



An Efficient Implementation of AES on 8-Bit AVR-Based Sensor Nodes

YoungBeom Kim and Seog Chung Seo^(✉) 

Department of Information Security, Cryptology, and Mathematics,
Kookmin University, Seoul, South Korea
{darania,scseo}@kookmin.ac.kr

Abstract. Recently, FACE-LIGHT was proposed on 8-bit AVR MCUs for fast AES encryption. FACE-LIGHT is an extended version of Fast AES-CTR mode Encryption (FACE) method which was firstly proposed for high-end processors and it is tailored for performance on 8-bit AVR MCUs. Even though it has achieved high performance, it has to suffer from the overhead caused by table generation. Thus, when the number of blocks is less than a certain number, the table generation overhead is greater than the gains from using the generated table in the process of encryption. In other words, FACE-LIGHT needs to generate new tables whenever the Initial Vector (IV) is changed. Thus, frequent table regeneration results in a significant performance degradation. In this paper, we present an efficient implementation of AES block cipher on 8-bit AVR Microcontrollers (MCUs). Our method combines ShiftRows, SubBytes, and MixColumns operations into one with column-wise fashion and makes full use of registers of AVR MCUs for high performance. With handcrafted assembly codes, our implementation has achieved 2,251, 2,706, and 3,160 clock cycles for 128-bit, 192-bit, and 256-bit security, respectively. Our implementation outperforms FACE-LIGHT with respect to overall performance including table generation and block encryptions until around 1,850 blocks (resp. 15,000 blocks) for 128-bit (resp. 192-bit) security. With respect to 256-bit security, our implementation always outperforms FACE-LIGHT without considering the table generation time. Our implementation operates in constant time and can be used for not only CTR mode, but also CBC mode differently from FACE-LIGHT.

Keywords: AES · Optimization · 8-bit AVR · Atmega · Embedded · WSN · Sensor node · Counter mode · CTR mode

1 Introduction

With development of Internet of Things (IoT) technology, Wireless Sensor Networks (WSNs) have been widely used for various applications such as smart

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494).

factory, smart farm, military surveillance systems, and home automation systems. In WSNs, sensor nodes collect information at where they are deployed and transmit it to the server with wireless communication. Thus, the data needs to be encrypted in order to provide confidentiality of the sensitive data. However, applying encryption to transmitted data is challenging on WSNs because the sensor nodes are very resource-constrained. For example, MICAz mote is a popular sensor node in WSNs and it is equipped with 8-bit AVR Atmega128 microcontroller clocked at 7.3728 MHz and it has only 4 KB SRAM. Atmega328p, our target MCU, is also widely used for various types of sensor nodes.

AES cipher is widely used in several communication standards for data confidentiality in WSNs. Until now, many researchers have presented efficient implementations of AES on 8-bit AVR MCUs. Among them, recently FACE-LIGHT was proposed for fast AES CTR encryption on 8-bit AVR MCUs [11]. FACE-LIGHT is an extended version of FACE(Fast AES-CTR mode Encryption) [3], originally proposed for high-end processors. Its main idea is to build Lookup tables of 4 KB containing all possible values until Round 2, and to utilize them for fast computation of the first two rounds. Actually, it can encrypt a 128-bit data block in 2,218 clock cycles (*cc*) with 128-bit security. Although it has achieved high performance, it has to suffer from the overhead caused by Lookup table generation. Actually, FACE-LIGHT needs to rebuild the Lookup tables whenever the nonce value in the IV(initialization vector) changes (In our estimation, it requires around 61,440 *cc* for generating Lookup tables). Therefore, FACE-LIGHT is not suitable for environments where the communication session is frequently refreshed. In WSNs, the communication session can be frequently refreshed because the sensor nodes are usually deployed on harsh environments.

In this paper, we present an efficient implementation of AES on 8-bit AVR MCUs for data confidentiality in WSNs. Our method integrates ShiftRows, Sub-Bytes, and MixColumns operations into one for fast round computation and makes full use of registers in the target MCU. Differently from FACE-LIGHT using 4 KB Lookup tables affected by IV value, our implementation utilizes constant Lookup tables of just 512 Bytes which is affordable in SRAM of 8-bit AVR MCUs. Furthermore, the table Lookup operations are optimized for high performance in our implementation. Therefore, with handcrafted assembly codes, our implementation can encrypt a 128-bit data block in 2,251 (resp. 2,706 and 3,160) *cc* with 128-bit (resp. 192-bit and 256-bit) security. Our implementation outperforms all existing AES implementations including FACE-LIGHT on 8-bit AVR with respect to overall performance including Lookup table generation and block encryption. In detail, our implementation provides improved performance compared with FACE-LIGHT until encrypting 1,850 blocks (resp. 15,000 blocks) for 128-bit (resp. 192-bit) security. Notably, our implementation always outperforms FACE-LIGHT without considering Lookup table generation time for 256-bit security. Our implementation executes in constant time, which provides the resistance against timing attack and it can be efficient executed with not only CTR mode, but also CBC mode differently from FACE-LIGHT.

Table 1. Notations

Symbol	Meaning
<i>state</i>	Internal State composed of 16 bytes (typically 4 by 4 matrix)
<i>Nb</i>	Number of Columns (32-bit words) comprising the <i>state</i>
<i>Nk</i>	Number of 32-bit words comprising the <i>state</i>
<i>Nr</i>	Number of rounds depending on key length
Subbytes()	Transformation using a non-linear byte substitution table (SBOX-Table)
ShiftRows()	Transformation cyclically Shifting the last three rows of the <i>state</i> by different offsets
MixColumns()	Transformation taking all of the columns of the <i>state</i> and mixing their data to produce new columns
AddRoundKey()	Transformation adding RoundKey to the <i>state</i> with XOR operation
KeyExpansion()	Routine generating a series of Round Keys from a master key

2 Overview of AES Block Cipher and 8-Bit AVR MCUs

2.1 Overview of AES Block Cipher

AES is the most widely used 128-bit block cipher providing 128, 192, and 256-bit security. Table 1 defines notations used in this paper. Each round of AES encryption except for the last round is composed of **Subbytes()**, **ShiftRows()**, **MixColumns()**, and **AddRoundKey()** functions. **MixColumns()** function is not executed in the last round. The number of rounds in the AES depends on the length of the key initially entered [1]. AddRoundKey updates the *state* by XORing the *state* with round key. Subbytes provides Confusion effect, and ShiftRows and MixColumns cause Diffusion effect during AES operation. SubBytes is a single substitution operation for bytes. ShiftRows rotates the last three rows of the *state*. Equation 1 is MixColumns process mixing bytes in each columns. In the Eq. 1, each byte in a column is updated with Eq. 2.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad (1)$$

$$\begin{aligned}
S'_{0,c} &= (02 \cdot S_{0,c}) \oplus (03 \cdot S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= S_{0,c} \oplus (02 \cdot S_{1,c}) \oplus (03 \cdot S_{2,c}) \oplus S_{3,c} \\
S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (02 \cdot S_{2,c}) \oplus (03 \cdot S_{3,c}) \\
S'_{3,c} &= (03 \cdot S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (02 \cdot S_{3,c})
\end{aligned} \tag{2}$$

In the above equation, $(02 \cdot S_{i,c})$ can be computed as a multiplication in $GF(2^8)$ with an irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. $(03 \cdot S_{i,c})$ can be executed with a multiplication in $GF(2^8)$ and additional addition in $GF(2^8)$. Since all bytes in the *state* need to be updated with following Eq. 2, MixColumns causes the most computational load in AES.

2.2 Overview of 8-Bit AVR MCUs

Table 2. AVR Assembly Instructions

Instruction	Operands	Operation	Clocks
MOV	Rd, Rr	Copy register from Rr to Rd , $Rd \leftarrow Rr$	1
LD	Rd, X	Load memory data from X to Rd , $Rd \leftarrow (X)$	2
LPM	Rd, Z	Load program memory from Z to Rd , $Rd \leftarrow (Z)$	3
ST	Rr, X	Store memory from Rr to X , $(X) \leftarrow Rr$	2
EOR	Rd, Rr	Exclusive-OR two register, $Rd \leftarrow Rd \oplus Rr$	1
LSL	Rd	Logical shift left, $Rd(n) \leftarrow Rd(n)$, $Rd(0) \leftarrow 0$	1
BRCC	k	Branch if Carry cleared, if $(C = 0)$ then $PC \leftarrow PC + k + 1$	1/2
SET	<i>None</i>	Set T in SREG, $T \leftarrow 1$	1
RJMP	k	Relative jump, $PC \leftarrow PC + k + 1$	2

8-bit AVR is the most widely used embedded MCU for low-priced embedded devices including sensor nodes in WSNs. The AVR's commands consist of operation codes and operand, which have more than 130 commands. operand can use registers, memory, and constant values as targets for the command, and sometimes, according to the command, it is included in the command code without an operation [5]. All AVR commands require less than 4 clock cycles to execute [7]. Table 2 shows operand and clock cycles of commands used in this paper [6]. Currently, there are various types of AVR microcontrollers, and they have various peripherals and memory sizes. AVR is divided into Flash memory, SRAM and EEPROM with a Harvard architecture. the AVR-MCU has 32 general-purpose registers with an 8-bit size, which are used for various roles, including basic private operation and bit operation. In particular, the R26 - R31 registers can be combined into two each and used as three 16-bit registers, X, Y, and Z registers. They are used as pointers to indirectly specify a 16-bit address for data memory. There is also a SREG(Status Register) that show the status and result after ALU calculation.

3 Analysis of Existing Implementations on 8-Bit AVR

3.1 Analysis of Otte et al.'s Implementation

In Otte et al.'s AES implementation [2], the *state* are stored in general-purpose registers of AVR to minimize memory access time. SBOX-Table is stored in Flash Memory and its value is loaded with LPM instruction. KeyExpansion is executed before beginning encryption process and the generated round keys are stored in memory. Otte et al.'s implementation requires 3 clock cycles to load a value in SBOX-Table located at Flash memory. Since Arduino UNO, the most popular 8-bit embedded device, has 2 KB of SRAM, SBOX-Table of 256-bytes is affordable in SRAM. Therefore, storing SBOX-Table in SRAM could achieve performance improvement because accessing a value in SRAM costs 2 clock cycles .

Algorithm 1. MixColumns of Otte et al.[2]

1: .irp row, in 0,1,2,3	17: EOR $T1, 1B$
2: MOV $r0, ST\backslash row\backslash()2$	18: 3: EOR $T1, r0$
3: EOR $r0, ST\backslash row\backslash()3$	19: EOR $ST\backslash row\backslash()1, T1$
4: MOV $T2, r0$	
	20: LSL $T2$
5: MOV $T0, ST\backslash row\backslash()0$	21: BRCC $3f$
6: EOR $ST\backslash row\backslash()0, ST\backslash row\backslash()1$	22: EOR $T2, 1B$
7: EOR $r0, ST\backslash row\backslash()0$	23: 3: EOR $T2, r0$
8: LSL $ST\backslash row\backslash()0$	24: EOR $ST\backslash row\backslash()2, T2$
9: BRCC $3f$	
10: EOR $ST\backslash row\backslash()0, 1B$	25: EOR $T0, ST\backslash row\backslash()3$
11: 3: EOR $ST\backslash row\backslash()0, r0$	26: LSL $T0$
12: EOR $ST\backslash row\backslash()0, T0$	27: BRCC $3f$
	28: EOR $T0, 1B$
13: MOV $T1, ST\backslash row\backslash()1$	29: 3: EOR $T0, r0$
14: EOR $T1, ST\backslash row\backslash()2$	30: EOR $ST\backslash row\backslash()3, T0$
15: LSL $T1$	31: .endr
16: BRCC $3f$	

Algorithm 1 is the MixColumns implementation of Otte et al.'s. In Algorithm 1, $STij(i, j \in [0, 3])$ is a register for maintaining the byte located at i -th row and j -th column in the *state*. Otte et al.'s implementation make full use of $Ti(i \in [0, 3])$ register in order to recycle the computed values without memory accessing. Otte et al.'s implementation requires branch instruction and label depending on the value of carry flag C in SREG to compute a multiplication in $GF(2^8)$. In other words, if the result of LSL instruction makes the carry flag set, the result needs to be XORed with $0x1b$. However, BRCC instruction requires a 2 clock cycles when the carry Flag is zero, and vice versa, 1 clock cycles occurs,

which results in non-constant timing execution. Therefore, Otte et al.'s implementation provides additional information to the attackers in view of timing side channel analysis.

3.2 Analysis of FACE-LIGHT

FACE-LIGHT is an optimized AES-CTR implementation on 8-bit AVR MCUs by constructing Lookup tables related to *IV* and recycling them for performance improvement [4]. Actually, FACE-LIGHT is an extended version of FACE [3] which was proposed for general purpose CPU environments. Although FACE is fast, however, it has the disadvantage of updating the Lookup tables every 256 encryption times. Furthermore, it requires 5 KB memory for storing Lookup tables. FACE-LIGHT reduces the size of Lookup tables into 4 KB and does not require to renew the Lookup tables during encryption process.

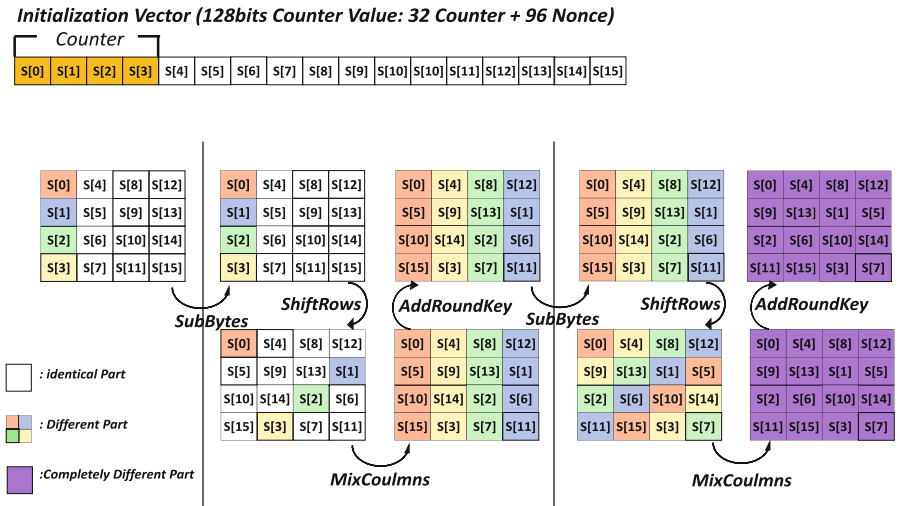


Fig. 1. Overview of FACE-LIGHT [4]

Figure 1 describes the process of FACE-LIGHT. It utilizes *IV* where first 4 bytes are used for counter value differently from the original CTR mode using last 4 bytes as the counter value. In Round 0, only operation is AddRoundKey. the plaintext and roundkey is XORed. Thus, as the counter value in *IV* increases, only 4 bytes of *state* changes. Round 1 performs SubBytes, ShiftRows, and MixColumns operations in the same way as a typical AES implementation. Through SubBytes operation, only 4 bytes differs from the previous block. Namely, the nonce parts in *IV* does not change (only counter parts change). This is the same as ShiftRows operation. However, in MixColumns operation, 8-bit data is mixed with 32-bit columns. Therefore, the counter value of *IV* is spread to the each

column through MixColumns operation in Round 1. For example, $S[0]$ affects $S[5]$, $S[10]$ and $S[15]$. The same applies to other counters $S[1]$, $S[2]$ and $S[3]$. In addition, there are only 256 cases for each counter. As the counter value of IV changes, the difference from the previous block in the Round 0 is at most 4 bytes, and the difference in the Round 1 is at most 16 bytes from the previous block. However, even though the change of counter values updates each column, this result can be saved as a Lookup table. SubBytes and ShiftRows operations in Round 2 can be also optimized because the diffusion process does not take place unlike Mixcolumns operation. The Lookup table used in FACE-LIGHT includes the results up to the SubBytes in Round 2, and ShiftRows can be omitted by assigning the results directly to the register.

Algorithm 2. Generation of Lookup table for $S[3]$ of IV in FACE-LIGHT

Require: byte *Counter*[4], byte *key*[4]

Ensure: byte *Lookuptable*[4][256]

```

1: byte state[4]
2: for  $i = 0$  to 255 do
3:    $state = Counter$ 
4:   MixColumns(state)
5:   AddRoundKey(Counter, key)
6:   Subbyte(state)
7:    $Lookuptable[4][i] = state$ 
8:    $Counter = Counter + 1$ 
9:    $i = i + 1$ 
10: end for
```

Among the IV's of FACE-LIGHT, $S[0]$, $S[1]$, $S[2]$, and $S[3]$ are counters and can have 256 values each. Algorithm 2 shows the creation of 256 byte tables for $S[3]$ in FACE-LIGHT. During initial 256 encryptions, only $S[3]$ changes, and during AddRoundKey and Subbytes of Round 0, the difference from the previous block is only 1 byte. The ShiftRows of 1 Round is implemented by direct means of bringing results, the actual necessary computations are MixColumns and AddRoundKey operation transformations, and the Subbytes of Round 2.

The FACE implementation requires updating some parts of Lookup tables after encryption 256 times. Furthermore, whole Lookup tables needs to be reconstructed whenever the IV is refreshed. In [3], the time for generating Lookup tables is almost same as time for encrypting 47 blocks on general purpose CPU environments. However, in view of cryptographic protocol, timing for generating Lookup tables needs to be included in the timing cost of whole encryption process. FACE-LIGHT basically follows the concept of FACE for optimizing the performance of AES-CTR on 8-bit AVR MCUs. Thus, it also requires huge timing for constructing Lookup tables of 4KB. In other words, FACE-LIGHT

needs to generate new Lookup tables whenever the IV is changed. Frequent regeneration of Lookup tables results in the significant performance degradation. Furthermore, 4 KB of Lookup tables needs to be located at the flash memory because it is larger than the size of SRAM in Arduino UNO which is the target of FACE-LIGHT [4]. Even though the source codes of FACE-LIGHT is open through github, the codes for generating Lookup tables are omitted [11]. Thus, we need to estimate the timing for Lookup table generation. In order to estimate the minimum timing of table generation, we have utilized the description of the table generation presented in FACE-LIGHT. In addition, from the open source codes of FACE-LIGHT, we have found out that FACT-LIGHT made use of Otte et al.'s implementation from Round 2 to the last Round. Therefore, the number of clock cycles for generating Lookup table is estimated based on the paper [4] and the implementation of Otte et al. [2]. Since our estimation is conservative (actually, we have excluded the timing for extra operations), it is likely that more clock cycles will be required if it is implemented in practice. Algorithm 2 shows the implementation of Look up table for S[3] of IV.

It takes 3 clocks cycles to perform LD and EOR commands for AddRoundKey operation of Round 0, and 4 clocks to perform MOV and LPM in the Subbytes operation of Round 1. In Otte et al.'s implementation, it takes 25 *cc* to calculate the each Columns [2]. The AddRoundKey in the Round 1 and Subbytes in the Round 2 require 28 clocks cycles, because all 4 bytes must be calculated. Therefore, a total of 15,360 *cc* $((3 + 4 + 25 + 28) * 256 = 15,360)$ are required to make the Lookup table[4][256] corresponding to S[3]. In an AVR environment, FACE-LIGHT has the advantage of having a table of 4KB and not renewing, which is done by creating a table for each counter. In other words, a total of 61,440 $(15360 * 4 = 61,440)$ is required to make a table for S[0], S[1], S[2], and S[3]. The 4KB Lookup table should be stored in the 32KB Flash memory in Atmega328P which is the target MCU of FACE-LIGHT. Since the Look up table is stored during the encryption process, the boot-loader system must be used to generate the Lookup table of FACE-LIGHT. Therefore, it is necessary to read the pages in Flash memory, then change the data, and rewrite the entire page area. Therefore, This incurs additional costs. Considering that both the Otte et al. based on AES-128 bit implementation and the FACE-LIGHT implementation require 2,600 *cc* or less, the table generation time for FACE-LIGHT requires a very large clock cycle. This is the same as the clock cycles that take about 24 encryption times.

Our estimation seems to be reasonable because the timing for Lookup table generation is almost same as encrypting 24 blocks on 8-bit AVR MCUs, which is less than that of FACE on conventional CPUs environments. Actually, 8-bit AVR MCUs have much less computing capabilities than conventional CPUs. Thus, it is more likely that Lookup table generation on 8-bit AVR MCUs requires more clock cycles than on conventional CPUs. However, since we have conservatively estimated the timing for Lookup table generation by omitting extra operations, our result of our estimation is less than that presented in FACE [3], which supports the rationale of our estimation.

In the next Section, we will present an efficient AES implementation techniques optimized on 8-bit AVR MCUs. Our implementation not only executes in constant time and but also provide fast encryption timing with only 512 bytes of constant Lookup table.

4 Proposed AES Implementation on 8-Bit AVR MCUs

4.1 Main Idea

Differently from implementation of Otte et al. and FACE-LIGHT, our implementation optimizes the process of encryption in a constant time manner. In addition, we propose a more generic AES implementation techniques which can be used for various mode of operations. Our main idea is to combine the transformations of SubBytes, ShiftRows, and Mixcolumns operations into one by column-wise fashion differently from row-wise implementations from the works of Otte et al. and FACE-LIGHT.

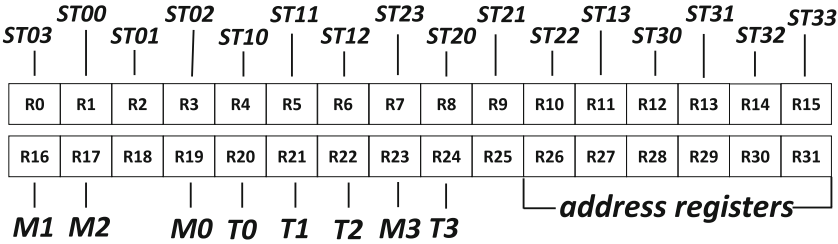


Fig. 2. Register scheduling for proposed implementation in the target AVR MCUs

In the row-wise implementation of Mixcolumns, it is effective to use branch instruction aforementioned in Sect. 3. However, this results in non-constant time execution. In addition, both works from Otte et al. and FACE-LIGHT locate SBOX-Table in Flash memory (Since the Lookup table in FACE-LIGHT is 4 KB it needs to be stored in Flash memory). However, in our implementation, we store SBOX-Table and MIX-Table in SRAM. We make use of MIX-Table containing the values of $(02 \cdot S_{i,c}, (i \in [0, 3])$ in Eq. 2) rather than computing it. Thus, our implementation requires only 512 bytes of Lookup tables having constant values. The 512 bytes of SBOX-Table and MIX-Table is constant and does not change for each encryption session. Furthermore, we minimize the number of Lookup tables accesses in the MixColumns operation, which will be described in detail with Sect. 4.2. With this approach, we can optimize the performance of the MixColumns operation with a constant time manner.

For efficiency, we maintain the *state* of the encryption process in 16 general-purpose registers in AVR MCUs. Figure 2 shows the register scheduling of our implementation. Through this scheduling, we can use assembly instruction such as MOVW that can be only applicable on even-numbered registers, which can reduce

by 1 clock cycle from 2 calls of MOV instructions. $R0-R15$ are used to maintain the values of the *state* in the encryption process. Thus, all *states* are updated in the register without memory accesses. Eight registers ($R16-R23$, alias as $M0, M1, M2, M3, T0, T1, T2$, and $T3$) are used as temporary registers to recycle the intermediate values in MixColumns. ($R26-R27$) keeps the address value of the generated round keys required for the AddRoundKey operation, ($R28-R29$) keeps the address of MIX-Table, and ($R30-R31$) keeps the address of SBOX-Table.

4.2 Proposed Register Scheduling and Implementation

$$\begin{aligned} \bar{S}_{i,c} &= SBOX(S_{i,c \ll i}) \text{ and } \tilde{S}_{i,c} = MIX(S_{i,c \ll i}), \text{ for } i \in [0, 3] \\ \begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} &= \underbrace{\begin{bmatrix} \tilde{S}_{0,c} \\ \tilde{S}_{0,c} \\ \tilde{S}_{0,c} \\ \tilde{S}_{0,c} \oplus \tilde{S}_{0,c} \end{bmatrix}}_A \oplus \underbrace{\begin{bmatrix} \tilde{S}_{1,c} \oplus \tilde{S}_{1,c} \\ \tilde{S}_{1,c} \\ \tilde{S}_{1,c} \\ \tilde{S}_{1,c} \end{bmatrix}}_B \oplus \underbrace{\begin{bmatrix} \tilde{S}_{2,c} \\ \tilde{S}_{2,c} \oplus \tilde{S}_{2,c} \\ \tilde{S}_{2,c} \\ \tilde{S}_{2,c} \end{bmatrix}}_C \oplus \underbrace{\begin{bmatrix} \tilde{S}_{3,c} \\ \tilde{S}_{3,c} \\ \tilde{S}_{3,c} \oplus \tilde{S}_{3,c} \\ \tilde{S}_{3,c} \end{bmatrix}}_D \quad (3) \end{aligned}$$

In our implementation, for efficiency, we combine SubBytes, ShiftRows, and MixColumns on a column-wise operation as shown in Eq. 3.

The c -th column of *state* is updated with Eq. 3. In other words, for computing four bytes of the c -th column, the results of A, B, C , and D parts in Eq. 3 need to be XORed. We maintain the intermediate result of each part in $M0, M1, M2$, and $M3$ registers. Namely, the value of the first row in each part is maintained at $M0$. Likewise, the second, the third, and the fourth bytes of each part are maintained in $M1, M2$, and $M3$ registers, respectively. For optimizing the usage of limited registers in AVR MCUs, we accumulate the results of each part in Eq. 3 in four registers $M0, M1, M2$, and $M3$. In other words, at first, the four bytes result from the computation of the part A are stored in $M0, M1, M2$, and $M3$. Then, when computing the part B, C , and D , each byte result is directly accumulated into one of $M0, M1, M2$, and $M3$. Finally, the accumulated bytes in $M0, M1, M2$, and $M3$ are stored in ST registers holding *state* bytes. Our implementation processes SubBytes operation with Table Lookup by accessing SBOX-Table in RAM. In other words, a $\tilde{S}_{i,c}$ corresponding to $S_{0,c}$ is retrieved from SBOX-Table. Similarly, $(02 \cdot S_{0,c})$ is efficiently computed with simple Table Lookup by accessing MIX-Table in RAM. $(03 \cdot S_{0,c})$ can be computed by XORing the result of SBOX-Table and the result of MIX-Table. For the efficiency of ShiftRows operation, we store the intermediate result of each part in Eq. 3 at the shifted position in the scheduled registers rather than actual shifting registers.

Algorithm 3 shows codes for computing the proposed Eq. 3 when $c = 0$ which is the combined operation for the first column of the *state*. The same logic can be used to compute columns of index $c \in 1, 3$. Note that the addresses for SBOX-Table and MIX-Table should be loaded Z and Y registers with LDI instruction.

Algorithm 3. Codes for proposed combined Subbytes, ShiftRows and Mix-Columns operations with equation 3 when $c = 0$

Part A Computation	Part C Computation
1: MOV $r30, ST00$	17: MOV $r30, ST22$
2: LD $M1, Z$	18: LD $T0, Z$
3: MOV $M2, M1 // M2 \leftarrow \bar{S}_{0,0}$	19: EOR $M0, T0 // M0 \leftarrow M0 \oplus \bar{S}_{2,0}$
4: MOV $M3, M2$	20: EOR $M1, T0$
5: MOV $r28, ST00$	21: EOR $M3, T0 // M3 \leftarrow M3 \oplus \bar{S}_{2,0}$
6: LD $M0, Y // M0 \leftarrow \tilde{S}_{0,0}$	22: MOV $r28, ST22$
7: EOR $M3, M0 // M3 \leftarrow \tilde{S}_{0,0} \oplus \bar{S}_{0,0}$	23: LD $T0, Y$
Part B Computation	Part D Computation
8: MOV $r30, ST11$	26: MOV $r30, ST33$
9: LD $T0, Z$	27: LD $T0, Z$
10: EOR $M0, T0$	28: EOR $M0, T0 // M0 \leftarrow M0 \oplus \bar{S}_{3,0}$
11: EOR $M2, T0 // M2 \leftarrow M2 \oplus \bar{S}_{1,0}$	29: EOR $M1, T0 // M1 \leftarrow M1 \oplus \bar{S}_{3,0}$
12: EOR $M3, T0 // M3 \leftarrow M3 \oplus \bar{S}_{1,0}$	30: EOR $M2, T0$
13: MOV $r28, ST11$	31: MOV $r28, ST33$
14: LD $T0, Y$	32: LD $T0, Y$
15: EOR $M1, T0 // M1 \leftarrow M1 \oplus \tilde{S}_{1,0}$	33: EOR $M3, T0 // M3 \leftarrow M3 \oplus \tilde{S}_{3,0}$
16: EOR $M0, T0 // M0 \leftarrow M0 \oplus \tilde{S}_{1,0} \oplus \bar{S}_{1,0}$	34: EOR $M2, T0 // M2 \leftarrow M2 \oplus \tilde{S}_{3,0} \oplus \bar{S}_{3,0}$

At the beginning of Algorithm 3, *state* registers $ST0c$, $ST1c$, $ST2c$, and $ST3c$ contain the value of $S_{0,c}$, $S_{1,c}$, $S_{2,c}$, and $S_{3,c}$ where $c \in 0, 3$. Lines 1–7 compute the part A of Eq. 3. Lines 1–2 load $ST00$'s corresponding SBox value ($\bar{S}_{0,0}$) from SBOX-Table and store it in $M1$ register. Likewise, lines 5–6 load $ST00$'s corresponding value ($\tilde{S}_{0,0}$) from MIX-Table and store it in $M0$ register. With line 7, the result of XOR operation with $M3$ and $M0$ is stored in $M3$. Through lines 1–7, $M0$, $M1$, $M2$, and $M3$ maintain the result of $\tilde{S}_{0,0}$, $\bar{S}_{0,0}$, $\bar{S}_{0,0}$, and $\tilde{S}_{0,0} \oplus \bar{S}_{0,0}$ which are the result of the part A in Eq. 3, respectively. Lines 8–16 accumulate the result of the part B to $M0$, $M1$, $M2$, and $M3$ registers. Note that we load $ST11$ rather than actually computing ShiftRows. The operation sequences of lines 8–16 (for computing the part B), lines 17–25 (for computing the part C), and lines 26–34 (for computing the part D) are identical except for the related registers. After finishing Algorithm 3, $M0$, $M1$, $M2$, and $M3$ registers contain $S'_{0,0}$, $S'_{1,0}$, $S'_{2,0}$, and $S'_{3,0}$, respectively.

The implementations in row-wise fashion, as shown in Eq. 2, calculate $(S'_{i,c}, (i \in [0, 3]))$ for each row, there are memory accesses for $S_{i,c}, (i \in [0, 3])$ for each calculation of rows. Also, non-constant time execution occurs by calculating in both $(02 \cdot S_{i,c})$ and $(03 \cdot S_{i,c})$ directly with branch instruction. The proposed column-wise implementation focuses on optimizing memory accesses for each $(S_{i,c}, (i \in [1, 3]))$ by reducing the duplicate memory accesses. Since our method requires only 512 bytes of Lookup Tables they can reside in SRAM, which

enables fast access compared with accessing data in flash memory. In addition, the proposed combined round transformations execute in constant-time, which makes the proposed implementation secure against timing attacks.

4.3 CTR Optimization

Our implementation is not limited to a specific mode of operation differently from FACE-LIGHT. In addition, its performance can be improved by using the characteristics of CTR mode like FACE and FACE-LIGHT. Similar to FACE-LIGHT, our implementation makes use of four bytes of the counter. Among 16 bytes of the *state*, 12 bytes are the same in the result of Round 0 (namely, AddRoundKey) of each block processing. Thus, we can store these 12 bytes as a Lookup table like [3]. When receiving a new nonce value, our implementation creates a Lookup table. The creation time for this Lookup table is negligible compared to generation time for Lookup table in FACE-LIGHT, and it contributes saving of clock cycles during Round 0.

Currently, 8-bit AVR MCUs are used for a variety of applications in WSNs (Wireless Sensor Networks) [8]. In WSN environments, since session-based cryptographic protocols require too much overhead for resource-constrained sensor nodes, broadcast-based communication protocols are mainly used. Therefore, using a long sequence of counter values is not easy for these protocols. In other words, encrypting many blocks sharing the same nonce value is not suitable for WSNs using 8-bit AVR MCUs [10]. In Sect. 3.2, we have shown that FACE-LIGHT requires a large number of clock cycles for creating a Lookup table. In environments where the nonce value is changed frequently, the overhead for creating a Lookup table surpasses the benefit of using the generated Lookup table.

5 Performance Analysis

Table 3. Comparison of AES implementations on AVR in terms of clock cycles

Security	Dinu et al. [9]	Otte et al. [2]	FACE-Light [4]	This work	This work (CTR)
AES-128	2,835	2,507	2,218	2,289	2,251
AES-192	N/A	2,991	2,702	2,746	2,706
AES-256	N/A	3,473	3,184	3,209	3,160

In this section, we compare our implementation with other existing implementations including FACE-LIGHT [4] on 8-bit AVR MCUs. Atmega328p is the target MCU and it is the most popular AVR MCU for Arduino UNO and various sensor nodes in WSNs. It equips Flash memory of 32 KB, EEPROM of 1 KB and internal SRAM of 2 KB. The measurement environment is Atmel Studio 7 and the source codes were compiled with `-OS` option. Table 3 represents a comparison

of AES's clock cycles for each implementation. The implementation of both Otte et al. [2] and Dinu et al. [9] are AES-ECB, and the FACE-LIGHT implementation is AES-CTR. In this paper, we present an efficient AES implementation that can be used in various operating mode and AES-CTR with optimization of FACE in Round 0. Creation time of Lookup Table is not included in Table 3. Our implementation improves performance over implementation of both Dinu et al. and Otte et al. and the performance of our implementation increases more in the AES-CTR. In particular, in AES implementation for 256-bit security, our implementation has better performance than both Otte et al. and FACE-LIGHT. Our AES-CTR implementation is more efficient as the key bit length increases, because FACE-LIGHT optimizes only Round 0 and Round 1. However, in this work, we optimize AES with combining the three operations(SubBytes, ShiftRows, MixColumns) and apply to all Rounds except the Round 0 and 10 Round(resp. 12 and 14 Round) for 128-bit (resp. 192-bit and 256-bit) security. Thus, our implementation is effective rather than FACE-LIGHT as the number of encryption rounds increases.

Table 4. Running time comparison of AES implementations using 128-bit key (Timings are measured by clock cycles. *NB* means the number of blocks.)

NB	1	50	100	200	800	1,850
Otte et al. [2]	2,507 (-)	125,350 (-)	250,700 (-)	501,400 (-)	2,005,600 (-)	4,665,527 (-)
FACE-Light. [4]	63,658 (-2,439%)	172,340 (-37.49%)	283,240 (-12.98%)	505,040 (-0.73%)	1,835,840 (+8.46%)	4,189,138 (+10.2%)
This Work (CTR)	2,251 (+10.2%)	112,550 (+10.2%)	225,100 (+10.2%)	450,200 (+10.2%)	1,800,800 (+10.2%)	4,189,111 (+10.2%)

Table 4 is a performance table of AES-CTR according to the number of encryption blocks. The value of the non-sets changes in all sessions. In the cryptographic protocol, the nonce value should change for each session. The table creation time must be considered in order to use FACE-LIGHT. In Github, the FACE-LIGHT code for the AVR environment exists [11], however no code for Lookup table generation exists, thus we estimate it based on the FACE-LIGHT implementation in Sect. 3. The estimated cost of generating the Lookup table is 61,440 clock cycles. The creation time of the Lookup table of proposed implementation in this paper is omitted because it is much smaller than the memory access time omitted during the estimation of generating the Lookup table of FACE-LIGHT. Table 4 shows that our implementation requires a lower clock cycles than the FACE-LIGHT up to 1,850 blocks for 128-bit security. The performance of our implementation and FACE-LIGHT are compared with the performance of Otte et al.'s implementation. Thus, our implementation always provides around 10.2% improvement performance compared with Otte et al.'s implementation. However, the entire performance of FACE-LIGHT is lower than

Otte et al.'s implementation until encryption 200 blocks. Based on Table 3, our implementation outperforms FACE-LIGHT until encrypting 15,000 blocks for AES-192. Regarding AES-256, our implementation is always faster than FACE-LIGHT without including table generation time based on Table 3.

6 Concluding Remarks

Due to the development of IoT technology, research to develop optimized performance in a limited environment is required. This is true not only for the basic performance of IoT, but also for encryption that needs to be additionally performed to protect personal information. Previously, a implementation such as FACE-LIGHT that optimizes AES in an 8-bit AVR MCUs has been proposed. In this paper, we present an AES implementation in an 8-bit AVR MCUs that is more efficient than FACE-LIGHT. We integrated major operations in AES and implemented them to make the most of the registers on the target MCU. Unlike FACE-LIGHT, the memory size used for the Lookup tables is reduced and optimized for use in SRAM. In addition, by using assembly instructions, we have completed an efficient implementation that eliminates unnecessary operations. Through the proposed optimization implementation, we obtained 2,251, 2,706, and 3,160 clock cycles for the key lengths of AES, 128-bit, 192-bit, and 256-bit, respectively. These results show up to 10.2% performance improvement over Otte et al. In this paper, only the optimization implementation for AES in the 8-bit AVR MCU is presented, but we will study various lightweight block encryption algorithms in various MCU in the future. This study can be used in a MCU in a constrained environment used in IoT devices.

References

1. NIST.: Announcing the ADVANCED ENCRYPTION STANDARD(AES). <https://www.nist.gov/publications/advanced-encryption-standard-aes>
2. Otte, D., et al.: AVR-crypto-lib (2009). <http://www.das-labor.org/wiki/AVRCrypto-Lib/en>
3. Park, J.H., Lee, D.H.: FACE: Fast AES CTR mode encryption techniques based on the reuse of repetitive data. In: IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 469–499 (2018)
4. Kim, K., Choi, S., Kwon, H., Liu, Z., Seo, H.: FACE-LIGHT: fast AES-CTR mode encryption for Low-End microcontrollers. In: Seo, J.H. (ed.) ICISC 2019. LNCS, vol. 11975, pp. 102–114. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40921-0_6
5. Z., Liu, H., J., Seo, J., Großschädl, H., W., Kim.: Efficient implementation of NIST-compliant elliptic curve cryptography for 8-bit AVR-based sensor nodes. In: IEEE Transactions Information Forensics and Security, pp. 1385–1397 (2016)
6. Atmel.: AVR Instruction Set Manual. <http://ww1.microchip.com/downloads/en/-devicedoc/atmel-0856-avr-instruction-set-manual.pdf>
7. Seo, S.C., Seo, H.: Highly efficient implementation of NIST-compliant koblitz curve for 8-bit AVR-based sensor nodes. IEEE Access 6, 67637–67652 (2018)

8. Tawalbeh, H., Hashish, S., Tawalbeh, L., Aldairi, A.: Security in wireless sensor networks using lightweight cryptography. In: ISSN Information Assurance and Security, pp. 118–123 (2017)
9. Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Le Corre, Y., Perrin, L.: FELICS-fair evaluation of lightweight cryptographic systems. In: NIST Workshop on Lightweight Cryptography, vol. 128 (2015)
10. Boneh, D., Gentry, C., Waters, B.: Collusion resistant broadcast encryption with short ciphertexts and private keys. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 258–275. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_16
11. Kim, K.H.: FACE-LIGHT (2019). <https://github.com/solowal/DEVELOP/tree/master/Source%20Code>