mathematics

MDPI

Article

High-Speed Implementation of PRESENT on AVR Microcontroller

Hyeokdong Kwon [1], Young Beom Kim [2], Seog Chung Seo [2,3] and Hwajeong Seo [1,*]

[1] Division of IT Convergence Engineering, Hansung University, Seoul 136-792, Korea; hyeok@hansung.ac.kr
[2] Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; darania@kookmin.ac.kr (Y.B.K.); scseo@kookmin.ac.kr (S.C.S.)
[3] Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, Korea
* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-2-760-8033

Abstract: We propose the compact PRESENT on embedded processors. To obtain high-performance, PRESENT operations, including an add-round-key, a substitute layer and permutation layer operations are efficiently implemented on target embedded processors. Novel PRESENT implementations support the Electronic Code Book (ECB) and Counter (CTR). The implementation of CTR is improved by using the pre-computation for one substitute layer, two diffusion layer, and two add-round-key operations. Finally, compact PRESENT on target microcontrollers achieved 504.2, 488.2, 488.7, and 491.6 clock cycles per byte for PRESENT-ECB, 16-bit PRESENT-CTR (RAM-based implementation), 16-bit PRESENT-CTR (ROM-based implementation), and 32-bit PRESENT-CTR (ROM-based implementation) modes of operation, respectively. Compared with former implementation, the execution timing is improved by 62.6%, 63.8%, 63.7%, and 63.5% for PRESENT-ECB, 16-bit PRESENT-CTR (RAM based implementation), 16-bit PRESENT-CTR (ROM-based implementation), and 32-bit PRESENT-CTR (ROM-based implementation) modes of operation, respectively.

Keywords: PRESENT; counter mode of operation; AVR; software implementation

Citation: Kwon, H.; Kim, Y.B.; Seo, S.C.; Seo, H. High-Speed Implementation of PRESENT on AVR Microcontroller. *Mathematics* **2021**, *9*, 374. https://doi.org/doi:10.3390/math9040374

Academic Editor: Raúl M. Falcón

Received: 7 January 2021
Accepted: 4 February 2021
Published: 13 February 2021

1. Introduction

Lightweight cryptography is getting more important than ever due to the emergence of the Internet of Things. The lightweight cryptography supports encryption in resource-constrained environments, such as sensor network, health care, and surveillance systems. Therefore, the implementation of lightweight cryptography aims at optimizing certain criteria, such as energy consumption, execution time, memory footprint, and chip size.

We propose a number of implementation techniques for well-known lightweight cryptography, namely PRESENT, and its Electronic Code Book (ECB) and Counter (CTR) on low-end embedded processors, where ECB encrypts the plaintext directly with the master key and CTR encrypts the counter value with the master key and then the result of encryption is XORed with the plaintext. In order to achieve optimal results on target microcontrollers, we used processor-specific optimizations for PRESENT block ciphers. Furthermore, the compact counter mode of PRESENT and its bit-slicing-based implementation are also presented. Novel implementation techniques for PRESENT block cipher can be extended to other lightweight cryptography algorithms and other platforms.

1.1. Contribution

1.1.1. Optimal Implementation of PRESENT Block Cipher on Embedded Processors

We implemented the PRESENT block cipher on low-end microcontrollers. The Alf and Vegard's RISC (AVR) processor is a resource-constrained device that is used extensively in low-end Internet of Things (IoT) applications, such as `Arduino UNO` and `Arduino MEGA`. The PRESENT-ECB implementation is optimized in terms of execution timing and other factors (e.g., code size and RAM). The word size of general purpose registers in the target

AVR microcontroller is 8-bit wise. All 16-bit wise PRESENT operations are optimized for 8-bit word and instruction set. Compared with the former implementation of PRESENT-ECB for a 128-bit security level on AVR microcontrollers, the proposed work improved the execution timing by 62.6% [1].

1.1.2. Pre-Computation for PRESENT with CTR

CTR is utilized in real applications and services, such as Transport Layer Security (TLS) and Virtual Private Network (VPN). CTR receives the input consisting of two parts, including constant nonce and variable counter. Since the nonce part is the constant variable, the constant nonce value is repeated several times throughout computations. For this reason, some computations of PRESENT block cipher can be optimized through pre-computation. By exploiting this feature, we further improved the execution timing of PRESENT-CTR. The method is a generic algorithm and can be implemented with other processors. Compared with the state-of-art implementation,the proposed works on embedded processors that have obtained performance enhancements by 63.8%, 63.7%, and 63.5% for 16-bit PRESENT-CTR (RAM), 16-bit PRESENT-CTR (ROM), and 32-bit PRESENT-CTR (ROM), respectively.

1.1.3. Open Source

The proposed PRESENT implementation is a public domain and full source codes are available at https://github.com/solowal/PRESENT_AVR (accessed on 7 January 2021). Source codes were written in (mixed) AVR assembly language (core algorithm) and C language (function call). Codes support four 128-bit PRESENT implementations, including PRESENT-ECB, PRESENT-CTR16 (RAM based implementation), PRESENT-CTR16 (ROM based implementation), and PRESENT-CTR32 (ROM based implementation). Projects were created and evaluated with `Atmel Studio 7.0` framework. Researchers can evaluate and re-create the result with the available source codes.

2. Related Works

2.1. PRESENT Block Cipher

PRESENT block cipher was introduced in CHES'07 [2]. PRESENT block cipher supports two parameters (i.e., PRESENT-64/80 and PRESENT-64/128). PRESENT block cipher requires 31 rounds and the Substitution-Permutation-Network (SPN) structure is adopted. PRESENT requires three computations including the substitution layer, permutation layer, and add-round-key.

The add-round-key operation performs exclusive-or computations with plaintext and round keys. Round keys ($roundkey = (roundkey_1, roundkey_2, ..., roundkey_{32})$) are generated from the key schedule. In particular, $roundkey_{32}$ is used for post-whitening. PRESENT block cipher uses a 4-bit substitution layer. The inner state of PRESENT block cipher ($S_{63}, ..., S_0$) can be seen as 16 4-bit words (w_{15} ... w_0), where one w word consists of four states (i.e., $w_x = \{S_{4 \cdot x+3} \parallel S_{4 \cdot x+2} \parallel S_{4 \cdot x+1} \parallel S_{4 \cdot x}\}$, $0 \leq x \leq 15$). The 4-bit substitution layer can be represented in Boolean operations for the bitslicing implementation. The PRESENT 4-bit S-box is designed for higher hardware efficiency and compact implementation. PRESENT block cipher uses a bit of permutation for the linear diffusion layer. The permutation layer performs bit permutation in the intermediate result. Each bit state (x) is permutated through $P(x)$.

2.2. Target Processor

The AVR microcontroller finds many interesting applications in embedded systems, such as sensor networks, surveillance systems, and health care. The number of available registers is only 32 8-bit long. Basic arithmetic instructions take a single clock cycle. The memory load/store instruction requires two clock cycles. The microcontroller supports an 8-bit instruction set, 128 KB of `FLASH` memory, 8 MHz of working frequency, two-stage pipeline design, and 4 KB of RAM (e.g., ATmega128). Among them, 6 registers (i.e.,

R26∼R31) are reserved for address pointers, and the remaining registers can be utilized for general purpose registers by a programmer. In particular, the R1 register is the ZERO register that should be cleared before function returns.

2.3. Former Implementations on Low-End Embedded Processors

Several works optimized the LEA on embedded processors [3–7]. They optimized execution timing and memory consumption. There are many implementations of lightweight cryptography such as CHAM, SPECK, and SIMON [5,7–17].

Many works are also devoted to improve the execution timing of AES on embedded processors [18–22]. In [23], the compact implementation of ARIA on low-end microcontrollers was proposed.

In CHES'17, optimized PRESENT implementation on embedded ARM CPUs was presented by using a novel decomposition of permutation layers (see Listing 1.2 of [24]), and bitsliced for the S-boxes [24]. A description of PRESENT is detailed in Algorithm 2 of [24]. Unlike a traditional PRESENT algorithm, it performs the permutation layer before the substitution layer. This order of computation is beneficial for bit-slicing-based substitution layer implementation.

In this paper, we presented the compact PRESENT implementation on AVR microcontrollers. We re-designed the PRESENT implementation for 8-bit architecture. Then, we also suggested the PRESENT-CTR. The CTR implementation technique optimizes 2 add-round-key, 2 permutation, and 1 substitution operations with a 1 look-up table operation.

3. Proposed Method

3.1. Optimization of PRESENT–ECB

For the efficient implementation of PRESENT block cipher, add-round-key, substituion, and permutation layers are optimized.

In Algorithm 1, add-round-key operation is described in a source code level. The computation is performed with XOR operations with round keys where XOR operation represents logical bitwise exclusive-or operation. The memory access for round keys is performed with the incremental memory pointer mode.

Algorithm 1: Add-round-key operation in assembly language.

Input: Intermediate data (reg0∼7), round key pointer (X).

Output: Output results (reg0∼7).

```
1: LD tmp, X+
2: EOR reg0, tmp

3: LD tmp, X+
4: EOR reg1, tmp

5: LD tmp, X+
6: EOR reg2, tmp

7: LD tmp, X+
8: EOR reg3, tmp

9: LD tmp, X+
10: EOR reg4, tmp

11: LD tmp, X+
12: EOR reg5, tmp

13: LD tmp, X+
14: EOR reg6, tmp

15: LD tmp, X+
16: EOR reg7, tmp
```

The efficient implementation of permutation ($P0$) is described in Algorithm 2. A 16-bit wise rotation operations are performed with LSR, ROR, LSL, and ROL instructions. Exclusive-or and logical and operations are performed with EOR and ANDI instructions. Similar to the $P0$ operation, the permutation ($P1$) is implemented, efficiently.

Algorithm 2: Permutation (*P*0) operation in assembly language.

Input: Intermediate data (reg0∼7).

Output: Result (reg0∼7).

```
//t=(X0⊕(ROR_u16(X1,1)))&0x5555
 1: MOVW tmp0, reg4
 2: LSR tmp1
 3: ROR tmp0
 4: EOR tmp0, reg6
 5: EOR tmp1, reg7
 6: ANDI tmp0, 0X55
 7: ANDI tmp1, 0X55
//X0=X0⊕t; X1=X1⊕(ROL_u16(t,1));
 8: EOR reg6, tmp0
 9: EOR reg7, tmp1
10: LSL tmp0
11: ROL tmp1
12: EOR reg4, tmp0
13: EOR reg5, tmp1
//t=(X2⊕(ROR_u16(X3, 1)))&0x5555;
14: MOVW tmp0, reg0
15: LSR tmp1
16: ROR tmp0
17: EOR tmp0, reg2
18: EOR tmp1, reg3
19: ANDI tmp0, 0X55
20: ANDI tmp1, 0X55
//X2=X2⊕t; X3=X3⊕(ROL_u16(t, 1));
21: EOR reg2, tmp0
22: EOR reg3, tmp1
23: LSL tmp0
24: ROL tmp1
25: EOR reg0, tmp0
26: EOR reg1, tmp1
//t=(X0⊕(ROR_u16(X2, 2)))&0x3333;
27: MOVW tmp0, reg2
28: LSR tmp1
29: ROR tmp0
30: LSR tmp1
31: ROR tmp0
32: EOR tmp0, reg6
33: EOR tmp1, reg7
34: ANDI tmp0, 0X33
35: ANDI tmp1, 0X33
//X0=X0⊕t; X2=X2⊕(ROL_u16(t, 2));
36: EOR reg6, tmp0
37: EOR reg7, tmp1
38: LSL tmp0
39: ROL tmp1
40: LSL tmp0
41: ROL tmp1
42: EOR reg2, tmp0
43: EOR reg3, tmp1
//t=(X1⊕(ROR_u16(X3, 2)))&0x3333;
44: MOVW tmp0, reg0
45: LSR tmp1
46: ROR tmp0
47: LSR tmp1
48: ROR tmp0
49: EOR tmp0, reg4
50: EOR tmp1, reg5
51: ANDI tmp0, 0X33
52: ANDI tmp1, 0X33
//X1=X1⊕t; X3=X3⊕(ROL_u16(t, 2));
53: EOR reg4, tmp0
54: EOR reg5, tmp1
55: LSL tmp0
56: ROL tmp1
57: LSL tmp0
58: ROL tmp1
59: EOR reg0, tmp0
60: EOR reg1, tmp1
```

The bitslicing substitution operation is performed with Boolean operations. Detailed descriptions are given in Algorithm 3. Boolean operations, such as logical XOR, AND, OR, and one's complement are performed with `EOR`, `AND`, `OR`, and `COM` instructions. To move two adjacent registers in a single instruction, `MOVW` instruction is utilized.

Algorithm 3: Substitution operation in assembly language.

Input: Intermediate data (reg0∼7).

Output: Result (reg0∼7).

```
//T1=x2⊕x1;

 1: MOVW tmp0, reg2
 2: EOR tmp0, reg4
 3: EOR tmp1, reg5

//T2=x1&T1;

 4: MOVW tmp2, reg4
 5: AND tmp2, tmp0
 6: AND tmp3, tmp1

//T3=x0⊕T2;

 7: MOVW tmp4, reg6
 8: EOR tmp4, tmp2
 9: EOR tmp5, tmp3

//T5=x3⊕T3;

10: MOVW tmp7, reg0
11: EOR tmp7, tmp4
12: EOR tmp8, tmp5

//T2=T1&T3;

13: MOVW tmp2, tmp0
14: AND tmp2, tmp4
15: AND tmp3, tmp5

//T1=T1⊕T5;

16: EOR tmp0, tmp7
17: EOR tmp1, tmp8

//T2=T2⊕x1;

18: EOR tmp2, reg4
19: EOR tmp3, reg5

//T4=x3|T2;
20: MOVW tmp6, reg0
21: OR tmp6, tmp2
22: OR tmp6, tmp3

//x2=T1⊕T4;

23: MOVW reg2, tmp0
24: EOR reg2, tmp6
25: EOR reg3, tmp6

//x3=x3⊕0xFFFF;

26: COM reg0
27: COM reg1

//T2=T2⊕x3;

28: EOR tmp2, reg0
29: EOR tmp3, reg1

//x0=x2⊕T2;

30: MOVW reg6, reg2
31: EOR reg6, tmp2
32: EOR reg7, tmp3

//T2=T2|T1;

33: OR tmp2, tmp0
34: OR tmp3, tmp1

//x1=T3⊕T2;

35: MOVW reg4, tmp4
36: EOR reg4, tmp2
37: EOR reg5, tmp3

//x3=T5;

38: MOVW reg0, tmp7
```

3.2. Optimization of PRESENT–CTR

For high-end IoT devices, such as 32-bit ARM-based processors, the size of the counter is fixed at 32-bit [20,25]. However, in an 8-bit ATmega processor, the memory size is limited to at least 2KB depending on the ATmega model (e.g., ATtiny). For this reason, block cipher encryption is usually performed by 2^{16} times [26]. From the security perspective of CTR mode, the attacker can pre-compute and collect ciphertext information relied on the IV. When the initial CTR mode is operated, the counter of IV (Initial Vector) is initialized to zero. If there is an unpredictable n-bit input in the encryption process other than the master key, the effective key size for Time-Memory Trade Off (TMTO) attack and Key Collision (KC) attacks increases by n-bit [27]. For an 8-bit AVR microcontroller with a small memory footprint, it is suitable to use a 16-bit counter. For general cases, a 32-bit counter is also widely used in practice. In this section, we present both PRESENT-CTR mode implementations with 16-bit and 32-bit counter modes of operation on the ATmega128 microcontroller.

PRESENT-CTR with a 16-bit counter is described in Figure 1. We represent the bit in square form. Since PRESENT block cipher performs 64-bit block-wise encryption, 64 squares are utilized (i.e., 64-bit data). The most left square and the most right square represent the first and last bit, respectively. Colored squares represent a counter part. The

remaining white squares represent nonce part. The computation is performed from top to bottom.

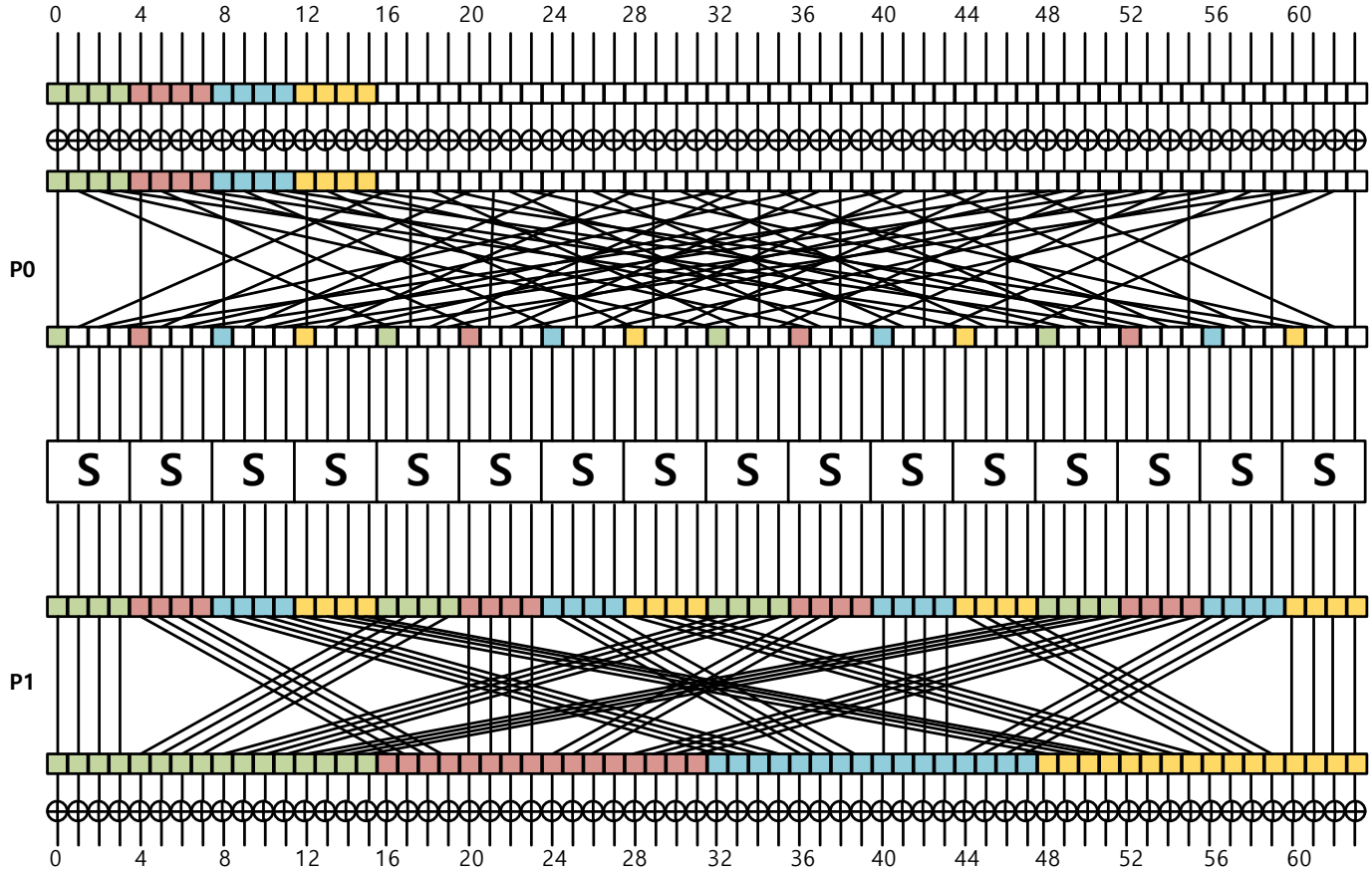

Figure 1. PRESENT-CTR with 16-bit counter.

1. First add-round-key. 64-bit plaintext is XORed with 64-bit round key. Since this is a bit-wise operation, each bits do not interfere with each other;
2. Permutation $P0$. The intermediate result is permuted. 16-bit counter values are distributed throughout the 64-bit intermediate result. Bits of the counter are arranged by 1 bit in the order of green, red, blue, and yellow according to a permutation rule;
3. Substitution. The 4-bit input values consist of 1-bit counter-part and 3-bit nonce part. The output of substitution can be pre-computed with the counter-part;
4. Permutation $P1$. The intermediate result is permuted again. After the permutation, the intermediate result is aligned by 16-bit wise;
5. Second add-round-key. The intermediate result is XORed with a second 64-bit round key.

The 4-bit data for each color of the initial 16-bit counter is distributed to the 16-bit data through permutation $P0$ and the bitslicing-substitution process. After the permutation $P1$ process is done, 16-bit data for each color is gathered regularly in the color (green, red, blue, and yellow) order of the initial counter. Through this, it is possible to predict 16-bit data through 4-bit of the initial counter. During the encryption process up to permutation $P1$, there is no interference between each color. For four 4-bit counter data, four 16-bit data can be pre-computed, independently. The required look-up table size is 128 bytes (4 colors $\times$ 2^4 counters $\times$ 16-bit size of data). A detailed description of look-up table generation is given in Algorithm 4. It generates 16 16-bit data with a counter divided into 4-bit data and repeats this process 4 times. The cost of generating a look-up table is less than performing PRESENT-ECB encryption by 4 times. We computed the pre-computation

table in a parallel way, which generates four look-up tables at once. Four index parts (1∼4-th bits, 5∼8-th bits, 9∼12-th bits, and 13∼16-th bits) generate four pre-computed outputs (1∼16-th bits, 17∼32-th bits, 33∼48-th bits, and 49∼64-th bits). This ensures the generation of pre-computation is independent of each other. The computation of a look-up table on AVR requires only 4022 clock cycles. This is roughly one time of PRESENT-ECB encryption. The look-up table can be stored in RAM or ROM. If we allocate the look-up table to RAM, we can access to the data with the `LD` instruction in 2 clock cycles. Otherwise, we can store it to ROM and access to the data with the `LPM` instruction in 3 clock cycles. The encryption process of PRESENT-CTR mode can be optimized away from the operation up-to the second add-round-key operation by using the created look-up table. Overall, this approach replaces the two permutation layers, two add-round-key, and one substitution layer to one look-up table accesses.

Algorithm 4: Generation of look-up tables for proposed PRESENT-CTR16 encryption.

Input: 64-bit block of Initial Vector (16-bit counter and 48-bit nonce) B, roundkeys $(roundkey_1, roundkey_2)$.

Output: Look-up tables for 16-bit counter ($LUT16_0$, $LUT16_1$, $LUT16_2$, $LUT16_3$).

1: $CTR \leftarrow 0$

2: $MASK \leftarrow$ 0xFFFFFFF0

3: **for** $i = 0$ **to** 3 **do**

4: $\quad C \leftarrow (B\&(MASK \lll 4i))|(CTR \ll 4i)$

5: $\quad$ **for** $j = 0$ **to** 15 **do**

6: $\quad\quad C \leftarrow C \oplus roundkey_1$

7: $\quad\quad C \leftarrow P_0(C)$

8: $\quad\quad C \leftarrow S_{Bitslicing}(C)$

9: $\quad\quad C \leftarrow P_1(C)$

10: $\quad\quad C \leftarrow C \oplus P(roundkey_2)$

11: $\quad\quad LUT16_i(j) \leftarrow C$

12: $\quad$ **end for**

13: **end for**

14: **return** $LUT16_0, LUT16_1, LUT16_2, LUT16_3$

Algorithm 5 shows the proposed PRESENT-CTR16 implementation using a 16-bit counter. In steps 2–5, look-up table access with 16-bit counter is performed. Afterward, the remaining PRESENT computations are performed. Listing 1 shows the AVR assembly code for the 16-bit data look-up. In order to improve performance, 16-bit LUT is performed with two 8-bit memory accesses. The memory access for 16-bit data is 9 clock cycles. This process is repeated 4 times. PRESENT encryption is optimized at the cost of just 36 clock cycles.

Listing 1. Look up table access for 16-bit counter.

```
.macro LUT16 LUT0, LUT1, OFFSET, T0, T1
    LDI R31, hi8(LUT0)
    MOV R30, OFFSET
    LPM T0, Z
    LDI R31, hi8(LUT1)
    LPM T1, Z
.endm
```

Algorithm 5: Proposed PRESENT-CTR16 encryption.

Input: 64-bit plaintext B, a key K.

Output: 64-bit ciphertext C.

$roundkey = (roundkey_1, roundkey_2, ..., roundkey_{32}) \leftarrow keySchedule(K)$

$C_{0\sim15} \leftarrow LUT16_0(B_{0\sim3})$

$C_{16\sim31} \leftarrow LUT16_1(B_{4\sim7})$

$C_{32\sim47} \leftarrow LUT16_2(B_{8\sim11})$

$C_{48\sim63} \leftarrow LUT16_3(B_{12\sim15})$

$C \leftarrow S_{Bitslicing}(C)$

for $i = 2$ **to** 15 **do**

  $C \leftarrow C \oplus roundkey_{2i-1}$

  $C \leftarrow P_0(C)$

  $C \leftarrow S_{Bitslicing}(C)$

  $C \leftarrow P_1(C)$

  $C \leftarrow C \oplus P(roundkey_{2i})$

  $C \leftarrow S_{Bitslicing}(C)$

end for

$C \leftarrow C \oplus roundkey_{31}$

$C \leftarrow P(C)$

$C \leftarrow S_{Bitslicing}(C)$

$C \leftarrow C \oplus roundkey_{32}$

return C

PRESENT-CTR with 32-bit counter is described in Figure 2. The 1-th to 16-th counters are indicated by a colored square. The 17-th to 32-th counters are indicated by symbol

squares. During the encryption process, the colored symbol square, which can be shown in Permutation $P1$, represents part of being affected by a color square and symbol square.

1. First add-round-key. Similarly to the 16-bit counter mode, the 64-bit plaintext is XORed with 64-bit round key. Since this is a bit-wise operation, bits do not interfere with each other;
2. Permutation $P0$. The intermediate result is permuted. 32-bit counter values are distributed throughout 64-bit intermediate results. The 16-bit to 32-bit of 32-bit counter are arranged one by one behind each color square;
3. Substitution. The 4-bit input values consist of a 2-bit counter part and 2-bit nonce part. The output of substitution can be pre-computed with the counter part;
4. Permutation $P1$. The intermediate result is permuted again. After the permutation, the intermediate result is aligned by 16-bit wise;
5. Second add-round-key. Similarly to the 16-bit counter mode of operation, the intermediate result is XORed with a second 64-bit round key.

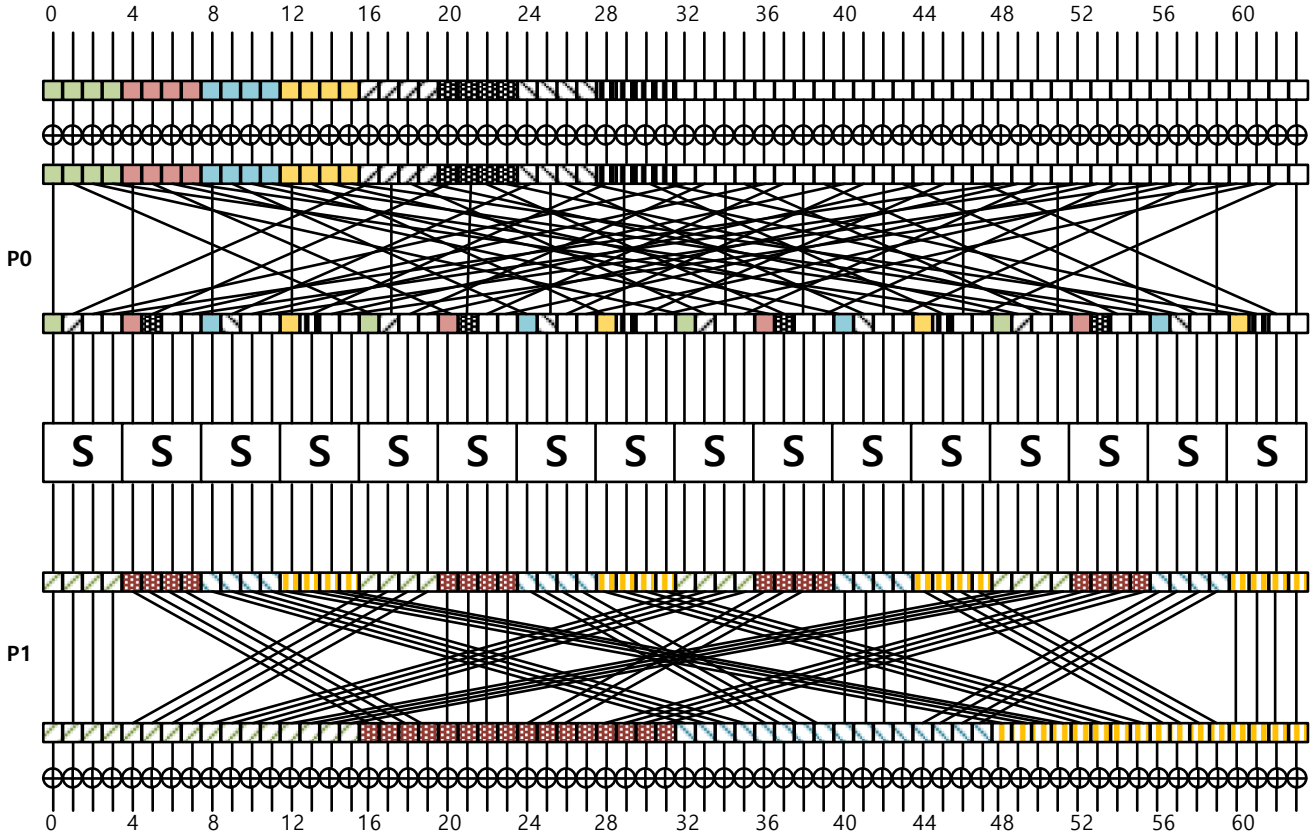

Figure 2. PRESENT-CTR with 32-bit counter.

The 8-bit data for each 4-bit color and 4-bit symbol parts of the initial 32-bit counter is distributed to the 16-bit data through permutation $P0$ and bitslicing-substitution process.

Unlike the 16-bit counter case, the counter-part represented by the colored square and the counter-part represented by the symbol square interfere with each other during the bitslicing-substitution process. This can be seen in detail in Figure 2. When permutation ($P1$) is completed, the 16-bit data mixed by color and symbol is gathered in the color and symbolic order of the initial counter. This allows the pre-computation of 16-bit data through the 8-bit (4-bit color and 4-bit symbol) of the initial counter. At this time, the required look-up table size is 2048 bytes (= 4 color and symbol $\times$ 2^8 counter $\times$ 16-bit size of data). Unlike the 16-bit PRESENT-CTR implementation, 32-bit PRESENT-CTR implementation requires a huge look-up table (i.e., 2048). We placed a look-up table in ROM instead of RAM. The manufacture of AVR provides secure memory-based architecture (i.e., CryptoMemory; https:

//www.microchip.com/design-centers/security-ics/mature-products/cryptomemory accessed on 7 January 2021). For real world implementation, we can utilize this technology. A detailed description of look-up table generation is given in Algorithm 6.

Algorithm 6: Generation of look-up tables for proposed PRESENT-CTR32 encryption.

Input: 64-bit block of Initial Vector (32-bit nonce and 32-bit counter) B, roundkeys $(roundkey_1, roundkey_2)$.

Output: Look-up tables for 32-bit counter ($LUT32_0$, $LUT32_1$, $LUT32_2$, $LUT32_3$).

1: $CTR \leftarrow 0$

2: $MASK \leftarrow$ 0xFFF0FFF0

3: **for** $i = 0$ **to** 3 **do**

4: $C \leftarrow (B\&(MASK \lll 4i))|(CTR_{0\sim3} \ll 4i)|(CTR_{4\sim7} \ll 4i + 16)$

5: **for** $j = 0$ **to** 256 **do**

6: $C \leftarrow C \oplus roundkey_1$

7: $C \leftarrow P_0(C)$

8: $C \leftarrow S_{Bitslicing}(C)$

9: $C \leftarrow P_1(C)$

10: $C \leftarrow C \oplus P(roundkey_2)$

11: $LUT32_i(j) \leftarrow C$

12: **end for**

13: **end for**

14: **return** $LUT32_0, LUT32_1, LUT32_2, LUT32_3$

Similarly to the 16-bit counter, the encryption process of PRESENT-CTR mode can be optimized from the operation up-to the second add-round-key by using the created look-up table. Overall, this approach replaces the two permutation layers, two add-round-key, and one substitution layer to one look-up table accesses.

Algorithm 7 shows the proposed PRESENT-CTR32 implementation using a 32-bit counter. In Steps 2∼5, 16-bit data look-up with 8-bit (4-bit color and 4-bit symbol) counter is performed. Listing 2 shows the AVR assembly code for the 32-bit data look-up. The cost of looking-up 16-bit data is 10 clock cycles. This process is repeated 4 times. This is optimized at the cost of just 40 clock cycles.

Algorithm 7: Proposed PRESENT-CTR32 encryption.

Input: 64-bit plaintext B, a key K.

Output: 64-bit ciphertext C.

1: $roundkey = (roundkey_1, roundkey_2, ..., roundkey_{32}) \leftarrow keySchedule(K)$

2: $C_{0\sim15} \leftarrow LUT32_0(B_{0\sim3} \| B_{16\sim19})$

3: $C_{16\sim31} \leftarrow LUT32_1(B_{4\sim7} \| B_{20\sim23})$

4: $C_{32\sim47} \leftarrow LUT32_2(B_{8\sim11} \| B_{24\sim27})$

5: $C_{48\sim63} \leftarrow LUT32_3(B_{12\sim15} \| B_{28\sim31})$

6: $C \leftarrow S_{Bitslicing}(C)$

7: **for** $i = 2$ **to** 15 **do**

8: $\quad C \leftarrow C \oplus roundkey_{2i-1}$

9: $\quad C \leftarrow P_0(C)$

10: $\quad C \leftarrow S_{Bitslicing}(C)$

11: $\quad C \leftarrow P_1(C)$

12: $\quad C \leftarrow C \oplus P(roundkey_{2i})$

13: $\quad C \leftarrow S_{Bitslicing}(C)$

14: **end for**

15: $C \leftarrow C \oplus roundkey_{31}$

16: $C \leftarrow P(C)$

17: $C \leftarrow S_{Bitslicing}(C)$

18: $C \leftarrow C \oplus roundkey_{32}$

19: **return** C

Listing 2. Look up table access for 32-bit counter.

```
.macro LUT32 LUT0, LUT1, OFFSET1, OFFSET2, T0, T1
    LDI R31, hi8(LUT0)
    MOV R30, OFFSET1
    ADD R30, OFFSET2
    LPM T0, Z
    LDI R31, hi8(LUT1)
    LPM T1, Z
.endm
```

4. Evaluation

In CHES'17, bitslicing-based PRESENT implementation was proposed [24]. It has been theoretically and practically proven that the bitslicing technique shows the best results in 32-bit or higher processors. However, bitslicing-implementation in an 8-bit AVR environment has not been explored before. In embedded devices, bitslicing optimizes the memory access for the substitution layer but it requires Boolean operations. The AVR microcontroller has 8-bit wise 32 general-purpose registers and it should be carefully optimized to achieve high performance in bitslicing implementation. We evaluated PRESENT-ECB and PRESENT-CTR implementations and compared them with former works. ATmega128 is selected as a microcontroller, which is one of the most popular AVR microcontrollers in wireless sensor networks. In the case of CTR mode, 16-bit counter and 32-bit counter versions are evaluated. The software was evaluated with `Atmel Studio 7` and `-Os` option. Benchmarks are checked in clock cycles per byte which occurs when each mode of operation is called once.

Table 1 describe the comparison between this work and former implementations. PRESENT-ECB encryption by Dinu et al. (80-bit) and Engel et al. (128-bit) required 930.8 and 1349.0 clock Cycles Per Byte (CPB), respectively [1,28]. On the other hand, the proposed PRESENT-ECB implementation uses almost the same RAM as the existing implementation, but only requires 504.2 clock cycles per byte. For the code size, the proposed implementation utilized two permutation operations ($P0$, $P1$). The code size is bigger than former works. Since the proposed PRESENT-CTR implementation is optimized further by utilizing pre-computation, the proposed PRESENT-CTR mode achieved a higher performance than the existing PRESENT-ECB mode. The code size of the CTR mode of operation is bigger than the ECB mode of operation, but it achieved 488.2, 488.7, and 491.6 CPB, for 16-bit counter (RAM), 16-bit counter (ROM), and 32-bit counter (ROM). In Table 2, the comparison of execution timing depending on the message size is given. The RAM based 16-bit counter mode of operation requires look-up table generation online. For this reason, performance is lower than the ROM-based 16-bit counter mode of operation. However, the RAM-based implementation outperforms when the length is over 8192 bytes. PRESENT implementations are publicly available at: https://github.com/solowal/PRESENT_AVR (accessed on 7 January 2021), where anyone can access PRESENT implementations.

Table 1. Comparison of PRESENT on target embedded processors (Alf and Vegard's RISC (AVR)) in terms of timing (cycles per byte), RAM (bytes), and code size (bytes), [1]: Pre-computation in RAM, [2]: Pre-computation in ROM, †: 16-bit counter, ‡: 32-bit counter. ECB: Electronic Code Book.

Method	Security Level	Mode of Operation	Code Size	RAM	Timing
[28]	80	ECB	760	281	930.8
[1]	128	ECB	660	280	1349.0
This work			956	282	504.2
		CTR †,[1]	1150	420	488.2
		CTR †,[2]	1152	292	488.7
		CTR ‡,[2]	3072	292	491.6

Table 2. Comparison of PRESENT on target embedded processors (AVR) in terms of timing (10^6 clock cycles) depending on message size (bytes), [1]: Pre-computation in RAM, [2]: Pre-computation in ROM, †: 16-bit counter, ‡: 32-bit counter.

Method	Message Size (bytes)				
	4096	8192	16,384	32,768	65,536
CTR †,1	2.0038	4.0037	8.0035	16.0030	32.0019
CTR †,2	2.0010	4.0038	8.0076	16.0153	32.0307
CTR ‡,2	2.0136	4.0273	8.0547	16.1095	32.2191

In Table 3, a comparison with other lightweight block cipher implementations on target-embedded processors is given. On the 8-bit AVR environment, previous PRESENT implementation using 128-bit key shows a lower performance than other lightweight cryptographic algorithms [28], since substitution and permutation layers of the PRESENT algorithm incurs considerable overheads in an 8-bit AVR environment. We achieved the execution timing improvement of target block cipher implementation to 504 clock cycles per byte in an 8-bit AVR environment. Therefore, we believe that our optimization results are not only actually usable from real 8-bit AVR microcontrollers but can be applied to various cryptographic application algorithms.

Table 3. Comparison of other implementations on target embedded processors (AVR) in terms of timing (cycles per byte), RAM (bytes), and code size (bytes).

Algorithm	Plaintext	Security Level	Code Size	RAM	Timing
PIPO [29]	64	128	320	31	197
SIMON [17]			290	24	253
RECTANGLE [28]			466	204	403
RoadRunneR [30]			196	24	477
PRESENT [this work]			956	282	504
SKINNY [28]			502	187	877
PRIDE [28]			650	47	969
PRESENT [1]			660	280	1349
CRAFT [31]			894	243	1,504

5. Conclusions

We presented compact ECB and CTR for PRESENT on embedded processors. The ECB mode of operation was efficiently implemented in an optimization of diffusion layer, substitute layer, and add-round-key operations. The operation was accelerated with pre-computation in CTR. This new approach optimized away PRESENT operations by the substitution layer of second round. Finally, PRESENT block cipher on target microcontrollers consumed 504.2, 488.2, 488.7, and 491.6 CPB for ECB, 16-bit CTR (RAM-based implementation), 16-bit CTR (ROM-based implementation), and 32-bit CTR (ROM-based implementation) modes of operation, respectively.

Author Contributions: Investigation, H.K. and Y.B.K.; Software, H.K., Y.B.K., S.C.S., and H.S.; Writing-original draft, H.K. and Y.B.K.; Writing-review and editing, H.K., Y.B.K., S.C.S., and H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research of Hyeokdong Kwon and Hwajeong Seo was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2020R1F1A1048478) and this research of Hyeokdong Kwon and Hwajeong Seo was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services). This research of YoungBeom Kim and Seog Chung Seo was funded by National Research Foundation of Korea: 2019R1F1A1058494.

Conflicts of Interest: The authors declare no conflicct of interest.

References

1. Engels, S.; Kavun, E.B.; Paar, C.; Yalçin, T.; Mihajloska, H. A non-linear/linear instruction set extension for lightweight ciphers. In Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic, Austin, TX, USA, 7–10 April 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 67–75.
2. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.; Seurin, Y.; Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Vienna, Austria, 10–13 September 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466.
3. Hong, D.; Lee, J.K.; Kim, D.C.; Kwon, D.; Ryu, K.H.; Lee, D.G. LEA: A 128-bit block cipher for fast encryption on common processors. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 19–21 August 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–27.
4. Seo, H.; Liu, Z.; Choi, J.; Park, T.; Kim, H. Compact implementations of LEA block cipher for low-end microprocessors. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 20–22 August 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 28–40.
5. Seo, H.; Jeong, I.; Lee, J.; Kim, W.H. Compact implementations of ARX-based block ciphers on IoT processors. *ACM Trans. Embed. Comput. Syst. (TECS)* **2018**, *17*, 1–16. [CrossRef]
6. Seo, H.; An, K.; Kwon, H. Compact LEA and HIGHT implementations on 8-bit AVR and 16-bit MSP processors. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 23–25 August 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 253–265.
7. Kim, Y.; Kwon, H.; An, S.; Seo, H.; Seo, S.C. Efficient Implementation of ARX-Based Block Ciphers on 8-Bit AVR Microcontrollers. *Mathematics* **2020**, *8*, 1837. [CrossRef]
8. Hong, D.; Sung, J.; Hong, S.; Lim, J.; Lee, S.; Koo, B.S.; Lee, C.; Chang, D.; Lee, J.; Jeong, K.; et al. HIGHT: A new block cipher suitable for low-resource device. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Yokohama, Japan, 10–13 October 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 46–59.
9. Eisenbarth, T.; Gong, Z.; Güneysu, T.; Heyse, S.; Indesteege, S.; Kerckhof, S.; Koeune, F.; Nad, T.; Plos, T.; Regazzoni, F.; et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In Proceedings of the International Conference on Cryptology in Africa, Ifrance, Morocco, 10–12 July 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 172–187.
10. Kim, B.; Cho, J.; Choi, B.; Park, J.; Seo, H. Compact Implementations of HIGHT Block Cipher on IoT Platforms. *Secur. Commun. Netw.* **2019**, *2019*, 5323578. [CrossRef]
11. Koo, B.; Roh, D.; Kim, H.; Jung, Y.; Lee, D.G.; Kwon, D. CHAM: A family of lightweight block ciphers for resource-constrained devices. In Proceedings of the International Conference on Information Security and Cryptology, Xi'an, China, 3–5 November 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 3–25.
12. Seo, H. Memory-efficient implementation of ultra-lightweight block cipher algorithm CHAM on low-end 8-bit AVR processors. *J. Korea Inst. Inf. Secur. Cryptol.* **2018**, *28*, 545–550.
13. Roh, D.; Koo, B.; Jung, Y.; Jeong, I.W.; Lee, D.G.; Kwon, D.; Kim, W.H. Revised Version of Block Cipher CHAM. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Korea, 4–6 December 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 1–19.
14. Kwon, H.; Kim, H.; Choi, S.J.; Jang, K.; Park, J.; Kim, H.; Seo, H. Compact Implementation of CHAM Block Cipher on Low-End Microcontrollers. In Proceedings of the International Conference on Information Security Applications, Jeju Island, Korea, 26–28 August 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 127–141.
15. Kwon, H.; An, S.; Kim, Y.; Kim, H.; Choi, S.J.; Jang, K.; Park, J.; Kim, H.; Seo, S.C.; Seo, H. Designing a CHAM Block Cipher on Low-End Microcontrollers for Internet of Things. *Electronics* **2020**, *9*, 1548. [CrossRef]
16. Beaulieu, R.; Shors, D.; Smith, J.; Treatman-Clark, S.; Weeks, B.; Wingers, L. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptol. EPrint Arch.* **2013**, *2013*, 404–449.
17. Beaulieu, R.; Shors, D.; Smith, J.; Treatman-Clark, S.; Weeks, B.; Wingers, L. The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers. In Proceedings of the International Workshop on Lightweight Cryptography for Security and Privacy, Istanbul, Turkey, 1–2 September 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 3–20.
18. Osvik, D.A.; Bos, J.W.; Stefan, D.; Canright, D. Fast software AES encryption. In Proceedings of the International Workshop on Fast Software Encryption, Seoul, Korea, 7–10 February 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 75–93.
19. McGrew, D.; Viega, J. The Galois/counter mode of operation (GCM). *Submiss. NIST Modes Oper. Process* **2004**, *20*, 1–27.

20. Park, J.H.; Lee, D.H. FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, 469–499.10.13154/tches.v2018.i3.469-499. [CrossRef]
21. Kim, K.; Choi, S.; Kwon, H.; Liu, Z.; Seo, H. FACE–LIGHT: Fast AES–CTR Mode Encryption for Low-End Microcontrollers. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Korea, 4–6 December 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 102–114.
22. Kim, K.; Choi, S.; Kwon, H.; Kim, H.; Liu, Z.; Seo, H. PAGE–Practical AES-GCM Encryption for Low-End Microcontrollers. *Appl. Sci.* **2020**, *10*, 3131. [CrossRef]
23. Seo, H.; Kwon, H.; Kim, H.; Park, J. ACE: ARIA-CTR Encryption for Low-End Embedded Processors. *Sensors* **2020**, *20*, 3788. [CrossRef]
24. Reis, T.B.; Aranha, D.F.; López, J. PRESENT runs fast. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, Taipei, Taiwan, 25–28 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 644–664.
25. Seo, H.; Lee, G.; Park, T.; Kim, H. Compact GCM implementations on 32-bit ARMv7-A processors. In Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea, 18–20 October 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 704–707.
26. Kim, Y.; Seo, S.C. An Efficient Implementation of AES on 8-Bit AVR-Based Sensor Nodes. In Proceedings of the International Conference on Information Security Applications, Jeju Island, Korea, 26–28 August 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 276–290.
27. McGrew, D.A. Counter mode security: Analysis and recommendations. *Cisco Syst. Novemb.* **2002**, *2*, *1–8.*.
28. Dinu, D.; Biryukov, A.; Großschädl, J.; Khovratovich, D.; Le Corre, Y.; Perrin, L. FELICS–fair evaluation of lightweight cryptographic systems. In Proceedings of the NIST Workshop on Lightweight Cryptography, Gaithersburg, MD, USA, 20–21 July 2015; Volume 128.
29. Kim, H.; Jeon, Y.; Kim, G.; Kim, J.; Sim, B.Y.; Han, D.G.; Seo, H.; Kim, S.; Hong, S.; Sung, J.; et al. A New Method for Designing Lightweight S-Boxes with High Differential and Linear Branch Numbers, and Its Application*. In Proceedings of the 23rd Annual International Conference on Information Security and Cryptology (ICISC 2020), Seoul, Korea, 2–4 December 2020; pp. 105–132.
30. Baysal, A.; Şahin, S. RoadRunneR: A small and fast bitslice block cipher for low cost 8-bit processors. In *Lightweight Cryptography for Security and Privacy*; Springer: Cham, Switzerland, 2015; pp. 58–76.
31. Beierle, C.; Leander, G.; Moradi, A.; Rasoolzadeh, S. CRAFT: Lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol.* **2019**, *2019*, 5–45. [CrossRef]