Compact LEA and HIGHT Implementations on 8-bit AVR and 16-bit MSP Processors

Hwajeong Seo*, Kyuhwang An, and Hyeokdong Kwon

Hansung University, Republic of Korea
hwajeong84@gmail.com, tigerk9212@gmail.com, hdgwon@naver.com

Abstract. In this paper, we revisited the previous LEA and HIGHT implementations on the low-end embedded processors. First, the general purpose registers are fully utilized to cache the intermediate results of delta variable during key scheduling process of LEA. By caching the delta variables, the number of memory access is replaced to the relatively cheap register access. Similarly, the master key and plaintext are cached during key scheduling and encryption of HIGHT block cipher, respectively. Second, stack storage and pointer are fully utilized to store the intermediate results and access the round keys. This approach solves the limited storage problem and saves one general purpose register. Third, indirect addressing mode is more efficient than indexed addressing mode. In the decryption process of LEA, the round key pair is efficiently accessed through indirect addressing with minor address modification. Fourth, 8-bit word operations for HIGHT is efficiently handled by 16-bit wise instruction of 16-bit MSP processors. Finally, the proposed LEA implementations on the representative 8-bit AVR and 16-bit MSP processors are fully evaluated in terms of code size, RAM and execution timing. The proposed implementations over the target processors (8-bit AVR processor, 16-bit MSP processor) are faster than previous works by (13.6%, 9.3%), (0.6%, 8.5%), and (3.4%, 1.5%) for key scheduling, encryption, and decryption, respectively. Similarly, the proposed HIGHT implementations on the 16-bit MSP processors are faster than previous works by 38.6%, 33.7%, and 33.6% for key scheduling, encryption, and decryption, respectively.

Keywords: LEA, HIGHT, AVR, MSP, Software Implementation

1 Introduction

In WISA'13, Lightweight Encryption Algorithm (LEA) was announced by the Attached Institute of ETRI [4]. Unlike previous Substitute Permutation Network (SPN) block ciphers, LEA algorithm follows simple Addition Rotation XOR (ARX) architecture which shows high speed computations in software and hardware implementations. Furthermore, the implementation is a regular fashion, which is secure against timing attack. In ICISC'13, LEA implementations on

* Corresponding Author

high-end ARM-NEON and GPGPU platforms are introduced [11]. The works present efficient parallel implementations in Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Thread (SIMT). In WISA'15, low-end 8-bit AVR processor is also evaluated [10]. The author presented speed and size optimized LEA implementation techniques for 8-bit AVR processors and compared the performance with representative ARX block ciphers, including SPECK and SIMON. The work proved that LEA is the most optimal block cipher for embedded environments. In WISA'16, both ARM and NEON instruction sets are fully utilized in interleaved way [12]. The work showed the interleaved approach efficiently hides the pipeline stalls between ARM and NEON instruction sets. Recently, block cipher competition is held by Luxembourg University (FELICS Triathlon). Many light-weight block ciphers are submitted and finally LEA and HIGHT implementations achieved the efficient block cipher implementations for Internet of Things (IoT) by considering three factors, including RAM, ROM, and execution timing [6]. The descriptions of compact LEA implementations are well described in [9]. In CHES'06, lightweight block cipher, HIGHT, was introduced [5]. The HIGHT block cipher consists of simple 8-bit wise ARX operations. The lightweight implementations were also reported in [9]. As listed above, many works have proved that LEA and HIGHT block ciphers are the promising block ciphers for both high-end computers and low-end microprocessors. However, still there are large room to improve the performance for 8-bit AVR and 16-bit MSP processors. In this paper, we re-visit previous results of LEA and HIGHT implementations on the 8-bit AVR and 16-bit MSP processors

The remainder of this paper is organized as follows. In Section 2, we recap the basic specifications of LEA and HIGHT block ciphers, FELICS triathlon, and target 8-bit AVR and 16-bit MSP processors. In Section 3, we present the compact implementations of LEA and HIGHT block ciphers on 8-bit AVR and 16-bit MSP processors. In Section 4, we evaluate the performance of proposed methods in terms of code size, RAM, and execution timing. Finally, Section 5 concludes the paper.

2 Related Works

2.1 LEA Block Cipher

In WISA 2013, Lightweight Encryption Algorithm (LEA) was announced by the Attached Institute of ETRI [4]. The LEA block cipher only consists of simple Addition-Rotation-XOR (ARX) operations, which replaces the expensive S-box operations. These lightweight features are appropriate to achieve the high performance for both software and hardware platforms. Furthermore, ARX operations are secure against timing attack and simple power analysis.

2.2 HIGHT Block Cipher

In CHES'06, lightweight HIGHT block cipher was introduced [5]. HIGHT block cipher also consists of ARX operations and supports 64-bit block size and 128-bit key size. The basic operations are 8-bit wise addition, rotation, and XOR,

Table 1. First and second 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)

and the number of round is 32. HIGHT block cipher is particularly efficient for low-end device and hardware implementations.

2.3 FELICS Triathlon

In 2015, the open-source software benchmarking framework named Fair Evaluation of Lightweight Cryptographic Systems (FELICS) was held by Luxembourg University. This is similar to SUPERCOP benchmark framework but the system is particularly targeting for low-end embedded processors, which are widely used in IoT and M2M services. 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 evaluated in three different scenarios including cipher operation, communication protocol, and challenge-handshake authentication protocol. In the FELICS Triathlon, more than one hundred different implementations of block and stream ciphers are submitted by world-wide researchers. In 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 on low-end processors [1, 6].

2.4 8-bit Embedded Platform AVR

Table 2. Instruction set summary for AVR

asm	Operands	Description	Operation	#Clock
ADD	Rd, Rr	Add without Carry	Rd ← Rd+Rr	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd⊕Rr	1
LSL	Rd	Logical Shift Left	C|Rd ← Rd<<1	1
ROL	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1

The 8-bit AVR embedded processor is equipped with an ATmega128 8-bit processor clocked at 7.3728 MHz. 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 ($r_{26} \sim r_{31}$) serve as the special pointers for indirect addressing. The remaining 26 registers are available for arithmetic operations. One

arithmetic instruction incurs one clock cycle, and memory instructions or 8-bit multiplication incurs two processing cycles. The detailed instructions are given in Table 2. Previous 8-bit microprocessor results showed that LEA is estimated to run at around 3,040 cycles for encryption on AVR AT90USB82/162 where AES best record is 1,993 cycles [4, 7]. The paper does not explore the specific LEA implementation techniques, but we assume that they used the separated mode to optimize performance in terms of speed by considering high performance. In case of AES encryption, they used the conventional lookup-based approach to reduce memory consumption [7]. The lookup tables are the forward and inverse S-boxes in total 512 bytes, because 32-bit look-up table access is not favorable for the low-end processors due to the limited storages. S-box pointer is always placed in Z register and the variable is stored into SRAM for fast access speed. For the efficient mix-column computation, a left shift with conditional branch to skip the bit-wise exclusive-or operation is established. Finally, the MixColumns step is implemented without the use of lookup tables as a series of register copies, XOR operations, taking a total of 26 cycles. The InvMixColumns step is implemented in a similar way, but it is more complicated routines, which takes a total of 42 cycles.

2.5 16-bit Embedded Platform MSP

Table 3. Instruction set summary for MSP

asm	Operands	Description	Operation	#Clock
ADD	Rr, Rd	Add without Carry	Rd ← Rd+Rr	1
XOR	Rr, Rd	Exclusive OR	Rd ← Rd⊕Rr	1
RLA	Rd	Logical Shift Left	C|Rd ← Rd<<1	1
RLC	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1

The MSP430 is a representative 16-bit embedded processor board with a clock frequency of 8∼16MHz, 32∼48KB of flash memory, 10KB of RAM, and 12 general purpose registers from r4 to r15 available [3]. Since these registers share pointer and user defined registers, the number of registers are much constrained than 8-bit AVR processors. The device also provides various arithmetics supporting full functions of ARX operations (See Table 3).

In [2], the implementation is based on the byte-oriented version, but the author has modified it to take advantage of the 16-bit platform. The first change was to improve the AddRoundKey function computing the bit-wise exclusive-or of 128-bit blocks in order to bit-wise exclusive-or 16-bit words at a time. The second change was to improve the use of 16-bit friendly lookup tables. The Subbytes, Shiftrows and Mixcolumns steps are combined in a single computation. This is well known 32-bit optimization of using precomputed tables with 256

Table 4. 32-bit rotations on 8-bit AVR, where Z and T are zero and temporal registers [9]

$\lll 1$	$\lll 8$	$\lll 16$ ($\ggg 16$)	$\ggg 8$	$\ggg 1$
`LSL X1`	`MOV T, X4`	`MOV T, X1`	`MOV T, X1`	`BST X1, 0`
`ROL X2`	`MOV X4, X3`	`MOV X1, X3`	`MOV X1, X2`	`LSR X4`
`ROL X3`	`MOV X3, X2`	`MOV X3, T`	`MOV X2, X3`	`ROR X3`
`ROL X4`	`MOV X2, X1`	`MOV T, X2`	`MOV X3, X4`	`ROR X2`
`ADC X1, Z`	`MOV X1, T`	`MOV X2, X4`	`MOV X4, T`	`ROR X1`
		`MOV X4, T`		`BLD X4, 7`
5 cycles	5 cycles	6 cycles	5 cycles	6 cycles

elements method. They exploited this for 16-bit version and it costs around 2 KB rather than 4 KB, conventional approach requires.

3 Proposed Method

3.1 Optimization of LEA for 8-bit AVR Processors

For LEA implementation on 8-bit AVR, 32-bit ARX operations with the 8-bit instructions should be optimized. The 32-bit addition in 8-bit instruction can be implemented in four consecutive 8-bit addition instructions (`add, adc`). Similarly, the 32-bit exclusive-or operation is implemented with four 8-bit exclusive-or instructions (`eor`). For the 32-bit rotation operations, assembly-level optimizations are required to get a small code size and fast execution timing. The optimized rotations are described in Table 4 [9].

In addition, the special right rotation offset by 3-bit is optimized through the special routine by 1 clock cycle (See Table 5). Instead of bit selection (`bst` and `bld`), the rotation bits are cached in temporal registers and applied to the destination register at last. This approach even reduces the 1 line of code size in each round.

Key Scheduling The key scheduling process generates the round key from master key and delta variables. For the high performance implementation, the number of memory access should be optimized. 128-bit master keys are reserved in the 16 general purpose registers. The part of delta variable (32-bit) are also reserved in the 4 general purpose registers, which are directly utilized when the delta variables are required. The remaining delta variables (96-bit) are loaded and stored again in the stack storages to avoid the overwriting of original delta variables in the memory.

Encryption & Decryption The encryption process consists of 24 rounds. Since same process is iterated in every 4 rounds, the length of loop is set to 4.

Table 5. 32-bit rotations on 8-bit AVR by 3-bit offsets, where $X1 \sim X4$ are data registers and T is temporal register [9]

without `bst/bld`	with `bst/bld`
`CLR T`	`Iteration #3{`
	`BST X1, 0`
`Iteration #3{`	`LSR X4`
`LSR X4`	`ROR X3`
`ROR X3`	`ROR X2`
`ROR X2`	`ROR X1`
`ROR X1`	`BLD X4, 7 }`
`ROR T }`	
`EOR X4, T`	
17 cycles	18 cycles

16 and 8 general purpose registers are assigned to the plaintext and temporal storages, respectively. The callee-saved register pair `(R28,R29)` is not used, which saves 2 bytes and 8 clock cycles for `PUSH` and `POP` instructions. The rotation operations are implemented in techniques of Table 4 and 5. The decryption process is reversed order of encryption. Main difference is round key access. The last index of round key is accessed and decremented to the first index. With this order of round key, the ciphertext is decrypted to the plaintext.

3.2 Optimization of LEA for 16-bit MSP Processors

The 32-bit addition operation on 16-bit MSP is implemented with two 16-bit addition instructions (`add, addc`). The 32-bit exclusive-or operation is implemented with two 16-bit exclusive-or instructions (`xor`). The optimized rotation techniques on 16-bit MSP are covered in [8, 9]. The optimized rotations are described in Table 6. For 8-bit left rotation, `swpb` and `xor` instructions are utilized. The `swpb` instruction performs byte-wise swap operation on the 16-bit variable, which exchanges the 8-bit values between lower and higher parts. Afterward, three `xor` instructions extract the swapped results. For 8-bit right rotation, the opposite routine of left rotation is required. For 1-bit right rotation, the `bit` instruction is used to check the least significant bit of the lower register (`X2`) and set the carry flag. Afterward the carry flag is updated to the most significant bit of the higher register (`X1`).

Key Scheduling Unlike 8-bit AVR processor, 16-bit MSP processor only equips 12 general purpose registers. For this reason, the register utilization is more important than AVR processor. 8, 1, and 3 general purpose registers are assigned to master key, delta variable address pointer, and temporal storages, respectively. However, additional 9 general purpose registers for counter and delta variables

Table 6. 32-bit rotations on 16-bit MSP, where T is temporal register [9]

$\lll 1$	$\lll 8$	$\lll 16$ ($\ggg 16$)	$\ggg 8$	$\ggg 1$
`RLA X2`	`SWPB X1`	`MOV X1, T`	`MOV.B X1, T`	`BIT #1, X2`
`RLC X1`	`SWPB X2`	`MOV X2, X1`	`XOR.B X2, T`	`RRC X1`
`ADC X2`	`MOV.B X1, T`	`MOV T, X1`	`XOR T, X1`	`RRC X2`
	`XOR.B X2, T`		`XOR T, X2`	
	`XOR T, X1`		`SWPB X1`	
	`XOR T, X2`		`SWPB X2`	
3 cycles	6 cycles	3 cycles	6 cycles	4 cycles

are required. In order to resolve the limited number of registers, stack pointer (`R1`) is utilized. Particularly, 128-bit delta variables, counter, and address pointer for round key are stacked and restored whenever the values are required in the code.

Encryption & Decryption The register utilization of encryption process is also important on 16-bit MSP processor. 8, 1, and 3 general purpose registers are assigned to plaintext, round key address pointer, and temporal storages. The counter variable is not stored in the register so the variable is stored in the stack. In LEA encryption, 6 32-bit round keys are required in each round. Among them, three round keys share same round key. This shows that only 4 round keys are required to perform the LEA encryption. However, MSP processor only provides very limited number of general purpose registers and four round keys cannot be reserved in the 3 16-bit registers. For this reason, part of 16-bit shared round key is reserved in the one register and the two registers are used to load the remaining round keys. In the straight-forward implementation, 8 16-bit indirect and 4 16-bit indexed memory access (28 clock cycles) are required[1]. By using cached round key, 8 16-bit indirect and 2 16-bit indexed memory access (22 clock cycles) are required. The rotation operations are performed by using Table 6. Four rounds are implemented and iterated by 6 times to complete the 24 rounds.

Decryption step is reversed order of encryption operation and the ARX operations of decryption step can be performed by following encryption step. The main difference is memory access pattern. The encryption routine accesses the round keys from the first to the last round keys. However, the decryption routine accesses the round keys from the last to the first round keys. Furthermore, MSP processor only supports incremental memory access and does not support decremental memory access. For the optimized memory access, the memory address offset is manually calculated. First, the offset is calculated. Second, the memory address is accessed by using indirect memory address mode in incremental order. In the straight-forward implementation, 12 16-bit indexed memory ac-

[1] indirect memory access requires 2 clock cycles and indexed memory access requires 3 clock cycles.

cess (36 clock cycles) are required. By using manual offset correction, 10 16-bit indirect and 4 offset correction (24 clock cycles) are required.

3.3 Optimization of HIGHT for 16-bit MSP Processors

The basic word size of HIGHT block cipher is 8-bit wise. For this reason, straightforward implementation of HIGHT on 16-bit MSP is inefficient. In this section, we explore the efficient 8-bit wise operations for 16-bit instructions of MSP processors

Key Scheduling In the key scheduling process, 12 general purpose registers are utilized. In particular, 8, 1, 1, 1, and 1 general purpose registers are assigned to master key variables, round key pointer, delta pointer, master key pointer, and loop counter, respectively. The master key pointer and loop counter are also used for temporal storages by pushing the data into the stack. The most expensive operation of key scheduling process is the update of delta variable. The detailed delta update is as follows.

$$\delta_{i+6} \leftarrow \delta_{i+2} \oplus \delta_{i-1}$$

The delta update requires bit-wise computations, which is inefficient for byte-wise platforms. For this reason, we used LUT-based approach to accelerate the performance. Another consideration is efficient 8-bit word handling for 16-bit MSP processors. The input/output length of LUT is 8-bit and memory access is expensive. In order to reduce the number of LUT accesses, 2 8-bit LUT results are loaded at once and used with post-processing. The detailed descriptions of round key generation are given in Algorithm 1. The round key generation requires addition of delta variable and master key. In Step 1, lower part of master key (`M0`) is moved to `TMP1`. In Step 2, 16-bit delta variable pair is loaded to `TMP2`. In Step 3 and 4, master key and delta variable is added and stored to the round key (`RP`). Similarly, from Step 5, higher part of master key (`M0`) and delta variable (`TMP2`) are prepared by using `SWPB` instruction. Finally, higher part is also stored to the round key (`RP`).

In each round key generation, the offset of master key is rotated by one. In the proposed implementation, two 8-bit master keys are stored in the 16-bit register and 8-bit wise rotation is inefficient for this alignment. For this reason, two rounds are directly performed and then the offset is updated by 2-word as follows.

`MOV M3, TMP1` → `MOV M2, M3` → `MOV M1, M2` → `MOV M0, M1` → `MOV TMP1, M0`

`MOV M7, TMP1` → `MOV M6, M7` → `MOV M5, M6` → `MOV M4, M5` → `MOV TMP1, M4`

Algorithm 1: Round key generation with 2 8-bit LUT access on MSP

Input: first and second master key pair (`M0`), delta variable pointer (`DP`), temporal registers (`TMP1, TMP2`)
Output: round key pointer (`RP`)

```
1: MOV.B M0, TMP1
2: MOV @DP+, TMP2
3: ADD.B TMP2, TMP1
4: MOV.B TMP1, 0(RP)

5: MOV M0, TMP1
6: SWPB TMP1
7: SWPB TMP2
8: ADD.B TMP2, TMP1
9: MOV.B TMP1, 1(RP)
```

Encryption & Decryption In the encryption operation, 12 general purpose registers are utilized. In particular, 8, 1, 1, 1, and 1 general purpose registers are assigned to plaintext, round key pointer, plaintext pointer, F0 function pointer, and F1 function pointer. The plaintext point is used for both temporal storage and loop counter by pushing the data into the stack. The expensive operations are F0 and F1 functions as follows.

$$\texttt{F0=X}^{\lll 1} \oplus \texttt{X}^{\lll 2} \oplus \texttt{X}^{\lll 7}$$

$$\texttt{F1=X}^{\lll 3} \oplus \texttt{X}^{\lll 4} \oplus \texttt{X}^{\lll 6}$$

The functions consists of XOR and rotation operations, which can be written in 2 256-byte pre-computed tables. In each round, four times of LUT accesses are required. The detailed descriptions of one round computation is given in Algorithm 2. In order to reduce the number of memory accesses, all plaintext variables are loaded to the 16-bit registers in 8-bit wise. The round key access is performed in byte wise since the number of temporal register is not enough to retain the round keys. The encryption round is performed with the F0/F1 LUT accesses and byte-wise XOR and addition operations. 8 rounds of encryption codes are written and iterated by 4 times to complete the 32 rounds. Similarly, the decryption round is performed in reversed order of encryption operation.

4 Evaluation

We evaluated LEA and HIGHT implementations on representative processors that are commonly used in IoT devices, namely 8-bit AVR and 16-bit MSP processors. The performance on the low-end devices (AVR and MSP) was evaluated in terms of code size (byte), RAM (byte), and execution time (clock cycle). In Table 7, the performance evaluation of LEA and HIGHT block ciphers is presented.

Algorithm 2: 1 round of HIGHT encryption using F0 and F1 LUTs

Input: plaintext registers (A0, A1, ..., A7), F0 pointer (F0P), F1 pointer (F1P), temporal register (TMP), round key pointer (RP)
Output: plaintext registers (A0, A1, ..., A7)

```
 1: MOV F1P, TMP
 2: ADD A0, TMP
 3: MOV.B @TMP, TMP
 4: XOR.B 0(RP),TMP
 5: ADD.B TMP, A1

 6: MOV F0P, TMP
 7: ADD A2, TMP
 8: MOV.B @TMP, TMP
 9: ADD.B 1(RP),TMP
10: XOR.B TMP, A3

11: MOV F1P, TMP
12: ADD A4, TMP
13: MOV.B @TMP, TMP
14: XOR.B 2(RP),TMP
15: ADD.B TMP, A5

16: MOV F0P, TMP
17: ADD A6, TMP
18: MOV.B @TMP, TMP
19: ADD.B 3(RP),TMP
20: XOR.B TMP, A7
```

This evaluation was based on the scenario of FELICS framework. The FELICS framework considers three scenarios, namely cipher operation, communication protocol, and challenge-handshake authentication protocol. Among them, we selected the cipher operation (i.e. scenario 0) to measure the performance of encryption and decryption operations. Scenario 0 evaluates the performance of the round key generation, encryption, and decryption for a single block. The results include the implementation of speed optimized assembly. The detailed descriptions are given in Table 7.

In 8-bit AVR processor, the proposed implementation achieved the highest performance among previous LEA implementations. Particularly, key scheduling, encryption, and decryption requires 203, 167, and 170 clock cycles/byte and enhances the performance by 13.6%, 0.6%, and 3.4%, respectively. Interestingly, the proposed implementation requires smaller RAM size than previous works since it only utilized the 16 callee-saved registers. For the code size, encryption and decryption operations achieved the 8 and 14 bytes smaller than previous works. For the key scheduling routine, previous works by [9] implemented only single round of key scheduling. However, the proposed work utilized the cached delta value and four rounds are implemented. For this reason, the size of key scheduling is larger than previous works but this achieved the highest performance and for the 128KB processor, 0.5KB is not high overheads. Similarly, in 16-bit MSP430 processor, the proposed LEA implementation achieved 175, 118, and 127 clock cycles for key scheduling, encryption, and decryption, respectively, which shows that the proposed implementations are faster than previous works by 9.3%, 8.5%, and 1.5%, respectively. Only code size of key scheduling achieved lower performance but other factors are better than previous works.

Table 7. Comparison results of LEA and HIGHT block ciphers on 8-bit AVR and 16-bit MSP in terms of code size (byte), RAM (byte), and execution time (clock cycle / byte)

Impl.	Code size (bytes)				RAM (bytes)			Execution time (cycles per byte)		
	EKS	ENC	DEC	SUM	EKS	ENC	DEC	EKS	ENC	DEC
LEA–AVR										
[4]	-	-	-	-	-	-	-	-	190	-
[10]	-	924	-	-	-	592	-	-	169	-
[9]	520	862	890	2,272	467	433	433	235	168	176
This Work	1,068	854	876	2,798	444	416	416	203	167	170
LEA–MSP										
[9]	314	650	654	1,618	456	440	440	193	129	129
This Work	830	596	650	2,076	450	416	416	175	118	127
HIGHT–MSP										
[9]	402	1,134	1,138	2,162	290	676	676	119	222	223
This Work	448	520	526	1,494	296	674	674	73	147	148

For the case of HIGHT block cipher, the proposed approach achieved the highest performance among them. The proposed implementation fully utilized the general purpose registers to cache the master key and plaintext/ciphertext for key scheduling and encryption/decryption operations. Particularly, 2 8-bit words delta variables are loaded and finely re-ordered to reduce the number of memory access for key scheduling. The proposed HIGHT implementation achieved 73, 147, and 148 clock cycles for key scheduling, encryption, and decryption, respectively, which shows that the proposed implementations are faster than previous works by 38.6%, 33.7%, and 33.6%, respectively.

5 Conclusion

One of the biggest challenges for Internet of Thing (IoT) is establishing the secure communications between resource constrained embedded processors. In order to ensure secure and robust transactions, we should conduct the encryption operation on sensitive and important information. In this paper, we explore the optimal implementations pursuing high speed and small memory footprint for the LEA and HIGHT block ciphers. This paper particularly concerned on the

number of memory accesses by using cached approach. This enhances the performance over the 8-bit AVR processor by 13.6%, 0.6%, and 3.4% for key scheduling, encryption, and decryption, respectively. The performance gain is also observed on the 16-bit MSP processor by 9.3%, 8.5%, and 1.5% for key scheduling, encryption, and decryption, respectively. Similarly, the proposed HIGHT implementations on the 16-bit MSP processors are faster than previous works by 38.6%, 33.7%, and 33.6% for key scheduling, encryption, and decryption, respectively.

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. 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.
2. C. P. Gouvêa and J. López. High speed implementation of authenticated encryption for the MSP430X microcontroller. In *Progress in Cryptology–LATINCRYPT 2012*, pages 288–304. Springer, 2012.
3. C. P. Gouvêa, L. B. Oliveira, and J. López. Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *Journal of Cryptographic Engineering*, 2(1):19–29, 2012.
4. D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee. LEA: A 128-bit block cipher for fast encryption on common processors. In *Information Security Applications–WISA 2013*, pages 3–27. Springer, 2013.
5. 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.
6. 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.
7. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.
8. T. Park, H. Seo, Z. Liu, J. Choi, and H. Kim. Compact implementations of LSH. In *International Workshop on Information Security Applications*, pages 41–53. Springer, 2015.
9. 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)*, 17(3):60, 2018.
10. H. Seo, Z. Liu, J. Choi, T. Park, and H. Kim. Compact implementations of LEA block cipher for low-end microprocessors. In *Information Security Applications–WISA 2015*. Springer, 2015.

11. H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi, and H. Kim. Parallel implementations of LEA. In *Information Security and Cryptology–ICISC 2013*, pages 256–274. Springer, 2013.
12. H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y. Nogami, Y. Zhu, and H. Kim. Parallel implementations of LEA, revisited. In *International Workshop on Information Security Applications*, pages 318–330. Springer, 2016.