

FACE-LIGHT: Fast AES CTR Mode Encryption for Low-end Microcontrollers

No Author Given

No Institute Given

Abstract. AES implementations on low-end microcontrollers have a relatively lower performance than other lightweight block ciphers. For this reason, optimized AES implementations for these microcontrollers should be considered. In this paper, we revisit the previous Fast AES CTR mode Encryption (FACE) method for high-end processors and tailored the method to the microcontrollers, which is FACE-LIGHT. We targeted the 32-bit counter mode and the encryption is performed in regular fashion. This optimized technique pre-computes the 2 Add-RoundKey, 2 Sub-Bytes, 2 Shift-Rows and 1 Mix-Columns operations in efficient way. The method is implemented on the representative low-end microcontrollers (i.e. 8-bit AVR). The execution timing of AES-CTR algorithm for 128-bit and 256-bit security levels achieved the 2218 and 3184 clock cycles, respectively. This is faster than previous works by 22% for 128-bit security level.

Keywords: AES, Software Implementation, Counter Mode

1 Introduction

In the hyper-connected era, the utilization of the Internet of Things (IoT) have been increasing than ever. At the same time, the interest of security for the IoT is growing. Information transmitted between IoT devices may contain private information, such as personal information and authentication information. Therefore, the communication between the devices must be performed in a secure channel. A number of encryption algorithms exist for such secure communications. The most widely used algorithm is a symmetric key encryption algorithm, of which AES block cipher is the most representative[1]. Attempts have been made to make the AES efficient and there are two methods to do so. The first is to improve the hardware design. There have been ways to improve the hardware design by introducing T-boxes to AES or using FPGAs to implement them in hardware[2]. This hardware approach guarantees fast operation, but there are limitations to it, which made it difficult to update the built-in cryptographic module when a new, improved implementation is released. Another way is to reduce the logic of the software. The most well known methods are SBOX with precomputation which replaces three main operations of the AES with a single lookup operation. In addition, the most notable work is bitslice method which is

faster than any previous implementation[3]. Like wise, there have been various attempts to make the AES more efficient.

Most IoT platforms are resource constrained devices, which have limited memory size and low computing power. In order to apply encryption, in such limited environment, research on lightweight cryptography, which is relatively simple and has short computation time, has been actively conducted. Typical lightweight encryption algorithms include LEA, HIGHT, SIMON, SPECK, and CHAM[4–7]. However, most of these lightweight cipher algorithms adopt the ARX (Addition, Rotation, XOR) structure, which has the disadvantage that it takes much additional time when applying the masking operation to prevent the side channel attack [8]. The aforementioned AES encryption algorithm is not considered as a lightweight encryption due to its relatively long computation time than other lightweight block ciphers. However, AES block cipher has been the worlds most used encryption algorithm for a long period of time, and is an international standard encryption algorithm with high security and generality. In addition, AES adopts the SPN (Substitution-Permutation-Network) structure, which requires relatively less time for masking computation than the ARX structure. If the AES can be optimized and implemented on a low-end processor, it can be used as a lightweight cipher with high security and generality. The detailed research contributions are as follows.

1.1 Research Contributions

- **Efficient AES-CTR Implementation for Low-end Microcontrollers** In contrast to most of previous AES-CTR implementations, which focused mostly on the high-end processors, this paper concretes on low-end microcontrollers. We revisited the previous works and tailored the method to fit into the low-end environments. The new method is FACE-LIGHT. This method achieved the 2,218, 2,702, and 3,184 clock cycles for AES-128, AES-192, and AES-256, respectively.
- **Extended Round than FACE** Compared with previous FACE method, the FACE-LIGHT can further cache one Sub-Bytes and one Shift-Rows computations from round 2. This can cache the round up-to half of 3 round. It is particularly efficient over 8-bit microcontrollers and could reduce the clock cycle by 1,967, 2,449, and 2,931, for 128-bit, 192-bit, and 256-bit, respectively.
- **Regular Computation** The FACE method needs to re-calculate the cache table in every 256-times of encryption. The proposed method doesn't need to perform table update from first to last encryptions.
- **Optimized Masked AES Implementation** For secure against side channel attack, we present the optimized masked AES implementations on 8-bit AVR. The masked AES takes an advantages of FACE-LIGHT to get fast execution timing.

This paper is organized as follows. In Section 2, we describes the AES, AES-CTR and first-order masking, as well as the relevant notations. Section 3 discusses the previous AES implementations and Fast AES CTR mode Encryption

(FACE) technique, which is the fastest AES-CTR implementation method. In Section 4, we introduce the proposed optimization technique. In Section 5, we evaluate the performance of the proposed implementation. Section 6 concludes this paper.

2 Preliminaries

2.1 Notation

We use the notation described in Table 1 throughout the paper. We denoted $S[i]$ as the input or output State of the j^{th} round of the i^{th} block. $M[i]$ denotes the expendable masking value used to cope with power analysis attack.

Table 1: Notation

Notation	Description
$S[i]$	k^{th} byte of input State
$M[i]$	Masking value

2.2 AES

The AES algorithm is the most widely used symmetric key encryption algorithm in the world. It has a fixed 128bit block size and is divided into 128bit, 192bit, and 256bit according to the length of the key. In each case, the round key is used 10, 12 and 14 times, and Add-RoundKey, Sub-Bytes, Shift-Rows and Mix-Columns operations are repeated to generate a ciphertext. The Add-RoundKey operation performs a XOR operation on round key. It only affects one byte per operation without affecting other bytes. The Sub-Bytes operation performs substitution operation by referring to SBOX table saved in advance. It does not affect any other bytes as the Add-RoundKey. The Shift-Rows operation changes the index of bytes. Lastly, during the Mix-Columns operation, a diffusion is applied by mixing the bytes in each column.

2.3 AES-CTR

The block cipher is utilized with the specific mode of operation for target applications. Among the modes, counter modes (CTR) is most widely used in practice. The counter value is combined with the nonce and used as the initialization vector (IV) of the block cipher. With the IV, some deterministic value is generated from block cipher, and XOR operation is performed on the plaintext to encrypt the data. The encryption or decryption of the block cipher does not depend on the previous block. Based on this nice feature, the encryption and decryption operations can be performed in parallel way which can improve the performance of block cipher. The detailed descriptions of AES-CTR encryption and decryption are given in Figure 1 and Figure 2.

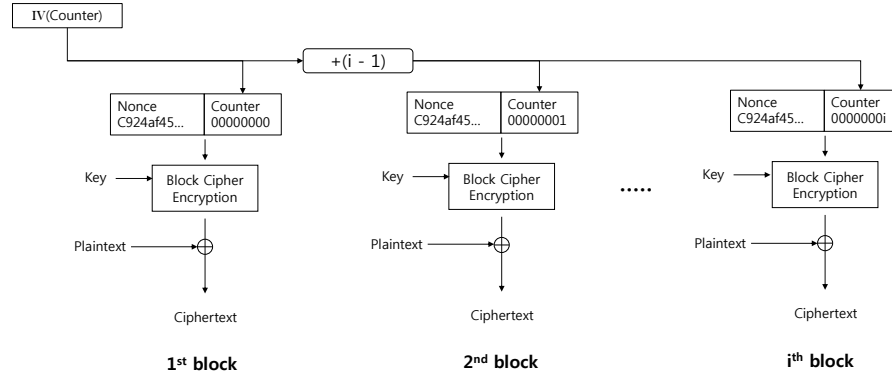


Fig. 1: Process of AES-CTR Encryption

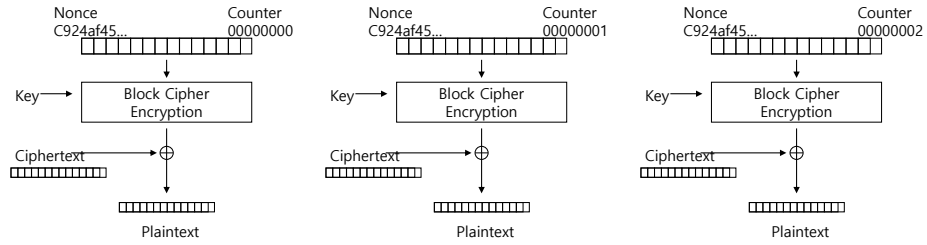


Fig. 2: Process of AES-CTR Decryption

2.4 First-order masking

The AES is divided into 128bit, 192bit and 256bit depending on the length of the key. The AES calculates the round key to be used for 10, 12 and 14 rounds each. Each round repeats the Add-RoundKey, Sub-Bytes, Shift-Rows and Mix-Columns operations to obtain the ciphertext. The following will describes AES-CTR-128 as a reference. In the encryption process of the AES algorithm, an attacker can analyze the data value of a specific operation using power analysis to find key values with high correlation coefficients. In order to cope with such attack, applying a masking value consisting of arbitrary values to each encryption operation can interfere with power analysis, making the process safe from CPA attack[9].

Figure 3 shows the masked AES algorithm. In the beginning of the AES, six random values are used to create the variables M0, M1, M2, M3, M4 and M5. After that, the algorithm use M2, M3, M4 and M5 for the factor of the Mix-Columns operation to use the output value as M6, M7, M8 and M9. The Mix-Columns operation is performed on the masking value in order to change M2, M3, M4 and M5, masked values in the ciphertext, into M6, M7, M8 and M9. The Sub-Bytes operation then masks SBOX. The 10 mask values and the masked

SBOX table are used to perform a masking operation on a specific operation for each round.

Before starting the first round, an XOR operation is performed on the plaintext/ciphertext of the (4×4) matrix, with the first row with M6, the second row with M7, the third row with M8 and the last row with M9. Then, from the first round, XOR operation is performed on all ciphertexts with M0 value in Add-RoundKey operation. Next, in order to remove the M6, M7, M8 and M9, XORs the same value of M6, M7, M8 and M9 which was used previously. In this process, if XOR operation was performed on M6, M7, M8 and M9 first, the vulnerability that temporarily exposes the ciphertext may occur. Therefore, it is safe to perform XOR operation after performing XOR operation with M0 value. After the process, the M0 value which masked the ciphertext gets removed while Sub-Bytes operation is performed which utilize the masked SBOX table. The ciphertext then gets masked with M1. In the Sub-Bytes operation, no masking operation is performed. Before the Mix-Columns operation, the first row of ciphertext gets masked with M2, the second row with M3, the third row with M4 and the last row with M5 values and previously masked value M1 gets removed. Lastly, during the Mix-Columns operation, the M2, M3, M4 and M5 are changed to M6, M7, M8 and M9. Through this procedure, the operations of Add-RoundKey, Sub-Bytes, Shift-Rows and Mix-Columns are repeated for ten rounds. After ten rounds, the ciphertext is masked with M6, M7, M8 and M9. In the last round, after the Add-RoundKey, Sub-Bytes and Shift-Rows operations are completed, the M1 value should be applied to the ciphertext as the masking value. Lastly, the last Add-RoundKey operation, which does not include the M6, M7, M8 and M9 masking operations, can be used to remove the M1 value and obtain the ciphertext from which all masking operations have been removed. As a result, all the masking value disappears through performing XOR operation on the same value. This results in the same result as AES without masking, but adds additional steps to the AES calculation process, which prevents power consumption measurement.

3 FACE: Fast AES CTR mode Encryption

In CHES'18, the efficient AES-CTR implementation (i.e. FACE) for high-end processor was suggested[10]. The FACE method utilizes the value of IV depending only on the change of counter values. Since the IV value, except for the counter value, remains the same as the following blocks, an identical pattern is repeated in specific part of the encrypted value until the Round 2 of AES. By utilizing this feature, repeated values can be stored in the cache table and used, which minimizes the encryption operation of subsequent blocks during the encryption operation, thereby effectively reducing the encryption operation time. This is done in five steps.

The first step is the FACE_{rd0} . In this step the FACE utilizes the fact that in Round 0, only the Add-RoundKey operation is performed. In the case of Add-RoundKey operation, a byte calculation is not affected by other bytes since it is

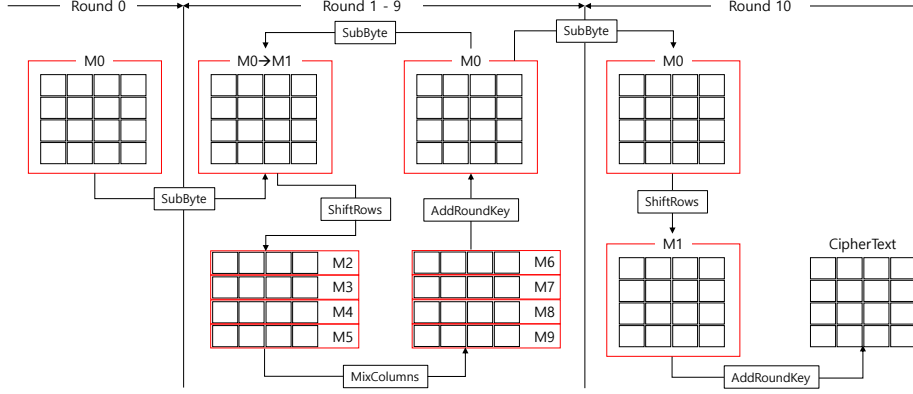


Fig. 3: Process of masked AES algorithm.

a XOR operation which only deals with single bytes independently. Therefore, as seen in following equation, the only byte difference between the first IV block and the second IV block is the last byte that is used as a counter. The Add-RoundKey operation can be minimized by storing the previous 12 bytes out of 16 bytes in the precomputed table.

$$\begin{aligned}
 IV_0 &: 0x00000000 \ 0x00000000 \ 0x00000000 \ 0x00000000 \\
 IV_1 &: 0x00000000 \ 0x00000000 \ 0x00000000 \ 0x00000001 \\
 &\dots \\
 IV_{255} &: 0x00000000 \ 0x00000000 \ 0x00000000 \ 0x000000FF
 \end{aligned}$$

The table is only replaced after a 2^{32-1} block operation where all unused 4 bytes are 0xFFFFFFFF. This approach requires only one cache update while processing 65.5GB of plaintext.

The description of the step is shown in Figure 4 and the Add-RoundKey result values, the State, can be seen stored in the cache. The cache consists of 4x4 bytes in total. Among the cache, only the values in the 0, 1 and 2 columns are reused. The values in the third column are not reused to minimize the cache update.

The second step is $FACE_{rd1}$. The State value from Round 0 is used as the input value of Round 1, where $Face_{rd1}$ reuses some of the values stored in the cache.

In Figure 5, it is shown that after the Round 1, the value of S[15], which was the only different byte from previous block, affects the whole first column through the Mix-Columns operation. The remaining columns except the first can be reused since they are not affected by S[15]. The reusable column values can be used until the value of S[15] exceeds 0xFF and affects S[14], which can be used up to 256 times.

The third step is $FACE_{rd1+}$. In this step, it is suggested that the value of the first column, which gets changed in the step $FACE_{rd1}$, be created as a cache

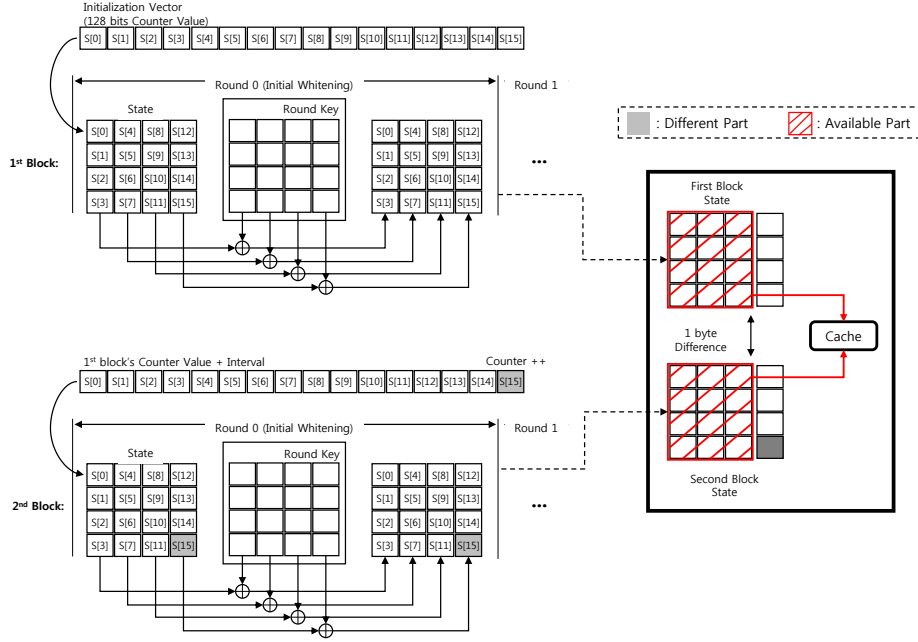


Fig. 4: Initial whitening of the first and the second block in CTR mode

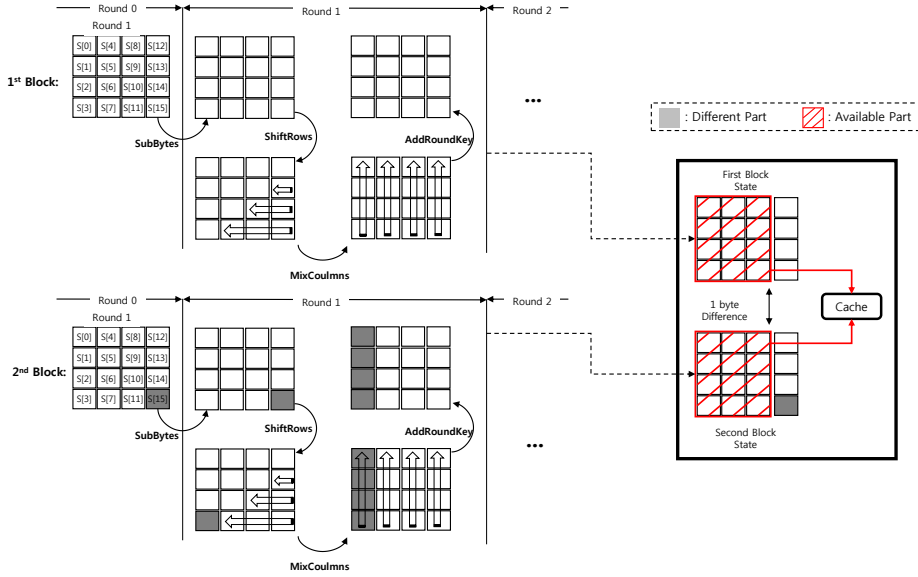


Fig. 5: Round 1 and the difference between the first and the second block

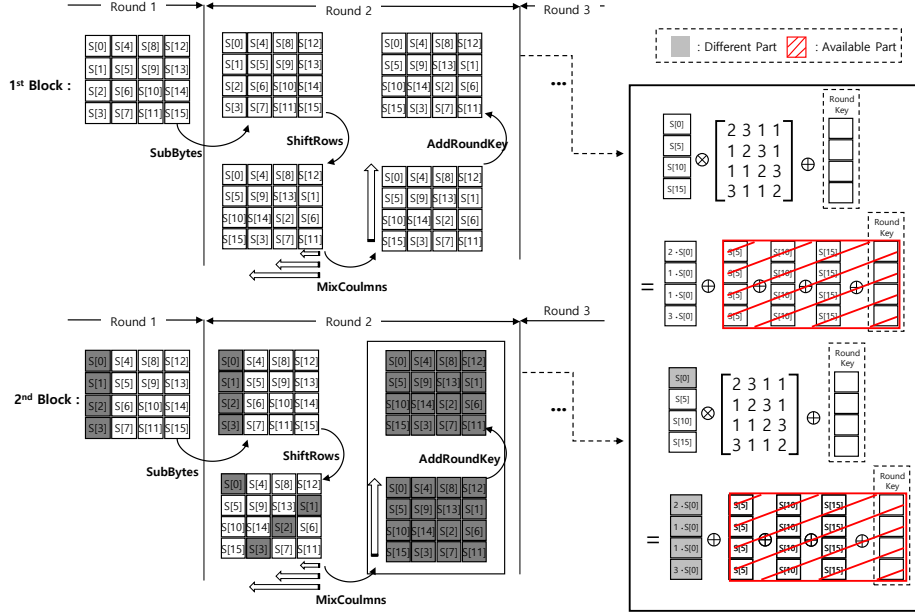


Fig. 6: Round 2 and the difference between the first and the second block

table through a precomputation. The value in the first column consists of $S[0]$, $S[5]$, $S[10]$ and $S[15]$, which get affected by $S[15]$ in the Mix-Columns operation. Therefore, $1KB(256 \times 4)$ of cache can be generated through a pre-calculation based on the $S[15]$ value that changes according to the counter value. The cache table can be created beforehand based on $S[15]$ since the value of $S[10]$, the high byte of $S[15]$, is not affected until a total of $0xFFFFFFFF$ blocks are initialized from $S[15]$ to $S[11]$, the 1,099,511,627,776th block (16 TB), is calculated.

The fourth step is $FACE_{rd2}$. This step deals with Round 2 which utilizes the output of the Round 1. Through Round 1 operation, the value of the first column of Round 2 is affected by the changes of the counter value.

Figure 6 shows the Round 2 process of the first and second blocks. $S[0]$, $S[1]$, $S[2]$ and $S[3]$, which are affected by the counter value, are spread to other columns by the Sub-Bytes operation.

The spreaded values affect all 16 Bytes through Mix-Columns operation. In conclusion, the whole byte is affected by the counter value. However, during the Mix-Columns operation, some of the values can be reused. Shown in Figure 6, operation values except $S[0]$ can be reused such as $S[5]$, $S[10]$, $S[15]$ and round key. These values will not change and repeat until the IV $S[14]$ is not affected by the counter increment. Therefore, making these 16 Bytes of unaffected values into cache table, can be used during operation up to the 256^{th} block.

In the final step, $FACE_{rd2+}$, it is suggested to create and store 4KBytes (16×256) table which contains the changing operation values which were excluded

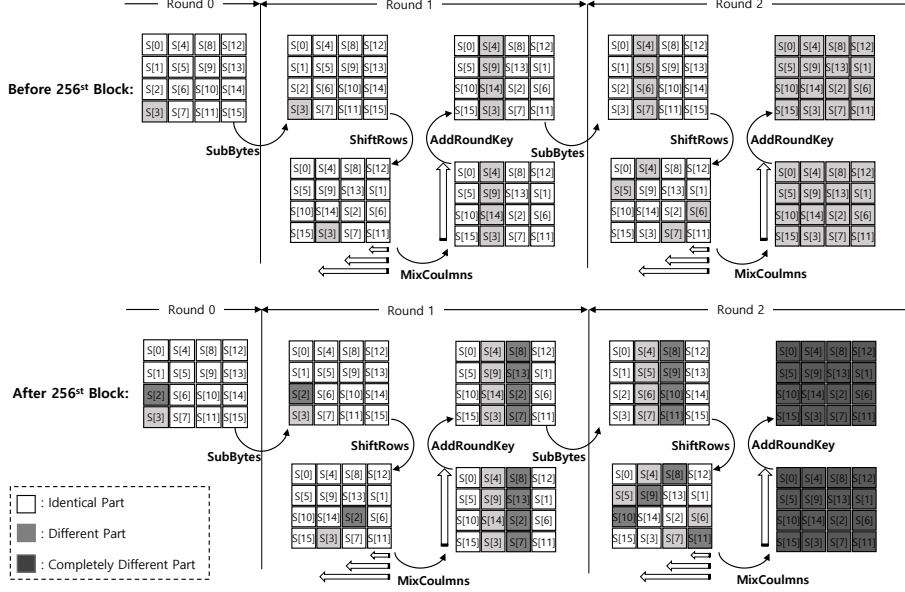


Fig. 7: Overview of FACE

from FACE_{rd2} and involves around $S[0]$, $S[1]$, $S[2]$ and $S[3]$. As in the case of FACE_{rd1+} , while calculating the 1,099,511,627,776th block (16 TB), a previously calculated table can be used.

The use of these lookup tables can provide significant advantages in cryptographic computation time, since it minimizes Round 0, Round 1 and Round 2 encryption operations in repeated blocks and uses the same precomputed values.

However, this work is only efficient for 8-bit counter mode. In Figure 7, if the last 8-bit counter value is 0 to 0xff, it can be used as a table except for the different parts painted in black. However, after 256 blocks, more values are affected as the value of $S[14]$ is changed as shown below. In this case, since we can not use pre-stored table, we need to update the cache table in each 256 times encryption. Since this frequent updates can be abused by attacker as an attack point (i.e. fault attack), we need to implement the FACE in regular form. In this paper, we revisit the FACE and optimize the method for low-end microcontrollers. The proposed FACE-LIGHT is regular fashion and we don't need to update the cache table throughout the whole AES-CTR life-cycle. The detailed descriptions of FACE are given in Figure 7.

4 FACE-LIGHT

In this section, we introduce the new implementation techniques for AES-CTR mode. Compared with previous work (i.e. FACE), we optimized the method for low-end microcontrollers. The CTR mode consists of nonce and counter values.

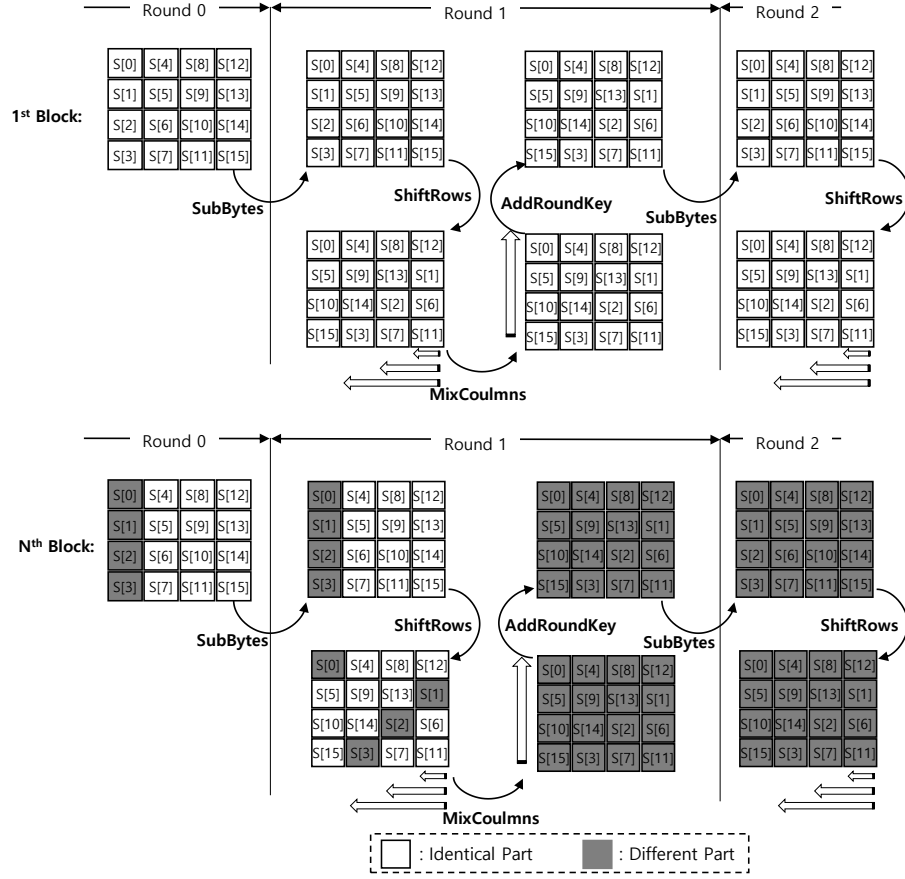


Fig. 8: Overview of FACE-LIGHT

For general setting, nonce and counter are set to 96-bit and 32-bit, respectively. The nonce is not changed throughout the whole sessions but the counter is changed in each transaction. Previous work focused on the low 8-bit and this needs to update the cache table in 256-times of encryption. In proposed work, we consider the 32-bit counter. We only need to set the pre-computed table in the initialization stage and it is not changed during the whole sessions. The detailed descriptions are given in Figure 8. The above figure indicates first block and the below figure indicates n -th block. Each square contains 8-bit data and the white and black colors represent identical and different parts, respectively.

Round 0 In Round 0, the only computation is Add-RoundKey. The plaintext and round key is xored. The only difference between two blocks is 4 bytes.

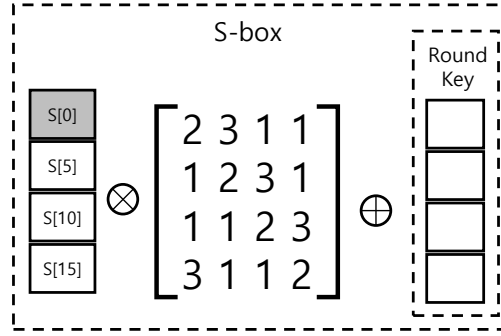


Fig. 9: Pre-computation for FACE-LIGHT

Round 1 In Round 1, Sub-Bytes, Shift-Rows, Mix-Columns, and Add-RoundKey operations are performed in order. The Sub-Bytes operation only changes the 8-bit input into the 8-bit output. This does not spread the values to the adjacent squares. The Shift-Rows tries to shift the square by certain offsets. In the Mix-Columns, the 8-bit data is mixed with 32-bit column. Finally, the Add-RoundKey adds the round keys to each square. Between first and N -th blocks, there is no common values. However, the both results are originated from IV. In particular, the low 32-bit value, namely counter value (i.e. $S[0]$, $S[1]$, $S[2]$, and $S[3]$), mainly contributes to the differences. In the Mix-Columns, $S[0]$ square influences to the first column (i.e. $S[5]$, $S[10]$, and $S[15]$). For this reason, each column (i.e. 32-bit) depends on the 8-bit square. Similarly, other columns are also based on the squares (i.e. $S[1]$, $S[2]$, and $S[3]$). This means each 32-bit column has only 256 cases. The following Add-RoundKey only applied to the each square so the pre-computation complexity is not changed. For this stage, the FACE performs in round 2 since they concern the 8-bit counter case with table update, while FACE LIGHT is 32-bit counter case without table update.

Round 2 In the previous stage, each column is based on the 8-bit square value. This is still maintained in Sub-Bytes operation which is only based on the each square value and the value is determined by 8-bit square of previous round. Since the Shift-Rows operation does not perform any mixing and changing on the value, the pre-computation is still working. For pre-computation, we need to keep 4 look-up tables. The input values are $S[0]$, $S[1]$, $S[2]$, and $S[3]$ in 8-bit wise. The length of output value is 32-bit wise. The total size for look-up table is 4 KB. In Figure 9, pre-computed table for FACE-LIGHT is described. In each table, we only receive the 8-bit input value and the value goes through the Mix-Columns, Add-RoundKey, and Sub-Bytes in this order. The Shift-Rows operation is not combined in the pre-computed table. It is optimized away by directly assigning the results to the specific squares.

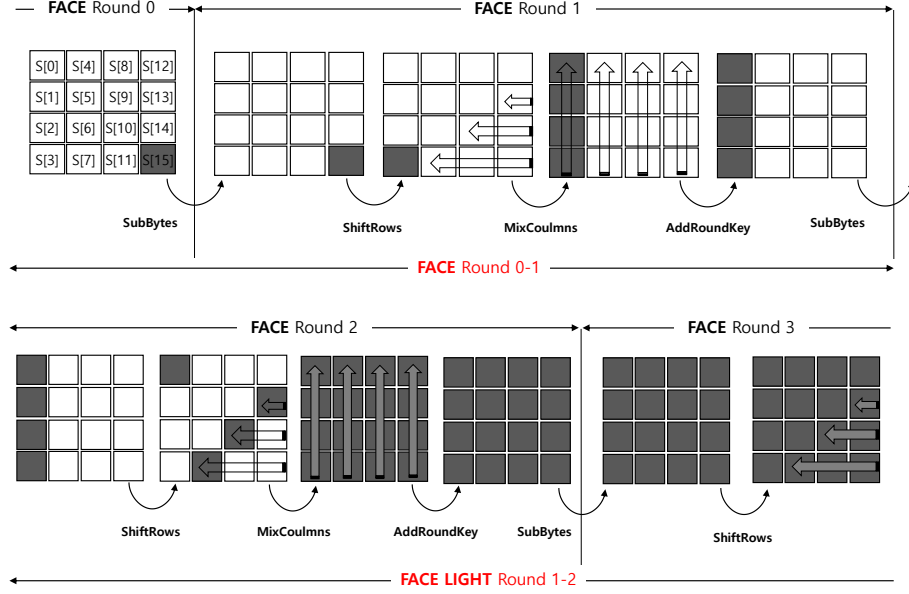


Fig. 10: Extended round for FACE round 3

Extended round for FACE The FACE-LIGHT is applied to the Round 2. This can be utilized for the FACE Round 3 since the different value is same with Round 2 of FACE-LIGHT. By using this technique, the Sub-Bytes and Add-RoundKey of Round 3 can be also cached. The detailed extended round for FACE are given in Figure 10.

Optimized Implementation The pre-computed table is stored before encryption operation. The 8-bit AVR microcontroller has very limited SRAM. For storing huge pre-computed table, we used the Program Memory. In each look-up table access, 32-bit results are extracted. For this reason, 8-bit input offset is extended to the 32-bit input offset by using quadrupling on the offset. Afterward, the input offset is added to the based address of each look-up table.

In AES algorithm, a substitution operation is performed during Sub-Bytes operation. During the operation, value of 256 Bytes to be substituted is stored in memory as a table called SBOX in advance to shorten the computation time. In Sub-Bytes operation, the values which to be substituted are loaded based on the input ciphertext. Therefore, when the value stored in the SBOX is loaded using the Z pointer (R30, R31), the lower memory address and the upper memory address of the SBOX are loaded in R30 and R31, respectively, and the ciphertext value to be used as the index value should be added to R30 to match the index. In this process, a carry might occur at the lower address of the SBOX memory loaded in R30, so that the value of R31 might needs to be increased by one. Therefore, if the ciphertext value is added to R30, the lower address of SBOX

memory, to refer to the SBOX memory, the ADC operation which performs carry operation, must be performed on R31, which stores the upper address of SBOX memory.

However, if the carry does not occur, the ADC is not only a meaningless operation, but it cause a big overhead since two operations(ADD, ADC) is performed for each Sub-Bytes operation for the index operation. To solve this problem, we utilize the memory size of the SBOX, 256Bytes. If the lower address of memory is set to 0x00, the carry value does not occur when indexing for memory access with the last index value of 0xff. Therefore, the ADC operation on R31, including the carry value can be omitted and the ciphertext can be loaded directly without performing the ADD operation on R30. In the same way, the computation time when storing or loading values in the MaskedSBOX table, a table created for masking, can also be reduced using the same address sorting method.

LD and ST instructions are used to load or store the key and the round key values stored in consecutive memory. In the case of the AVR, 16-bit addresses are accessed using the X(R26, R27), Y(R28, R29) and Z(R30, R31) pointers. Therefore, when accessing the repetitive memory or the peripheral memory, the memory address to be accessed should be set by using the ADD and ADC operators in the register that constitutes the pointer. In this case, carry value operation should be included as described above which causes a large overhead. Therefore, the address access without additional operator was performed by using LDD, STD and memory address post incremental instruction provided by AVR.

Masking Operation Optimization Masking operations aim to prevent the attacker from accurately measuring power usage by adding useless operations without affecting the encryption operations themselves. Therefore, the clock cycle is inevitably longer than the conventional encryption operation. In this paper, in order to minimize this disadvantage, we propose two methods.

The first method is as follows. Before performing encryption operation, 10 sets of round keys, from Round 0 to Round 9, pre-computed by fixed key encryption, should perform XOR operation with the value M0. Then carry out XOR operation on the first row with M6, the second row with M7, the third row with M8 and the last row with M9. In the case of Round 10, the last round, does not include M6, M7, M8 and M9 operations, but XORs the M1 value to all of the key values. If the corresponding operation is performed in advance, the XOR operation repeated in each round is minimized, and the memory in which the masking value is stored does not have to be loaded in each round. It can be observed in Algorithm 1 that M0 is XORed with M6, M7, M8, M9 in line 4. At label 1, M6, M7, M8 and M9 are XORed row by row for a total of 10 key sets from Round 0 to Round 9. By XORing M0 value to each mask value in advance, 160 XOR operations can be reduced. It is also shown in label 2, M1 XORs with the last round key.

The second method is as follows. Sub-Bytes operation is a relatively clock cycle consuming since it needs to access SBOX memory stored as a table. In addition, after performing Sub-Bytes operation, the masked value, M0, needs to

be replaced with M1. Therefore, overhead occurs when executing the memory load operation and XOR operation every round. In order to minimize the overhead, MaskedSBOX table should have been precomputed and be referenced in the Sub-Bytes operation. The detailed descriptions of the implementation are given in Algorithm 2.

Algorithm 1 Masking operation optimization

1: mov HI, ROUND	15: ld r0, Z	
2: lsl HI	16: eor r0, MASK8	28: ld r0, Z
3: lsl HI	17: st Z+, r0	29: eor r0, MASK1
		30: st Z+, r0
4: eor MASK6, MASK0	18: ld r0, Z	
5: eor MASK7, MASK0	19: eor r0, MASK9	31: ld r0, Z
6: eor MASK8, MASK0	20: st Z+, r0	32: eor r0, MASK1
7: eor MASK9, MASK0		33: st Z+, r0
	21: dec HI	
8: 1:	22: brne 1b	34: ld r0, Z
9: ld r0, Z		35: eor r0, MASK1
10: eor r0, MASK6	23: ldi HI, 4	36: st Z+, r0
11: st Z+, r0		
	24: 2:	37: dec HI
12: ld r0, Z	25: ld r0, Z	38: brne 2b
13: eor r0, MASK7	26: eor r0, MASK1	
14: st Z+, r0	27: st Z+, r0	

Algorithm 2 Generating Masked SBOX

1: ldi r31, hi8(MSBOX)	8: mov r30, r0
2: ldi r29, hi8(SBOX)	9: eor r30, T0
3: ldi xREDUCER, 0x1b	
	10: st Z, r26
4: 1:	
5: mov r28, r0	11: inc r0
6: ld r26, Y	12: brne 1b
7: eor r26, T1	

5 Evaluation

In this paper, we utilized the 8-bit AVR microcontoller. In particular, we used the Arduino UNO platform, which equips the ATmega328 processor. The hard-

ware follows the Harvard architecture and the working frequency is 16MHz. It contains 32 8-bit general purpose registers and has a total of 131 instruction sets. Flash memory is 32KB in size, with 1KB of EEPROM and 2KB of internal SRAM. We compile the code in -OS option and the performance is compared in terms of clock cycles. For performance measurement, the software was developed with Arduino IDE and Atmel Studio 7 on an Arduino UNO board with Atmega328p. All the functions in the program were implemented in Assembly except the loop function. The Arduino-UNO board's frequency was 16MHz and the results were obtained using the Arduino IDE and Atmel Studio 7 for accurate performance measurements. In addition, in order to confirm that the proposed AES is safe against power analysis attack, the power consumption during encryption operation is measured by Chipwhisperer-Lite (CW1173). In addition, CPA was performed by collecting 5000 waveforms in the third round with the output value of the Sub-Bytes operation as the middle value.

Table 2 shows the comparison of the clock cycles of four AES. Previous AES implementations are under ECB mode. The proposed method is CTR mode by using FACE-LIGHT method and extended FACE which FACE-LIGHT is applied. The main differences are mode of operation and their input values. In our setting, we used some special case. For this reason, FACE-LIGHT is faster than previous state-of-art by 617 clock cycles and Extended FACE is faster by 868 clock cycles. The proposed method efficiently passes the certain routines with pre-computed tables.

Table 2: Comparison of AES implementations on 8-bit AVR microcontroller in terms of clock cycles.

Security level	Dinu et al. [11]	Otte et al. [12]	FACE-LIGHT (This work)	Extended FACE (This work)
AES-128	2,835	2,507	2,218	1,967
AES-192	N/A	2,991	2,702	2,449
AES-256	N/A	3,473	3,184	2,931

In Table 3, the comparison results between FACE and FACE-LIGHT are given. The FACE-LIGHT does not require table update during computations. This means the encryption timing is always regular fashion. The timing information indicates the order of messages. By measuring the timing, we can get these information and this can be linked to the privacy issue. The target processor varies in each method. The FACE is for 32-bit processor while the FACE-LIGHT is for 8-bit microcontrollers. In addition, the FACE can be expanded till Round 2 while our work, FACE-LIGHT, can be expanded till Round 3.

Existing lightweight ciphers have also been studied for the implementation of additional masking techniques to cope with side channel attacks. However, most lightweight ciphers with ARX structures have significant overhead since the ciphers must undergo Arithmetic-to-Boolean operation during masking op-

Table 3: Comparison of FACE and FACE-LIGHT.

	FACE [10]	FACE-LIGHT (This work)
Table update	✓	–
Constant timing	–	✓
Target processor	32-bit or above	8-bit
Expandable Round	Round 2	Round 3

erations. On the other hand, the SPN structure, AES, has a relatively short operation time. Therefore it has an advantage over other lightweight ciphers in the masking operation, which is a side channel countermeasure.

Table 4: Comparison of LEA and optimized masked AES implementations on 8-bit AVR microcontroller in terms of clock cycles.

LEA-128 [13]	Masked LEA-128 [14]	Masked AES-128 (This work)
2,688	36,589	6,219

Table 4 shows the clock cycles of none masked LEA, masked LEA and AES encryption algorithm with masking operation. In the case of masked LEA-128, the encryption operation time increases rapidly when masking is applied to LEA. However, it can be observed that the proposed AES has robustness against power consumption analysis attack while having less computation time.

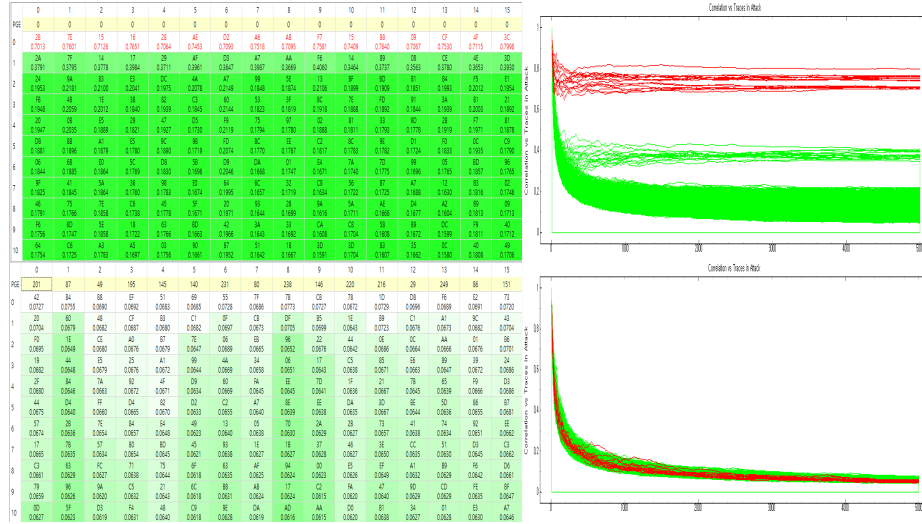


Fig. 11: Comparison of non Masked AES and Masked FACE-LIGHT on CPA attack

Figure 11 shows a graph of key value and correlation coefficient estimated by performing CPA on AES without mask and AES presented in this paper. In case of the AES without masking operation, the correlation coefficient of all key values is significantly higher than other values. However, in the case of the AES proposed in this paper, the attacker cannot guess the key value since all of the key values have equal correlation coefficients.

6 Conclusion

In this paper, we demonstrate the implementation of AES-CTR encryption algorithm software optimization on 8-bit AVR microcontrollers. The proposed FACE-LIGHT efficiently improves the performance for AES-CTR encryption, since we optimized the FACE to the 8-bit microcontrollers and 32-bit counter cases. We will apply this method to the AES-GCM, which consists of CTR and polynomial multiplication. The other consideration is for the other lightweight block ciphers. We will investigate this technique to extend to the other encryption modes or block ciphers.

References

1. N.-F. Standard, “Announcing the advanced encryption standard (AES),” *Federal Information Processing Standards Publication*, vol. 197, no. 1-51, pp. 3–3, 2001.
2. S. Morioka and A. Satoh, “A 10-gbps full-aes crypto design with a twisted bdd s-box architecture,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, pp. 686–691, 2004.
3. M. Matsui and J. Nakajima, “On the power of bitslice implementation on intel core2 processor,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 121–134, Springer, 2007.
4. D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, “LEA: A 128-bit block cipher for fast encryption on common processors,” in *International Workshop on Information Security Applications*, pp. 3–27, Springer, 2013.
5. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, *et al.*, “HIGHT: A new block cipher suitable for low-resource device,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 46–59, Springer, 2006.
6. R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, “The SIMON and SPECK lightweight block ciphers,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2015.
7. B. Koo, D. Roh, H. Kim, Y. Jung, D.-G. Lee, and D. Kwon, “CHAM: a family of lightweight block ciphers for resource-constrained devices,” in *International Conference on Information Security and Cryptology*, pp. 3–25, Springer, 2017.
8. L. Goubin, “A sound method for switching between boolean and arithmetic masking,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 3–15, Springer, 2001.
9. C. Herbst, E. Oswald, and S. Mangard, “An aes smart card implementation resistant to power analysis attacks,” in *International conference on applied cryptography and network security*, pp. 239–252, Springer, 2006.

10. 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.
11. D. Dinu, A. Biryukov, J. Großschädl, D. Khovratovich, Y. Le Corre, and L. Perrin, “FELICS—fair evaluation of lightweight cryptographic systems,” in *NIST Workshop on Lightweight Cryptography*, vol. 128, 2015.
12. D. Otte *et al.*, “AVR-crypto-lib,” *Online: <http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>*, 2009.
13. H. Seo, I. Jeong, J. Lee, and W. Kim, “Compact implementations of ARX-based block ciphers on IoT processors,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 3, p. 60, 2018.
14. E. Park, S. Oh, and J. Ha, “Masking-based block cipher LEA resistant to side channel attacks,” *Journal of the Korea Institute of Information Security and Cryptology*, vol. 27, no. 5, pp. 1023–1032, 2017.