Hindawi
Security and Communication Networks
Volume 2019, Article ID 5323578, 10 pages
https://doi.org/10.1155/2019/5323578

Research Article

Compact Implementations of HIGHT Block Cipher on IoT Platforms

Bohun Kim,[1] **Junghoon Cho**,[1] **Byungjun Choi**,[1] **Jongsun Park**,[1] **and Hwajeong Seo**[2]

[1]*Korea University, Electrical Engineering, Seoul 136-701, Republic of Korea*
[2]*Hansung University, IT Engineering, 116 Samseong-Yoro-16-Gil Seongbuk-gu, Seoul 136-792, Republic of Korea*

Correspondence should be addressed to Hwajeong Seo; hwajeong84@gmail.com

Received 12 July 2019; Revised 22 November 2019; Accepted 7 December 2019; Published 31 December 2019

Academic Editor: Petros Nicopolitidis

Recent lightweight block cipher competition (FELICS Triathlon) evaluates efficient implementations of block ciphers for Internet of things (IoT) environment. In the competition, the implementation of HIGHT block cipher achieved the most efficient lightweight block cipher, in terms of code size (ROM), memory (RAM), and execution time. In this paper, we further investigate lightweight features of HIGHT block cipher and present the optimized implementations of both software and hardware for low-end IoT platforms, including resource-constrained devices (8-bit AVR and 32-bit ARM Cortex-M3) and application-specific integrated circuit (ASIC). By using proposed optimization methods, the implemented HIGHT block cipher shows better performance compared to previous state-of-the-art implementations.

1. Introduction

Internet of things (IoT) infrastructure consists of heterogeneous devices, ranging from low-end resource-constrained devices to high-end server computers. The devices are usually located in remote area and transmit the collected sensor data to server platforms. However, low-end platforms only support very limited computation power and storage capacity (i.e., RAM and ROM), due to production costs and battery life. Under this condition, block cipher implementations should perform fast encryption operations for target applications. For some lightweight IoT hardware architectures, such as passive RFID tags, a key aspect of implementation is minimization of the chip area. According to previous works, the maximum layout area for security part of the RFID tags was limited to 2,000 gate equivalents (GEs) [1]. Although the precise constraint figure might be mitigated a little, the chip area is still an essential point of lightweight implementation and smaller area is beneficial in terms of production cost. In particular, these IoT applications may use user's private and sensitive information. This information should be handled securely. In order to keep the information secret, every network packet should be encrypted before packet transmissions. To meet lightweight requirements of block cipher, the block cipher implementations for low-end IoT devices are actively studied. In 2015, the University of Luxembourg held lightweight block cipher competition for IoT devices. The purpose of competition was to find the best block cipher implementation for IoT environment, in terms of code size, memory, and execution time. In the competition, HIGHT block cipher was selected as the efficient block cipher for low-end IoT devices [2]. However, very few software and hardware implementations have previously evaluated the HIGHT block cipher. There is a room to improve the performance of HIGHT implementations. In this paper, we present efficient implementations of HIGHT block cipher, in terms of software and hardware aspects. The implementation results show that HIGHT block cipher is very lightweight. This feature is suitable for applications on the low-end devices. The main contributions of this paper are summarized as follows.

1.1. Contribution

1.1.1. Size-Optimized Bit/Digit-Serial HIGHT Implementation on ASIC. We implemented the first serial HIGHT implementation on ASIC. Text and key registers are replaced

by shift register to eliminate multiplexers (MUXs) for data control. Exploiting only one 1-byte shift register, auxiliary functions F0 and F1 in units of 1 byte are implemented in units of 1 bit. In order to support bit/digit-serial modes, the dataflow of text register and key register is efficiently scheduled. As a result, the size of bit-serial implementation is reduced to half of the round-based implementation.

1.1.2. Speed- and Size-Optimized HIGHT Implementation on 8-Bit AVR Devices. For the speed optimization, bitwise operations in delta update, F0 function, and F1 function are replaced into the Look-Up-Table (LUT). In order to ensure efficient memory access, the LUT is stored in 8-bit alignment. The available registers are efficiently allocated to reduce the number of memory access. For the size optimization, delta update, F0 function, and F1 function are implemented in bitwise operations, rather than LUT. The on-the-fly method reduces the code size. During key scheduling, the master key management scheme is introduced. This scheme rotates the master keys and selects the suitable keys in each round.

1.1.3. Speed-Optimized HIGHT Implementation on 32-Bit ARM Cortex-M3 Devices. The 8-bitwise operations of HIGHT are implemented in instruction-level parallel way on the 32-bit word. This technique performs two or four bytes operations simultaneously. For key scheduling, the novel approach to perform the addition of 7-bit and 8-bit operands is presented. For encryption and decryption functions, all available registers are efficiently allocated to reduce the number of memory access.

The remainder of this paper is organized as follows. In Section 2, FELICS Triathlon competition, HIGHT block cipher, and target platforms are introduced. In Section 3, optimization techniques for HIGHT block cipher in hardware and implementations of HIGHT block cipher on 8-bit AVR devices are covered. In Section 5, we present HIGHT implementations on 32-bit ARM Cortex-M3 devices. Finally, Section 6 concludes the paper.

2. Related Works

2.1. FELICS Triathlon. In 2015, the software-based block cipher implementation benchmarking framework named Fair Evaluation of Lightweight Cryptographic Systems (FELICS) was held by the University of Luxembourg. The system provides similar evaluation features and functions to SUPERCOP framework, but the FELICS framework targets specific devices, which are main low-end IoT platforms. Three platforms, including 8-bit AVR, 16-bit MSP, and 32-bit ARM, were evaluated, in terms of three metrics: execution time, RAM, and code size. The optimized block cipher implementations were tested in three scenarios: cipher operation, communication protocol, and challenge-handshake authentication protocol. In the competition, one hundred implementations were submitted from international researchers. In the competition, LEA won the first round and HIGHT won the second round (see Table 1). The other block ciphers, including SPECK and Chaskey, also showed notable features [3, 4]. The one interesting insight is that the top-ranked block cipher implementations usually follow the Addition, Rotation, and bitwise eXclusive-or (ARX) architecture rather than traditional Substitution-Permutation-Networks (SPN). The ARX operations are relatively fast and cheap in both hardware and software implementations than SPN. Furthermore, ARX operations are performed in constant time, which even makes it immune to timing attacks and simple power analysis. FELICS results also showed that 32-bit word size is optimal for 8-bit, 16-bit, and 32-bit devices since 32-bit word operation can be easily computed on these devices but 8-bit word is inefficient for 16-bit and 32-bit devices.

Table 1: Winners of FELICS Triathlon (block size/key size).

Rank	First Triathlon	Second Triathlon
1	LEA (128/128)	HIGHT (64/128)
2	SPECK (64/96)	Chaskey (128/128)
3	Chaskey (128/128)	SPECK (64/128)

2.2. HIGHT Block Cipher. In CHES'06, lightweight block cipher, HIGHT, was introduced in South Korea and was enacted as ISO/IEC 18033-3 international block cryptographic algorithm standard [2]. HIGHT block cipher targets low-end devices and low-cost hardware implementations. It has the ARX architecture, which supports 64-bit block size and 128-bit key size. HIGHT algorithm consists of Initial Transformation, Round Function, and Final Transformation. The number of round is 32, except for the transformations. The detailed descriptions of the round function and key generation are given in Figure 1. The basic operations of round function are 8-bit wise addition (+in square), exclusive-or (+in circle), and rotation. The notation ($<<<n$) in F0 function and F1 function means n-bit left rotation of 8-bit value. There is no rotation in the last round. The key scheduler produces 32-bit whitening keys for the transformation and 32-bit subkeys for the round functions. The whitening keys are extracted from the master key, and the subkeys are generated by adding the part of the master key and the output of the 7-bit linear feedback shift register (LFSR).

2.3. Previous Works on ASIC. For ASIC implementation of block ciphers, the round-based architectures, performing a round function in a clock, have been widely adopted. The hardware implementation offers a reasonable trade-off between the area and the throughput. Existing implementations of HIGHT block cipher in ASIC also strike the balance between both features [2, 5]. In [2], a round-based hardware implementation for HIGHT encryption mode was implemented. To support both encryption and decryption operations, the unified architecture sharing common datapath for both encryption and decryption operations was proposed in [5]. However, the implementations may not be suitable for low-end devices, such as passive RFID tags, where the area and power consumption are limited [6, 7]. In order to dramatically reduce the area and power consumption of the circuit, serial implementation is considered. Although ARX ciphers other than HIGHT have been implemented in serial

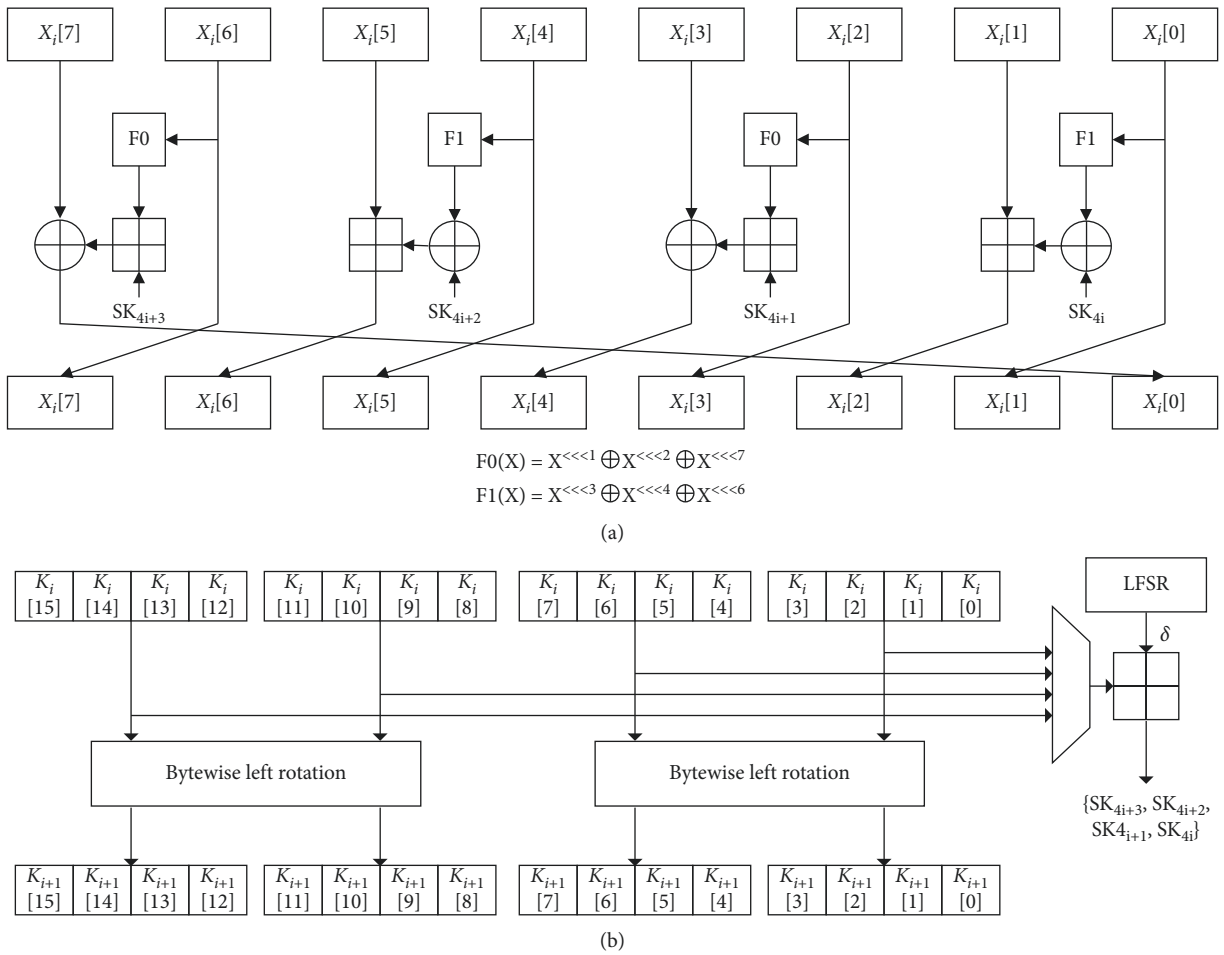

Figure 1: (a) Round function and (b) subkey generation of HIGHT encryption, where X, K, and SK represent input data, master key, and subkey, respectively.

architecture, HIGHT needs more dedicated optimization for the serial implementation.

2.4. Previous Works on 8-Bit AVR Processors. 8-bit AVR embedded processor is clocked at 7.37 MHz and provides 128 KB EEPROM chip and 4 KB RAM [8]. 8-bit AVR processor has the RISC architecture and only 32 registers are available. Among the registers, 6 registers are used to indicate the indirect address. Remaining 26 general purpose registers are utilized to perform the instructions, where one arithmetic instruction requires 1 clock cycle and memory instruction requires 2 clock cycles. On the 8-bit AVR processor, many block cipher algorithms are evaluated. The encryption of AES was performed and ARX-based block ciphers including LEA, SPECK, and SIMON are also efficiently implemented [3, 9, 10, 11]. In WISA'18, the optimized HIGHT implementation on 16-bit MSP processors is suggested [12]. However, the implementation uses the 16-bit MSP instruction set-based optimization which is not available in 8-bit AVR processors. HIGHT implementation on 8-bit AVR processor is also suggested but previous works missed several optimization techniques covered in our paper [13].

2.5. Previous Works on 32-Bit ARM Cortex-M3 Processors. The Cortex-M is a family of 32-bit devices used in embedded environments. The device is designed to be energy efficient, while being fast enough to provide high performance in applications. ARM Cortex-M3 is a 32-bit device based on ARMv7-M architecture developed by ARM Holdings. Cortex-M3 was announced in 2004. Cortex-M3 has 32-bit registers and a Thumb/Thumb-2 instruction set that supports both 16-bit and 32-bit operations. Arithmetic instructions take one clock cycle, but memory access instructions take more. The devices support a barrel-shifter feature, which performs rotated or shifted registers without additional cost. In [14], the LEA implementations through on-the-fly method over ARM Cortex-M3 devices were proposed. They utilized available registers to retain many parameters as possible and optimized the rotation operation with the barrel-shifter feature. In [15], the lightweight block cipher CHAM was implemented on ARM Cortex-M3 devices. Compared with the SPECK block cipher, the CHAM block cipher showed better performance. In [16], highly optimized AES-CTR assembly implementations for the ARM Cortex-M3 and M4 devices were introduced. The implementations were about twice as fast as the existing

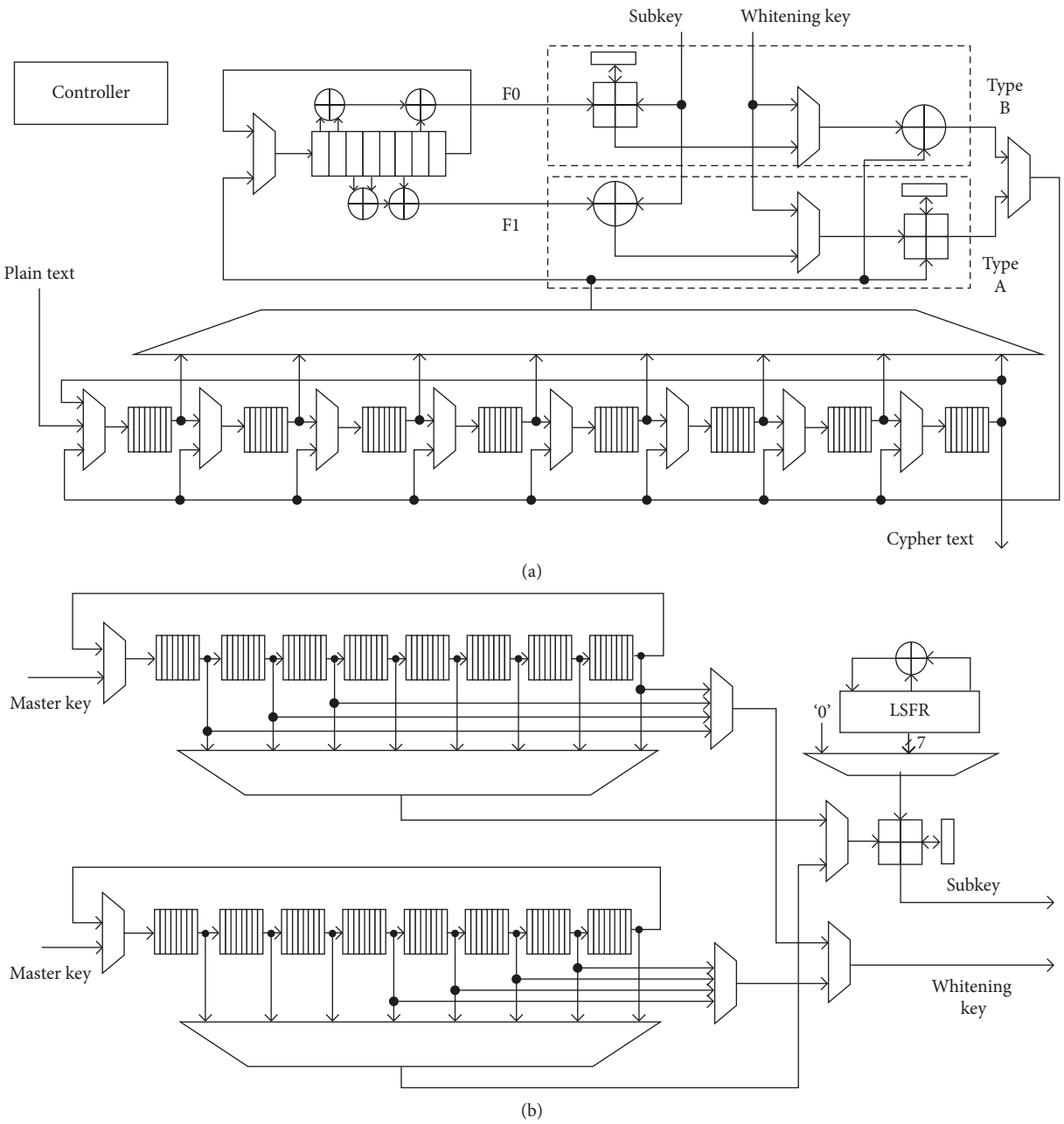

Figure 2: Bit-serial architecture: (a) round function; (b) key scheduler.

implementations. The implementations included an architecture-specific instruction scheduler and register allocator. In [13], LEA and HIGHT block ciphers were evaluated on ARM Cortex-M3 devices. Pseudo-SIMD technique was used for HIGHT implementation on ARM Cortex-M3. This technique could perform two 8-bit words at once on a 32-bit ARM processor.

3. Hardware Implementation

3.1. Serial Implementation. In this section, we present novel bit-serial and digit-serial hardware implementations of HIGHT block cipher. The serial hardware operates in units of one bit or a few bits. This compact hardware is suitable for resource-constrained devices such as passive RFID tags with limited chip area and peak power consumption [7, 15, 17].

3.1.1. Bit-Serial Implementation. The proposed bit-serial architecture is shown in Figure 2. The top architecture consists of the round function module and key scheduler like conventional round-based architecture of [5]. The round function module performs the round function as well as the initial and final transformations. The key scheduler provides subkey and whitening key for the round function module. In the bit-serial architecture, round function module and key

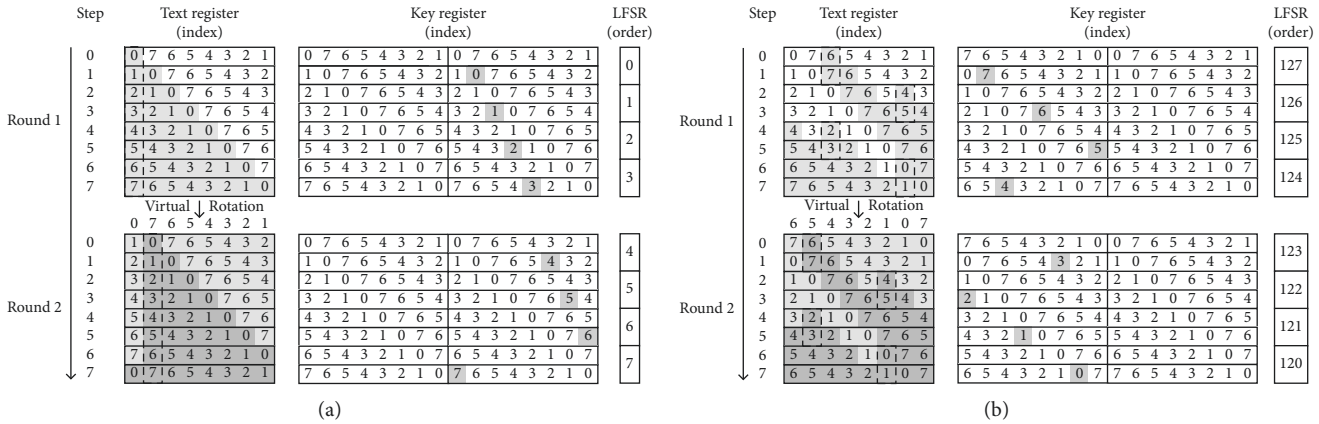

Figure 3: Bit-serial dataflow: (a) encryption; (b) decryption.

scheduler have 1-bit datapath and performs bitwise operations. The text and key registers are shift registers which perform shift operation every clock cycle without exception. Since this type of shift register does not have MUX for input selection of flip-flop, the area of the flip-flop is reduced to approximately 60–70% of the conventional flip-flop [7]. This scheme is effective for overall area reduction because registers occupy a large portion of HIGHT hardware.

Since the 64-bit text shifts 1 bit at every clock cycle, a round function is performed for 64 clock cycles. In this section, we define a step as the process of one byte operation (i.e., one round takes 8 steps and one step takes 8 clock cycles in the bit-serial architecture). The 64-bit text is divided into eight 1-byte words as shown in Figure 1. The even words ($X[0]$, $X[2]$, $X[4]$, and $X[6]$) are processed in even steps and the odd words ($X[1]$, $X[3]$, $X[5]$, and $X[7]$) are processed in odd steps. To perform the round function of bit-serial encryption, one of the eight indices in the text register is selected for each round. The process is performed from the least significant bit (LSB) of the selected index, while the text is shifting. In the even steps, the selected even word is fed to the next index of text register and the additional 1-byte shift register, which exists to support F functions, simultaneously without any operations. In the odd steps, the additional register generates the outputs of F functions by 1 bit while performing cyclic shift. Due to this scheme, the bitwise processing is allowed even though F functions have a dependency on a byte. Consequently, the fundamental operation of the round function is performed with the output of F function, the subkey, and the selected odd word in 1-bit unit. The round operation in odd step is classified into two types. In Figure 2(a), Type A is selected for the operation of $X[1]$ and $X[5]$ while type B is selected for that of $X[3]$ and $X[7]$. To support 1-bit addition, 1-bit flip-flops are used to store carries. One output of two types of operation is selected properly and fed to the next index of text register. Figure 3 shows the dataflow of bit-serial round operation at each step. The index of the register that receives the text after the round operation is indicated by the dotted box in the text register. The dark box in the text register indicates the text where the round operation is done. After 64-bit round operation is completed, another index is selected to perform next round. This change of the processing index acts as a virtual bytewise rotation after a round of HIGHT algorithm is shown in Figure 3. When initial or final transformation is performed instead of round function, the data are processed with the whitening key in the even step. The transform functions share the XOR gates and the adders with the round function.

Table 2: Comparison results of serial architectures @ 80 MHz.

Mode	Bits	Area (GEs)	Power (uW)	Latency	Tech.
Enc	1	1111/1625	5.412/0.303	2176	250/65
	2	1180/1738	6.498/0.337	1088	
	4	1312/1926	8.234/0.392	544	
	8	1604/2282	7.713/0.346	272	
	64	2269/3033	9.394/0.383	34	
	64 [2]	3048	—	34	250
Enc/Dec	1	1172/1711	5.538/0.308	2176	250/65
	2	1237/1809	6.534/0.338	1088	
	4	1385/2004	8.462/0.391	544	
	8	1695/2386	8.663/0.357	272	
	64	2560/3322	12.58/0.454	34	
	64 [5]	2608	10.8*	34	350

*Power consumption measured @ 100 KHz.

In the key register in Figure 2(b), 128-bit key register consists of two 64-bit shift registers. LFSR for subkey generation is also 7-bit shift register. The LFSR is updated to the value of the next order in even step and it provides the output by one bit in odd step. The output of LFSR is added to the selected key to produce a subkey. The index of selected key is marked as dark box in the key register in Figure 3. The whitening keys for transformations are extracted at the fixed positions of the key register. The key register can perform the virtual bytewise rotation similar to the text register.

In case of decryption, since HIGHT is a Feistel structure, round function of decryption is similar to that of encryption while some additions are replaced by subtractions and the direction of the rotation is the opposite of encryption. In the key scheduling, the direction of the rotation and the order of LFSR is also the opposite of encryption. In order to implement LFSR compactly, the order of LFSR has to be sequentially decreased from the last order. Meanwhile, for the

Table 3: Comparison of speed-optimized HIGHT block cipher implementations on 8-bit AVR in terms of code size (bytes), RAM (bytes), and execution time (cycles per byte).

Impl.	Code size (bytes)				RAM (bytes)			Execution (cycles per bytes)		
	EKS	ENC	DEC	SUM	EKS	ENC	DEC	EKS	ENC	DEC
[18]	—	—	—	—	—	—	—	—	371	371
[13]	386	1090	1096	2060	302	682	682	58	160	161
This work	886	1920	1920	4726	294	666	666	47	157	157
AES [19]	808	918	1118	2588	219	226	226	95	177	231

operation of odd words, the output of F function is needed. Therefore, the process of even word which corresponds to the odd word must be done beforehand. Therefore, the processing index is changed every other steps as shown in Figure 3(b).

3.1.2. Digit-Serial Implementation. Digit-serial architecture is a serial architecture that the unit of processing is in a few bits. The size of text and key register is the same as the bit-serial architecture. The difference is that n-bit datapath is used and shift registers perform n-bit shift in n-bit-serial architecture. Processed data are n-bit from LSB of each 8-bit word. The option for the unit bit-width of process provides a trade-off between area, power, and throughput. The larger unit induces larger area, power, and higher throughput.

3.1.3. Evaluation. The proposed architectures are coded in Verilog. The programmed codes are synthesized, and the area is measured using Synopsys Design Compiler. The power consumption is measured using Primetime PX at frequency of 80 MHz. For a variety of comparisons, we used two standard cell libraries, including Samsung 65 nm and TSMC 250 nm CMOS technology. Gate equivalents (GEs) can vary depending on the standard cell libraries, which have different relative sizes of 2-input NAND gate compared to other gates. In Table 2, the measurements of area and power consumption of the proposed bit-serial and digit-serial implementations are given. The round-based implementation has been added to be compared only with serial implementations. In the table, "Enc" mode indicates the architecture supports encryption only while "Enc/Dec" mode indicates that it supports both encryption and decryption. 64-bit unit means round-based architecture. The area of the bit-serial architectures is 45.8%–53.6% of that of the round-based architectures. Most of the area of the serial architectures is occupied by registers that are essential for storing key and text, and a little combinational logic takes up. The latency of the bit-serial architecture is 2176 clocks and those of the digit-serial architectures are in inverse proportion to operation bits. We can choose one of the architectures relying on the implementation condition. Note that 8-bit-serial architectures consume less power than 4-bit-serial architectures in general because bytewise operation is more efficient for HIGHT algorithm. In conclusion, these serial architectures achieve much smaller area and power consumption with the expense of the throughput compared to the conventional round-based architectures.

Table 4: Delta update process in 8-bit AVR instruction sets.

Input: temporal storage (TEMP), input delta (DELTA) Output: output delta (DELTA)
1: CLR TEMP
2: BST DELTA, 0
3: BLD DELTA, 7
4: BST DELTA, 3
5: BLD TEMP, 7
6: EOR DELTA, TEMP
7: LSR DELTA

4. Software Implementation on 8-Bit AVR

The compact HIGHT implementation on 8-bit AVR devices requires efficient 8-bit ARX operations. Particularly, 8-bit AVR devices efficiently perform the 8-bit addition and 8-bit bitwise exclusive-or instructions since the device has 8-bit word size. For the rotation operation, 8-bit AVR devices only provide rotation with 1-bit offset. The efficient offset and direction of rotation is determined by each case. For the 1-bit left rotation, two instructions (LSL X1; ADC X1, ZERO) are performed in 2 clock cycles. For the 1-bit right rotation, three instructions (BST X1, 0; LSR X1; BLD X1, 7) are performed in 3 clock cycles.

4.1. Speed-Optimized Implementation. The speed-optimized implementation emphasized the fast execution timing rather than code size. Part of routines is unrolled to achieve the speed-optimal result.

4.1.1. Key Scheduling. The computation intensive part of key scheduling operation is an update of delta variable. The update of delta variable requires a series of bitwise operations, and this is inefficient for the bytewise machine or software implementation. For the high-speed implementation, the bitwise operations are simply replaced into the Look-Up-Table (LUT). The delta update is efficiently performed in one memory access. In order to ensure the fast memory access, the LUT is stored in 8-bit aligned format where the address pointer is 16-bit wise. When the address is 8-bit aligned, we only need to update 8-bit offset only for memory access. The registers are also efficiently allocated to reduce the number of memory access. Sixteen registers for master keys, four registers for delta and round key pointers, and one register for temporary storage are allocated, respectively.

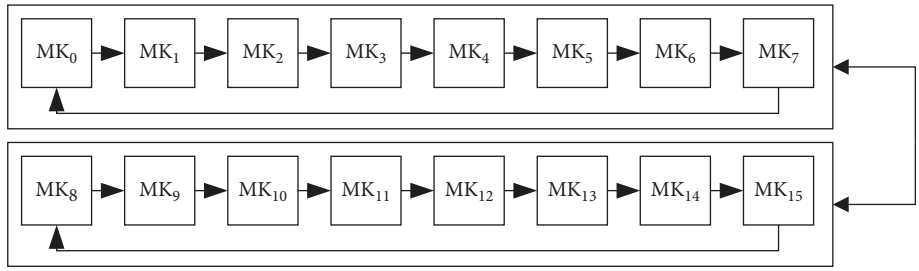

Figure 4: Key management in key scheduling of HIGHT block cipher.

4.1.2. Encryption and Decryption. Each F0 function and F1 function requires 8-bit wise 3 rotations and 2 XOR operations. Similarly, these functions are efficiently computed with 1-time LUT access. Each LUT requires 256 bytes. In total, 512-byte RAM is used to store the 2 LUTs. The registers are also efficiently allocated. Eight registers for the plaintext, four registers for the round keys, one register for the zero value, four registers for the F0 and F1 memory addresses, one register for temporal storage, two registers for round key, and two registers for temporal pointer are allocated, respectively.

4.1.3. Evaluation. In Table 3, comparison results of speed-optimized HIGHT block ciphers on 8-bit AVR microprocessors are given. The previous fastest implementation achieved 58, 160, and 161 clock cycles for key scheduling, encryption, and decryption operations, respectively [13]. The proposed implementation only requires 47, 157, and 157 clock cycles. The performance improvements are 18.9%, 1.8%, and 2.4% for key scheduling, encryption, and decryption, respectively. On the other hand, the proposed implementation consumes double code size compared with [13]. In total, 2.7 KB ROM is more consumed than previous works but this is only 2% of 8-bit AVR ROM. For this reason, this is not critical issue on this target platform. The RAM size is similar to the previous works.

4.2. Size-Optimized Implementation. In order to minimize the code size, fully looped implementation is considered. The size minimization rolls the source codes by the number of iteration (N). If the size of source code is (S), the size of looped version is calculated in ($NS + A$), where (A) represents overheads including counter, offset, and branch operations. Since the execution timing and code size are trade-off relation, the performance of rolled implementation is relatively slower than unrolled implementation.

4.2.1. Key Scheduling. In the key scheduling process, 22 general purpose registers out of 32 general purpose registers are utilized. In particular, 16, 1, 1, 2, and 2 general purpose registers are assigned to master key variables, delta variable (d), temporal storage, counter variables, and memory pointer, respectively. The key scheduling process requires the update of delta variable as follows:

Table 5: F0 operation in 8-bit AVR instruction sets.

Input: input data (X) Output: output data ($F0 = X<<<1$ XOR $X<<<2$ XOR $X<<<7$)
1: MOV TMP, X
2: LSL TMP
3: ADC TMP, ZERO//$X<<<1$
4: MOV F0, TMP
5: LSL TMP
6: ADC TMP, ZERO//$X<<<2$
7: EOR F0, TMP//$X<<<1$ XOR $X<<<2$
8: LSL TMP
9: ADC TMP, ZERO
10: SWAP TMP//$X<<<7$
11: EOR F0, TMP//$X<<<1$ XOR $X<<<2$ XOR $X<<<7$

Table 6: F1 operation in 8-bit AVR instruction sets.

Input: input data (X) Output: output data ($F1 = X<<<3$ XOR $X<<<4$ XOR $X<<<6$)
1: MOV TMP, X
2: LSL TMP
3: ADC TMP, ZERO
4: LSL TMP
5: ADC TMP, ZERO
6: MOV F1, TMP
7: SWAP F1//$X<<<6$
8: LSL TMP
9: ADC TMP, ZERO//$X<<<3$
10: EOR F1, TMP//$X<<<3$ XOR $X<<<6$
11: LSL TMP
12: ADC TMP, ZERO//$X<<<4$
13: EOR F1, TMP//$X<<<3$ XOR $X<<<4$ XOR $X<<<6$

$$d[i+6] = d[i+2] \text{ XOR } d[i-1]. \quad (1)$$

For the update, $(i-1)$ and $(i+2)$-th bits are bitwise exclusive-ored and assigned to the $(i+6)$-th bit. Since the minimum word size of AVR devices is a byte, the optimized bitwise operation is required. The detailed process is given in Table 4. First temporal register is initialized to zero (TEMP). Afterward, 1st and 4th bits are extracted with BST instruction sets and stored to 8th bit with BLD instruction sets. Lastly, both bits are exclusive-ored and shifted to the right by 1 bit.

The key scheduling is performed in two routines. First, lower round keys from 1st to 8th and higher round keys from 9th to 16th are separately performed. In each loop, the

Table 7: Comparison of size-optimized HIGHT block cipher implementations on 8-bit AVR in terms of code size (bytes), RAM (bytes), and execution time (cycles per byte).

Impl.	Code size (bytes)				RAM (bytes)			Execution (cycles per bytes)		
	EKS	ENC	DEC	SUM	EKS	ENC	DEC	EKS	ENC	DEC
[20]	—	—	—	—	—	—	—	—	2,438	2,520
[13]	386	716	722	1,824	302	682	682	58	210	211
This work	188	234	234	656	162	147	147	213	317	317

round keys are rotated by 1 byte to order the round key. For the size optimization, only 1-byte calculation is implemented and the calculation is repeated by the number of computation words. The detailed descriptions are given in Figure 4. First, one round key group between lower and higher parts is selected and rotated by 1 byte in each time. Afterward, the round key group is exchanged. To align the byte order, MOV and MOVW instruction sets are utilized. The MOV is useful for 1-byte rotation and MOVW is useful for 2-byte rotation.

4.2.2. Encryption and Decryption. The encryption of HIGHT block cipher consists of Initial Transformation, Round Function, and Final Transformation. Among them, Round Function consists of 32 rounds. For the size-optimized implementation, Initial Transformation, Round Function, and Final Transformation are only implemented once. In particular, Round Function is iterated by 32 times. In Round Function, the expensive operations are F0 and F1 functions. The detailed descriptions of F0 and F1 functions in 8-bit AVR instruction sets are given in Tables 5 and 6, respectively. The F0 operation requires three rotation operations by different offsets. Since the rotation operations share similar computation routines, the minimum number of rotation routine is generated as follows:

$$\begin{aligned} X[1] &= X \lll 1; \\ X[2] &= X[1] \lll 1; \\ X[3] &= X[2] \lll 1; \\ X[7] &= X[3] \lll 4. \end{aligned} \tag{2}$$

Similarly, the F1 function is implemented as follows:

$$\begin{aligned} X[2] &= X \lll 2; \\ X[6] &= X[2] \lll 4; \\ X[3] &= X[2] \lll 1; \\ X[4] &= X[3] \lll 1. \end{aligned} \tag{3}$$

For high-speed computations, register usages are also optimized. Only total 16 registers are utilized. Among them, 3 registers are only set to callee-saved registers, which avoids the stack push/pop operations before/after function call. Particularly, 8 registers for plaintext, 1 register for zero value, 2 registers for temporal storage, 1 register for counter, and 4 registers for pointers are allocated, respectively. The decryption operation can be implemented in a reversed order of encryption operation. Detailed implementations are similar to the encryption operation.

Table 8: Optimized key scheduling.

Input: four 7-bit delta (A), four 8-bit operand (B) Output: four 8-bit result (A)
1: AND TEMP, B, $0 \times 7F7F7F7F$
2: ADD TEMP, A, TEMP
3: AND A, B, 0×80808080
4: EOR A, A, TEMP
5: Return A

4.2.3. Evaluation. In Table 7, comparison results of size-optimized HIGHT block cipher implementations on 8-bit AVR devices are given. The previous smallest implementation achieved 386, 716, and 722 bytes for key scheduling, encryption, and decryption operations, respectively [13]. The proposed implementation only requires 188, 234, and 234 bytes and optimizes the code size by 51.2%, 67.3%, and 67.5% for key scheduling, encryption, and decryption operations, respectively. On the other hand, the proposed implementation requires more clock cycles for computations than [13]. Particularly, the proposed implementation requires 213, 317, and 317 clock cycles for key scheduling, encryption, and decryption operations, respectively. The execution timing is still competitive enough to support efficient encryption computations.

5. Software Implementation on 32-Bit ARM Cortex-M3

The word size of HIGHT block cipher is 8 bit, while 32-bit ARM Cortex-M3 performs the operation in 32-bit wise. In order to efficiently utilize the 32-bit word, two or four 8 bytes are computed in parallel way. This approach improves the performance by reducing the unused space in registers.

5.1. Speed-Optimized Implementation. The implementation on ARM Cortex-M3 focused on the fast execution timing rather than code size. Part of routines is unrolled to achieve the optimal performance.

5.1.1. Key Scheduling. We used 13 general purpose registers to maintain several variables. In particular, 1 register for master key pointer, 1 register for round key pointer, 4 delta variables, 1 register for counter, 2 registers for temporal variables, and 4 registers for round keys are allocated, respectively. In terms of computations, the key scheduling requires 7-bit or 8-bit wise ARX computations. Since rotation and XOR are linear operations, it can be easily

Table 9: Comparison of speed-optimized HIGHT block cipher implementations on 32-bit ARM Cortex-M3 in terms of code size (bytes), RAM (bytes), and execution time (cycles per byte).

Impl.	Code size (bytes)				RAM (bytes)			Execution (cycles per bytes)		
	EKS	ENC	DEC	SUM	EKS	ENC	DEC	EKS	ENC	DEC
Reference [13] w/LUT	316	860	896	1560	324	704	704	34	269	298
Reference [13] w/o LUT	316	344	384	1044	324	180	180	37	258	287
This work	154	352	352	858	316	180	180	33	197	234
AES [19]	410	1116	1564	2834	264	268	268	55	222	298

computed on 32-bit word machine. However, the addition operation is nonlinear operation, so performing 8-bit wise nonlinear operation on 32-bit word is inefficient. In Table 8, the 4-way addition with 7-bit delta and 8-bit operand is described. First, the lower 7-bit of operand is stored in the temporal register (i.e., TEMP). Next, two 7-bit variables are added, which may generate the 8th bit setting. In line 3 and 4, the most significant bit of operand is extracted and added to the results of 7-bit addition.

5.1.2. Encryption and Decryption. For encryption and decryption operations, we used 13 general registers. In particular, 1 register for block pointer, 1 register for round key pointer, 4 delta variables, 1 register for MASK, 1 register for counter, 2 registers for round key, and 3 registers for temporal variables are allocated, respectively. For the parallel computation, eight 8-bit bytes are stored in four 32-bit words with the margin. This representation efficiently performs the round function since in each round function two 8-bit operations are paired. The rotation operations for F0 and F1 functions are computed with the barrel-shifter feature, where the barrel-shifter does not consume additional execution timing. Same techniques are applied to the implementation of decryption.

5.1.3. Evaluation. Implementation of HIGHT block cipher on 32-bit ARM Cortex-M3 devices is previously suggested by Seo et al. [13]. The comparison results are given in Table 9. Compared with previous works, we improved the key scheduling operation with parallel addition techniques. This method handles four-byte addition in parallel way. The code size is reduced by 51% and execution time is improved by 1 cycle per byte. For encryption and decryption operations, we utilized general purpose registers to efficiently handle the data. The code size shows similar size but we achieved better performance than previous works by 23% and 18% for encryption and decryption operations, respectively.

6. Conclusion

One of the biggest challenges for IoT is secure and robust transactions between low-end IoT devices. In order to establish the secure communication channel, all message packets should be encrypted by using secure block ciphers. Block cipher operations should be performed without delays to increase the service availability. In terms of chip size, the low area implementation reduces production cost and peak power consumption. In this paper, we presented compact implementations of promising lightweight block cipher, namely, HIGHT, in software (i.e., 8-bit AVR and 32-bit ARM Cortex-M3) and hardware (i.e., ASIC). For high performance or low cost, several software and hardware optimization techniques are presented. Proposed methods are not limited to the HIGHT block cipher since the methods are generic approaches and can be easily used for other ARX block ciphers.

Data Availability

The source codes are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

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

References

[1] A. Juels and S. A. Weis, "Authenticating pervasive devices with human protocols," in *Proceedings of the Annual International Cryptology Conference*, pp. 293–308, Springer, Santa Barbara, CA, USA, August 2005.

[2] D. Hong, J. Sung, S. Hong et al., "HIGHT: a new block cipher suitable for low-resource device," in *Cryptographic Hardware and Embedded Systems—CHES*, pp. 46–59, Springer, Berlin, Germany, 2006.

[3] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 175, ACM, San Francisco, CA, USA, June 2015.

[4] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: an efficient MAC algorithm for 32-bit microcontrollers," in *Selected Areas in Cryptography–SAC*, pp. 306–323, Springer, Berlin, Germany, 2014.

[5] Y.-I. Lim, J.-H. Lee, Y. You, and K.-R. Cho, "Implementation of HIGHT cryptic circuit for RFID tag," *IEICE Electronics Express*, vol. 6, no. 4, pp. 180–186, 2009.

[6] P. Israsena and S. Wongnamkum, "Hardware implementation of a tea-based lightweight encryption for rfid security," in *RFID Security*, pp. 417–433, Springer, Berlin, Germany, 2008.
[7] J. Jean, A. Moradi, T. Peyrin, and P. Sasdrich, "Bit-sliding: a generic technique for bit-serial implementations of SPN-based primitives," *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 687–707, 2017.
[8] A. Corporation, "ATmega128(L) datasheet (rev. 2467O–AVR–10/06)," October 2006, http://www.atmel.com/dyn/resources/proddocuments=doc2467:pdf.
[9] 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 *Lightweight Cryptography for Security and Privacy*, pp. 3–20, Springer, Berlin, Germany, 2014.
[10] D. Hong, J. Lee, D. Kim, D. Kwon, K. H. Ryu, and D. Lee, "LEA: a 128-bit block cipher for fast encryption on common processors," in *Information Security Applications*, pp. 3–27, Springer, Berlin, Germany, 2013.
[11] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright, "Fast software AES encryption," in *Fast Software Encryption*, pp. 75–93, Springer, Berlin, Germany, 2010.
[12] H. Seo, K. An, and H. Kwon, "Compact LEA and HIGHT implementations on 8-bit AVR and 16-bit MSP processors," in *Proceedings of the International Workshop on Information Security Applications*, pp. 253–265, Springer, Jeju Island, Korea, August 2018.
[13] H. Seo, I. Jeong, J. Lee, and W.-H. Kim, "Compact implementations of ARX-based block ciphers on IoT processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 3, p. 60, 2018.
[14] H. Seo, "High speed implementation of LEA on ARM Cortex-M3 processor," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 22, no. 8, pp. 1133–1138, 2018.
[15] B. Koo, D. Roh, H. Kim, Y. Jung, D. Lee, and D. Kwon, "CHAM: a family of lightweight block ciphers for resource-constrained devices," in *Proceedings of the International Conference on Information Security and Cryptology*, pp. 3–25, Springer, Seoul, Korea, December 2017.
[16] P. Schwabe and K. Stoffelen, "All the AES you need on Cortex-M3 and M4," in *Proceedings of the International Conference on Selected Areas in Cryptography*, pp. 180–194, Springer, St. John's, Canada, August 2016.
[17] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong, "The SIMECK family of lightweight block ciphers," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 307–329, Springer, Saint Malo, France, September 2015.
[18] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A survey of lightweight-cryptography implementations," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 522–533, 2007.
[19] D. Dinu, A. Biryukov, J. Großschädl, D. Khovratovich, Y. Le Corre, and L. Perrin, "FELICS–fair evaluation of lightweight cryptographic systems," in *Proceedings of the NIST Workshop on Lightweight Cryptography*, Gaithersburg, MA, USA, July 2015.
[20] T. Eisenbarth, Z. Gong, T. Güneysu et al., "Compact implementation and performance evaluation of block ciphers in ATtiny devices," in *Proceedings of the Inter-national Conference on Cryptology in Africa*, pp. 172–187, Springer, Rabat, Morocco, July 2012.