All the HIGHT You Need on Cortex–M4

No Author Given

No Institute Given

Abstract. In this paper, we present high-speed and secure implementations of HIGHT block cipher on 32-bit ARM Cortex-M4 microcontrollers. We utilized both data parallelism and task parallelism to reduce the execution timing. In particular, we used the 32-bit wise ARM–SIMD instruction sets to perform the parallel computations in efficient way. Since the HIGHT block cipher is constructed upon 8-bit word, four 8-bit operations are performed in the 32-bit wise ARM–SIMD instruction of ARM Cortex-M4 microcontrollers. We also presented a novel countermeasure against fault attack on target microcontrollers. The method achieved the fault attack resistance with intra-instruction redundancy feature with reasonable performance. Finally, the proposed HIGHT implementation achieved much better performance and security level than previous works.

Keywords: HIGHT Block Cipher, ARM Cortex-M4, Parallel Implementation, Software Implementation, Fault Attack Resistance

1 Introduction

Internet of Things (IoT) applications become feasible services as the technology of embedded processors are developed. In order to provide fully customized services, the IoT applications need to analyze and process big data and the data should be securely encrypted before packet transmission. However, the data encryption is high computation overheads for the low-end IoT devices with limited computation frequency, energy, and storage. For this reason, lightweight block cipher algorithms should be implemented in efficient manner to fit into the certain requirements of applications. In order to evaluate the efficiency of block cipher algorithms in objective manner, Fair Evaluation of Lightweight Cryptographic Systems (FELICS) evaluated the implementations of block ciphers on low-end IoT devices [3]. FELICS framework fairly evaluated the all block ciphers, including Addition, Rotation, and bitwise eXclusive-or (ARX) and Substitution-Permutation Network (SPN) based block ciphers, on low-end IoT devices, including 8-bit AVR, 16-bit MSP, and 32-bit ARM Cortex-M3 microcontrollers. The evaluation metric is execution time, code size, and RAM. However, they didn't consider the recent ARM microcontroller, namely Cortex-M4.

In this paper, we introduce the optimization techniques for HIGHT block cipher on 32-bit ARM Cortex-M4 microcontrollers. We utilized both data parallelism and task parallelism to reduce the execution timing. In particular, we

used the ARM–SIMD instruction sets to perform the parallel computations in efficient way. Since the HIGHT block cipher has 8-bit word, four 8-bit operations can be efficiently performed in a ARM–SIMD instruction of ARM Cortex-M4 microcontrollers. To get compact results, we used platform-specific assembly-level optimizations for HIGHT block ciphers since the features of IoT platforms vary (e.g. word size, number of registers, and instruction set). We also presented a novel countermeasure against fault attack on target microcontrollers. The method achieved the fault attack resistance with intra-instruction redundancy feature. Finally, high-speed and secure HIGHT implementation achieved much better performance and security than previous works. The proposed implementation methods for HIGHT block cipher can be used for other ARX based block ciphers, such as SPECK and SIMON, straightforwardly.

Summary of Research Contributions

The contributions of our work are summarized as follows.

1. *Optimized task and data parallel implementation of HIGHT* A modern 32-bit ARM processor provides a byte-wise SIMD feature, which performs four bytes addition or subtraction operation without overflow or underflow problem. This SIMD instruction is used to perform the parallel computations for HIGHT block cipher. The specialized rotation routines are also used to optimize the `F0` and `F1` functions of HIGHT block cipher.
2. *Fault attack resistance techniques for HIGHT* We introduced the new approach to resist the fault attack for HIGHT block cipher on the 32-bit ARM Cortex-M4 microcontrollers. We used intra-instruction redundant implementation and randomly shuffle the data to increase the randomness of data location. All routines are finely optimized on the target microcontrollers.
3. *HIGHT implementations on 32-bit ARM Cortex-M4 in open source* We share all HIGHT implementations for reproduction of results. The following link provides the source codes: `https://bit.ly/2ZogBRI`.

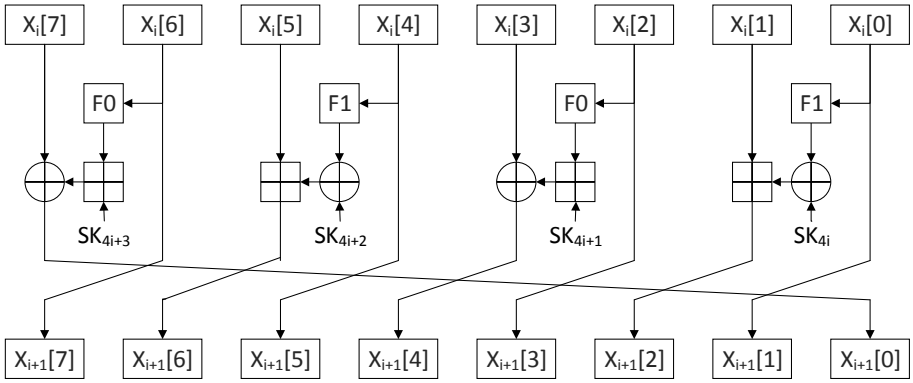
The remainder of this paper is organized as follows. In Section 2, we overview HIGHT block cipher, FELICS framework and previous block cipher implementations on the 32-bit ARM Cortex-M microcontrollers. In Section 3, we introduce compact and secure implementations of HIGHT block cipher for the 32-bit ARM Cortex-M4 microcontrollers. In Section 4, we summarize our experimental results and compare the results with the state-of-the-art works. In Section 5, we conclude the paper.

2 Related Works

2.1 HIGHT Block Cipher

In CHES06, lightweight block cipher HIGHT, was introduced by Korea, and it was enacted as ISO/IEC 18033-3 international block cryptographic algorithm

standard [9]. Since the HIGHT has lightweight features, this is suitable for the low-end IoT applications . The block size is 64-bit and key size is 128-bit. HIGHT block cipher performs 8-bit wise ARX operations. The encryption or decryption operation requires 32 round functions. In each round function, a 64-bit round key is required. In total, 2,048-bit of round keys are needed to process a 64-bit plaintext during the encryption routine.



$$F0(X) = X^{<<<1} \oplus X^{<<<2} \oplus X^{<<<7}$$

$$F1(X) = X^{<<<3} \oplus X^{<<<4} \oplus X^{<<<6}$$

Fig. 1. One round of HIGHT encryption; `X` and `SK` represent input data and session key

2.2 FELICS

In 2015, a benchmarking framework of software based block cipher implementations named Fair Evaluation of Lightweight Cryptographic Systems (FELICS) was held by Luxembourg University. The FELICS benchmarking framework only targets the low-end embedded devices, including 8-bit AVR ATmega128, 16-bit MSP430, and 32-bit ARM Cortex-M3. The FELICS framework provides unified methods for measuring the performance of block ciphers in terms of code size, RAM, and execution timing under the same compiler specifications and target platform. The implementations were evaluated in three scenarios, including cipher operation, communication protocol, and challenge-handshake authentication protocol. By using the framework, the lightweight block cipher competition (i.e. FELICS Triathlon) was held by Luxembourg University. In the competition, more than one hundred block cipher implementations on low-end IoT devices

were submitted by world-wide cryptographic engineers. After competition, they reported the most optimized results and HIGHT implementation won the second round. This result shows that HIGHT block cipher has the one of the most reasonable lightweight block cipher. For this reason, we decide to investigate the optimized HIGHT implementations in modern low-end microcontrollers (i.e. 32-bit ARM Cortex-M4 microcontrollers) in this paper.

2.3 Previous Block Cipher Implementations on IoT Devices

Several works have investigated the implementation of block ciphers on IoT embedded processors. In the past, 8-bit AVR and 16-bit MSP microcontrollers were representative low-end target processors. Many works reported the optimized block cipher implementations on 8-bit AVR and 16-bit MSP microcontrollers [6,7,12,2,4] In recent works, they consider the advanced 32-bit ARM microcontrollers as the reasonable low-end microcontrollers in terms of computation power and energy consumption. In order to fully utilize the 32-bit word size of ARM processors, LEA block cipher selected the 32-bit wise Addition, Rotation, and eXclusive-or (ARX) operations. For this reason, LEA implementation on ARM microcontroller (i.e. ARM926EJ-S) shows higher performance than previous AES implementations [8]. In [15], the LEA implementations through on-the-fly method over ARM Cortex-M3 processors were proposed. They utilized the available registers to retain many parameters as possible and optimized the rotation operation with the barrel-shifter techniques. In [10], the lightweight block cipher CHAM was implemented on ARM Cortex-M3 microcontrollers. Compared with the SPECK block cipher, the CHAM block cipher shows better performance. In [14], highly-optimized AES-CTR assembly implementations for the ARM Cortex-M3 and M4 microcontrollers were introduced. The implementations are about twice as fast as existing implementations. The implementations include an architecture-specific instruction scheduler and register allocator. In [16], LEA and HIGHT block ciphers are evaluated on ARM Cortex-M3 microcontrollers. In particular, pseudo-SIMD technique is used for HIGHT implementation on ARM Cortex-M3. This technique can perform two encryption operations at once on a 32-bit ARM processor. In this paper, we further improve the performance of HIGHT block cipher on ARM Cortex-M4 microcontrollers by using SIMD instruction and parallelism. Furthermore, the secure design against fault attack is also introduced.

2.4 ARM Cortex-M Microcontrollers

The ARM Cortex-M is a family of 32-bit processors for use in embedded microcontrollers. The microcontrollers are designed to be energy efficient, while being fast enough to provide high performance in applications.

ARM Cortex-M4 ARM Cortex-M4 is a 32-bit microcontroller based on ARMv7-M architecture developed by ARM Holdings. ARM Cortex-M4 was announced

in 2010. Cortex-M4 has 32-bit registers and a Thumb/Thumb-2 instruction set that supports both 16-bit and 32-bit operations. Arithmetic instructions take one cycle but memory access instructions take more cycles. The microcontrollers supports barrel-shifter, which performs rotated or shifted registers without additional costs. In particular, the Cortex-M4 supports additional instructions for digital signal processing than Cortex-M3. In this paper, We used the `MK20DX256VLH7` development board. This equips 256 KB of flash memory, 64 KB of RAM, and 2 KB of EEPROM. In can run at up-to 72 MHz. The detailed instructions for ARM Cortex-M4 are given in Table 1. In particular, `UADD8` and `USUB8` instructions perform four byte-wise SIMD operations. This is very efficient to handle carry-less addition and borrow-less subtraction operations.

Table 1. Instruction set summary for 32-bit ARM Cortex-M4

Mnemonics	Operands	Description	Operation	#Clock
ADD	C, A, B	Add word without Carry	C ← A+B	1
EOR	C, A, B	Exclusive OR	C ← A⊕B	1
AND	C, A, B	Bitwise-AND	C ← A&B	1
ORR	C, A, B	Bitwise-OR	C ← A|B	1
LSL	C, A, B	Shift Left	C ← A<<B	1
ROR	C, A, B	Rotate Right	C ← A>>>B	1
MOV	C, A	Move	C ← A	1
PUSH	A	Push word	STACK ← A	2
POP	A	Pop word	STACK → A	2
UADD8	C, A, B	Add bytes without Carry	C ← A+B	1
USUB8	C, A, B	Sub bytes without Carry	C ← A-B	1

3 Proposed Methods

The block cipher encryption usually performed in either sequential or parallel way. The sequential implementation can only compute single block at once, while the parallel implementation utilizes the multiple computing units (e.g. multiple cores) or SIMD instruction sets (e.g. ARM-SIMD, NEON, and AVX2) to perform multiple blocks at once. The parallel computations ideally result in a speed-up of n over sequential execution, where n is the number of parallel computation units. For this reason, the parallel implementation is usually considered to achieve the fast and efficient computations than serial implementations. In this paper, we focused on the parallel computation of lightweight block cipher (i.e. HIGHT) on low-end microcontrollers (i.e. 32-bit ARM Cortex-M4). Furthermore, we improve the security against the fault attack by using parallel implementation.

There are largely two ways of parallelism techniques, including data parallelism and task parallelism. First, the data parallelism is running the same

Table 2. Register utilization for Key Scheduling on ARM Cortex-M4.

Register	Utilization
R0	master key pointer → delta pointer
R1	round key pointer
R2∼R5	delta variables
R6	loop counter
R7∼R8	temporal variables
R9∼R12	round keys

Table 3. Register utilization for task-parallel, data-parallel and fault-resistance Encryption on ARM Cortex-M4.

Register	Task-Parallel	Data-Parallel	Fault Resistance
R0	plaintext pointer	plaintext pointer	plaintext pointer → random number
R1	round key pointer	round key pointer	round key pointer
R2∼R5	plaintext	plaintext	plaintext
R6	mask	mask	mask
R7	loop counter	temporal variable	temporal variable
R8∼R9	round key	round key	round key
R10∼R12	temporal variables	temporal variables	temporal variables
R14	–	loop counter	loop counter

task on different components of data. Second, the task parallelism, also known as function parallelism or control parallelism, runs many different tasks at the same time on the same data. In this paper, we target both task and data parallel HIGHT implementations on 32-bit ARM Cortex-M4 microcontrollers.

3.1 Key Scheduling

Round key generation requires byte-wise addition operations and byte-wise rotation operations. The byte-wise rotation operations are performed with barrel-shifter and masked approach. The four byte-wise addition is easily performed with UADD8 instruction at once. Among 14 general purpose registers (R0∼R12, R14), we utilized 13 registers for key scheduling of HIGHT block cipher as described in Table 2. For the task parallel encryption, only one round key is stored in the word by two bytes. The round key format is {??, SK_{4i+2}, ??, SK_{4i+0}} or {??, SK_{4i+3}, ??, SK_{4i+1}}, where SK and ?? represent the round keys and byte padding, respectively.

For the data parallel encryption, the same round key is duplicated to remaining part (i.e. two bytes out of four bytes) of registers and the word is fully used (i.e. {SK_{4i+2}, SK_{4i+2}, SK_{4i+0}, SK_{4i+0}} or {SK_{4i+3}, SK_{4i+3}, SK_{4i+1}, SK_{4i+1}}).

3.2 Encryption & Decryption

In this section, we introduce two encryption modes, including task parallelism and data parallelism, for HIGHT block cipher. The task parallel implementation

Table 4. Comparison between w/o and w/ SIMD instruction sets for HIGHT computations.

Computations	w/o SIMD	w/ SIMD
XOR after ADD	`ADD`→`EOR`→`AND`	`UADD8` → `EOR`
ADD after XOR	`EOR`→`ADD`→`AND`	`EOR` → `UADD8`

performs one encryption block in parallel way, while the data parallel implementation performs two or more encryption blocks at once. The detailed register utilization is available in Table 3. The decryption can be implemented with same techniques used for encryption.

Task Parallelism In order to perform task parallel HIGHT computation, two bytes are paired (i.e. {??, X_i[4], ??, X_i[0]}, {??, X_i[5], ??, X_i[1]}, {??, X_i[6], ??, X_i[2]}, and {??, X_i[7], ??, X_i[3]}, where X and ?? represent the plaintext and byte padding, respectively.). The `F0` and `F1` functions are performed with masked rotation and exclusive-or, which is proposed in [16]. For the two computations (i.e. XOR after ADD and ADD after XOR), the computations are getting much simpler with `UADD8` (SIMD) instruction than previous masked approach. The comparison between with and without SIMD instruction sets is given in Table 4.

Data Parallelism The data parallel HIGHT implementation performs two encryption blocks at once. Each encryption is performed in task parallel and two task parallel blocks are combined and performed in data parallel approach. For the data parallel implementation, four bytes are paired and two bytes are duplicated (i.e. {X_i[4], X_i[4], X_i[0], X_i[0]}, {X_i[5], X_i[5], X_i[1], X_i[1]}, {X_i[6], X_i[6], X_i[2], X_i[2]}, and {X_i[7], X_i[7], X_i[3], X_i[3]}, where X represents the plaintext.). The XOR after ADD and ADD after XOR operations are performed with the technique described in Table 4.

The both `F0` and `F1` functions require rotation operation. In the task parallel implementation, the data is stored with padding blocks. This prevents the data overflow between bytes. However, the data parallel implementation fully utilizes the block, which means no padding or margin. In order to prevent the data overflow, the format is converted to the padded version, whenever the rotation operation is performed. The detailed descriptions are given in Algorithm 1. In Step 1 $\sim$ 8, the half word is extracted from the word (`R2`) and `F0` function is performed with half word. From Step 9 to 16, the remaining half word of `R2` is performed. Finally, the Step 17 merges the two results.

Fault Attack Resistance The high-speed implementation is not enough for real world cryptography implementations. For this reason, we need to provide the sufficient security against physical attack (e.g. fault attack). In this section, we cover the new HIGHT design to prevent the fault attack.

Algorithm 1 `F0` function for data parallel implementation.

Input: `R2`, temporal variables (`R10` and `R11`)
Output: `R6`

```
 1: AND R11,R2, #0x00FF00FF
 2: LSL R10, R11,#1
 3: EOR R10, R10, R11, LSR #7
 4: EOR R10, R10, R11, LSL #2
 5: EOR R10, R10, R11, LSR #6
 6: EOR R10, R10, R11, LSL #7
 7: EOR R10, R10, R11, LSR #1
 8: AND R6, R10, #0x00FF00FF
 9: AND R11,R2, #0xFF00FF00
10: LSL R10, R11, #1
11: EOR R10, R10, R11, LSR #7
12: EOR R10, R10, R11, LSL #2
13: EOR R10, R10, R11, LSR #6
14: EOR R10, R10, R11, LSL #7
15: EOR R10, R10, R11, LSR #1
16: AND R10, R10, #0xFF00FF00
17: ORR R6, R6, R10
```

The fault model performs the fault injection on a cryptography implementation and manipulates the instruction opcodes (i.e. instruction faults) or data (i.e. computation faults). The fault attack is under very strong assumption but it is possible by a certain attacker with sophisticated equipment and sufficient funding (e.g. government agency). In this paper, we used the identical fault models used in previous works to evaluate our secure implementations [13,17]. The detailed descriptions of four computational fault models are as follows.

- **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:** 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 can change the program flow (the opcode of an instruction) by fault injection. The well-known approach is replace the operation into no-operation (`nop`) instruction.

Fault attack detection in software is proposed in [1]. They perform the duplicate encryption, which is based on the time redundancy of encryption. When the instruction duplication and triplication are performed, the performance is degraded by a factor of 3.4 and 10.6, respectively. Furthermore, the sophisticated fault injection may break the duplicate encryption. Second approach is an information redundancy based encryption. This approach evaluates the additional check variables or parity bits for fault detection. However, this approach also cannot figure out the instruction set level fault attack.

In SAC'16, the intra-instruction redundancy based fault attack countermeasure is suggested [13]. The method implemented the redundant bit-slicing and provides the ability to detect both instruction faults and computation faults.

However, the bit-slicing implementation is only efficient over certain computers with a number of general purpose registers for block ciphers without linear operations. In [11], they evaluated the block cipher PRIDE and TRIVIUM on the Cortex-M3/M4 microcontrollers. They utilized the intra-instruction redundancy. Based on previous intra-instruction redundancy technique, in FDSC'17, they introduce the automatic vectorization compiler to mitigate the fault attack [5]. In WISA'17, the intra-instruction redundancy based fault attack countermeasure on NEON instruction (i.e. SIMD instruction set of 32-bit ARM Cortex-A) is introduced [17]. The implementation shuffles the variables each time to make attack difficult. The shuffling is based on random numbers and the random numbers are also generated in each encryption, simultaneously. Finally, they applied to the LEA encryption to achieve high security against fault attacks. However, there is no practical fault detection on low-end devices with random shuffling method.

In this paper, we utilized the intra-instruction redundancy based fault attack countermeasure for HIGHT block cipher on the low-end Cortex-M4 microcontrollers. In order to mitigate the fault attack, the data is formatted in intra-instruction redundancy and shuffled in each round. In order to efficiently handle the random shuffling, we suggested the novel shuffling technique. Proposed model is based on the combination of data and task parallelism. The overall procedures of secure HIGHT implementation are as follows:

```
Message Loading →                    Message Duplication →
Message / Round Key Shuffling #1 →   Round Function #1 →
...
Message / Round Key Shuffling #32 →  Round Function #32 →
Last Message Shuffling →             Last Round Function →
Fault Attack Check →                 Message Storing
```

The message duplication is easily implemented with barrel-shifter and bit-wise or operations. When the registers (`R2, R3, R4, R5`) are formatted in paired two bytes (i.e. {??, X_i[4], ??, X_i[0]}, {??, X_i[5], ??, X_i[1]}, {??, X_i[6], ??, X_i[2]}, and {??, X_i[7], ??, X_i[3]}), the duplication is performed as follows:

```
ORR R2, R2, R2, LSL#8 → ORR R3, R3, R3, LSL#8 →
ORR R4, R4, R4, LSL#8 → ORR R5, R5, R5, LSL#8
```

Afterward, in each round, we perform 16-bit wise swap shuffling, which exchanges lower 16-bit and higher 16-bit, when the random bit is set to 1. The HIGHT block cipher consists of 32 rounds. For the full round shuffling, we need 32-bit random numbers. The Cortex-M4 microcontroller has 32-bit wise word and one random word can retain 32-bit random numbers. Another consideration is shuffling condition. The shuffling on the message is accumulated but the new round key is shuffled in each time per round. For this reason, we maintain the accumulated shuffling conditions. Due to lack of register, the accumulated shuffling condition is stored in the `STACK`. The detailed descriptions are given in

Algorithm 2 Message and Round Key Shuffling.

Input: message variables (R2∼R5), round key variables (R8 and R9), temporal variables (R10 and R11) random number (R0)
Output: shuffled message variables (R2∼R5), shuffled round key variables (R8 and R9)

```
 1: AND R10, R0, #1
 2: LSL R10, R10, #4
 3: ROR R0, R0, #1
 4: ROR R2, R2, R10
 5: ROR R3, R3, R10
 6: ROR R4, R4, R10
 7: ROR R5, R5, R10
 8: POP {R11}
 9: EOR R10, R10, R11
10: PUSH {R10}
11: ROR R8, R8, R10
12: ROR R9, R9, R10
```

Algorithm 3 Fault Attack Check.

Input: message variables (R2∼R5), temporal variables (R6∼R12 and R14),
Output: check word (R0)

```
 1: AND R6, R2, #0xFF00FF00
 2: AND R7, R3, #0xFF00FF00
 3: AND R8, R4, #0xFF00FF00
 4: AND R9, R5, #0xFF00FF00
 5: AND R2, R2, #0x00FF00FF
 6: AND R3, R3, #0x00FF00FF
 7: AND R4, R4, #0x00FF00FF
 8: AND R5, R5, #0x00FF00FF
 9: EOR R10, R6, R2, LSL #8
10: EOR R11, R7, R3, LSL #8
11: EOR R12, R8, R4, LSL #8
12: EOR R14, R9, R5, LSL #8
13: ORR R10, R10, R11
14: ORR R12, R12, R14
15: ORR R0, R10, R12
```

Algorithm 2. In Step 1 $\sim$ 3, 1-bit random is extracted from R0. When the random bit is set, the offset register (R10) is set to 16. Otherwise, the offset register is set to 0. From Step 4 $\sim$ 7, the message variables are shuffled depending on the offset register. From Step 8 $\sim$ 10, the accumulated offset is loaded from STACK and the current offset is accumulated and stored again into the STACK. Finally, the accumulated shuffling offset is used for round key shuffling in Step 11 $\sim$ 12.

After full round functions, we need to check the fault attack by comparing the duplicated data. The detailed descriptions are given in Algorithm 3. In Step 1 $\sim$ 8, the four bytes pair is divided into two groups. In Step 9 $\sim$ 12, we check whether both results output identical or not. In Step 13 $\sim$ 15, we accumulated all different bits. Finally, we return the check word (R0).

4 Evaluation

In this section, we evaluate the proposed implementation on 32-bit ARM Cortex-M4 microcontrollers. The detailed comparison is available in Table 5. Since this is the first HIGHT implementation on 32-bit ARM Cortex-M4, we only report

Table 5. Comparison of HIGHT block cipher results on 32-bit ARM in terms of code size (byte), RAM (byte), and execution time (clock cycle)

Impl.	Code size (bytes)				RAM (bytes)			Execution time (cycles per byte)		
	EKS	ENC	DEC	SUM	EKS	ENC	DEC	EKS	ENC	DEC
32-bit ARM Cortex-M3										
w/ LUT [16]	316	860	896	1,560	324	704	704	34	269	298
w/o LUT [16]	316	344	384	1,044	324	180	180	37	258	287
32-bit ARM Cortex-M4										
Task parallel	116	348	332	796	316	180	180	18	76	71
Task/Data parallel	160	592	544	1,296	316	188	188	49	56	55
Fault resistance	160	536	520	1,216	316	188	188	49	143	143

the previous works on 32-bit ARM Cortex-M3 as a reference. The comparison between task parallel and data parallel implementations shows that the data parallel is faster than task parallel in key scheduling by 62 %, because data parallel needs to perform key duplication for two block encryption. The encryption and decryption operations of data parallel shows better performance than task parallel by 25 % and 22 %, respectively. The data parallel performs two plaintext at once while the task parallel only performs a plaintext. The code size of data parallel is almost twice larger than task since data parallel needs to perform two plaintext, which requires additional routines. The fault resistance version is similar size of data parallel implementation. The execution time is slower than others since the fault resistance version only perform one encryption and the shuffling operation consumes additional clock cycles. One nice property is that the proposed fault approach is only 2x slower than task parallel implementation.

In terms of security model, we tested several different fault attack scenario studied in previous works as follows [13].

- **Random Word:** The adversary has no control on the number of faulty bits. The adversary can only create random faults in the target word (32-bit).
- **Random Byte:** The adversary can tune the fault injection to randomly affect a single byte of the 32-bit data.
- **Random Bit:** The fault injection can be tuned to affect single bit of the target word.
- **Chosen Bit Pair:** The adversary can inject faults into two chosen, adjacent bits of the target word.

The security comparison of proposed method is given in 6. In the unprotected HIGHT implementation, any computation or instruction fault injection attacks are easily exploited by the adversaries because the unprotected implementation

Table 6. Security comparison of proposed method for HIGHT block cipher on 32-bit ARM Cortex-M4, where Rand: Random shuffling, RW: random word, RB: random byte, Rb: random bit, CbP: chosen bit pair, IS: instruction skip.

Method	Instruction	Rand	RW	RB	Rb	CbP	IS
Seo et al. [16]	ARM	–	–	–	–	–	–
Proposed Method	SIMD	√	√	√	√	√	–

doesn't include fault detection mechanism. The bitslicing approach by [13] is not working for HIGHT since HIGHT consists of some non-linear operations such as addition and subtraction. The previous SIMD implementation by [17] is efficient and secure but it is only working on the high-end processors. Unlike previous work, we targeted low-end processors. We used the data parallel implementation and random shuffling feature. This design efficient prevents several fault attack models based on random byte, random bit, and chosen bit pair. However, we can partly prevent the random word attack. Since the ARM word contains original data and duplicated data, the random word can influence both original data and duplicated data. When the same bit position is selected, the attack succeeds but it is very low possibility. The instruction skip attack is trade-off with other security. If we use the known answer data for fault attack detection, we can check the skip attack but we cannot figure out other attacks. This is limitation of low-end microcontroller. We will extend this method to high-end processors to cover all attack surfaces.

5 Conclusion

In this paper, we presented new compact and secure fault attack countermeasures for HIGHT block cipher algorithm on representative low-end microcontrollers, namely 32-bit ARM Cortex-M4. We firstly optimize the HIGHT block cipher, in terms of task parallelism and data parallelism. For the secure implementation, we proposed the intra-instruction redundancy by using optimized data parallel implementation. This new technique successfully prevent several fault attack models that is infeasible in previous HIGHT implementations on low-end devices.

The proposed methods improved the performance and security of HIGHT implementations. For this reason, there are many future works remained. First, we can directly apply the fault attack countermeasures to the other ARX block ciphers, such as SPECK and SIMON. Recent works on 32-bit ARM Cortex-M do not consider the any secure measures proposed in this paper. We can enhance the security by applying the proposed method, straightforwardly. Second, we only explore the 32-bit ARM Cortex-M4 platform in this paper. However, there are many low-end microcontrollers, such as 8-bit AVR and 16-bit MSP microcontrollers. We will explore the new block cipher implementation techniques for these low-end devices.

6 Acknowledgement

This work was supported as part of Military Crypto Research Center(UD170109ED) funded by Defense Acquisition Program Administration(DAPA) and Agency for Defense Development(ADD).

References

1. A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010.
2. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers. In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 3–20. Springer, 2014.
3. A. Biryukov, J. Großschädl, Y. L. Corre, A. Stemper, V. Velichkov, D. Khovratovich, L. Perrin, D. Dinu, and A. Udovenko. Felics - fair evalutation of lightweight cryptographic systems. NIST Workshop on Lightweight Cryptography, 2015. Avialable for download at `https://www.cryptolux.org/index.php/FELICS`.
4. B. Buhrow, P. Riemer, M. Shea, B. Gilbert, and E. Daniel. Block cipher speed and energy efficiency records on the MSP430: System design trade-offs for 16-bit embedded applications. In *International Conference on Cryptology and Information Security in Latin America*, pages 104–123. Springer, 2014.
5. Z. Chen, J. Shen, A. Nicolau, A. Veidenbaum, N. F. Ghalaty, and R. Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced SIMDization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 57–64. IEEE, 2017.
6. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In *International Conference on Cryptology in Africa*, pages 172–187. Springer, 2012.
7. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.
8. D. Hong, J. Lee, D. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee. LEA: A 128-bit block cipher for fast encryption on common processors. In *International Workshop on Information Security Applications*, pages 3–27. Springer, 2013.
9. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, et al. HIGHT: A new block cipher suitable for low-resource device. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 46–59. Springer, 2006.
10. 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 *International Conference on Information Security and Cryptology*, pages 3–25. Springer, 2017.
11. B. Lac, A. Canteaut, J. J. Fournier, and R. Sirdey. Thwarting fault attacks using the internal redundancy countermeasure (IRC). *IACR Cryptology ePrint Archive*, 2017:910, 2017.

12. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *International Workshop on Fast Software Encryption*, pages 75–93. Springer, 2010.
13. C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy, 2016.
14. P. Schwabe and K. Stoffelen. All the AES you need on Cortex-M3 and M4. In *International Conference on Selected Areas in Cryptography*, pages 180–194. Springer, 2016.
15. H. Seo. High speed implementation of LEA on ARM Cortex-M3 processor. *Journal of the Korea Institute of Information and Communication Engineering*, 22(8):1133–1138, 2018.
16. H. Seo, I. Jeong, J. Lee, and W. Kim. Compact implementations of ARX-based block ciphers on IoT processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):60, 2018.
17. H. Seo, T. Park, J. Ji, and H. Kim. Lightweight fault attack resistance in software using intra-instruction redundancy, revisited. In *International Workshop on Information Security Applications*, pages 3–15. Springer, 2017.