Optimized Implementation of SM4 on AVR Microcontrollers and ARM Processors

Hyeokdong Kwon[1], Hyunjun Kim[1], Eum Si Woo[1], Minjoo Shim[1], Wai-Kong Lee[2], Zhi Hu[3], and Hwajeong Seo[1][0000-0003-0069-9061]

[1]IT Department, Hansung University, Seoul (02876), South Korea,
{korlethean, khj930704, shuraatum, minjoos9797, hwajeong84}@gmail.com
[2]Department of Computer Engineering,
Gachon University, Seongnam, Incheon (13120), Korea,
waikonglee@gachon.ac.kr [3]Central South University, China,
huzhi_math@csu.edu.cn

Abstract. The SM4 block cipher is a Chinese domestic crpytographic that was introduced in 2003. Since the algorithm was developed for the use in wireless sensor networks, it is mandated in the Chinese National Standard for Wireless LAN WAPI (Wired Authentication and Privacy Infrastructure). The SM4 block cipher uses a 128-bit block size and a 32-bit round key. This consists of 32 rounds and one reverse translation R. In this paper, we present the optimized implementation of the SM4 block cipher on 8-bit AVR microcontrollers, which are widely used in wireless sensor devices, and the optimized implementation of SM4 on 64-bit ARM processors with the parallel computation, which are widely used in smartphone and tablet. In the AVR microcontroller, it is implemented in three versions, including speed-optimization, memory-optimization, and code-optimization. As a result, speed-optimization, memory-optimization, and code-optimization achieved 205.2 cycles per byte, 213.3 cycles per byte and 207.4 cycles per byte, respectively. This is faster than the reference implementation written in C (1670.7 cycles per byte). The implementation on 64-bit ARM processors is 8.62 cycles per byte. This is faster than the reference C code implementation (120.07 cycles per byte).

Keywords: 8-bit AVR Microcontrollers · 64-bit ARM Processors · Software Implementation · SM4 Block Cipher.

1 Introduction

A number of sensor nodes are used to collect the data in wireless sensor networks. Tiny sensor nodes have limited computation resources, such as computing power, memory size, and battery life. Since cryptographic algorithms are based on complicated operations, it is difficult to achieve the high availability for wireless sensor networks in secure packets. To resolve this problem, lightweight block cipher algorithms have been proposed. Lightweight cryptography algorithms require low resources than ordinary cryptographic algorithms. The SM4 block cipher is one of the lightweight block cipher, which is Chinese National Standard for wireless LAN WAPI (Wired Authentication and Privacy Infrastructure). [1]

In this paper, we propose optimized implementations of the SM4 block cipher on both low-end 8-bit AVR microcontrollers and high-end 64-bit ARM processors. Main contributions are as follow:

1.1 Contributions

- **Optimized implementations of the SM4 block cipher on 8-bit AVR microcontrollers.** We implemented the SM4 block cipher on low-end AVR microcontrollers. SM4 block cipher requires the 128-bit block size, while AVR microcontrollers only support 8-bit wise general purpose registers. Therefore, the efficient register allocation should be considered. We proposed the optimal register allocation. Furthermore, the SM4 block cipher requires the 32-bit wise rotation operation, while 8-bit wise operations are performed on AVR microcontrollers. We suggested the optimized implementation of 32-bit wise rotation on 8-bit development environments.
- **Parallel implementations of the SM4 block cipher on 64-bit ARM Processors.** 64-bit ARM processors support SIMD (Single Instruction Multiple Data) features, which can process the data in a parallel way. We propose the parallel implementation of the SM4 block cipher in a 12-way. This implementation encrypts 12 plaintext blocks at once through SIMD instructions. For the optimal implementation, we introduce the vector register allocation plan with arrangement and efficient instructions for the optimized parallel implementation.

2 Backgrounds

2.1 SM4 Block Cipher

The SM4 block cipher is one of a China domestic crpytographic algorithm, which was first published in 2003. It was a cryptographic standard issued by the OSCCA (Office of State Commercial Crpytography Administration) [2]. Table 1 shows the list of SM4 parameters. The left of figure 1 describe encryption tasks for the SM4 block cipher.

Table 1. Parameters for the SM4 block cipher.

Block size	Round key size	Rounds (Encryption)	Rounds (Key schedule)
128-bit	32-bit	32	32

The SM4 block cipher consists of 4 computations; Round function (`F`), Permutations (`T` and `T'`), Nonlinear transformation (`tau`), Linear transformations (`L`) and (`L'`), and S-box (`S`).

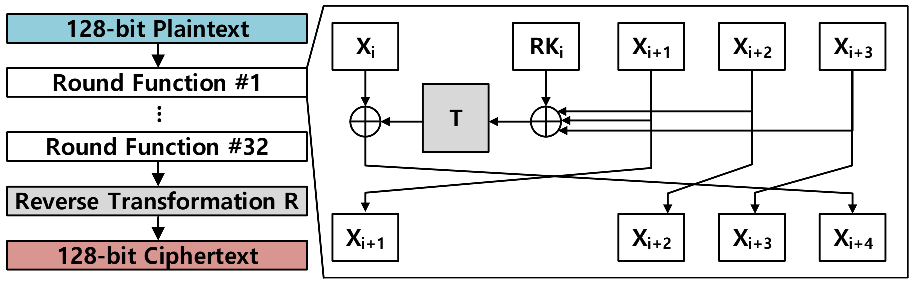

Fig. 1. Encryption flow of the SM4 block cipher and the round function structure.

Round function (F). The plaintext of the SM4 block cipher is divided in four 32-bit units, called X. Round function (F) requires 5 arguments, which are X_0, X_1, X_2, and X_3, and round key. F can be defined as the following equation.

$$\mathtt{F}(X_0, X_1, X_2, X_3, \mathrm{rk}) = X_0 \oplus \mathtt{T}(X_1 \oplus X_2 \oplus X_3 \oplus, \mathrm{rk})$$

The right of Figure 1 represents the Round function F structure.

Permutations T and T'. T is the permutation function that requires 32-bit input values, and makes 32-bit outputs. It has the reversible feature. Permutations T and T' consists of tau and L.

Nonlinear transformation tau. The nonlinear transformation (tau) uses 4 S-boxes, which needs 32-bit inputs and returns 32-bit outputs. It is performed in a parallel way. Each input value does not affect each other. Nonlinear transformation tau can be represented as follow, where A and B are 32-bit input value and 32-bit output value, respectively. The type of a_i and b_i is a 8-bit wise string.

$$A = (a_0, a_1, a_2, a_3); \quad \mathtt{tau}(\mathrm{A}) = (\mathrm{S}(a_0), \mathrm{S}(a_1), \mathrm{S}(a_2), \mathrm{S}(a_3));$$
$$(b_0, b_1, b_2, b_3) = \mathtt{tau}(\mathrm{A}); \quad B = (b_0, b_1, b_2, b_3);$$

Linear transformations L, and L'. Linear transformations (L, and L') mainly perform rotate operations. Input values from output of tau, and operates 32-bit wise. L, and L' are can be defined as follow, where B is 32-bit input value, and ROTL represents the rotation to the left operation.

$$\mathtt{L}(B) = B \oplus (\mathtt{ROTL}(B, 2)) \oplus (\mathtt{ROTL}(B, 10)) \oplus (\mathtt{ROTL}(B, 18)) \oplus (\mathtt{ROTL}(B, 24))$$
$$\mathtt{L'}(B) = B \oplus (\mathtt{ROTL}(B, 13)) \oplus (\mathtt{ROTL}(B, 23))$$

S-box S. The S-box (S) transforms the 8-bit input value to the 8-bit output value with the S-box table. Input values are from the nonlinear transformation (tau).

2.2 Target Processor: 8-bit Low-end AVR Microcontrollers.

AVR microcontrollers is the 8-bit based Harvard architecture, which is widely used for wireless sensor networks. It has 32 8-bit general purpose registers and 133 instructions. Most of instructions are taken less than 4 clock cycles [3]. We evaluated the performance on ATmega128. This is the one of 8-bit AVR microcontroller family. It has 128KB of programmable flash memory, 4KB internal SRAM, 4KB EEPROM, and 64KB optional external memory space [4]. AVR registers are denoted as `R0` to `R31`. Some registers have special features as follows:

– **ZERO register:** `R1` is the zero register that always represents 0 value. However, it can be used freely for general purposes. This `R1` register should be zeroed at the end of the operation.
– **Callee saved registers:** `R2-R17` and `R28-R29` are callee saved registers (i.e. non-volatile registers). These registers saved important values (i.e. long-lived values and data from callee). These must be preserved in the stack before it is used.
– **Pointer address registers:** `R26-R31` can be used as a pointer address by combining two registers. When these are used for the pointer address, these are written as `X` (`R26-R27`), `Y` (`R28-R29`), `Z` (`R30-R31`) notation. `R28-R29` are also callee saved registers.

2.3 Target Processor: 64-bit High-end ARM Processors.

ARMv8-A is the next generation ARM architecture of ARMv7, simply called ARMv8. It has two architectures, which are 32-bit AArch32 and 64-bit AArch64. In this paper, we targeted the AArch64 architecture, in short A64. A64 has 32 64-bit general purpose scalar registers that can handle 32-bit, and 64-bit data. In addition, there are 32 128-bit vector registers, it can be utilized for the parallel implementation with SIMD [5]. We used vector registers to implement the SM4 block cipher in a parallel way.

2.4 Related works.

In this section, we introduce optimized implementations of block ciphers on AVR microcontrollers and ARM processors. In [7], the revised version of CHAM was optimized on 8-bit AVR microcontrollers. In [7], they suggested optimized 8-bit wise rotation and 32-bit wise rotation. This implementation utilized the pre-calculation technique with the counter mode of operation. In [6], parallel implementations are presented. In [8], the optimized ARIA block cipher was presented. They optimized primitive operations, including rotation operation, a subsitute-layer, and a diffusion-layer on the low-end AVR microcontroller. In [9], they proposed the compact implementation of PRESENT block cipher, which is introduced in CHES'07 [10]. It optimally implemented the PRESENT through pre-computation technique. In [11], the compact implementation of AES (Advanced Encryption Standard) block cipher on Intel processors was presented (i.e. FACE). This implementation applied pre-computation technique that pre-calculate repetitive values, and reused them. In ICISC'19, they proposed optimized implementation of FACE on the AVR microcontroller was presented [12]. It extended the pre-computation to the round 3. The implementation is also secure against CPA (Correlation Power Analysis).

3 Optimized Implementation of the SM4 Block Cipher

In this Section, we introduce the optimized implementation of the SM4 block cipher on 8-bit AVR microcontrollers and 64-bit ARM processors. The optimal performance is achieved through efficient register allocation and instruction techniques.

3.1 8-bit Low-end AVR Microcontrollers

Instruction set. AVR microcontrollers have useful instruction sets. Generally instructions take 1 or 2 clock cycles. Instructions used to implement the optimized SM4 block cipher are summarized in Table 2 [6].

Table 2. Summarized instruction set of AVR microcontrollers for optimized SM4 block cipher. `Rd`: Destination register, `Rr`: Source register.

Instruction	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
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	1
LSL	Rd	Logical Shift Left	C | Rd ← Rd << 1	1
ROL	Rd	Rotate Left Through Carry	C | Rd ← Rd << 1 || C	1
MOV	Rd, Rr	Copy Register	Rd ← Rr	1
MOVW	Rd, Rr	Copy Register Word	Rd + 1:Rd ← Rr + 1:Rr	1
LD	Rd, X(or Y, Z)	Load Indirect	Rd ← X(or Y, Z)	2
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	3
ST	X(or Y, Z), Rr	Store Indirect	X(or Y, Z) ← Rr	2
PUSH	Rr	Push Register on Stack	STACK ← Rr	2
POP	Rd	Pop Register from Stack	Rd ← STACK	2

Register utilization. For the optimized implementation, we efficiently allocated registers. Detailed descriptions are as follows:

– X **blocks.** In Section 2.1, the SM4 block cipher stores 128-bit plaintext into 4 32-bit X. However, 8-bit AVR microcontrollers have 8-bit wise registers that can only represent the 8-bit data. Four registers are required to handle one 32-bit X. As a result, there are 4 X that quarters of plaintext. 16 registers are required to store the whole plaintext.

- **Round key, and `T` input/output.** Each `F` requires a 32-bit round key. 4 8-bit registers used to save the round key. The round key is used as the input value of `T` by performing the XOR operation with X blocks. Therefore, round key registers are also used to store parameters or results of `T`.
- **Nonlinear operation.** The nonlinear transformation (`tau`) performs the rotation operation. 8 registers are required for result and intermediate values of rotation. 4 out of 8 registers store the `tau` output result.
- **Address pointer.** In order to load the value into a register on AVR microcontrollers, it should be accessed through the address pointer. In this case, there are 3 kinds of values for the function call; Plaintext, Round key, and S-box values. We allocate `X` pointer for loading plaintext, and storing ciphertext, `Y` pointer for Round key, and `Z` pointer for S-box values. Especially, the `X` pointer address (`R26` and `R27`) do not need to during round functions. These registers are used to store temporary values. The `R30` register is always fixed to 0 value, because it stored the lower address of S-Box. This can be used to the temporary ZERO register.
- **Loop index.** Using the `CPI` instruction, it is possible to compare a register value with a constant value. To implement the loop statement, it requires only one register to store loop index. This register is shared with the temporary value register. It needs to preserve an index value on the stack. It can be implement through `PUSH` and `POP` instructions.

Figure 2 shows the whole register allocation. Each rectangle represents 8-bit register. Two-colored registers are used for multiple purposes.

Fig. 2. Register allocation for the SM4.

Optimized implementation of 32-bit wise rotation. The rotation of the SM4 block cipher is 32-bit wise operation, but AVR microcontrollers perform only 8-bit wise. 32-bit wise rotation can be implemented with following instructions; `LSL, ROL, ADC, MOV,` and `MOVW`. Each rotation can be implemented following Table 3. When input and output values of the 8 or 16 rotation operation are in the same register, it needs one temporary register. This takes more clock cycles. We separated input and output registers. This makes results of rotation in different registers. This implementation eliminates the temporary register, and takes less clock cycles to transfer values to the temporary register than the previous method.

Table 3. Optimized 32-bit wise rotation operation on 8-bit environments. i, j: number of registers.

32-bit ROL_1	32-bit ROL_8	32-bit ROL_{16}	32-bit ROL_{24}
$LSL\ R_i$ $ROL\ R_{i+1}$ $ROL\ R_{i+2}$ $ROL\ R_{i+3}$ $ADC\ R_i,\ ZERO$	$MOV\ R_i,\ R_{j+3}$ $MOV\ R_{i+1},\ R_j$ $MOV\ R_{i+2},\ R_{j+1}$ $MOV\ R_{i+3},\ R_{j+2}$	$MOVW\ R_i,\ R_{j+2}$ $MOVW\ R_{i+2},\ R_j$	$MOV\ R_i,\ R_{j+1}$ $MOV\ R_{i+1},\ R_{j+2}$ $MOV\ R_{i+2},\ R_{j+3}$ $MOV\ R_{i+3},\ R_j$
5 cycles	4 cycles	2 cycles	4 cycles

Efficient S-Box implementation. In this paper, there are three optimization perspectives on AVR microcontrollers (speed-optimization, memory-optimization, and code-optimization). In terms of speed-optimization, storing the S-box in RAM is effective. The LD instruction loads the S-Box value with 2 clock cycles. This can get the S-Box value, quickly. On the other hand, for the memory-optimization perspective, S-Box can be saved to flash memory. In Section 2.2, it was confirmed that the AVR microcontroller has larger flash memory than RAM. Therefore, the memory-optimized implementation can be useful in situations where the lack of RAM. The memory-optimization can be implemented with the LPM instruction, which takes 3 clock cycles. As a result, the memory-optimization takes a longer executing time than the speed-optimization. For the case of code-optimization, we utilized the looped implementation, which sacrificed the performance but achieved the optimal code size.

3.2 64-bit high-end ARM Processors

On the 64-bit ARMv8 processor, the efficient implementation is possible by using vector registers. When implemented in a parallel-way, 12 plaintexts can be encrypt at once. Since ARMv8 has 32 vector registers, we utilized these registers in an optimal way. First, vector registers (`v0-v11`) are storing plaintext. Second, vector registers (`v12-v15`) have intermediate values, and the `v15` register is also used for saving the round key value. Third, `v16-v31` registers used for the S-Box look-up table. The SM4 encryption is performed on ARM processors as following order; Loading phase, Register transpose step, Round function layer, and Storing phase.

Instructions summary. Table 4 shows instructions for implementing the SM4 block cipher, in a parallel-way. Most of instructions are vector instructions, except the `ADR` instruction. The `ADR` instruction is used to the store address of S-Box table. The ARMv8 processor has 32 128-bit vector register, which can be calculate in a parallel-way. Some instructions require to specify the memory arrangement. In Table 4, the memory arrangement is omitted for the convenience.

Table 4. Instructions set for optimized implementation of the SM4 block cipher; `Xd`: destination scalar register, `Xn`: source scalar register, `Vd`: destination vector register, `Vt`: transferred vector register, `Vn, Vm`: source vector register.

asm	Operands	Description	Operation
ADR	Xd, (Label)	Form PC-relative address	Xd ← Label
EOR	Vd, Vn, Vm	Bitwise Exclusive OR	Td ← Vn ⊕ Vm
LD1	Vd1-4, (Xn)	Load multiple single-element structures	Vd1-4 ← (Xn)
LD1R	Vt, (Xn)	Load single-element and replicate to all lanes	Vt ← (Xn)
MOVI	Vt, #imm	Move Immediate	Vt ← #imm
SHL	Vd, Vn, #shift	Shift Left	Vd ← Vn << #shift
SRI	Vd, Vn, #shift	Shift Right and Insert	Vd ← Vn >> #shift
ST1	Vt1-4, (Xn)	Store multiple single-element structures	(Xn) ← Vt1-4
SUB	Vd, Vn, Vm	Subtract	Vd ← Vn - Vm
TBL	Vd, Vn, Vm	Table vector Lookup	Vd ← Vn[Vm]
TBX	Vd, Vn, Vm	Table vector lookup extension	Vd ← Vn[Vm]
UZP1	Vd, Vn, Vm	Unzip vectors primary	Vd ← Vn[even], Vm[even]
UZP2	Vd, Vn, Vm	Unzip vectors secondary	Vd ← Vn[odd], Vm[odd]

Loading phase. Algorithm 1 shows the implementation of Loading phase. Using 3 `LD1` instructions, 12 128-bit plaintexts are stored in vector registers (`v0-v11`). At this point, the post-incremented memory access is used to adjust the address pointer offset. Therefore, it is possible to reduce the execution time for calculating additional addresses. After that, the table look-up of S-Box is performed through `TBL` and `TBX` instructions.

Algorithm 1 Loading 12-plaintext in vector instructions.

Input: Memory address = [x1]
Output: Plaintexts = [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11]

```
1: LD1.4S v0, v1, v2 ,v3, [x1], #64
2: LD1.4S v4, v5, v6 ,v7, [x1], #64
3: LD1.4S v8, v9, v10 ,v11, [x1], #64
```

Register transpose step. Algorithm 2 is transpose step with `UZP1` and `UZP2` instructions in a source code level. The `UZP1` instruction reads an even numbered vector elements from the source register, and stores it to the destination register. The `UZP2` instruction does same operation, but read an odd numbered elements. In this process, registers are grouped by four and 32-bit blocks are arranged to be stored in one register. In total, 3 iterations are repeated to align 12 plaintexts. At the end of encryption, the transpose step is performed once again, to retrieve vector registers. Figure 3 shows the operation process of `UZP1` and `UZP2` instructions.

Algorithm 2 Alignment of the plaintext in vector instructions.

Input: PT0 = [v_a.4s], PT1 = [v_b.4s], PT2 = [v_c.4s], PT3 = [v_d.4s]

Output: X_0 = [v_a.4s], X_1 = [v_b.4s], X_2 = [v_c.4s], X_3 = [v_d.4s]

```
UZP1.4S v12, va, vb
UZP2.4S v13, va, vb
UZP1.4S v14, vc, vd
UZP2.4S v15, vc, vd

UZP1.4S va, v12, v14
UZP1.4S vb, v13, v15
UZP2.4S vc, v12, v14
UZP2.4S vd, v13, v15
```

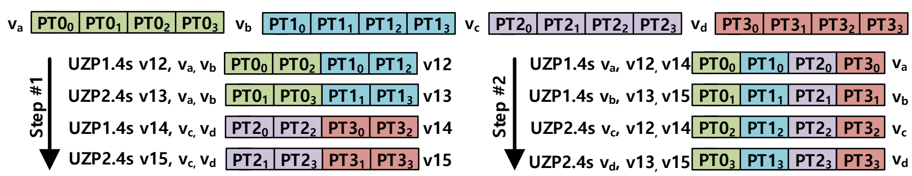

Fig. 3. `UZP1` and `UZP2` instructions process.

Round function layer. The source codes for Round function layer are shown at line 1-8 of Algorithm 3, which operates the nonlinear transformation (`tau`). It is implemented by `TBL` and `TBX` instructions to seek the S-box table. `TBL` and `TBX` instructions read a value from a vector element in the index source register, search each result as an index in the byte table of the source table register, and write the result to the destination register. The first 64 bytes of S-Box is extracted through the `TBL` instruction. The `TBX` instruction searches the table in the next range of previous `TBL` instruction. To search for the next branch of S-Box, subtraction to the value of the index source register by `0x40` and then using the `TBX` instruction are performed, subsequently.

In Algorithm 3, line 9-20 shows the source code that implements linear transformations (`L`) of the round function. The rotation operation is implemented using the left shift operations `SHL` and `SRI` instructions. Using only three registers (`v12, V13, v14`,), `v15` is used as a temporary register to store the round key value. In order to use only 3 registers, the rotation operation is performed and then XOR is performed, immediately.

Storing phase. In the last storing phase, the encryption result is saved. Algorithm 4 is to perform an operation that stores the ciphertext in the memory.

Algorithm 3 Round Function of the plaintext in vector instruction.

Input: S-Box input = [v_a.16b]
Output: S-Box output = [v_a.16b]
1: `MOVI v13.16b, #0x40`
2: `TBL v12.16b, v16.16b-v19.16b,` v_a`.16b`
3: `SUB` v_a`.16b,` v_a`.16b, v13.16b`
4: `TBX v12.16b, v20.16b-v23.16b,` v_a`.16b`
5: `SUB` v_a`.16b,` v_a`.16b, v13.16b`
6: `TBX v12.16b, v24.16b-v27.16b,` v_a`.16b`
7: `SUB` v_a`.16b,` v_a`.16b, v13.16b`
8: `TBX` v_a`.16b, v28.16b-v31.16b,` v_a`.16b`
9: `SHL.4s v13, v12, #2`
10: `SRI.4s v13, v12, #30`
11: `EOR.16b` v_a`, v12, v13`
12: `SHL.4s v13, v12, #10`
13: `SRI.4s v13, v12, #22`
14: `EOR.16b` v_a`, v13,` v_a
15: `SHL.4s v13, v12, #18`
16: `SRI.4s v13, v12, #14`
17: `EOR.16b` v_a`, v13,` v_a
18: `SHL.4s v13, v12, #24`
19: `SRI.4s v13, v12, #8`
20: `EOR.16b` v_a`, v13,` v_a

Algorithm 4 Storing 12-plaintexts in vector instruction.

Input: Ciphertexts = [`v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10 ,v11`]
Output: Memory address = [`x0`]
1: `st1.4s v0, v1, v2, v3, [x0], #64`
2: `st1.4s v4, v5, v6, v7, [x0], #64`
3: `st1.4s v8, v9, v10, v11, [x0], #64`

The result value (`v0-v11`) is stored in the memory address (`x0`) by 512-bits in a post incremental method, and 12 ciphertexts are stored by performing a total of 3 operations.

4 Evaluation

In this chapter, we present the evaluation of proposed implementations. The evaluation is conducted separately for each implementation environment.

4.1 Evaluation Criteria

The performance evaluation is based on clock cycles per byte (cpb). The notation (cpb) represents the number of clock cycles to process a byte.

4.2 Efficient Implementations of SM4 Block Cipher on 8-bit AVR Microcontrollers

Proposed implementations are targeted for the ATmega128 processor, which is one of AVR family. Source codes are implemented over `Microchip studio` framework, and compiled `-O2` option. Since there are no other SM4 block cipher implementations on AVR microcontrollers. Performance comparisons are

Table 5. Comparison result on 8-bit AVR microcontrollers. Symbols (s, m, and c) represent speed, memory, and code-optimized implementations, respectively.

Measurement	Reference C	**This work**^{s}	**This work**^{m}	**This work**^{c}
Timing [cpb]	1670.69	**205.2**	213.3	207.4
RAM [bytes]	418	418	**162**	418
ROM [bytes]	2856	5888	6144	**884**

Table 6. Comparison result of execution timing (cycles per byte) on 64-bit ARM Processors.

Reference C	**This work**
120.07	**8.62**

done with reference C code implementations. Comparison results are shown in Table 5. Reference C code takes 1670.69 cpb (clock cycles per byte), while the proposed speed-optimization implementation achieved 205.2 cpb, memory-optimization implementation recorded 213.3 cpb, and code-optimization implementation reached 207.4 cpb. The reason for result is that the proposed implementation is implemented in an optimal form using an AVR assembly. In particular, the efficient rotation is used in the Linear transformation (`L`), it makes better performance than the reference C code implementation. In addition, it can be compare each criteria. Speed-optimization achieved best performance than the others, Memory-optimization requires the least RAM size, and Code-optimization has the least ROM size.

4.3 Speed-optimization of SM4 Block Cipher on 64-bit ARM Processors

This section analyzes and evaluates the performance of the SM4 encryption implementation on ARMv8. It was written using Xcode and the calculation speed was measured by Apple A13 Bionic. The Apple a13 Bionic is a 64-bit ARM-based single chip (2.65 GHz) designed by Apple. The performance comparison is done with the reference code implemented in C language. Results are shown in Table 5. For the reference code, the execution timing is 120.07 cpb. The proposed implementation achieved 8.62 cpb, showing a performance improvement of 1,293%.

5 Conclusion

In this paper, we present optimized implementations of the SM4 block cipher on AVR microcontrollers and ARM processors. The AVR microcontroller achieves the performance improvement through optimized register allocation and instruction sets by pursuing speed, memory, and code size in a limited 8-bit environment. The ARM processor supports the high performance implementation with the parallel operation. The parallel computation enables the fast computation and uses it to achieve the optimized implementation. This paper will be helpful to implement the SM4 block cipher in various environments, including both low-end and high-end Internet of Things.

References

1. Cheng, H., Ding, Q.: 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, pp. 1628–1631. IEEE, Harbin, China (2012)
2. IETF, `https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10`. Last accessed 21 April 2018
3. Microchip document, `https://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf`. Last accessed 8 Nov 2014
4. Kim, Y.B., Kwon, H.D., An, S.W., Seo, H.J., Seo, C.S.: Efficient Implementation of ARX-Based Block Ciphers on 8-Bit AVR Microcontrollers. Mathematics **8**(10), 22 pages (2020)
5. Seo, H.J., Liu, Z., Longa, P., Hu, Z.: SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 1–20 (2018)
6. Kwon, H.D., An, S.W., Kim, Y.B., Kim, H.J., Choi, S.J., Jang, K.B., Park, J.H., Kim, H.J., Seo, S.C., Seo, H.J.: Designing a CHAM Block Cipher on Low-End Microcontrollers for Internet of Things. Electronics **9**(9), 16 pages (2020)
7. Kwon, H.D., Kim, H.J., Choi, S.J., Jang, K.B., Park, J.H., Kim, H.J., Seo, H.J.: Compact Implementation of CHAM Block Cipher on Low-End Microcontrollers. In: You I. (eds) Information Security Applications. WISA 2020. Lecture Notes in Computer Science, vol 12583. pp. 127–141. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65299-9_10
8. Seo, H.J., Kwon, H.D., Kim, H.J., Park, H.H.: ACE: ARIA-CTR Encryption for Low-End Embedded Processors. Seonsors **20**(13), 15 pages (2020)
9. Kwon, H.D., Kim, Y.B., Seo, S.C., Seo, H.J.: High-Speed Implementation of PRESENT on AVR Microcontroller. Mathematics **9**(4), 15 pages (2021)
10. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier P., Verbauwhede I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, LNCS, vol. 4727, pp 450–466. Springer, Berlin, Heidelberg (2007) https://doi.org/10.1007/978-3-540-74735-2_31
11. Park, J.H., Lee, D.H.: FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 469–499 (2018)
12. Kim, K.H., Choi, S.J., Kwon, H.D., Zhe, L., Seo, H.J.: Fast AES–CTR Mode Encryption for Low-End Microcontrollers. In: Seo J. (eds.) Information Security and Cryptology – ICISC 2019, LNCS, vol. 11975, pp 102–114. Springer, Cham (2019) https://doi.org/10.1007/978-3-030-40921-0_6