cortex-A72 as MCU [19], and it supports four quad cores. It also supports 4GB of internal memory space.

2) ASIMD INSTRUCTION SETS

Since ARMv7, Single Instruction Multiple Data (SIMD) called NEON [17], which can process multiple data in parallel, was supported. With SIMD instructions, multiple data can be efficiently processed with one instruction at a time. Also, the parallel control of 128-bit vector...
TABLE 2. Summary of ASIMD Instruction set on ARMv8 NEON architecture [17] (cc: Clock Cycles).

| Instruction | Operands | Operation | cc |
|-------------|----------|-----------|----|
| MOV         | V_m, V_n | Copy vector register from V_n to V_m, V_m ← V_n | 5  |
| ADD         | V_d, V_m, V_n | Add two vector registers, V_d = V_m + V_n | 5  |
| EOR         | V_d, V_m, V_n | XOR two vector registers, V_d = V_m ⊕ V_n | 5  |
| SHR         | V_m, V_n, #n | Shift Left (immediate), V_m ← V_n(#n) | 5  |
| SHR         | V_m, V_n, #n | Shift Right and Insert (immediate) | 5  |
| REV         | V_m, V_n | Reverse vector elements in word halfwords | 5  |
| TRN1        | V_d, V_m, V_n | Transpose vectors (primary) | 5  |
| TRN2        | V_d, V_m, V_n | Transpose vectors (secondary) | 5  |
| TBL         | V_d, {V_m}, V_n | Table vector Lookup | 5  |
| UZP1        | V_d, V_m, V_n | Unzip vectors (primary) | 5  |
| UZP2        | V_d, V_m, V_n | Unzip vectors (secondary) | 5  |
| DUP         | V_m, X_n | Duplicate vector register to general-purpose register | 5  |
| LD4         | {V_n - V_m}, [ ] | Load multiple 4-element structures to four registers {V_n - V_m} | 8  |
| ST4         | {V_n - V_m}, [ ] | Store multiple 4-element structures from four registers {V_n - V_m} | 8  |

register is available in units of 8-bit, 16-bit, 32-bit and 64-bit. In this paper, vector registers are controlled in parallel as eight bit-wide for HIGHT, 16 bit-wide for revised CHAM-64/128, and 32 bit-wise for revised CHAM-128/256 and CHAM-128/256. In addition, ARMv8 supports ASIMD which is improved SIMD over the previous SIMD. Table 2 summarizes ASIMD instructions and clock cycle (cc) of each instruction. Since ARMv8 supports 5-stage pipeline, it can be seen that instructions related to operation between registers consume 5 clock cycles, while instructions related to memory access consume more clock cycles than instructions related to operation between registers.

### III. RELATED WORKS

#### A. PREVIOUS IMPLEMENTATION OF ARX-BASED KOREAN BLOCK CIPHERS ON ARM-NEON

NEON is an instruction architecture that can process operations in parallel. Since ARMv7, there has been a lot of researches on crypto optimization while supporting NEON architecture. In WISA [20], [21], the authors presented optimized implementation for LEA and CHAM-64/128 by processing multiple data in parallel via the NEON architecture. On Cortex-A9 and Raspberry Pi 3B boards, multiple plaintexts were processed in parallel through the ARM’s instruction set and NEON architecture, respectively. By using four multi-cores through OpenMP, the shift operation on each core was optimized using the ARM’s barrel shifter.

In ICISC’19 [14], the authors provided an optimization and a fault attack countermeasure for HIGHT through the ARM’s instruction set and SIMD instruction set on ARM Cortex-M4. For optimization, data and task parallelism were applied in the HIGHT encryption process. In addition, For a fault attack countermeasure, an IIR and a random-shuffling method were applied to detect the computation faults and to make it difficult for an adversary to inject faults. ICISC’19 presented AES optimized implementation using ARMv8 NEON architecture [22]. ARMv8 supports Cryptography Extension, so that fast encryption and decryption are possible, but current work proposed AES implementation faster than Cryptography Extension by using only ASIMD instructions. In this paper, by using the ARMv8 NEON architecture on Raspberry Pi 4B board, we present optimized implementation for HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 by efficiently applying task and data parallelism. In addition, we provide security against fault attacks through IIR, KC slices, and random-shuffling.

#### B. FAULT ATTACKS ON BLOCK CIPHERS AND COUNTERMEASURES

1) FAULT ATTACK MODEL

In this section, we describe the identical fault attack model employed in previous works to evaluate our proposed countermeasures against fault attack [12], [13]. Fault attack, one of the active side-channel attacks, injects faults into an embedded device in which cryptography is in motion to randomly change the value of specific words in the program or change the flow of the program. Basically, we can divide faults into computation faults and instruction faults. First, the computation faults are to randomly change a specific word in the program by injecting a fault. The higher the assumption of the attacker, the more precise the fault injection is possible. Computation faults can be divided into four faults according to the size of the fault injection, and the four computation fault models are as follows [12], [13].

-Random word: The adversary can target a specific word in a program and change its value into a random value unknown to the adversary.
- **Random byte**: The adversary can target a specific word in a program and change a single byte of it into a random value unknown to the adversary.

- **Random bit**: The adversary can target a specific word in a program and change a single bit of it into a random value unknown to the adversary.

- **Chosen bit pair**: As the attack with the highest assumption of the adversary, the adversary can target a chosen bit pair of a specific word in a program, and change it into a random value unknown to the adversary.

Second, the instruction faults are the fault attack that changes the flow of a program by injecting faults. A typical example is instruction skip by changing the opcode for a specific instruction to nop(no-operation).

2) **PREVIOUS FAULT ATTACK AND COUNTERMEASURES ON BLOCK CIPHER**

The fault injection attack on block ciphers has been studied with various fault injection models. The most common attack is the differential fault analysis (DFA). DFA is a combination of a differential cryptanalysis attacking a block cipher and a fault injection attack of a side-channel analysis. The point of the DFA is how to reality inject a fault and if the bit length of fault injection attack of a side-channel analysis. The point of a differential cryptanalysis attacking a block cipher and a fault to the last round of CHAM. It is possible to find four pairs of cipher-texts and incorrect ciphertexts. In addition to these, there are DFA studies on various block ciphers.

Since the secret key of the block ciphers can be recovered through fault injection, when implementing the block cipher, the countermeasure against fault attack should be applied. The representative countermeasures against fault attack are fault detection-based and injection-based. We provide security to fault attack by targeting the fault detection-based. The classical technique is instruction duplication and triplication [23], which applies encryption several times through time redundancy to detect faults by consistently comparing ciphertext with redundancy ciphertext. However, this technique has a performance overhead of 3.4 and 10.6 times, respectively, and it has a disadvantage that it can be easily bypassed for an adversary who consistently injects faults into original data and redundant data. Another fault detection-based technique is to use information redundancy, such as parity bits or additional check variables, to detect faults in data [23]. However, they have a performance overhead of 3.5 to 4.7.

In WISA’17 [13], The authors proposed an efficient fault attack countermeasure for secure implementation of LEA by applying IIR and KC slices in NEON architecture. The previous bit-slicing structure was changed to a 32-bit word-wise to secure the chosen bit pair. In addition, the author described a random-shuffling utilizing the NEON architecture, but the previous work [12] did not describe how to make random-shuffling. However, since it performs a random-shuffling each round, it takes a lot of overheads. In this paper, we propose an efficient random-shuffling method using faster and fewer registers than previous random-shuffling method.

IV. PROPOSED SECURE AND EFFICIENT IMPLEMENTATIONS ON ARMV8 WITH ASIMD INSTRUCTIONS

A. PROPOSED OPTIMIZATION APPROACHES FOR ARX-BASED KOREAN BLOCK CIPHERS

1) **COMMON OPTIMIZATION TECHNIQUE**

In this section, we introduce the common optimization ideas applied in the optimized implementations of HIGHT and revised CHAM-64/128, CHAM-128/128, and CHAM-128/256. Our common optimization ideas are as follows. First, since the vector register has an interval of 128-bit, we fully utilize 128-bit of the vector register to efficiently process multiple data. Second, to apply task parallelism, words that perform the same operation in plaintext should be loaded into the same vector register. So, we make it possible loading the words of plaintext blocks into vector registers so that task parallelism can automatically be achieved without overheads through LDN and STN instructions. Finally, we reduce the number of memory access by accessing the memory with the maximum number of registers.

In encryption process, the data and task parallelism are effectively applied. The data parallelism processes multiple data simultaneously, and the task parallelism processes multiple operations simultaneously. So, HIGHT processes eight encryptions simultaneously, revised CHAM-64/128 processes four encryption simultaneously, and revised CHAM-128/128 and CHAM-128/256 processes two encryptions simultaneously. In the key scheduling, after loading master key into one vector register, task parallelism is applied to process multiple operations at once. Furthermore, since
TABLE 3. Vector register scheduling about optimized implementation and a fault attack countermeasure for HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256.

| Cipher     | Method                   | Register Scheduling |
|------------|--------------------------|---------------------|
| HIGHT      | Optimization             | v0-v3: PT           |
|            |                          | v4-v7: TEMP          |
|            |                          | v8-v11: Round key    |
| Fault      | Counterm easure          | v0-v3: PT, RT       |
|            |                          | v4-v7: TEMP          |
|            |                          | v8-v11: Round key    |
|            |                          | v12-v15: PT, RT      |
|            |                          | v16-v19: TEMP        |
|            |                          | v30: Random Table    |
| Revised CHAM| Optimization          | v0-v1: PT           |
|            |                          | v2-v4: TEMP          |
|            |                          | v5-v8: Round key     |
| Fault      | Counterm easure      | v0-v1: PT           |
|            |                         | v2-v4: TEMP          |
|            |                         | v5-v8: Round key     |
|            |                         | v9-v10: PT, RT       |
|            |                          | v11-v14: TEMP        |
|            |                          | v30: Random Table    |

Algorithm 1: HIGHT Key Schedule and Round Key Duplication

Require: v0(Masterkey), v9(Delta)  
Ensure: v5,v6,v7,v9(Duplication Round Key)

1. ADD v1.16b, v0.16b, v9.16b
2. DUP v2.16b, v1.b[0]
3. Duplication Round Key #1
4. TRN1 v5.16b, v3.16b, v2.16b
5. DUP v2.16b, v1.b[1]
6. Duplication Round Key #2
7. TRN1 v5.16b, v3.16b, v2.16b
8. DUP v2.16b, v1.b[4]
9. Duplication Round Key #3
10. TRN1 v5.16b, v3.16b, v2.16b
11. DUP v2.16b, v1.b[5]
12. DUP v3.16b, v1.b[7]
13. Duplication Round Key #4
14. ST1 {v5.16b-v8.16b},[x1],#64

2) HIGHT OPTIMIZATION TECHNIQUES

a: KEY SCHEDULE PROCESS

In the key scheduling process of HIGHT block cipher, 1-byte of delta updated by LFSR and 1-byte of the master key performs a modular addition by $2^8$ operation to generate a 1-byte round key, and it is repeated 128 times in total to generate all round key. Since all operations are performed independently, we present a method to apply task parallelism to one register to generate multiple round keys at once. The proposed key scheduling method loads the master key into vector registers. After the delta is precomputed as the table, the delta table is loaded to the eight vector registers. By applying NEON’s ADD instruction to process the modular addition by $2^8$ in parallel on a byte-wise, the modular addition by $2^8$ operation to generate all round keys of HIGHT is effectively reduced from 128 operations to eight operations. After generating all round keys, a round key duplication process is required to match the task and data parallelism applied to encryption process. That is, the round key should be a duplication such as (SK0, SK4, SK0, SK4, ..., SK0, SK4).

Algorithm 1 shows the HIGHT’s key scheduling and round key duplication. In Step 1, a 16-byte round key are generated through task parallelism at once. In Step 2-3, the necessary round keys for encryption process are respectively duplicated. In Step 4, the duplicated round keys are transposed to apply task parallelism. To reduce memory access, we repeat the above process to three more times, and then stored from registers to memory at once. After repeating the above process eight times, all the round keys are stored in the memory.

b: ENCRYPTION PROCESS

To optimize HIGHT encryption process, we introduce task and data parallelism. The words of the 0, 4th index, 1,5th index, 2, 6th index, and 3, 7th index perform the same operation in the round function of HIGHT. Thus, we store the words of the 0, 4th index, 1, 5th index, 2, 6th index, and 3, 7th index to four registers to apply task parallelism respectively. In that case, when operating the 0, 1, 2, and 3rd indexes, the values of the 4, 5, 6, and 7th indexes are operated at the same time, and the overheads of an operation can be halved. Fig. 3 shows the structure of the vector register in HIGHT encryption process. Since HIGHT operates in units of 8-bit words, a maximum of eight plaintexts can be stored when considering task parallelism to a 128-bit vector register. Thus, in the HIGHT encryption process, eight encryptions are processed simultaneously with one encryption.

Algorithm 2 shows one round encryption of HIGHT with the data and task parallelism. Step 1 is the process of loading the round key from memory to a vector register. Since LD4 is an instruction that takes a lot of load and task parallelism has already been applied to the round key, the round key is loaded from memory into the vector register through the LD1 instruction, which is faster than the LD4 instruction. Through Step 1, It loads duplication round keys of two rounds into the vector register at a time. Step 2-9 is a F1 function. In NEON architecture, when implementing a rotate shift,
it can be implemented through SHL and SRI instructions, and it is a frequently used instructions to implement the rotate shift. Step 10-11 and Step 20-21 are round function. Step 12-19 is F0 function. Finally, Step 22 performs an 8-bit rotate left shift on the result of the round function. HIGHT performs a left rotate shift by 8-bit after each round function. We easily implemented the left rotate shift by using the REV instruction. After repeating the above process 32 times, since task parallelism is applied to the encryption process, the result value is stored from the register to memory through the ST4 instruction.

Algorithm 2
HIGHT i-Th Round Function Data, Task Parallel Implementation (i ∈ [1, 32], PT: PlainText, RK: Round Key, CT: Ciphertext)

Require:
- v0-v3(PT), v8-v11(RK)

Ensure:
- v0-v3(CT)

1: LD1{v8.16b-v11.16b}, [x2],#64
2: SHL v4.16b,v3.16b,#3
3: SRI v4.16b,v3.16b,#4
4: SHL v4.16b,v3.16b,#5
5: SRI v4.16b,v3.16b,#4
6: EOR v6.16b,v4.16b,v5.16b
7: SHL v4.16b,v1.16b,#7
8: SRI v4.16b,v1.16b,#1
9: EOR v4.16b,v4.16b,v6.16b
10: EOR v4.16b,v8.16b,v4.16b
11: ADD v2.16b,v4.16b,v2.16b
12: SHL v4.16b,v1.16b,#1
13: SRI v4.16b,v1.16b,#7
14: SHL v5.16b,v1.16b,#2
15: SRI v5.16b,v1.16b,#6
16: EOR v6.16b,v4.16b,v5.16b
17: SHL v4.16b,v1.16b,#7
18: SRI v4.16b,v1.16b,#1
19: EOR v4.16b,v4.16b,v6.16b
20: ADD v4.16b,v9.16b,v4.16b
21: EOR v0.16b,v6.16b,v0.16b
22: REV16 v0.16b, v0.16b

Algorithm 3 shows the key scheduling and round key duplication process of revised CHAM-64/128. Step 1-2 and Step 3 process the ROL1 and ROL8 operations. Since revised CHAM-64/128 is in a 16-bit unit, ROL8 can be effectively implemented as REV instruction instead of SHL and SRI instructions. Step 4-5 performs the ROL11 operation. Step 6 performs an XOR operation on the result of ROL1 and ROL8. Through Step 7, half of the round keys is generated at once. In Step 8, instead of performing an XOR operation on ROL1 and the master key, by using the previous result, the remaining round keys are generated. Step 9-20 efficiently duplicate round keys to apply data and task parallelism in encryption process. The procedure is the same as HIGHT, and the word of revised CHAM-64/128 is 16-bit, so only the lane is changed to 16-bit.

b: ENCRYPTION PROCESS

In this section, we present the optimized implementation that applies data and task parallelism to revised CHAM-64/128,
Algorithm 3 Revised CHAM-64/128 Key Schedule and Round Key Duplication

| Require: | \(v_0\) (Master key)  |
| Ensure:  | \(v_6,v_7,v_8,v_9\) (Duplicated Round Key) |

| Round key Duplication | \#1 |
|-----------------------|-----|
| 1: SHL \(v_1.8h,v_0.8h,\#1\)     |
| 2: SRI \(v_1.8h,v_0.8h,\#15\)   |

| Round key Duplication | \#2 |
|-----------------------|-----|
| 3: REV16 \(v_2.16b,v_0.16b\) |

| Round key Duplication | \#3 |
|-----------------------|-----|
| 4: SHL \(v_3.8h,v_0.8h,\#11\) |
| 5: SRI \(v_3.8h,v_0.8h,\#15\) |

| Master key \(\oplus\) ROLL1 | 6: EOR \(v_3.16b,v_1.16b,v_0.16b\) |
|-------------------------------|----------------------------------|

| Master key \(\oplus\) ROLL1 \(\oplus\) ROLL8 | 7: EOR \(v_4.16b,v_5.16b,v_2.16b\) |
|-------------------------------------------|----------------------------------|

| Master key \(\oplus\) ROLL1 \(\oplus\) ROLL11 | 8: EOR \(v_5.16b,v_5.16b,v_3.16b\) |
|---------------------------------------------|----------------------------------|

\[
\text{FIGURE 4. Revised CHAM-64/128 for encryption vector register setting.}
\]

| v0.8h | PT1[0] | PT1[2] | PT2[0] | PT2[2] | PT3[0] | PT3[2] | PT4[0] | PT4[2] |
|-------|--------|--------|--------|--------|--------|--------|--------|--------|
| v1.8h | PT1[1] | PT1[3] | PT2[1] | PT2[3] | PT3[1] | PT3[3] | PT4[1] | PT4[3] |

CHAM-128/128, and CHAM-128/256. Revised CHAM consists of odd and even rounds. When the round function is completed, a rotate left shift is applied in each round word unit, and after the fourth round is completed, it returns to the original word position. In addition, round 1, 3 and 2, 4 rounds perform the same round function operation. Through these characteristics, we efficiently apply data and task parallelism to process 1, 3 and 2, 4 rounds simultaneously based on the four rounds. So, four encryptions are processed at once.

Fig. 4 and 5 show the configuration of vector registers for revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 applying data and task parallelism. Revised CHAM-64/128 consists of 16-bit words, while revised CHAM-128/128 and CHAM-128/256 consist of 32-bit words. We describe based on revised CHAM-64/128, and for revised CHAM-128/128 and CHAM-128/256, only the word unit, the number of round keys, and composition of ROL8 are different, but the rest are the same as revised CHAM-64/128.

Algorithm 4 shows the four round encryption process in revised CHAM-64/128. Before the encryption starts, the plaintext should be loaded from memory into a register with the LD2 instruction, and the counter should be set to enable the task parallelism. Step 1 is for fetching the second unencrypted word value to match the inputs of the 1st and 3rd rounds. In Step 2, plaintext and counter perform the XOR operation. Step 3-4 performs ROL1, and Step 5 performs an XOR operation on the result of ROL1 and the round key. In Step 6, After performing the modular addition by 2^8 on the above result and the value which was performed an XOR operation on a counter, the 1, 3 round encryption complete by performing the ROL8 operation in Step 7. The input for rounds 2, 4 are matched through Step 8. Step 9 performs an XOR operation with counter, and Step 10 performs ROL8 operation. In Step 12, the above result and the value which is performed an XOR operation with a counter are performed a modular addition by 2^8. After ROL1 operation is performed via Step 13-14, encryption of the round 2 and 4 completes, and finally the counter value is increased by 4 through Step 15-16. A total of 22 repetitions of the above process will complete encryption.

B. PROPOSED FAULT ATTACK COUNTERMEASURES WITH INTRA-INSTRUCTION REDUNDANCY

1) OVERALL DESIGN

In this section, we describe a fault attack countermeasure and the overall process of a fault attack countermeasure...
for HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256. For the security for fault attacks, we apply IIR to detect computation faults. However, if the same fault is injected into the original data and redundant data, the same ciphertext can be generated and the above countermeasures can be bypassed. Thus, we add the known plaintext for the known plaintext and ciphertext pairs to the vector register to detect the instruction faults. In addition, random-shuffling is applied according to a random bit to make it difficult for an adversary to target a specific word in a program. Since we use the same countermeasure as the previous work, the whole process of the countermeasure is also the same as the previous work [13].

Fig. 6 shows the overall process of a fault attack countermeasure. The first step loads messages from the memory into a register. Then, to apply the IIR, the messages are duplicated. After the messages are duplicated, random-shuffling is applied to the messages and the round key based on the random bit before the round function proceeds. HIGHT processes random-shuffling for each round, and since revised CHAM processes multiple rounds simultaneously, random-shuffling of revised CHAM is applied to every four rounds at once. After the round function is completed, the encrypted data is sorted to the correct position through the last-random-shuffling. Finally, faults are checked and if faults aren’t injection, the ciphertext is stored in memory and ciphertext is returned.

2) PROPOSED RANDOM SHUFFLING

In this section, we introduce a new random-shuffling method. In the proposed random-shuffling, Algorithms 5, 6, and 7 are set based on HIGHT. In the case of a revised CHAM, by changing the lane according to the word size of the plaintext, it can be easily applied to these algorithms. Masking was employed in the previous random-shuffling method [13], but since we simply implement the random-shuffling by looking up the random table efficiently using TBL instruction, the number of registers required is less than that in the previous random-shuffling and much more efficient in terms of performance. The proposed random-shuffling is applied to all 8-bit, 16-bit, 32-bit, and 64-bit word units like the previous random-shuffling. When the random bit is 1, a swap is applied, but when the random bit is 0, a swap is not applied.

Algorithm 5 shows a new random-shuffling method. The v30 vector register stores random table value, and the initial

---

**Algorithm 4** Revised CHAM-64/128 Four Round Function Data, Task Parallel Implementation, PT: PlainText, RK: Round Key, CTR: Counter, CT: Ciphertext

**Require:** v0-v1(PT), v21-v28(RK), v29,v30(CTR)

**Ensure:** v0-v1(CT)

1. Pre-preparation for 2, 4 round inputs
   - REV32 v4.8h,v0.8h
   - EOR v2.16b,v30.16b,v0.16b
   - SHL v3.8h,v1.8h,#1
   - SRI v3.8h,v1.8h,#15
   - ROL1 operation
   - EOR v3.16b,v3.16b,v25.16b
   - ADD v2.8h,v2.8h,v3.8h
   - ROL8 Operation
   - ADD v29.8h,v29.8h,v20.8h
   - ADD v30.8h,v30.8h,v5.8h

**Algorithm 5** Proposed Message and Round Key Random Shuffling, Where w5: Random Bit, w6: Current State, v0-v3: Plaintext v30: Random Table, v8-v9: Roundkey

1. Extracting the Random Bit
   - AND w7,w5,#1
   - ROR w5,w5,#1
   - EOR w6,w6,w7

2. Generating Random Table
   - DUP v5.16b,w7
   - SHL v5.16b,v5.16b,#1
   - EOR v6,v6,v7

3. Looking up Random Table
   - TBL v0.16b,{v0.16b},v5.16b
   - TBL v1.16b,{v1.16b},v5.16b
   - TBL v2.16b,{v2.16b},v5.16b
   - TBL v3.16b,{v3.16b},v5.16b
   - TBL v6.16b,v6.16b
   - TBL v7.16b,v7.16b

4. Add 4 to Counter
   - ADD v29.8h,v29.8h,v20.8h
   - ADD v30.8h,v30.8h,v5.8h
value is \((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)\). In Step 1, 1-bit of the random bit is obtained. In Step 2, the next random bit is prepared by applying a 1-bit rotate right shift to the random bit. In Step 3, the current random bit value is continuously accumulated to apply random-shuffling to the round key. In Step 4, 1-bit from the random bit is duplicated in the vector register in byte units. In Step 5, a 1-bit left-shift operation is performed in byte units of the vector register to which duplication is applied. In Step 6, if the random bit is 0, the v6 register retains the initial value, and if the bit is 1, the v6 register stores the swap table value. Step 7-9 generate a random table for round key random shuffling though the accumulated values. In Step 10-13, random-shuffling is applied to each vector register by looking up the random table generated in Step 4-6. In Step 14-15, random-shuffling is applied to the round key via a random table generated from the accumulated value in Step 7-9.

After encryption is complete, the data applied random-shuffling in each round requires a last-random-shuffling to return the correct ciphertext. Algorithm 6 shows the last-random-shuffling. In Step 1, the current state value accumulated so far is duplicated in the register. In Step 2, a random table to apply the random-shuffling is generated. In Step 3-6, the encrypted data is returned to the correct position by looking up the random table.

Algorithm 6 Proposed Last Random Shuffling, Where w6: Current State, v0-v3: Plaintext, v30: Random Table

| Generating Random Table | Looking up Random Table |
|-------------------------|-------------------------|
| 1: DUP v5.16b,w5         | 3: TBL v0.16b,v16b,v5.16b |
| 2: EOR v5.16b,v5.16b,v30.16b | 4: TBL v1.16b,v16b,v5.16b |
|                         | 5: TBL v2.16b,v16b,v5.16b |
|                         | 6: TBL v3.16b,v16b,v5.16b |

After applying last-random-shuffling, we should determine to return the ciphertext by checking whether there was an error injection attack by an aggressive attacker during encryption. In WISA’17 [13], the authors proposed a fault checking method based on the ARMv7 NEON architecture. In ICISC’19 [14], the authors proposed a fault checking method based on the ARM instruction set. However, in the ARMv8 NEON architecture, there is a difference in instruction set from the ARMv7 NEON architecture, so it is difficult to apply the previous fault checking method. Therefore, we propose a fault checking method based on the ARMv8 NEON architecture.

The proposed fault checking method is the Algorithm 7. In Step 1, only the encrypted data is extracted from the vector register through UZP1 instruction. In Step 2, only redundant data is extracted from the vector register through UZP2 instruction. In Step 5-7, an XOR operation is performed between the redundant data and the actual encrypted data value to check for computation fault. Through Step 8-11, the KC slices are extracted from the register that stores the encrypted data. The KC slices are aligned in a correct order using TBL instruction via Step 12. In Step 13, the instruction fault is detected by comparing the KC slices with the known ciphertext. Finally, in Step 14, all the resultant values perform an XOR operation to check the fault attack during encryption. If no faults are injected in encryption process, the result value is 0 and the ciphertext is returned.

Algorithm 7 Proposed Fault Attack Check Routine

| Require: v0-v3(PlainText), v11(KC: Known Ciphertext), v12(Correct Position Table) | Ensure: v4 (Fault Attack Check) |
|-----------------------------------------------------------------------------------|----------------------------------|
| 1: DUPLICATE v5.16b, v5.16b, v5.16b, v30.16b                                    | 8: MOV v10.h[0], v5.h[3]         |
| 2: EOR v5.16b, v5.16b, v5.16b, v6.16b                                           | 9: MOV v10.h[1], v5.h[7]         |
| 3: TBL v0.h[0], v0.h[16], v16b, v5.h[16]                                        | 10: MOV v10.h[2], v7.h[3]        |
| 4: EOR v5.16b, v5.16b, v6.16b                                                    | 11: MOV v10.h[3], v7.h[7]        |

3) IMPLEMENTATION OF FAULT COUNTERMEASURE FOR HIGHT

In this section, we present the implementation of fault attack countermeasures in HIGHT using the NEON architecture. Our implementation applies IIR and KC slices to effectively detect faults. The previous work [14] presented the HIGHT implementation of fault attack countermeasures in the target ARM Cortex-M4 platform using the ARM and SIMD instruction sets, but it was unable to detect the instruction faults due to insufficient space in the register. Since our target platform supports 128-bit vector registers, we detect efficiently the instruction faults by adding a known plaintext for known plaintext and ciphertext pairs to the vector register. Furthermore, both data and task parallelism are applied to improve performance, and three encryptions are processed simultaneously.

Fig. 7 shows the vector register configuration of fault attack countermeasures in HIGHT. PT is data in plaintext, and RT is redundant data for original plaintext by applying IIR. KC slices are to detect faults such as instruction skip. Moreover, since the IIR is applied, the round key should be duplicated to match this format. In the proposed round key duplication process, if the lane of the vector is increased to 8h in the TRN1 instruction, the round key is easily duplicated as shown in register configuration of Fig. 7.
First, plaintext and known plaintext are loaded into vector registers. Then, to apply the IIR, the data is duplicated. The data duplication method uses the `MOV` instruction to fill the lower 64-bit with the upper 64-bit, and then the vector order is rearranged as shown in Fig. 7 by using the `TBL` instruction. After that, random shuffling is applied to the message and the round key in each round based on the random bit. When the data and round key are properly shuffled, the round function is performed. HIGHT consists of 32 rounds, so a 32-bit random bit is required. After the 32 rounds are completed, the data encrypted and the redundant data encrypted through last-random-shuffling is rearranged to the correct position. Finally, via fault checking, we verify whether an active attacker injects faults during the encryption process. If no faults are injected, the flag value returns to 0 and the ciphertext is stored in memory. When storing in memory, the vector register in which the `UZP1` instruction was applied during fault checking has original data out of correct order, so the vector order is correctly rearranged for the vector registers of value applied in the `UZP1` instruction, and the rearranged vector registers are stored in memory.

4) IMPLEMENTATION OF FAULT COUNTERMEASURE FOR REVISED CHAM

The revised CHAM family is classified into revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 according to parameters in plaintext and key, and the word unit in which the operation is performed consists of 16-bit, 32-bit, and 32-bit, respectively. We describe based on revised CHAM-64/128. In revised CHAM-64/128, a 128-bit vector register is performed in 16-bit units, and the IIR and KC slices are applied to efficiently detect the computation faults and instruction faults.

Fig. 8 shows the vector register configuration of fault attack countermeasures for revised CHAM-64/128. PT is data for plaintext, RT is redundant data which applied IIR for plaintext. KC slices are data for known ciphertext. Since plaintext data is duplicated, the round key should also be set as shown in Fig. 8. As with HIGHT, it is easy to set by increasing vector lane to 4s in the proposed key duplication process.

First, we load plaintext data and known plaintext into vector registers. To duplicate the data, the lower 64-bit of a vector register is filled with the upper 64-bit of a vector register through the `MOV` instruction. After that, by using the `TBL` instruction, a vector order is set as in Fig. 8. Previously, each round was performed by random-shuffling, but to improve the performance, we apply the task parallelism that performs every four rounds at simultaneously, so random-shuffling is performed at once in every four rounds. In addition, we improve performance through the proposed random shuffling method. When the data and the round key are properly shuffled based on the random bit, proposed random shuffling is applied once each 4 rounds. The inputs in the 2, 4 rounds should match the encrypted 0th word and the 2nd word of the previous block to apply the task parallelism, but random-shuffling is applied to each round input value, so the random-shuffling should also be applied to intermediate value. Revised CHAM-64/128 consists of 88 rounds, and since random shuffling is applied in units of four rounds, a total of 12-bit of random bit is required. When the 88 round is completed, last-random-shuffling applies to return the encrypted data to the correct position. After that, we verify whether there was a fault attack during encryption with the proposed fault check. Revised CHAM-64/128 consists of 16-bit words, when the fault check is performed, so we should change the lane from 16b to 8h when extracting the KC slices in the proposed fault check algorithm. After the fault check, if no fault is injected, the KC slices should be removed from the register which stores encrypted data to return the ciphertext. Only the encrypted data are extracted through the `TRN1` instruction from the register to which the `UZP1` and `UZP2` instructions are applied.
Finally, the vector order of the encrypted data is aligned through the TBL instruction and stored in memory.

V. PERFORMANCE AND SECURITY ANALYSIS

We evaluate the performance of optimized implementation and implementation of the fault attack countermeasure for HIGHT, and revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 on Raspberry Pi 4B board. In particular, our board supports 1.5GHz Quad-Core 64-bit and ARM Cortex-A72(ARMv8) microcontroller. We used all registers provided by NEON to improve performance, and implemented it considering a pipeline stall. The environment was compiled via AArch64-linux-gnu-gcc in VSCodium terminal on Ubuntu 19.10. In this section, we are divided into HIGHT and revised CHAM to compare performance with previous research results and security against fault attacks.

A. PERFORMANCE COMPARISON OF HIGHT IMPLEMENTATIONS

Table 4 shows the comparison results of the performance and security analysis for HIGHT. Seo et al. presented optimized implementation and fault attack countermeasures for HIGHT by using SIMD instruction set on ARM-Cortex-M4. Task and data parallelism was presented using an ARM-based 32-bit general purpose register, and an effective round key duplication method was also presented. In the fault attack implementation, instruction fault was unable to detect due to the lack of space in the register. Reference of KISA is the result of measuring the reference implementations of HIGHT provided by KISA in the ARMv8 environment. Since it is a reference implementations, it consists of c language, and the optimization and a fault attack countermeasure are not applied.

Previous random-shuffling is difficult to compare in paired situations because the random shuffling method proposed in the previous work [13] is adapted to the ARMv7 environment. As the previous random-shuffling method simply consists of AND, ORR, and MOV instructions, we ported previous random-shuffling to the ARMv8 environment for comparison in fair situations. The above method was known as the fastest random-shuffling. Previous work (fault resistance) is a version by applying a fault attack countermeasure using previous random-shuffling ported to the ARMv8 environment in HIGHT. Our proposed data and task parallelism show performance improvement of about 8 times than the KISA reference implementation. In addition, our proposed key scheduling method shows about 10 times improvement than the KISA reference implementation. By performing a lot of operations and data simultaneously, it showed much improved performance more than the KISA reference implementation. However, there is a disadvantage that it is vulnerable to fault attack because it is applied to a fault attack countermeasure.

Finally, our proposed fault attack countermeasure detects the computation faults and instruction faults by applying IIR and adding KC slices in the register. Through the IIR, since we put the plaintext and the redundant plaintext in vector registers, we can effectively counter computation faults such as random-bit, random-byte, random-word, and chosen bit pair by comparing the plaintext and the redundant plaintext of the encrypted result. However, if the same faults are injected into plaintext and redundant plaintext so that it generates the same ciphertext, the faults is not detected. The faults like this is an example of instruction skips. Thus, we added KC slices to efficiently detect to instruction faults by comparing them with known ciphertext. Besides, by applying random-shuffling for each round, an attacker make it difficult to inject a specific word in the program. There is a disadvantage that overheads are caused because random-shuffling is applied for each round, but we tackled this disadvantage using the proposed
random-shuffling method, and implemented efficiently using fewer registers than the previous random-shuffling method. Even though this work (fault resistance) is applied to fault attack countermeasures, it is about 100% faster than the reference implementation without fault attack countermeasures. Furthermore, it is about 50% faster than a countermeasure applied in the previous random shuffling method. This work is much more efficient than the previous works in terms of performance and security in the ARMv8 environment, a popular low-power embedded microcontroller.

**B. PERFORMANCE COMPARISON OF REVISED CHAM IMPLEMENTATION**

Table 5 shows the comparison results of performance and security analysis for revised CHAM. Since there is no existing revised CHAM implementation on ARMv8, the basic comparison is based on the reference implementations [3] and directly revised CHAM with previously proposed random-shuffling method on ARMv8 for fair comparison. Since the reference implementations is in C language, optimization and fault attack countermeasure are not applied.

This work applied revised CHAM-64/128 data and task parallelism improves 15 times for key scheduling and 38 times for encryption than the reference implementations. In addition, revised CHAM-128/128 and CHAM-128/256 are improved about 13 times, respectively. we achieve improved performance by processing the proposed round key duplication and multiple data simultaneously and four rounds at once. However, since fault attack countermeasures are not applied, there is a disadvantage

| Work | Platform | RW | RB | Rb | CbP | Computation Fault | Instruction Fault | KS | ENC |
|------|----------|----|----|----|-----|-------------------|-------------------|----|-----|
| Reference of revised CHAM-64/128 [3] | ARM Cortex-A72 | -  | -  | -  | -   | -                 | -                 | 3  | 230 |
| Reference of revised CHAM-128/128 [3] | ARM Cortex-A72 | -  | -  | -  | -   | -                 | -                 | 5  | 120 |
| Reference of revised CHAM-128/256 [3] | ARM Cortex-A72 | -  | -  | -  | -   | -                 | -                 | 11 | 130 |

**TABLE 5.** Running time and fault attack resistance comparison of revised CHAM implementations (RW: Random Word, RB: Random Byte, Rb: Random bit, CbP: Chosen bit Pair. CPB: Clock per Byte, KS: Key Schedule, ENC: Encryption).
that it is vulnerable to a fault attack. This work (revised CHAM-64/128 fault resistance) shows a improved performance of about 30% compared with a previous work (a revised CHAM-64/128 fault resistance), and also shows a improved performance of about 80% and 70% in revised CHAM-128/128 and CHAM-128/256, respectively.

Revised CHAM had a disadvantage that overheads occur as random-shuffling was performed in each round, but we reduced the number of random-shuffling by processing four rounds simultaneously. Also, by presenting more efficient random-shuffling than the previous random-shuffling, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 all show faster performance than previous work (fault resistance). Revised CHAM-64/128 was added IIR and KC slices into register to effectively detect faults. However, revised CHAM-128/128 and CHAM-128/256 was added only IIR due to the limitation of vector register space. The instruction faults have a high assumption of attackers, so the computation faults are more important than instruction faults in real world, and instruction faults can easily be applied by removing task parallelism in vector registers. In addition, this works with fault attack countermeasure are about are about 130%, 230%, and 190% faster than the reference implementations without any countermeasures for revised CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively.

Our approach is the first work to provide fast and secure implementation for revised CHAM-64/128, CHAM-128/128 and CHAM-128/256 in the ARMv8 platform, a widely used low-power embedded microcontroller. We presented the optimized implementation and efficiently fault attack countermeasure for revised CHAM-64/128, CHAM-128/128, CHAM-128/256. In addition, all this work for data and task parallel showed improved results than the reference implementations, and even thought the fault attack countermeasures were applied, all this work for fault resistance achieved faster performance than the reference implementations.

VI. CONCLUDING REMARKS

With the development of IoT technology, various IoT devices exist, and they communicate with each other. In communication, security is essential to safely provide services to users. However, since IoT devices are a limited environment, it is difficult to apply security, and each environment has a different memory, CPU performance, and instruction set. Recently, ARMv8-based devices are widely used as mobile IoT devices. Thus, we presented efficient and secure implementations of ARX-based block ciphers (HIGHT and revised CHAM) on ARMv8 platforms. For efficiency, we proposed task and data parallel processing mechanisms by taking advantage of NEON architecture embedded in the ARMv8 platforms. Furthermore, we proposed efficient key scheduling and round key duplication for task parallel processing. Through the proposed methods, the proposed HF version with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 achieved about 8 times, 38 times, 13 times, and 13 times, respectively, compared with the reference implementations. To prevent fault attacks on the block cipher implementations, we presented secure version HS equipped with software fault countermeasures. For the resistance against fault attacks, we proposed efficient fault attack countermeasure based on IIR and KC slices. In addition, we proposed fault checking method optimized on target platforms and enhanced random-shuffling method compared with the previously proposed random-shuffling method. Through the proposed methods, the proposed HS version with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 achieved about 50%, 30%, 80%, and 70% of improved performance compared with the previous best results. Furthermore, the proposed HS version with HIGHT, revised CHAM-64/128, CHAM-128/128, and CHAM-128/256 showed about 100%, 130%, 230%, and 190%, respectively, faster than the reference implementations without the fault attack countermeasure. Thus, we achieved the most fast and secure implementation on the ARMv8 platforms. In the future, we plan to apply the proposed method to various lightweight block ciphers, such as SIMON, SPARK. Our work contributes to the security in IoT environments by applying not only cryptography efficiently in embedded devices that are used the ARMv8 processors, but also a countermeasure against fault attacks which are vulnerable to embedded devices.

REFERENCES

[1] KISA: HIGHT Algorithm Specification. Accessed: Aug. 20, 2020. [Online]. Available: https://seed.kisa.or.kr/kisa/algorithm/EgovHightInfo.do

[2] Information Technology—Security Techniques—Encryption Algorithms—Part 3: Block Ciphers. Standard ISO/IEC 18033-3:2010. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:18033-3:ed-2:v1:en

[3] D. Roh, B. Koo, Y. Jung, I. W. Jeong, D.-G. Lee, and D. Kwon, “Revised version of block cipher CHAM,” in Information Security and Cryptology. Seoul, South Korea: Springer, 2019, pp. 1–19.

[4] E. Bihani and A. Shamir, “Differential fault analysis of secret key cryptosystems,” in Advances in Cryptology. Santa Barbara, CA, USA: Springer, 1997, pp. 513–525.

[5] G. Piret and J.-J. Quisquater, “A differential fault attack technique against SPN structures, with application to the AES and KHAZAD,” in Cryptographic Hardware and Embedded Systems. Cologne, Germany: Springer, 2003, pp. 77–88.

[6] M. Tunstall, D. Mukhopadhyay, and S. Ali, “Differential fault analysis of the advanced encryption standard using a single fault,” in Information Security Theory and Practice, Security and Privacy of Mobile Devices in Wireless Communication. Heraklion, Greece: Springer, 2011, pp. 224–233.

[7] W. Li, D. Gu, and J. Li, “Differential fault analysis on the ARIA algorithm,” Inf. Sci., vol. 178, no. 19, pp. 3727–3737, Oct. 2008.

[8] K. Jeong, Y. Lee, J. Sung, and S. Hong, “Differential fault analysis on block cipher SEED,” Math. Comput. Model., vol. 55, nos. 1–2, pp. 26–34, Jan. 2012.

[9] D. Jap and J. Breier, “Differential fault attack on LEA,” in Information Security and Communication Technology. Daejeon, South Korea: Springer, 2015, pp. 265–274.

[10] Y. Lee, J. Kim, J. H. Park, and S. Hong, “Differential fault analysis on the block cipher hight,” in Future Information Technology, Application, and Service, vol. 1. Leuven, Belgium: Springer, 2012, pp. 407–416.

[11] H. Kwon and J. Ha, “Fault injection attack on lightweight block cipher CHAM,” J. Korea Inst. Inf. Secur. Cryptol., vol. 28, no. 5, pp. 1071–1078, 2018.
[12] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, “Lightweight fault attack resistance in software using intra-instruction redundancy,” in Selected Areas in Cryptography. St. John’s, NL, Canada: Springer, 2016, pp. 231–244.

[13] H. Seo, T. Park, J. Ji, and H. Kim, “Lightweight fault attack resistance in software using intra-instruction redundancy, revisited,” in Information Security Applications. Jeju Island, South Korea: Springer, 2017, pp. 3–15.

[14] H. Seo and Z. Liu, “All the high you need on cortex-M4,” in Information Security and Cryptology. Seoul, South Korea: Springer, 2019, pp. 70–83.

[15] B. Koo, D. Roh, H. Kim, Y. Jung, D.-G. Lee, and D. Kwon, “CHAM: A family of lightweight block ciphers for resource-constrained devices,” in Information Security and Cryptology. Seoul, South Korea: Springer, 2017, pp. 3–25.

[16] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, “Simon and speck: Block ciphers for the Internet of Things,” NIST, Gaithersburg, MD, USA, Tech. Rep. 12, 2015.

[17] ARM Developer: A64 SIMD Vector Instructions. Accessed: Aug. 20, 2020. [Online]. Available: https://developer.arm.com/documentation/dui0801/j/a64-simd-vector-instructions/mov–vector–element-?lang=en

[18] ARM Developer: Arm Architecture Reference Manual Armv8, for Armv8-A Architecture Profile. Accessed: Aug. 20, 2020. [Online]. Available: https://developer.arm.com/documentation/ddi0487/fc

[19] Raspberry Pi 4B Specification. Accessed: Aug. 20, 2020. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

[20] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y. Nogami, Y. Zhu, and H. Kim, “Parallel implementations of LEA, revisited,” in Information Security Applications. Jeju Island, South Korea: Springer, 2016, pp. 318–330.

[21] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y. Nogami, Y. Zhu, and H. Kim, “Parallel implementations of CHAM,” in Information Security Applications. Jeju Island, South Korea: Springer, 2018, pp. 93–104.

[22] H. Fuji, F. C. Rodrigues, and J. Lopez, “Fast AES implementation using ARMv8 ASIMD without cryptography extension,” in Information Security and Cryptology. Seoul, South Korea: Springer, 2019, pp. 84–101.

[23] A. Barenghi, L. Breviglieri, I. Koren, G. Pelosi, and F. Regazzoni, “Countermeasures against fault attacks on software implemented AES: Effectiveness and cost,” in Information Security and Cryptology. Scottsdale, AZ, USA: Association for Computing Machinery, 2010, pp. 1–10.

[24] KISA: HEIGHT Algorithm Reference Source Code. Accessed: Aug. 20, 2020. [Online]. Available: https://seed.kisa.or.kr/kisa/Board/18/detailView.do

JINGYO SONG received the bachelor’s degree from the Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea. He is currently pursuing the master’s degree with the Department of Financial Information Security, Kookmin University. His research interests include optimization of cryptographic algorithms and designing efficient side-channel attacks countermeasures in IoT devices.

SEOG CHUNG SEO (Member, IEEE) received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2005, the M.S. degree in information and communications from the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2007, and the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. He worked as a Research Staff Member with the Samsung Advanced Institute of Technology (SAIT) and Samsung DMC Research and Development Center, from September 2011 to April 2014. From 2014 to 2018, he was a Senior Research Member of the Affiliated Institute of ETRI, South Korea. He currently works as an Assistant Professor with Kookmin University, South Korea. His research interests include public-key cryptography, its efficient implementations on various IT devices, cryptographic module validation program, network security, and data authentication algorithms.