Light-weight Block Cipher: HIGHT is Back!

No Author Given

No Institute Given

Abstract. Recent light-weight block cipher competition (FELICS Triathlon) evaluates the efficient implementations of block cipher for Internet of Things (IoT) applications. After the competitions, HIGHT is selected as one of the most efficient light-weight block cipher by considering the code size, memory and execution time. In this paper, we further investigate the light-weight features of HIGHT block cipher and present the optimization techniques over classical resource constrained microcontrollers and high-end embedded processors. The implementation results are evaluated on the 8-bit AVR and 32-bit ARM-NEON processors and compared with other representative light-weight block ciphers such as SPECK and SIMON.

Keywords: FELICS Triathlon, HIGHT, SPECK, SIMON, Internet of Things, Software Implementation

1 Introduction

Internet of Things (IoT) applications need to exchange the private and confidential information between various IoT platforms. In order to keep the information in secret, every network packet should be encrypted before any packet transmissions. IoT infrastructure consists of heterogeneous devices ranging from resource constrained microcontroller to high-end embedded processors. The microcontrollers are dispatched to the target place and transmit the sensor data to the server platform. However, the low-end platforms only support very limited computation power and storage due to production costs and battery issues so we need to find the compact implementations in terms of code size, memory and execution time for IoT services. The high-end embedded processors collect the huge amount of sensor data from many microcontrollers and need to process them without time delay. For this reason, the processors should provide the high-speed packet encryptions. In order to improve the encryption performance, the light-weight block ciphers are also actively studied. Recently Luxembourg university held light-weight block cipher competition. The purpose of competition is to find the best block cipher for IoT environments by considering the factors such as code size, memory and execution time. After the competitions, HIGHT block cipher is selected as one of the most efficient block cipher. In this paper, we explore the light-weight HIGHT block cipher and present the efficient implementation techniques for both classical resource constrained microcontrollers and high-end embedded processors for IoT services.

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)

The remainder of this paper is organized as follows. In Section 2, we introduce FELICS Triathlon competition, specifications of HIGHT and target platforms. In Section 3, we present the compact implementations of HIGHT block cipher. In Section 4, we evaluate the performance of proposed methods in terms of clock cycles and code size. Finally, Section 5 concludes the paper.

2 Related Works

2.1 FELICS Triathlon

In 2015, the open-source software benchmarking framework named Fair Evaluation of Lightweight Cryptographic Systems (FELICS) was held by Luxembourg university. The framework is similar to SUPERCOP but the system is targeting for embedded devices, which are widely used in IoT environments. Total three different platforms including 8-bit AVR, 16-bit MSP and 32-bit ARM were selected and three different metrics such as execution time, RAM and code size were evaluated. The implementations are tested in three different scenarios including cipher operation, communication protocol and challenge-handshake authentication protocol to select the best implementation of light-weight block cipher. In the FELICS Triathlon, more than one hundred different implementations of block and stream ciphers are submitted from international researchers. After the competition, LEA won first triathlon and HIGHT won second triathlon (See Table 1). The other block ciphers including SPECK and Chaskey also show the competitive performance [4, 13]. The one interesting insight is majority of light weight block ciphers follow the ARX architecture. Unlike traditional Substitution-Permutation-Networks (SPN), the round function involves only three operations including modular addition, rotation with fixed offsets and bit-wise exclusive-or. The ARX operations have many advantages over SPN variant. The operations are relatively fast and cheap in both hardware and software implementations. Furthermore, the operations are performed in constant time, which is even immune to timing attacks.

2.2 HIGHT Block Cipher

In CHES'06, a light-weight block cipher, HIGHT was introduced [12]. The HIGHT consists of simple ARX operations and supports 64-bit block size and 128-bit key size. The basic operations are 8-bit wise addition, exclusive-or and rotation and only 32 times of round functions are required. With these lightweight features, HIGHT block cipher can achieve the high performance in both low resource processors and high-end devices. The following are the notations of HIGHT block cipher.

Notations

- ☐ P: 64-bit plaintext, consisting of eight 8-bit words $P = P_7||P_6||...||P_1||P_0$
- ☐ C: 64-bit ciphertext, consisting of eight 8-bit words $C = C_7||C_6||...||C_1||C_0$
- ☐ X_i: 64-bit intermediate value (an input of i-th round in the encryption/decryption function), consisting of eight 8-bit words $X_i = X_{i,7}||X_{i,6}||...||X_{i,1}||X_{i,0}$
- ☐ MK: 128-bit (16 8-bit) master key. $MK = MK_{15}||MK_{14}||...||MK_1||MK_0$
- ☐ WK: 64-bit (8 8-bit) whitening key. $WK = WK_7||WK_6||\ ...||WK_1||WK_0$
- ☐ SK: sub-key, consisting of 128 8-bit words $SK = SK_{127}||SK_{126}||...||SK_1||SK_0$
- ☐ $X \oplus Y$: XOR (eXclusive OR) of bit strings X and Y with same length
- ☐ $X \boxplus Y$: Addition modulo 2^8 of 8-bit strings X and Y
- ☐ $X \boxminus Y$: Subtraction modulo 2^8 of 8-bit strings X and Y
- ☐ $X^{\lll i}$: Left rotation by i-bit on a 8-bit value X
- ☐ $F_0(X)$: First auxiliary function $(X^{\lll 1} \oplus X^{\lll 2} \oplus X^{\lll 7})$
- ☐ $F_1(X)$: Second auxiliary function $(X^{\lll 3} \oplus X^{\lll 4} \oplus X^{\lll 6})$

Key Schedule The key schedule function consists of WhiteningKeyGeneration and SubkeyGeneration. The WhiteningKeyGeneration generates the 8 whiteningkeybytes $(WK_0, ..., WK_7)$ for the initial and final transformations. The SubkeyGeneration generates the 128 subkeybytes $(SK_0, ..., SK_{127})$ for 32 rounds. The sub-algorithm, namely ConstantGeneration, supplies the 128 7-bit constants $(\delta_0, ..., \delta_{127})$ for the SubkeyGeneration. δ_0 is fixed as 1011010_2 and follows 7-bit LFSR $(x^7 + x^3 + 1 \in \mathbb{Z}_2[x])$. The detailed key schedule algorithm is drawn in Algorithm 1.

Encryption The encryption function consists of InitialTransformation, RoundFunction and FinalTransformation. The InitialTransformation transforms a plaintext $(P_0, ..., P_7)$ into the input of the first RoundFunction by using the 4 whiteningkeybytes $(WK_0, ..., WK_3)$. The RoundFunction consists of 32 rounds and uses two auxiliary functions (F_0 and F_1) to transform the input variables. The FinalTransformation outputs the ciphertext $(C_0, ...C_7)$ by using the 4 whiteningkeybytes $(WK_4, ..., WK_7)$. The detailed encryption algorithm is drawn in Algorithm 2.

Algorithm 1 Key Schedule

```
Require: Master key $MK$
Ensure: Whitening key $WK$, Sub-key $SK$
 1: for $i = 0$ to 7 do
 2:     if $0 \leqq i \leqq 3$ then
 3:         $WK_i \leftarrow MK_{i+12}$
 4:     else
 5:         $WK_i \leftarrow MK_{i-4}$
 6:     end if
 7: end for
 8: $s_0 \leftarrow 0,\ s_1 \leftarrow 1,\ s_2 \leftarrow 0,\ s_3 \leftarrow 1, s_4 \leftarrow 1,\ s_5 \leftarrow 0,\ s_6 \leftarrow 1$
 9: $\delta_0 \leftarrow s_6||s_5||s_4||s_3||s_2||s_1||s_0$
10: for $i = 1$ to 127 do
11:     $s_{i+6} \leftarrow s_{i+2} \oplus s_{i-1},\ \delta_i \leftarrow s_6||s_5||s_4||s_3||s_2||s_1||s_0$
12: end for
13: for $i = 0$ to 7 do
14:     for $j = 0$ to 7 do
15:         $SK_{16\cdot i+j} \leftarrow MK_{j-i \ mod \ 8} \boxplus \delta_{16\cdot i+j}$
16:     end for
17:     for $j = 0$ to 7 do
18:         $SK_{16\cdot i+j+8} \leftarrow MK_{j-i \ mod \ 8} \boxplus \delta_{16\cdot i+j+8}$
19:     end for
20: end for
21: return $WK$, $SK$
```

Decryption The decryption function follows the reverse order of encryption function. The $\boxplus$ operation is replaced into $\boxminus$.

2.3 8-bit Embedded Platform AVR

The 8-bit AVR embedded processor is equipped with an ATmega128 8-bit processor clocked at 7.3728 MHz [1]. It has a 128 KB EEPROM chip and 4 KB RAM chip. The ATmega128 processor has RISC architecture with 32 registers. Among them, 6 registers serve as the special pointers for operand and result in indirect address mode. The remaining 26 general purpose registers are available for arithmetic operations. One arithmetic instruction requires only one clock cycle, and memory instructions or 8-bit wise multiplication require two processing cycles. With these instructions, 8-bit wise ARX operations are simply performed with combinations of one or two basic instructions. On the AVR processor, many block cipher algorithms are evaluated. The AES block cipher was implemented in 1,993 cycles for encryption since AES consists of 8-bit wise instructions, which are efficient over 8-bit platform [14]. Recently, ARX-based block ciphers (SPECK and SIMON) introduced by NSA show high performance on AVR processors [4, 3].

Algorithm 2 Encryption

```
Require: Plaintext $P$, Whitening key $WK$, Sub-key $SK$
Ensure: Ciphertext $C$
 1: $X_{0,0} \leftarrow P_0 \boxplus WK_0,\ X_{0,1} \leftarrow P_1\ ,\ X_{0,2} \leftarrow P_2 \oplus WK_1,\ X_{0,3} \leftarrow P_3$
 2: $X_{0,4} \leftarrow P_4 \boxplus WK_2,\ X_{0,5} \leftarrow P_5\ ,\ X_{0,6} \leftarrow P_6 \oplus WK_3,\ X_{0,7} \leftarrow P_7$
 3: for $i = 0$ to 31 do
 4:     $X_{i+1,1} \leftarrow X_{i,0},\ X_{i+1,3} \leftarrow X_{i,2},\ X_{i+1,5} \leftarrow X_{i,4},\ X_{i+1,7} \leftarrow X_{i,6}$
 5:     $X_{i+1,0} \leftarrow X_{i,7} \oplus (F_0(X_{i,6}) \boxplus SK_{4i+3}),\ X_{i+1,2} \leftarrow X_{i,1} \boxplus (F_1(X_{i,0}) \boxplus SK_{4i+2})$
 6:     $X_{i+1,4} \leftarrow X_{i,3} \oplus (F_0(X_{i,2}) \boxplus SK_{4i+1}),\ X_{i+1,6} \leftarrow X_{i,5} \boxplus (F_1(X_{i,4}) \boxplus SK_{4i})$
 7: end for
 8: $C_0 \leftarrow X_{32,1} \boxplus WK_4,\ C_1 \leftarrow X_{32,2}\ ,\ C_2 \leftarrow X_{32,3} \oplus WK_5,\ C_3 \leftarrow X_{32,4}$
 9: $C_4 \leftarrow X_{32,5} \boxplus WK_6,\ C_5 \leftarrow X_{32,6}\ ,\ C_6 \leftarrow X_{32,7} \oplus WK_7,\ C_7 \leftarrow X_{32,0}$
10: return $C$
```

2.4 32-bit Embedded Platform ARM-NEON

Advanced RISC Machine (ARM) is an instruction set architecture (ISA) design by ARM for high-performance 32-bit embedded applications. Although ARM cores are usually more larger and complex than AVRs and MSPs, most ARM designs ensure the low power consumptions and high code density. The ARM family has developed from the traditional ARM1 to advanced Cortex architectures in these days. The ARMv7 models provide large number of pipeline stages and various caches, SIMD extensions (i.e. NEON engine) and multiple load/store architecture. Most instructions of the ARM are performed in a single cycle except memory access and some of arithmetic instructions. Particularly, the in-line barrel shifter instruction performs the rotation/shift operations on the second operand without timing delays.

NEON is a 128-bit SIMD architecture for the ARM Cortex-A series. One of the biggest difference between traditional ARM processors and new ARM-NEON processors is NEON engine features. The NEON engine offers 128-bit wise registers and instructions. Each register is considered as a vector of elements of the same data type and this data type can be signed/unsigned 8-bit, 16-bit, 32-bit, or 64-bit. This feature provides a more precise operation in various word sizes, which allows us to perform multiple data in single instruction. With this features, the NEON engine can accelerate data processing by at least 3X that provided by ARMv5 and at least 2X that provided by ARMv6 SIMD instructions. For 8-bit wise ARX operations, we utilized the NEON instructions, which support 8-bit wise vector instruction sets. There are many works suggested on NEON architecture. In CHES 2012, NEON-based cryptography implementations including Salsa20, Poly1305, Curve25519 and Ed25519 were presented [6]. In order to enhance the performance, the authors provided novel rotation and compact multiplication-reduction operations with NEON instructions. In CT–RSA'13, ARM–NEON implementation of `Grøstl` shows that 40 % performance enhancements than the previous ARM implementation [11]. In HPEC 2013, a multiplicand reduction method for ARM-NEON was introduced for the NIST curves [10]. In CHES 2014, the Curve41417 implementation adopts 2-level

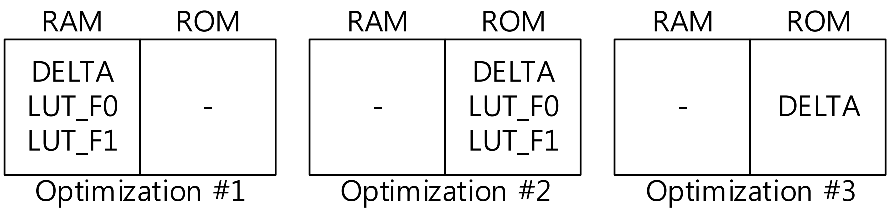

Fig. 1. Three different storage utilizations for optimized HIGHT implementations

Karatsuba multiplication in the redundant representation [5]. In ICISC 2014, Seo et al. introduced a novel 2-way Cascade Operand Scanning (COS) multiplication for RSA implementation [16]. Recently, Ring-LWE implementation is also accelerated by taking advantages of NEON instructions [2].

3 Proposed Method

The 8-bit AVR processor equips the limited computing power and storage capacities. We need to carefully design the algorithm to meet the requirements of speed and size factors for IoT applications. For 32-bit ARM-NEON processor, the architecture has powerful hardware specifications and supports the SIMD instruction sets. Unlike traditional SISD instructions, we need to parallelize the sequential data type. In this section, we introduce compact HIGHT implementations over both low-end and high-end processors.

3.1 HIGHT Implementations over AVR

The key scheduling performs both Whitening-Key-Generation and Sub-Key-Generation steps. The Whitening-Key-Generation only needs to copy the part of master keys to Whitening-Keys. For Sub-Key-Generation, the constant delta variables should be generated by following LFSR computations over primitive polynomial ($x^7 + x^3 + 1$). Since the LFSR computation consists of bit by bit operations, the computation over ordinary word-wise platform is not favorable to handle the bit-wise operations. The initial delta variable is fixed to 1011010_2, which allows us to pre-compute the whole delta variables in off-line by utilizing the storages. By replacing the constant variable generation into LUT access operation, the bit-wise exclusive-or instructions with master keys and delta variables to generate the sub-key are only required in the key scheduling algorithm.

The encryption performs Initial Transformation, Round Function and Final Transformation. Both Initial and Final Transformations require 8-bit wise addition and bit-wise exclusive-or operations. This can be performed with the 8-bit wise `add` and `eor` instructions. The Round function consists of 8-bit wise addition, bit-wise exclusive-or operations and two auxiliary functions ($F_0 = x^{\lll 1} \oplus x^{\lll 2} \oplus x^{\lll 7}$ and $F_1 = x^{\lll 3} \oplus x^{\lll 4} \oplus x^{\lll 6}$). Each auxiliary function requires three times of rotations and two bit-wise exclusive-or operations. Since the each function has 8-bit input and output, we can pre-compute the whole results ($256 = 2^8$) and store them into LUT. By replacing the operations into LUT access, we can avoid several basic instructions, which improves the performance significantly. The LUT access model is also divided into two models by considering storage types including RAM or ROM. The RAM storage requires two clock cycles to access but ROM requires three clock cycles. However, the storage of RAM are smaller and more scarce resources than that of ROM so the implementation techniques are determined by considering the specifications of target platforms. Alternatively we can directly perform the both auxiliary functions without LUT. The first auxiliary function ($F_0 = x^{\lll 1} \oplus x^{\lll 2} \oplus x^{\lll 7}$) is re-organized into ($F_0 = x^{\lll 1} \oplus x^{\lll 2} \oplus swap(x^{\lll 3})$), which saves one time of rotation, where *swap* operation exchanges the lower 4-bit and the higher 4-bit in single instruction. The second auxiliary function ($F_1 = x^{\lll 3} \oplus x^{\lll 4} \oplus x^{\lll 6}$) is also re-organized into ($F_1 = x^{\lll 3} \oplus x^{\lll 4} \oplus swap(x^{\lll 2})$), which saves two times of rotation. For efficient 8-bit rotation, the shift operation by $(5, 6, 7)$-bit can be replaced by $(3, 2, 1)$-bit shifts in counter direction. The above three approaches have different storage usages as described in Figure 1. For all cases, we used pre-computed delta variables since the manual computation routine requires long execution time and code size. The optimization #1 stores the whole precomputed information into the RAM so the performance is the highest but it requires huge RAM storage. The optimization #2 stores the LUT into the ROM, which degrades the performance but this utilizes the relatively large ROM storage. The optimization #3 only store the delta variables into the ROM so the minimum storage is required but the performance is the lowest among the approaches. Since HIGHT decryption has a similar structure of encryption, the techniques for encryption is applied to the decryption in reverse order.

Particularly the HIGHT block cipher requires 64-bit wise block size and 32-bit wise round key in each round. This light-weight property requires the very small number of working registers. By using small number of registers, we can avoid to use callee-saved registers, which require `PUSH/POP` instructions before function call. Furthermore, we can store whole round keys and plaintext into the registers during computations. For efficient register alignments, `MOVW` instruction relocates two adjacent registers to destination within single clock cycle. In case of memory load/store, we used the post-increment or pre-decrement accesses. This does not impose additional clock costs to calculate the indirect address. In order to set the counter, we use `LDI` instruction to assign the value directly to the registers. The `ADIW` and `SBIW` instructions conduct addition and subtraction by word with immediate value. These operations are used to modify memory address. Finally, `INC` and `DEC` operations are used to increase and decrease the counter variables. Another considerations are pre-computation of round key. If the target platform has enough RAM to store the parameters, whole round key pairs can be pre-computed before encryption or decryption operation, which can avoid the overheads of key generation. For key scheduling operation, we firstly load secret key pairs and delta variables by the number of rounds. After key generation, we stored whole round key pairs into RAM. For encryption operation, we

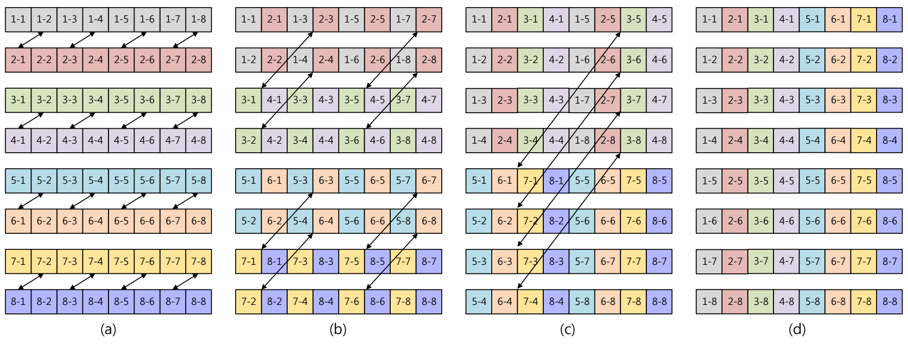

Fig. 2. Data alignments with transpose operations, (a) 8-bit wise, (b) 16-bit wise, (c) 32-bit wise, (d) completed alignment

load plaintext together with round keys and perform the encryption step without key generation. In case of decryption, we load ciphertext and access round keys in reverse order to perform decryption step. On the other hands, on-the-fly method generates round key on the spot and then directly encrypts plaintext with these round key pairs. This approach reduces the RAM consumptions but every encryption requires key scheduling, which degrades the performance.

Lastly, the 32-round of computations are iteration of 4 8-round computations. So we only write 8-round operations in optimal way and four more outer iteration is performed. If we write the program in looped way, the size of source code (S) is significantly reduced to $(\frac{S}{N} + A)$ with small overheads for loop controls, where (N) indicates number of iteration and (A) represents overheads for counter, offset and branch operations.

3.2 HIGHT Implementations over ARM-NEON

In order to perform the parallel computations, the input data should be aligned for SIMD friendly format. The HIGHT block cipher has 64-bit block size and every operation is performed in 8-bit wise. Each 8-bit data is encrypted in different computations. We group the each 8-bit data with same index in the plaintext. The NEON register supports 128-bit and this can contain 16 8-bit data packets. In our implementation, we perform the encryption with 16 different plaintext. In Figure 2, the data alignments for 64-bit register case is described[1]. Each row describes the 64-bit wise register. The register consists of 8 8-bit wise data sets. Total three times of transpose operations with 8-bit, 16-bit and 32-bit wise are performed subsequently and these operations output the fully aligned data sets. Each row contains same index of different plaintexts.

The most expensive operations for HIGHT are auxiliary functions in encryption and decryption steps. The auxiliary functions consist of bit-wise exclusive-

[1] The data alignments for 128-bit register case can be easily accomplished with simple modifications from presented example.

Table 2. Pre-computed look-up tables for auxiliary functions in 4-bit wise

Auxiliary function	Look-up table values
F0_LOW	0xA523A82E_BF39B234_91179C1A_8B0D8600
F0_HIGH	0x5A328AE2_FB932B43_1971C9A1_B8D06800
F1_LOW	0x4B13FBA3_2A729AC2_89D13961_E8B05800
F1_HIGH	0xB431BF3A_A227A92C_981D9316_8E0B8500

Table 3. Parallel Encryption in OpenMP

```
#pragma omp parallel private(id) shared(PLAINTEXT)
    #pragma omp for
      for(id=0;id<TOTAL;id++)
          Encryption(ROUNDKEY,PLAINTEXT[8×id]);
```

or and rotation operations in 8-bit format. We first implement the function in straight-forward way with SIMD instructions. The NEON engine supports vectorized instruction so we can perform 16 different data set at once with single instruction. This improves the performance over large word processor such as 32-bit ARM processor. The alternative approach is exploiting the pre-computed look-up table. The auxiliary functions have 8-bit input and output values. The ARM-NEON processor supports 4-bit wise LUT operation (`VTBL`) and this can translate the each auxiliary function into 2 4-bit LUT operations. We precomputed all LUT values as described in Table 2 and perform the each auxiliary function with 2 LUT accesses with low and high 4-bit index in the input variables.

We further optimize the register usages in order to reduce the memory accesses[2]. The proposed implementations used 8 registers for plaintext, 1 registers for round keys, 4 registers for LUT and 3 registers for temporal registers. For round key access, the one round key is loaded to the ARM's `R` registers and then this 32-bit variable is duplicated by 4 times to the NEON registers in each round with `vdup` instruction. Furthermore, recent ARM-NEON processors have multiple cores and each core can perform independent work loads in parallel way. In order to exploit the full capabilities of multiple processing, we used SIMT programming library, namely OpenMP. Our target device has four physical cores, which can lead to theoretically 4 times of performance improvements. We re-scheduled previous our implementations using OpenMP programming to support parallel functions. The detailed program codes are described in Table 3. The `id` variables are critical data and defined as `private`. The `PLAINTEXT` is defined as `shared` variables. The `PLAINTEXT` is indexed by `id` variables, which ensures that each thread accesses to different data. We assigned each encryption operation into single thread and the four different encryptions are performed in four different cores. Each core performs 8 encryptions at once and total 32 (8×4) parallel encryptions are performed in 4 different cores.

[2] NEON architecture has 16 128-bit `Q` registers

Table 4. Comparison of scaled overhead for block ciphers (64-bit block, 128-bit key), †: excluding initial and final rounds

Features	HIGHT	SPECK	SIMON
Scaled in 8-bit addition	4	4	–
Scaled in 8-bit bit-wise and	–	–	4
Scaled in 8-bit exclusive-or	12	8	12
Scaled in 8-bit rotation (offset)	16	12	8
Round key in each round (byte)	4	4	4
Number of Rounds	$32^{\dagger}$	27	44
Scaled speed (cost/penalty)	1,280 / –	864 / 1.5	1,408 / 0.9
Scaled size per round (cost/penalty)	72 / –	56 / 1.3	56 / 1.3

4 Results

4.1 Scaled Performance

In this section, we evaluate the representative light-weight block ciphers including HIGHT, SPECK and SIMON in terms of execution time (in clock cycle) and code size (in byte). The block ciphers share simple ARX architectures but detailed features are different to each other. In order to draw the fair comparison results, we scaled the all operations into 8-bit wise basic operations. In terms of execution time, the clock cycle is scaled in following equation {`(#add + #and + #xor + #rotation + #round-key` × `2)`× `#round`}. The basic arithmetic and logical operations require only one clock cycle and memory access operation require 2 clock cycles per each byte. When the one round complexity is calculated, the clock cycle is multiplied by the number of rounds to get total complexity. For code size, every single instruction requires 2 bytes in 8-bit processor. The scaled code size is drawn as following equation (`#add + #and + #xor + #rotation + #round-key`)× `2`.

The HIGHT block cipher consists of only 8-bit operations and the scaled complexity is 1,280 $\{(4{+}0{+}12{+}16{+}2{\times}4){\times}32\}$. By contrast, SPECK and SIMON with 64-bit block size versions require 32-bit wise operations. The execution time is scaled and the scaled complexities are 864 $\{(4+0+8+12+2\times 4)\times 27\}$ and 1,408 $\{(0+4+12+8+2\times 4)\times 44\}$ for SPECK and SIMON, respectively. In the same manner, the code size costs are calculated. The HIGHT, SPECK and SIMON require 72 $\{(4+0+12+16+4)\times 2\}$, 56 $\{(4+0+8+12+4)\times 2\}$ and 56$\{(0+4+12+8+4)\times 2\}$, respectively. The detailed comparison results are drawn in Table 4.

4.2 Performance Comparison

In Table 5, the comparison results of separated implementations in terms of scaled execution time and code size are drawn. The SPECK shows the highest speed among three block ciphers since the block cipher requires small number of rounds. However, the SPECK needs long offsets for the rotation operation,

Table 5. Separated encryption results in execution time (in cycle/byte) and code size (in byte) on AVR platform; †: scaled results

Method	ENC	ENC†	ROM	ROM†	RAM
HIGHT w/ RAM LUT	216	**216**	254	**254**	640
HIGHT w/ ROM LUT	232	**232**	766	**766**	136
HIGHT w/o LUT	320	**320**	248	**248**	136
HIGHT [7]	346	**346**	286	**286**	136
SPECK [7]	125	**188**	542	**705**	100
SPECK [3]	122	**183**	628	**816**	108
SIMON [7]	224	**202**	354	**460**	176
SIMON [3]	221	**199**	436	**567**	176

Table 6. On-the-fly encryption/decryption results in execution time (in cycle/byte) and code size (in byte) on AVR platform

Method	ENC	DEC	ROM	RAM
HIGHT w/ RAM LUT	325	334	672	640
HIGHT w/ ROM LUT	357	366	1,312	–
HIGHT w/o LUT	452	461	912	–
HIGHT [9]	371	371	5,672	–
HIGHT [8]	2,438	2,520	402	32

which introduces the large code size (705 bytes). The SIMON requires many number of round function but the rotation operation is simpler than that of SPECK. This feature introduces the lower performance but smaller code size than the SPECK. The HIGHT block cipher is implemented in three different approaches. Among them, w/o LUT approach shows the most balanced results in terms of execution time, RAM and ROM. The implementation introduces the encryption in 320 cycle/byte together with 136 bytes of RAM and 248 bytes of ROM. The execution timing is the lowest among three block ciphers. However, this is practically fast enough and code size is the smallest in the list. The results show that all light-weight block ciphers are strong players so we can choose the certain block cipher by considering the specifications of target platforms.

The alternative approach is the on-the-fly implementation, which generates the round keys in each round and uses them directly in encryption or decryption operation so the implementation does not require additional storage for round keys. Since the round key generation can be performed in parallel fashion, the decryption round keys are computed in parallel way. The detailed results are described in Table 6. For comparison, w/o LUT implementation shows the most balanced performance since w/ ROM LUT requires huge LUT on the ROM storage. Even though the minimum ROM consumption is achieved by [8], the result shows high overheads on execution time.

This paper presents the first HIGHT implementations over NEON platform. For this reason, we compared our HIGHT implementations and SPECK/SIMON block ciphers as described in Table 7, which have same block size and key size.

Table 7. Separated encryption results in execution time (in cycle/byte) on ARM-NEON platform; †: scaled results

Method	`ENC`	`ENC`†
HIGHT w/ LUT	18.7	**18.7**
HIGHT w/o LUT	25.9	**25.9**
SPECK [15]	16.5	**24.8**
SIMON [15]	31.9	**28.7**

Table 8. Performance evaluation of multiple-thread encryptions

No. thread	1	2	3	4	5	6	7	8
cycle/byte	18.7	11.5	7.7	6.0	7.9	7.1	7.1	6.5

The HIGHT implementations consist of w/ LUT and w/o LUT approaches. The w/ LUT method efficiently performs the auxiliary functions and shows better performance than w/o LUT approach. Compared with other light-weight block ciphers such as SPECK and SIMON, the proposed HIGHT implementations show higher performance than that of SPECK and SIMON.

The performance is accelerated again by exploiting the full multiple cores in the platforms. Our target platform is quad-core architecture so it improves the performance by increasing the number of threads. Interestingly, the performance decreases when the number of threads is larger than the number of cores, because many number of threads requires a number of context-switching procedures and this causes performance bottleneck. The detailed results are described in Table 8. Finally, fully parallelized HIGHT implementations achieved the 6.0 cycle/byte with 4 threads. This is the highest performance reported ever[3].

5 Conclusion

One of the biggest challenges for Internet of Things (IoT) applications are the secure and robust transactions between resource constrained embedded processor and server platform. In order to ensure the secure communications, we should encrypt the sensitive and confidential information by using secure block cipher operations. The operations should be performed in short execution time to reduce the energy consumptions and increase the network availability. In this paper, we explore the compact implementations of one of the most promising light-weight block cipher, namely HIGHT over both 8-bit AVR and 32-bit ARM-NEON architectures. For high performance, auxiliary functions are intensively investigated and several optimization techniques are introduced. The implementations are evaluated on the target processors and high performance is achieved over both platforms.

[3] For reproduction of results, the source codes will be public domain

References

1. Atmel Corporation. ATmega128(L) Datasheet (Rev. 2467O–AVR–10/06). Available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, Oct. 2006.
2. R. Azarderakhsh, Z. Liu, H. Seo, and H. Kim. NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture.
3. 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*, pages 3–20. Springer, 2014.
4. 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*, page 175. ACM, 2015.
5. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer strikes back: new DH speed records. In *Advances in Cryptology–ASIACRYPT 2014*, pages 317–337. Springer, 2014.
6. D. J. Bernstein and P. Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.
7. D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov. FELICS Triathlon. Available for download at `https://www.cryptolux.org/index.php/FELICS_Triathlon`, 2016.
8. 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 *Progress in Cryptology–AFRICACRYPT 2012*, pages 172–187. Springer, 2012.
9. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, (6):522–533, 2007.
10. A. Faz-Hernández, P. Longa, and A. H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV–GLS curves. In *Topics in Cryptology–CT-RSA 2014*, pages 1–27. Springer, 2014.
11. S. Holzer-Graf, T. Krinninger, M. Pernull, M. Schläffer, P. Schwabe, D. Seywald, and W. Wieser. Efficient vector implementations of AES-based designs: a case study and new implemenations for Grøstl. In *Topics in Cryptology–CT-RSA 2013*, pages 145–161. Springer, 2013.
12. 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 *Cryptographic Hardware and Embedded Systems–CHES 2006*, pages 46–59. Springer, 2006.
13. 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 2014*, pages 306–323. Springer, 2014.
14. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.
15. T. Park, H. Seo, and H. Kim. Parallel implementations of SIMON and SPECK. In *2016 International Conference on Platform Technology and Service (PlatCon)*, pages 1–6. IEEE, 2016.
16. H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim. Montgomery modular multiplication on ARM-NEON revisited. In *Information Security and Cryptology–ICISC 2014*, pages 328–342. Springer, 2014.