Compact Implementations of LEA Block Cipher for Low-End Microprocessors

Hwajeong Seo[1], Zhe Liu[2], Jongseok Choi[1], Taehwan Park[1], and Howon Kim[1⋆]

[1] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,jschoi85,pth5804,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu}@uni.lu

Abstract. In WISA'13, a novel lightweight block cipher named LEA was released. This algorithm has certain useful features for hardware and software implementations, i.e., simple ARX operations, non-S-box architecture, and 32-bit word size. These features are realized in several platforms for practical usage with high performance and low overheads. In this paper, we further improve 128-, 192- and 256-bit LEA encryption for low-end embedded processors. Firstly we present speed optimization methods. The methods split a 32-bit word operation into four byte-wise operations and avoid several rotation operations by taking advantages of efficient byte-wise rotations. Secondly we reduce the code size to ensure minimum code size. We find the minimum inner loops and optimize them in an instruction set level. After then we construct the whole algorithm in a partly unrolled fashion with reasonable speed. Finally, we achieved the fastest LEA implementations, which improves performance by 10.9% than previous best known results. For size optimization, our implementation only occupies the 280B to conduct LEA encryption. After scaling, our implementation achieved the smallest ARX implementations so far, compared with other state-of-art ARX block ciphers such as SPECK and SIMON.

Keywords: Low-power Encryption Algorithm, AVR, Speed Optimization, Speed Optimization

1 Introduction

In 2013, Low-power Encryption Algorithm (LEA) was announced by the Attached Institute of ETRI [7]. This algorithm has software-friendly architecture and efficient implementation results on wide range of computational devices from high-end machines such as personal computers and smart phones, to low-end

⋆ Corresponding Author

Table 1. Instruction set summary for AVR [2]

Mnemonics	Operands	Description	Operation	#Clock
ADD	Rd, Rr	Add without Carry	Rd ← Rd+Rr	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd+Rr+C	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd⊕Rr	1
LSL	Rd	Logical Shift Left	C|Rd ← Rd<<1	1
LSR	Rd	Logical Shift Right	Rd|C ← 1>>Rd	1
ROL	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1
ROR	Rd	Rotate Right Through Carry	Rd|C ← C||1>>Rd	1
LD	Rd, X	Load Indirect	Rd ← (X)	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	2

microprocessors such as AVR and ARM processors are also drawn in previous papers [7, 8]. In this paper, we re-visit previous results on low-end devices and further improve performance in various platforms. This result can contribute to compact design of LEA in terms of high speed and low capacity and ensure the secure and robust communications between low-end devices.

The remainder of this paper is organized as follows. In Section 2, we recap the basic specifications of LEA and target platform. In Section 3, we present the compact implementations of LEA 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 LEA Block Cipher

In 2013, Low-power Encryption Algorithm (LEA) was announced by the Attached Institute of ETRI [4]. This algorithm has simple Addition-Rotation-eXclusive-or (ARX) and non-S-box architecture for high performance on both software and hardware environments. LEA is a block cipher with 128-bit block and word size is 32-bit. Various security levels including 128-bit, 192-bit and 256-bit are available. The number of rounds is 24, 28 and 32 for 128-, 192- and 256-bit keys, respectively. The algorithm consists of key schedule, encryption and decryption operations.

2.2 8-bit Embedded Platform AVR

The 8-bit AVR embedded processor is equipped with an ATmega128 8-bit processor clocked at 7.3728 MHz [2]. 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 (R26∼R31) 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

Table 2. 32-bit instructions over 8-bit AVR, where `R12-R15`, `R16-R19` and `R20` represent destination, source and temporal registers, respectively

Addition	Exclusive-or	Right Rotation
`ADD R12, R16`	`EOR R12, R16`	`CLR R20 ROR R12`
`ADC R13, R17`	`EOR R13, R17`	`LSR R15 ROR R20`
`ADC R14, R18`	`EOR R14, R18`	`ROR R14 EOR R15, R20`
`ADC R15, R19`	`EOR R15, R19`	`ROR R13`

multiplication incur two processing cycles. In Table 1, the detailed instructions used in this paper are drawn.

Previous 8-bit microprocessor results show 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 [7, 6]. Former implementation used separated mode to optimize performance in terms of speed by considering high performance. In case of AES, they used the conventional approach [6] to reduce memory consumption and high speed. The lookup tables are the forward and inverse S-boxes, each 256 bytes, because 32-bit look-up table access is not favorable due to limited storages over low-end devices. S-box pointer is always placed in `Z` register and the variable is stored into `SRAM` for fast access speed. For efficient MixColumn 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, xors operations, taking a total of 26 cycles. The InvMixColumns step is implemented similarly, but is more complicated and takes a total of 42 cycles. Recently, ARX-based block ciphers (SPECK and SIMON) are introduced [4, 3]. They provides efficient rotation operation by multiplying the byte and general multi-precision addition, rotation and exclusive-or operations are studied.

3 Proposed Method

Unlike modern processors, embedded processor provides limited computing power and storage capacities. We need to carefully re-design the algorithm to meet the requirements of speed and size factors over resource constrained environments. In this section, we introduce LEA implementation techniques for low-end microprocessors.

3.1 On the Fly versus Separate Computation Modes

LEA block cipher consists of key schedule and encryption/decryption. The key schedule generates each round pair to be used for encryption. If target platform has enough storages, whole round key pairs can be pre-computed in offline and stored into storages. By selecting the methods of round key generation, we can achieve the two opposite properties including size and speed. Firstly, on the fly method generates round key on the spot and then directly encrypts plaintext

Algorithm 1 Efficient Shift Offset and Direction in AVR

Require: direction d, offset o
Ensure: direction d, offset o
1: $o = o \bmod 8$
2: **if** o>4 **then** $o = 8 - o$, $d = !d$
3: **return** d, o

with these round key pairs. The main benefits are two folded. Additional storages for round keys are not needed and source code size is reduced by rolling the encryption and key scheduling. Secondly, separated computation mode literally executes key schedule and encryption processes separately. The round keys are computed in offline and then stored into temporal storages. After then these values are simply loaded and used during encryption or decryption process. The method can avoid the key generation process.

3.2 Speed Optimization

Core operations of LEA are 32-bit wise addition, bit-wise exclusive-or and rotation. When it comes to a 8-bit processor, 32-bit wise instruction is not straightforwardly computable. For this reason, we sliced a 32-bit instruction into four 8-bit instructions. The detailed process is described in Table 2. In case of addition and bit-wise exclusive-or, four 8-bit instructions are conducted for 32-bit single instruction. From 9th to 32th bit, carry bits are concerned during addition operation. In case of rotation, we shift four 8-bit registers and then conduct carry handling to rotate the carry bits. Among operations, rotation operation is particularly crucial operation in microprocessors, because there is no 32-bit wise rotation supported and carry handling is complicated process than any other operations. In order to overcome this problem, we set several efficient computation strategy. Firstly, shift operation over 8-bit offset is omitted because 8-bit shift is simply established by ordering of inner word. Secondly, as we can find in [1], shift operation by $(5, 6, 7)$-bit is simply replaced by $(3, 2, 1)$-bit shifts in opposite direction. The detailed 8-bit shift process is available in Algorithm 1. Firstly offsets in multiple times of 8-bit are reduced and then remaining bits over 4-bit is changed into opposite direction. For left rotation by 9-bit, we simply conduct one bit left shift with register arrangements so this approach saves 8-bit left rotation instructions.

For further improvements of performance, we retained variables in registers rather than memory. For encryption, we allocate sixteen registers (`R0` $\sim$ `R15`) for plaintext (`X0, X1, X2, X3`) and eight registers (`R16` $\sim$ `R23`) are reserved for purpose of temporal storages. Combining proposed rotation techniques and register allocation, we can schedule registers described in Figure 1. The figure describes from round 1 to round 4 of encryption, where each box represents 8-bit and remaining steps are iterated in same order. As we explained before, shift with over 8-bit is computed without cost by ordering the results. In case of 128-bit key scheduling, sixteen registers are assigned for master key and remaining

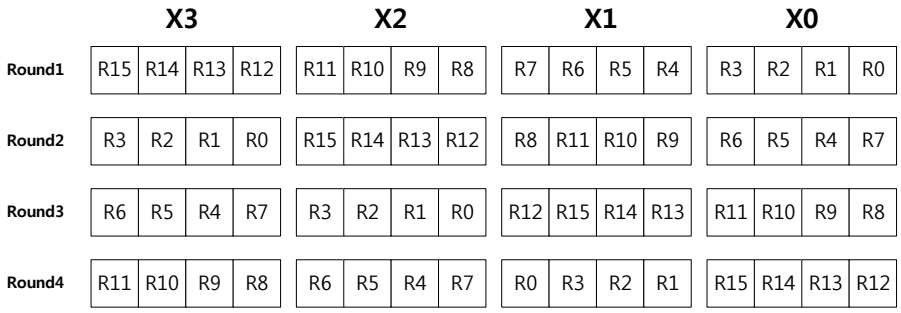

Fig. 1. Register alignments for LEA encryption in AVR

registers are used for temporal registers. The delta variable is not retained into registers due to limited number of registers so it is obtained from memory in every round.

3.3 Size Optimization

In previous section, we show the highest speed record with loop unrolled optimizations. This ensures the best possible performance but it increases code size significantly. In order to minimize the code size, an implementation with "rolled" loops is the most feasible choice. The size-first method rolls whole source codes by number of iteration (N). If size of source code is (S), the size of looped version is calculated in ($\frac{S}{N} + A$), where (A) represents overheads of counter, offset and branch operations. However, the performance is relatively slower than that of unrolled version. One possible solution is to partially unroll the loops. For instance, the body of the loop can be replicated multiple times, which replaces a number of loop iterations by non-iterated straight-line code.

Minimum Loop Implementation 128-bit LEA block cipher consists of 24 rounds. As described in Figure 2, each round again boiled down to three addition, six exclusive-or and three rotation operations. These operations are grouped into three inner loops and iterated by three times in a round. In order to minimize the source code, we only implemented single inner loop operation and then iterated the inner loop by three times[3]. This process computes one round function. After then, 24 times of round operations are iterated. Of the 26 registers, the 16 registers are assigned for plaintext and four registers for rotation counter, one for round counter and five for temporal registers. Since the number of general purpose registers is highly limited, efficient scheduling of register is important.

[3] The 32-bit wise inner loops are optimal choice because each instruction set occupies 2 bytes and 32-bit instruction only needs four consecutive instructions(8 bytes = 4×2) If we use 8-bit instruction as a minimum loop for 32-bit addition, we should use 1 `ADD`, 1 `MOV`, 1 `INC`, 1 `CPSE` and 1 `RJMP` and total 10 bytes with far slow performance.

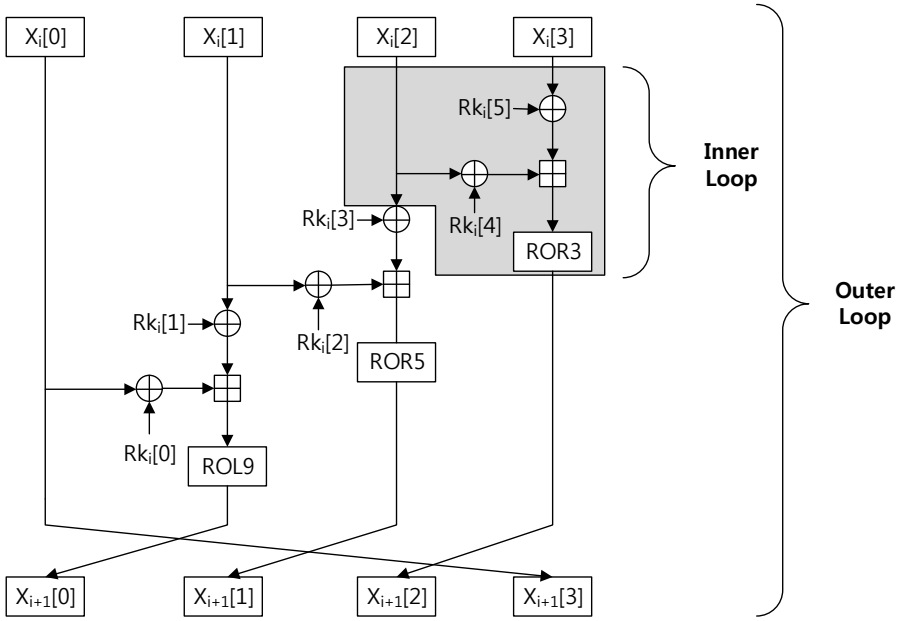

Fig. 2. Inner and outer loops of LEA encryption

We firstly computed value in `Xi[3]` and then stored the results, because `Xi[3]` is once used but not used in following operations. After each round, the variable `X` is shifted by one word size (32-bit) to align the variables for looped operation. Cases of 192-, 256-bit implementations are also achieved with same program but different number of rounds (28 and 32 times), because their basic architectures are identical to that of 128-bit encryption.

Since LEA decryption has a similar structure of encryption. The techniques for encryption can be applied to the decryption with simple modification. Each round in Figure 3 consists of three subtraction, six exclusive-or and three rotation operations. This inner loop is iterated by three times in a round and then each round operation is repeated by 24 times for 128-bit decryption. The decryption computes opposite way in that of encryption, so we firstly use `Xi[0]` and then store the results into `Xi[0]` registers, because following operations do not need `Xi[0]` variables any more. After each round operation, variable alignments follow by word size (32-bit). The 16 registers are assigned for ciphertext and four registers for rotation offsets, one for round counter and five for temporal registers.

Key scheduling consists of eight rotation and four addition operations. The loop is grouped into four identical inner loops. The loop contains two rotation and four addition operations. Firstly whole keys are loaded into registers and then key scheduling is conducted by size of inner loop. Each round has different rotation count and delta variables. Due to limited number of registers, we re-

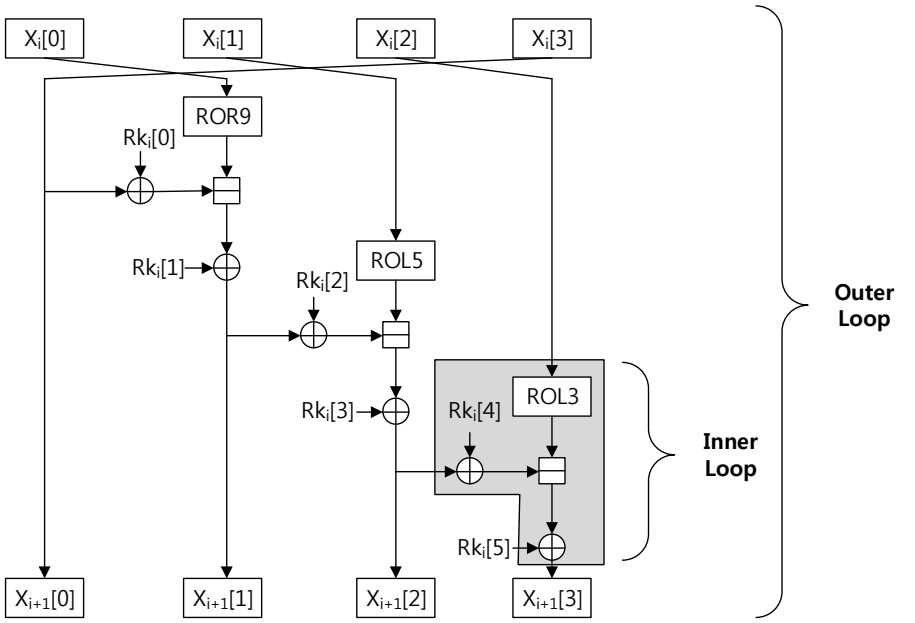

Fig. 3. Inner and outer loops of LEA decryption

load delta variables every time when it is in needs. The counter values are also re-loaded from memory. Since rotation and delta variables are changed in every round, we should schedule these offsets with counter variables. For efficient 128-bit loop encryption implementations, we stored duplicated six 32-bit round key pairs. The 16 registers are assigned for secret/round key and four registers for rotation counter, two for round counter and six for temporal registers. In order to reduce the source code size, program is written in looped fashion. For looped version, we re-aligned plaintext, ciphertext and round keys in every inner round. Looped version always accesses to same index of registers but we should ensure that the registers contain different variables by the round. We rotated destination registers by word size in an every inner round to align the variables properly.

Separated Method Block cipher can be computed in separated key scheduling and encryption/decryption operations. This method firstly computes whole key chains once and stores them into storages. And then encryption/decryption operations follow. Since key scheduling method is executed once before encryption process, separated mode can avoid 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 load plaintext and round keys by 128-bit and 192-bit to conduct encryption process. In case of decryption, we load ciphertext and access round keys in reverse order to conduct decryption process.

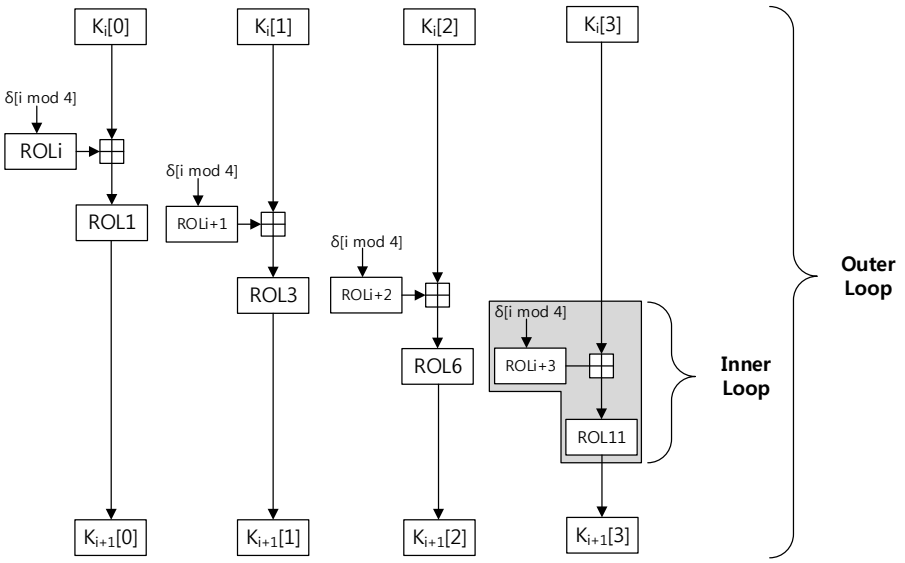

Fig. 4. Inner and outer loops of LEA key scheduling

On-the-fly Method On-the-fly method is the challenging task over resource constrained embedded processors because this operation mode is required to retain more number of parameters including delta, round key, plaintext and counter variables for rotation and round. Since the storages for these parameters are beyond capacity of registers in embedded processors, additional memory is in needs. Memory access is one of the most expensive operations but we should retain intermediate variables to memory due to limited number of registers. For on-the-fly computations, we firstly load secret key, delta and counter variables and then generate round keys in every session. The round key pairs are not placed within registers because sub-sequent operations also need many registers for plaintext and counter variables. In each inner round, we re-load round keys to generate the next round key pairs and intermediate ciphertext pairs are also re-loaded and placed back to memory after computations. This process is iterated by the number of rounds.

Decryption is conducted in reverse way of encryption process. In the case of encryption, round key pairs are generated from initial secret keys and each key is used directly for the encryption computations. On the other hand, decryption conducts the operation from last round key pairs to first. For starter, we conduct key scheduling in ordinary way to get final round key pairs after then we traced back from last to first round keys. While conducting key scheduling in reverse order, decryption process is conducted by each round. The key scheduling in reverse order follows reverse order of ordinary key scheduling. Firstly delta variables are rotated by number of `i`. And then round keys are rotated. For better performance, we conduct right rotation. Right rotation can compute the inverse

Table 3. Scaled overheads of block ciphers in 128-bit encryption cases

Features	LEA	SPECK	SIMON
Scaled in 32-bit addition	3	2	-
Scaled in 32-bit bit-wise and	-	-	2
Scaled in 32-bit exclusive-or	6	4	10
Scaled in 32-bit rotation (offset)	7	6	6
Size of master key (bit)	128	128	128
Total size of round key (bit)	4608	2048	2176
Number of Rounds	24	32	34
Scaled panelty point in speed (clock)	-	+40	-19
Scaled panelty point in size (proportion)	-	1.57	1.1

operation of left rotation. After then both values are subtracted. The loops are iterated by the number of rounds.

Scalable Implementation In order to support various protocols and environments, we need to ensure diverse security levels. If we implement one by one over the embedded processors, it would consume more capacities to store. For this reason, we implemented scalable encryption and decryption operations. Thanks to simple architecture of LEA block cipher, we can easily implement scalable encryption/decryption operations. LEA block cipher provides three different key sizes including 128-, 192-, and 256-bit. The basic instruction is same for all key sizes but only the number of round is varied. By assigning the number of rounds differently, we can readily scale the three different encryption models. In case of decryption, opposite order of encryption operations are conducted. As like encryption case, we can only alter the round numbers to conduct the operations.

Loop Friendly Instruction Set For register realignments, `MOVW` instruction relocates two adjacent register to destination within single clock cycle. In case of memory load/store, we can use 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. The `ADIW` and `SBIW` conduct addition and subtraction by word with immediate value. These are used to modify memory address. Finally, `INC` and `DEC` operations are used to increase and decrease the counter variables.

3.4 Implementations for 16-bit MSP Embedded Processors

The proposed techniques are not limited to 8-bit AVR processors. We also applied to other resource constrained devices such as 16-bit MSP processors. As like 8-bit AVR, 32-bit wise ARX instructions are split into two consecutive 16-bit operations. In case of rotation operation, we also adopted efficient rotation method in Algorithm 1. For left rotation by 9-bit, we can replace the operations into right shift by 7-bit (`16-9`) together with register ordering. Of twelve 16-bit

registers, two or three registers are assigned for pointer and remaining ten or nine registers are available for general purposes. However, plaintext and round keys are larger than register capacities. For this reason, register utilizations should be taken into accounts.

For LEA encryption process, we allocate eight registers (`R4` $\sim$ `R11`) for plaintext (`X0, X1, X2, X3`) and remaining registers are used for temporal storages. The order of plaintext is also aligned by using techniques introduced in Figure 1. In case of 128-bit key scheduling, eight registers are assigned for master key and remaining registers are used for temporal registers. The delta variable is not kept in the registers due to lack of register. The variables are re-loaded from memory in every session. Finally we compute the LEA key scheduling and encryption in 206.4 and 157.6 cycles/byte. This is the first LEA implementation on MSP processors. We compared results with AES implementations. The results show that 180 cycles/byte for AES encryption [5].

4 Results

4.1 Speed Optimization

In Table 4, comparison results of speed factor on AVR are described[4]. The results introduced in WISA'13 [7] computes LEA encryption within 190 cycles/byte. Our optimized implementation achieved 169.2 cycles/byte, which improves performance by 10.9%. We also compared with other block ciphers. Of many methods, we selected the most well known block cipher AES. AES follows SPN architecture but LEA is ARX architecture. For fair comparison, we brought the most well-known ARX based block ciphers such as SPECK and SIMON.

Firstly, optimized LEA shows lower performance than optimized AES by 26.4%. This is obvious that LEA is targeting the 32-bit processor but AES is for 8-bit processors. Secondly, we compared with SPECK and SIMON. Direct comparison with both algorithm is also unfair because number of arithmetic is different to each other. In case of 128-bit SPECK encryption, each round consists of 64-bit rotation by 3-bit and one 64-bit addition and two 64-bit exclusive-or operations. When they are scaled into 32-bit operations each 64-bit operation is split into 2 32-bit operation. Another factor is round key size. If round key is getting large, memory access frequently happens and latency should be lower. The SPECK has relatively small round key sizes (2048 bits). In case of 128-bit SIMON encryption, each round consists of 64-bit rotation by 3-bit and one 64-bit logical `AND` and five 64-bit exclusive-or operations. When they are scaled into 32-bit operation, the operations are changed into 10-bit rotation, two `AND` and six exclusive-or. The SIMON has also small round key sizes by 2176 bits. We scaled the operation by calculating the overheads of each operation. One 32-bit operation needs four 8-bit operations (4 clock cycles) and 1 byte memory accesses

[4] The performance is measured in clock cycles and bytes for timing and code size. Precise results are measured in AVR studio and program is complied with optimization level 2.

Table 4. Speed optimized results on AVR, encryption is measured in cycles/byte and code size in bytes, *: estimated results, P/C: Pre-computed

Method	ARCH	KEY	ENC	ENC(scaled)	ROM(byte)	RAM(byte)
Speed(Separated)						
LEA 128-bit	ARX	P/C	169.2	**169.2**	924	592
LEA 192-bit	ARX	P/C	224.6	N/A	1004	688
LEA 256-bit	ARX	P/C	256.1	N/A	1004	784
LEA 128-bit [7]	ARX	P/C	190	**190**	N/A	N/A
SPECK 128-bit [3]	ARX	P/C	143	**183**	452	256
SPECK 192-bit [3]	ARX	P/C	147	N/A	632	272
SPECK 256-bit [3]	ARX	P/C	151	N/A	522	288
SIMON 128-bit [3]	ARX	P/C	337	**318**	510	544
SIMON 192-bit [3]	ARX	P/C	339	N/A	646	552
SIMON 256-bit [3]	ARX	P/C	357	N/A	522	576
AES 128-bit [6]	SPN	P/C	124.5	N/A	956*	N/A

needs 2 clock cycles. With this conversion, we can draw objective complexity (cycles/byte) of LEA ($168 = \frac{16\times\#round(24)\times4+2\times\frac{roundkey(4608)}{8}}{16}$), SPECK ($128 = \frac{12\times32\times4+2\times\frac{2048}{8}}{16}$) and SIMON ($187 = \frac{18\times34\times4+2\times\frac{2176}{8}}{16}$). The detailed comparison in Table 3. After scaling, our work is faster than SPECK and SIMON by 7.5 % and 46.8 %, respectively.

4.2 Size Optimization

One 32-bit operation consists of four 8-bit operations. Each 8-bit operation needs 2 bytes for program instructions. In case of memory accesses, one byte access operation needs 2 bytes. We calculated the relative costs in each round of block cipher as follows. In case of LEA, its complexity is ($176 = \#operation(16) \times 4 \times 2 + roundkey(192)/8 \times 2$) and for SPECK it is ($112 = \#operation(12) \times 4 \times 2 + roundkey(64)/8 \times 2$) and for SIMON it is ($160 = \#operation(18) \times 4 \times 2 + roundkey(64)/8 \times 2$). We compute relative complexity and after scaling LEA implementation is 35% smaller than SPECK and SIMON. Furthermore, we presented on-the-fly and parameterized version with only 1286 and 592 bytes, respectively. The parameterized version is readily available in LEA because it shares same structures throughout the all security levels.

5 Conclusion

One of the biggest challenges for Internet of Things is secure communications between small and 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 new light-weight

Table 5. Size optimized results on AVR, Key and Enc are measured in cycles/byte and code size in bytes. P/C: Pre-computed

Method	KEY	ENC	DEC	ROM(byte)	ROM(scaled)	RAM(byte)
Size(On-the-fly)						
LEA 128-bit	N/A	2576	5029	1286	**1286**	88
Size(Parameterized)						
LEA 128-bit	N/A	729.8	748.8	592	**592**	596
LEA 192-bit	N/A	849.8	871.8	592	N/A	692
LEA 256-bit	N/A	969.8	994.8	592	N/A	788
Size(Separated)						
LEA 128-bit	P/C	729.6	N/A	280	**280**	592
LEA 192-bit	P/C	849.6	N/A	280	N/A	688
LEA 256-bit	P/C	969.6	N/A	280	N/A	784
SPECK 128-bit [3]	P/C	169	N/A	278	**436**	264
SPECK 192-bit [3]	P/C	174	N/A	330	N/A	272
SPECK 256-bit [3]	P/C	179	N/A	348	N/A	280
SIMON 128-bit [3]	P/C	346	N/A	392	**431**	544
SIMON 192-bit [3]	P/C	351	N/A	392	N/A	552
SIMON 256-bit [3]	P/C	366	N/A	404	N/A	576

block cipher, LEA. This paper presents several optimization techniques including efficient 32-bit wise ARX operations and minimum inner loop scheduling. Furthermore, to the best of our knowledge, this is first scalable LEA block cipher implementations over AVR processor. The program can compute various key sizes with single code and no further modifications.

References

1. D. F. Aranha, R. Dahab, J. López, and L. B. Oliveira. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, 2010.
2. 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.
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.
4. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.
5. 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.
6. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software aes encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.
7. K. H. Ryu and D.-G. Lee. Lea: A 128-bit block cipher for fast encryption on common processors. *Information Security Applications*, page 3.

8. 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, 2014.