Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2020.DOI

Optimized Implementation of SM4 on AVR Microcontrollers, RISC-V Processors, and ARM Processors

HYEOKDONG KWON[1] HYUNJUN KIM[1] SIWOO EUM[2] MINJOO SIM[2] HYUNJI KIM[1] WAI-KONG LEE[3] ZHI HU[4] and HWAJEONG SEO[2]

[1]Dept of Information Computer Engineering, Hansung University, Seoul, South Korea (e-mail: {korlethean, khj930704, khj1594012}@gmail.com)
[2]Division of IT Convergence engineering, Hansung University, Seoul, South Korea (e-mail: {shuraatum, minjoos9797, hwajeong84}@gmail.com)
[3]Department of Computer Engineering, Gachon University, Seongnam, South Korea (e-mail: waikong.lee@gmail.com)
[4]School of Mathematics and Statistics, Central South University, Changsha, China (e-mail: huzhi$_m ath@csu.edu.cn$)

Corresponding author: Hwajeong Seo (e-mail: hwajeong84@gmail.com).

ABSTRACT At 2003, the SM4 block cipher was introduced that is a Chinese domestic cryptographic. It is mandated in the Chinese National Standard for Wireless LAN Wired Authentication and Privacy Infrastructure (WAPI), because the algorithm was developed for use in wireless sensor networks to provide safety network environment. The SM4 block cipher uses a 128-bit block size and a 32-bit round key. It 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; the optimized implementation of the SM4 block cipher on 32-bit RISC-V processors, which are open-source-based computer architectures, and the optimized implementation of SM4 on 64-bit ARM processors with parallel computation, which are widely used in smartphones and tablets. In the AVR microcontroller, three versions are implemented for various purposes, including speed-optimization, memory-optimization, and code size-optimization. As a result, the speed-optimization, memory-optimization, and code size-optimization versions 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 language (1670.7 cycles per byte). The implementation on 32-bit RISC-V processors achieved 128.8 cycles per byte. This is faster than the reference implementation written in C language (345.7 cycles per byte). The implementation on 64-bit ARM processors achieved 8.62 cycles per byte. This is faster than the reference implementation written in C language (120.07 cycles per byte).

INDEX TERMS 8-bit AVR Microcontrollers, 32-bit RISC-V Processors, 64-bit ARM Processors, Software Implementation, SM4 Block Cipher

I. INTRODUCTION

Numerous sensor nodes are used to collect the data in wireless sensor networks. Tiny sensor nodes have limited computation resources, such as computing power, memory space, and battery life. However, most cryptographic algorithms are based on complex mathematical problems, so it is difficult to operate cryptographic algorithms on IoT devices. Thus, a Lightweight block cipher algorithms has been proposed to operate cryptographic algorithms on insufficient environment. Lightweight cryptography algorithms require less resources than ordinary cryptographic algorithms. The SM4 block cipher is a one of lightweight block cipher family, which is the Chinese National Standard for wireless LAN Wired Authentication and Privacy Infrastructure (WAPI) [1]. The SM4 block cipher, which first appeared in 2003, and now used as a standard block cipher in China. SM4 is very suitable for hardware implementation, so there are many research results of hardware implementation of SM4. For instance, [2] shows the optimized field-programmable gate array design for SM4, and it uses 7% less resources than the pipelined implementation. We focused on software implementation, not hardware implementation.

In this paper, we propose optimized implementations of the SM4 block cipher in terms of software implementation. We targeted three platforms that low-end 8-bit **AVR micro-**

Table 1: Parameters for the SM4 block cipher.

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

controllers, 32-bit **RISC-V processors** and high-end 64-bit **ARM processors**. The main contributions of this work are summarized below.

A. CONTRIBUTIONS

- **Optimized implementations of the SM4 block cipher on 8-bit AVR microcontrollers.** SM4 block cipher uses a 128-bit block size. However, AVR microcontrollers has only 70-bit wise general purpose registers. Therefore, an effective register scheduling plan should be considered. We make register allocation plan so that it is an efficient implementation of SM4 block cipher. Furthermore, SM4 block cipher requires the 32-bit wise rotation operation, whereas AVR microcontrollers only can perform 8-bit wise operations. We propose how to efficiently implement 32-bit rotation through 8-bit instructions on AVR microcontrollers.
- **Optimized implementations of the SM4 block cipher on 32-bit RISC-V processors.** RISC-V is a brand new open-source based computer architecture, which is supports new kinds of instruction sets. In this paper, present the first optimized implementation of SM4 on 32-bit RISC-V processors. The focus of optimized implementation is to implement S-Box operation by using RISC-V instructions.
- **Parallel implementations of the SM4 block cipher on 64-bit ARM processors.** 64-bit ARM processors has vector registers and vector instructions that can process on data in parallel (Single Instruction Multiple Data, SIMD). In this paper, we propose SM4 block cipher model that encrypts 12 plaintexts in parallel approach. It can be implement through vector instructions of ARMv8 processor. Also, ARMv8 processor has 32 vector registers, so focus on plaintext register and S-Box register configuration.

II. BACKGROUND

A. SM4 BLOCK CIPHER

SM4 block cipher is a Chinese domestic cryptographic system that was first published in 2003. It was established as a cryptographic standard by the Office of State Commercial Cryptography Administration (OSCCA) [3]. SM4 parameters are listed in Table 1. Also, left part of Figure 1 presents encryption tasks and internal structure of SM4 block cipher.

SM4 block cipher consists of five computation components; round function (F), permutations (T and T′), nonlinear transformation (tau), linear transformations (L) and (L′), and S-Box (S).

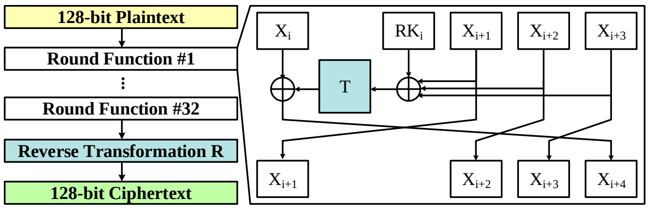


Figure 1: Encryption flow of the SM4 block cipher and the round function structure. Left: encryption tasks, Right: round function structures.

1) Round function (F).

128-bit Plaintext of SM4 block cipher is splitted into four 32-bit units, which named X. Round function F needs five arguments that plaintext fragments (X_0, X_1, X_2, and X_3), and round key. F is defined by 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 side of Figure 1 shows Round function F structure.

2) Permutations T and T′.

Permutation functions T and T′ require 32-bit input values, and return 32-bit output values. These two functions has reversible feature, and consisted with tau and L.

3) Nonlinear transformation tau.

Nonlinear transformation (tau) takes four S-Boxes, which requires 32-bit inputs and makes 32-bit outputs. It can be performed parallel-way. Also, input values do not affect each other. Nonlinear transformation tau can be represented as follows, where A and B are a 32-bit input value and a 32-bit output value, respectively. The type of a_i and b_i is an 8-bit-wise string.

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

4) Linear transformations L, and L′.

Linear transformations (L, and L′) perform 32-bit wise rotation operations. Its input values are output of Nonlinear transformation tau. L, and L' are defined as follows, where B is a 32-bit input value, and ROTL represents rotation to the left.

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

5) S-Box S.

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

B. TARGET PROCESSOR: 8-BIT LOW-END AVR MICROCONTROLLER.

An AVR microcontroller has 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 the instructions take less than four clock cycles [4]. We evaluated the performance on an ATmega128, which is an 8-bit AVR microcontroller. It has 128KB of programmable flash memory, 4KB internal SRAM, 4KB EEPROM, and 64KB optional external memory space [5]. AVR registers are denoted as `R0` to `R31`. Some registers have the following special features:

- **ZERO register:** `R1` is the zero register, which 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 save important values (i.e. long-lived values and data from a callee). These must be preserved in the stack before they are 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, they are written as `X` (`R26`–`R27`), `Y` (`R28`–`R29`), `Z` (`R30`–`R31`) notation. `R28`–`R29` are also callee saved registers.

C. TARGET PROCESSOR: 32-BIT RISC-V PROCESSORS.

RISC-V is a new computer CPU structure that has been under development at UC Berkeley since 2010. It is not intended just for academic or research purposes, but for industrial commercialization. The main feature of the RISC-V processor is that the basic instruction set is provided by the consortium, but there are no restrictions on the extended instructions that users can add. Therefore, when utilizing this, it is possible to increase the speed of the target application service by customizing the RISC-V processor. In this paper, the 32-bit structure RV32I used for performance comparison provides 32-bit registers 32 (`x0`–`x31`) [6].

D. 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, namely, 32-bit AArch32 and 64-bit AArch64. In this paper, we targeted the AArch64 architecture, which we simply refer to A64. A64 has 32 64-bit general-purpose scalar registers, which can handle 32-bit, and 64-bit data. In addition, there are 32 128-bit vector registers, which can be utilized for the parallel implementation with SIMD [7]. We used vector registers for parallel implementation the SM4 block cipher.

E. RELATED WORKS.

In this section, we introduce optimized implementations of block ciphers on embedded processors. In [8], a revised version of CHAM was optimized on 8-bit AVR microcontrollers. In [8], they suggested optimized 8-bit-wise rotation and 32-bit-wise rotation. This implementation utilized a pre-calculation technique with a counter mode of operation. In [9], parallel implementations were presented. In [10], an optimized ARIA block cipher was presented. They optimized primitive operations, including rotation operation, a substitute-layer, and a diffusion-layer on a low-end AVR microcontroller. In [11], they proposed a compact implementation of PRESENT block cipher, which was introduced at CHES'07 [12]. It optimally implemented the PRESENT through a pre-computation technique. In [13], a compact implementation of an Advanced Encryption Standard (AES) block cipher on Intel processors was presented (i.e. FACE). This implementation applied a pre-computation technique that pre-calculates repetitive values and reuse them. At ICISC'19, they proposed optimized implementation of FACE on an AVR microcontroller [14]. It extended the pre-computation to round 3. This implementation is also secure against Correlation Power Analysis (CPA). SIMON [15] is a block cipher announced by the US National Security Agency in 2015, and there are results of optimized implementation of SIMON with pre-computation counter mode applied on AVR microcontrollers [16]. Lightweight block cipher PIPO [17] is a block cipher announced in Korea in 2020, and an optimal implementation with side-channel attack resistance was proposed in 2021 [18].

Research on optimized implementation for RISC-V as well as AVR microcontrollers is active. In [19], and [20] multiplication primitive optimized technique was presented. The proposed technique optimizes polynomial-multiplication, allowing multiplication to be calculated at a high speed. In [21], the optimal implementation of the block cipher CHAM on RISC-V is the result of a high-speed operation using a technique that omits block movement in the internal operation of CHAM. There are also research results that implemented various cryptography using the assembly instructions of RISC-V. [22] used RV32I instruction set to implement optimized table-based AES, bitsliced AES, ChaCha stream cipher, and Keccak-f[1600] permutation inside SHA3.

ARM processors are much more powerful than AVRs and RISC-Vs, and are often used in high-end mobile devices such as smartphones. ARIA [23] block cipher was established as a Korean standard in 2004 and an international standard in 2010. In 2021, optimized implementation of ARIA on ARM Cortex-M3 was proposed [24]. Also, Format-Preserving Encryption, which is often used in IoT devices, was implemented on AVR and ARM processor [25]. The implementation was targeted for ARMv8, a 64-bit ARM processor. In [26], which optimally implemented the block cipher PIPO on the ARM processor, applied the parallel optimal implementation technique using the vector register of the ARM processor. FrodoKEM [27] is one of the NIST Post Quantum Cryptography Standardization Round 3 Alternate Candidate, and the optimized implementation on ARM processor has been proposed [28]. The proposed technique uses the vector register of ARMv8 to parallelize the multiplication process of FrodoKEM and greatly optimizes the time required for multiplication.

Although it is not an optimized implementation, there is also a study of side-channel attacks on SM4 block cipher. In [29] tried collision-based attack, however it has more efficiently attack then [30]. The main ideas of [29], first they combined the look-up tables `T-boxes`. Second, to remove the effect of dual cipher, it shuffled the order of data in the `T-boxes`. Third, it made suitable collision function for attack to white-box SM4 implementation.

III. OPTIMIZED IMPLEMENTATION OF THE SM4 BLOCK CIPHER

This section presents the optimized implementation of SM4 block cipher on 8-bit AVR microcontrollers, 32-bit RISC-V processors, and 64-bit ARM processors. Optimal performance is achieved through efficient register scheduling plan and instruction techniques.

A. 8-BIT LOW-END AVR MICROCONTROLLERS

1) Instruction set.

8-bit AVR microcontrollers has powerful instruction sets. In general, AVR instructions take one or two clock cycles. Table 2 lists AVR instructions used to implement optimized SM4 block cipher.

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

Ins.	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

2) Register utilization.

AVR microcontroller has 8-bit general purpose registers, but only 32 are provided. So it must be need to efficient register scheduling plan, because SM4 takes many operation process steps.

- **X blocks.** In Section II-A, the SM4 block cipher stores 128-bit plaintext into four 32-bit X. However, 8-bit AVR microcontrollers have 8-bit-wise registers that can only represent 8-bit data. Four registers are required to handle one 32-bit X. As a result, there are four X that each handle a quarter of plaintext. A total of 16 registers are required to store the whole plaintext.
- **Round key, and `T` input/output.** Each `F` requires a 32-bit round key. Four 8-bit registers are 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 the results of `T`.
- **Nonlinear operation.** Nonlinear transformation (`tau`) performs the rotation operation. Eight registers are required for the result and intermediate values of rotation. Four out of eight registers store the `tau` output result.
- **Address pointer.** To load a value into a register on AVR microcontrollers, it must be accessed through an address pointer. In this case, there are 3 kinds of values for the function call, namely, plaintext, round key, and S-Box values. We allocate an `X` pointer for loading plaintext, and storing ciphertext, a `Y` pointer for round key, and a `Z` pointer for S-Box values. In particular, the `X` pointer address (`R26` and `R27`) is not needed to during round functions. These registers are used to store temporary values. The `R30` register is always fixed to 0 value, because it stores the lower address of S-Box. This can be used as a 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, only one register is needed to store loop index. This register is shared with the temporary value register. It must to preserve an index value on the stack and can be implemented through `PUSH` and `POP` instructions.

Figure 2 shows all of register allocation plan. Each rectangle represents single 8-bit register and two-colored registers mean multiple purposes registers.

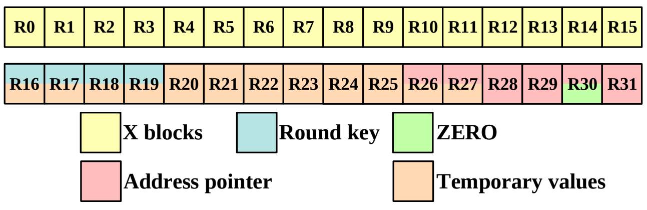


Figure 2: Register allocation of AVR for the SM4.

3) Optimized implementation of 32-bit wise rotation.

SM4 block cipher uses 32-bit wise operation, but AVR microcontrollers has only 8-bit registers that can be performed 8-bit wise operation. But, 32-bit wise rotation can be implemented with some instructions that are `LSL, ROL, ADC, MOV,` and `MOVW`. Table 3 shows implementation codes for each rotation. The operation speed can be increased by different input and output registers for 8, 16 and 24 rotation operation. It reduces the time it takes for instructions to store values in temporary registers and return them. Figure 3 shows this difference. Figure 3 shows only the case of 8 rotations, but the other cases are the same.

Table 3: Optimized 32-bit wise rotation operation on 8-bit environments, where i and j represent specific 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

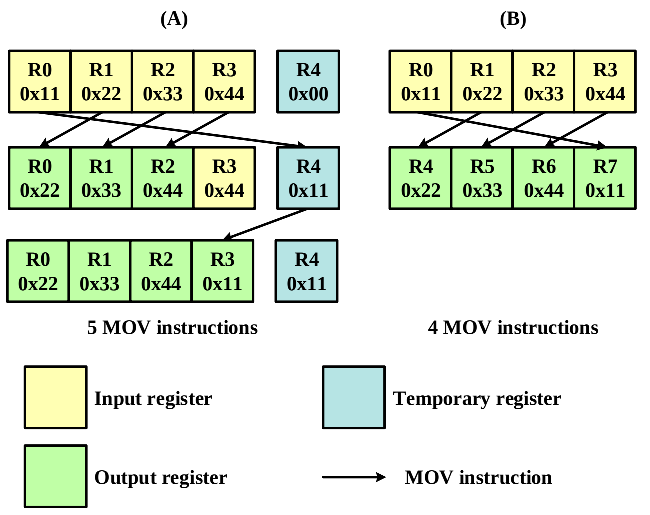


Figure 3: Difference in rotation operation according to the use of temporary registers. (A): using temporary register, the input and output registers are the same, and 5 MOV instructions are used. (B): temporary register not used, input and output registers are different, and 4 MOV instruments are used.

4) Efficient S-Box implementation.

In this work, there are three optimization approaches to AVR microcontrollers (speed-optimization, memory-optimization, and code size-optimization). In terms of speed-optimization, storing the S-Box in RAM is effective. The LD instruction loads the S-Box value with two clock cycles, which can get the S-Box value quickly. On the other hand, from the memory-optimization perspective, the S-Box can be saved to flash memory. In Section II-B, it was confirmed that the AVR microcontroller has larger flash memory than RAM. Therefore, the memory-optimized implementation can be useful in situations in which there is the lack of RAM. The memory-optimization can be implemented with the LPM instruction, which takes three clock cycles. Thus, the memory-optimization implementation takes a longer execution time than the speed-optimization implementation. For the code size-optimization approach, we utilized a looped implementation, which sacrificed the performance but achieved the optimal code size.

B. 32-BIT RISC-V PROCESSORS

A 32-bit RISC-V processor supports 32-bit wise instructions. This is useful to perform the 32-bit wise operations of SM4. For the optimal implementation in RISC-V, we propose rotation optimization and efficient S-Box implementation.

1) Optimized Rotation Implementation.

Rotation operation is not supported in RISC-V. Therefore, rotation is implemented using the `SLLI`, `SRLI`, and `OR` instructions. `ROL`(n) can be implemented by `OR` the value of `SLLI`(n) and `SRLI`$(32-n)$. This process can be seen in Figure 4.

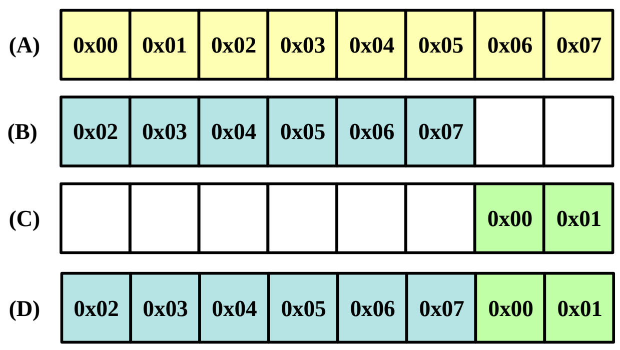


Figure 4: The process of rotation operation on RISC-V. (A): Original value, (B): SLLI(n) applied to (A), (C): SRLI(32-n) applied to (B), (D): result value of (B) or (C).

2) Efficient S-Box implementation.

RISC-V uses 32-bit registers. However, in the S-Box, it is converted to a pre-computed value in bytes. Therefore, it is necessary to convert a 32-bit value by dividing it into 8-bit units. For the implementation, a stack pointer (SP) and load unsigned byte (LUB) are used. The SP has the address of the current stack, and the LBU loads only the 1-byte value of the indicated address. The S-Box process is the same as Algorithm 1. In Algorithm 1, the result value of the S-Box is stored in T1, and A2 has the S-Box address.

Algorithm 1 Efficient S-Box implementation in RISC-V.

```
Input: S-Box In = T0
Output: S-Box Out = T1
 1: SW T0, 0(SP)
 2: LBU T1, 3(SP)
 3: ADD T0, A2, T1
 4: LBU T1, 0(T0)
 5: SLLI T1, T1, 24
 6: LBU T2, 2(SP)
 7: ADD T0, A2, T2
 8: LBU T2, 0(T0)
 9: SLLI T2, T2, 16
10: XOR T1, T1, T2
11: LBU T2, 1(SP)
12: ADD T0, A2, T2
13: LBU T2, 0(T0)
14: SLLI T2, T2, 8
15: XOR T1, T1, T2
16: LBU T2, 0(SP)
17: ADD T0, A2, T2
18: LBU T2, 0(T0)
19: XOR T1, T1, T2
```

Table 4: Instructions set of ARM64 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	Bit-wise 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) ← Vt1-34
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]

C. 64-BIT HIGH-END ARM PROCESSORS

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

1) Instructions summary.

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

2) Loading phase.

Algorithm 2 shows the implementation of the loading phase. Using three `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 2 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
```

3) Register scheduling plan.

ARM64 has 31 general purpose registers and 32 vector registers. Each vector register has a size of 128-bit, so 12 vector registers are used to store 12 plaintexts. Since operations are performed between registers, there is no additional register use. The remaining registers store the S-Box. Vector registers can use arrangement specifiers to adjust the arithmetic units of their internal values. For example, if 'b' is used, calculations are performed in units of 8 or 16 8-bits. Figure 5 shows an example of a register scheduling scheme and arrangement specifier.

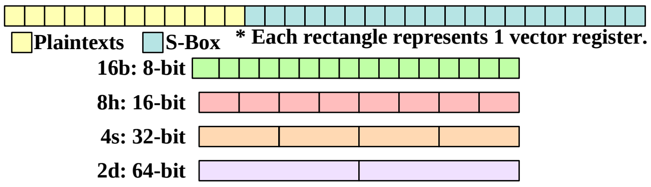


Figure 5: Register scheduling plan and description of arrangement specifier for ARM64.

4) Register transpose step.

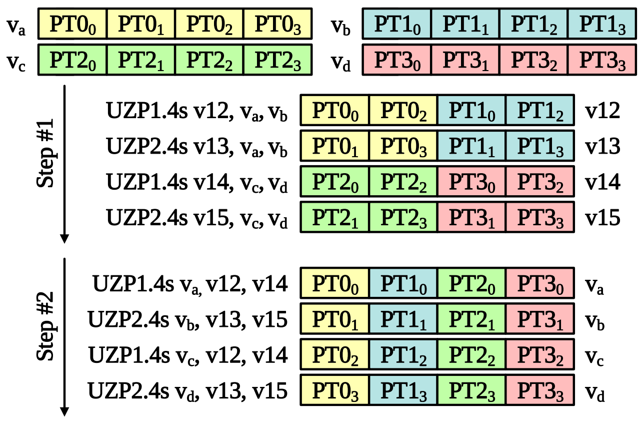
Algorithm 3 is the transpose step with `UZP1` and `UZP2` instructions at the a source-code level. The `UZP1` instruction reads even-numbered vector elements from the source register, and stores them in the destination register. The `UZP2` instruction does a similar operation, but reads odd-numbered elements. In this process, registers are grouped by four, and 32-bit blocks are arranged to be stored in one register. In total, three iterations are repeated to align 12 plaintexts. At the end of encryption, the transpose step is performed once again to retrieve vector registers. Figure 6 shows the operation process of `UZP1` and `UZP2` instructions.

Algorithm 3 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]
1: UZP1.4S v12, $v_a$, $v_b$
2: UZP2.4S v13, $v_a$, $v_b$
3: UZP1.4S v14, $v_c$, $v_d$
4: UZP2.4S v15, $v_c$, $v_d$
5: UZP1.4S $v_a$, v12, v14
6: UZP1.4S $v_b$, v13, v15
7: UZP2.4S $v_c$, v12, v14
8: UZP2.4S $v_d$, v13, v15
```

5) Round function layer.

Source codes for the round function layer are shown at lines 1–8 of Algorithm 4, which carries out the nonlinear transformation (`tau`). It is implemented by `TBL` and `TBX` instructions to seek the S-Box table. The `TBL` and `TBX`



v_a $PT0_0$ $PT0_1$ $PT0_2$ $PT0_3$
v_b $PT1_0$ $PT1_1$ $PT1_2$ $PT1_3$
v_c $PT2_0$ $PT2_1$ $PT2_2$ $PT2_3$
v_d $PT3_0$ $PT3_1$ $PT3_2$ $PT3_3$
Step #1
UZP1.4s v12, v_a, v_b → $PT0_0$ $PT0_2$ $PT1_0$ $PT1_2$ v12
UZP2.4s v13, v_a, v_b → $PT0_1$ $PT0_3$ $PT1_1$ $PT1_3$ v13
UZP1.4s v14, v_c, v_d → $PT2_0$ $PT2_2$ $PT3_0$ $PT3_2$ v14
UZP2.4s v15, v_c, v_d → $PT2_1$ $PT2_3$ $PT3_1$ $PT3_3$ v15
Step #2
UZP1.4s v_a, v12, v14 → $PT0_0$ $PT1_0$ $PT2_0$ $PT3_0$ v_a
UZP2.4s v_b, v13, v15 → $PT0_1$ $PT1_1$ $PT2_1$ $PT3_1$ v_b
UZP1.4s v_c, v12, v14 → $PT0_2$ $PT1_2$ $PT2_2$ $PT3_2$ v_c
UZP2.4s v_d, v13, v15 → $PT0_3$ $PT1_3$ $PT2_3$ $PT3_3$ v_d

Figure 6: `UZP1` and `UZP2` instructions process for SM4.

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 the S-Box are extracted through the `TBL` instruction. The `TBX` instruction searches the table in the next range of the previous `TBL` instruction. To search for the next S-Box branch, subtraction to the value of the index source register by `0x40` is performed, and then the `TBX` instruction is carried out.

In Algorithm 4, lines 9–20 show 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. To use only three registers, the rotation operation is performed and then XOR is performed immediately.

6) Storing phase.

In the last storing phase, the encryption result is saved. Algorithm 5 is to perform an operation that stores the ciphertext in the memory. 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 through a total of three operations.

IV. EVALUATION

In this section, we present the evaluation of proposed implementations. The evaluation was conducted separately for each implementation environment. The performance evaluation was based on clock cycles per byte (cpb).

Table 5: Comparison result on 8-Bit AVR microcontrollers. Symbols (s, m, and c) represent speed, memory, and code size-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**

Algorithm 4 Round Function of the plaintext in vector instruction.

```
Input: S-Box In = [v_a.16b]
Output: S-Box Out = [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 5 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
```

A. EFFICIENT IMPLEMENTATIONS OF SM4 BLOCK CIPHER ON 8-BIT AVR MICROCONTROLLERS

The proposed implementations are intended for the ATmega128 processor, which is an AVR. Source codes were implemented over the `Microchip Studio` framework and compiled using the `-O2` option. Because there are no other SM4 block cipher implementations on AVR microcontrollers, performance comparisons were done with reference C code implementations. The 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, the memory-optimization implementation recorded 213.3 cpb, and the code size-optimization implementation reached 207.4 cpb. These excellent results are that the proposed implementation is implemented in an optimal form using an AVR assembly. In particular, efficient rotation is used in the linear transformation (`L`); thus, it achieves better performance than the reference C code implementation. In addition, it can be compared in terms of each criteria. The speed-optimization approach achieved better performance than the others, the

Table 6: Comparison result of execution timing (cycles per byte) on 32-bit RISC-V processors (left) and 64-bit ARM processors (right).

RISC-V		ARM	
Reference C	**This work**	Reference C	**This work**
345.7	**128.8**	120.07	**8.62**

memory-optimization approach requires the least RAM size; and the code size-optimization approach has the smallest ROM size.

B. IMPLEMENTATIONS OF SM4 BLOCK CIPHER ON 32-BIT RISC-V PROCESSORS

In this section we present the analysis and evaluation of the performance of the SM4 encryption implementation on RISC-V. In this work, the implementation was not applied any kinds of optimization techniques. It is only proceed performance measurements on RISC-V. The RISC-V implementation does not use extensions and relies on the RV32I-based ISA. For the performance measurement, the HiFive1 Rev B development board with a 32-bit E31 RISC-V core was used. The results are shown in the left part of Table 6. For the reference code, the execution timing was 345.7 cpb. The implementation achieved 128.8 cpb, which reflects a performance improvement by 2.68×.

C. SPEED-OPTIMIZATION OF SM4 BLOCK CIPHER ON 64-BIT ARM PROCESSORS

In this section we present the analysis and evaluation of 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 was done with the reference code implemented in C language. The results are shown in the right part of Table 6. For the reference code, the execution timing was 120.07 cpb. The proposed implementation achieved 8.62 cpb, which reflects a performance improvement by 12.93×.

V. CONCLUSION

In this paper, optimized implementation of SM4 block cipher was introduced. The target environments are 8-bit AVR microcontrollers, 32-bit RISC-V processors, and 64-bit ARM processors. Proposed implementation improved performance of SM4 block cipher compared to previous approaches. We hope that proposed techniques becomes helpful to implement SM4 block cipher in various environments, including both low-end and high-end IoT environments.

References

[1] H. Cheng and Q. Ding, "Overview of the block cipher," in 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, pp. 1628–1631, IEEE, 2012.

[2] S. Abed, R. Jaffal, B. J. Mohd, and M. Alshayeji, "Performance evaluation of the SM4 cipher based on field-programmable gate array implementation," IET Circuits, Devices & Systems, vol. 15, no. 2, pp. 121–135, 2021.

[3] "Internet engineering task force." https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10. Accessed: 2018-04-21.

[4] "Microchip document." https://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf. Accessed: 2014-08-11.

[5] Y. Kim, H. Kwon, S. An, H. Seo, and S. C. Seo, "Efficient implementation of ARX-based block ciphers on 8-bit AVR microcontrollers," Mathematics, vol. 8, no. 10, p. 1837, 2020.

[6] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual volume 2: Privileged architecture version 1.7," tech. rep., University of California at Berkeley Berkeley United States, 2015.

[7] H. Seo, Z. Liu, P. Longa, and Z. Hu, "SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 1–20, 2018.

[8] H. Kwon, H. Kim, S. J. Choi, K. Jang, J. Park, H. Kim, and H. Seo, "Compact implementation of CHAM block cipher on low-end microcontrollers," in International Conference on Information Security Applications, pp. 127–141, Springer, 2020.

[9] H. Kwon, S. An, Y. Kim, H. Kim, S. J. Choi, K. Jang, J. Park, H. Kim, S. C. Seo, and H. Seo, "Designing a CHAM block cipher on low-end microcontrollers for internet of things," Electronics, vol. 9, no. 9, p. 1548, 2020.

[10] H. Seo, H. Kwon, H. Kim, and J. Park, "ACE: ARIA-CTR encryption for low-end embedded processors," Sensors, vol. 20, no. 13, p. 3788, 2020.

[11] H. Kwon, Y. Kim, S. C. Seo, and H. Seo, "High-speed implementation of PRESENT on AVR microcontroller," Mathematics, vol. 9, no. 4, p. 374, 2021.

[12] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in International workshop on cryptographic hardware and embedded systems, pp. 450–466, Springer, 2007.

[13] J. H. Park and D. H. Lee, "FACE: Fast AES CTR mode encryption techniques based on the reuse of repetitive data," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 469–499, 2018.

[14] K. Kim, S. Choi, H. Kwon, Z. Liu, and H. Seo, "FACE–LIGHT: fast AES–CTR mode encryption for low-end microcontrollers," in International Conference on Information Security and Cryptology, pp. 102–114, Springer, 2019.

[15] 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, pp. 1–6, 2015.

[16] H.-D. Kwon, K.-B. Jang, H.-J. Kim, and H.-J. Seo, "The fast implementation of block cipher SIMON using pre-computation with counter mode of operation," Journal of the Korea Institute of Information and Communication Engineering, vol. 25, no. 4, pp. 588–594, 2021.

[17] H. Kim, Y. Jeon, G. Kim, J. Kim, B.-Y. Sim, D.-G. Han, H. Seo, S. Kim, S. Hong, J. Sung, et al., "PIPO: A lightweight block cipher with efficient higher-order masking software implementations," in International Conference on Information Security and Cryptology, pp. 99–122, Springer, 2020.

[18] H. Kim, M. Sim, S. Eum, K. Jang, G. Song, H. Kim, H. Kwon, W.-K. Lee, and H. Seo, "Masked implementation of PIPO block cipher on 8-bit AVR microcontrollers," in International Conference on Information Security Applications, pp. 171–182, Springer, 2021.

[19] H.-j. Seo, H.-d. Kwon, K.-b. Jang, and H. Kim, "Optimized implementation of scalable multi-precision multiplication method on RISC-V processor for high-speed computation of post-quantum cryptography," Journal of the Korea Institute of Information Security & Cryptology, vol. 31, no. 3, pp. 473–480, 2021.

[20] H. Seo, H. Kwon, S. Eum, K. Jang, H. Kim, H. Kim, M. Sim, G. Song, and W.-K. Lee, "All the polynomial multiplication you need on RISC-V," Cryptology ePrint Archive, 2021.

[21] M.-J. Sim, S.-W. Eum, H.-D. Kwon, G.-J. Song, and H.-J. Seo, "Implementation of ultra-lightweight block cipher algorithm revised CHAM on 32-bit RISC-V processor," in Proceedings of the Korea Information Processing Society Conference, pp. 217–220, Korea Information Processing Society, 2021.

[22] K. Stoffelen, "Efficient cryptography on the RISC-V architecture," in International Conference on Cryptology and Information Security in Latin America, pp. 323–340, Springer, 2019.

[23] D. Kwon, J. Kim, S. Park, S. H. Sung, Y. Sohn, J. H. Song, Y. Yeom, E. Yoon, S. Lee, J. Lee, et al., "New block cipher: ARIA," in International

conference on information security and cryptology, pp. 432–445, Springer, 2003.

[24] H. Seo, H. Kim, K. Jang, H. Kwon, M. Sim, G. Song, and S. Uhm, "Compact implementation of ARIA on 16-bit MSP430 and 32-bit ARM Cortex-M3 microcontrollers," Electronics, vol. 10, no. 8, p. 908, 2021.

[25] H. Kim, M. Sim, K. Jang, H. Kwon, S. Uhm, and H. Seo, "Masked implementation of format preserving encryption on low-end AVR microcontrollers and high-end ARM processors," Mathematics, vol. 9, no. 11, p. 1294, 2021.

[26] S. W. Eum, H. D. Kwon, H. J. Kim, K. B. Jang, H. J. Kim, J. H. Park, G. J. Song, M. J. Sim, and H. J. Seo, "Optimized implementation of block cipher PIPO in parallel-way on 64-bit ARM processors," KIPS Transactions on Computer and Communication Systems, vol. 10, no. 8, pp. 223–230, 2021.

[27] M. Naehrig, E. Alkim, J. W. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, V. Nikolaenko, C. Peikert, et al., "FrodoKEM: learning with errors key encapsulation," NIST PQC Round, vol. 2, p. 4, 2020.

[28] H. Kwon, K. Jang, H. Kim, H. Kim, M. Sim, S. Eum, W.-K. Lee, and H. Seo, "ARMed Frodo," in International Conference on Information Security Applications, pp. 206–217, Springer, 2021.

[29] R. Wang, H. Guo, J. Lu, and J. Liu, "Cryptanalysis of a white-box SM4 implementation based on collision attack," IET Information Security, vol. 16, no. 1, pp. 18–27, 2022.

[30] Y. Shi, W. Wei, and Z. He, "A lightweight white-box symmetric encryption algorithm against node capture for WSNs," Sensors, vol. 15, no. 5, pp. 11928–11952, 2015.

HYEOKDONG KWON received the B.S. and M.S. degrees in IT convergence engineering at Hansung University. He is currently Ph.D candidate in Hansung university. His research interests include cryptography implementation, information security, and machine learning.

HYUNJUN KIM received the B.S. and M.S. degrees in IT convergence engineering at Hansung University. He is currently Ph.D candidate in Hansung university. His research interests include side-channel analysis, and cryptogrpahy implementation.

SIWOO EUM received the B.S. degrees in IT convergence engineering at Hansung University. He is currently M.S. candidate in Hansung university. His research interests include cryptography implementation, and information security.

MINJOO SIM received the B.S. degrees in IT convergence engineering at Hansung University. She is currently M.S. candidate in Hansung university. Her research interests include cryptography implementation, and information security.

HYUNJI KIM received the B.S. and M.S. degrees in IT convergence engineering at Hansung University. She is currently Ph.D candidate in Hansung university. Her research interests include artificial intelligence, machine learning, and information security.

WAI-KONG LEE (Member, IEEE) received the B.Eng. degree in electronics and the M.Eng.Sc. degree from Multimedia University, in 2006 and 2009, respectively, and the Ph.D. degree in engineering from University Tunku Abdul Rahman (UTAR), Malaysia, in 2018. From 2009 to 2012, he served as a Research and Development Engineer for several multinational companies, including Agilent Technologies (now known as Keysight), Malaysia. He served as an Assistant Professor and the Deputy Dean (research and development) for the Faculty of Information and Communication Technology, UTAR. He was a Visiting Scholar at Carleton University, Canada, in 2017, Feng Chia University, Taiwan, in 2016 and 2018, and OTH Regensburg, Germany, in 2015, 2018, and 2019. He is currently a Postdoctoral Researcher at Gachon University, South Korea. His research interests include cryptography, GPU computing, numerical algorithms, the Internet of Things (IoT), and energy harvesting. He served as a Reviewer for several international journals, such as IEEE Transactions on Dependable and Secure Computing, in 2016 and 2017, IEEE Sensors Journal, IEEE Internet of Things Journal, from 2018 to 2021, and IEEE Transactions on Industrial Informatics, from 2018 to 2021.

ZHI HU received the BS and PhD degrees from the School of Mathematical Sciences, Peking University, China, in 2007 and 2012, respectively. He was a postdoctoral researcher fellow with Beijing International Center for Mathematical Research (BICMR) from 2012 to 2014. After that, he joined the School of Mathematics and Statistics, Central South University, China, where he currently is a lecturer. His research interests include cryptography and information security, especially in elliptic curve cryptography.

HWAJEONG SEO received the B.S.E.E., M.S. and Ph.D degrees in Computer Engineering at Pusan National University. He is currently an assistant professor in Hansung university. His research interests include Internet of Things and information security.